Lecture 18:

Transactional Memory

Parallel Computing

Stanford CS149, Winter 2019

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
  - Lock-free data structures
  - 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 software implementation of transactional memory
- 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 <u>what</u> to do (maintain atomicity of this code), not <u>how</u> to do it
  - No explicit use or management of locks
- **System implements synchronization as necessary to ensure atomicity**
  - System <u>could</u> implement atomic { } using locks (see this later)
  - Implementation discussed today uses optimistic concurrency: maintain serialization only in situations of true contention (R-W or W-W conflicts)

Declarative vs. imperative abstractions

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

- Imperative: programmer states how 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. (the effectively all happen at the same time)

What is the consistency model for TM?

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 mutual locking specified by programmer

```
public Object get(Object key) {
  synchronized (myHashMap) {   // per-hashmap lock 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?

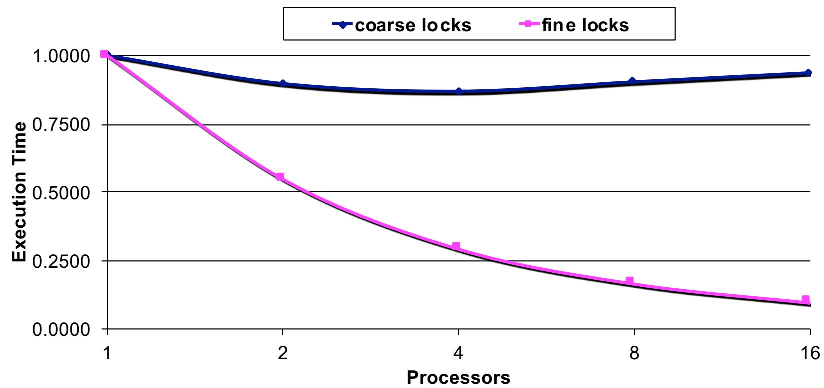
```
public Object get(Object key)  {
  int idx = hash(key);            // compute hash
  HashEntry e = buckets[idx];     // find bucket
  while (e != null)  {            // find element in bucket
    if (equals(key, e.key))
      return e.value;
    e = e.next;
  }
  return null;
}
```

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

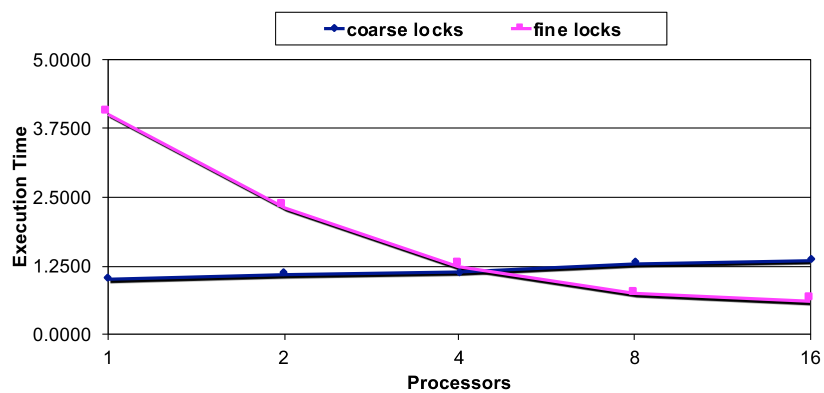
Review: performance of fine-grained locking

Reducing contention via fine-grained locking leads to better performance

Hash-Table



Balanced Tree



Transactional HashMap

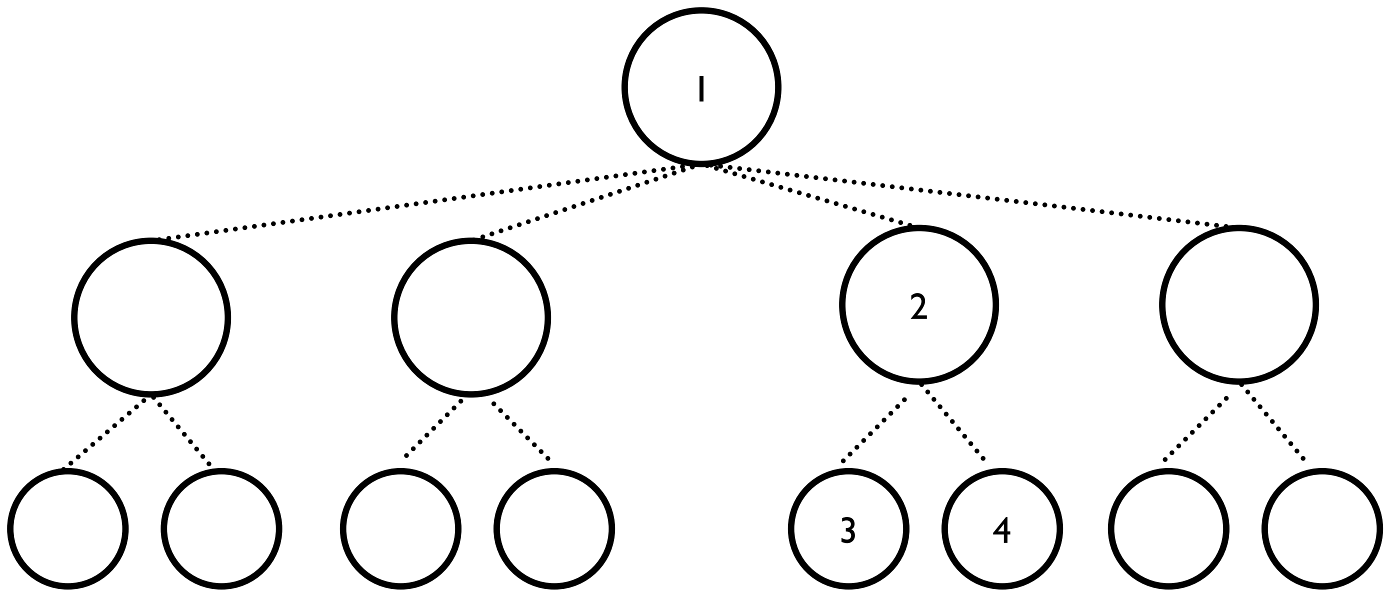
- Simply enclose all operation in atomic block
  - Semantics of atomic block: system ensures atomicity of logic within block

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

- 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

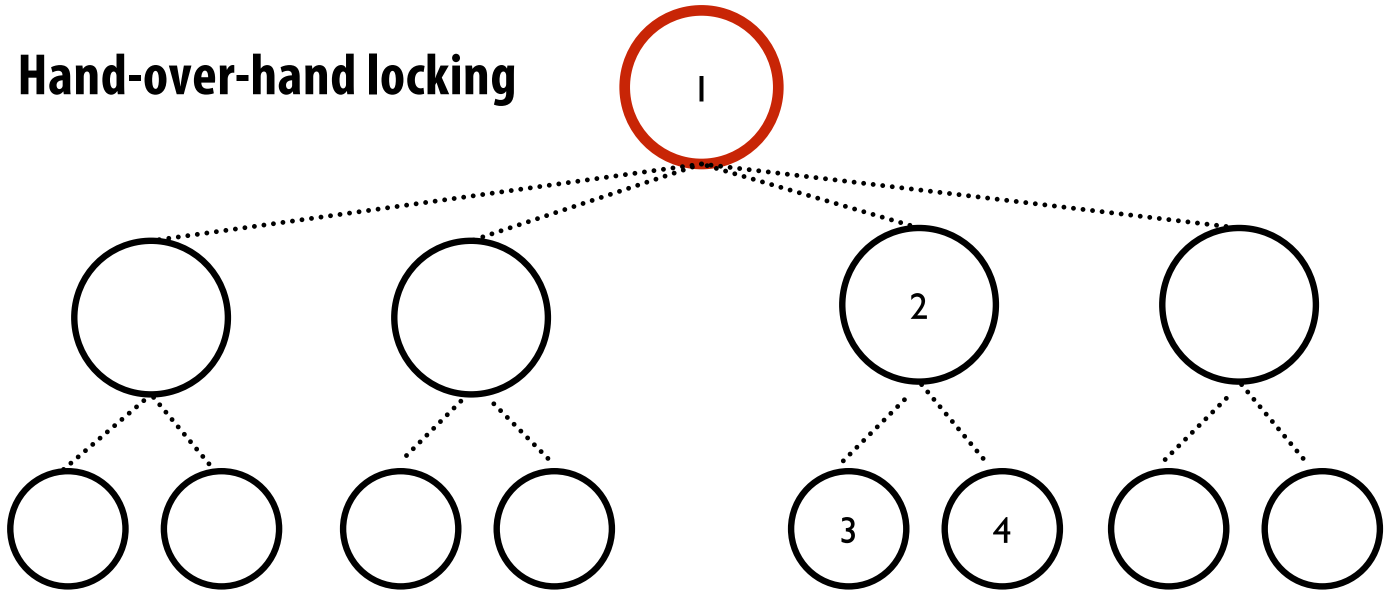
Goal: modify nodes 3 and 4 in a thread-safe way



Fine-grained locking example

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

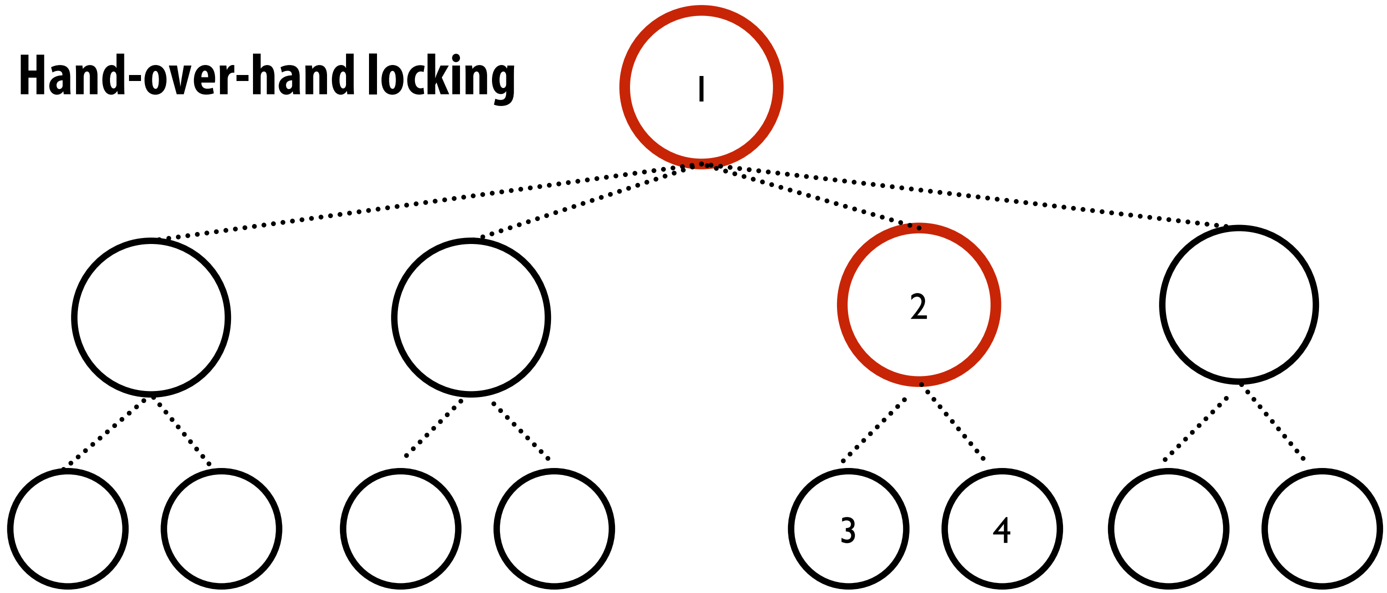
Hand-over-hand locking



Fine-grained locking example

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

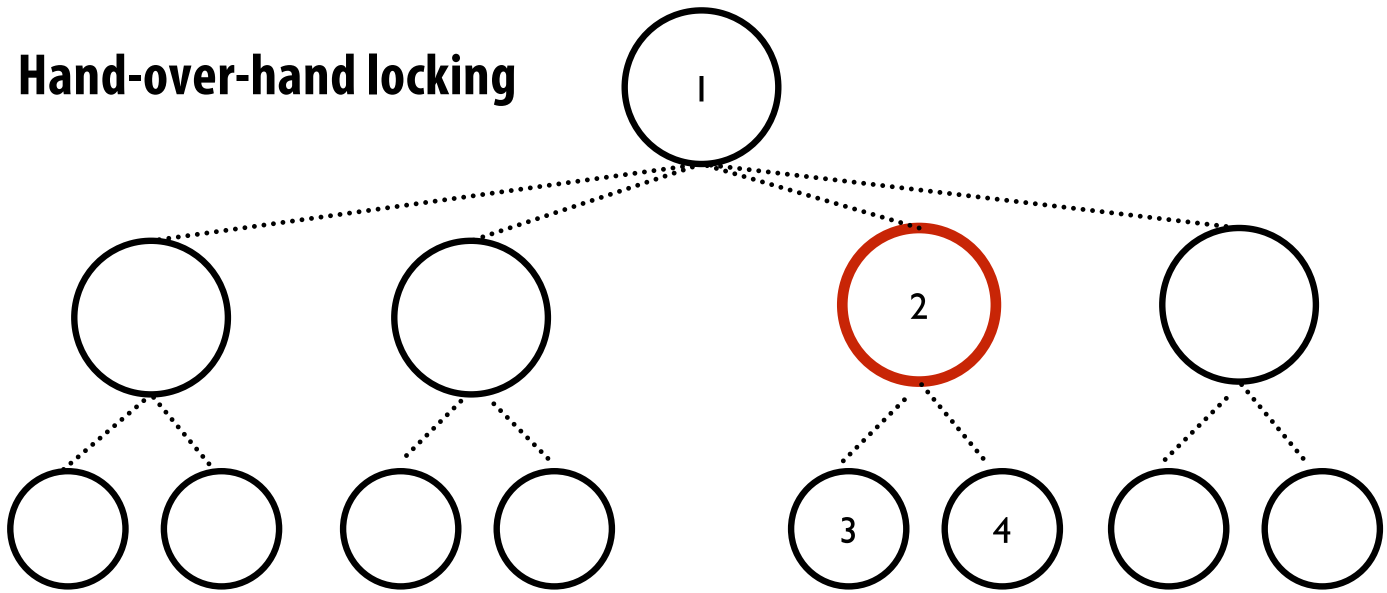
Hand-over-hand locking



Fine-grained locking example

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

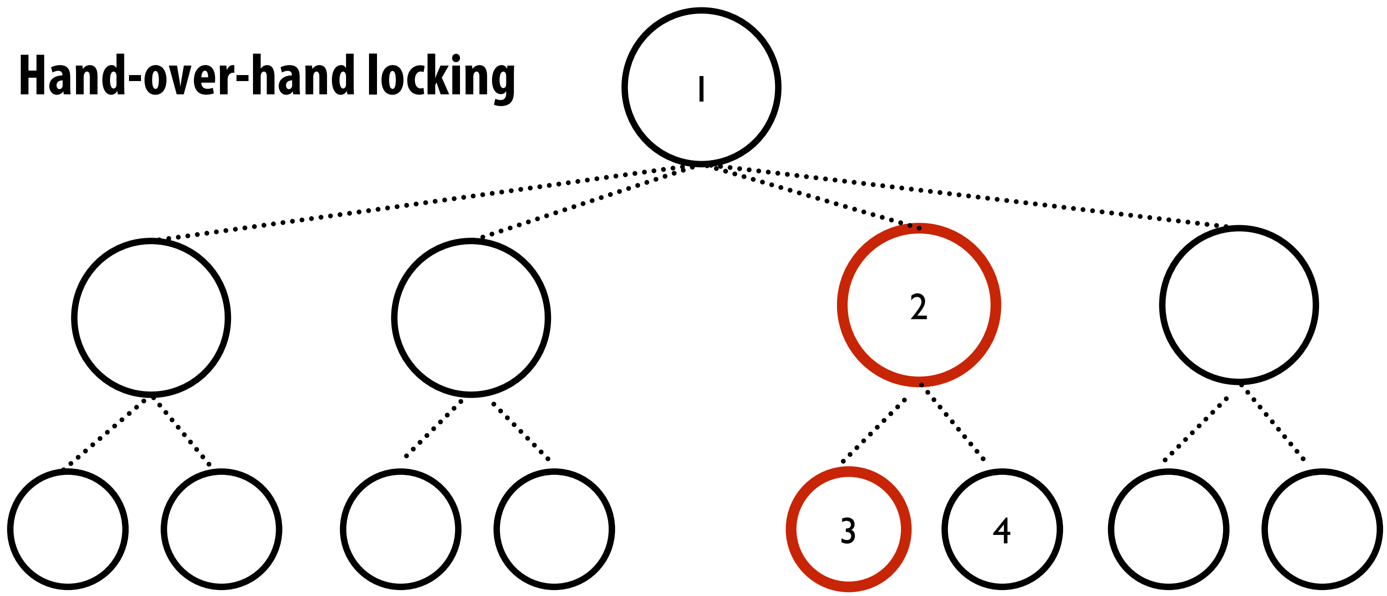
Hand-over-hand locking



Fine-grained locking example

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

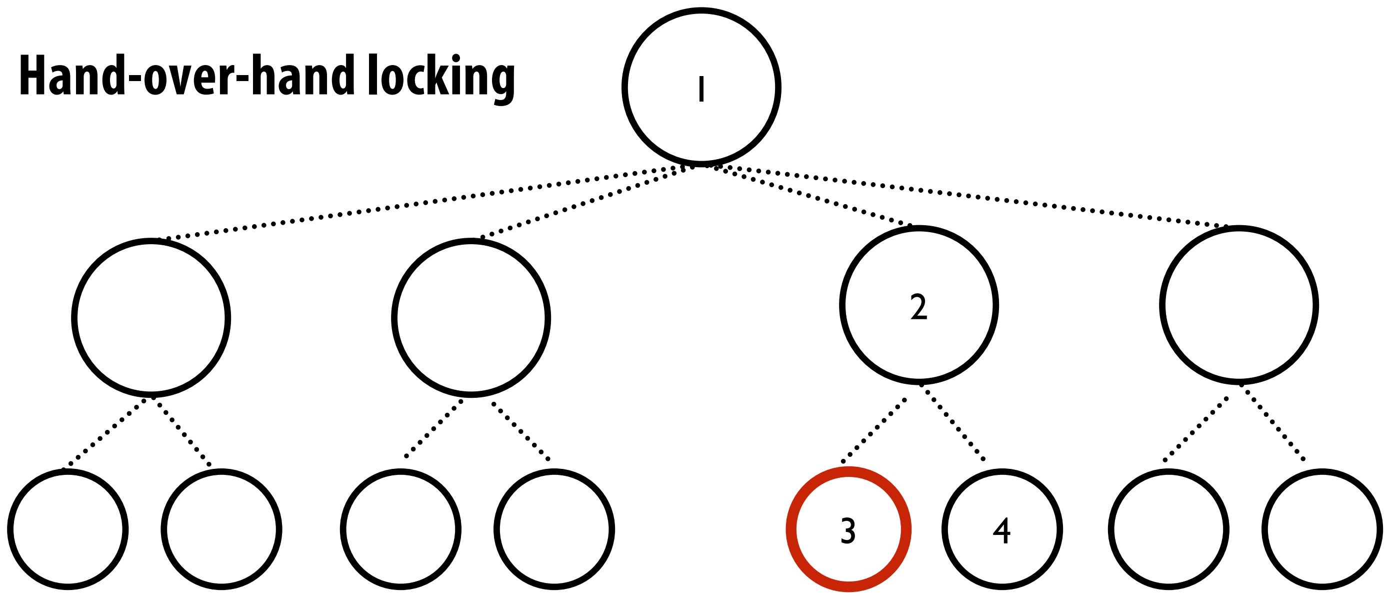
Hand-over-hand locking



Fine-grained locking example

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

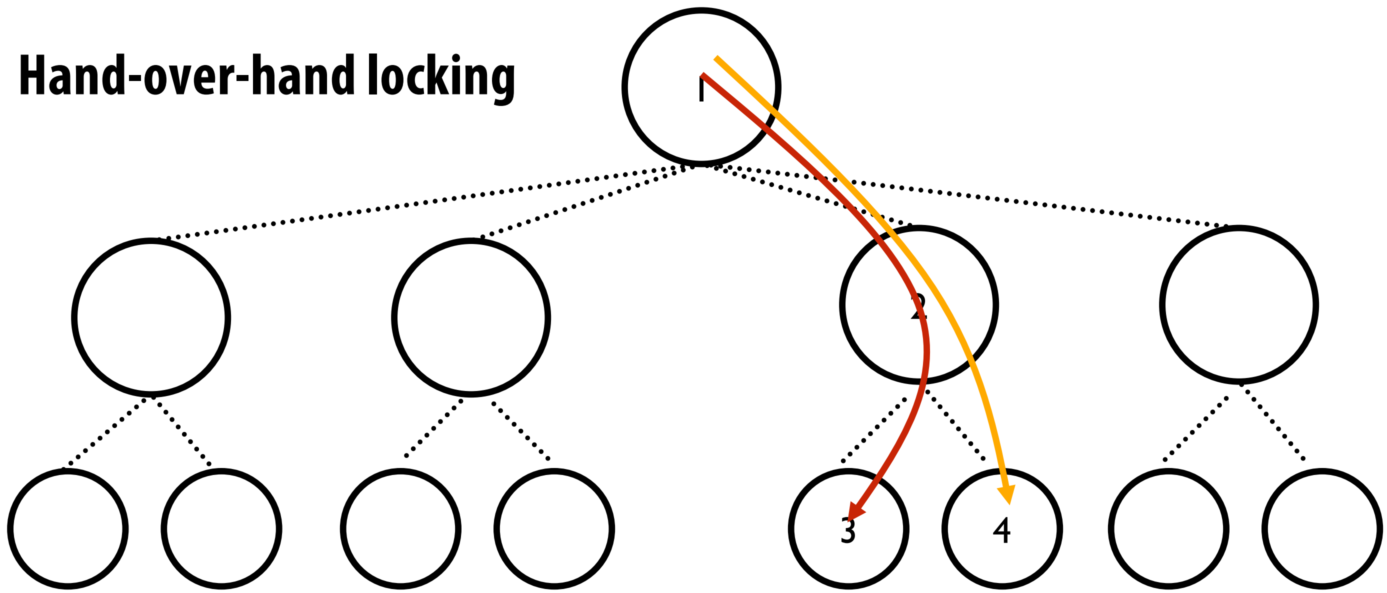
Hand-over-hand locking



Fine-grained locking example

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

Hand-over-hand locking

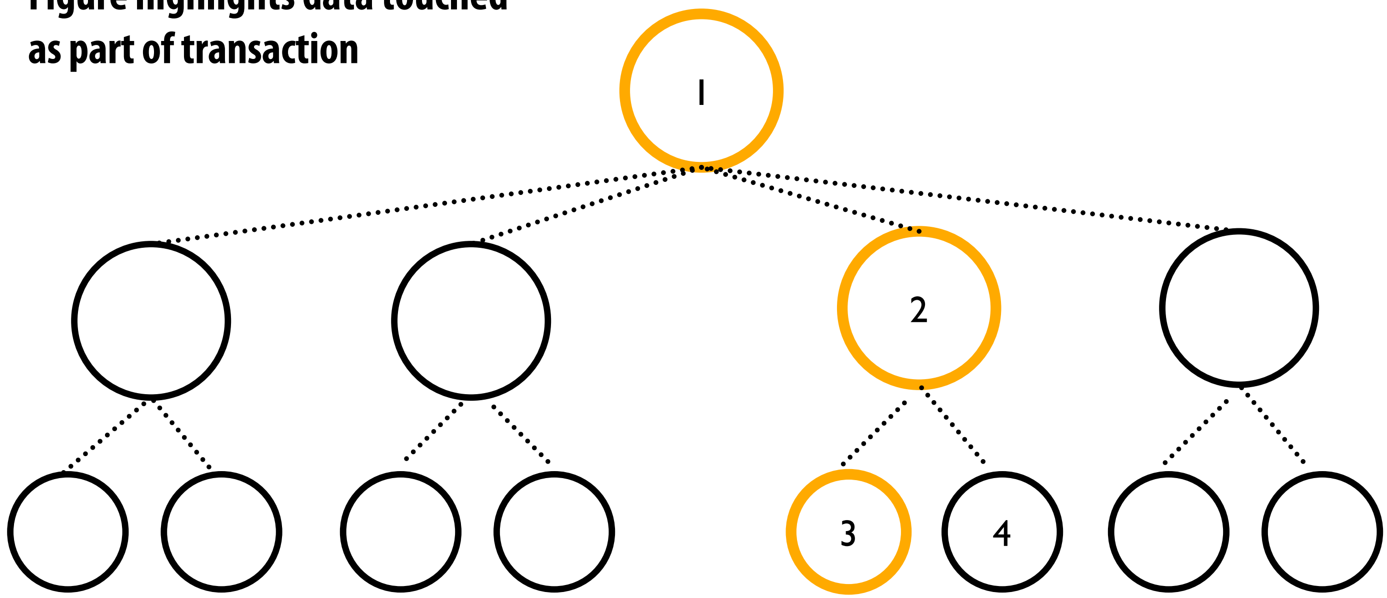


Locking can prevent concurrency

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

Transactions example

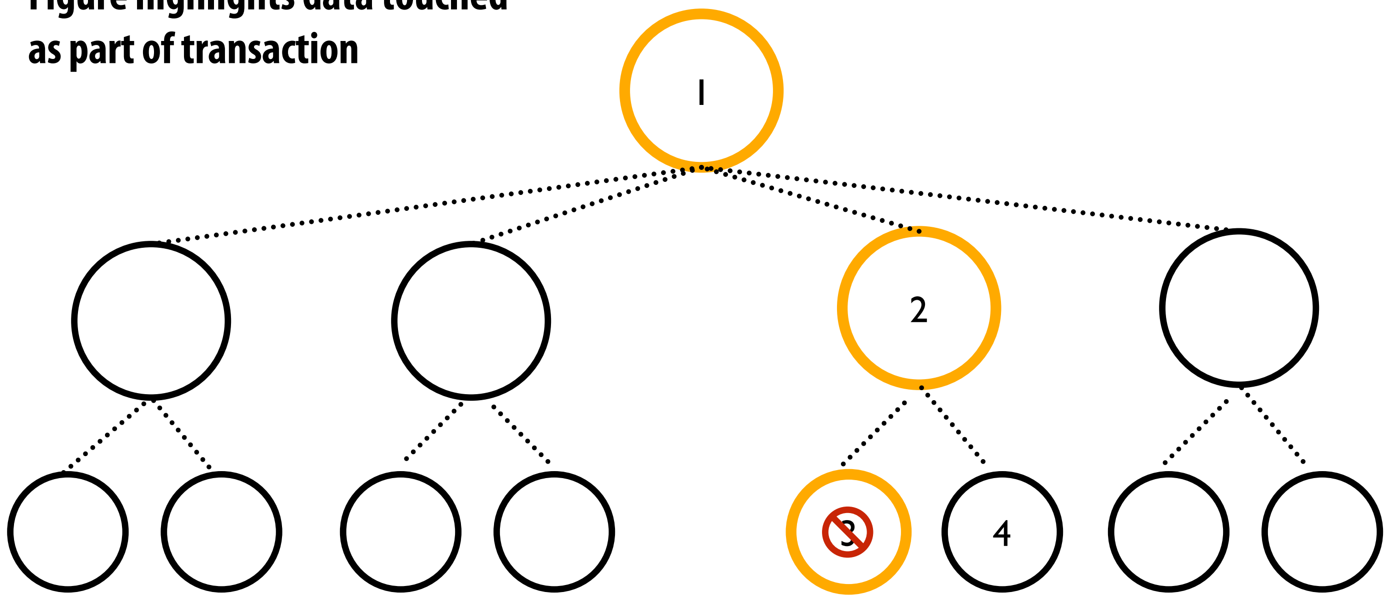
Figure highlights data touched as part of transaction



Transaction A
READ: 1, 2, 3

Transactions example

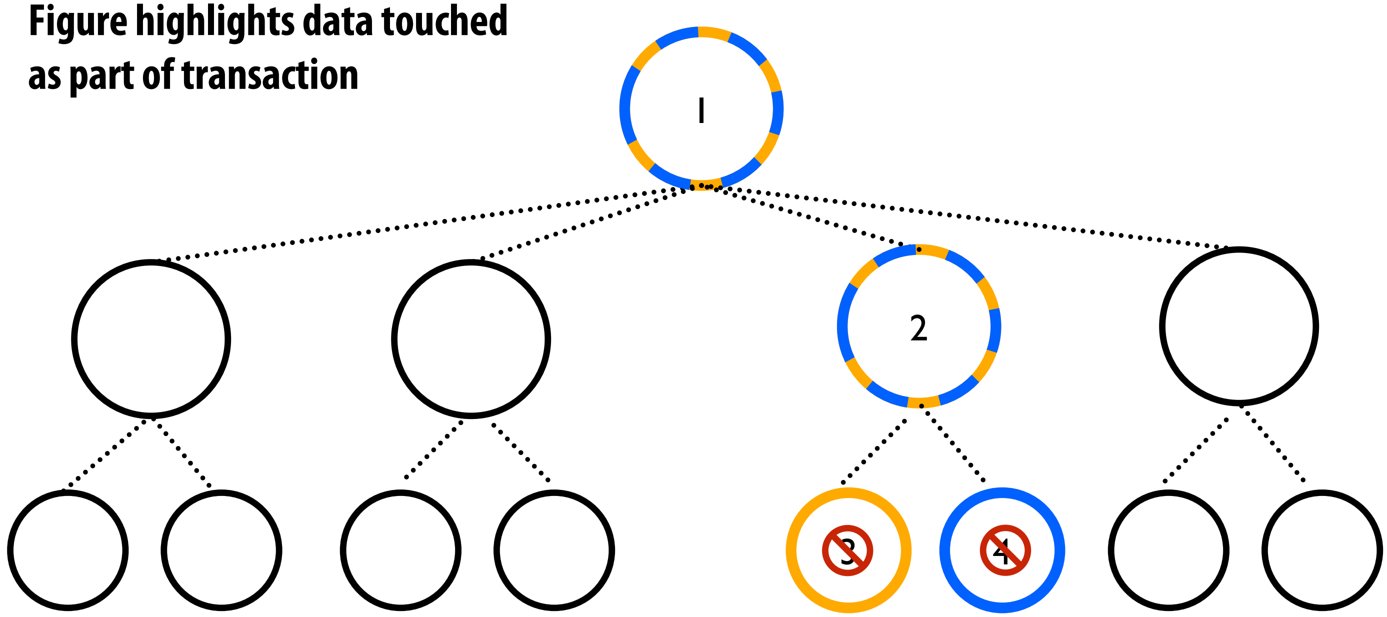
Figure highlights data touched as part of transaction



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

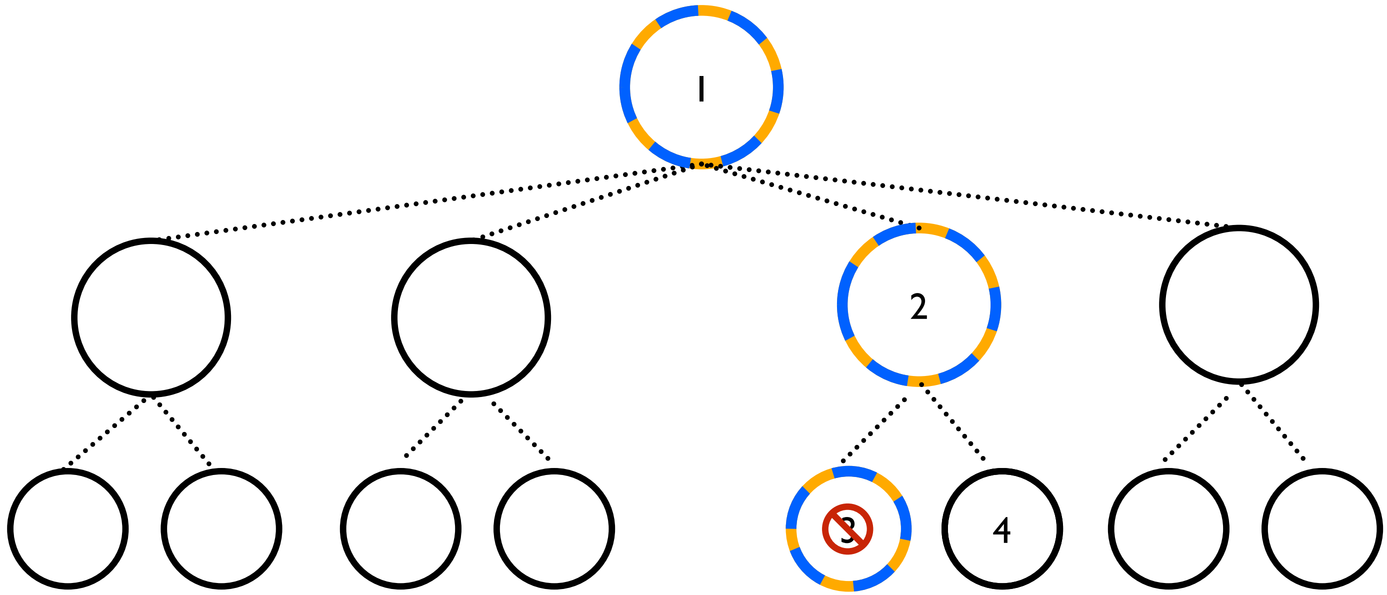
Transaction B
READ: 1, 2, 4
WRITE: 4

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

Slide credit: Austen McDonald

Transactions example #2

(Both transactions modify node 3)



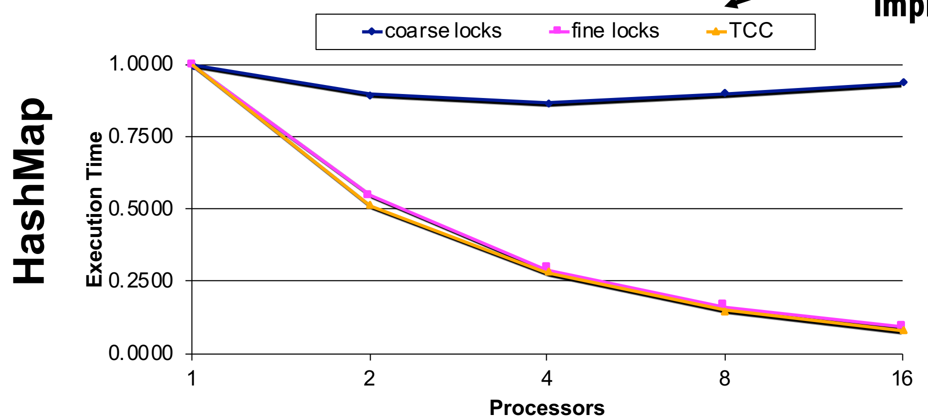
Transaction A
READ: 1, 2, 3
WRITE: 3

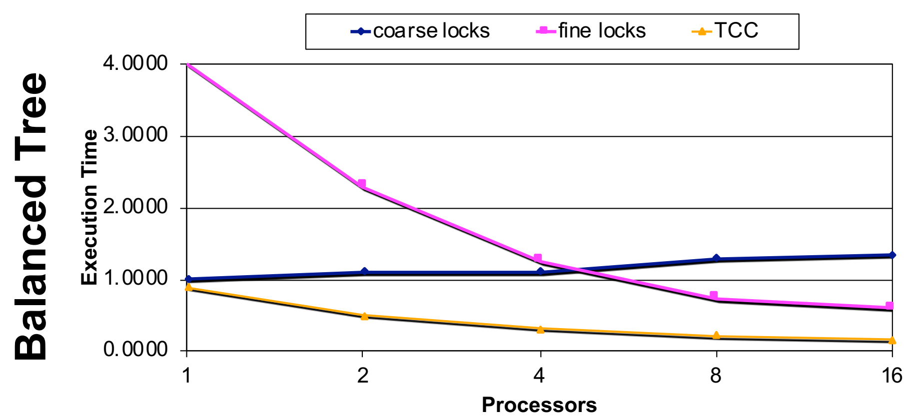
Transaction B
READ: 1, 2, 3
WRITE: 3

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

Performance: locks vs. transactions

"TCC" is a TM system implemented in hardware





Another motivation: failure atomicity

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

Another motivation: composability

```
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 a lock and a hard (to implement) place !**
  - Coarse-grain locks: low performance
  - Fine-grain locking: good for performance, but mistakes 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
  - System-wide policies can break software modularity

- **Programmer caught between an lock and a hard (to implement) place**
  - Coarse-grain locks: low performance
  - Fine-grain locking: good for performance, but mistakes 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 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

- 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]]++;
}
```

Self-check: atomic { } $\neq$ 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 should be 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
  - While maintaining concurrency as much as possible
- Two key implementation questions
  - Data versioning policy: How does the system manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions?
  - Conflict detection policy: how/when does the system determine that two concurrent transactions conflict?

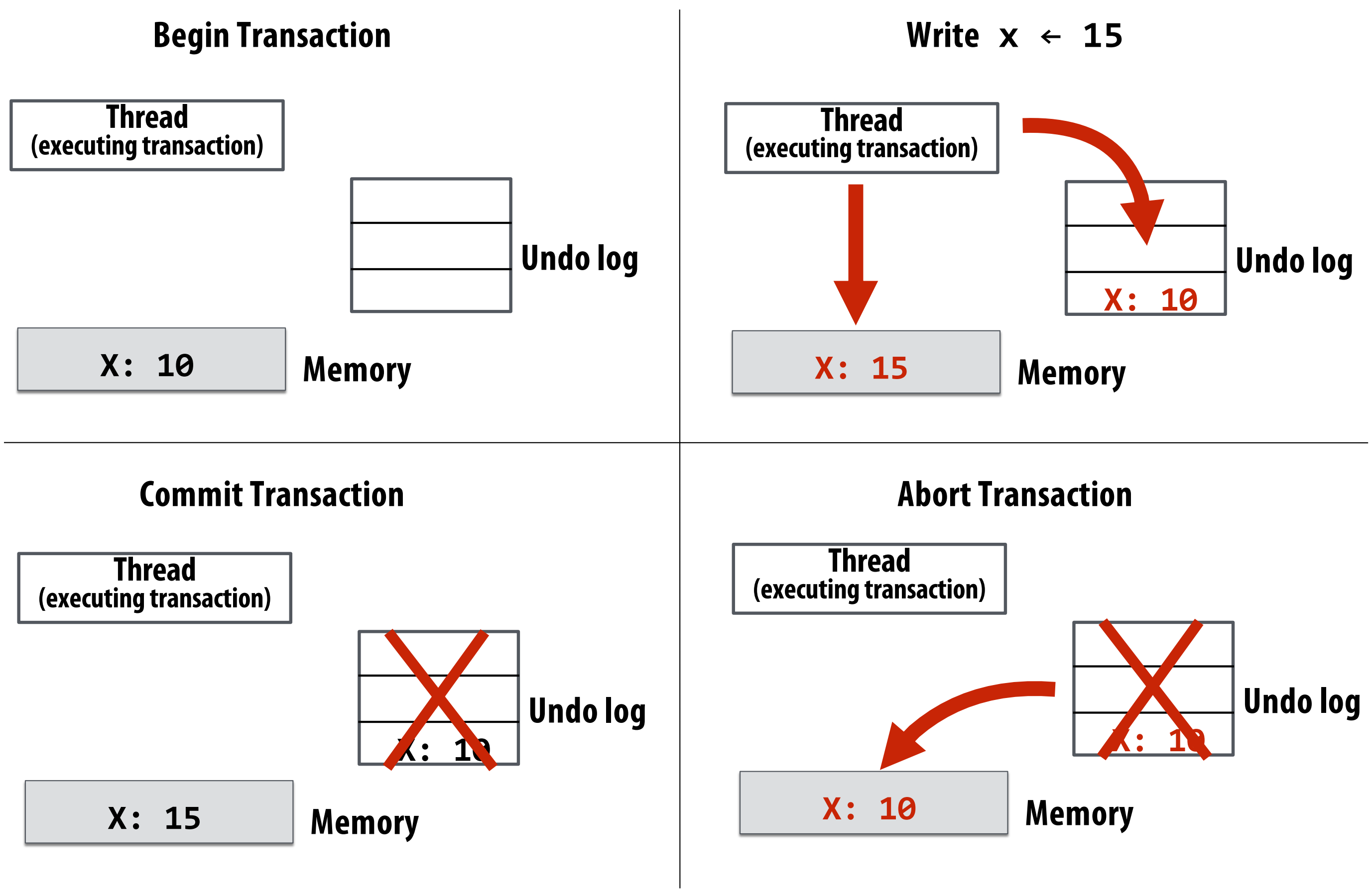
Data versioning policy

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)

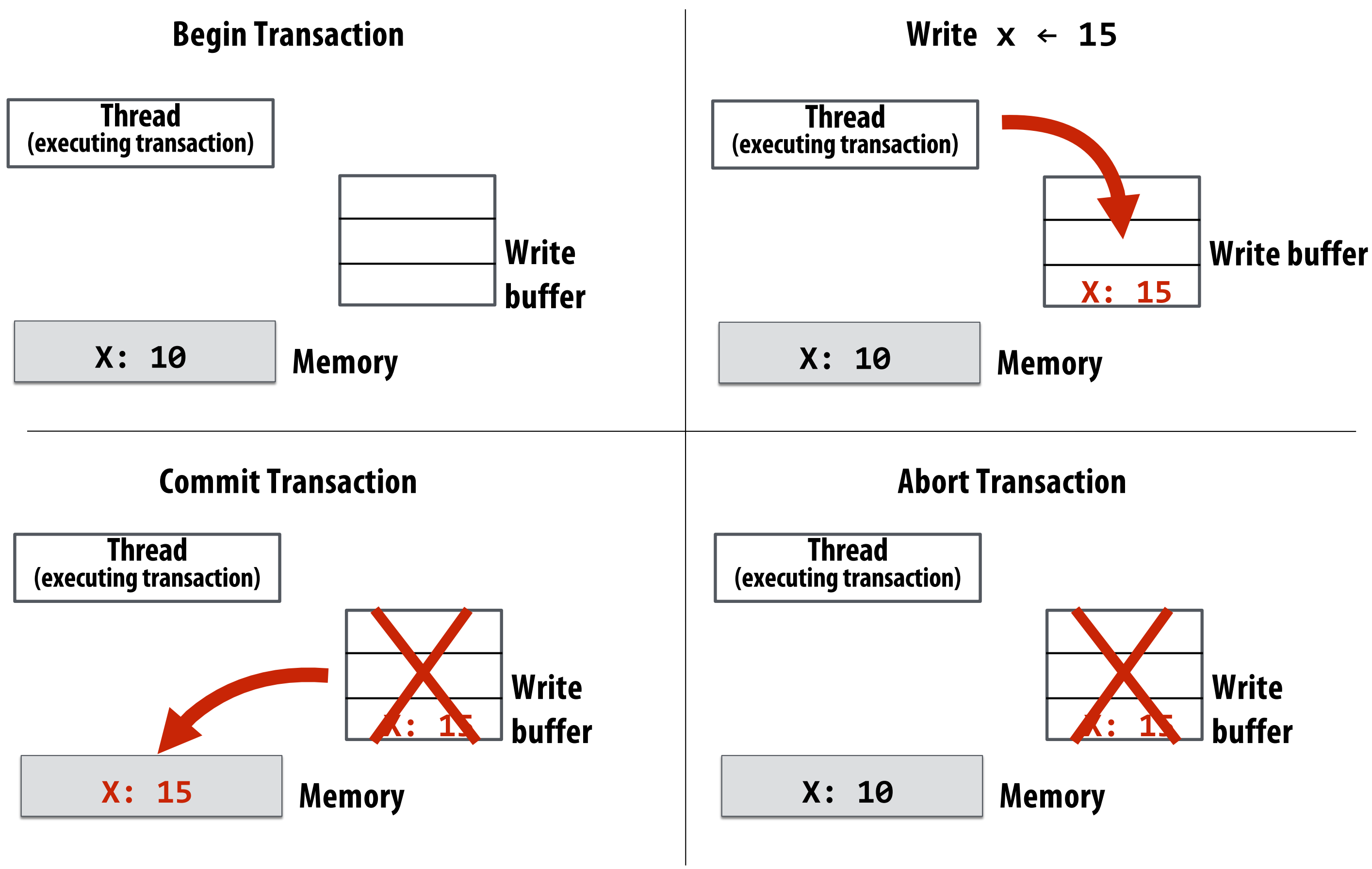
Eager versioning

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



Lazy versioning

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



Data versioning

- Goal: 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 (but not yet committed) 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 during the transaction
  - Write-set: addresses written during the transaction

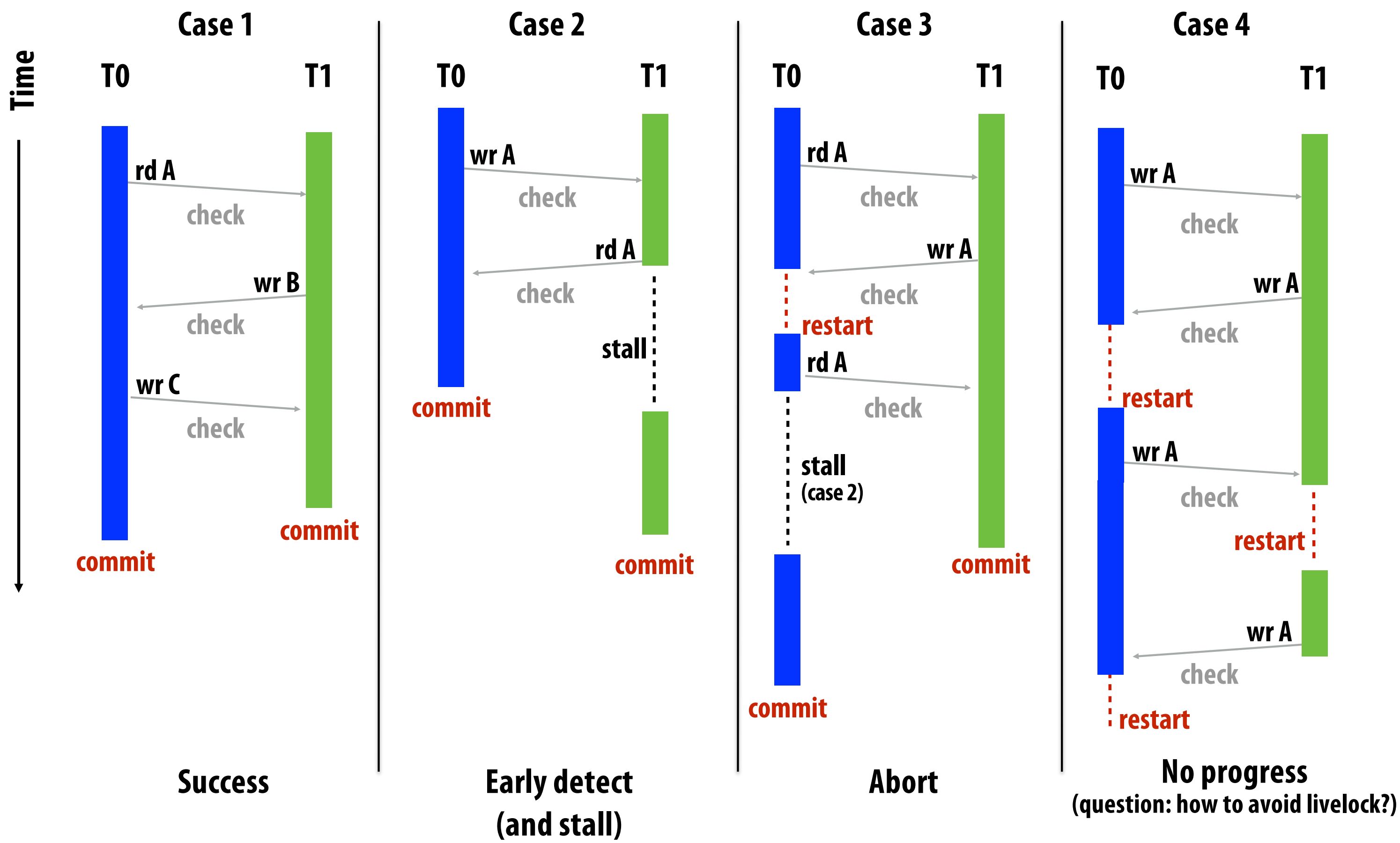
Pessimistic detection

- Check for conflicts (immediately) during loads or stores
  - 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 policies to handle common case fast

Pessimistic detection examples

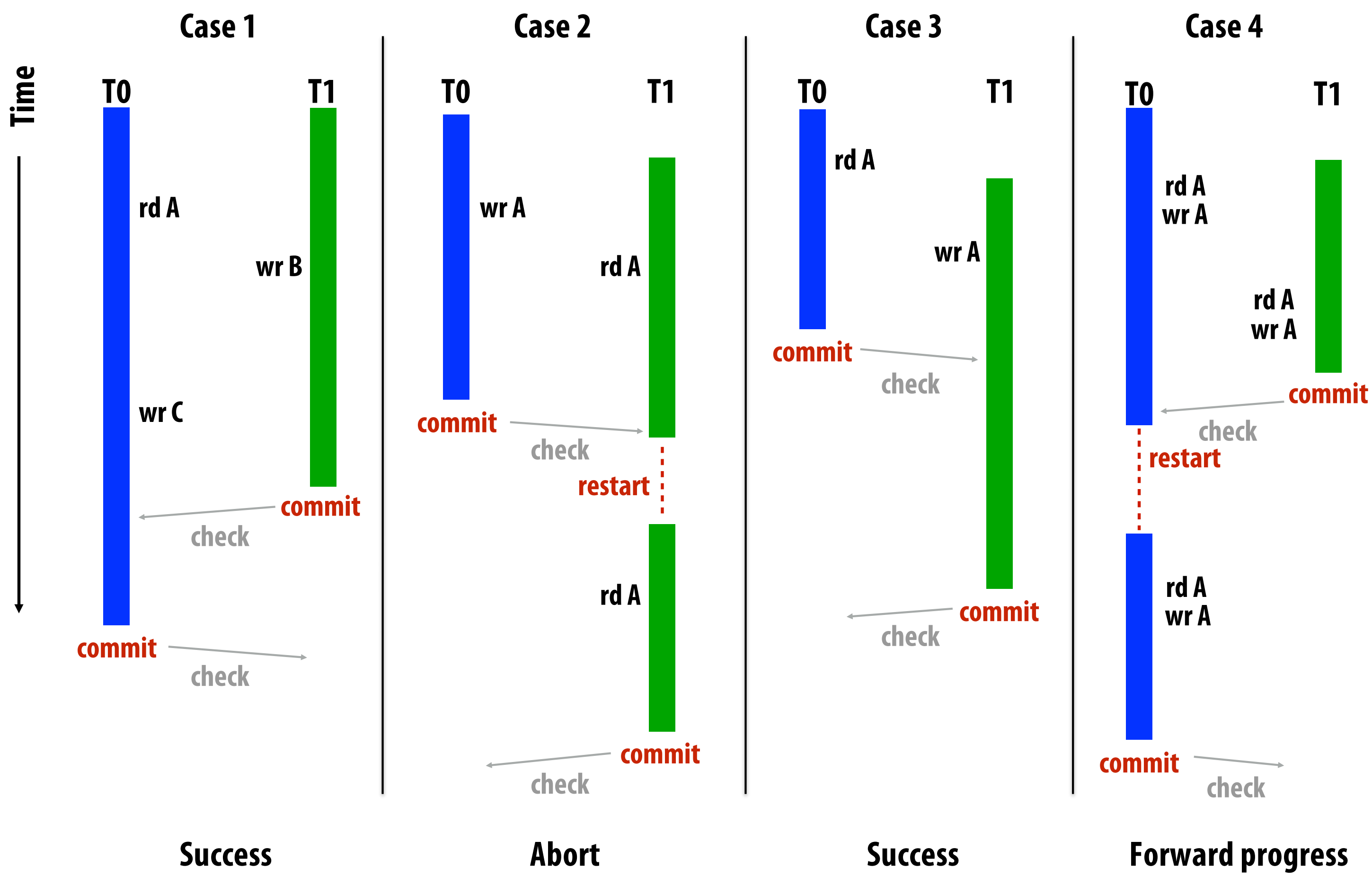
Note: diagrams assume "aggressive" contention manager on writes: writer wins, so other transactions abort)



Optimistic detection

- Detect conflicts when a transaction attempts to commit
  - 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

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

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

Software Transactional Memory

```
atomic {
   a.x = t1
   a.y = t2
   if (a.z == 0) {
   a.x = 0
   a.z = t3
   }
}
```

```
tmTxnBegin()
tmWr(&a.x, t1)
tmWr(&a.y, t2)
if (tmRd(&a.z) != 0) {
   tmWr(&a.x, 0);
   tmWr(&a.z, t3)
}
tmTxnCommit()
```

- Software barriers (STM function call) for TM bookkeeping
  - Versioning, read/write-set tracking, commit, …
  - Using locks, timestamps, data copying, …
- Requires function cloning or dynamic translation
  - Function used inside and outside of transaction

STM Runtime Data Structures

- **Transaction descriptor (per-thread)**
  - Used for conflict detection, commit, abort, …
  - Includes the read set, write set, undo log or write buffer

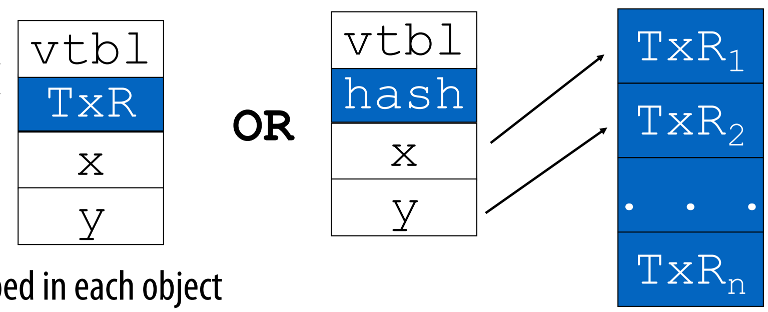
- **Transaction record (per data)**
  - Pointer-sized record guarding shared data
  - Tracks transactional state of data
    - Shared: accessed by multiple readers
      - Using version number or shared reader lock
    - Exclusive: access by one writer
      - Using writer lock that points to owner
    - BTW: same way that HW cache coherence works

Mapping Data to Transaction Records

Every data item has an associated transaction record

Java/C#

```
class Foo {
    int x;
    int y;
}
```



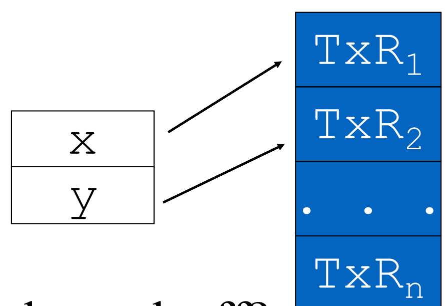
Embed in each object

Hash fields or array elements to global table

f(obj.hash, field.index)

C/C++

```
struct Foo {
    int x;
    int y;
}
```



Address-based hash into global table

Cache-line or word granularity

What's the tradeoff?

Conflict Detection Granularity

- **Object granularity**
  - Low overhead mapping operation
  - Exposes optimization opportunities
  - False conflicts (e.g. Txn 1 and Txn 2)
- **Element/field granularity (word)**
  - Reduces false conflicts
  - Improves concurrency (e.g. Txn 1 and Txn 2)
  - Increased overhead (time/space)
- **Cache line granularity (multiple words)**
  - Matches hardware TM
  - Reduces storage overhead of transactional records
  - Hard for programmer & compiler to analyze
- **Mix & match per type basis**
  - E.g., element-level for arrays, object-level for non-arrays

Txn 1

```
a.x = …
a.y = …
```

Txn 2

```
… = … a.z …
```

An Example STM Algorithm

- Based on Intel's McRT STM [PPoPP'06, PLDI'06, CGO'07]
  - Eager versioning, optimistic reads, pessimistic writes

- Based on timestamp for version tracking
  - Global timestamp
    - Incremented when a writing xaction commits
  - Local timestamp per xaction
    - Global timestamp value when xaction last validated

- Transaction record (32-bit)
  - LS bit: 0 if writer-locked, 1 if not locked
  - MS bits
    - Timestamp (version number) of last commit if not locked
    - Pointer to owner xaction if locked

STM Operations

- STM read (optimistic)
  - Direct read of memory location (eager)
  - Validate read data
    - Check if unlocked and data version ≤ local timestamp
    - If not, validate all data in read set for consistency
  - Insert in read set
  - Return value

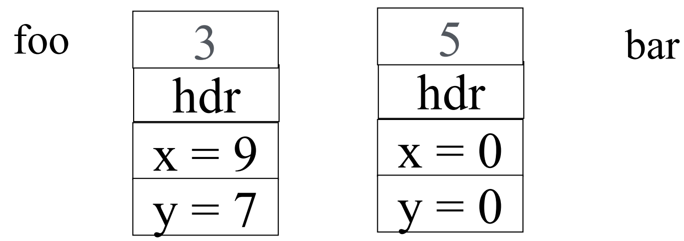
- STM write (pessimistic)
  - Validate data
    - Check if unlocked and data version ≤ local timestamp
  - Acquire lock
  - Insert in write set
  - Create undo log entry
  - Write data in place (eager)

STM Operations (cont)

- Read-set validation
  - Get global timestamp
  - For each item in the read set
    - If locked by other or data version > local timestamp, abort
  - Set local timestamp to global timestamp from initial step

- STM commit
  - Atomically increment global timestamp by 2 (LSb used for write-lock)
  - If preincremented (old) global timestamp > local timestamp, validate read-set
    - Check for recently committed transactions
  - For each item in the write set
    - Release the lock and set version number to global timestamp

STM Example



X1

```
atomic {
 t = foo.x;
 bar.x = t;
 t = foo.y;
 bar.y = t;
    }
```

X2

```
atomic {
t1 = bar.x;
t2 = bar.y;
    }
```

- **X1 copies object foo into object bar**
- **X2 should read bar as [0,0] or [9,7]**

STM Example

foo

3
hdr
x = 9
y = 7

bar

~~7~~ ~~5~~ 1
x = ~~0~~ 9
y = ~~0~~ 7

Commit

X1

```
atomic {
 t = foo.x;
 bar.x = t;
 t = foo.y;
bar.y = t;
    }
```

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

Undo <bar.x, 0> <bar.y, 0>

X2 waits

Abort

X2

```
atomic {
t1 = bar.x;
t2 = bar.y;
     }
```

Reads <bar, 5> <bar, 7>

No local or global time stamps
Each object has a time stamp

Challenges for STM Systems

- **Overhead of software barriers**
- **Function cloning**
- **Robust contention management**
- **Memory model (strong Vs. weak atomicity)**

Optimizing Software Transactions

```
atomic {
   a.x = t1
   a.y = t2
   if (a.z == 0) {
   a.x = 0
   a.z = t3
   }
}
```

```
tmTxnBegin()
tmWr(&a.x, t1)
tmWr(&a.y, t2)
if (tmRd(&a.z) != 0) {
   tmWr(&a.x, 0);
   tmWr(&a.z, t3)
}
tmTxnCommit()
```

- Monolithic barriers hide redundant logging & locking from the compiler

Optimizing Software Transactions

```
atomic {
   a.x = t1
   a.y = t2
   if (a.z == 0) {
   a.x = 0
   a.z = t3
   }
}
```

- Decomposed barriers expose redundancies

```
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
txnOpenForWrite(a)
txnLogObjectInt(&a.y, a)
a.y = t2
txnOpenForRead(a)
if(a.z != 0) {
 txnOpenForWrite(a)
 txnLogObjectInt(&a.x, a)
 a.x = 0
 txnOpenForWrite(a)
 txnLogObjectInt(&a.z, a)
 a.z = t3
}
```

Optimizing Software Transactions

```
atomic {
   a.x = t1
   a.y = t2
   if (a.z == 0) {
   a.x = 0
   a.z = t3
   }
}
```

```
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
txnLogObjectInt(&a.y, a)
a.y = t2
if (a.z != 0) {
   a.x = 0
   txnLogObjectInt(&a.z, a)
   a.z = t3
}
```

- Allows compiler to optimize STM code
- Produces fewer & cheaper STM operations

Compiler Optimizations for STM

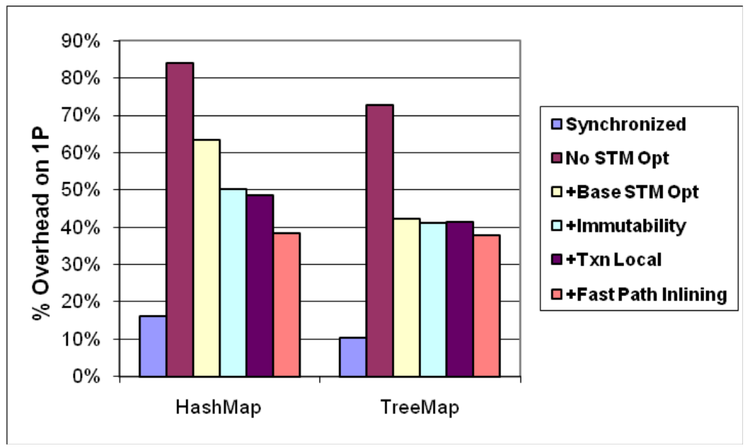
- Standard compiler optimizations
  - CSE, PRE, dead-code elimination, …
  - Assuming IR supports TM, few compiler mods needed

- STM-specific optimizations
  - Partial inlining of barrier fast paths
    - Often written in optimized assembly
  - No barriers for immutable and transaction local data

- Impediments to optimizations
  - Support for nested transactions
  - Dynamically linked STM library
  - Dynamic tuning of STM algorithm

Effect of Compiler Optimizations

- 1 thread overheads over thread-unsafe baseline



- With compiler optimizations
  - <40% over no concurrency control
  - <30% over lock-based synchronization

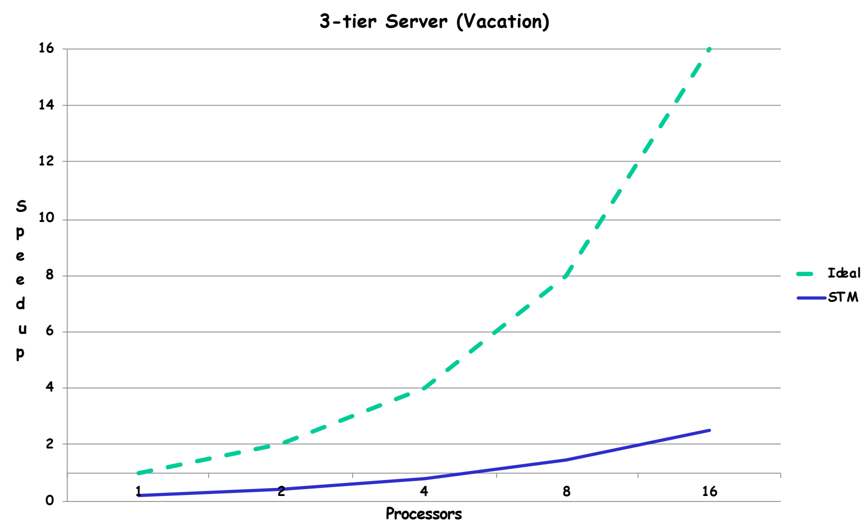
Function Cloning

- Problem: need two version of functions
  - One with and one without STM instrumentation

- Managed languages (Java, C#)
  - On demand cloning of methods using JIT

- Unmanaged languages (C, C++)
  - Allow programmer to mark TM and pure functions
    - TM functions should be cloned by compiler
    - Pure functions touch only transaction-local data
      - No need for clones
    - All other functions handled as irrevocable actions
  - Some overhead for checks and mode transitions

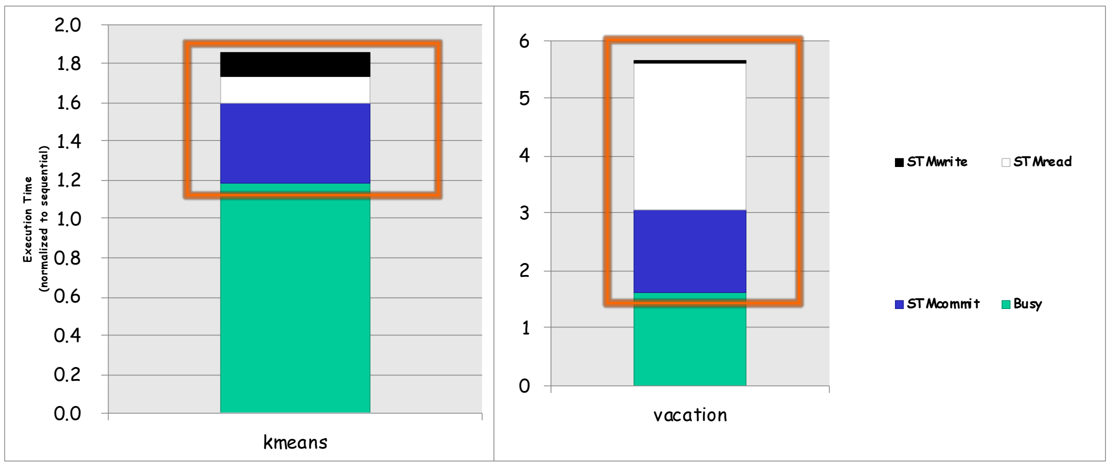
Motivation for Hardware Support



- STM slowdown: 2-8x per thread overhead due to barriers
  - Short term issue: demotivates parallel programming
  - Long term issue: energy wasteful
- Lack of strong atomicity
  - Costly to provide purely in software

Why is STM Slow?

- Measured single-thread STM performance



- 1.8x – 5.6x slowdown over sequential
- Most time goes in read barriers & validation
  - Most apps read more data than they write

Types of Hardware Support

- Hardware-accelerated STM systems (HASTM, SigTM, USTM, …)
  - Start with an STM system & identify key bottlenecks
  - Provide (simple) HW primitives for acceleration, but keep SW barriers

- Hardware-based TM systems (TCC, LTM, VTM, LogTM, …)
  - Versioning & conflict detection directly in HW
  - No SW barriers

- Hybrid TM systems (Sun Rock, …)
  - Combine an HTM with an STM by switching modes when needed
    - Based on xaction characteristics available resources, …

	HTM	STM	HW-STM
Write versioning	HW	SW	SW
Conflict detection	HW	SW	HW

HTM Performance Example

- 2x to 7x over STM performance
  - Within 10% of sequential for one thread
  - Scales efficiently with number of processors
  - Uncommon cases not a performance challenge

STM Efficiency

- Old question: what is the overhead of STM?
  - 1.3x – 6x

- New question: what is the performance of a well engineered system using STM vs. a well engineered system using fine-grained locks?
  - Use STM aware data structures
    - They don't rely on STM, but play nicely with STM
  - Use STM to compose these data structures
  - Nathan Bronson

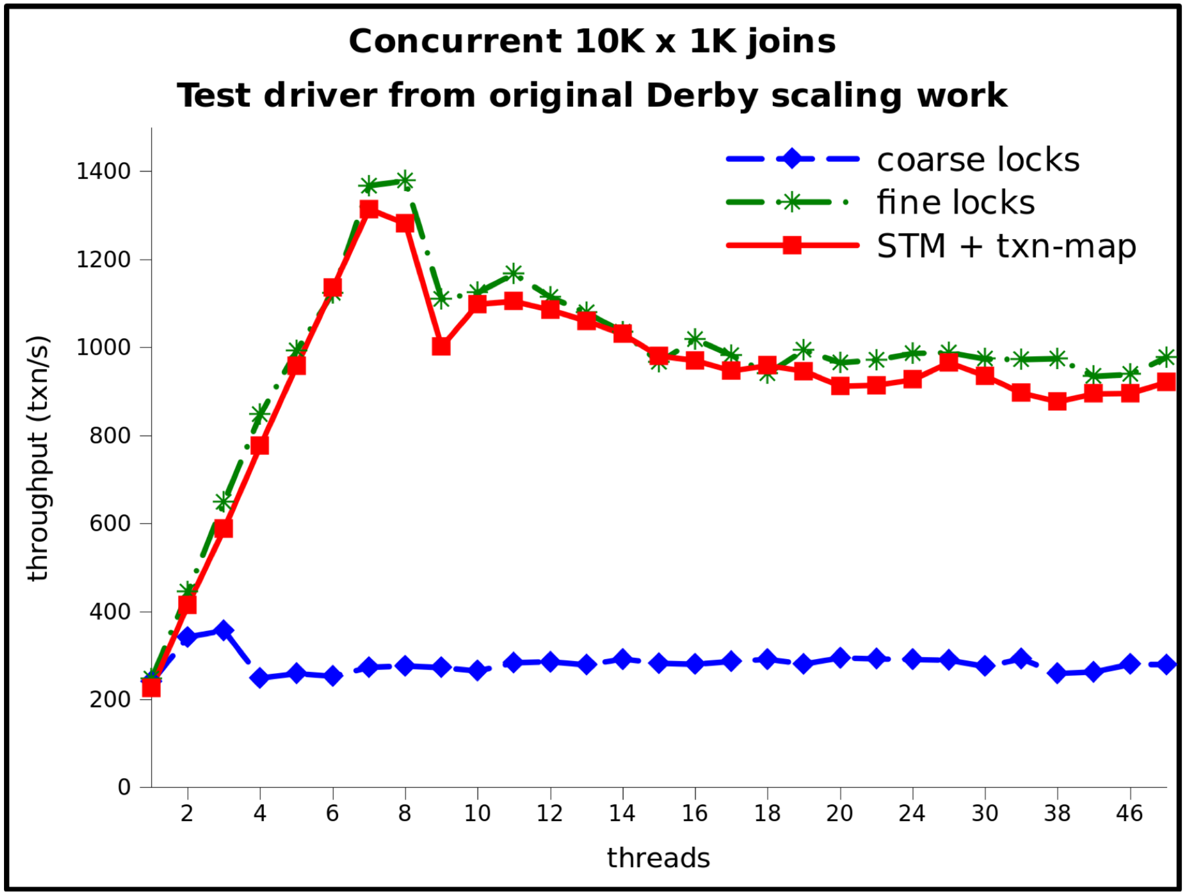
Lock manager inside Apache's Derby SQL Database

- Row-level locks
  - Multiple lock modes
  - Tricky conflict and queue logic
  - Automatic deadlock cycle detection
  - Per-row, per-txn, and per-group operations

- Using ConcurrentHashMaps + fine-grained locks
  - 2204 non-comment lines of Java
  - 128 lines of discussion to prove that new code is thread safe!
    - Informal proof that the deadlock detector is not itself subject to deadlock!

- Using STM + HashTrieTxnMaps
  - 418 non-comment lines of Scala
  - A number of corner cases avoided (races, timeouts, etc.)

Performance Comparison (Read Heavy)



TM Implementation Summary 1

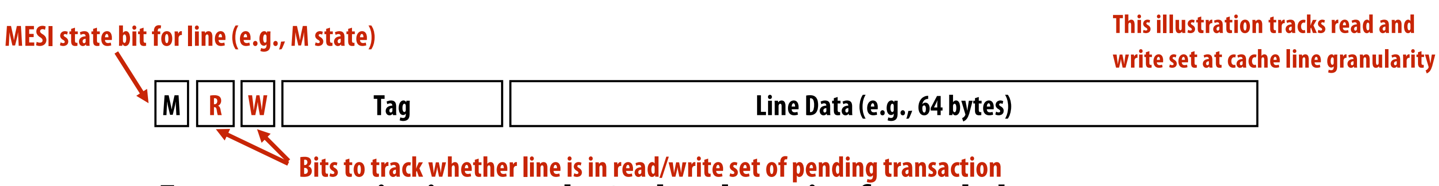
- TM implementation
  - Data versioning: eager or lazy
  - Conflict detection: optimistic or pessimistic
    - Granularity: object, word, cache-line, …
- Software TM systems
  - Compiler adds code for versioning & conflict detection
    - Note: STM barrier = instrumentation code
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, …)
    - Transactional record per data (locked/version)

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)

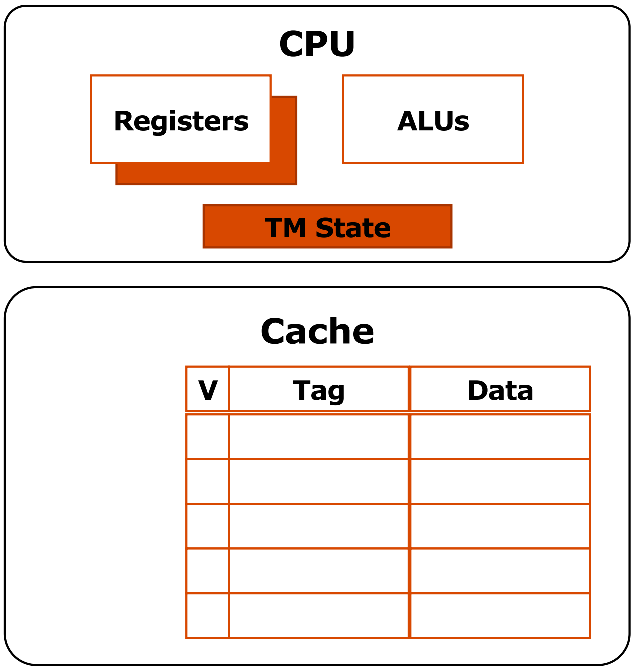
HTM design

- 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



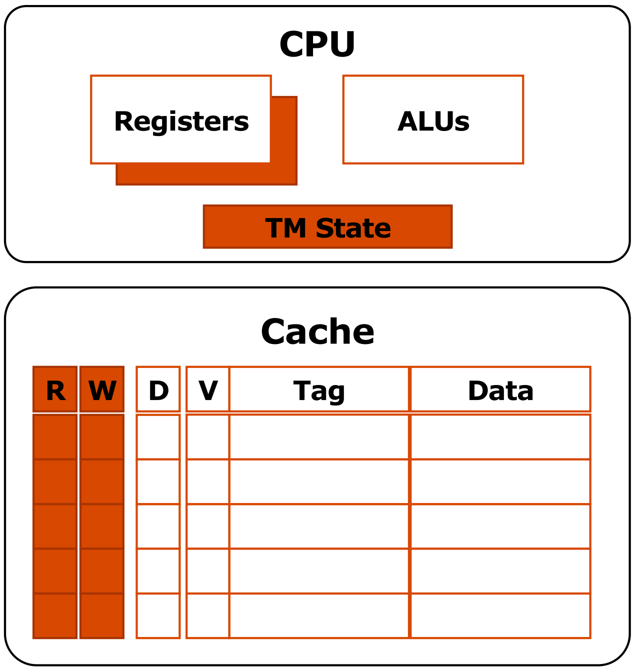
  - For eager versioning, need a 2nd cache write for undo log
- Coherence requests check R/W bits to detect conflicts
  - Observing shared request to W-word is a read-write conflict
  - Observing exclusive (intent to write) request to R-word is a write-read conflict
  - Observing exclusive (intent to write) request to W-word is a write-write conflict

Example HTM implementation: lazy-optimistic



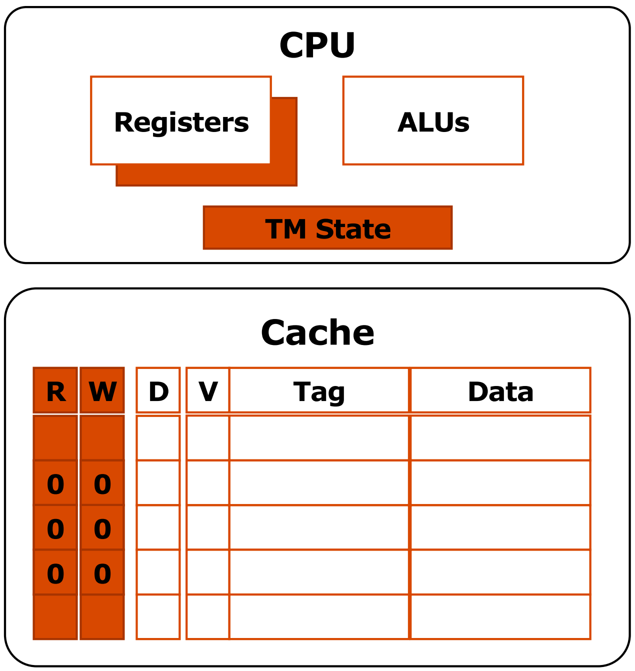
- **CPU changes**
  - Ability to checkpoint register state (available in many CPUs)
  - TM state registers (status, pointers to abort handlers, ...)

Example HTM implementation: lazy-optimistic



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

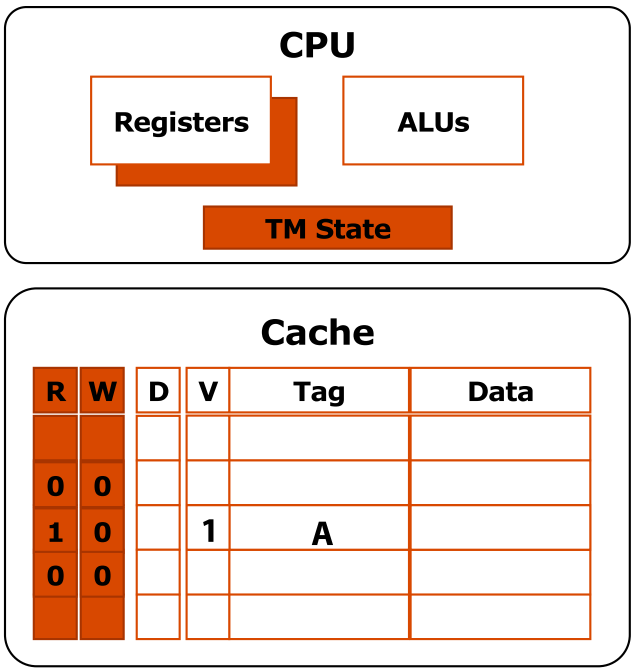
HTM transaction execution




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

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

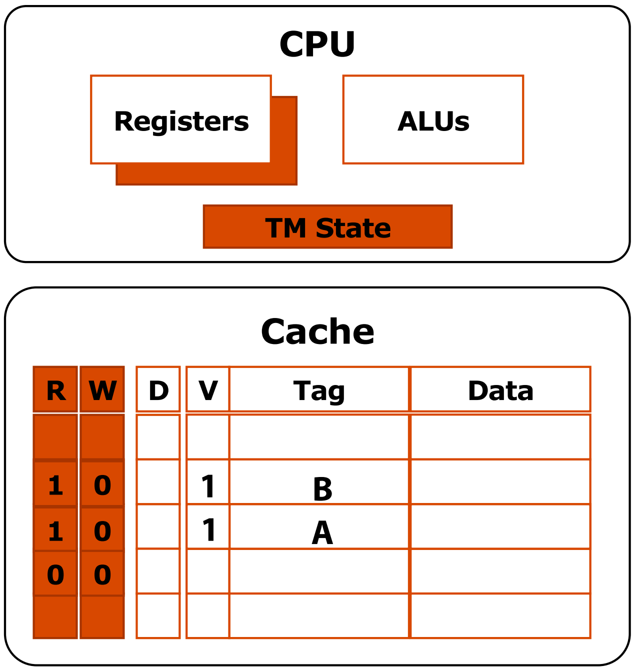
HTM transaction execution




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

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

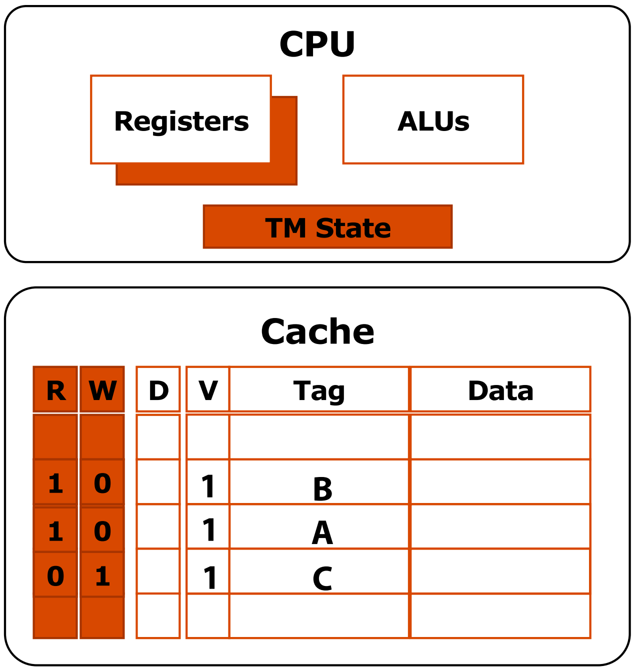
HTM transaction execution




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

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

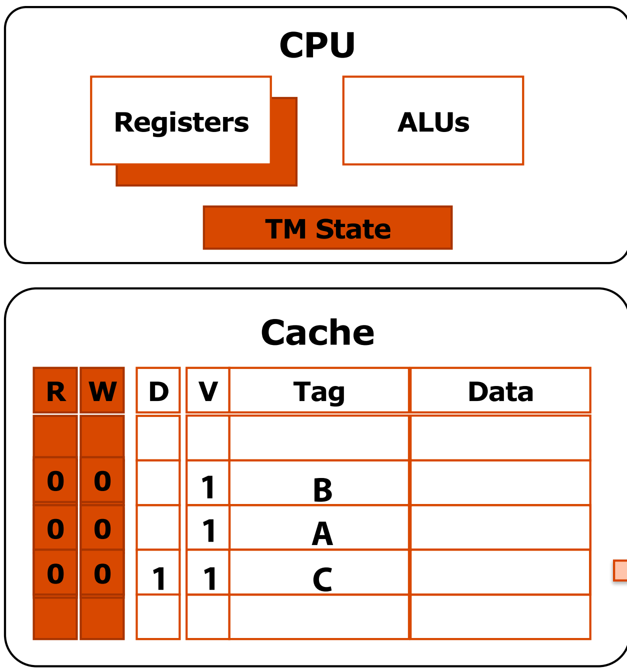
HTM transaction execution




```
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



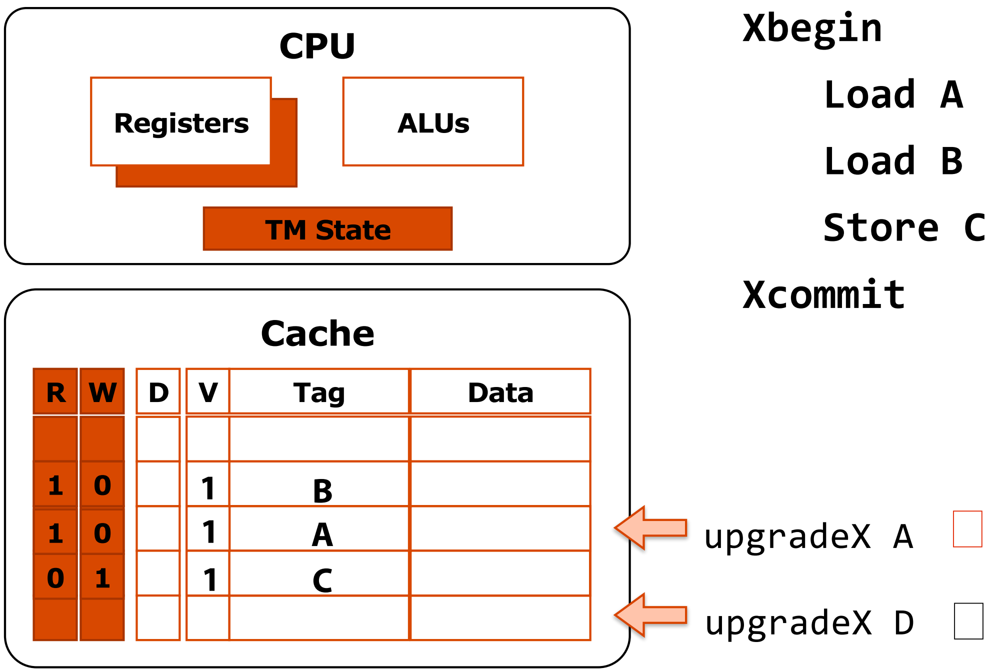

```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

upgradeX C
(result: C is now in 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
```

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

Summary: transactional memory

- Atomic construct: declaration that atomic behavior must be preserved by the system
  - 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
- Software TM systems
  - Compiler adds code for versioning & conflict detection
    - Note: STM barrier = instrumentation code
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, …)
    - Transactional record per data (locked/version)
- Hardware transactional memory
  - Versioned data is kept in caches
  - Conflict detection mechanisms built upon coherence protocol