Transactional Memory

Concepts, Implementations, & Opportunities

Christos Kozyrakis

Pervasive Parallelism Lab

Stanford University

http://ppl.stanford.edu/~christos

My Background

- Assistant Professor of EE & CS @ Stanford
  - PhD from UC Berkeley, BS from U. of Crete
  - Research focus: computer systems
    - Architecture, design, runtimes, programming models, …
- Active research projects
  - Transactional memory (**http://tcc.stanford.edu**)
  - Systems security (**http://raksha.stanford.edu**)
  - Energy-efficient data-centers (**http://joulesort.stanford.edu**)
- Past research
  - Network switches (ugrad work @ ICS-FORTH)
  - Multimedia processors (grad work @ UC Berkeley)

My Experience on Transactional Memory

- Hardware support
  - TCC architecture [ISCA'04, ASPLOS'04, PACT'05], HTM virtualization [ASPLOS'06]
  - ISA for HTM systems [ISCA'06]
  - SigTM hybrid system [ISCA'07]
- Programming environments
  - Java+TM=Atomos [SCOOL'05, PLDI'06], transctional collection classes [PPoPP'07]
  - OpenMP+GCC+TM=OpenTM [PACT'07, **http://opentm.stanford.edu**]
- Applications
  - Basic characterization [HPCA'05, WTW'06]
  - STAMP benchmark suite [IISWC'08, **http://stamp.stanford.edu**]
- Full-system prototypes
  - ATLAS FPGA-based prototype for HTM [DATE'07, FPGA'07]
- TM beyond concurrency control
  - Fix DBT races [HPCA'08], replay/tuning/debugging on ATLAS [ISCA'07 tutorial]

Acknowledgements

- Ali Adl-Tabatabai & Bratin Saha (Intel)
  - Slides from our joined tutorial
  - Hot Chips'06, PACT'06, PPoPP'07, PACT'07
- My co-authors on TM papers
  - TCC group at Stanford
  - Ali Adl-Tabatabai, Bratin Saha, Jim Larus
- The TM research community
  - TM bibliography: **http://www.cs.wisc.edu/trans-memory**
  - Extensive listing of TM papers

Course Objectives

- We will
  - Introduce basic TM concepts & interfaces
  - Cover a wide range of implementation tradeoffs
  - Discuss opportunities beyond parallelism
  - Provide basis for further reading & research on TM

- Non-goals
  - Discuss every paper on TM technology
    - Impossible for an active research field
  - Conclude with a single, optimal implementation
    - Although we will draw some important insights
  - Go over a large number of performance graphs
    - Prefer to focus on insights instead
  - Discuss how TM integrates with other novel ideas for parallelism

Perspective: TM & Parallel Programming

- The challenges of parallel programming
  1. Finding independent tasks in the algorithm
  2. Mapping tasks to execution units (e.g. threads)
  3. Defining & implementing synchronization
     - Races, deadlock avoidance, memory model issues
  4. Composing parallel tasks
  5. Recovering from errors
  6. Portable & predictable performance
  7. Scalability
  8. Locality management
  9. All the sequential issues as well…

Course Outline

- Lecture 1
  - TM introduction & programming concepts
- Lecture 2
  - Introduction to TM implementation
  - Software TM systems
- Lecture 3
  - Hardware support for TM
- Lecture 4
  - Hardware/software interface for TM
  - TM uses beyond concurrency control

Course Etiquette

- Please ask questions
  - Best way to set course pace & focus
  - Best way to get most out of the course fee
    - You could study my slides at your home
  - Other students will benefit from your questions

- Keep in mind
  - Must cover a decent subset of the material, so…
    - May defer some questions till an appropriate slide
    - May defer some questions for offline
    - May only provide the insight & a pointer to the details
  - I don’t have all the answers…

Questions?

Lecture 1: TM Concepts & Programming

- Outline
  - TM definition & key advantages
  - TM programming constructs
  - Caveats and open issues

- Disclaimer
  - The exact semantics and constructs for TM in various languages are still an open research issue
  - The goal of this lecture is to introduce the constructs & related issues in order to motivate the implementation
    - Will not provide formal/strict semantics

Motivation: The Parallel Programming Crisis

- Multi-core chips ⇒inflection point for SW development
  - Scalable performance now requires parallel programming
- Parallel programming up until now
  - Limited to people with access to large parallel systems
  - Using low-level concurrency features in languages
    - Thin veneer over underlying hardware
  - Too cumbersome for mainstream software developers
    - Difficult to write, debug, maintain and even get some speedup
- We need better concurrency abstractions
  - Goal = easy to use + good performance
  - 90% of the speedup with 10% of the effort

Transactional Memory (TM)

- Memory transaction [Lomet'77, Knight'86, Herlihy & Moss'93]
  - 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

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 [Kung’81]
  - Slow down only on conflicts (R-W or W-W)

Advantages of TM

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

Example: Java 1.4 HashMap

- Fundamental data structure
  - Map: Key → Value

```
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;
 }
```

- Not thread safe – 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 map m
          return m.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

Concurrent HashMap (Java 5)

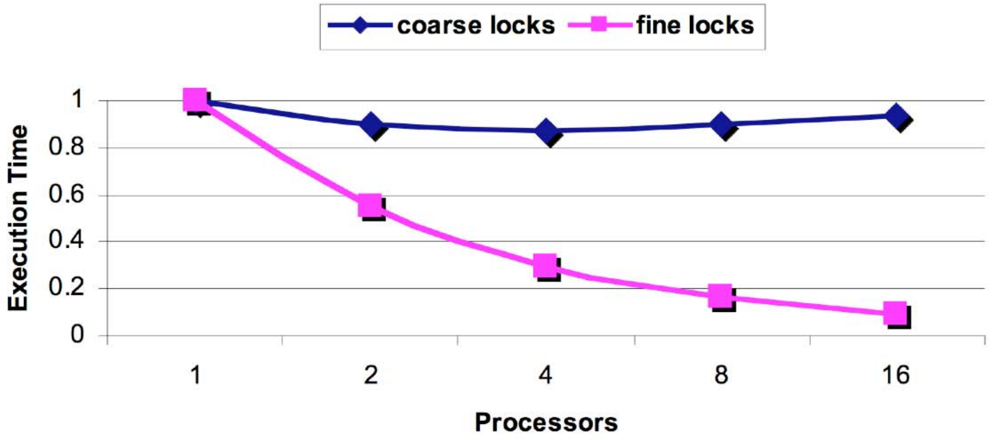
```
public Object get(Object key) {
  int hash = hash(key);
  // Try first without locking...
  Entry[] tab = table;
  int index = hash & (tab.length - 1);
  Entry first = tab[index];
  Entry e;

  for (e = first; e != null; e = e.next) {
    if (e.hash == hash && eq(key, e.key)) {
      Object value = e.value;
      if (value != null)
        return value;
      else
        break;
    }
  }
  …
```

```
…
// Recheck under synch if key not there or interference
Segment seg = segments[hash & SEGMENT_MASK];
synchronized(seg) {
  tab = table;
  index = hash & (tab.length - 1);
  Entry newFirst = tab[index];
  if (e != null || first != newFirst) {
    for (e = newFirst; e != null; e = e.next) {
      if (e.hash == hash && eq(key, e.key))
        return e.value;
    }
  }
  return null;
 }
}
```

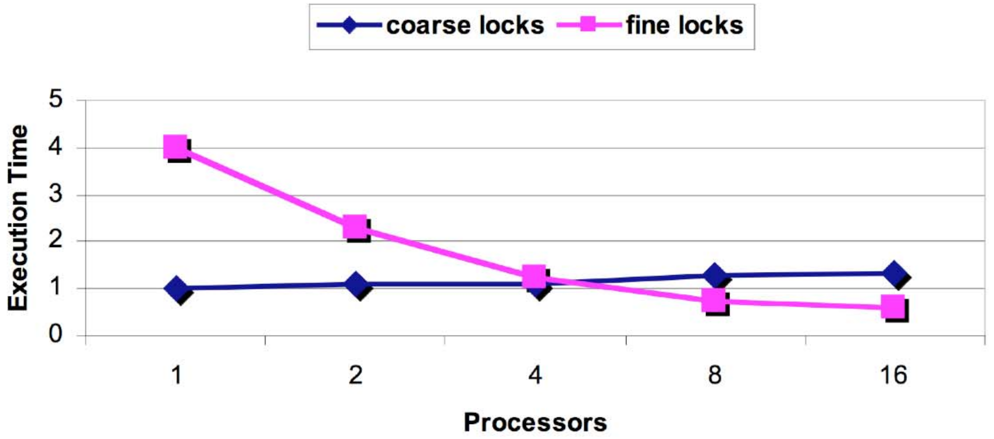
- Fine-grain synchronized concurrent HashMap
  - Pros: fine-grain parallelism, concurrent reads
  - Cons: complex & error prone

Performance: Locks

Hash-Table



Balanced Tree



Transactional HashMap

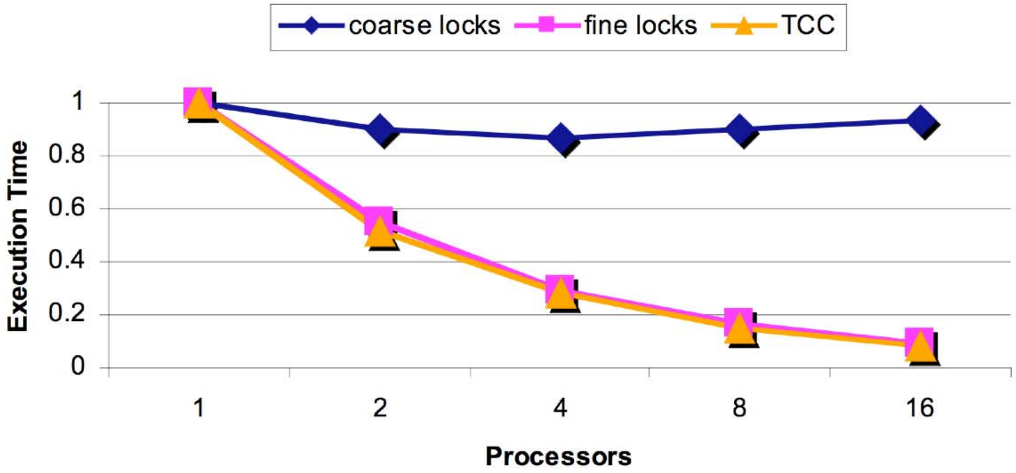
- Simply enclose all operation in atomic block
  - System ensures atomicity

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

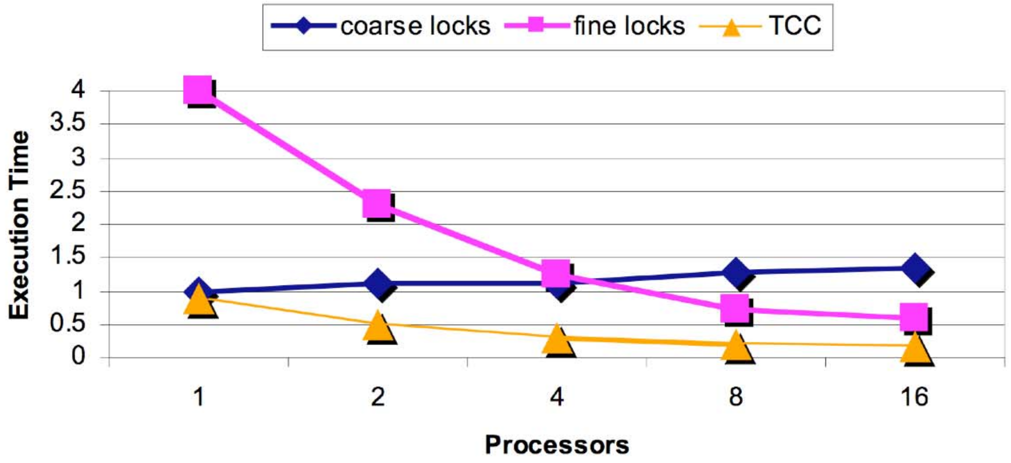
- Transactional HashMap
  - Pros: thread-safe, easy to program
  - Q: good performance & scalability?
    - Depends on the implementation, but typically yes

Performance: Locks Vs Transactions

HashMap



Balanced Tree



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
    - No locks held by a failing threads…
  - Open question: how to best communicate exception info

Composability: Locks

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

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

- Composing lock-based code is tough
  - Goal: hide intermediate state during transfer
  - Need <u>global</u> locking methodology now…
- Between the rock & the hard place
  - Fine-grain locking: can lead to deadlock

Composability: Locks

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

```
void transfer(C, D, amount)
   synchronized(bank){
        withdraw(C, amount);
        deposit(A, amount);
   }
```

- Composing lock-based code is tough
  - Goal: hide intermediate state during transfer
  - Need global locking methodology now…
- Between the rock & the hard place
  - Fine-grain locking: can lead to deadlock
  - Coarse-grain locking: no concurrency

Composability: Transactions

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

- Transactions compose gracefully
  - Programmer declares global intend (atomic transfer)
    - No need to know of a 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)

Programming with TM (continued)

- Basic atomic blocks: atomic{}
- User-triggered abort: abort
- Conditional synchronization: retry
- Composing code sequences: orelse
- Integration with parallel models: OpenTM

User-triggered Abort

- Abort statement
  - Undo current transaction (no visible writes)
  - Jump to a specified code location
    - User Vs. system initiated abort

- Abort uses
  - Check high-level invariants in user code
  - Error and exception handling

```
void transfer(A, B, amount)
   atomic{
      try {
        work();
      }
      catch(error1) {  fix_code(); }
      catch(error2) {  abort(); }
   }
```

Conditional Synchronization with Retry

```
Object blockingDequeue
// Block until queue is not empty
   atomic{
       if (isEmpty()) retry;
       return dequeue();
}
```

- Retry statement
  - Rolls back current transaction
  - Waits for change in state accessed by the transaction
    - Everything or what specified with a watch() statement
  - Store by another thread implicitly signals blocked thread
    - No lost wake up compared to traditional wait-notify schemes

- Alternative: conditional atomic statements
  - Specify & test condition at transaction start

Composing Code Sequences

```
atomic{
  q1.blockingDequeue()
} orelse {
  q2.blockingDequeue();
} orelse {
  q3.blockingDequeue();
}
```

- Orelse statement
  - Allows composition of alternative code statements
  - If one clause fails due to retry, try next alternative
    - Sequential order of clauses

Integration with Parallel Models

- 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=42, group=6)
  for (i=0; i<N; i++) {
     bin[A[i]] = bin[A[i]]+1;
   }
```

TM Caveats and Open Issues

- TM Vs. Locks
- I/O and unrecoverable actions
- Interaction with non-transactional code

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
  - Atomic blocks can suffer from atomicity violations
    - Atomic action in algorithm split into two atomic blocks

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?
- How can we code this pattern with atomic blocks?

Example: Atomicity Violation

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

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

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

- What should be the transaction boundaries for the thread 1 code?

I/O and Other Irrevocable Actions

- Challenge: difficult to undo output & redo input
  - I/O devices, I/O registers,…
- Alternative solutions (open problem)
  - Buffer output & log input
    - Finalize output & clear log at commit
    - Does not work if atomic does input after output
  - Guarantee that transaction will not abort
    - Abort interfering transactions or sequentialize the system
    - Does not work with abort(), input-after-output
  - Transaction-based systems
    - Multiple transactional devices (TM, log-based FS, …)
    - Manager coordinates transactions across devices
      - See IBM's Quicksilver system as a pre-TM era example

Interactions with Non-Transactional Code

- Two basic alternatives

- Weak atomicity
  - Transactions are serializable only against other transations
  - No guarantees about interactions with non-transactional code
- Strong atomicity
  - Transactions are serializable against all memory accesses
  - Non-transactional loads/stores are 1-instruction transactions

- The tradeoff
  - Strong atomicity seems intuitive
  - Predictable interactions for a wide range of coding patterns
  - But, strong atomicity has high overheads for software TM

Example of Atomicity Challenges

```
//Thread 1
atomic(){
  t1 = A;
  …
  …
  t2 = A;
}
```

```
//Thread 2
A++;
```

- With strong atomicity
  - t1==t2 always
  - Thread 2 may cause thread 1 transaction to abort
- With weak atomicity
  - t1 may not be equal to t2
  - Depends on exact interleaving, TM implementation, …

Example of Atomicity Challenges

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

```
//Thread 2
t=A;
```

- With strong atomicity
  - Thread 2 reads value of A before or after transaction
- With weak atomicity
  - Thread 2 may also read intermediate value
  - Depends on exact interleaving, TM implementation, …

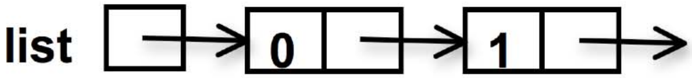
An Example without Races: Privatization

Thread 1

```
synchronized(list) {
 if (list != NULL) {
      e = list;
      list = e.next;
}}
r1 = e.x;
r2 = e.x;
assert(r1 != r2);
```

Thread 2

```
synchronized(list) {

  if (list != NULL) {

      p = list;

      p.x = 9;

}
```



- Privatization example
  - Thread 1 removes first element from list
  - Correctly synchronized code with locks
    - Thread 1 assertion should always succeed
  - What happens if we use atomic() instead?

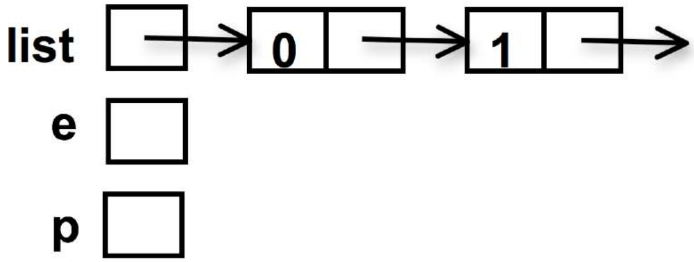
Privatization on a Weakly Atomic TM

Thread 1

```
atomic{
  if (list ! = NULL) {
      e = list;
      list = e.next;
}}
r1 = e.x;
r2 = e.x;
assert(r1 != r2);
```

Thread 2

```
atomic{
  if (list!=NULL) {
      p = list;
      p.x = 9;
}
```



- Assuming an eager-versioning STM system
  - Similar issues with lazy-versioning without strong atomicity
  - Similar issues with publication patterns

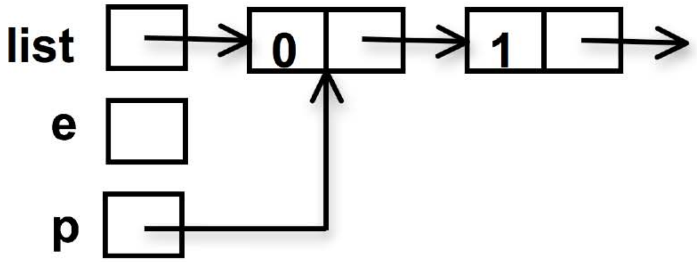
Privatization on a Weakly Atomic TM

Thread 1

```
atomic{
   if (list ! = NULL) {
        e = list;
        list = e.next;
}}
r1 = e.x;
r2 = e.x;
assert(r1 != r2);
```

Thread 2

```
atomic{
   if (list!=NULL) {
        p = list;
        p.x = 9;
}
```



- Assuming an eager-versioning STM system
  - Similar issues with lazy-versioning without strong atomicity
  - Similar issues with publication patterns

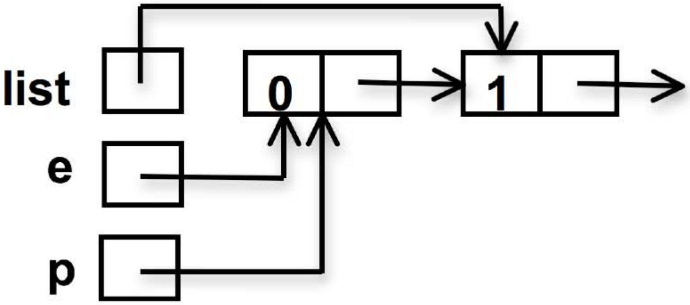
Privatization on a Weakly Atomic TM

Thread 1

```
atomic{
  if (list ! = NULL) {
      e = list;
      list = e.next;
}}
r1 = e.x;
r2 = e.x;
assert(r1 != r2);
```

Thread 2

```
atomic{
  if (list!=NULL) {
      p = list;
      p.x = 9;
}
```



- Assuming an eager-versioning STM system
  - Similar issues with lazy-versioning without strong atomicity
  - Similar issues with publication patterns

Privatization on a Weakly Atomic TM

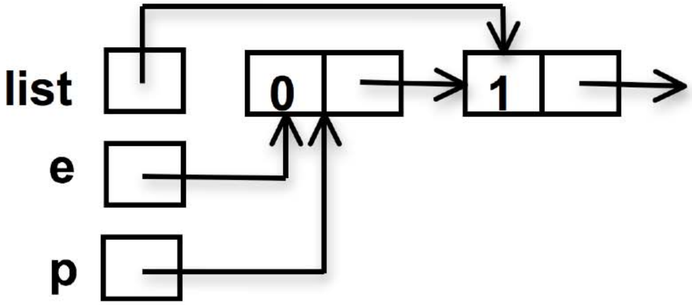
Thread 1

```
atomic{
  if (list ! = NULL) {
      e = list;
      list = e.next;
}}
r1 = e.x;
r2 = e.x;
assert(r1 != r2);
```

Commit

Thread 2

```
atomic{
  if (list!=NULL) {
      p = list;
      p.x = 9;
}
```



- Assuming an eager-versioning STM system
  - Similar issues with lazy-versioning without strong atomicity
  - Similar issues with publication patterns

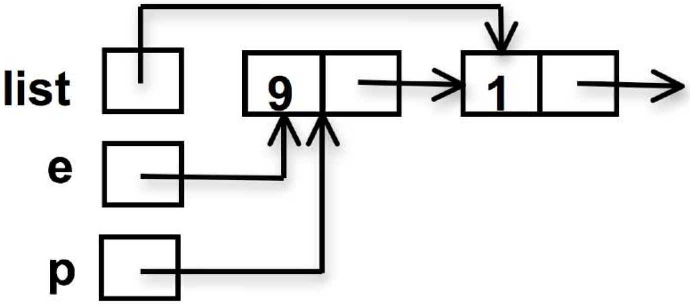
Privatization on a Weakly Atomic TM

Thread 1

```
atomic{
   if (list ! = NULL) {
        e = list;
        list = e.next;
}}
r1 = e.x;
r2 = e.x;
assert(r1 != r2);
```

Thread 2

```
atomic{
   if (list!=NULL) {
        p = list;
        p.x = 9;
}
```



- Assuming an eager-versioning STM system
  - Similar issues with lazy-versioning without strong atomicity
  - Similar issues with publication patterns

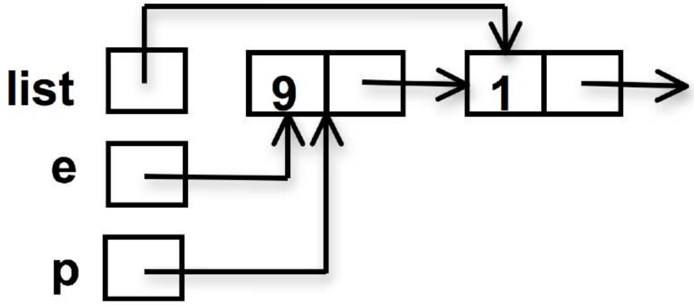
Privatization on a Weakly Atomic TM

Thread 1

```
atomic{
   if (list ! = NULL) {
        e = list;
        list = e.next;
}}
r1 = e.x;  // r1 = 9
r2 = e.x;
assert(r1 != r2);
```

Thread 2

```
atomic{
   if (list!=NULL) {
        p = list;
        p.x = 9;
}
```



- Assuming an eager-versioning STM system
  - Similar issues with lazy-versioning without strong atomicity
  - Similar issues with publication patterns

Privatization on a Weakly Atomic TM

Thread 1

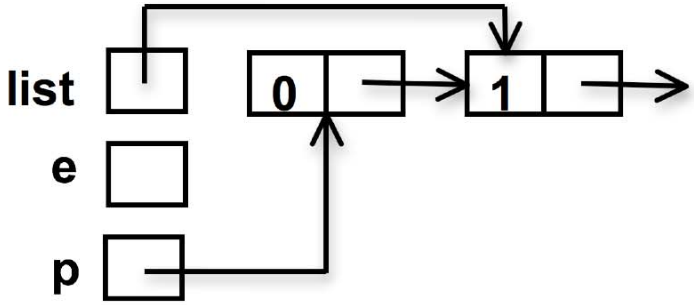
```
atomic{
   if (list ! = NULL) {
        e = list;
        list = e.next;
}}
r1 = e.x;  // r1 = 9
r2 = e.x;
assert(r1 != r2);
```

⇦ ⇨

Thread 2

```
atomic{
   if (list!=NULL) {
        p = list;
        p.x = 9;
}
```

Abort



- Assuming an eager-versioning STM system
  - Similar issues with lazy-versioning without strong atomicity
  - Similar issues with publication patterns

Privatization on a Weakly Atomic TM

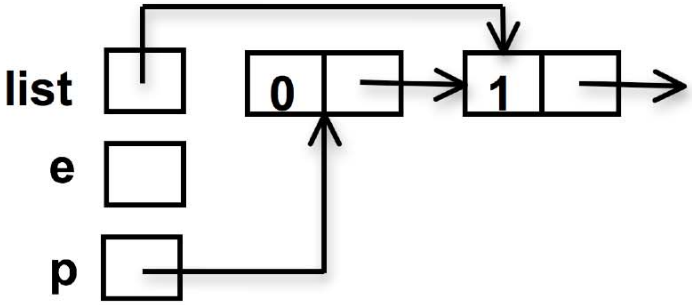
Thread 1

```
atomic{
  if (list ! = NULL) {
       e = list;
       list = e.next;
}}
r1 = e.x;  // r1 = 9
r2 = e.x;  // r2 = 0
assert(r1 != r2);
```

Thread 2

```
atomic{
  if (list!=NULL) {
       p = list;
       p.x = 9;
}
```

Abort



- Assuming an eager-versioning STM system
  - Similar issues with lazy-versioning without strong atomicity
  - Similar issues with publication patterns

Privatization on a Weakly Atomic TM

Thread 1

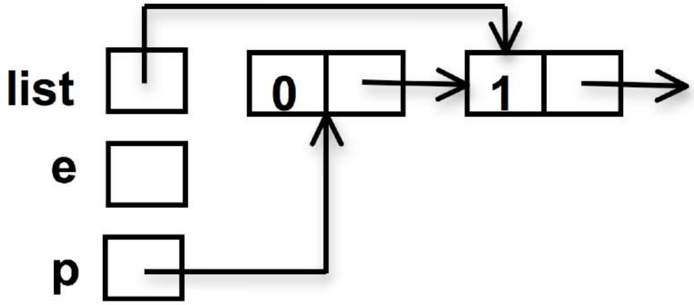
```
atomic{
  if (list ! = NULL) {
     e = list;
     list = e.next;
}}
r1 = e.x;  // r1 = 9
r2 = e.x;  // r2 = 0
assert(r1 != r2);
```

Fail

Thread 2

```
atomic{
  if (list!=NULL) {
     p = list;
     p.x = 9;
}
```

Abort



- Assuming an eager-versioning STM system
  - Similar issues with lazy-versioning without strong atomicity
  - Similar issues with publication patterns

Potential Solutions (Open Issue)

- Strong atomicity using hardware support
  - Full hardware TM or hardware-based conflict detection
- Optimize software overhead for strong atomicity
  - Through compiler optimizations for private and non-shared data
  - Possible for managed languages; difficult for unmanaged
- Programming models that explicitly segregate transactional from non transactional data
  - Allows correct handling of privatization & publication patterns
- Alternative system semantics
  - Single lock atomicity, disjoint lock atomicity, …
  - Guarantees & costs in between strong and weak atomicity
  - Similar to the discussion on relaxed consistency models

Lecture 1: Select References

Basics

- Herlihy & Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures, ISCA, 1993
- Kung & Robinson. On Optimistic Concurrency Control, ACM Trans. on DBs, 1981

TM Overview

- Larus & Rajwar. Transactional Memory, Morgan & Claypool Publishers, 2007
- Larus & Kozyrakis. Transactional Memory, CACM, 2008

TM Programming & Caveats

- Harris & Fraser. Language Support for Lightweight Transactions, OOPSLA, 2003
- Haris. Composable Memory Transactions, PPoPP, 2005
- Carlstrom et al. The Atomos Transactional Programming Language, PLDI, 2006
- Adl-Tabatabai et al. Compiler and runtime support for efficient software transactional memory, PLDI, 2006
- Lu et al. AVIO: Detecting Atomicity Violation Bugs via Access Interleaving Invariants, ASPLOS, 2006
- Shpeisman et al. Enforcing Isolation and Ordering in STM, PLDI, 2007
- Yoo et al. Kicking the Tires of Software Transactional Memory: When the Going Gets Tough, SPAA, 2008
- Welc et al. Irrevocable Transactions and their Applications, SPAA, 2008

Questions?

Transactional Memory

Concepts, Implementations, & Opportunities

Christos Kozyrakis

Pervasive Parallelism Lab

Stanford University

http://ppl.stanford.edu/~christos

Lecture 1 Summary

- 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, …

Lecture 1 Summary (cont)

- TM applicability
  - Apps with irregular or unstructured parallelism
    - Difficult to prove independence in advance
    - Difficult to partition data in advance
  - Examples: 3-tier system, graphs apps, AI apps, …

- A note to keep in mind
  - TM does not generate new parallelism
    - It just helps you tap into what is there
  - TM target: 90% of benefit @ 10% of work
    - Given infinite time & a lock, you should always be able to do as well as TM (roughly)

Lecture 2:
TM Implementation & Software TM

- Outline
  - Implementation requirements for TM
    - Data versioning techniques
    - Conflict detection techniques
    - Design space tradeoffs
  - Software TM systems (STM)
    - STM data structures
    - Example STM algorithm
    - STM optimizations & challenges

TM Implementation Basics

- TM systems must provide atomicity and isolation
  - Without sacrificing concurrency
- Basic implementation requirements
  - Data versioning
  - Conflict detection & resolution
- Implementation options
  - Hardware transactional memory (HTM)
  - Software transactional memory (STM)
  - Hybrid transactional memory
    - Hardware accelerated STMs and dual-mode systems

Data Versioning

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

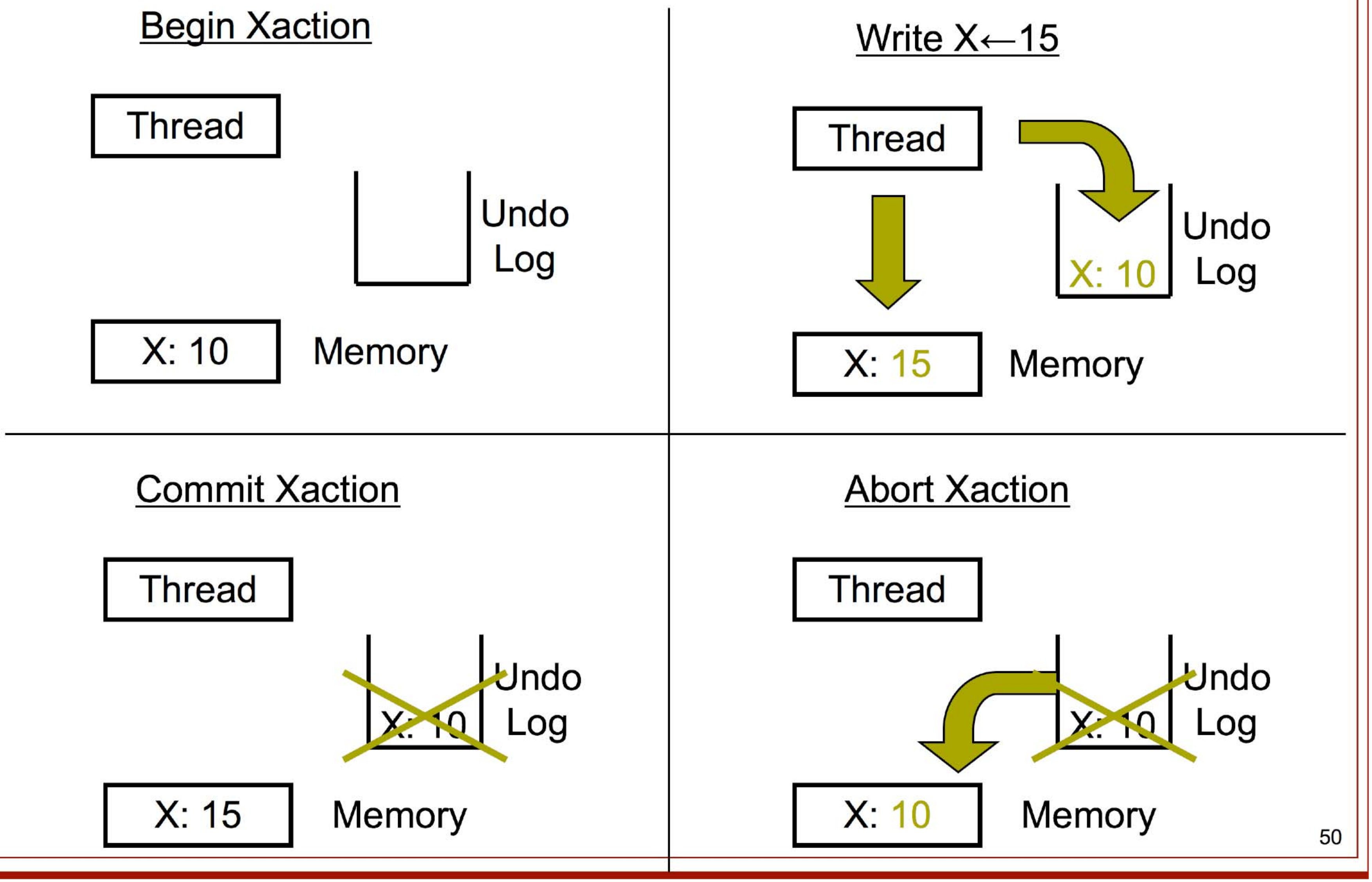
Eager versioning (undo-log based)

- Update memory location directly
- Maintain undo info in a log
- + Faster commit, direct reads (SW)
- – Slower aborts, fault tolerance issues

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, indirect reads (SW)

Eager Versioning Illustration



Begin Xaction

Thread

Undo Log

X: 10

Memory

Write X←15

Thread

Undo Log

X: 10

X: 15

Memory

Commit Xaction

Thread

Undo Log

X: 10

X: 15

Memory

Abort Xaction

Thread

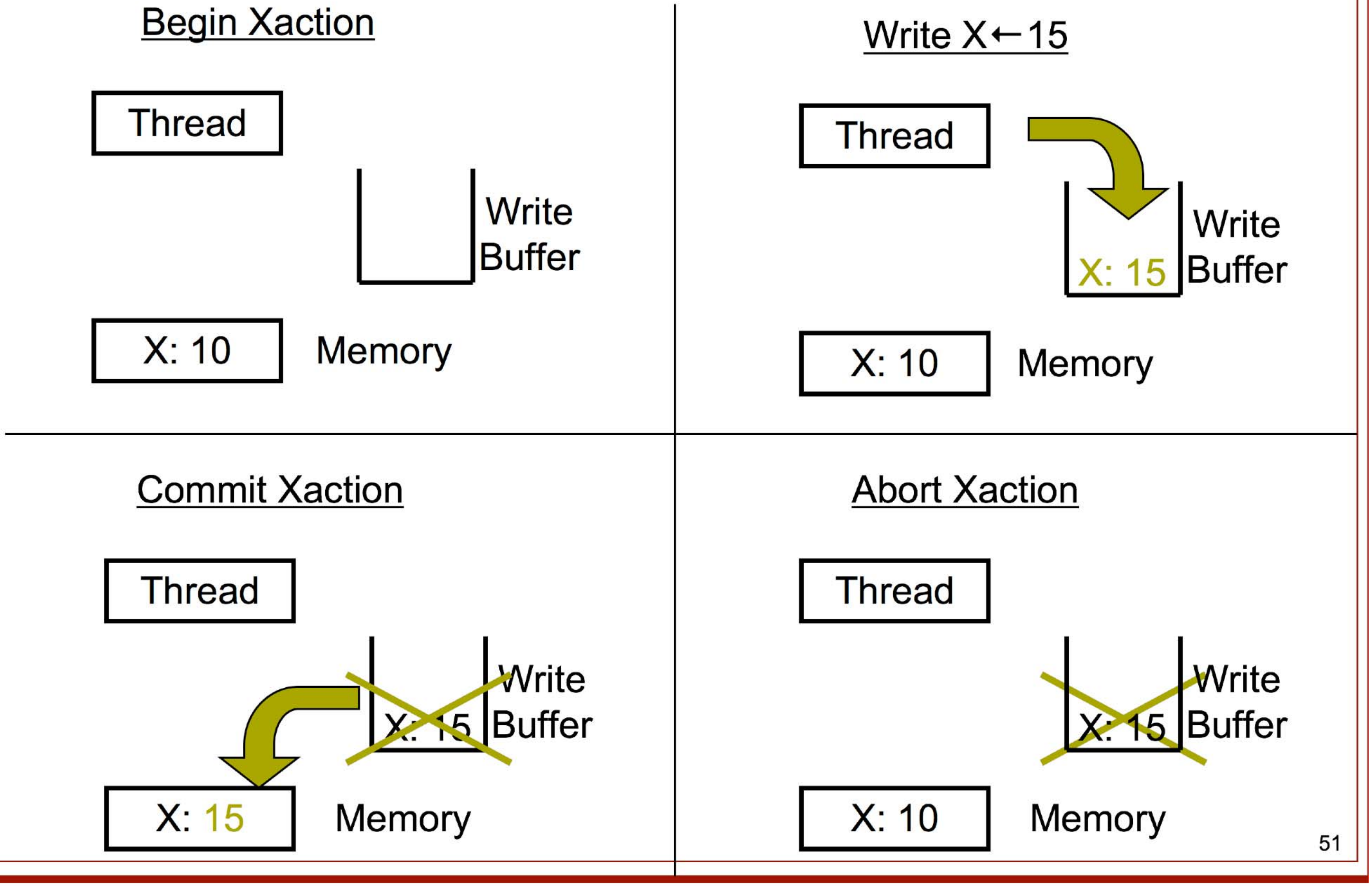
Undo Log

X: 10

X: 10

Memory

Lazy Versioning Illustration



Begin Xaction

Thread

Write Buffer

X: 10

Memory

Write X←15

Thread

X: 15

Write Buffer

X: 10

Memory

Commit Xaction

Thread

X: 15

Write Buffer

X: 15

Memory

Abort Xaction

Thread

X: 15

Write Buffer

X: 10

Memory

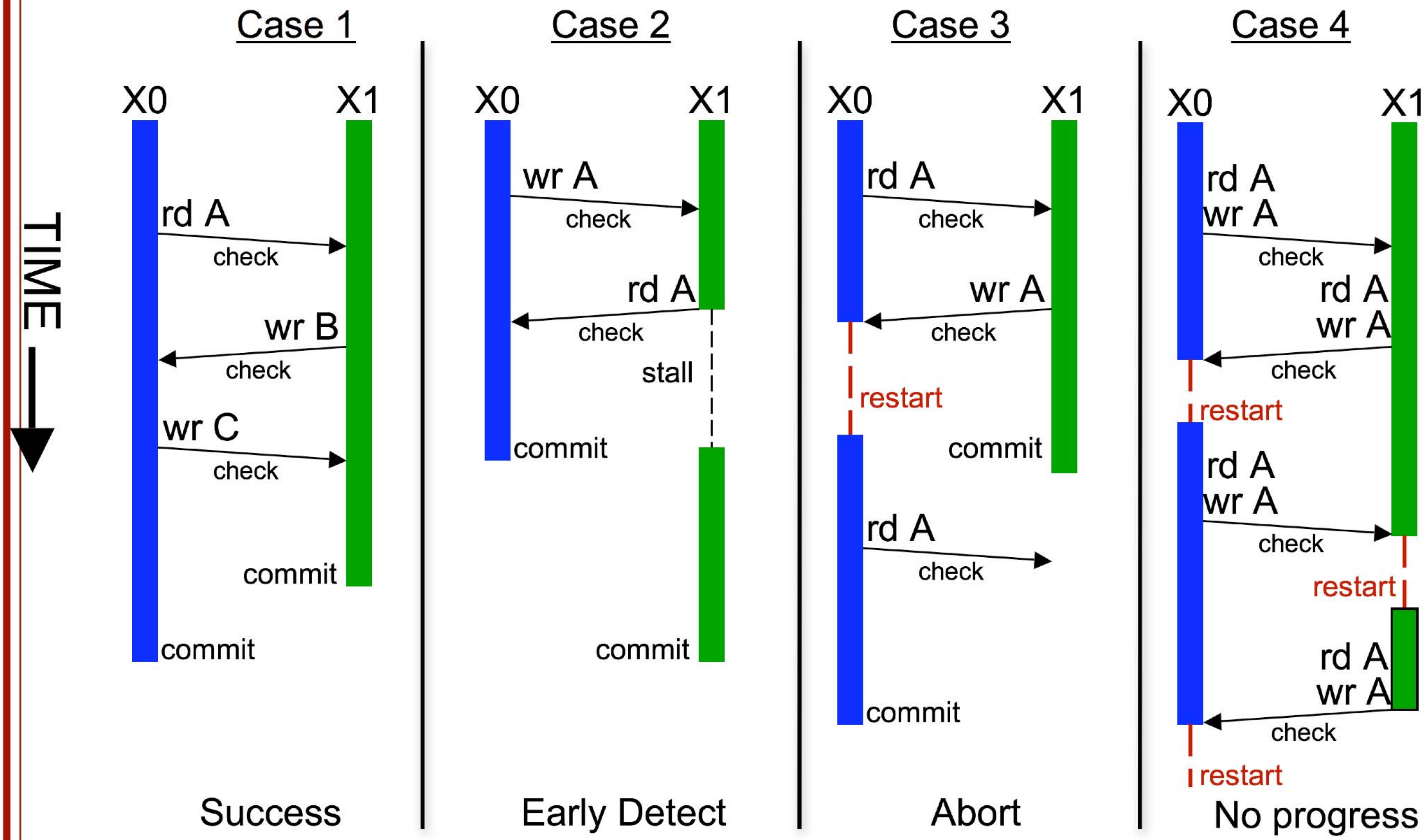
Conflict Detection

- Detect and handle conflicts between transaction
  - Read-Write and (often) Write-Write conflicts
  - 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
  - SW: SW barriers using locks and/or version numbers
  - HW: check through coherence actions
- Use contention manager to decide to stall or abort
  - Various priority policies to handle common case fast

Pessimistic Detection Illustration

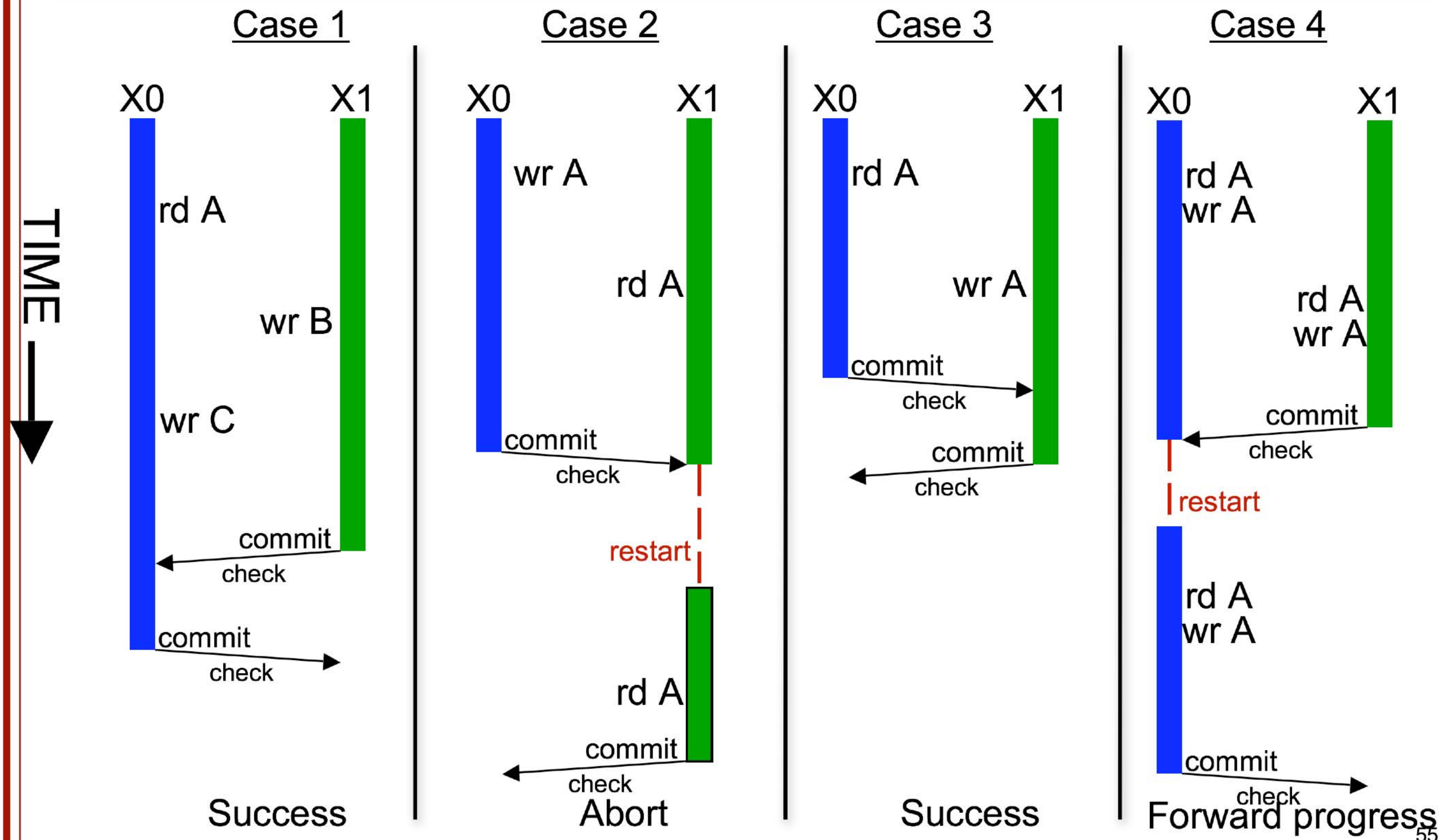


Conflict Detection (cont)

2. Optimistic detection
    - Detect conflicts when a transaction attempts to commit
        - SW: validate write/read-set using locks or version numbers
        - 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: optimistic & pessimistic schemes together
    - Several STM systems use optimistic for reads and pessimistic for writes

Optimistic Detection Illustration



Conflict Detection Tradeoffs

- Pessimistic conflict detection (aka encounter or eager)
  - + Detect conflicts early
    - Undo less work, turn some aborts to stalls
  - – No forward progress guarantees, more aborts in some cases
  - – Locking issues (SW), fine-grain communication (HW)

- Optimistic conflict detection (aka commit or lazy)
  - + Forward progress guarantees
  - + Potentially less conflicts, shorter locking (SW), bulk communication (HW)
  - – Detects conflicts late, still has fairness problems

Conflict Detection Granularity

- Object granularity (SW/hybrid)
  - + 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
  - + Works for both HW/SW

- Mix & match ➔ 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
- 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 is still an open questions
  - May be different for HW, SW, and hybrid
  - Will discuss further in STM and HTM sections of the course

Questions?

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 for TM bookkeeping
  - Versioning, read/write-set tracking, commit, …
  - Using locks, timestamps, data copying, …
- Requires function cloning or dynamic translation

STM Approaches

- Static Vs dynamic
  - Static: declare in advance all data access
  - Dynamic: dynamically handle accesses in program
  - Nearly all recent STMs are dynamic

- Non-blocking Vs lock-based
  - Non-blocking: rely on non-blocking algorithms
    - Non-blocking STMs use lazy versioning
    - Overhead of reads (indirection or search write-buffer)
  - Lock-based: rely on blocking locks
    - Can implement eager versioning (fast reads)
    - There are also lock-based lazy systems (e.g., TL2)

- Will focus on dynamic, lock-based STMs

STM Runtime Data Structures

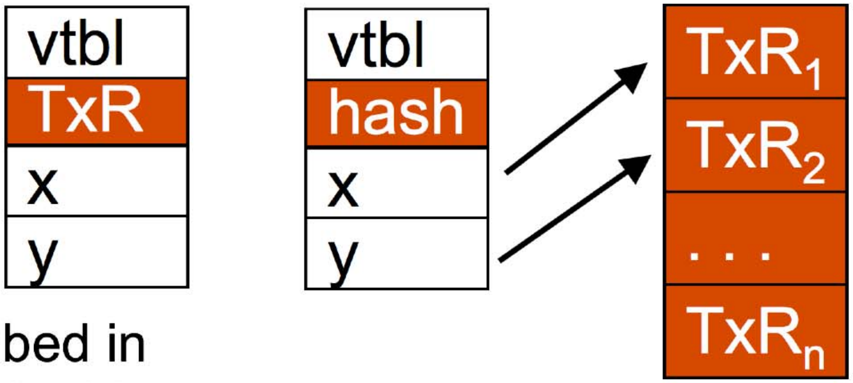
- Transaction descriptor (per-thread)
  - Used for conflict detection, commit, abort, …
  - Includes the read set, write set, undo log or write buffer
- Transaction memento (per thread)
  - Used for nesting & partial rollback
  - Includes checkpoints of machine and transaction descriptor
- Transaction 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

Mapping Data to Transaction Records

Every data item has an associated transaction record

Java/C#

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



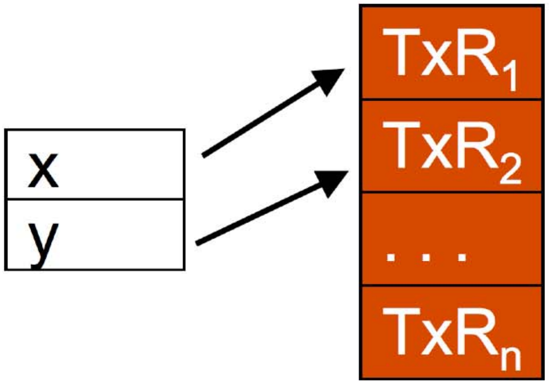
Embed in each object

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

Conflict Detection Granularity

- Object granularity
  - Low overhead mapping operation
  - Exposes optimization opportunities
- Element/field granularity
  - Reduces false sharing
  - Improves scalability
- Cache line granularity
  - 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 an 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 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

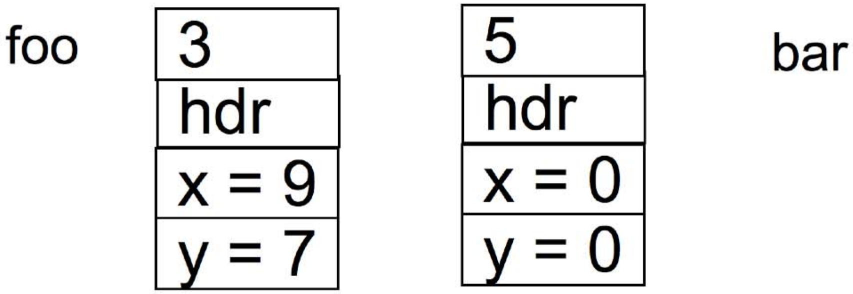
- STM write (pessimistic)
    - Validate data
        - Check if unlocked and data version ≤ local timestamp
    - Acquire lock
    - Insert in write set
    - Create undo log entry
    - Write data in place (eager)

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
  - If old global timestamp > local timestamp, validate read-set
  - For each item in the write set
    - Release the lock and increment version number by 2

STM Illustration


foo

3
hdr
x = 9
y = 7

5
hdr
x = 0
y = 0

bar

T1

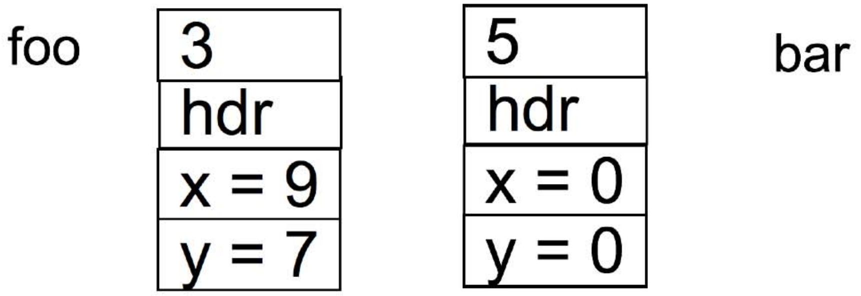
```
atomic {
   t = foo.x;
   bar.x = t;
   t = foo.y;
   bar.y = t;
}
```

T2

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

- T1 copies object foo into object bar
- T2 should read bar to be [0,0] or [9,9]

STM Illustration



foo

3
hdr
x = 9
y = 7

5
hdr
x = 0
y = 0

bar

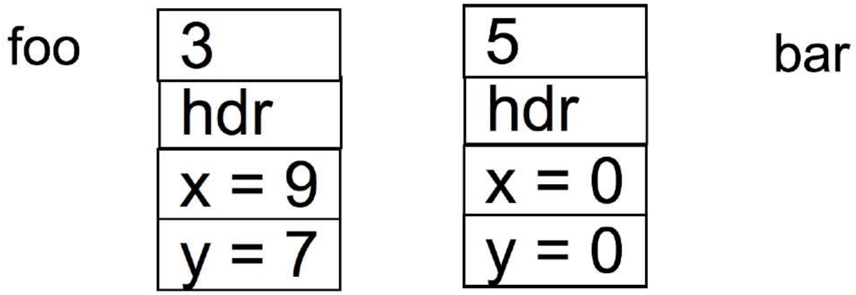
T1

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

T2

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

STM Illustration



foo

3
hdr
x = 9
y = 7

5
hdr
x = 0
y = 0

bar

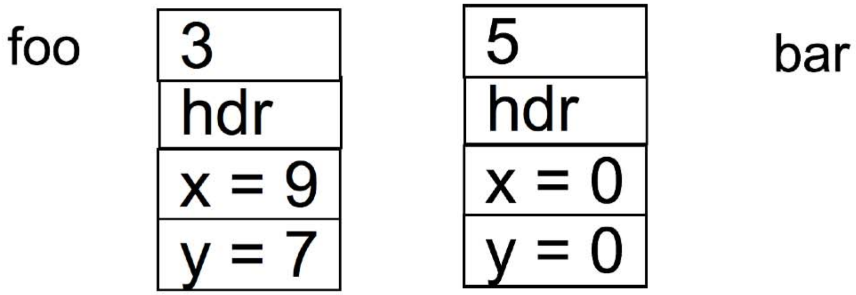
T1

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

T2

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

STM Illustration



T1

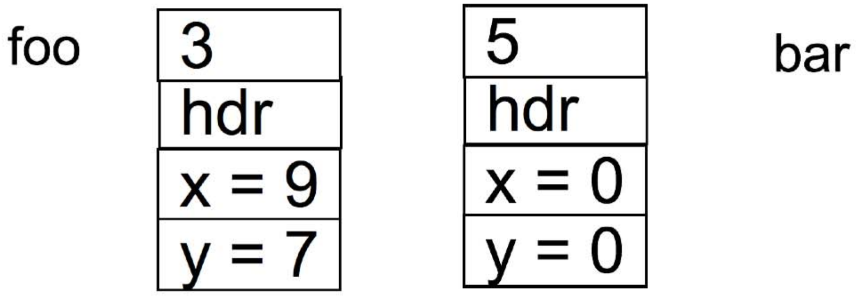
```
atomic {
    t = foo.x; ←
    bar.x = t;
    t = foo.y;
    bar.y = t;
}
```

Reads <foo, 3>

T2

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

STM Illustration



foo

3
hdr
x = 9
y = 7

bar

5
hdr
x = 0
y = 0

T1

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

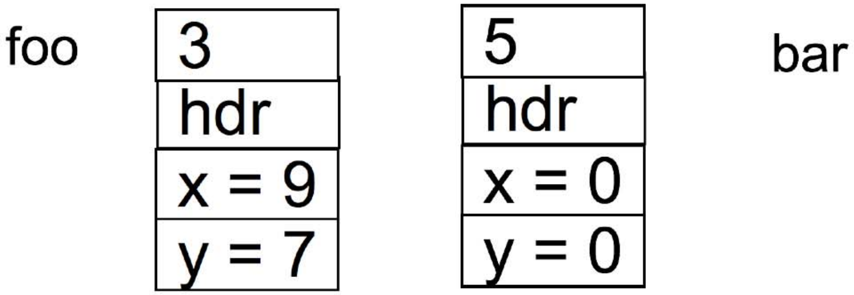
Reads <foo, 3>

T2

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

Reads <bar, 5>

STM Illustration



T1

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

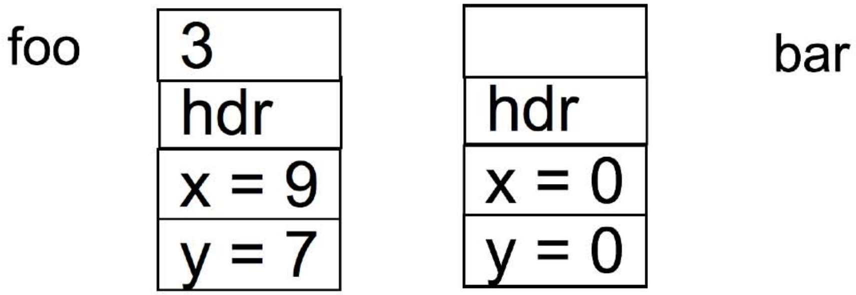
Reads <foo, 3>

T2

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

Reads <bar, 5>

STM Illustration



foo

3
hdr
x = 9
y = 7

hdr
x = 0
y = 0

bar

T1

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

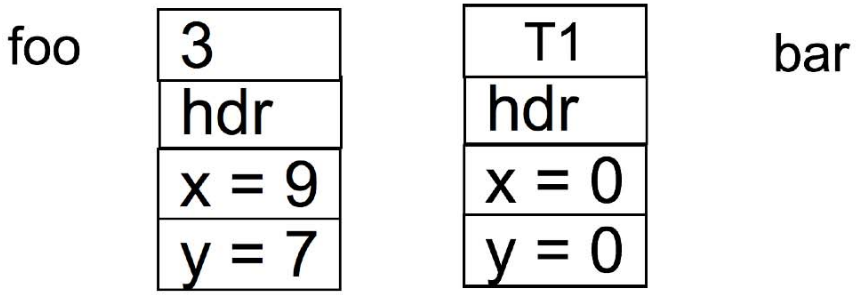
Reads <foo, 3>

T2

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

Reads <bar, 5>

STM Illustration



T1

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

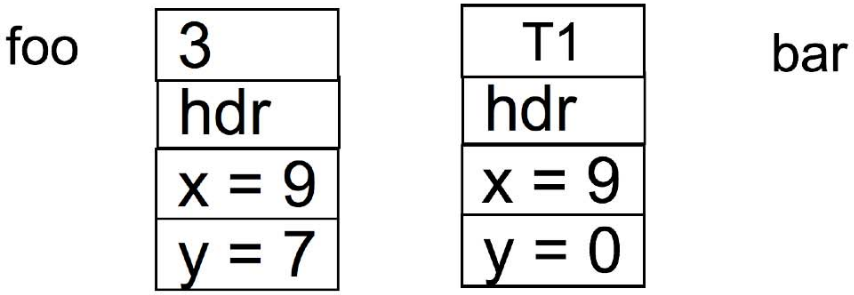
Reads <foo, 3>

T2

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

Reads <bar, 5>

STM Illustration



foo

3
hdr
x = 9
y = 7

bar

T1
hdr
x = 9
y = 0

T1

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

Reads <foo, 3>

Writes <bar, 5>

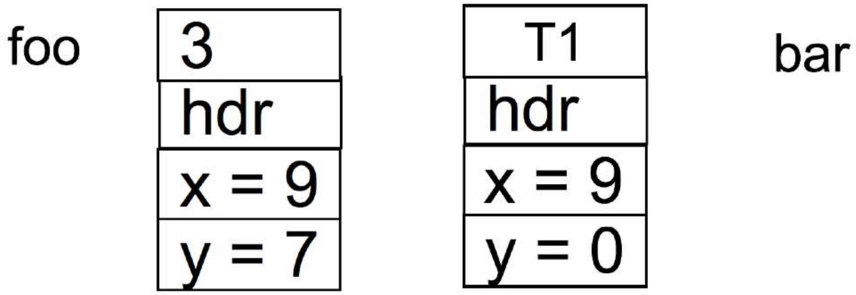
Undo <bar.x, 0>

T2

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

Reads <bar, 5>

STM Illustration



foo

3
hdr
x = 9
y = 7

T1
hdr
x = 9
y = 0

bar

T1

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

Reads <foo, 3>

Writes <bar, 5>

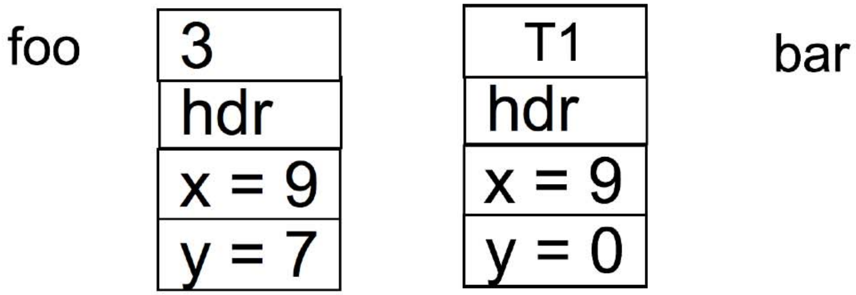
Undo <bar.x, 0>

T2

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

Reads <bar, 5>

STM Illustration



T1

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

Reads <foo, 3>

Writes <bar, 5>

Undo <bar.x, 0>

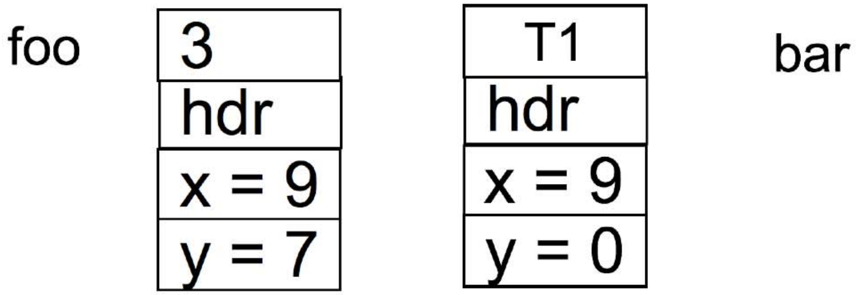
T2

T2 waits →

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

Reads <bar, 5>

STM Illustration



foo

3

hdr

x = 9

y = 7

T1

hdr

x = 9

y = 0

bar

T1

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

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

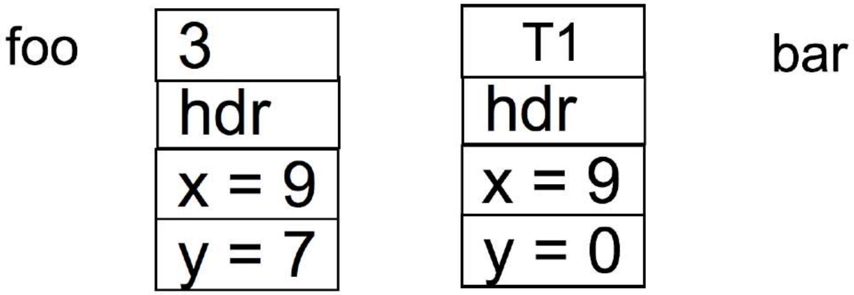
Undo <bar.x, 0>

T2

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

Reads <bar, 5>

STM Illustration



T1

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

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

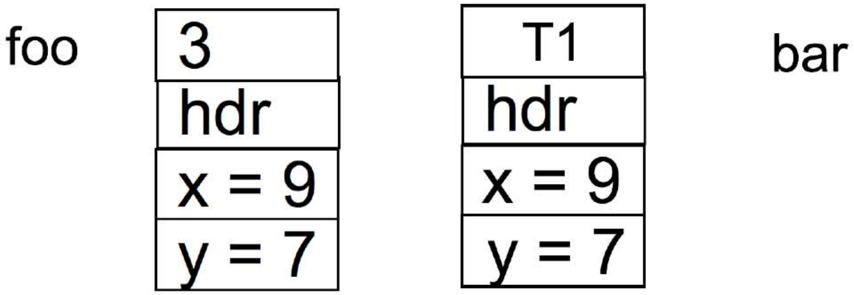
Undo <bar.x, 0>

T2

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

Reads <bar, 5>

STM Illustration



foo

3

hdr

x = 9

y = 7

T1

hdr

x = 9

y = 7

bar

T1

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

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

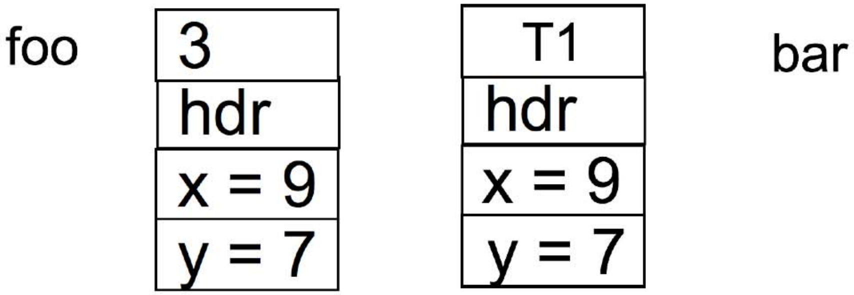
Undo <bar.x, 0> <bar.y, 0>

T2

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

Reads <bar, 5>

STM Illustration



foo

3

hdr

x = 9

y = 7

T1

hdr

x = 9

y = 7

bar

<u>T1</u>

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

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

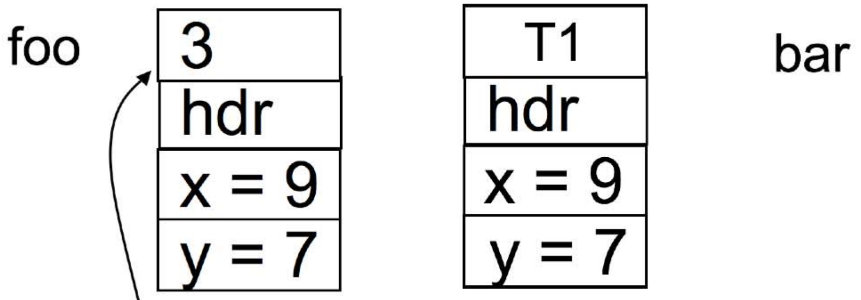
Undo <bar.x, 0> <bar.y, 0>

<u>T2</u>

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

Reads <bar, 5>

STM Illustration



foo

3

hdr

x = 9

y = 7

T1

hdr

x = 9

y = 7

bar

T1

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

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

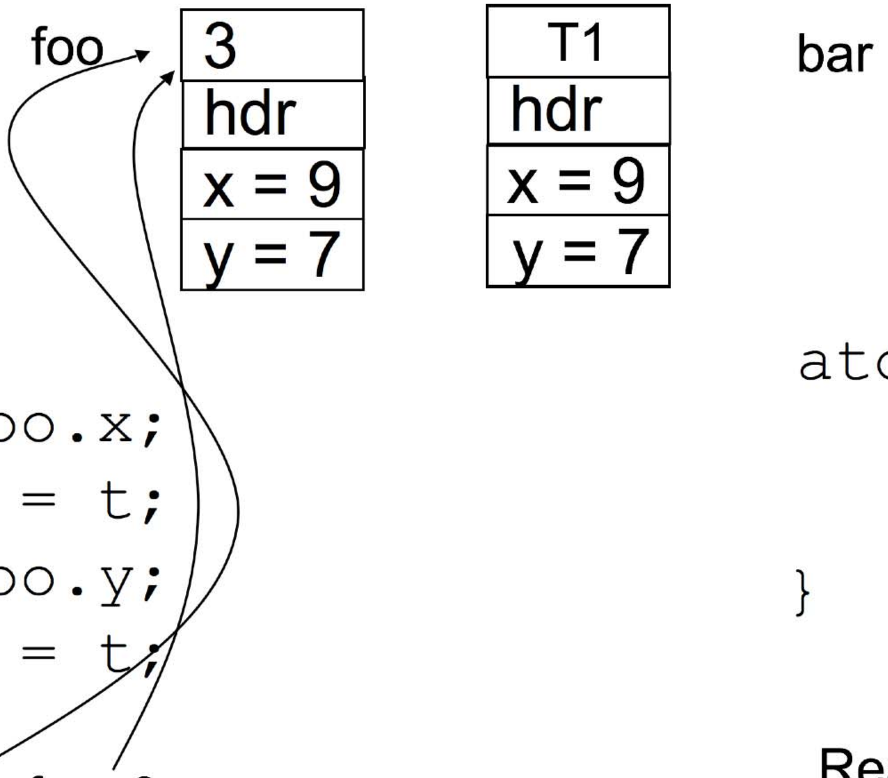
Undo <bar.x, 0> <bar.y, 0>

T2

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

Reads <bar, 5>

STM Illustration



foo

3
hdr
x = 9
y = 7

T1
hdr
x = 9
y = 7

bar

T1

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

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

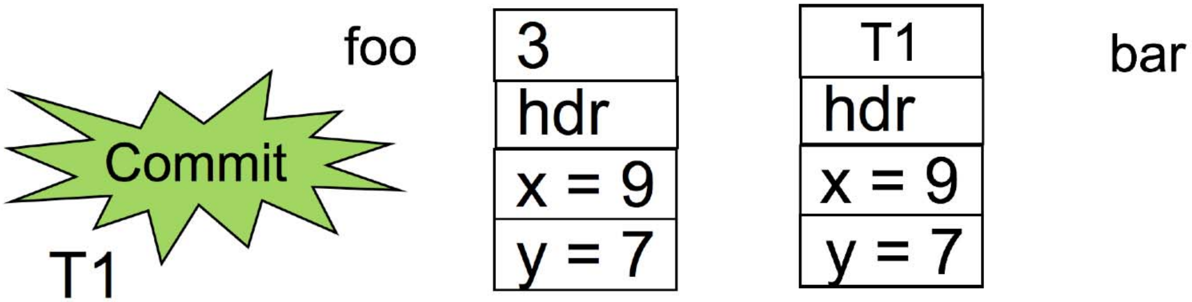
Undo <bar.x, 0> <bar.y, 0>

T2

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

Reads <bar, 5>

STM Illustration



Commit

foo

3

hdr

x = 9

y = 7

T1

hdr

x = 9

y = 7

bar

T1

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

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

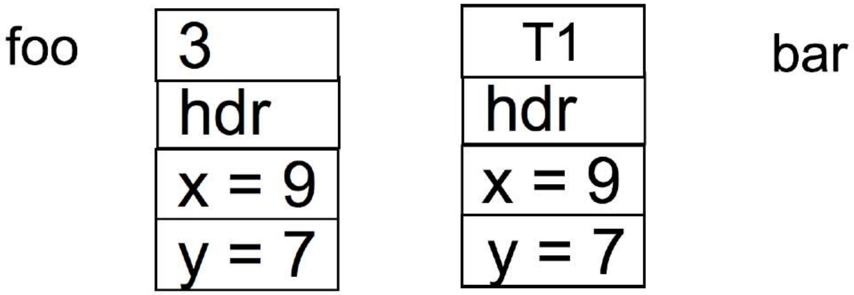
Undo <bar.x, 0> <bar.y, 0>

T2

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

Reads <bar, 5>

STM Illustration



T1

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

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

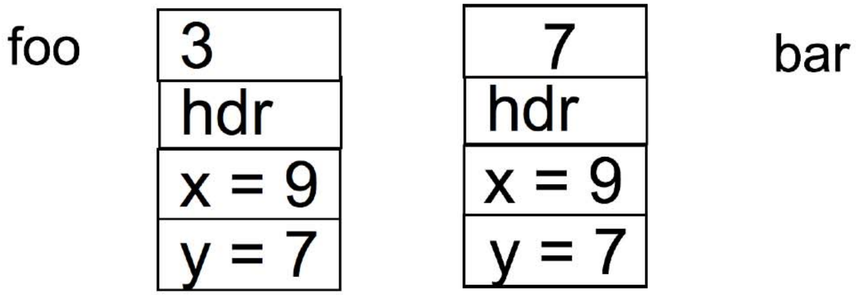
Undo <bar.x, 0> <bar.y, 0>

T2

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

Reads <bar, 5>

STM Illustration



foo

3
hdr
x = 9
y = 7

7
hdr
x = 9
y = 7

bar

T1

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

Reads <foo, 3> <foo, 3>

Writes

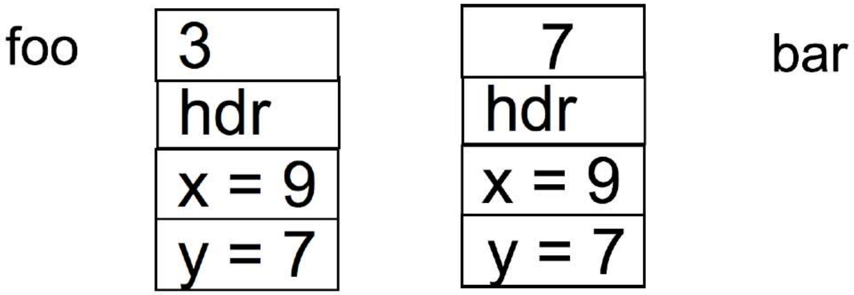
Undo <bar.x, 0> <bar.y, 0>

T2

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

Reads <bar, 5>

STM Illustration



foo

3
hdr
x = 9
y = 7

bar

7
hdr
x = 9
y = 7

T1

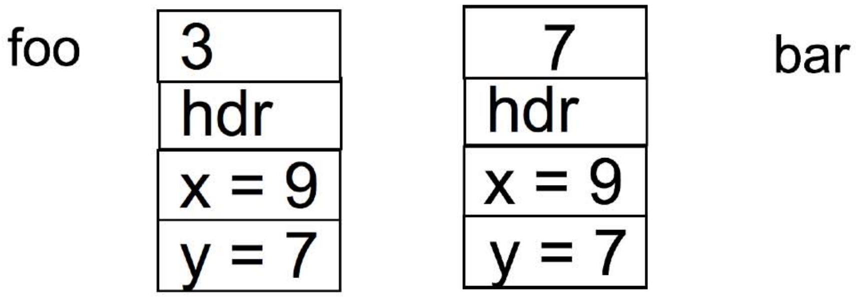
```
atomic {
   t = foo.x;
   bar.x = t;
   t = foo.y;
   bar.y = t;
}
```

T2

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

Reads <bar, 5>

STM Illustration



T1

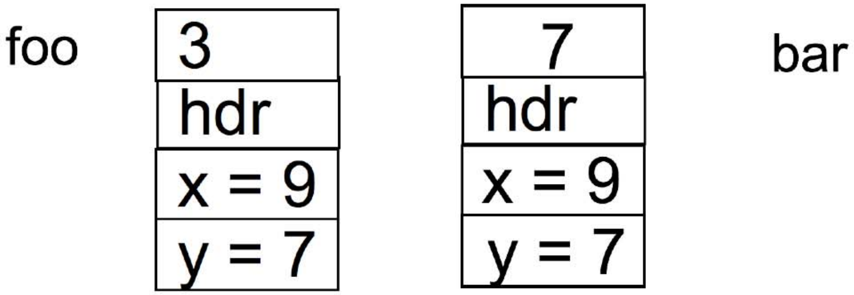
```
atomic {
    t = foo.x;
    bar.x = t;
    t = foo.y;
    bar.y = t;
}
```

T2

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

Reads <bar, 5> <bar, 7>

STM Illustration



T1

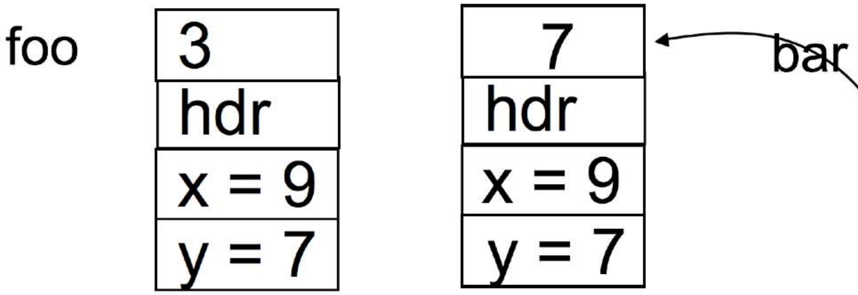
```
atomic {
   t = foo.x;
   bar.x = t;
   t = foo.y;
   bar.y = t;
}
```

T2

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

Reads <bar, 5> <bar, 7>

STM Illustration



T1

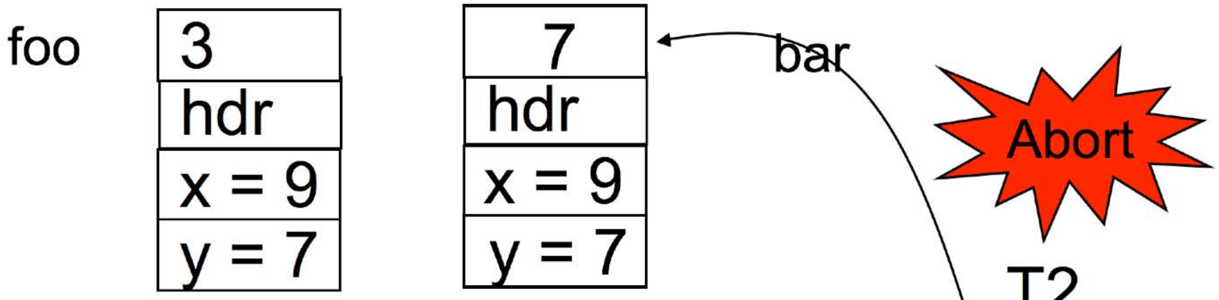
```
atomic {
   t = foo.x;
   bar.x = t;
   t = foo.y;
   bar.y = t;
}
```

T2

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

Reads <bar, 5> <bar, 7>

STM Illustration



foo

3
hdr
x = 9
y = 7

bar

7
hdr
x = 9
y = 7

T1

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

T2

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

Reads <bar, 5> <bar, 7>

Challenges for STM Systems

- Overhead of software barriers

- Function cloning

- Robust contention management

- Memory model (strong Vs. weak atomicity)
  - See comments in Lecture 1

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

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
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
}
```

- Decomposed barriers expose redundancies

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
txnOpenForRead(a)
if(a.z != 0) {

  txnLogObjectInt(&a.x, a)
  a.x = 0

  txnLogObjectInt(&a.z, a)
  a.z = t3
}
```

- Decomposed barriers expose redundancies

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) {

  txnLogObjectInt(&a.x, a)
  a.x = 0

  txnLogObjectInt(&a.z, a)
  a.z = t3
}
```

- Decomposed barriers expose redundancies

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
}
```

- Decomposed barriers expose redundancies

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
}
```

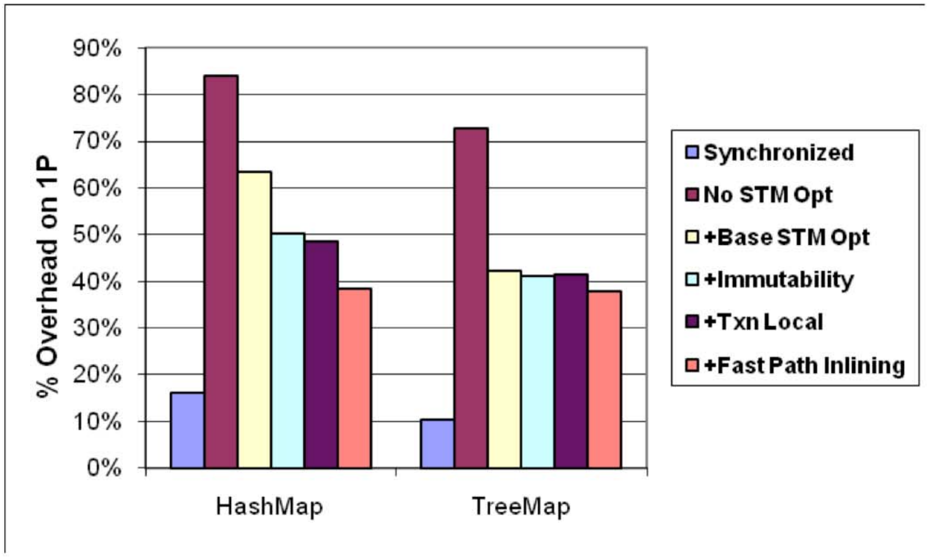
- Allows compiler to optimize STM code
- Produces fewer & cheaper STM operations

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

Robust Contention Management

- How to handle pathological contention cases without too much overhead for case of low contention?

- Two approaches for STM systems
  - Adjust STM algorithm
    - Switch between versioning & detection schemes
    - Adjust concurrency scale
  - Use proper contention management policy
    - Select conflict transactions to stall or abort
    - Select when transaction will restart

Example: Intel C++ STM Execution Modes

- Optimistic mode
  - Optimistic conflict detection for reads
  - Pessimistic 2-phase locking for writes
  - Quiescence for privatization safety
- Pessimistic mode
  - Pessimistic 2-phase locking for reads & writes
  - Can co-exist with optimistic transactions
- Obstinate mode
  - One pessimistic transaction with highest priority
  - Guaranteed not to fail
- Serial mode
  - One transaction at a time single global lock

Contention Management Policies for STM

- Thorough study by Scherer & Scott (PODC'05)
  - Nonetheless, still an active area of research
  - The following actions are takes by a requesting xaction that observes a conflict with an enemy xaction

- Policies
  - Polite: stall requestor with randomized backoff
    - After some retries, acquire highest priority
  - Karma: xaction priority = size of read & write set
    - Abort enemy if its priority is lower, otherwise stall request
    - Requestor aborted when its retries exceed difference in priorities
    - Priority not reset when xaction aborts
  - Eruption: Karma with priority boosting
    - Add the priority of a stalled xaction to that of the conflict transaction

Contention Management Policies (Cont)

- Policies (cont)
  - Kindergarten: take turns in object access
    - Hit-list of xactions that have stalled/aborted this one in the past
    - Hit-list determines if an xaction should stall or abort the enemy
  - Timestamp: age-based using timestamps
    - Older xaction wins conflicts
  - Published timestamp: avoids old zombie xactions
    - If conflicting xaction is too old, abort it
    - Double the threshold for "too old" on each restart
  - Polka: best of Karma and Polite
    - Karma priorities + randomized backoff interval

- How to evaluate CM policies
  - Measure throughput and fairness
  - Consider scalability
  - Consider wide range of workloads

Lecture 2: Select References

- Overview
  - Larus & Rajwar. Transactional Memory, Morgan & Claypool Publishers, 2007
  - Adl-Tabatabai. Unlocking Concurrency: Multi-core Programming with Transactional Memory, ACM Queue, 2006
  - Larus & Kozyrakis. Transactional Memory, CACM, 2008
- Software Transactional Memory
  - Shavit & Touitou. Software Transactional Memory, PODC. 1995
  - Herlihy et al. Software Transactional Memory for Dynamic-sized Data Structures, PODC, 2003
  - Marathe et al. Adaptive Software Transactional Memory. ISDC, 2005
  - Scherer & Scott. Advanced Contention Management for Dynamic Software Transactional Memory, PODC, 205
  - Shavit & Dice, What Really Makes Transactions Faster. Transact, 2006
  - Saha et al. Implementing a high performance software transactional memory. PPoPP 2006
  - Adl-Tabatabai et al, Compiler and runtime support for efficient software transactional memory. PLDI, 2006
  - Harris et al. Optimizing Memory Transactions. PLDI, 2006
  - Wang et. al. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. CGO, 2007.

Questions?

Transactional Memory

Concepts, Implementations, & Opportunities

Christos Kozyrakis

Pervasive Parallelism Lab

Stanford University

http://ppl.stanford.edu/~christos

Lecture 2 Summary

- 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
  - Design options
    - Static Vs dynamic, non-blocking Vs lock-based
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, …)
    - Transactional record per data (locked/version)

Lecture 2 Summary (cont)

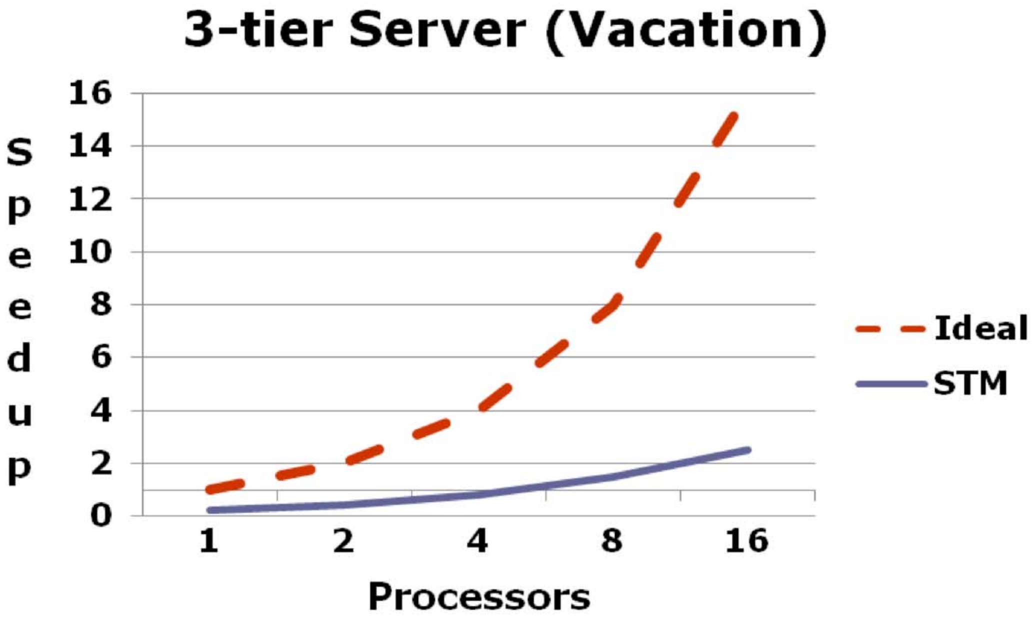
- Intel McRT STM
  - Eager versioning, optimistic reads, pessimistic writes
  - Read barriers check version number
  - Write barrier acquire locks
  - Commit validates the read-set and releases locks
  - Periodic validation needed to avoid doomed transactions

- Optimizations
  - Decomposed barriers to allow redundancy elimination
  - No barriers for private or transaction local data
  - Switch between STM algorithms
  - Contention management

Lecture 3:
Hardware Support for TM

- Outline
  - Hardware-accelerated STMs
    - Motivation
    - HASTM
    - SigTM
  - Hardware-based TM (HTM)
    - Basic HTM mechanism
    - Example HTM system
    - HTM challenges and opportunities

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

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

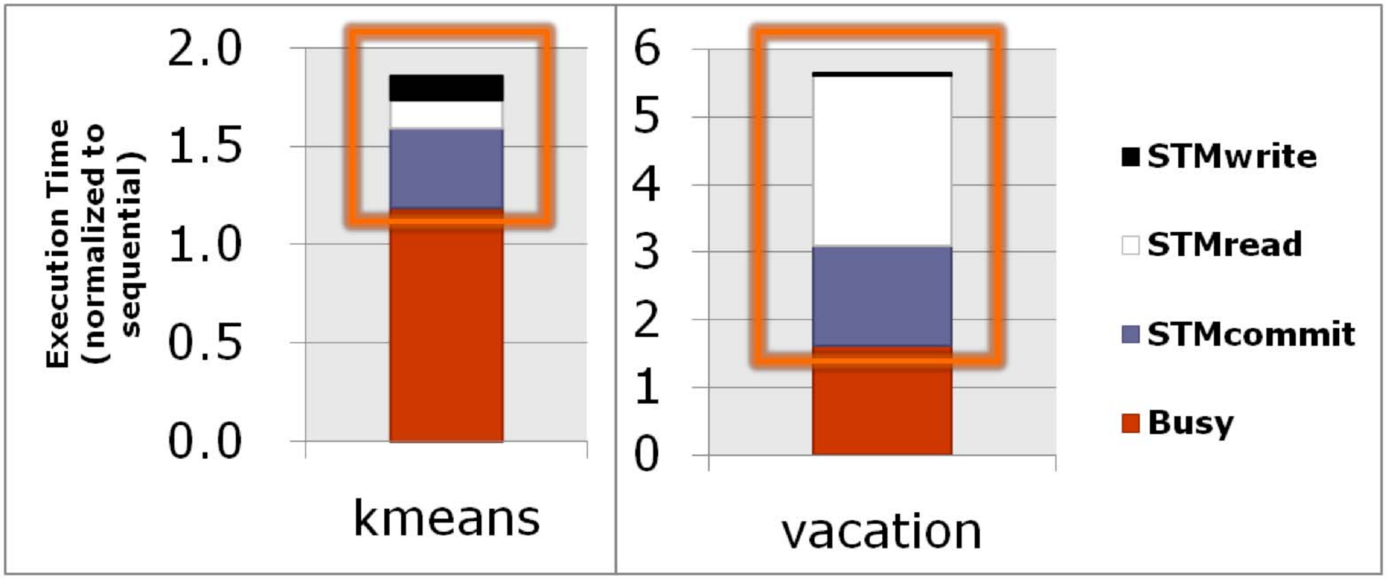
- Hardware-based TM systems (TCC, LTM, VTM, LogTM, …)
  - Versioning & conflict detection directly in HW

- 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

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 read

Hardware-accelerated STM (HASTM)

- Proposed by Intel in MICRO'06

- Hardware primitives
  - Per-thread mark bits at granularity of cache lines
  - Used to build fast filters to speedup read barriers

- Functionality exposed to SW
  - SW can set mark bit for an address
  - SW checks if mark bit was previously set
    - No other thread has touched line since marked
    - Supports conflict detection and barrier filtering
  - SW checks if other threads have written any marked lines
    - Implements fast validation

HASTM Hardware Implementation

- Extend each private cache line with mark bits
  - Mark bits set & read by software
  - Mark bit reset by HW on eviction or coherence action
  - HW instruction to query of any mark bits reset

- Potential extensions
  - Separate mark bits for read & write marking
  - Separate mark bits for nesting levels
  - Mark bits throughout memory hierarchy
    - Including main memory (encoded in ECC bits)
    - Helps support strong atomicity
    - UFO design in ISCA'08

HASTM Algorithm

- Assume the STM algorithm in Lecture 2

- HASTM read operation
  - Check if mark bit already set
  - If not set, mark bit and add to read set
    - Redundant barriers are filtered dynamically

- HASTM validation
  - Check if any mark bits were reset
  - If no, validation is complete
  - If yes, run software validation (slow)
    - To separate between capacity evictions & true conflicts

HASTM System Issues

- Insufficient cache capacity
  - Mark bits are only an acceleration mechanism
  - Cache evictions cause mark bits to be lost
    - HASTM reverts to (slow) software validation
  - Mark bits can be sized just for common case

- Interrupts, context switches, page faults, …
  - Mark bits are lost
    - HASTM reverts to slow software validations
  - When xaction resumes, mark bits provide some help
    - Filtering of redundant read barriers…

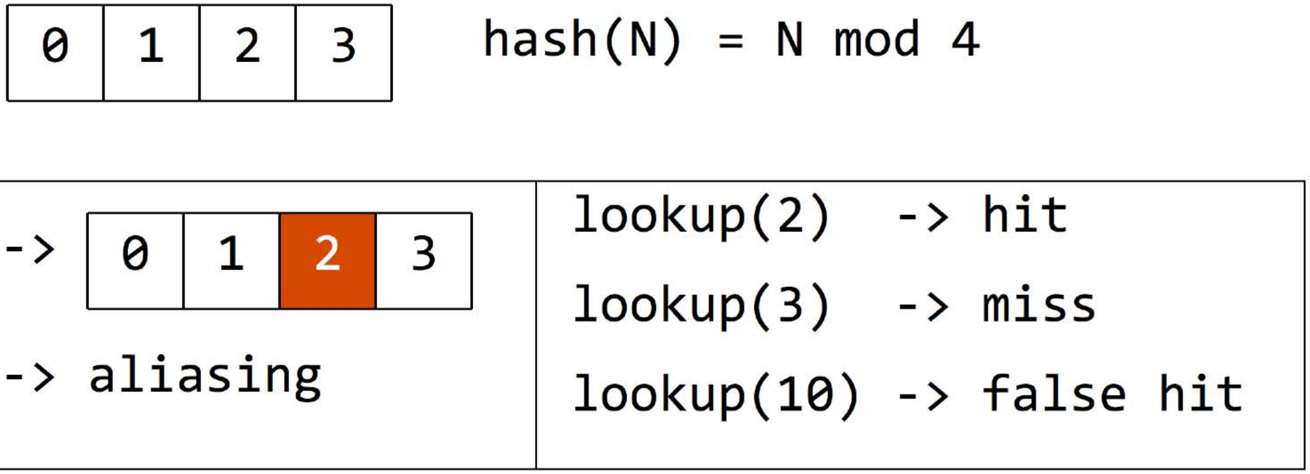
SigTM Motivation

- Accelerate STM at low hardware cost
  - Similar to goals Intel HASTM
- Do not modify caches
  - Complex interactions with coherence, prefetching, etc…
  - Place all TM acceleration in isolated unit
- Provide strong atomicity
  - Enable conflict detection between transactional and non-transactional accesses
  - Without limited by cache capacity and without adding metadata throughout memory hierarchy

SigTM Hardware

- Each HW thread has 2 HW signatures (read & write)
  - Each signature implemented by a Bloom filter
    - Fixed-size bit array with set of hash functions
  - No other HW modifications (e.g., no extra cache bits)

- Operations on signature (Bloom filter): insert & lookup



0	1	2	3

`hash(N) = N mod 4`

```
insert(2) -> 0 1 2 3

insert(6) -> aliasing
```

```
lookup(2)  -> hit

lookup(3)  -> miss

lookup(10) -> false hit
```

SigTM Hardware (cont)

- How SigTM uses its signatures:
  - Tx read/write →insert address into read/write signature
  - Coherence messages →look up address in signature
    - Enabled/disabled by software
- If lookup hits in signature, either:
  - Trigger SW abort handler (conflict detection)
  - NACK remote request (atomicity & isolation enforcement)
- Signatures may generate false conflicts
  - Performance but not correctness issue
  - Reduce with longer signatures & better hash functions
- With this HW, how does the SW change?

SigTMread

```
SigTMread(addr) {
  read_sig_insert(addr); // 1 instruction
  return *addr;
}
```

- No need to build SW read-set
  - Replaced by read signature
- Read signature provides continuous validation
  - Snoops coherence messages & any hits cause abort
  - Hits due to writes by non-transactional code as well
- Write barriers are similar
  - No write-set, but need versioning code

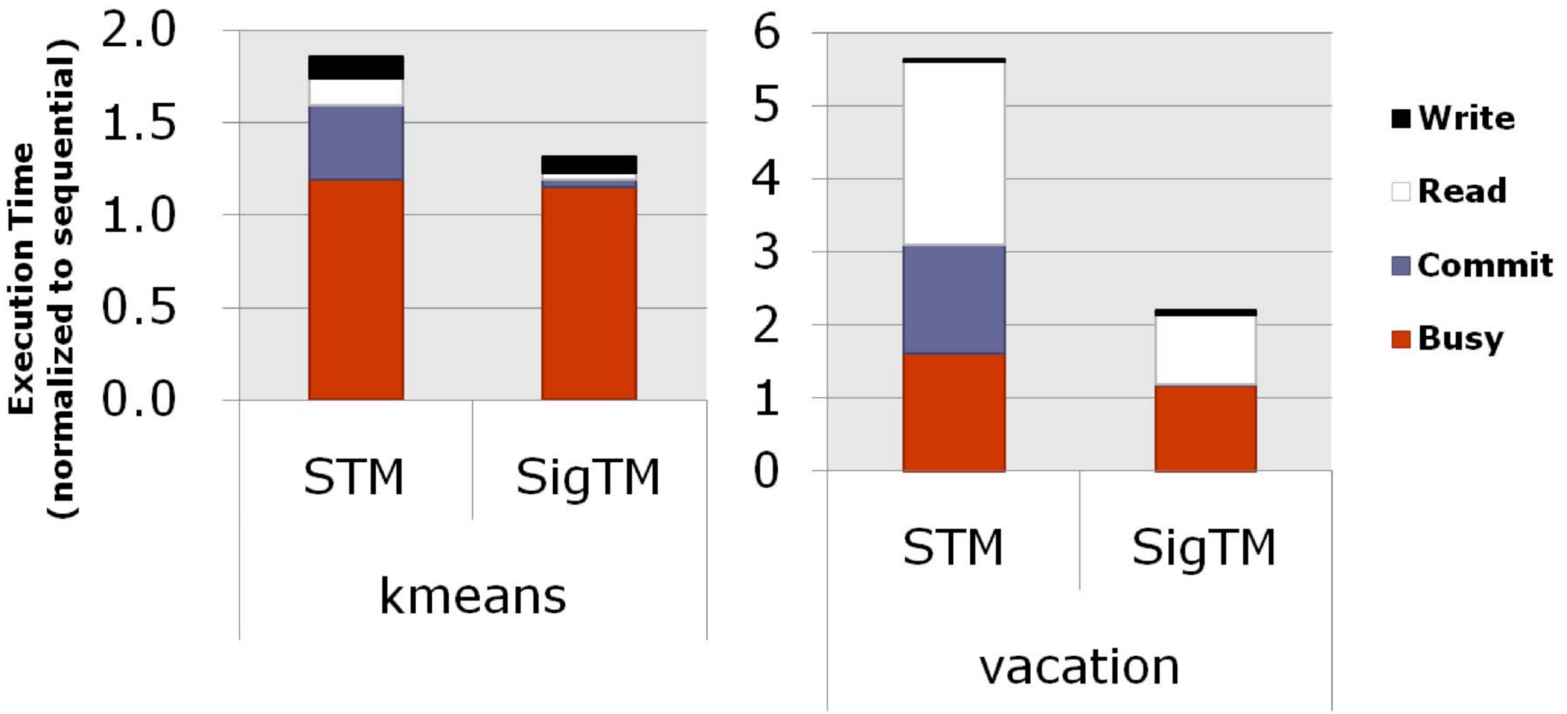
SigTMcommit

```
SigTMcommit() {
  read_sig_reset();
  disable_read_sig_lookup();
  write_sig_reset();
  disable_write_sig_lookup();
}
```

- Read signature eliminates need to validate read-set
  - Snoops coherence messages and reports conflicts
- Write signature eliminates locks
  - Snoops coherence messages and report
- Abort is more complex but also accelerated by SigTM
  - Write signature used to ensure undo atomicity

SigTM Overhead

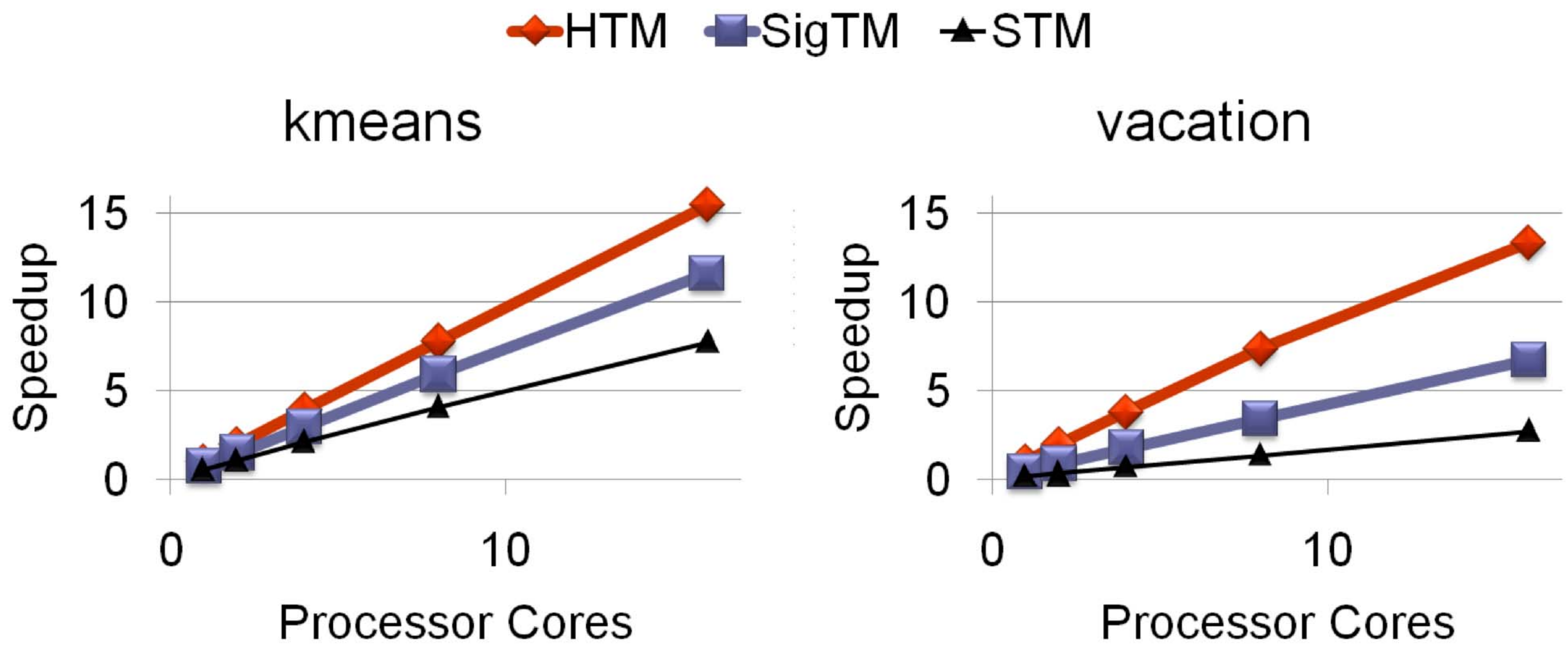
- Measured single-thread performance on STM and SigTM



- SigTM effectively accelerates read & commit

SigTM Scaling

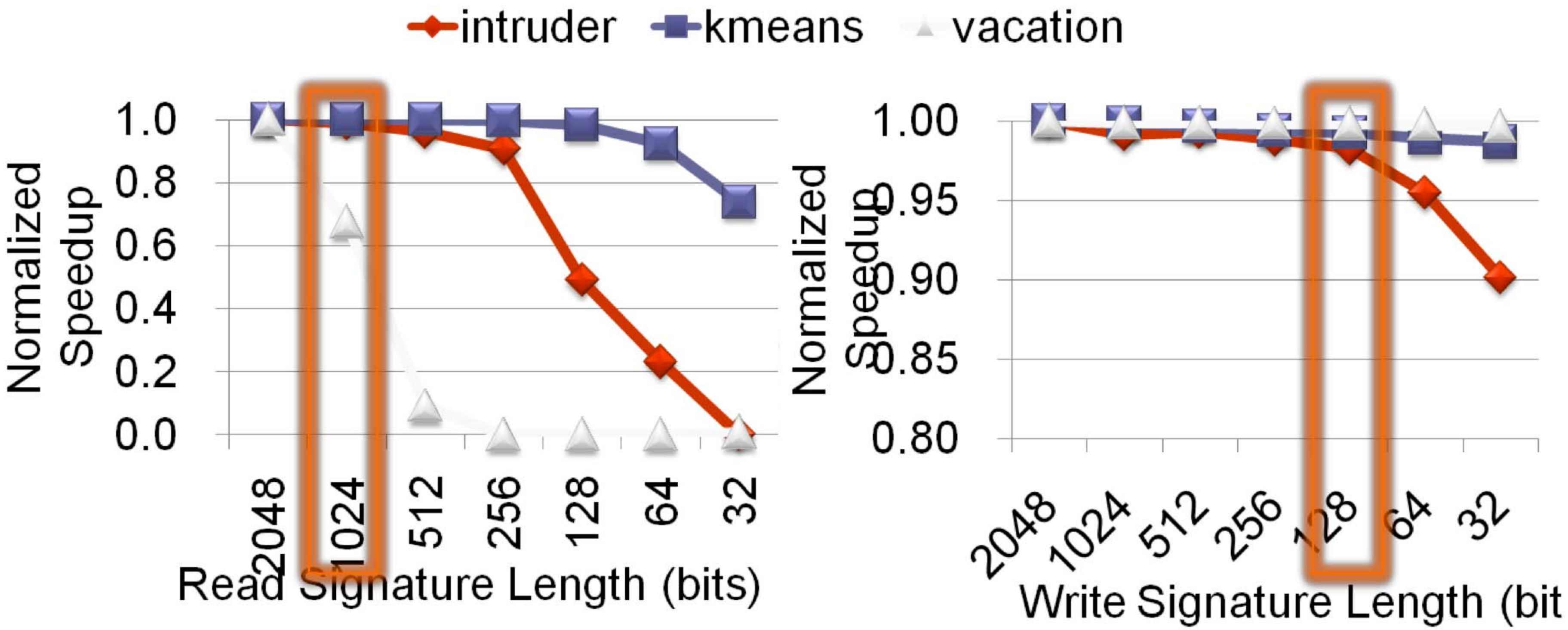
- Measured speedup on 1–16 cores



- SigTM faster than STM but slower than full HW system
  - Roughly a 2x gap between design points

How Much Hardware Does it Cost?

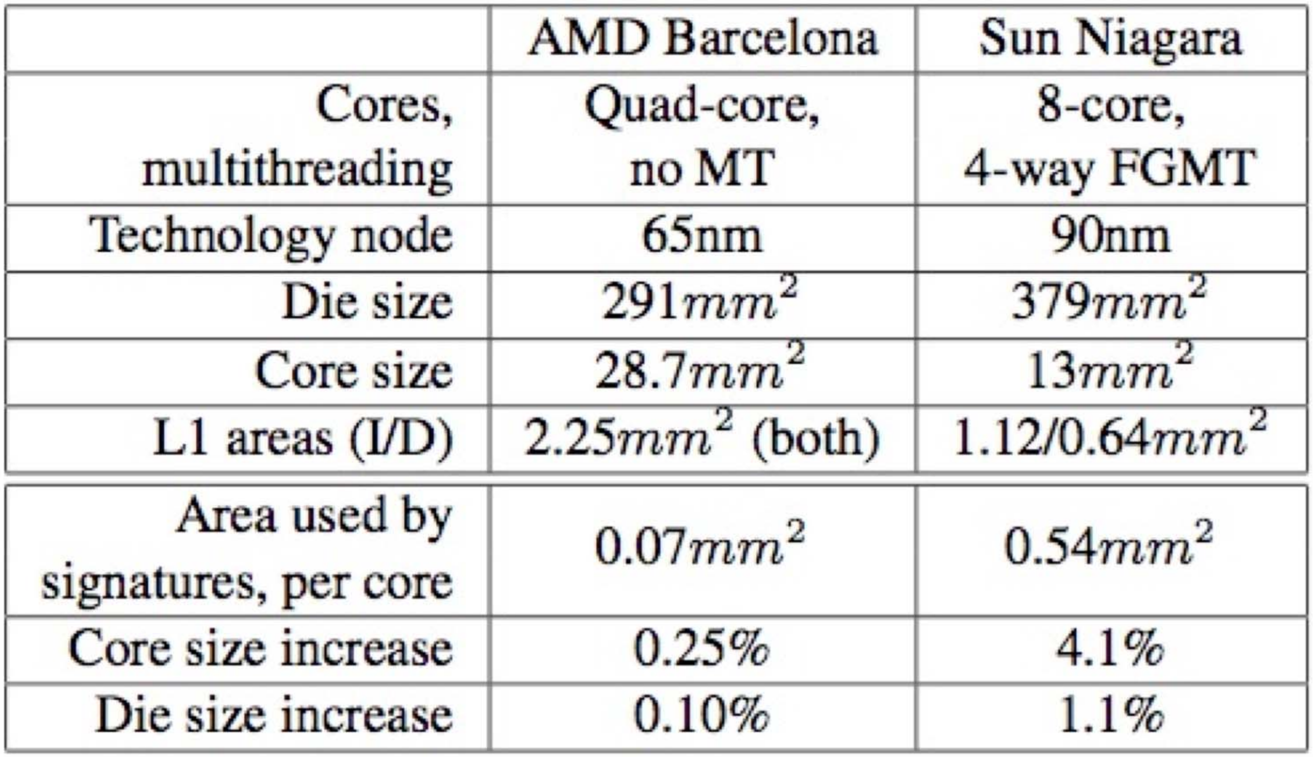
- Measured performance drop as signatures get shorter



- Recommend 1024 bits for read sig, 128 bits for write sig

Signature HW Cost

- [Sanchez 07]



	AMD Barcelona	Sun Niagara
Cores, multithreading	Quad-core, no MT	8-core, 4-way FGMT
Technology node	65nm	90nm
Die size	$291mm^2$	$379mm^2$
Core size	$28.7mm^2$	$13mm^2$
L1 areas (I/D)	$2.25mm^2$ (both)	$1.12/0.64mm^2$
Area used by signatures, per core	$0.07mm^2$	$0.54mm^2$
Core size increase	0.25%	4.1%
Die size increase	0.10%	1.1%

Table 4: Area estimates in real systems

HASTM Vs. SigTM

- Similarities
  - Acceleration for STM with cost-effective HW
- Differences
  - HASTM bits limited to cache capacity
  - SigTM signatures can cause false conflicts
  - Signatures are compact & can manipulate in SW
    - E.g., save and restore on nested xaction boundaries
  - Signatures are bound to physical addresses
    - Invalidated by paging events
  - Signatures can provide strong atomicity
    - Through continuous lookups of coherence events
    - HASTM requires metadata across memory hierarchy

Questions?

Hardware TM Summary

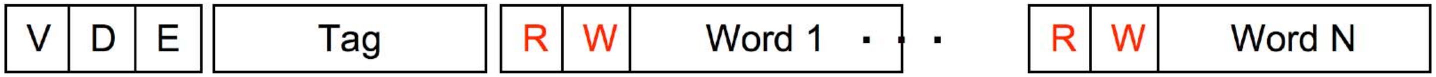
- Data versioning in caches
  - Cache the write-buffer or the undo-log
  - Cache metadata to track read-set and write-set
  - Can do with private, shared, and multi-level caches

Hardware TM Summary

- Data versioning in caches
  - Cache the write-buffer or the undo-log
  - Cache 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 & directory coherence
- Notes
  - Register checkpoint must be taken at transaction begin
  - Virtualization of hardware resources discussed later
  - HTM support similar for TLS and speculative lock-elision
    - Some hardware can support all three models actually

HTM Design

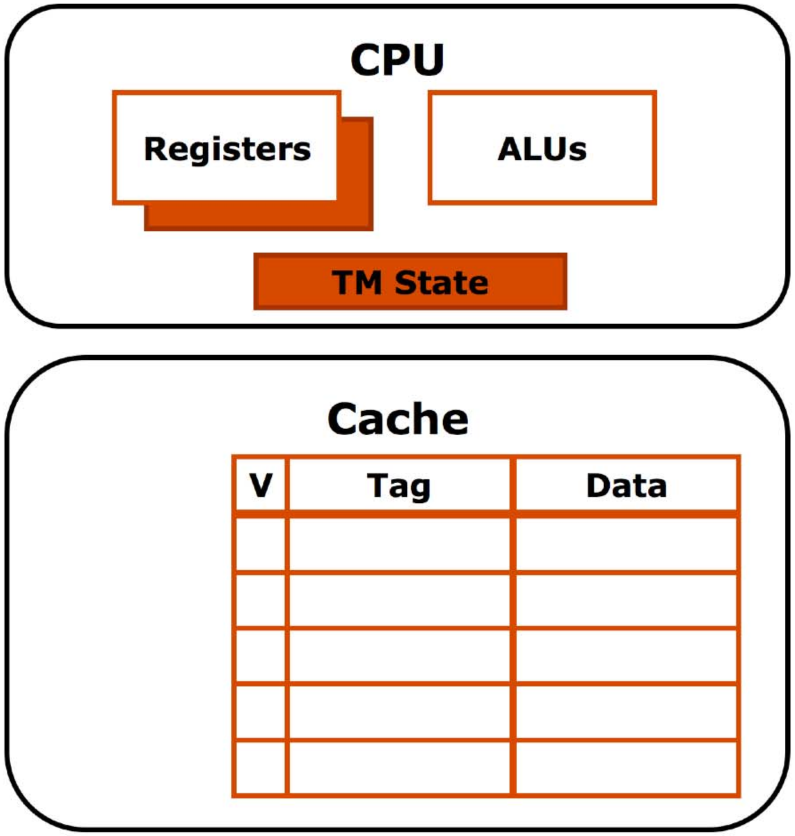
- Cache lines annotated to track read-set & write set
  - R bit: indicates data read by transaction; set on loads
  - W bit: indicates data written by transaction; set on stores
    - R/W bits can be at word or cache-line granularity
  - R/W bits gang-cleared on transaction commit or abort
  - For eager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



V D E | Tag | R W Word 1 . . . R W Word N

- 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

Registers

ALUs

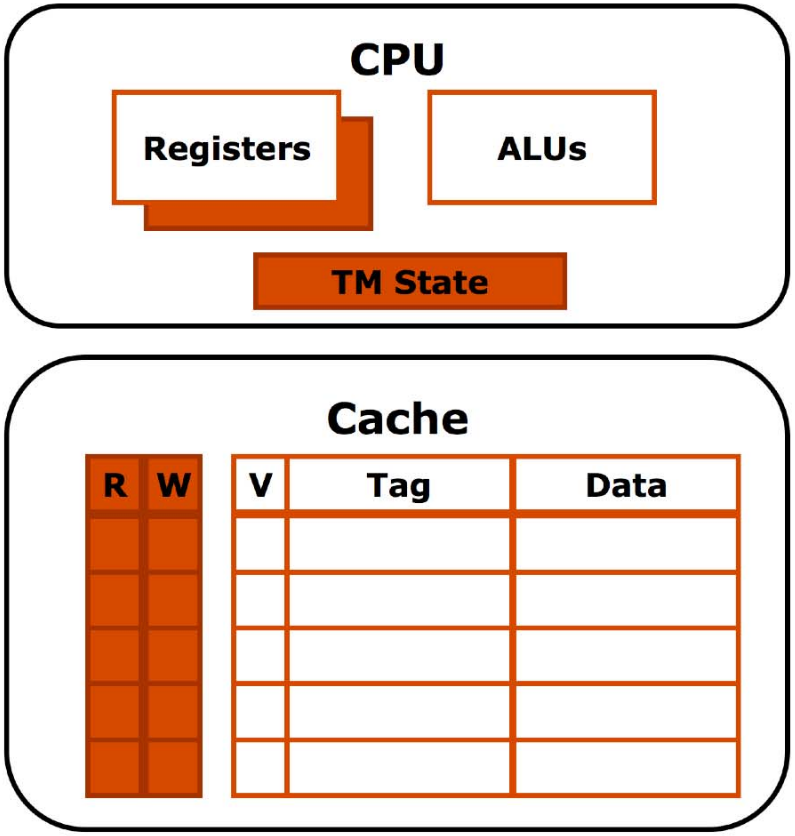
TM State

Cache

V	Tag	Data

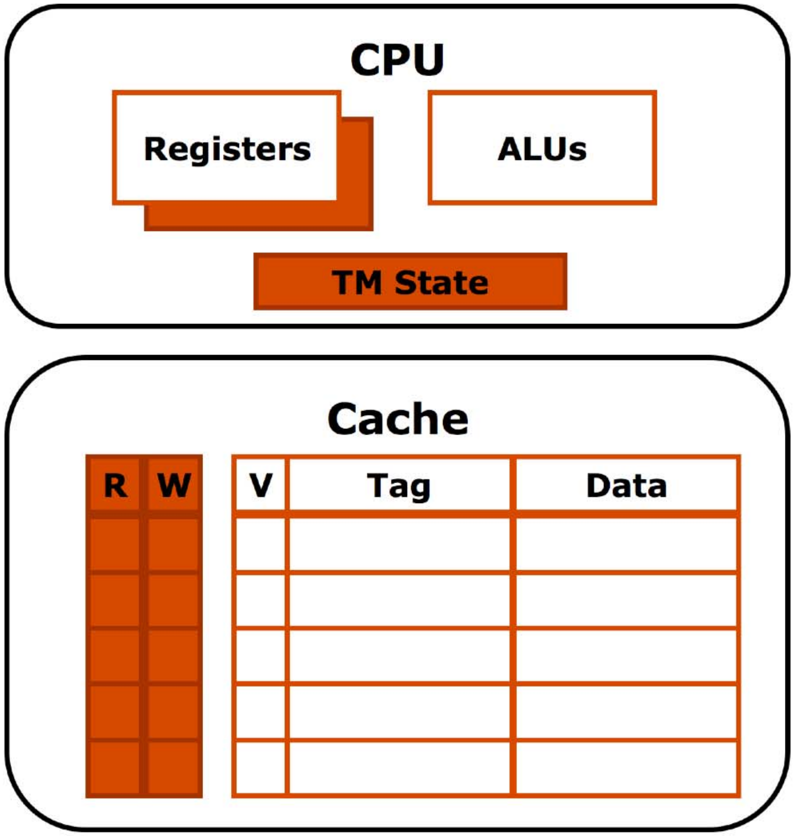
- CPU changes
  - Register checkpoint (available in many CPUs)
  - TM state registers (status, pointers to handlers, …)

Example HTM: Lazy Optimistic



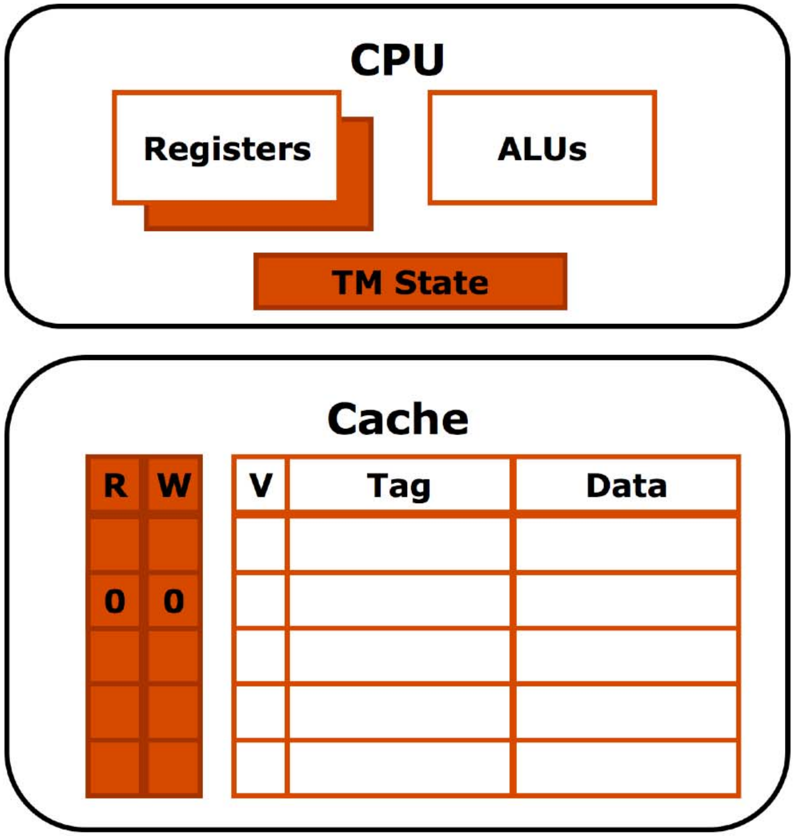
- Cache changes
  - R bit indicates membership to read-set
  - W bit indicates membership to write-set

HTM Transaction Execution



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

HTM Transaction Execution

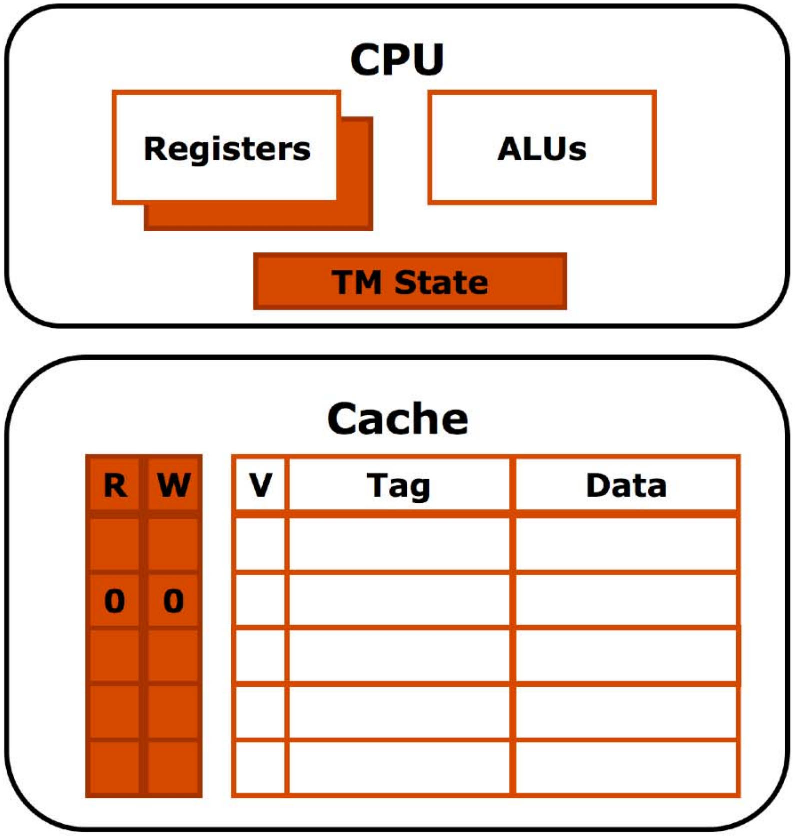


R	W	V	Tag	Data
0	0			

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

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

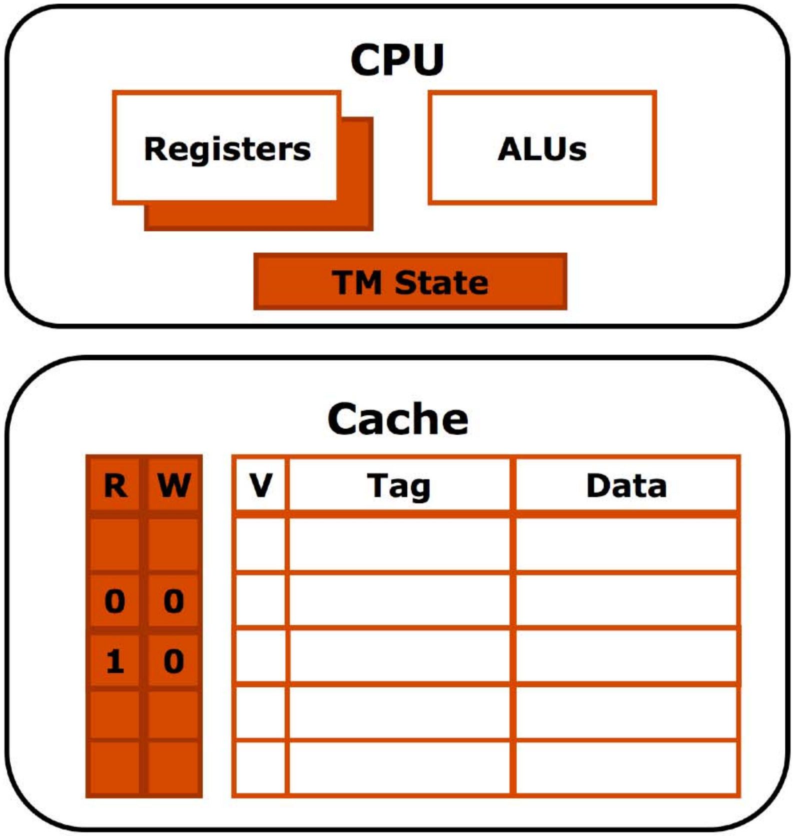
HTM Transaction Execution



R	W	V	Tag	Data
0	0			

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

HTM Transaction Execution

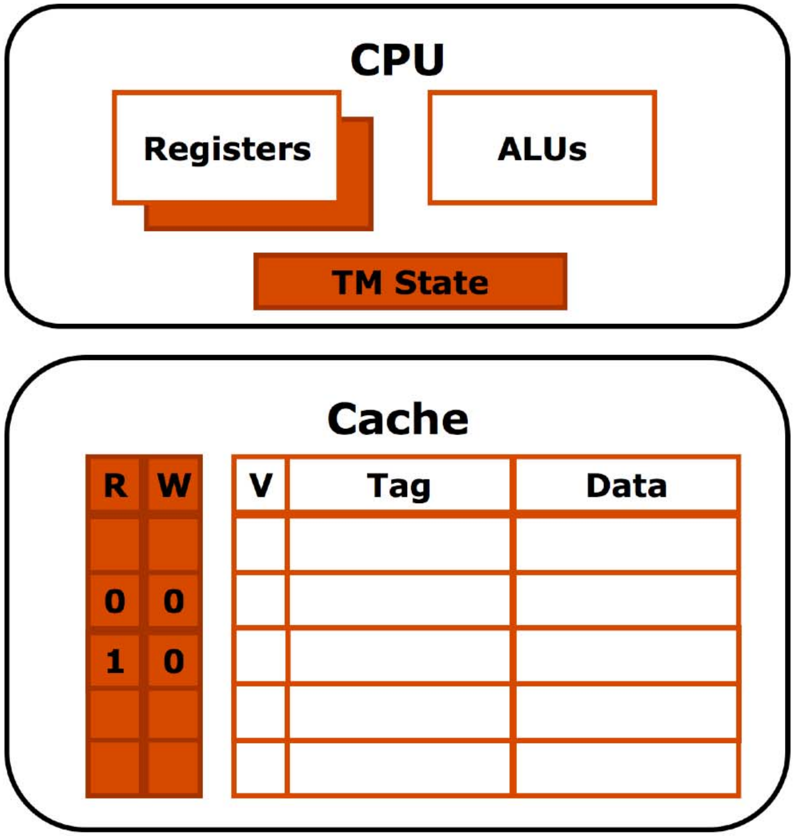


R	W	V	Tag	Data
0	0			
1	0			

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

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

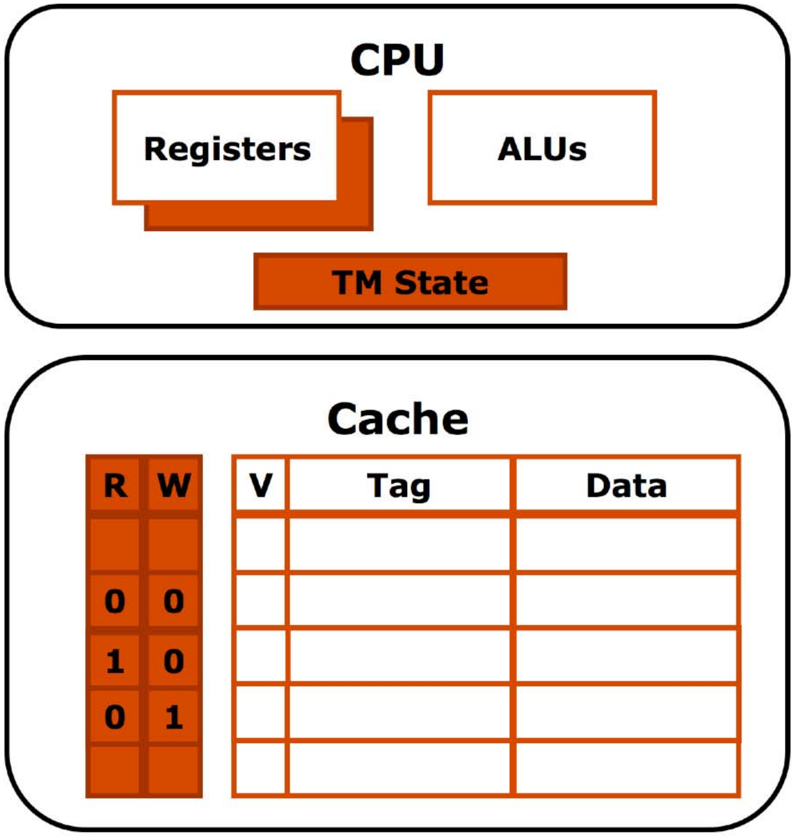
HTM Transaction Execution



R	W	V	Tag	Data
0	0			
1	0			

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

HTM Transaction Execution

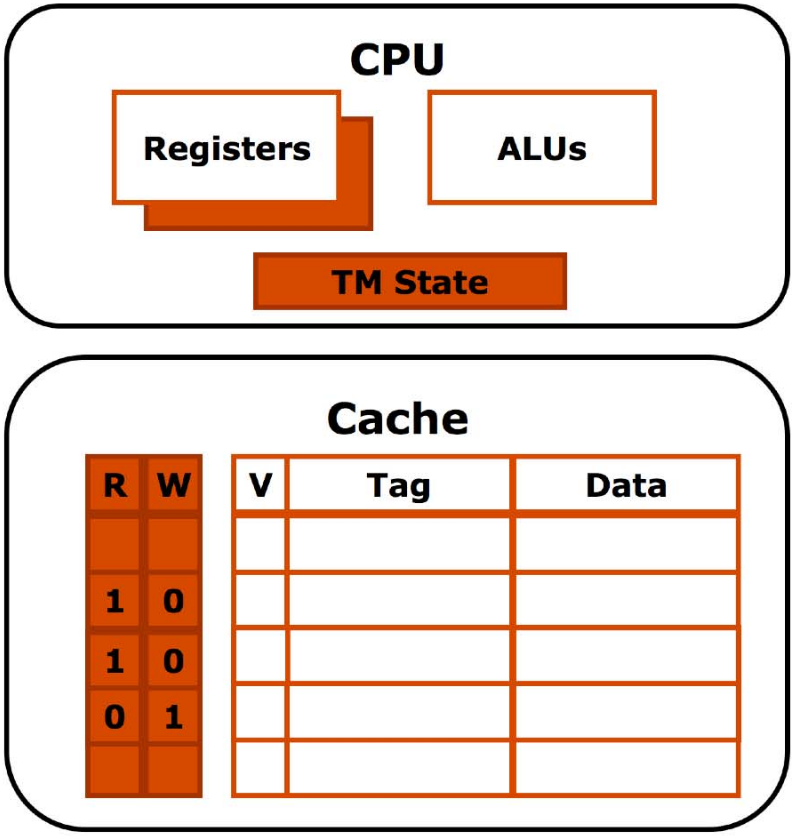


R	W	V	Tag	Data
0	0			
1	0			
0	1			

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

- Store operation
  - Serve cache miss if needed (eXclusive if not shared, Shared otherwise)
  - Mark data as part of write-set

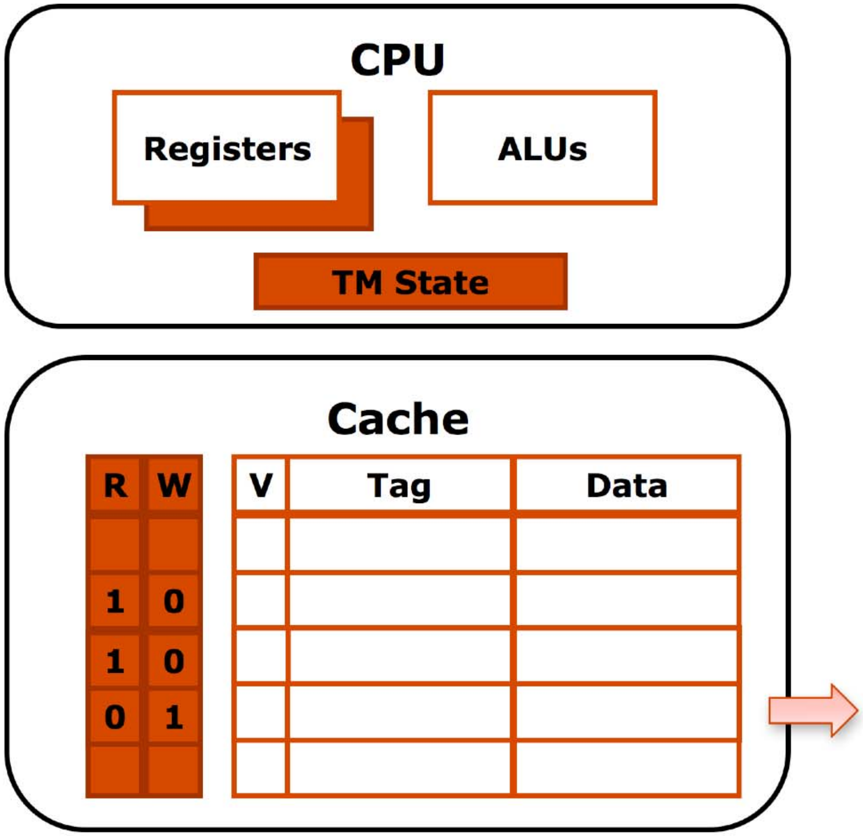
HTM Transaction Execution



R	W	V	Tag	Data
1	0			
1	0			
0	1			

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

HTM Transaction Execution



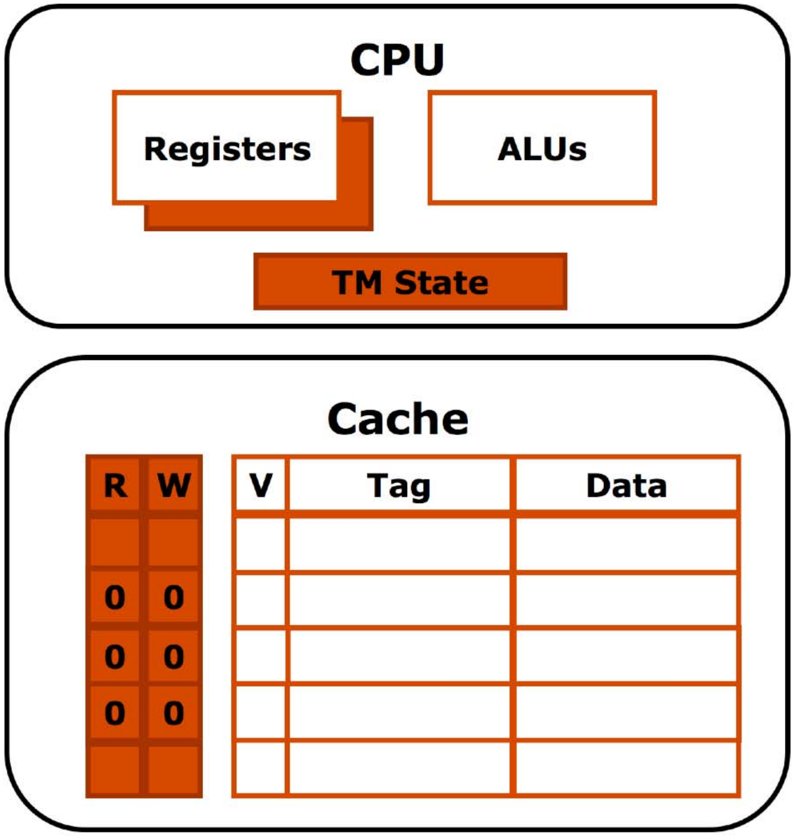
R	W	V	Tag	Data
1	0			
1	0			
0	1			

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

upgradeX B

- Fast, 2-phase commit
    - Validate: request exclusive access to write-set lines (if needed)

HTM Transaction Execution

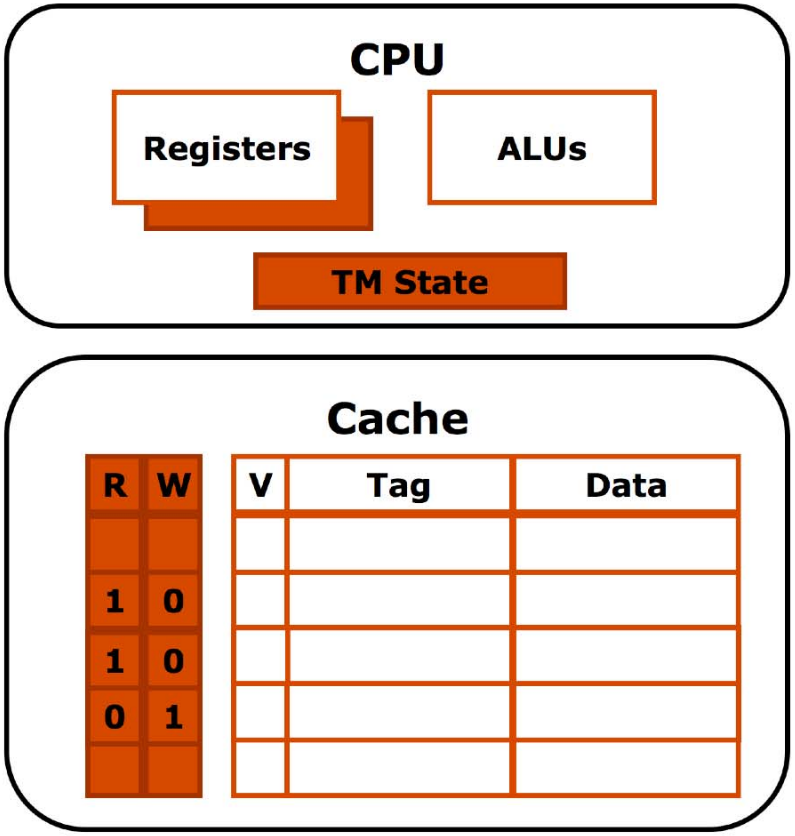


R	W	V	Tag	Data
0	0			
0	0			
0	0			

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

- Fast, 2-phase commit
  - Validate: request exclusive access to write-set lines (if needed)
  - Commit: gang-reset R & W bits, turns write-set data to valid (dirty) data

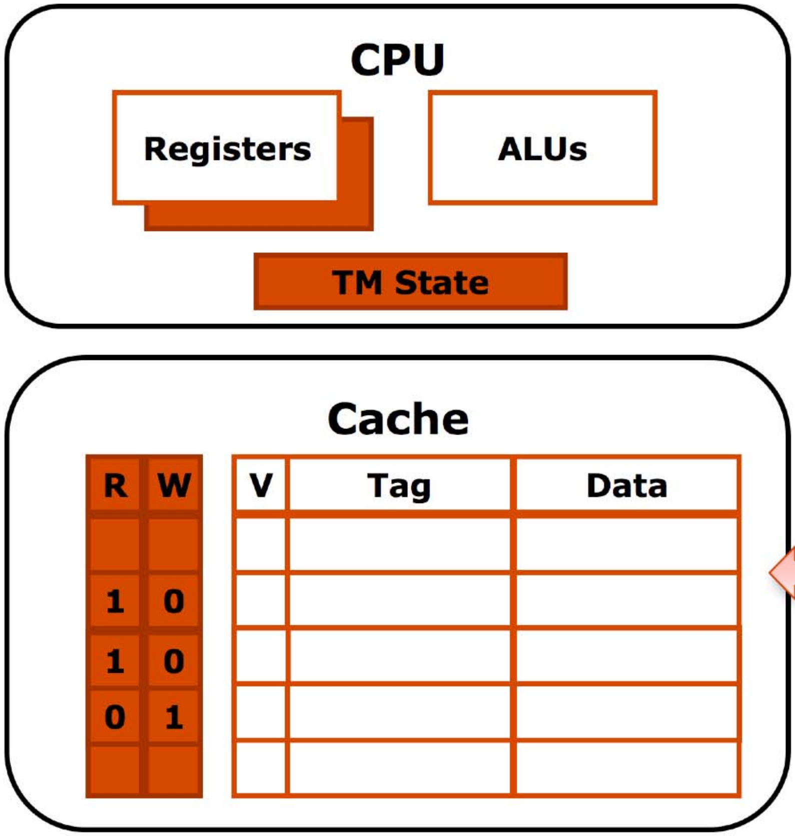
HTM Conflict Detection



R	W	V	Tag	Data
1	0			
1	0			
0	1			

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

HTM Conflict Detection



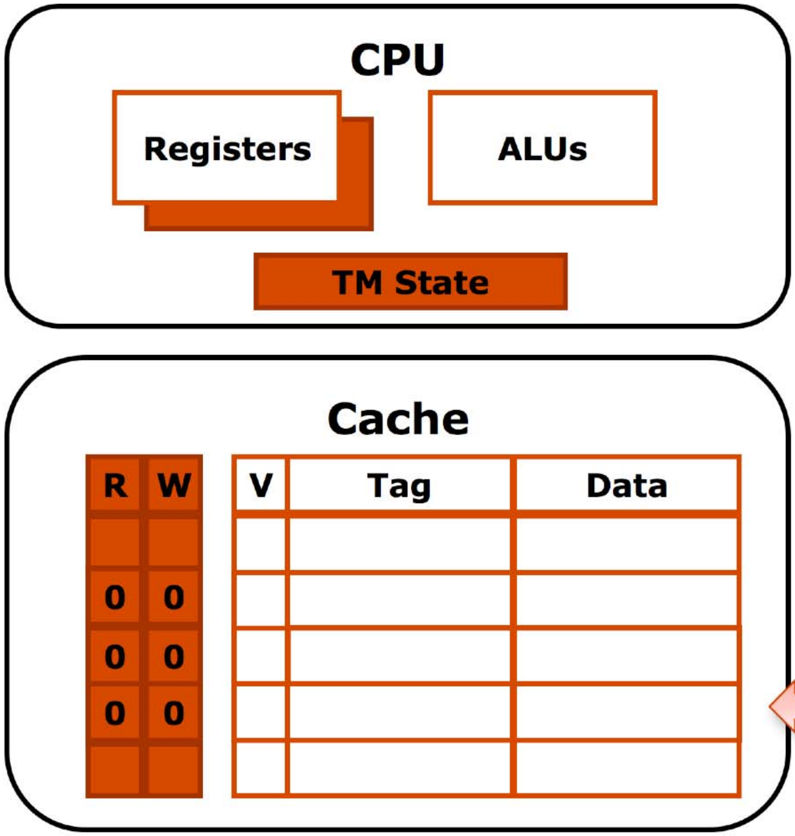
R	W	V	Tag	Data
1	0			
1	0			
0	1			

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

upgradeX D ☑

- Fast conflict detection & abort
  - Check: lookup exclusive requests in the read-set and write-set

HTM Conflict Detection



R	W	V	Tag	Data
0	0			
0	0			
0	0			

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

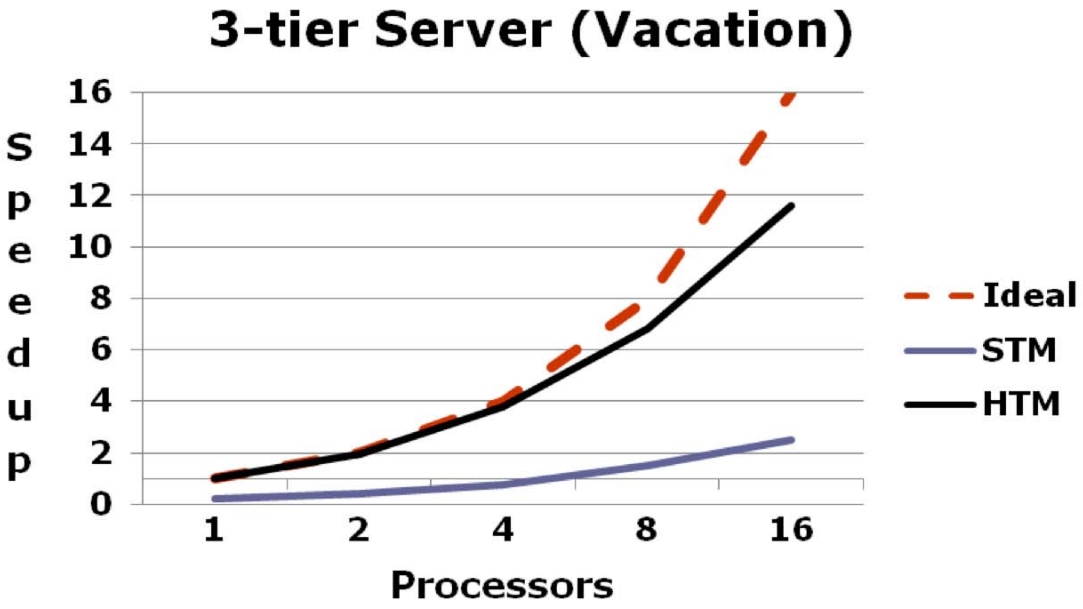
upgradeX A ☒

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

HTM Advantages

- Transparent
    - No need for SW barriers, function cloning, DBT, …
- Fast common case behavior
    - Zero-overhead tracking of read-set & write-set
    - Zero-overhead versioning
    - Fast commit & abort without data movement
    - Continuous validation of read-set
- Strong isolation
    - Conflicts detected on non-xaction loads/stores as well
- Can simplify multi-core hardware [ISCA'04, Ceze'07]
    - Replace existing coherence with transactional coherence

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

HTM Challenges and Opportunities

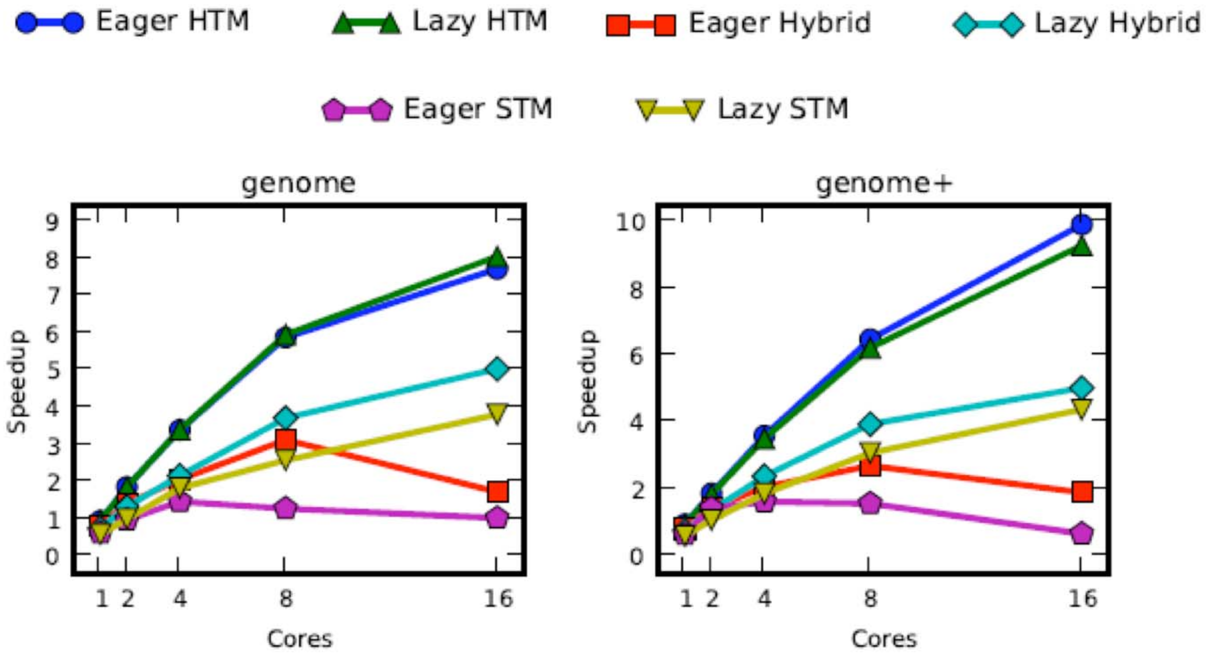
- Performance pathologies
  - How to handle problematic contention caches?
- Virtualization of hardware resources
  - What happens when HW resources are exhausted?
- HW/SW interface
  - How does HTM support flexible SW environments?

HTM Performance Pathologies

- Pathologies: contention cases that cause bottlenecks
  - Understanding the cause is important in addressing the issue
  - Enumerated by Bobba et al. in ISCA’07

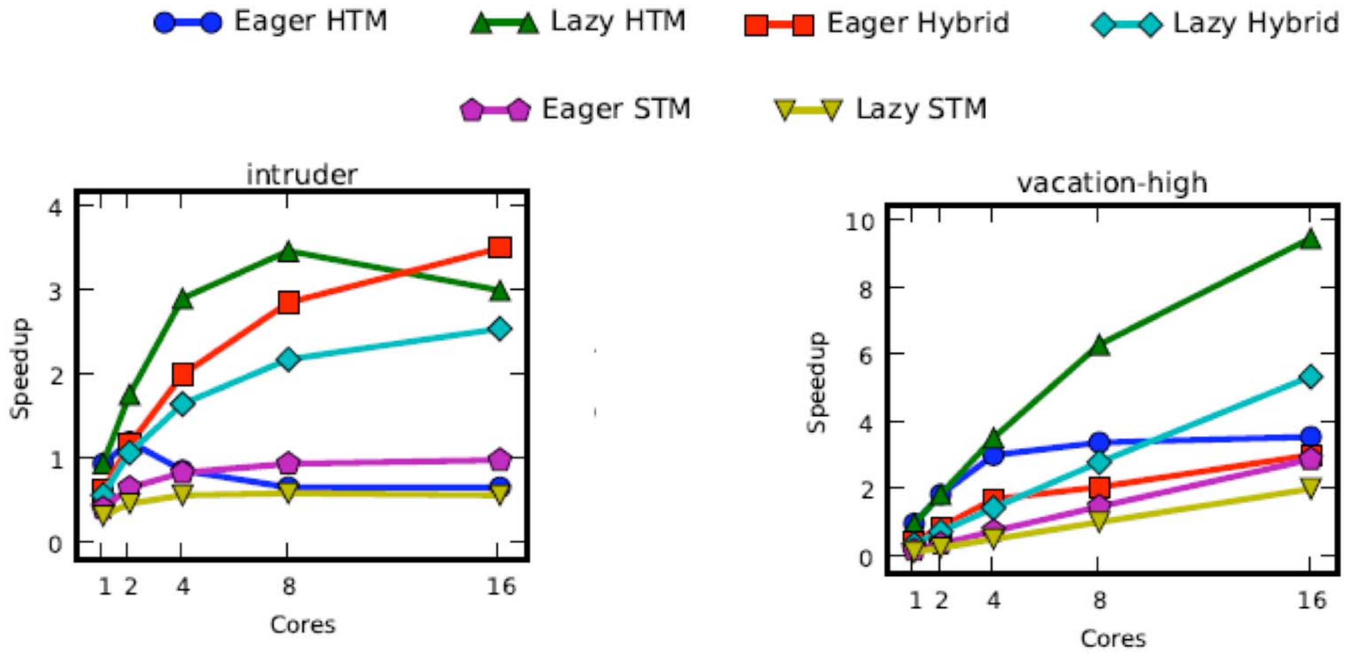
- Optimistic conflict detection
  - Default policy: committing xaction wins
    - Guarantees forward progress for the overall system
  - Pathologies: starving elder, restart convoy
- Pessimistic conflict detection
  - Default policy: requesting xaction wins OR requesting xaction stalls
    - No guarantees of forward progress
    - Need some way to detect deadlocks (conservative or accurate)
  - Pathologies: friendly fire, futile stall, starving writer, dueling upgrades

Do Pathologies Matter?



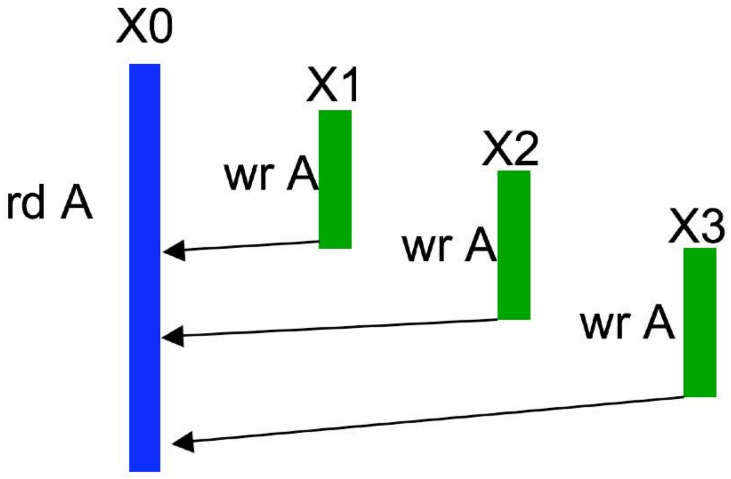
- In many cases, not at all
  - Low contention scenarios
  - All HW schemes perform similarly

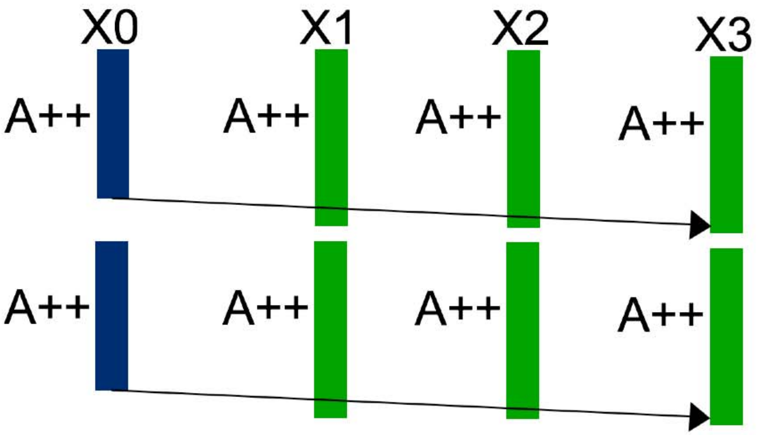
Do Pathologies Matter?



- In other cases, it matters a lot
  - HTMs slow down to STM/hybrid levels
  - The exact case & system matters

Pathologies for Optimistic Conflict Detection





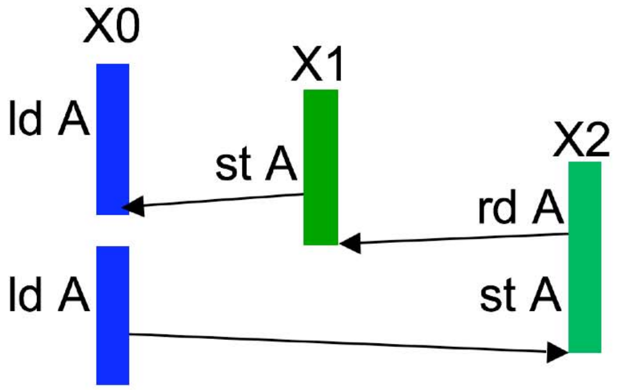
Starving elder

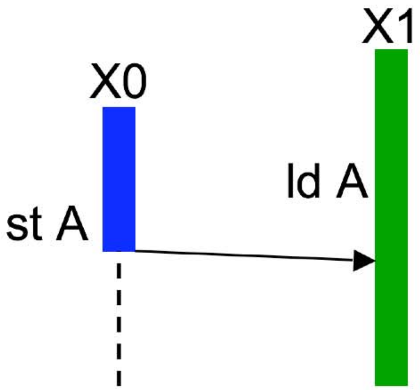
- Problem: long xaction aborted by small xactions
- Fix: after some retries, prioritize long xaction

Restart convoy

- Problem: one xaction aborts many dependent xactions
- Fix: restart after randomized (linear) backoff

Pathologies for Pessimistic Conflict Detection





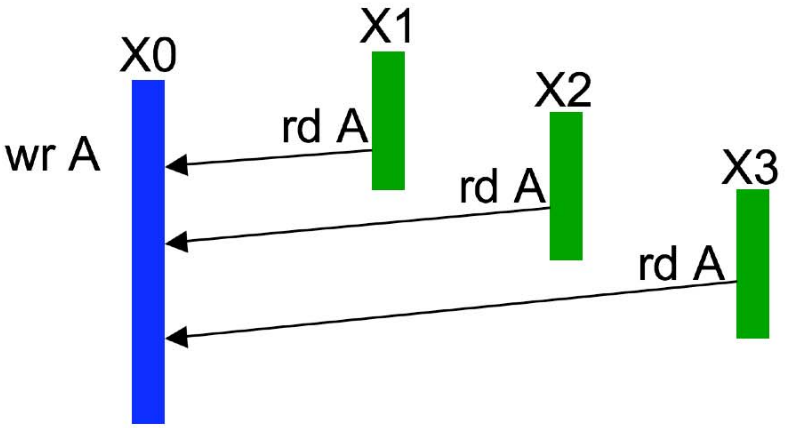
Friendly Fire

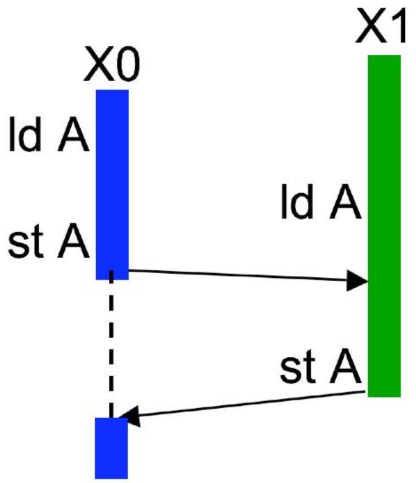
- Problem: livelock if requesting xaction wins conflict
- Fix: age-based conflict handling (using timestamps)

Futile Stall

- Problem: stall due to xaction that later aborts
- Fix: ?

Pathologies for Pessimistic Conflict Detection (cont)





Starving Writer

- Problem: stall/abort writer due to frequent reader
- Fix: prioritize writers over readers based on-age

Dueling upgrades

- Problem: stalls due to concurrent read-mod-writes
- Fix: Detect read-mod-writes and prioritize their reads

Discussion on HTM Pathologies

- Pathologies for optimistic detection
  - Easy to fix with a single policy
  - Restart after randomized backoff
  - After N retries, use priority mechanism
- Pathologies for pessimistic detection
  - Difficult to handle all in robust manner
  - Complex and sometimes conflicting fixes
- In general, optimistic detection has been shown to be more robust to contention scenarios
  - For both HW and SW TM system

HTM Virtualization

- Time virtualization ➔ What if time quanta expires?
  - Interrupts, paging, and context switch within xaction
  - What happens to the state in caches?

- Space virtualization ➔ What if caches overflow?
  - Where is the write-buffer or log stored?
  - How are R & W bits stored and checked?

- Observations: most transactions are currently small
  - Small read-sets & write-sets
  - Short in terms of instructions
  - No guarantees that this trend will continue
    - Programmer sloppiness Vs. conflicts

Time Virtualization

- Idea: rethink interrupt processing/assignment for multicore

- Three-tier interrupt handling for low overhead
  1. Defer interrupt until next short transaction commits
     - Use that processor for interrupt handling
  2. If interrupt is critical, rollback youngest transaction
     - Most likely, the re-execution cost is very low
  3. If a transaction is repeatedly rolled back due to interrupts
     - Use space virtualization to swap out (typically higher overhead)
     - Only needed when most threads run very long transactions (rare)

- Key assumption
  - Rolling back a short xaction cheaper than virtualizing it
  - Eliminates most of the complexity of time virtualization

Space Virtualization: Hybrid TM Schemes

- Idea: combine HTM + STM (Intel HyTM, Sun PhTM, …)
  - HW provides best-effort acceleration
  - SW provides virtualization in difficult cases
  - (Likely) the TM implementation for the Sun Rock processor
- Operation
  - Start transaction in HTM mode
  - On cache overflow or interrupt, switch to STM mode
- Challenges
  - Interactions between HTM and STM transactions
    - Must detect conflicts correctly
  - Contention management policies
    - How frequently to switch to STM?
    - Switch a single or all xactions to STM?
  - Providing strong atomicity
    - Weakest model of the two sets the semantics

Space Virtualization: Complete Schemes

- Key idea: map TM metadata structures to virtual memory
  - VM is practically unbounded
  - HTM resources act as a fast cache for metadata structure
- Virtualizing data-versioning
  - Eager: undo-logs need no special handling
    - Per-thread logs can be mapped to VM directly
    - Caches capture the working-set of undo-logs naturally
    - Cost: extra cache pressure and traffic
  - Lazy: write-buffers require special handling
    - Option 1: unified overflow structure in VM (hash-table)
    - Option 2: per-thread overflow structure in VM
    - Option 3: virtualize write-buffers using per-thread log
    - Challenge: knowing when to access the overflow structures

Space Virtualization: Complete Schemes (cont)

- Virtualizing conflict detection
  - Handling of read-set and write-set metadata
  - Option 1: use signatures for overflown metadata
    - Very simple but provides probabilistic conflict detection
    - Can be problematic in the presence of paging
  - Option 2: pervasive metadata across memory hierarchy
    - Store metadata everywhere, including DRAM
    - Expensive but eliminates overflow issue
  - Option 3: read-set and write-set metadata in VM
    - Shared or per-thread structures
    - Accurate conflict detection
    - Use signatures to filter accesses to metadata in VM

Space Virtualization: Example Implementations

- Intel VTM
  - Maps write-buffer and TM metadata to virtual memory
  - HW and firmware used to handle misses, relocation
  - Cache line granularity, signatures to reduce VM lookups
- Stanford XTM
  - Uses OS virtualization capabilities
  - On overflow, switch to a page-based TM system
  - No HW/firmware needed, transparent to SW, page-based granularity
- UCSD PTM
  - Similar to XTM but hardware manages overflow metadata in VM
  - Requires HW caches at memory controller but maintains fine granularity
- Wisconsin LogTM-SE
  - Undo-log mapped in virtual memory to begin with
  - Metadata virtualization using signatures

Lecture 3: Select References

Overview

- Adl-Tabatabai. Unlocking Concurrency: Multi-core Programming with Transactional Memory, ACM Queue, 2006
- Larus & Kozyrakis. Transactional Memory, CACM, 2008

Hardware-accelerated Transactional Memory

- Saha, et. al. Architecture Support for Software Transactional Memory Micro, 2006
- Minh et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees, ISCA, 2007
- Baugh et. Al. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory, ISCA, 2008

Hardware Transactional Memory

- Herlihy and Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures, ISCA, 1993
- Hammond, et al. Transactional Memory Coherence and Consistency, ISCA, 2004
- Rajwar et al. Virtualizing Transactional Memory. ISCA, 2005
- McDonald et al. Characterization of TCC on Chip-Multiprocessors. PACT 2005, 2005
- Moore et al. LogTM: Log-Based Transactional Memory. HPCA, 2006
- Kumar et al. Hybrid Transactional Memory, PPoPP, 2006

Lecture 3: Select References

Hardware Transactional Memory (cont)

- Chung et al. The Common Case Transactional Behavior of Multithreaded Programs, HPCA, 2006
- Chung et al. Tradeoffs in Transactional Memory Virtualization, ASPLOS, 2006
- Minh et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees, ISCA, 2007
- Chuang et al. Unbounded Page-Based Transactional Memory, ASPLOS, 2006
- Bobba et al. Performance Pathologies in Hardware Transactional Memory, ISCA, 2007
- Ceze et al. BulkSC: Bulk Enforcement of Sequential Consistency, ISCA, 2007
- Sanchez et al. Implementing Signatures for Transactional Memory, MICRO, 2007

Questions?

Transactional Memory

Concepts, Implementations, & Opportunities

Christos Kozyrakis

Pervasive Parallelism Lab

Stanford University

http://ppl.stanford.edu/~christos

Lecture 3 Summary

- STM performance
  - 2x to 8x per thread slowdown due to instrumentation
  - Most time spent on read barriers & validation

- Hardware accelerated TM
  - Conflict detection in HW; data versioning in SW
  - HASTM: per cache-line mark bits
    - Used for filtering & acceleration
    - Fall back to SW when mark cache lines evicted
  - SigTM: per-thread signatures
    - Conservative tracking of read-set & write-set
    - Continuous conflict detection, strong isolation

Lecture 3 Summary (cont)

- Hardware TM
  - Cache to store undo-log or write-buffer
  - Per cache-line R/W bits for read/write set tracking
  - Conflict detection on coherence events

- HTM challenges
  - Contention pathologies
    - Need robust contention management policy
    - Optimistic HTM systems
      - Randomized back off + prioritize after N retries
  - Virtualization of HW resources
    - Time and space virtualization

Lecture 4:
Hardware Support for TM

- Outline
  - Hardware-based TM (cont)
    - HW/SW interface
    - Example uses (brief)
  - Application examples (new)
    - STAMP benchmarks
    - Use of transactions & basic statistics
  - TM uses beyond concurrency control (brief)
    - Motivation and challenges
    - Example uses

Motivation for Rich HTM Interface

- HTM thus far has a simple SW interface
  - Instructions to define start/end of transaction
- How does SW control an HTM?
  - How does HTM interact with library-based SW?
  - How do we handle I/O & system calls within xactions?
  - How do we handle exceptions & contention within xaction?
  - How do we support novel TM programming constructs?
    - Retry, orelse, …
  - How do we support uses beyond concurrency control?
- Need an expressive ISA for HTM systems

A Flexible HW/SW Interface for HTM

- Features for flexible HTM interface
  1. Architecturally visible 2-phase commit
  2. Support for transactional handlers
  3. Support for nested transactions
  4. Instructions for private or idempotent accesses

- Implementation notes
  - HW: metadata support for nested transactions
    - Need HW support and virtualization
  - SW: xaction begin/end similar to function call/return
  - SW: xaction handlers similar to user-level exceptions
    - Virtually all complexity in software

Two-phase Transaction Commit

- Conventional: monolithic commit in one step
  - Finalize validation (no conflicts)
  - Atomically commit the transaction write-set
- New: two-phase commit process
  - `xvalidate` finalizes validation, `xcommit` commits write-set
  - Other code can run in between two steps
    - Code is logically part of the transaction
- Example uses
  - Finalize I/O operations within transactions
  - Coordinate with other SW for permission to commit
    - Correctness/security checkers, system transactions, …

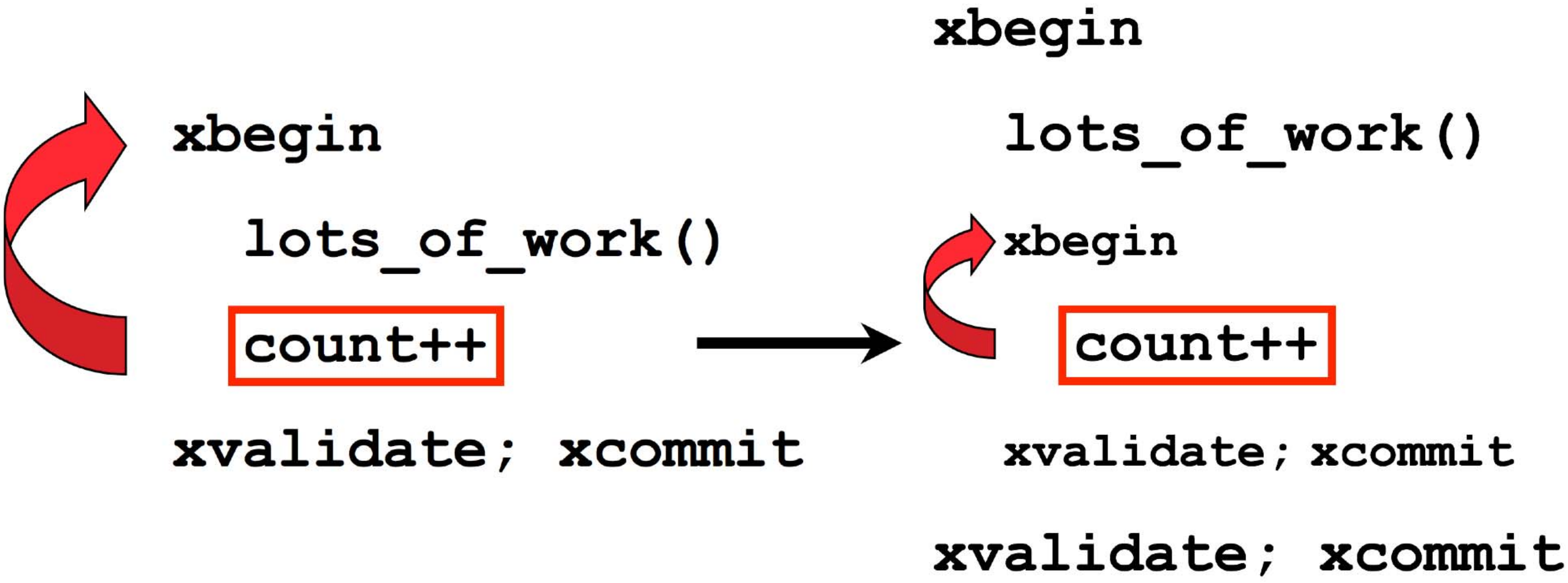
Transactional Handlers

- Conventional: TM events processed by hardware
  - Commit: commit write-set and proceed with following code
  - Abort on conflict: rollback transaction and re-execute
- New: all TM events processed by software handlers
  - Fast, user-level handlers for commit, conflict, and abort
  - Software can register multiple handlers per transaction
    - Stack of handlers maintained in software
  - Handlers have access to all transactional state
    - They decide what to commit or rollback, to re-execute or not, …
- Example uses
  - Contention managers, I/O operations within transactions, conditional synchronization

Non-Transactional Loads and Stores

- Conventional: all loads/stores tracked by HTM
  - Regardless of the type of data accesses
- New: instructions for non-transactional loads/stores
  - Non-transactional load: not tracked in read-set
  - Non-transactional store: not tracked in write
    - Appropriate for local or private data
  - Idempotent store: not versioned
    - Appropriate for data transaction-local data
- Example uses
  - Optimizations to eliminate spurious conflicts & overflow cases
  - Object-based hybrid TM (track headers only)

Closed-nested Transactions



```
xbegin
  lots_of_work()
  count++
xvalidate; xcommit
```

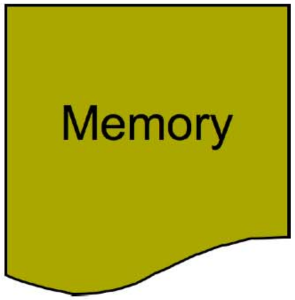
```
xbegin
  lots_of_work()
  xbegin
    count++
  xvalidate; xcommit
xvalidate; xcommit
```

- Closed Nesting
  - Composable libraries
  - Alternative control flow upon nested abort
  - Performance improvement (reduce abort penalty)

Closed-nested Transactions

Closed-nested Semantics

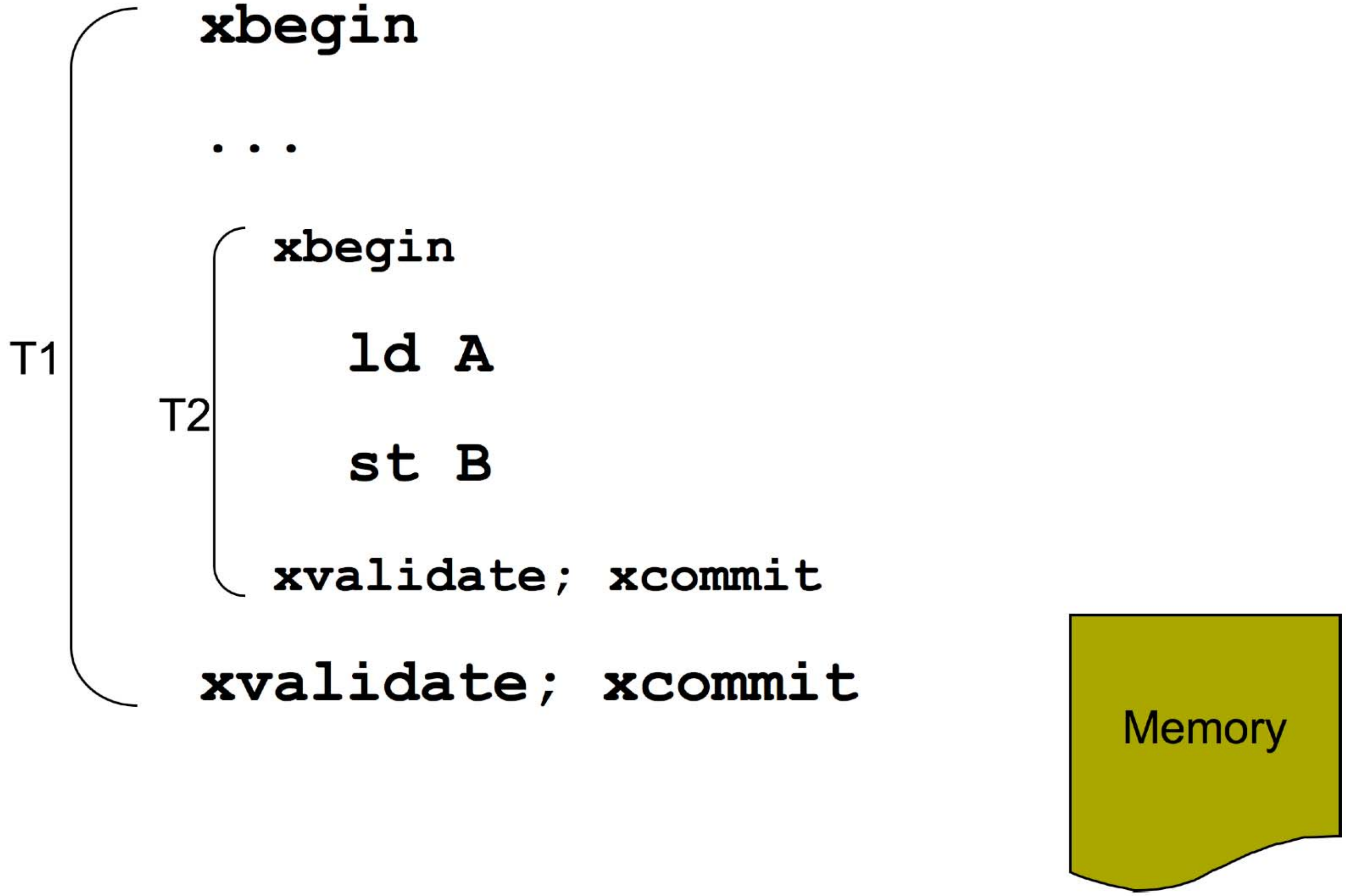
```
xbegin
...
  xbegin
    ld A
    st B
  xvalidate; xcommit
xvalidate; xcommit
```



Memory

Closed-nested Transactions

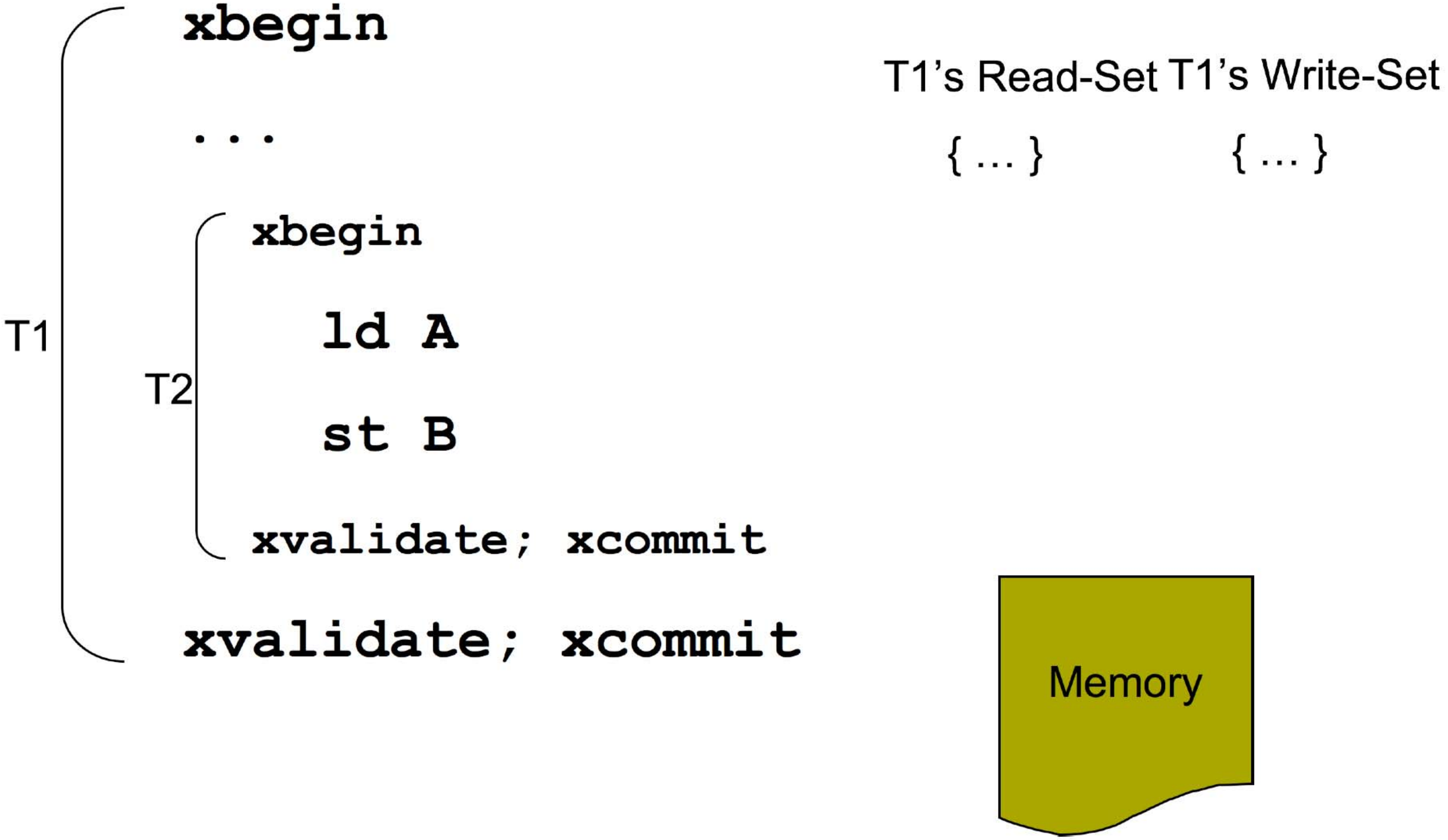
Closed-nested Semantics



```
xbegin
...
  xbegin
     ld A
     st B
  xvalidate; xcommit
xvalidate; xcommit
```

Closed-nested Transactions

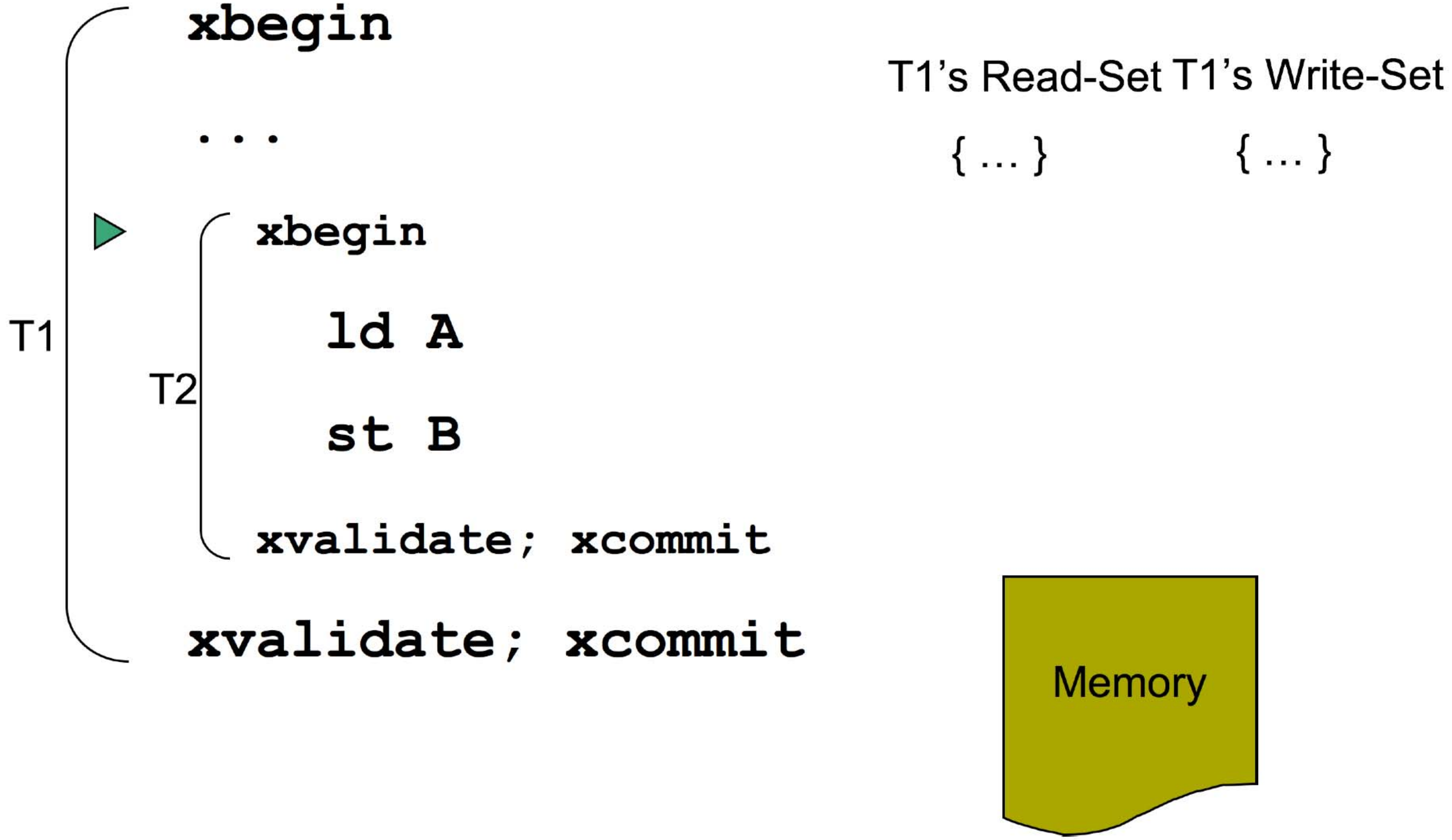
Closed-nested Semantics



```
xbegin

...

    xbegin

        ld A

        st B

    xvalidate; xcommit

xvalidate; xcommit
```

Closed-nested Transactions

Closed-nested Semantics



```
xbegin
...
   xbegin
      ld A
      st B
   xvalidate; xcommit
xvalidate; xcommit
```

T1

T2

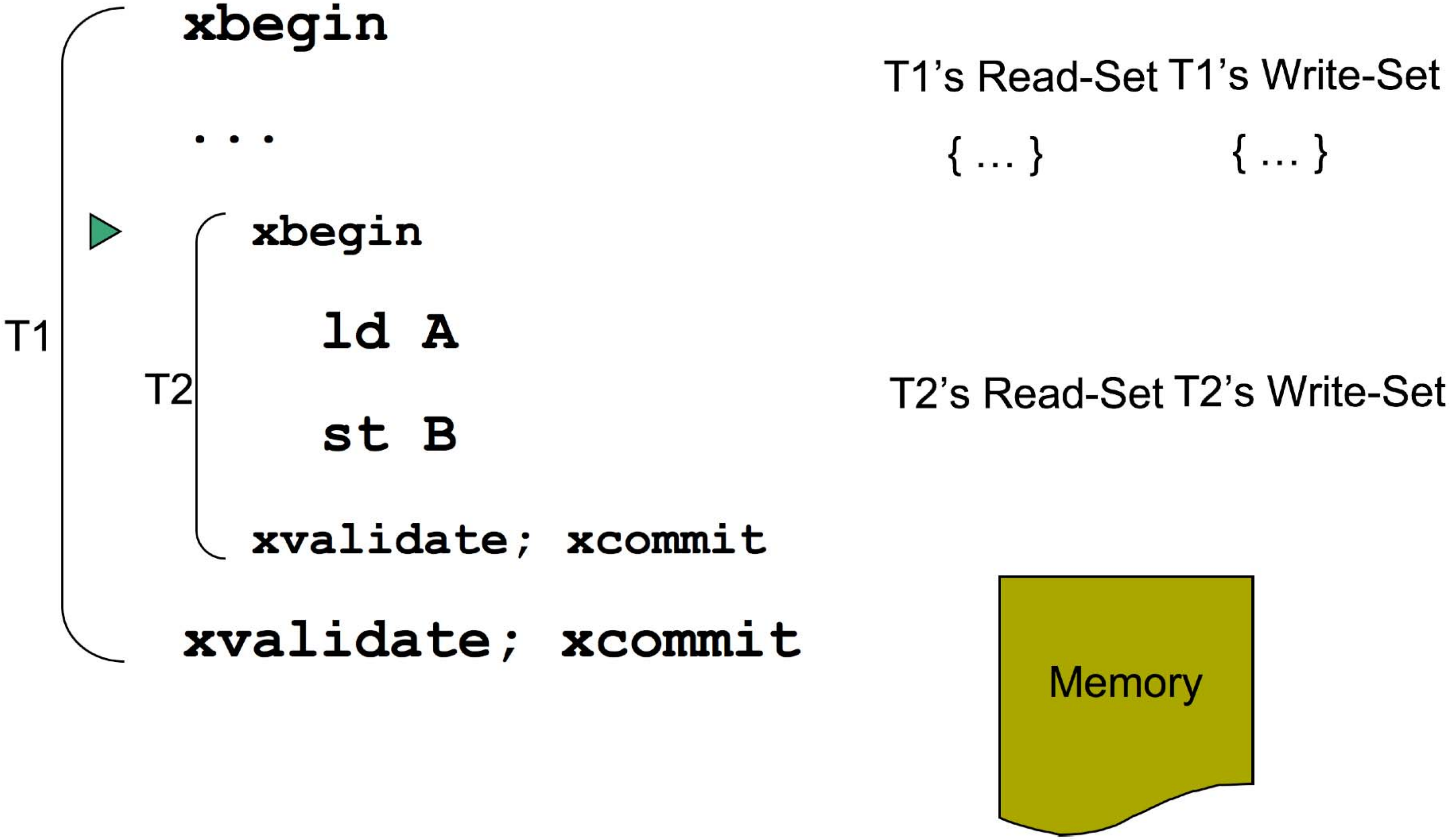
T1's Read-Set T1's Write-Set

{ … } { … }

Memory

Closed-nested Transactions

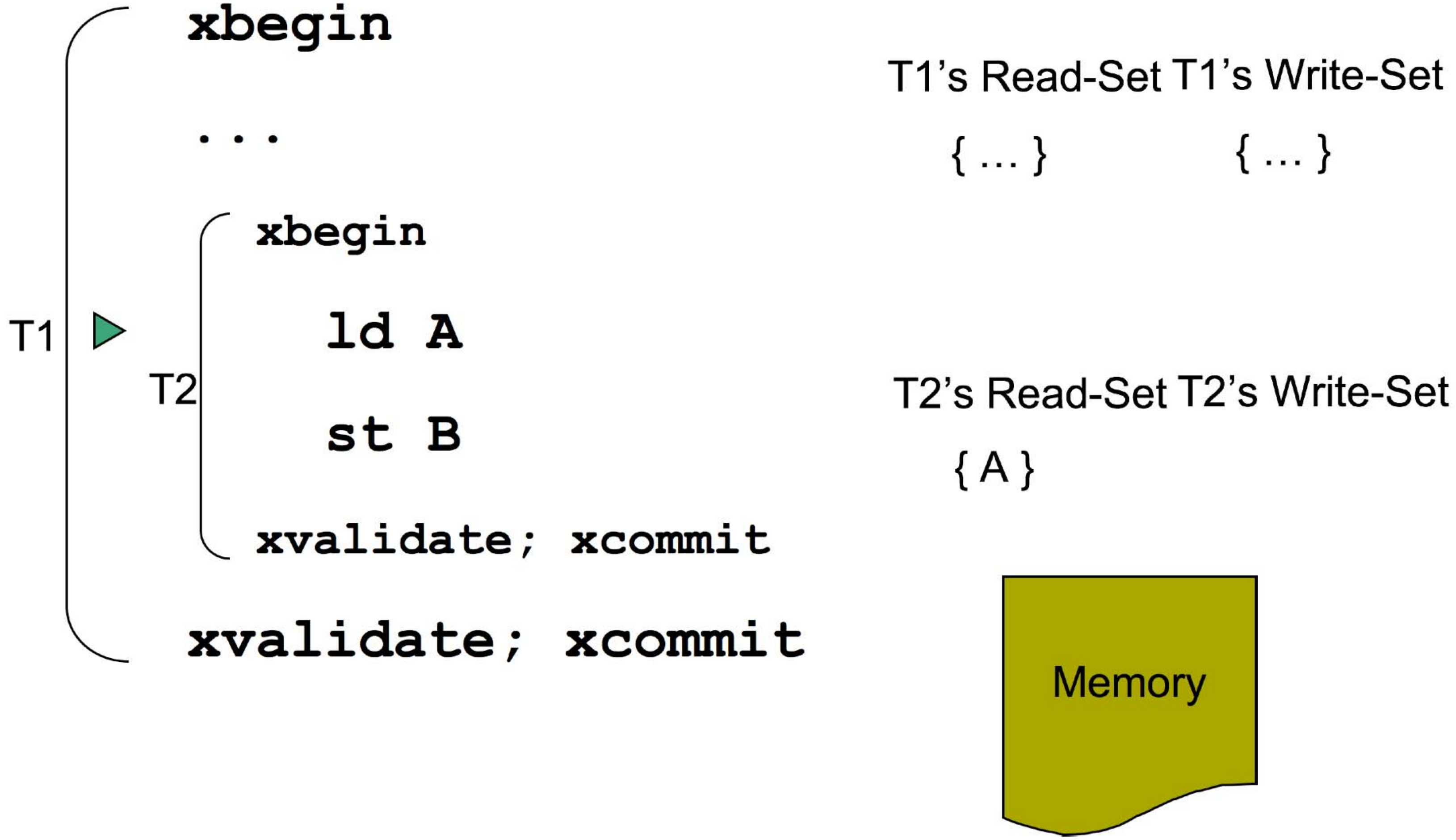
Closed-nested Semantics



```
xbegin
...
    xbegin
        ld A
        st B
    xvalidate; xcommit
xvalidate; xcommit
```

Closed-nested Transactions

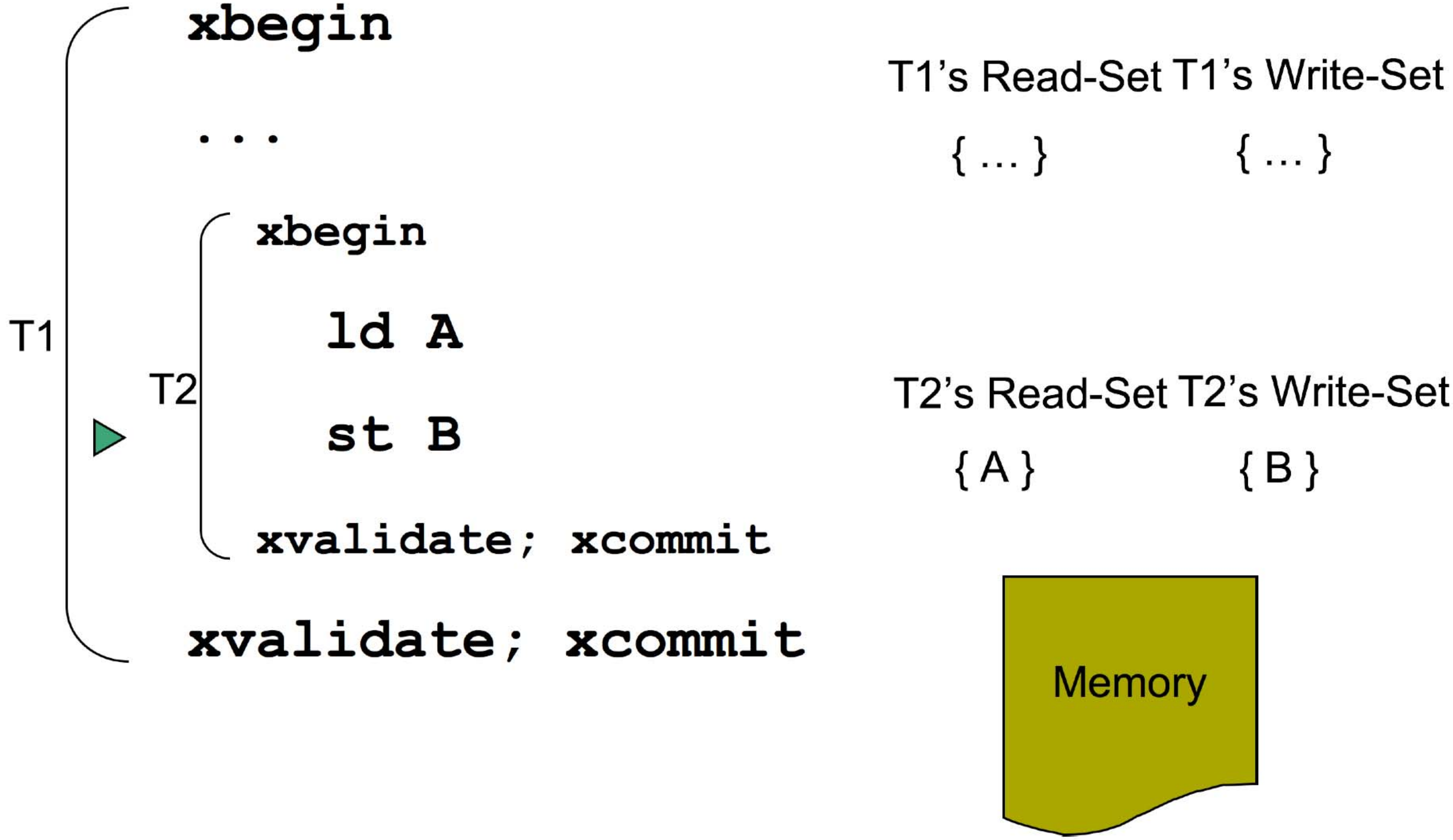
Closed-nested Semantics



```
xbegin
...
   xbegin
      ld A
      st B
   xvalidate; xcommit
xvalidate; xcommit
```

Closed-nested Transactions

Closed-nested Semantics



```
xbegin
...
   xbegin
      ld A
      st B
   xvalidate; xcommit
xvalidate; xcommit
```

T1's Read-Set T1's Write-Set

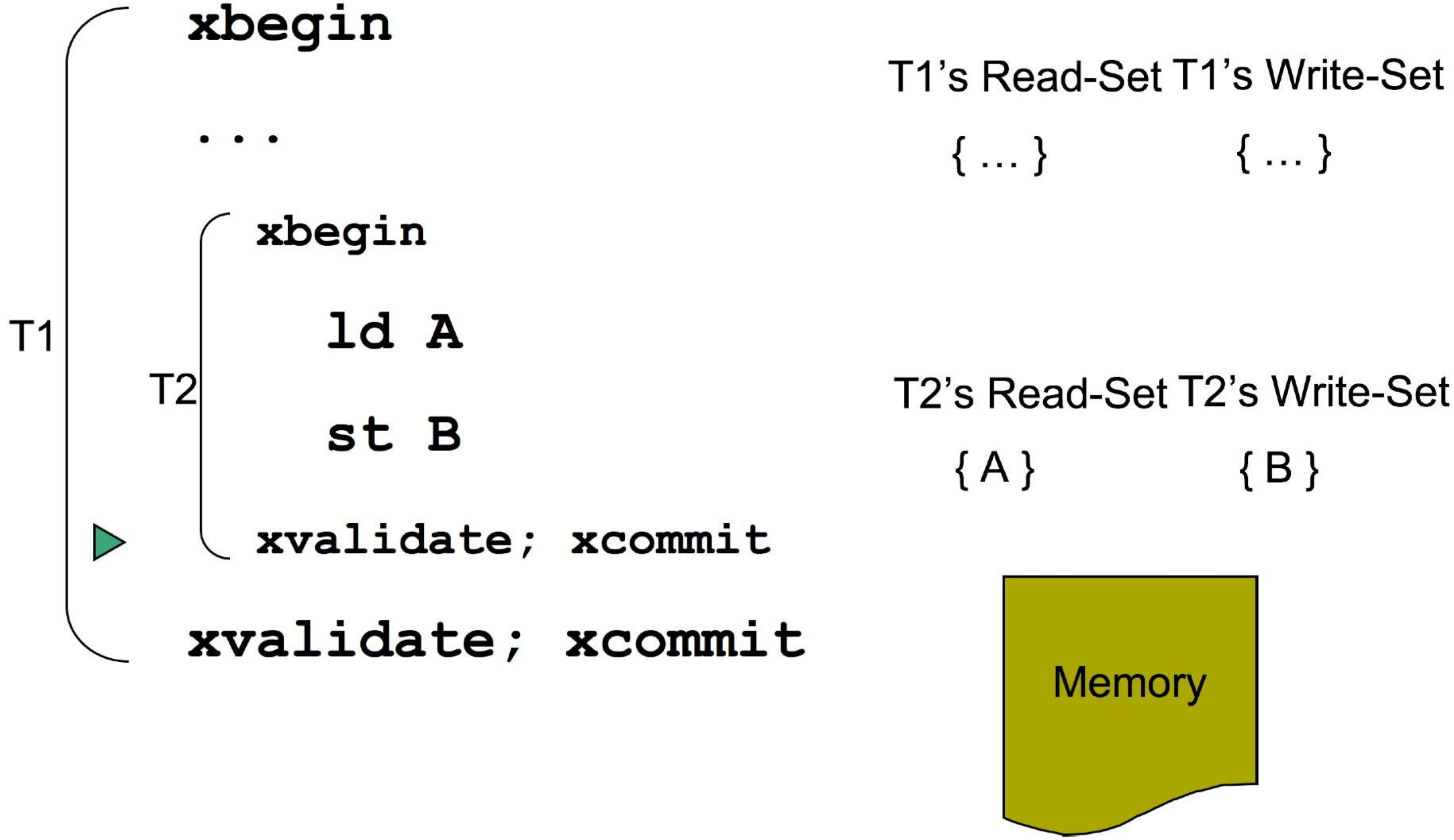
{ … } { … }

T2's Read-Set T2's Write-Set

{ A } { B }

Closed-nested Transactions

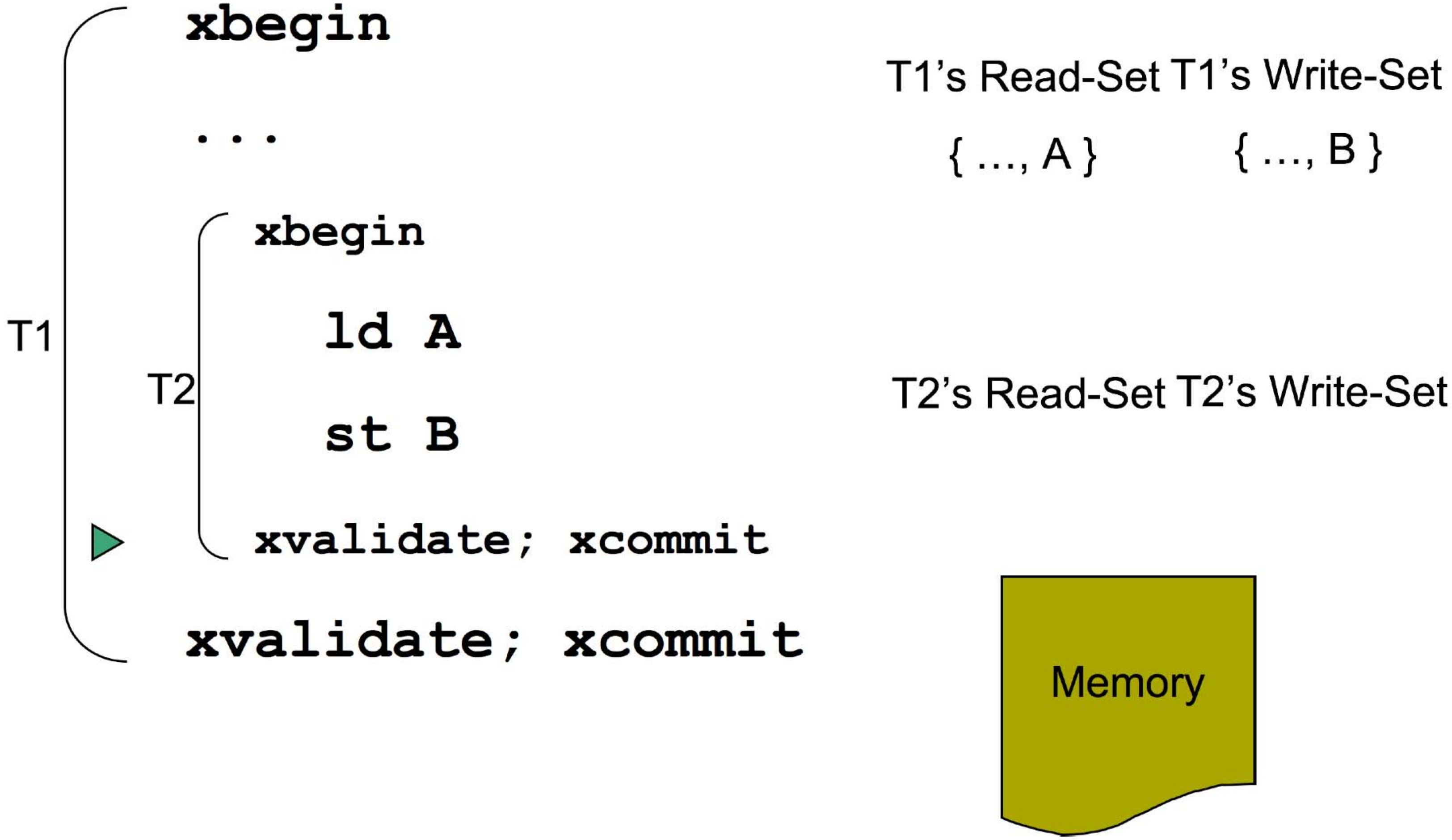
Closed-nested Semantics



```
T1  xbegin
    ...
    T2  xbegin
            ld A
            st B
        xvalidate; xcommit
    xvalidate; xcommit
```

Closed-nested Transactions

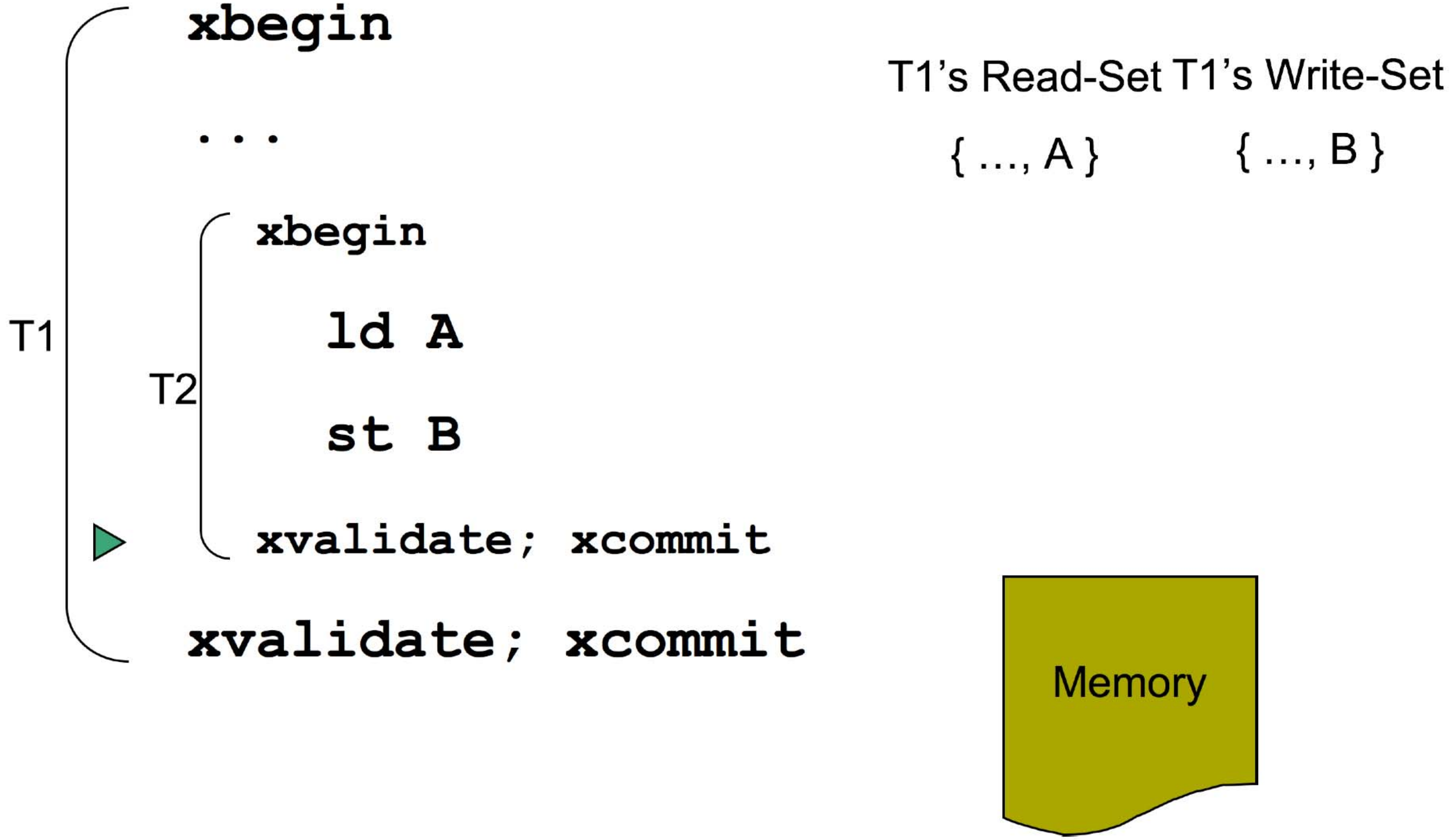
Closed-nested Semantics



```
xbegin
...
  xbegin
    ld A
    st B
  xvalidate; xcommit
xvalidate; xcommit
```

Closed-nested Transactions

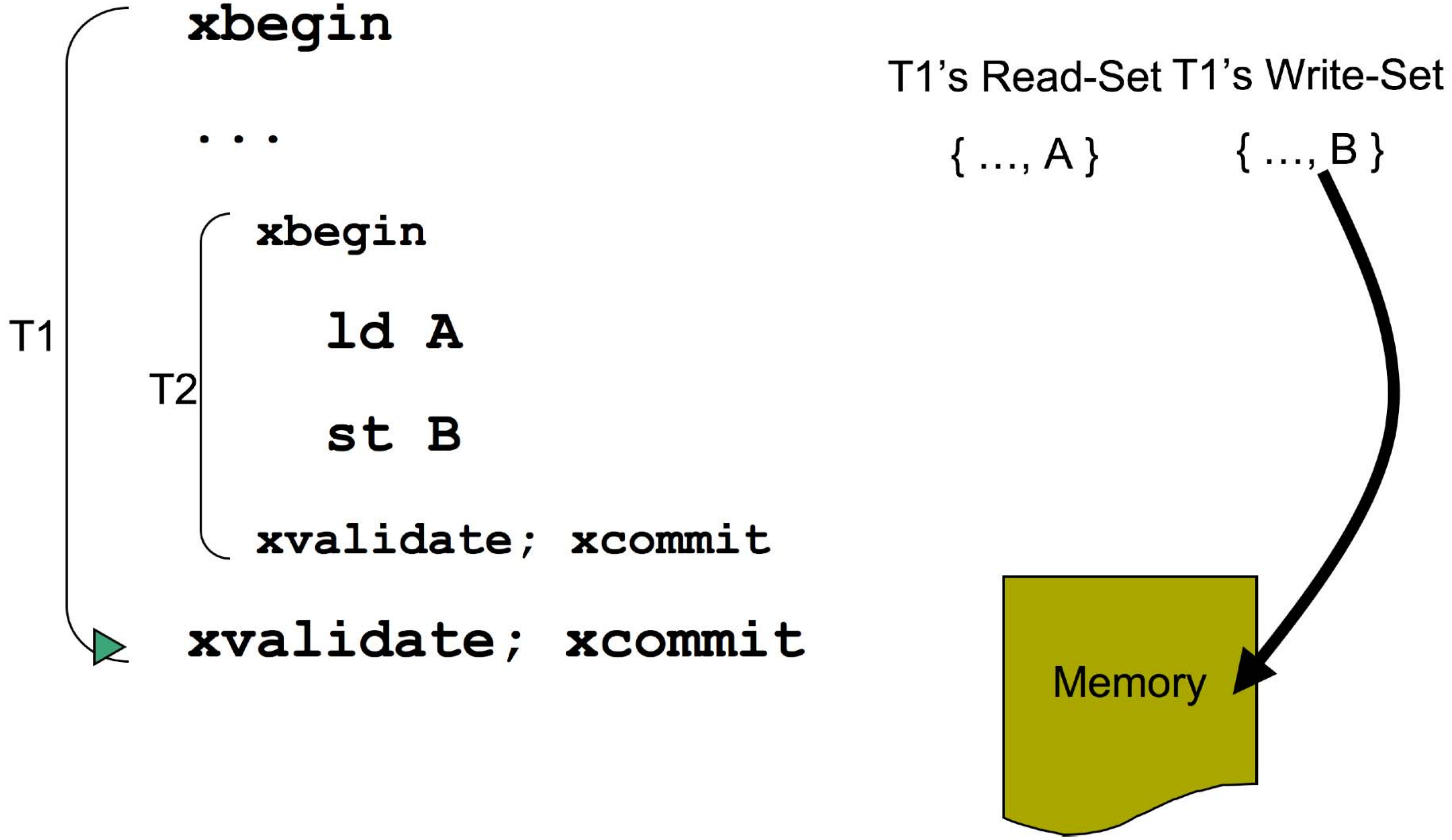
Closed-nested Semantics



```
xbegin
...
   xbegin
      ld A
      st B
   xvalidate; xcommit
xvalidate; xcommit
```

Closed-nested Transactions

Closed-nested Semantics



```
xbegin
...
  xbegin
    ld A
    st B
  xvalidate; xcommit
xvalidate; xcommit
```

T1

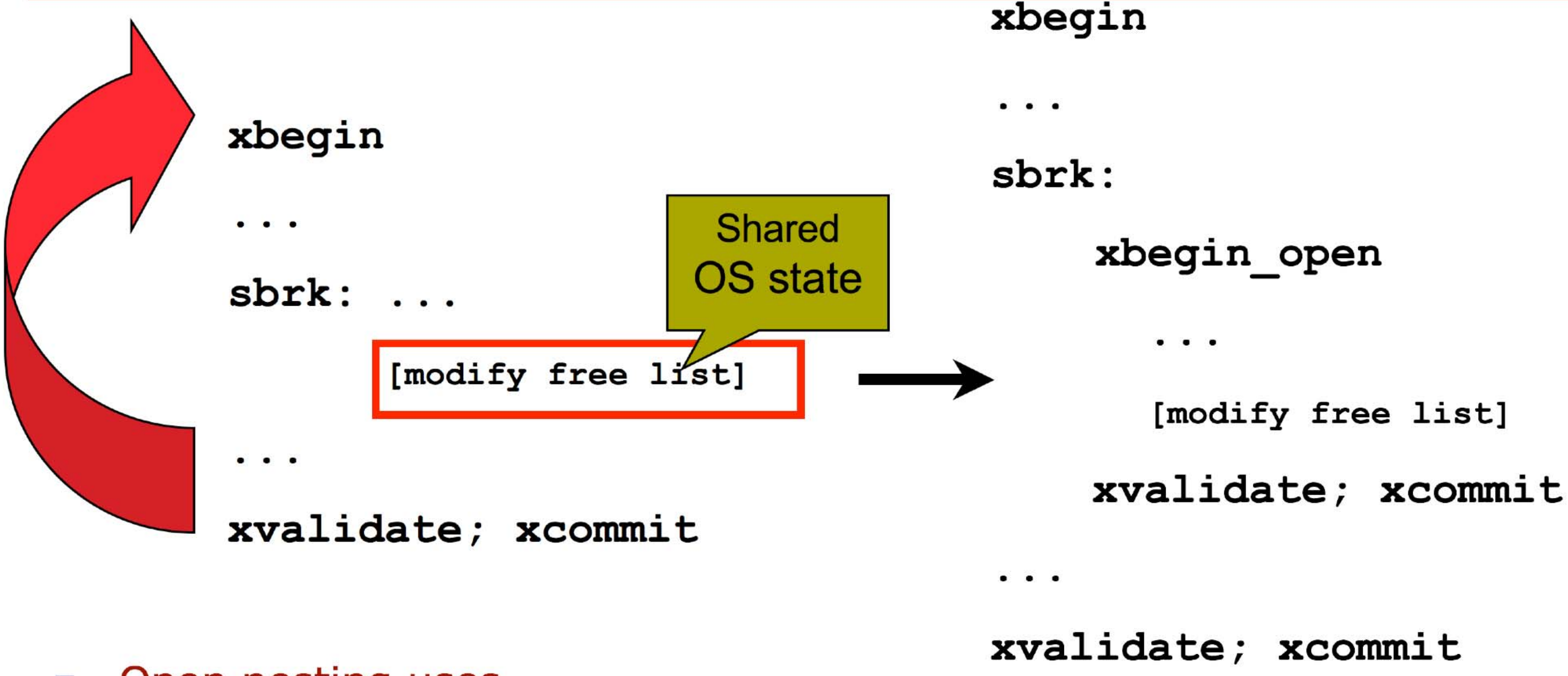
T2

T1's Read-Set T1's Write-Set

{ …, A } { …, B }

Memory

Open-nested Transactions



```
xbegin

...

sbrk: ...

        [modify free list]

...

xvalidate; xcommit
```

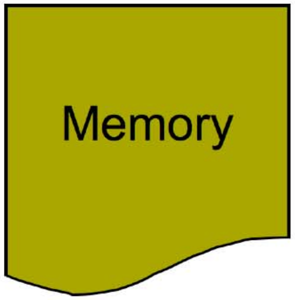
```
xbegin

...

sbrk:

    xbegin_open

      ...

      [modify free list]

    xvalidate; xcommit

...

xvalidate; xcommit
```

- Open nesting uses
  - Escape surrounding atomicity to update shared state
    - System calls, communication between transactions/OS/scheduler/etc.
  - Performance improvements
- Open nesting provides atomicity & isolation for enclosed code
  - Unlike pause/escape/non-transactional regions

Open-nested Transactions

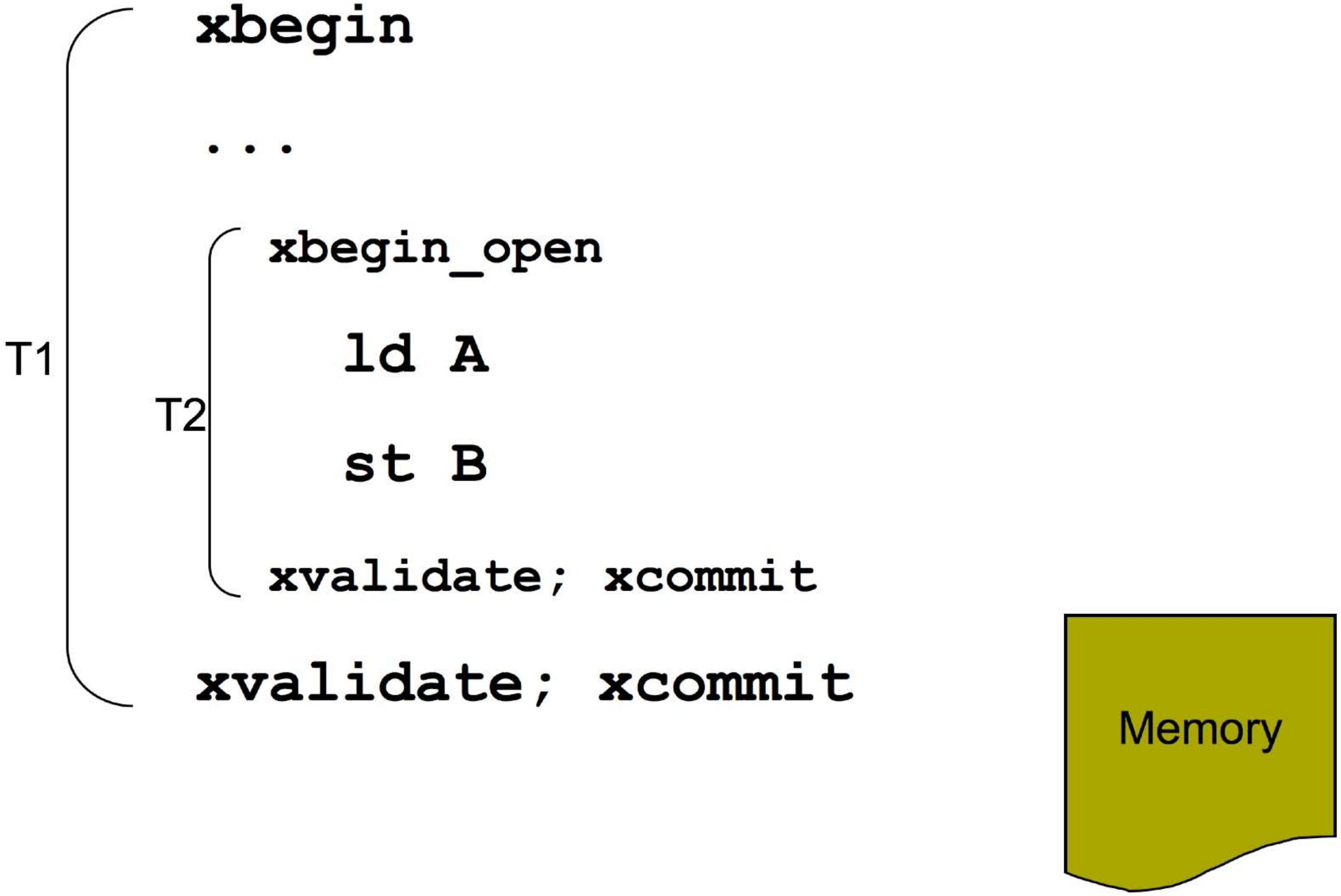
Open-nested Semantics

```
xbegin
...
  xbegin_open
    ld A
    st B
  xvalidate; xcommit
xvalidate; xcommit
```



Open-nested Transactions

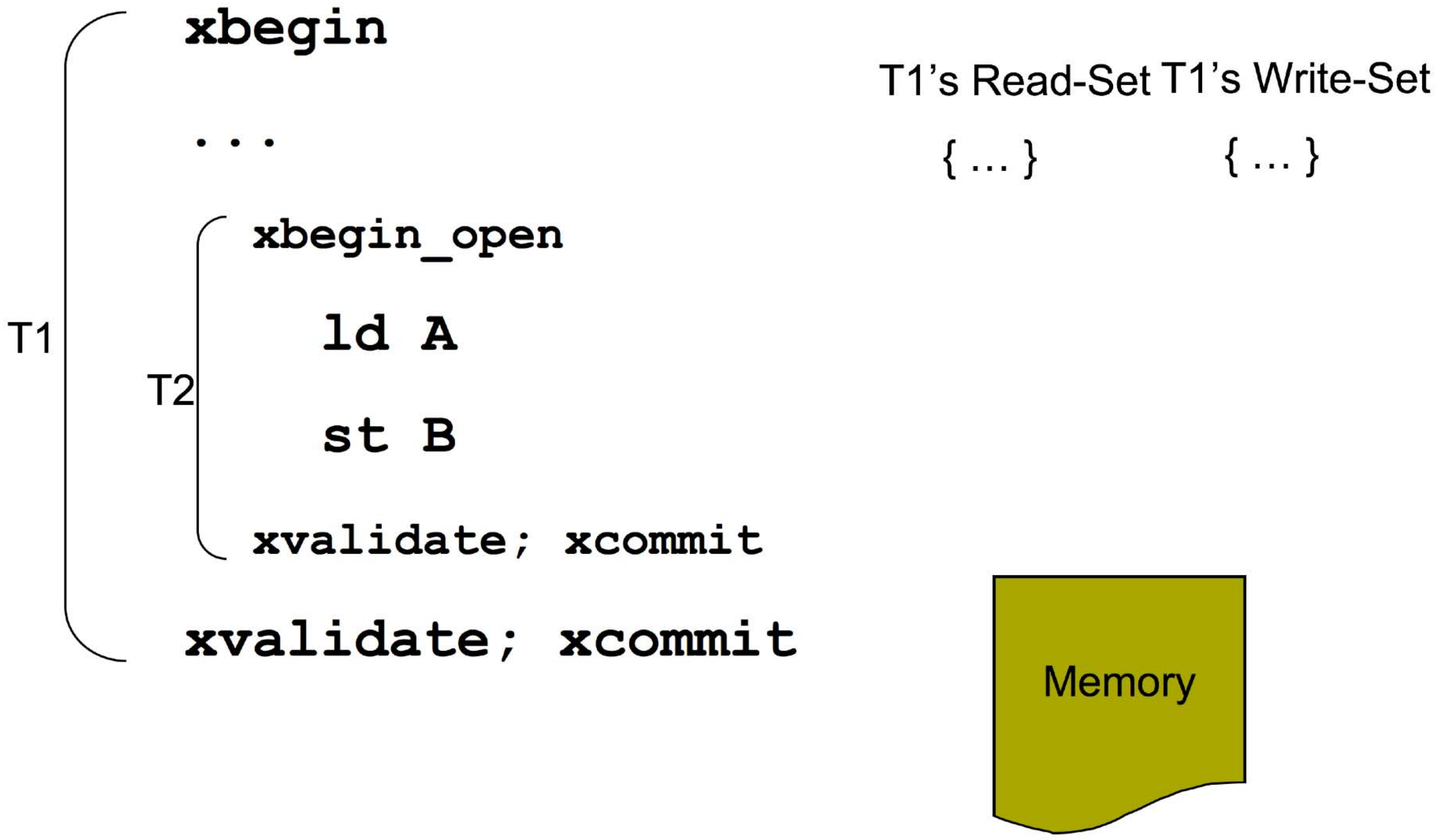
Open-nested Semantics



```
xbegin

...

  xbegin_open

    ld A

    st B

  xvalidate; xcommit

xvalidate; xcommit
```

Open-nested Transactions

Open-nested Semantics



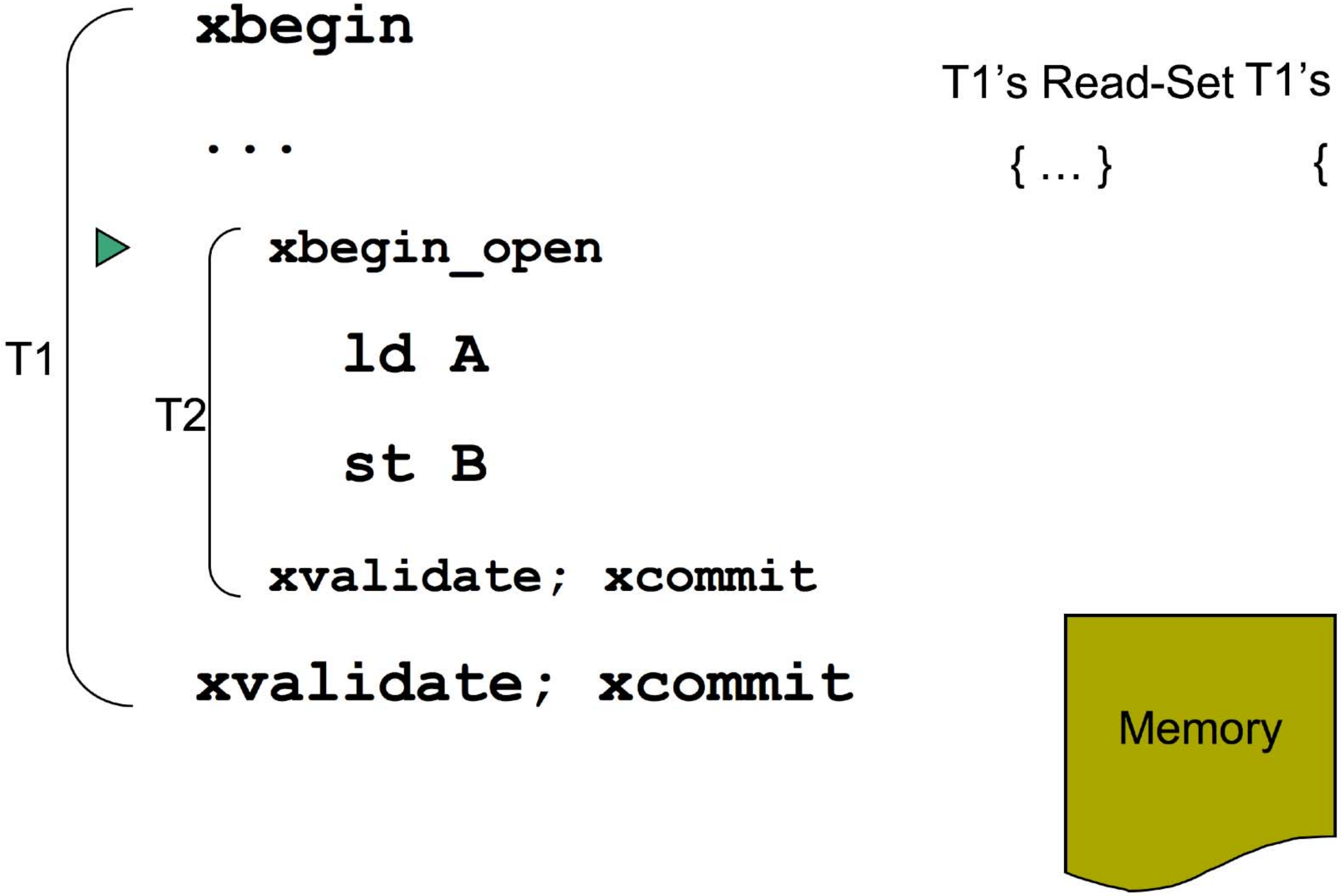
```
T1  xbegin
    ...
T2    xbegin_open
        ld A
        st B
      xvalidate; xcommit
    xvalidate; xcommit
```

T1's Read-Set T1's Write-Set

{ … } { … }

Open-nested Transactions

Open-nested Semantics



```
xbegin
...
    xbegin_open
       ld A
       st B
    xvalidate; xcommit
xvalidate; xcommit
```

T1

T2

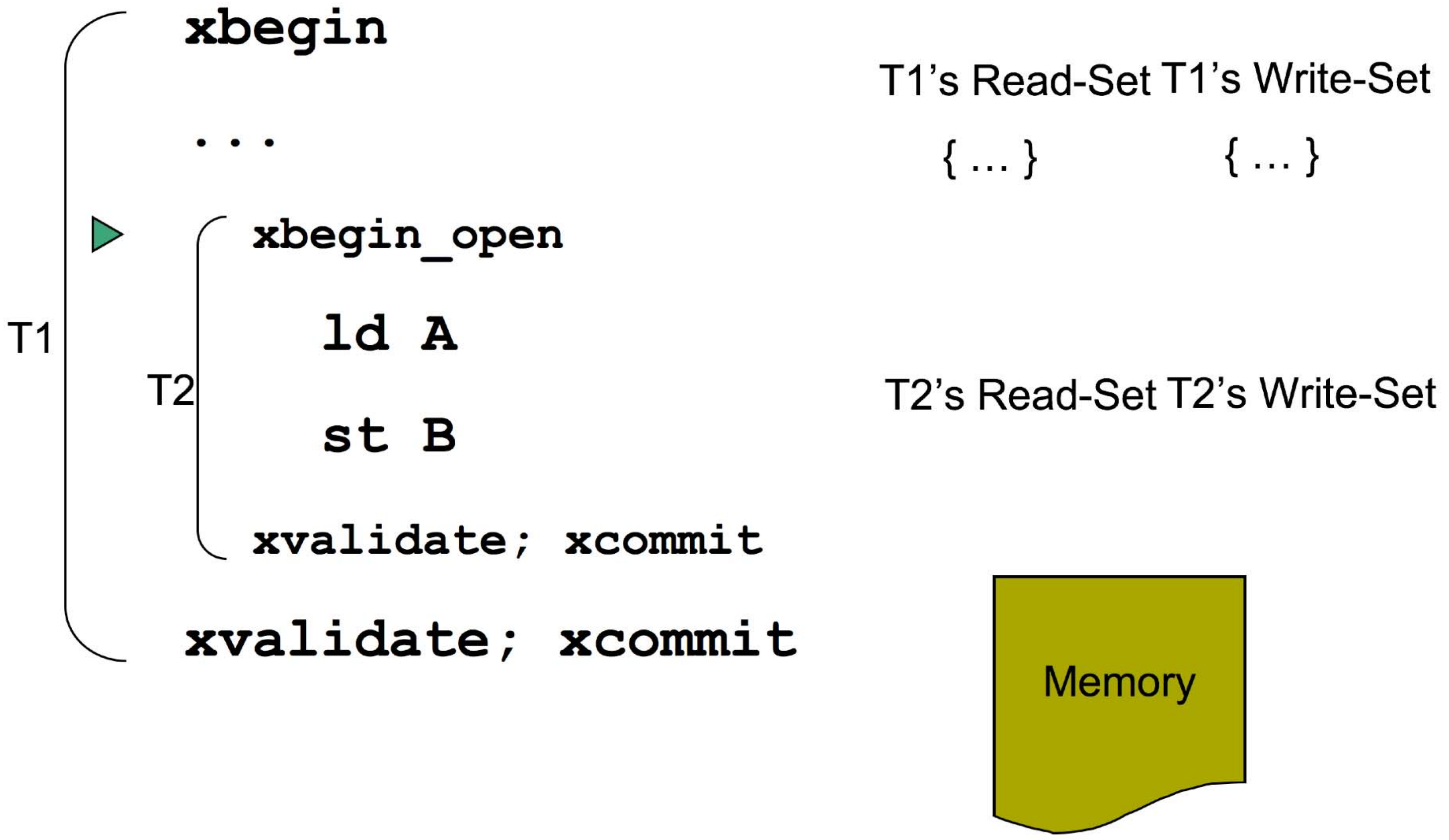
T1’s Read-Set T1’s Write-Set

{ … } { … }

Memory

Open-nested Transactions

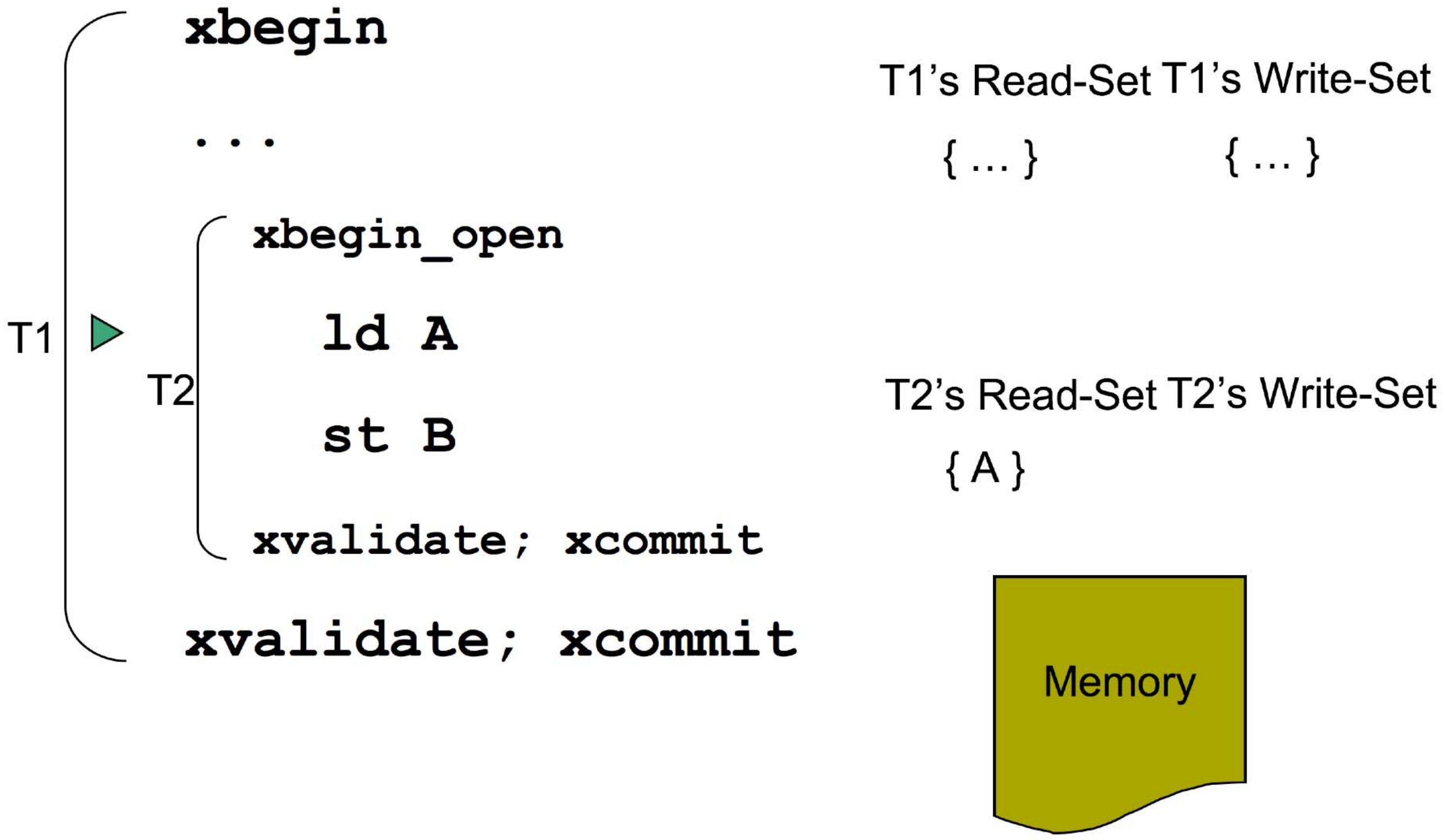
Open-nested Semantics



```
xbegin
...
▶   xbegin_open
      ld A
      st B
    xvalidate; xcommit
xvalidate; xcommit
```

Open-nested Transactions

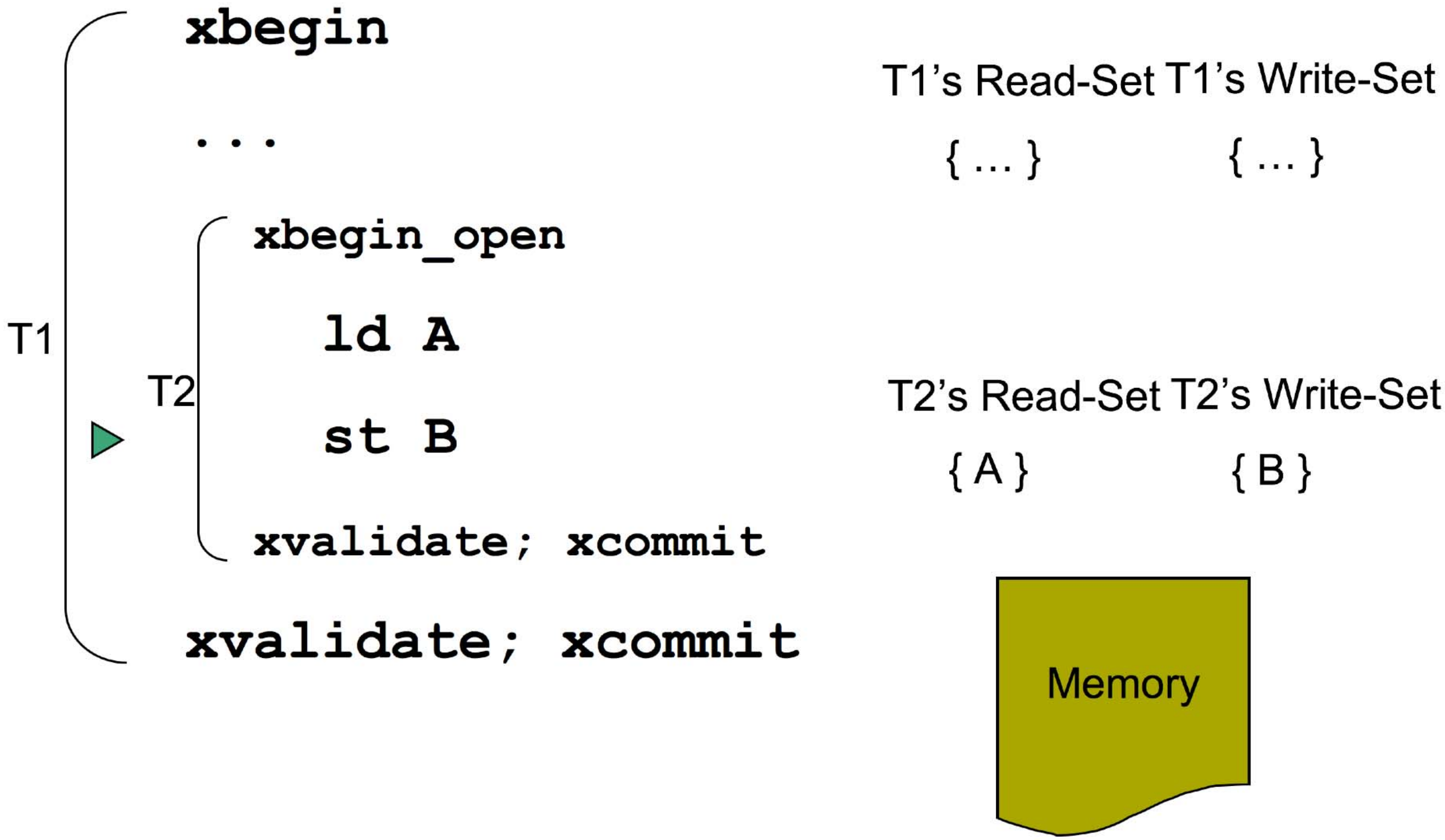
Open-nested Semantics



```
xbegin

...

  xbegin_open

     ld A

     st B

  xvalidate; xcommit

xvalidate; xcommit
```

Open-nested Transactions

Open-nested Semantics



```
xbegin
...
  xbegin_open
    ld A
    st B
  xvalidate; xcommit
xvalidate; xcommit
```

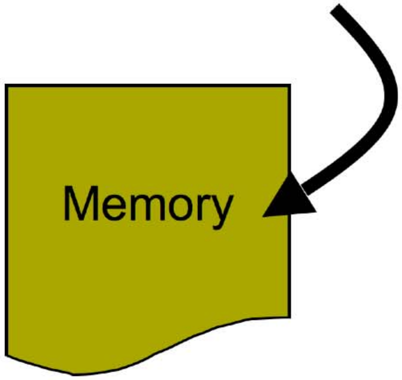
Open-nested Transactions

Open-nested Semantics

```
       xbegin

       ...

         xbegin_open

             ld A
T1
     T2      st B

  ▷      xvalidate; xcommit

       xvalidate; xcommit
```

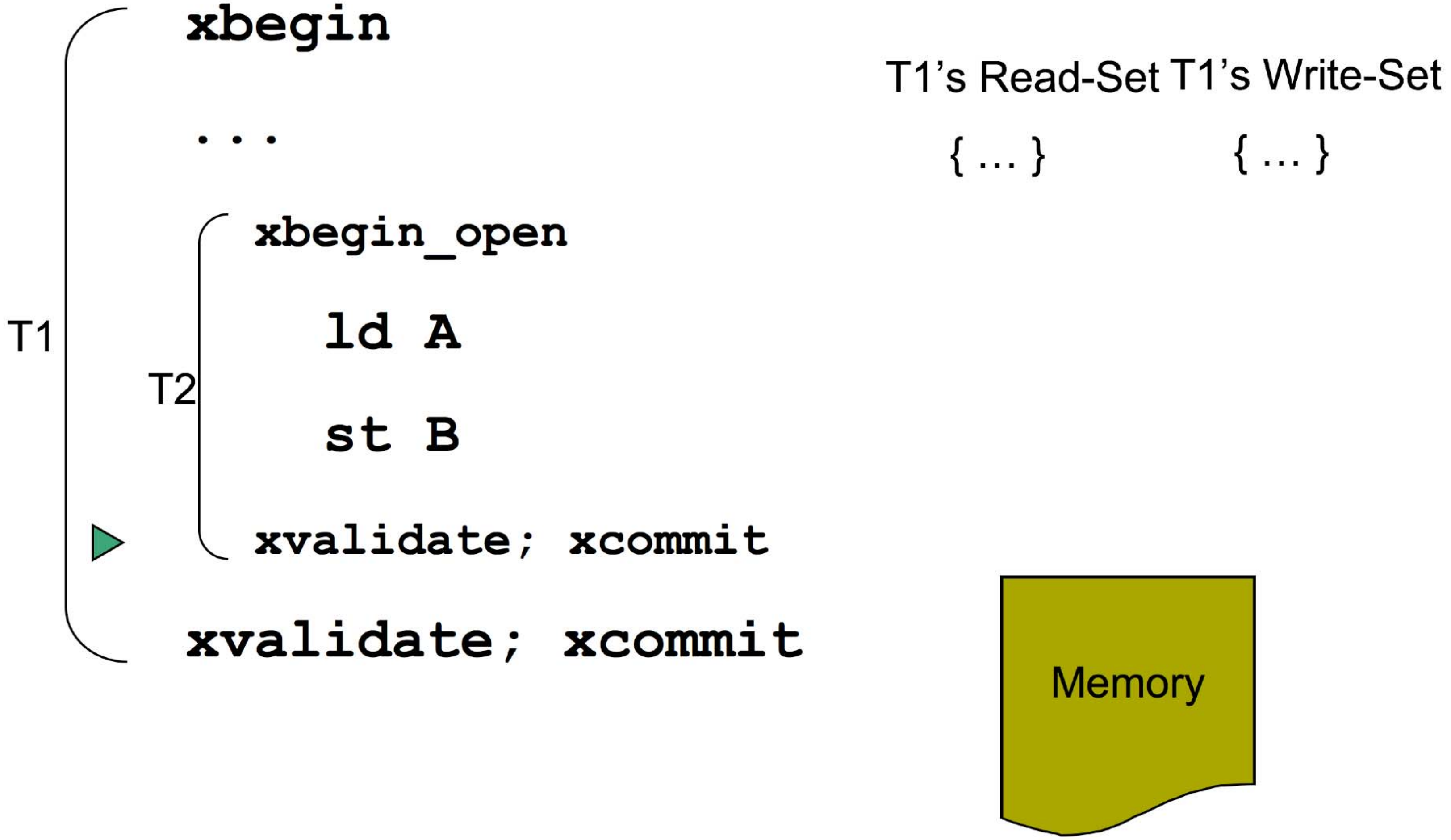
T1's Read-Set	T1's Write-Set
{ … }	{ … }

T2's Read-Set	T2's Write-Set
{ A }	{ B }



Open-nested Transactions

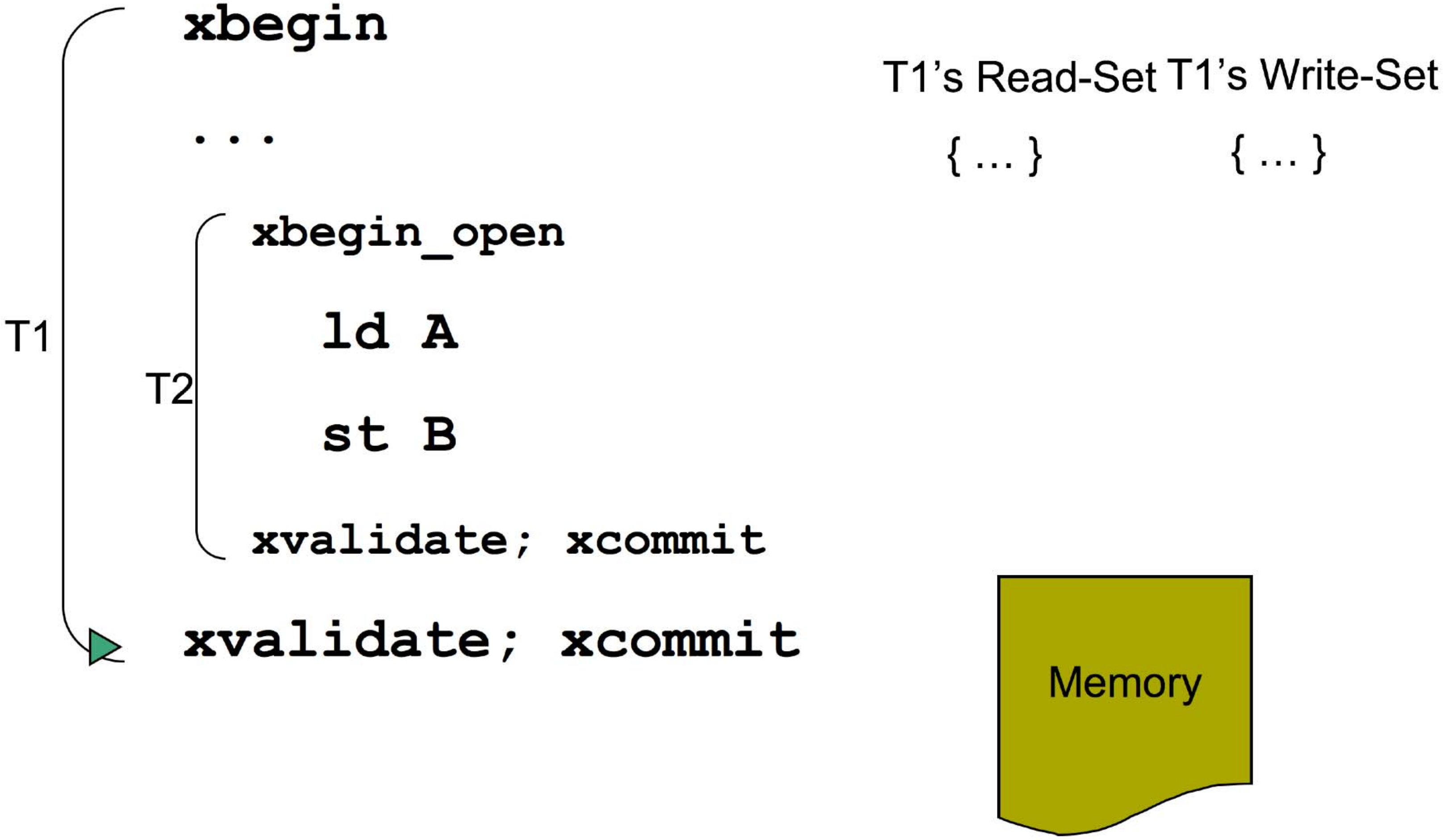
Open-nested Semantics



```
xbegin
...
  xbegin_open
     ld A
     st B
  xvalidate; xcommit
xvalidate; xcommit
```

Open-nested Transactions

Open-nested Semantics



```
xbegin
...
  xbegin_open
    ld A
    st B
  xvalidate; xcommit
xvalidate; xcommit
```

T1

T2

T1’s Read-Set T1’s Write-Set

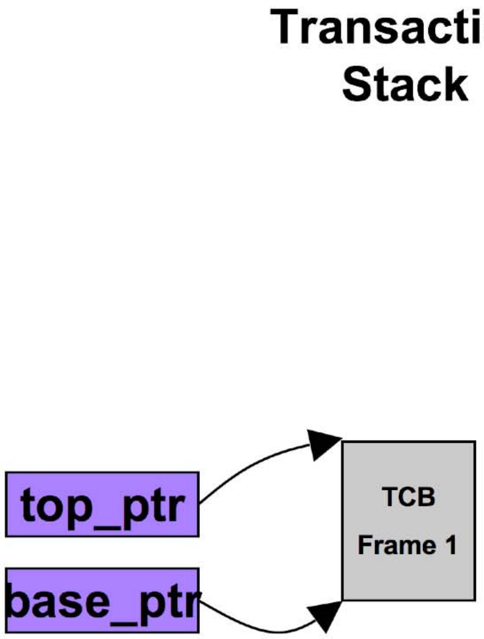
{ … } { … }

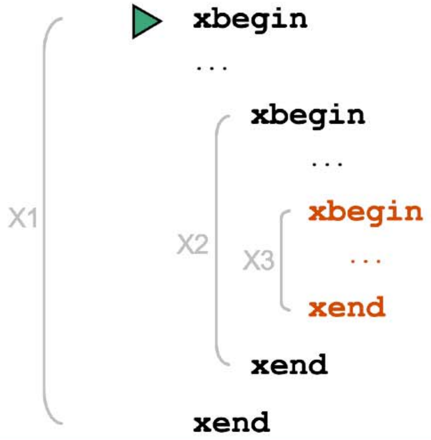
Memory

Implementation Overview

- Software
  - Stack to track state and handlers
    - Like activation records for function calls
    - Works with nested transactions, multiple handlers per transaction
  - Handlers like user-level exceptions
- Hardware
  - A few new instructions & registers
    - Registers mostly for faster access of state logically in the stack
    - To provide information to handlers
  - Modified cache design for nested transactions
    - Independent tracking of read-set and write-set
- Key concepts
  - Nested transactions supported similarly to nested function calls
  - Handlers implemented as light-weight, user-level exceptions

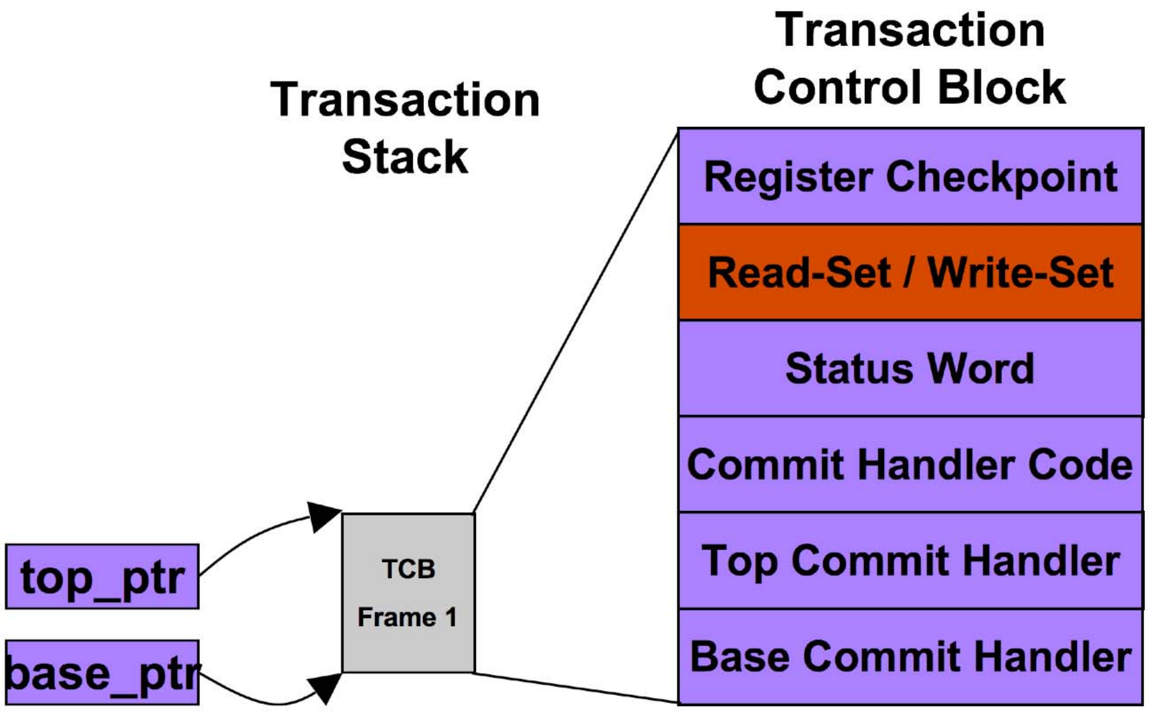
Transaction Stack





```
xbegin
...
   xbegin
      ...
      xbegin
         ...
      xend
   xend
xend
```

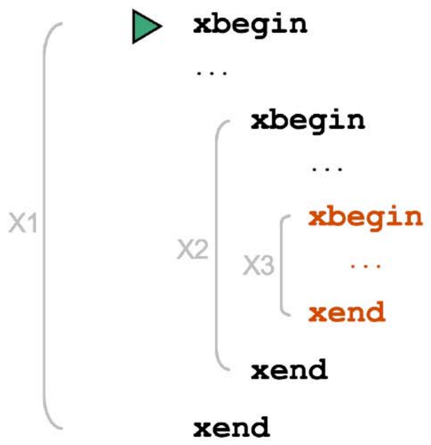
Transaction Stack



🟧 = in cache / log

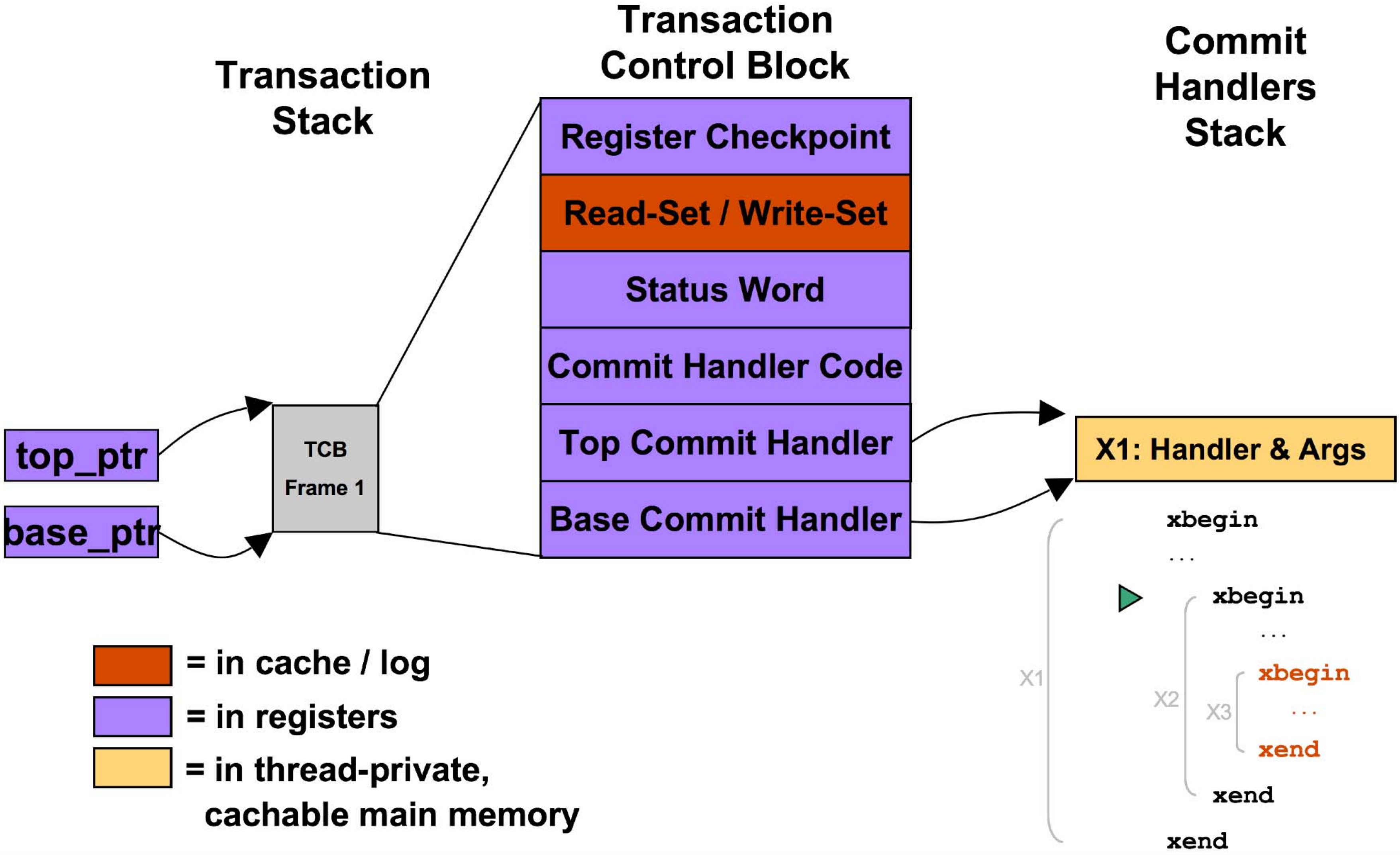
🟪 = in registers

🟨 = in thread-private, cachable main memory



```
▶ xbegin
  ...
    xbegin
      ...
      xbegin
        ...
      xend
    xend
  xend
```

Transaction Stack



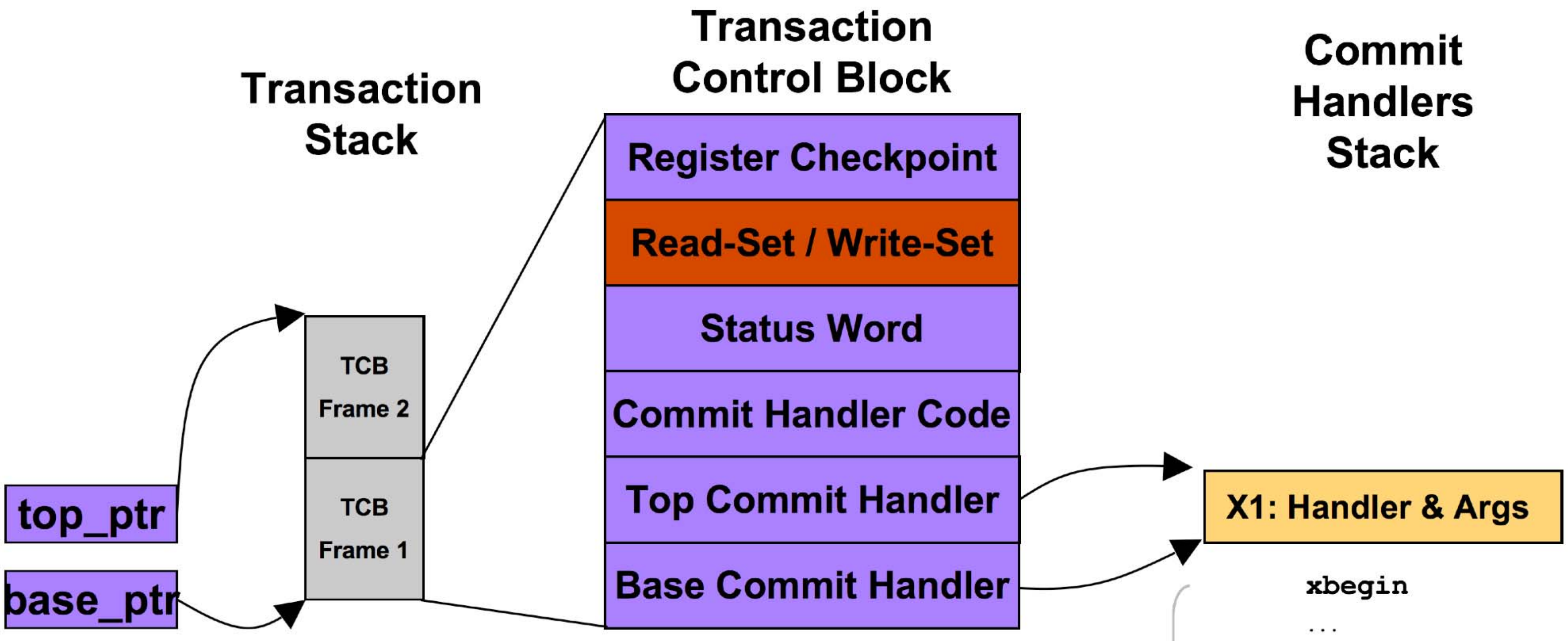
```
xbegin
...
   xbegin
      ...
      xbegin
         ...
      xend
   xend
xend
```

🟧 = in cache / log

🟪 = in registers

🟨 = in thread-private,
cachable main memory

Transaction Stack



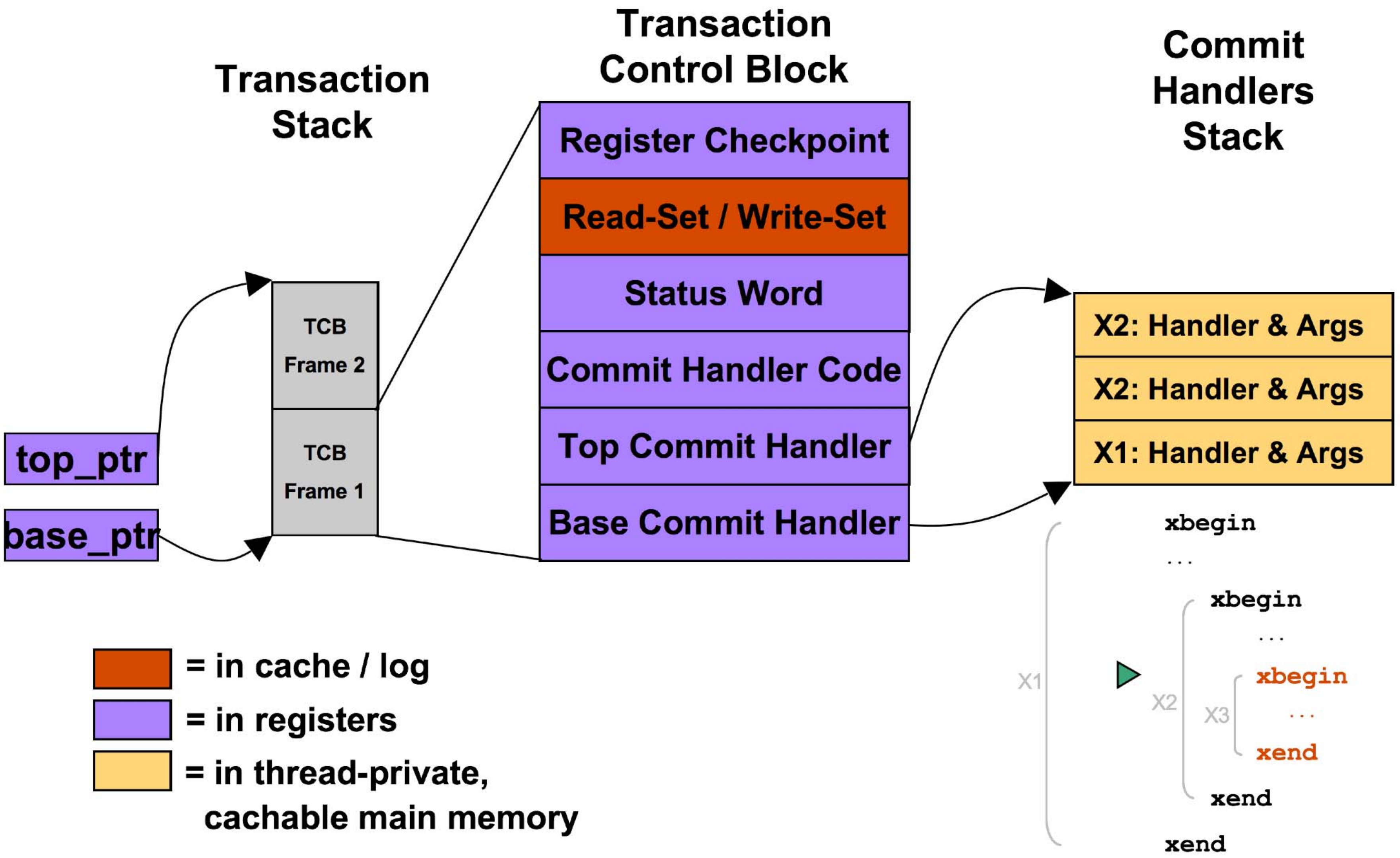
🟧 = in cache / log

🟪 = in registers

🟨 = in thread-private, cachable main memory

```
         xbegin
         ...
      ▶     xbegin
            ...
               xbegin
X1     X2   X3 ...
               xend
            xend
         xend
```

Transaction Stack



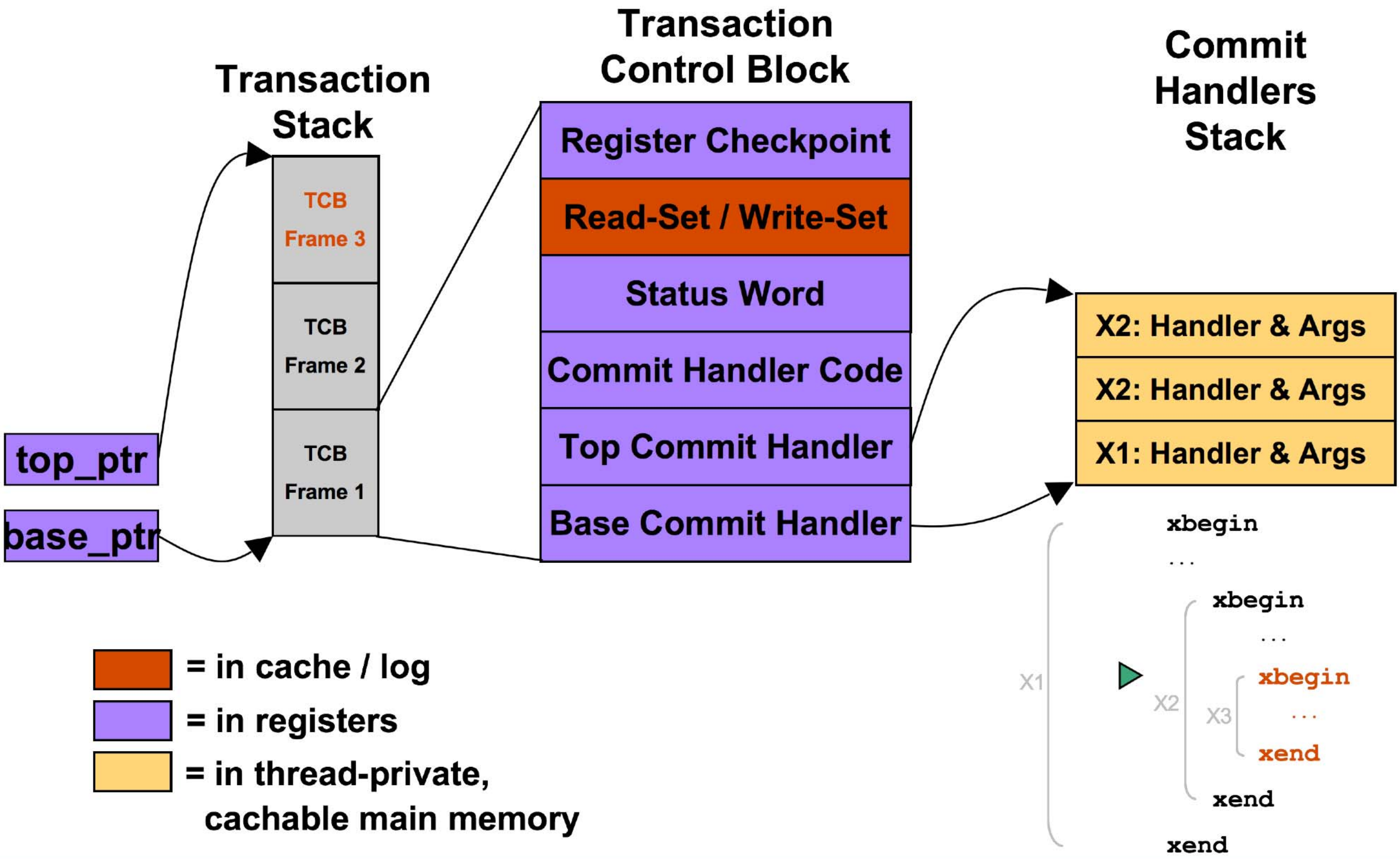
```
xbegin
...
    xbegin
        ...
        xbegin
            ...
        xend
    xend
xend
```

🟧 = in cache / log

🟪 = in registers

🟨 = in thread-private,
cachable main memory

Transaction Stack



Transaction Stack

- TCB Frame 3
- TCB Frame 2
- TCB Frame 1

top_ptr

base_ptr

Transaction Control Block

- Register Checkpoint
- Read-Set / Write-Set
- Status Word
- Commit Handler Code
- Top Commit Handler
- Base Commit Handler

Commit Handlers Stack

- X2: Handler & Args
- X2: Handler & Args
- X1: Handler & Args

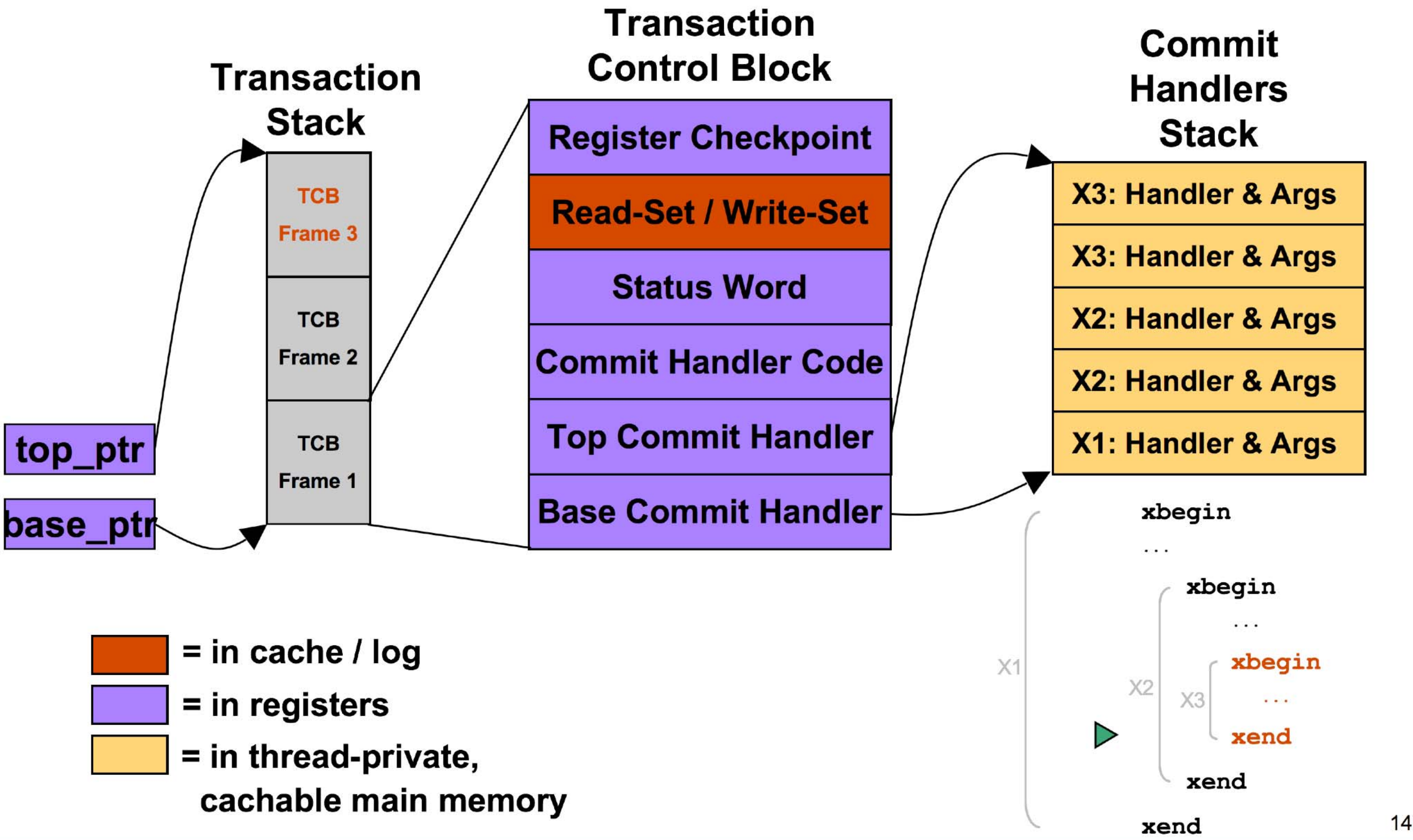
```
xbegin
...
   xbegin
     ...
      xbegin
        ...
      xend
   xend
xend
```

🟧 = in cache / log

🟪 = in registers

🟨 = in thread-private, cachable main memory

Transaction Stack



Transaction Stack

TCB Frame 3

TCB Frame 2

TCB Frame 1

top_ptr

base_ptr

Transaction Control Block

Register Checkpoint

Read-Set / Write-Set

Status Word

Commit Handler Code

Top Commit Handler

Base Commit Handler

Commit Handlers Stack

X3: Handler & Args

X3: Handler & Args

X2: Handler & Args

X2: Handler & Args

X1: Handler & Args

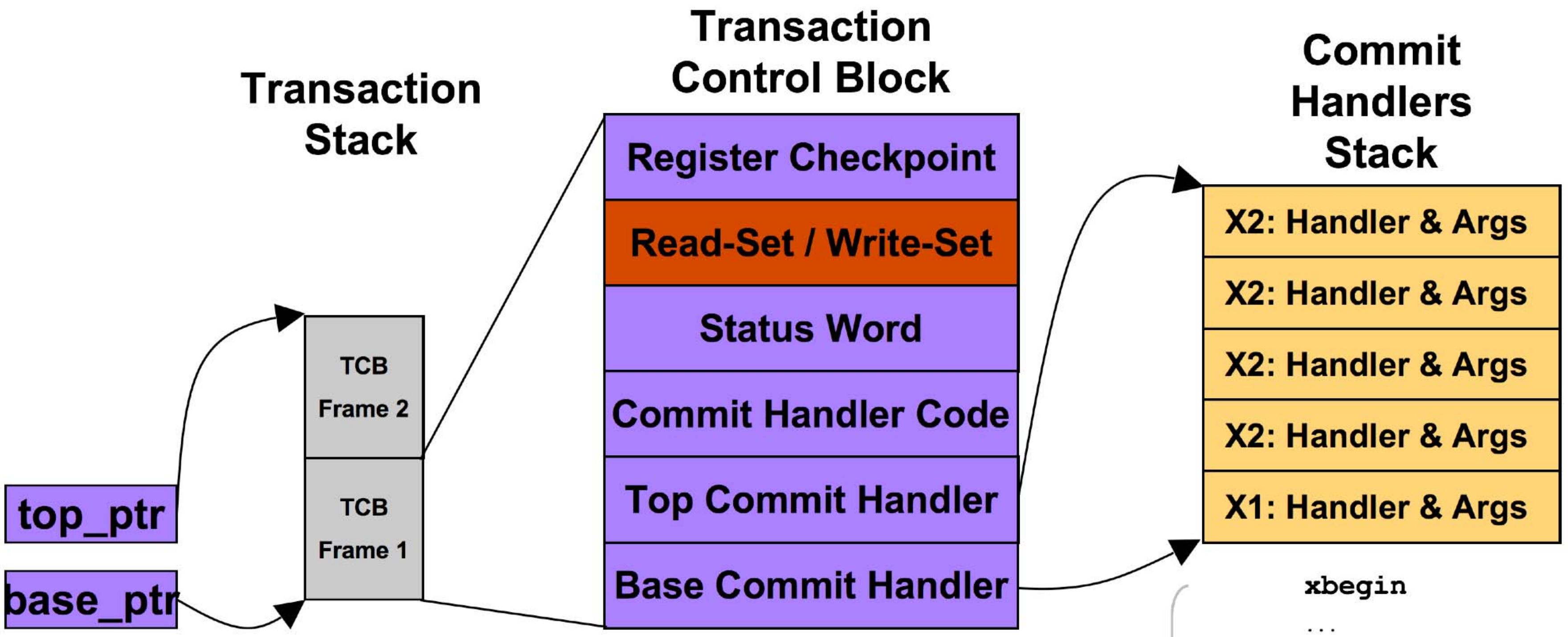
```
        xbegin
        ...
           xbegin
            ...
              xbegin
X1     X2   X3  ...
              xend
           xend
        xend
```

🟧 = in cache / log

🟪 = in registers

🟨 = in thread-private, cachable main memory

Transaction Stack



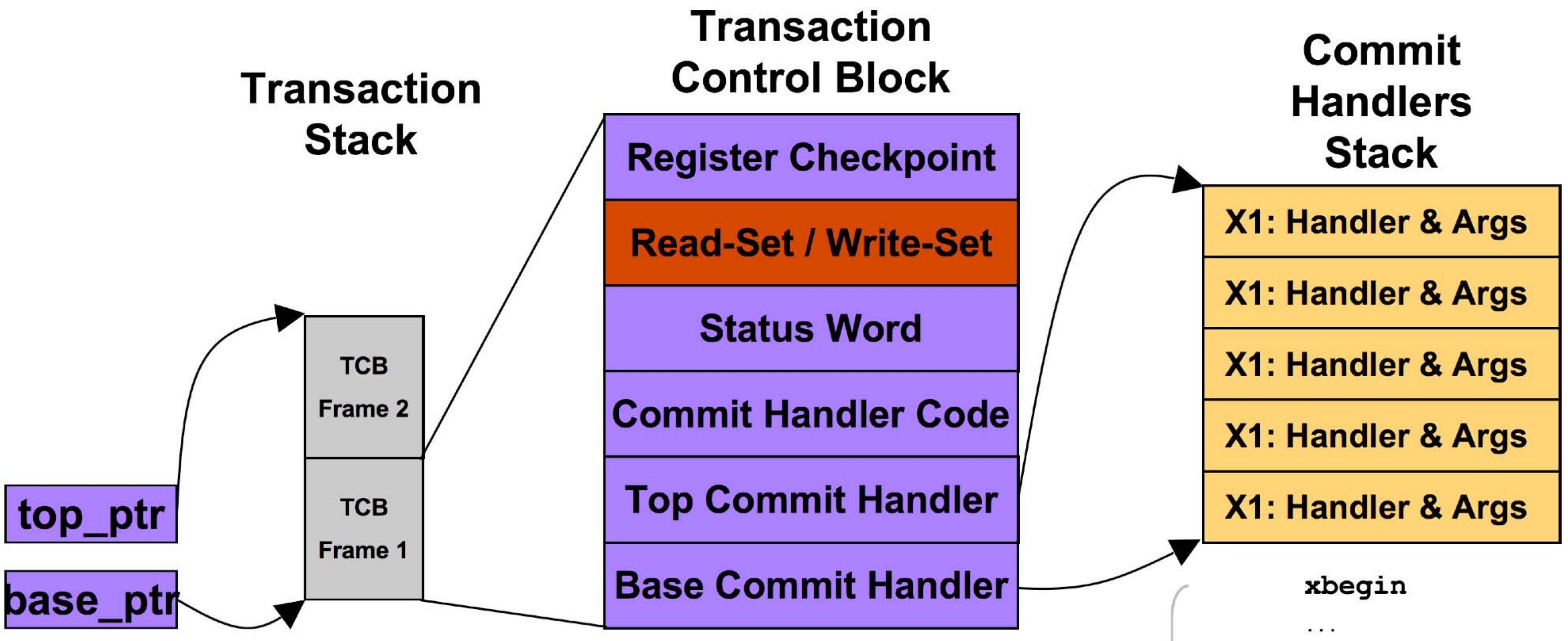
```
          xbegin
          ...
              xbegin
                  ...
                  xbegin
X1         X2  X3   ...
                  xend
              xend
          xend
```

🟧 = in cache / log

🟪 = in registers

🟨 = in thread-private,
cachable main memory

Transaction Stack



Transaction Stack

Transaction Control Block

Commit Handlers Stack

top_ptr

base_ptr

TCB Frame 2

TCB Frame 1

Register Checkpoint

Read-Set / Write-Set

Status Word

Commit Handler Code

Top Commit Handler

Base Commit Handler

X1: Handler & Args

X1: Handler & Args

X1: Handler & Args

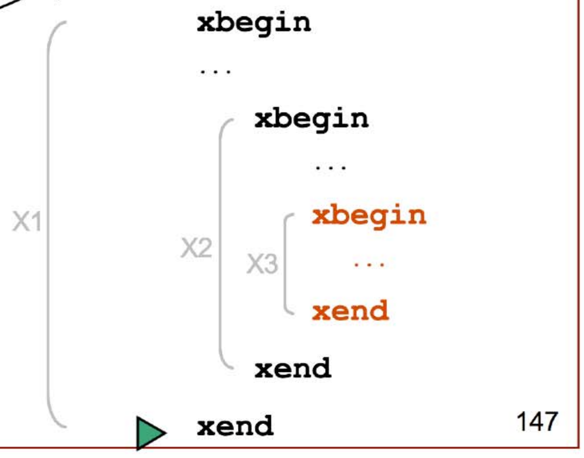
X1: Handler & Args

X1: Handler & Args

🟧 = in cache / log

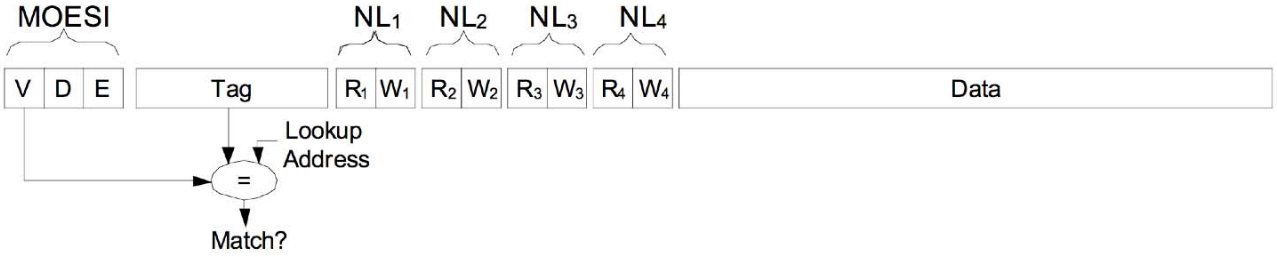
🟪 = in registers

🟨 = in thread-private,
cachable main memory

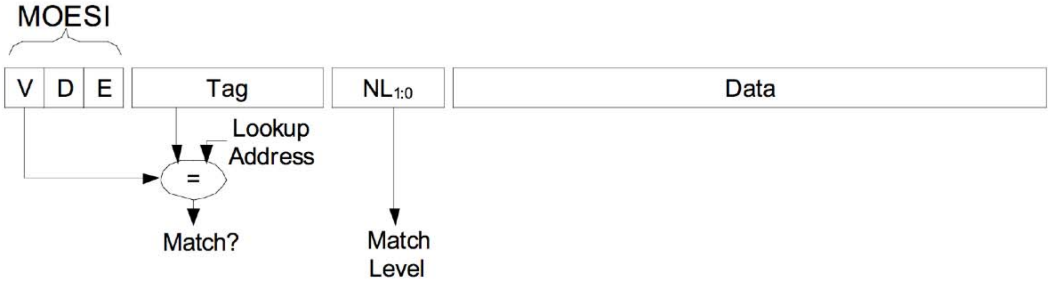


```
xbegin
...
    xbegin
      ...
        xbegin
          ...
        xend
    xend
xend
```

HW Support for Nested Read-Sets & Write-Sets



(a)



(b)

- Two Options: multi-tracking (a) Vs. associativity-based (b)
  - Differences in cost of searching, committing, and merging
  - Multi-tracking best with eager versioning, associativity best with lazy
  - Both schemes benefit from lazy merging on commit
- Need virtualization to handle overflow of nesting levels

Example Use: Transactional I/O

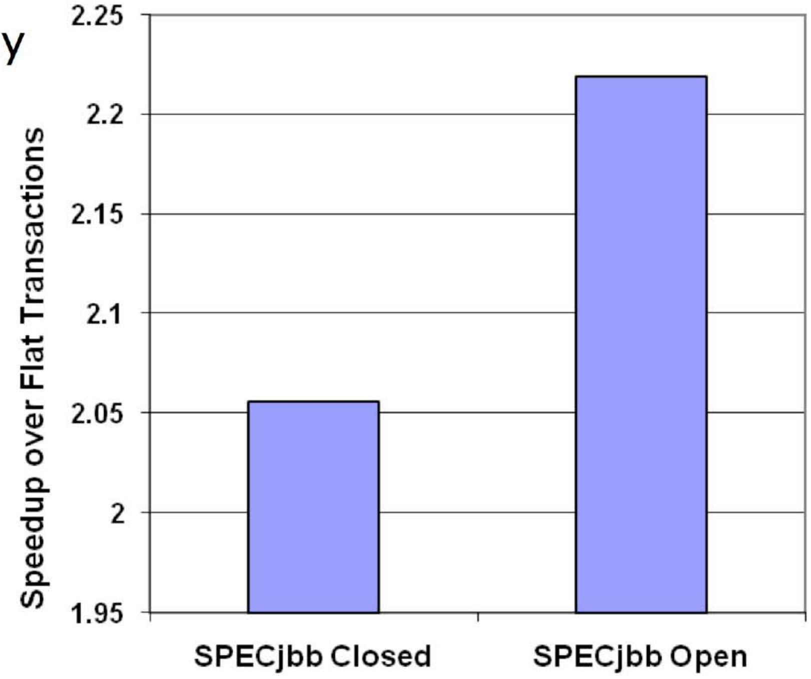
```
xbegin
 write(buf, len):
  register violation handler to de-alloc tmpBuf
  alloc tmpBuf
  cpy tmpBuf <- buf

  push &tmpBuf, len; commit handler stack
  push _writeCode; commit handler stack

xvalidate
 pop _writeCode and args
 run _writeCode
xcommit
```

Example Use: Performance Tuning

- Single warehouse SPECjbb2000
  - One transaction per task
    - Order, payment, status, …
  - Irregular code with lots of concurrency

- Speedup on an 8-way TM CMP

- Closed nesting: speedup 3.94
  - Nesting around B-tree updates to reduce conflict cost
  - 2.0x over flattening
- Open nesting: speedup 4.25
  - For unique order ID generation to reduce number of violations
  - 2.2x over flattening



Example Use: Conditional Synchronization with Retry

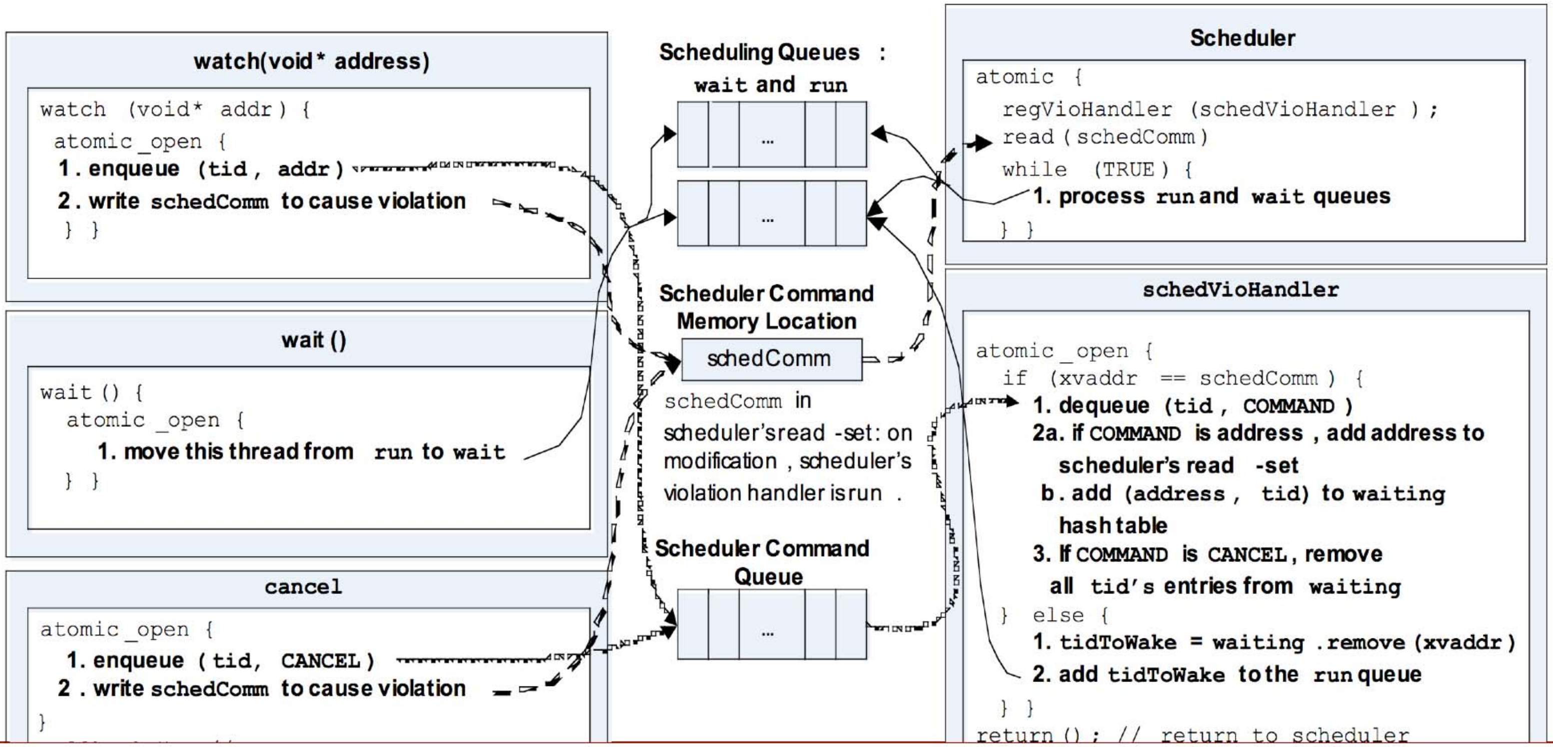
- Runtime system for Atomos' watch() and retry() constructs

Consumer:

```
atomic {
  regVioHandler (cancel);
  if (!available ) {
    watch (&available );
    wait (); }
  available  = false;
  consume (); }
```

Producer:

```
atomic {
  regVioHandler (cancel);
  if (available ) {
    watch (&available );
    wait (); }
  available  = true;
  produce (); }
```



watch(void* address)

```
watch (void* addr) {
 atomic_open {
  1. enqueue (tid, addr)
  2. write schedComm to cause violation
 } }
```

wait ()

```
wait () {
  atomic _open {
     1. move this thread from  run to wait
  } }
```

cancel

```
atomic_open {
   1. enqueue ( tid,  CANCEL )
  2 . write schedComm  to cause violation
}
```

Scheduler Command Memory Location

schedComm in scheduler's read -set: on modification, scheduler's violation handler is run .

Scheduler Command Queue

Scheduler

```
atomic {
  regVioHandler (schedVioHandler );
  read ( schedComm )
  while  (TRUE ) {
    1. process run and wait queues
  } }
```

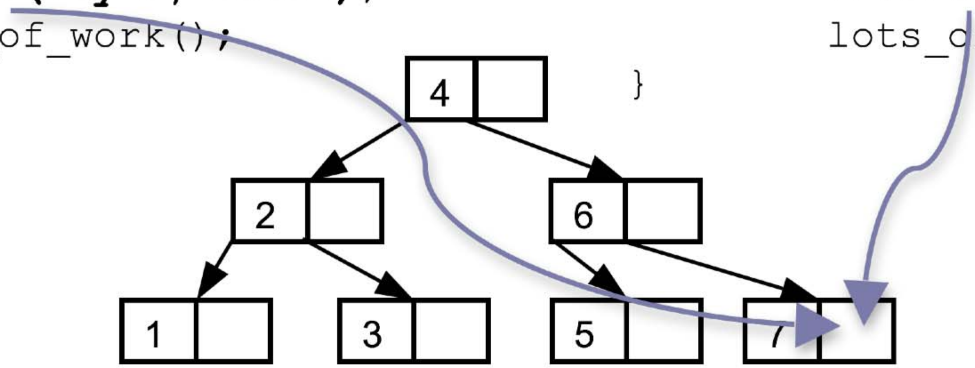
schedVioHandler

```
atomic _open {
  if  (xvaddr  == schedComm ) {
    1. dequeue (tid, COMMAND )
    2a. if COMMAND is address , add address to
        scheduler's read  -set
     b. add (address ,  tid) to waiting
        hash table
    3. If COMMAND  is CANCEL, remove
       all tid's entries from waiting
  } else {
    1. tidToWake = waiting .remove (xvaddr )
    2. add tidToWake  to the run queue
  } }
return () ; // return to scheduler
```

Example Use: Semantic Concurrency Control

```
Thread 1:
atomic{
        lots_of_work();
        insert(key=8, data1);
        lots_of_work();
}
```

```
Thread 2:
atomic{
        lots_of_work();
        insert(key=9, data2);
        lots_of_work();
}
```

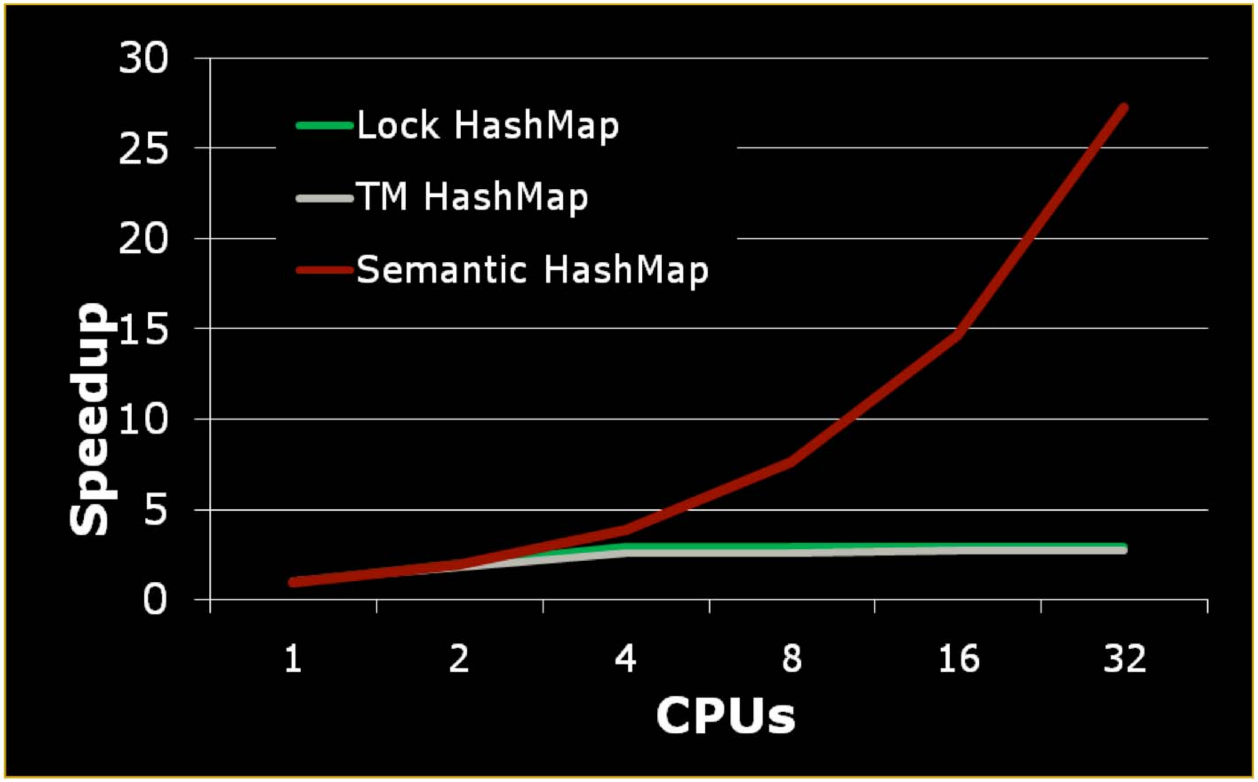


- Is there a conflict?
  - TM: yes, W-W conflict on a memory location
  - App logic: no, operation on different keys
- Common performance loss in TM programs
  - Large, compound transactions

Example Use: Semantic Concurrency Control

- Semantic concurrency in Atomos [PLDI'06]
  - From memory to semantic dependencies
  - Similar to multi-level transactions from DBs
- Transactional collection classes [PPoPP'06]
  - Read ops track semantic dependencies
    - Using open nested transactions
  - Write ops deferred until commit
    - Using open nested transactions
  - Commit handler checks for semantic conflicts
  - Commit handler performs write ops
  - Commit/abort handlers clear dependencies

Example Use: Semantic Concurrency Control



- TestCompound
  - Long transaction with 2 map operations
  - Semantic concurrency ⇒scalable performance

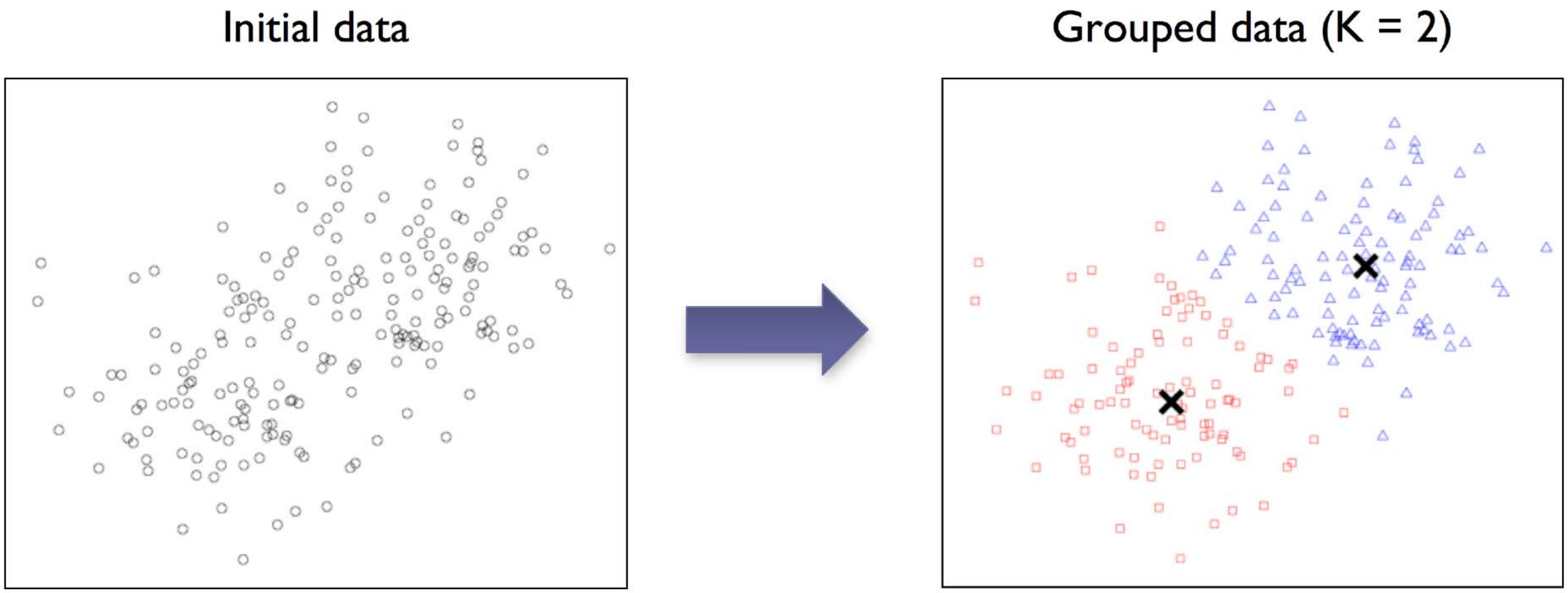
Questions?

Example Applications: STAMP

Application	Domain	Description
bayes	Machine learning	Learns structure of a Bayesian network
genome	Bioinformatics	Performs gene sequencing
intruder	Security	Detects network intrusions
kmeans	Data mining	Implements K-means clustering
labyrinth	Engineering	Routes paths in maze
ssca2	Scientific	Creates efficient graph representation
vacation	Online transaction processing	Emulates travel reservation system
yada	Scientific	Refines a Delaunay mesh

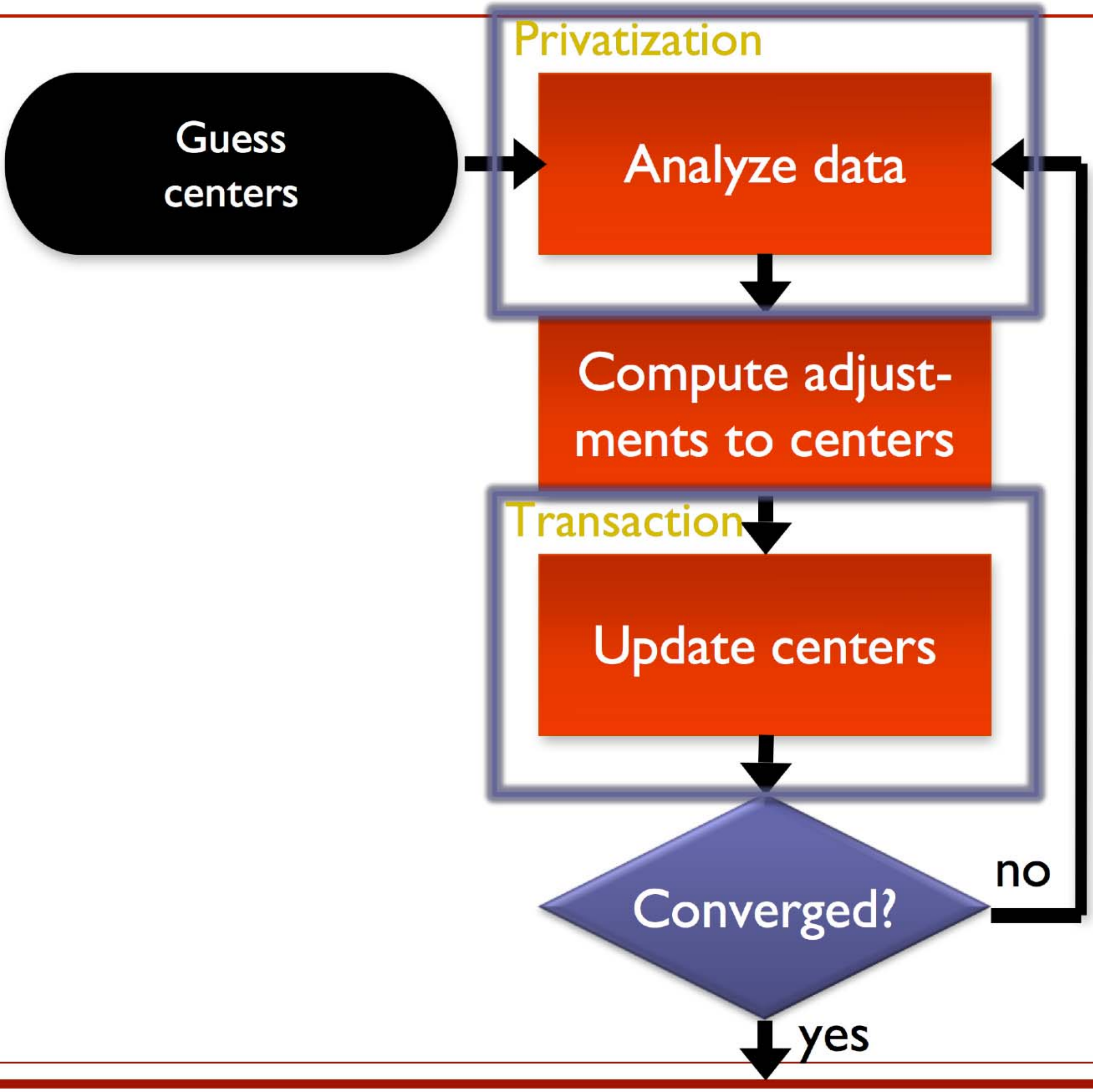
Kmeans Description

- Groups data into K clusters



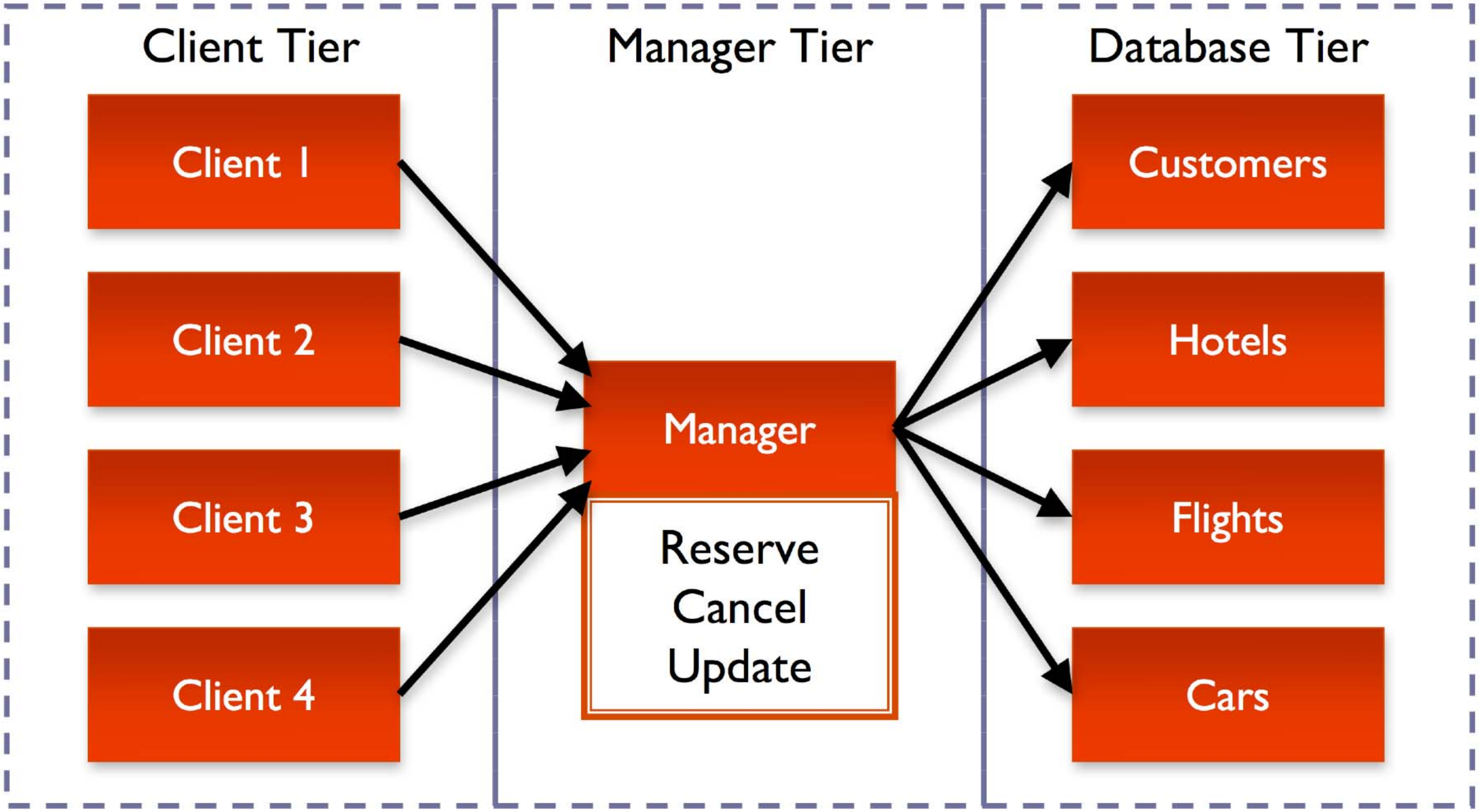
- Possible applications:
  - *Biology:* plant and animal classification
  - *WWW:* analyze web traffic for patterns

Kmeans Algorithm

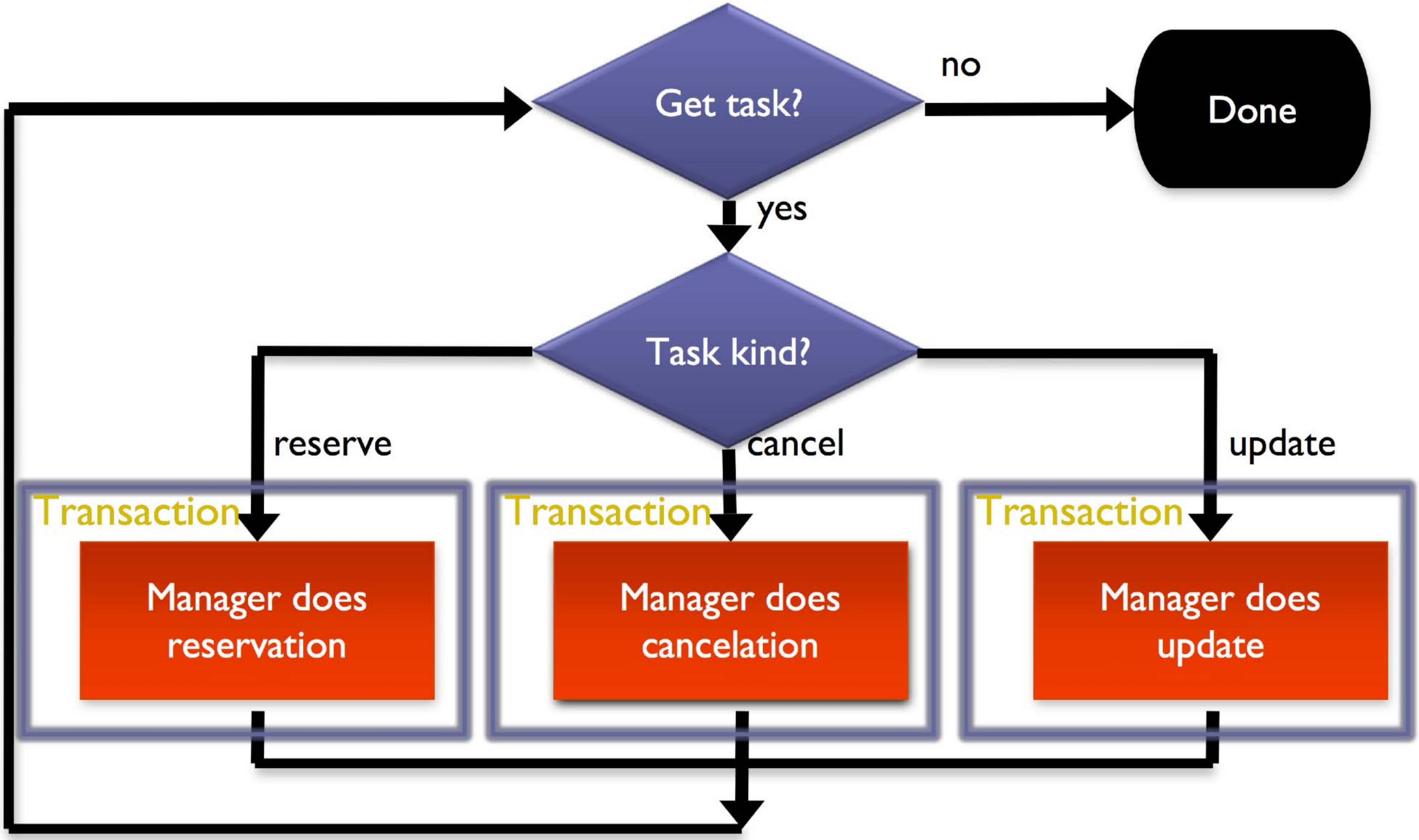


Vacation Description

- Emulates travel reservation system
  - Similar to 3-tier design in SPECjbb2000



Vacation Algorithm



STAMP Characterization

Application	Per Transaction				Time in Transactions
	Instructions	Reads	Writes	Retries	
bayes	60584	24	9	0.59	83%
genome	1717	32	2	0.14	97%
intruder	330	71	16	3.54	33%
kmeans	153	25	25	0.81	3%
labyrinth	219571	35	36	0.94	100%
ssca2	50	1	2	0.00	17%
vacation	3161	401	8	0.02	92%
yada	9795	256	108	2.51	100%

Questions?

- STAMP available at http://stamp.stanford.edu
  - Code for HTM/STM, datasets, configs…
  - Performance results for STM, HTM, hybrids

TM Uses Beyond Concurrency Control

- TM hardware consists of
  - Memory versioning HW
  - Fine-grain access tracking HW
  - HW to enforcing ordering
  - Fast exception handlers
- Motivation for using TM beyond concurrency control
  - Amortize hardware cost
  - Provide additional benefits for HW vendors and system users
  - Concurrency is not the only important problem in computing
    - Security, fault-tolerance, debugging, …
- Challenges
  - Potential mismatch of interfaces
  - Co-existence of transactions with other uses

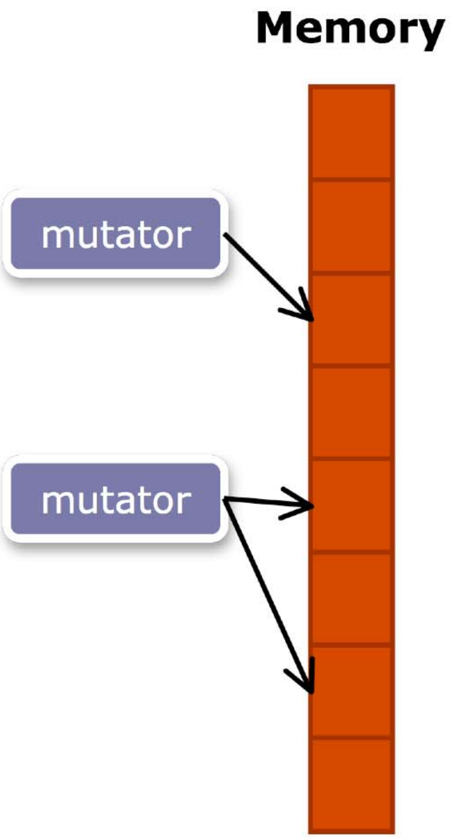
Applying TM Hardware

- Availability
  - Global & local checkpoints (versioning, order)
- Security
  - Fine-grain read/write barriers (tracking)
  - Isolated execution (versioning)
  - Thread-safe dynamic binary translation (all)
- Debugging
  - Deterministic replay (order)
  - Parallel step-back (versioning)
  - Infinite, fast watchpoints (tracking)
  - Atomicity violation detectors (tracking, order)
  - Performance tuning tools (tracking)
- Snapshot-based services (versioning)
  - Concurrent garbage collector
  - Dynamic memory profiler
  - User-level copy-on-write

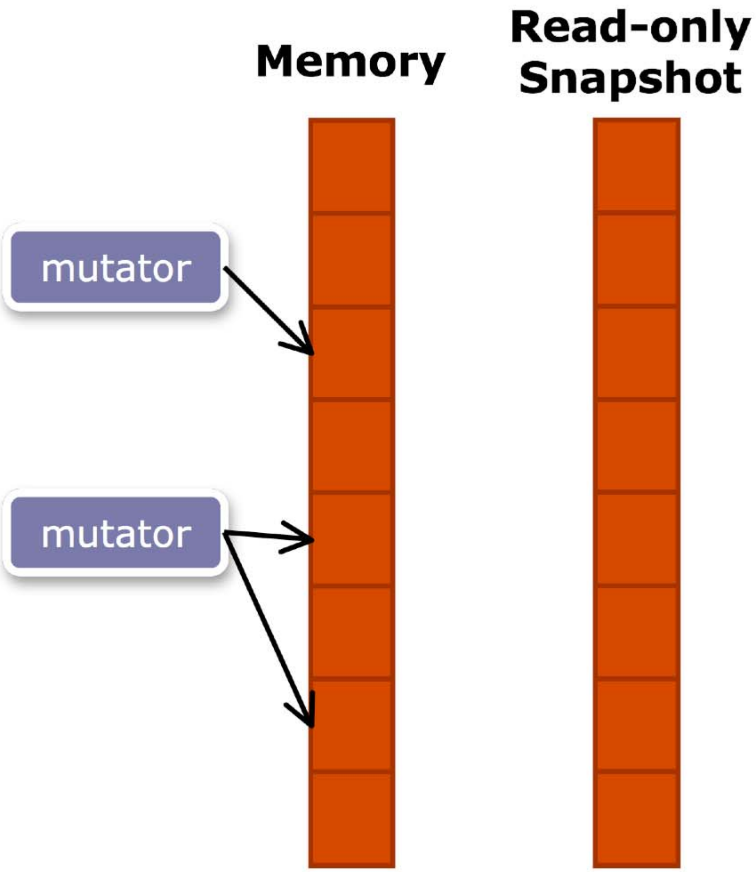
TM Vs. Other System Approaches

- Alternative implementation techniques
  - Virtual memory system: versioning & tracking at page granularity
  - Dynamic binary translation (DBT): custom SW instrumentation
- Potential advantages of TM
  - Finer granularity tracking (compared to page-based)
  - User-level handling (compared to OS handling))
  - No instrumentation overhead (compared to BDT)
  - Automatic handling of interactions with other programs/tools
- Note
  - Conflict detection accuracy matters for several applications
  - Can combine TM with alternative implementation techniques
    - HTM for common case, other techniques for virtualization or higher accuracy

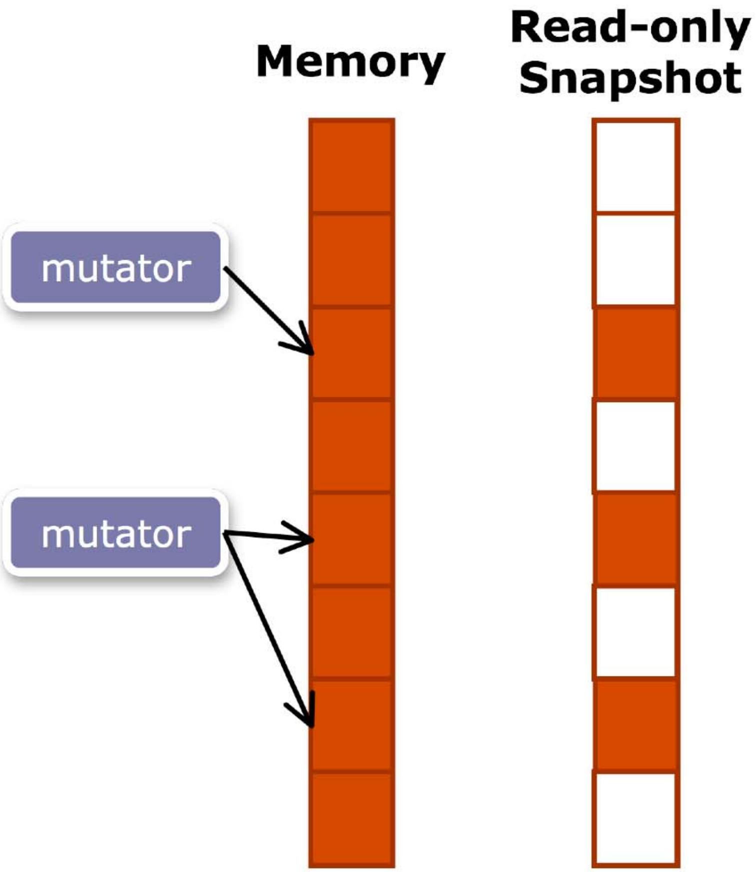
Example Use: Memory Snapshot



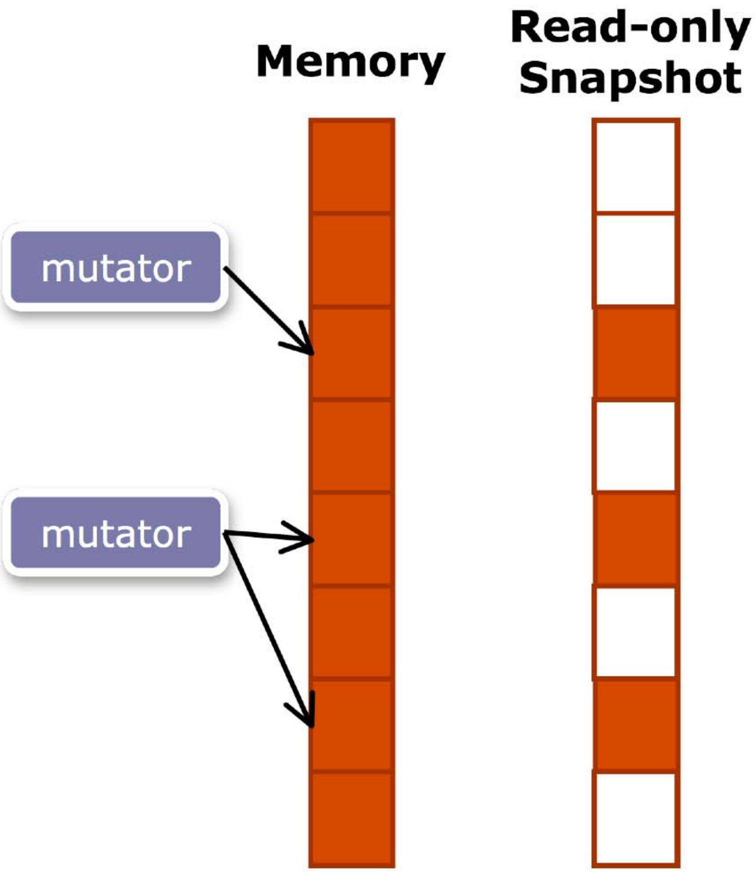
Example Use: Memory Snapshot



Example Use: Memory Snapshot

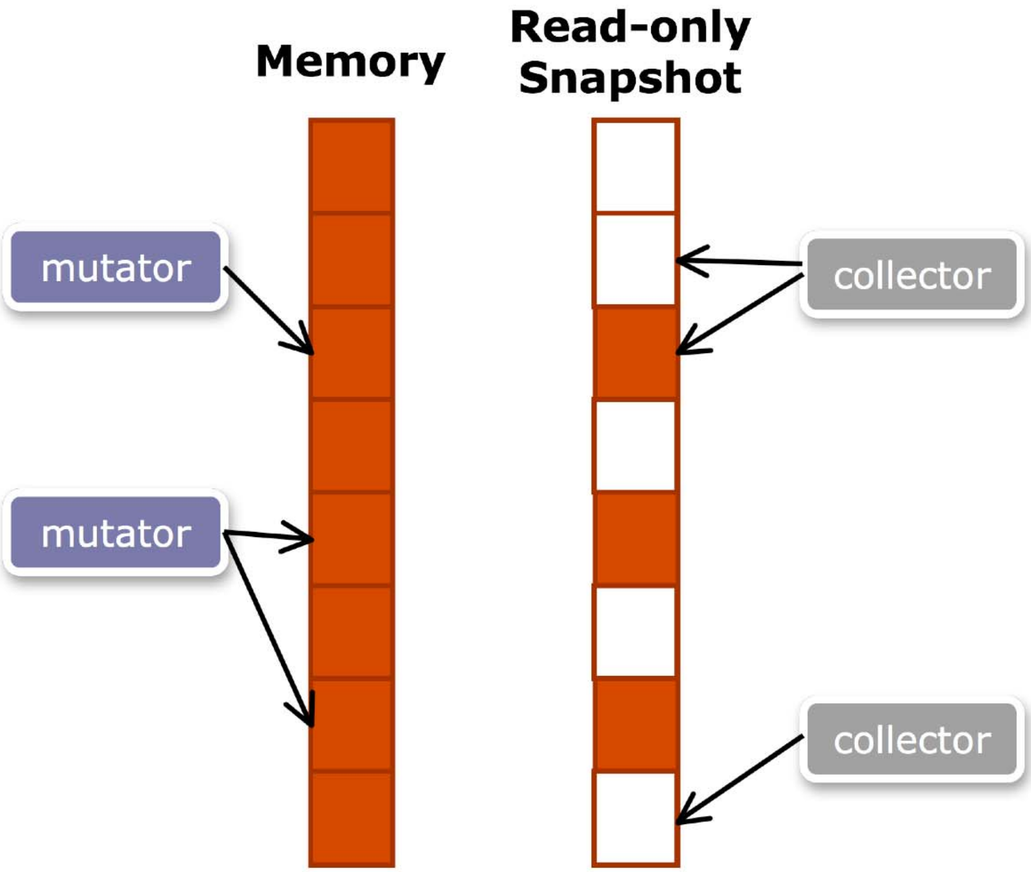


Example Use: Memory Snapshot



- Snapshot
  - Read-only image
  - Multiple regions
  - Access by ≥ 1 threads

Example Use: Memory Snapshot



- Snapshot
  - Read-only image
  - Multiple regions
  - Access by ≥ 1 threads
- Applications
  - Service threads that analyze memory in parallel with app threads
  - Garbage collection, heap & stack analysis, copy on write, …

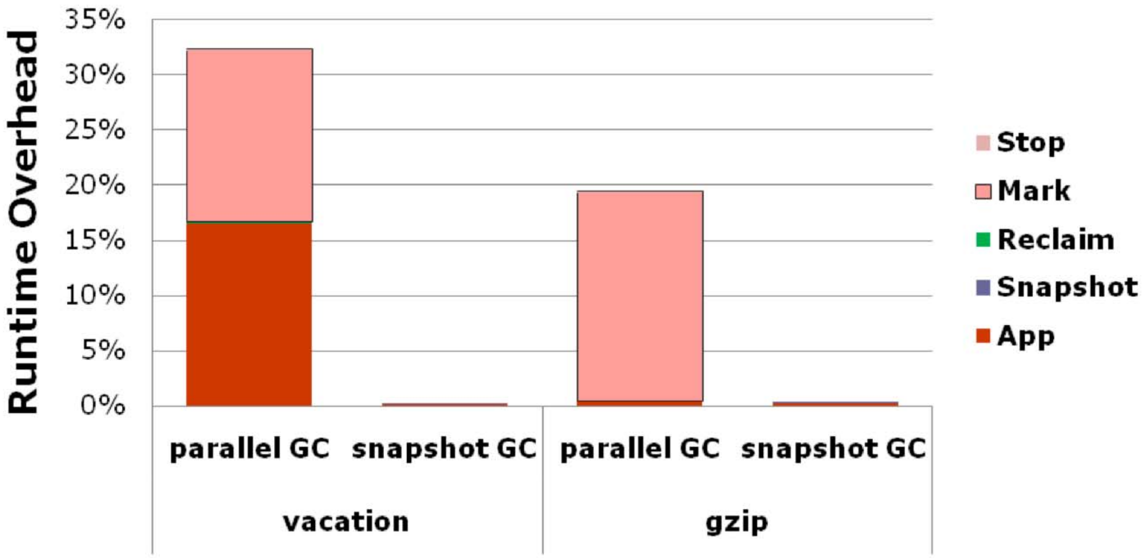
TM Hardware ⇒Snapshot

- Feature correspondence
  - TM metadata ⇒track data written since or read from snapshot
  - TM versioning ⇒storage for progressive snapshot
    - Including virtualization mechanism
  - TM conflict detection ⇒catch errors
    - Writes to read-only snapshot

- Differences & additions
  - Single-thread Vs. multithread versioning
  - Table to describe snapshot regions

- Resulting snapshot system
  - Scan (create) snapshot in O(# CPUs)
  - Update (write) and read in O(1)
  - Memory overhead up to O(# memory locations written)

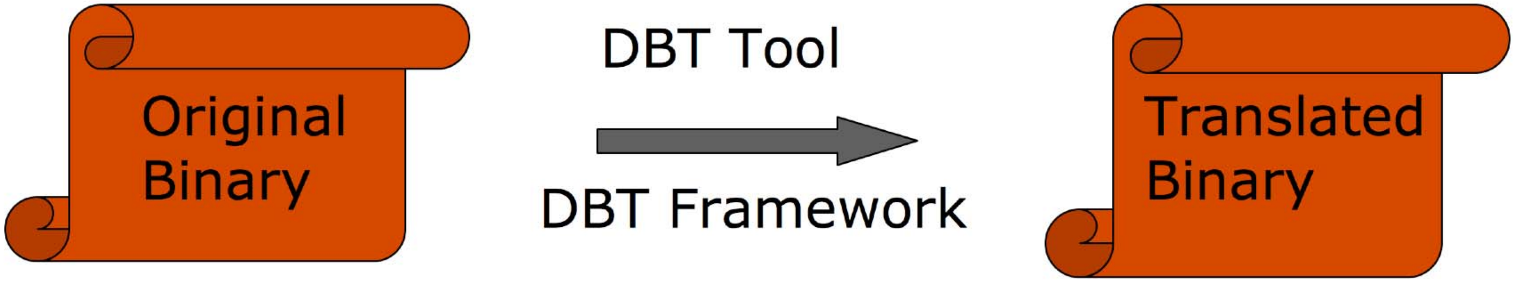
GC Overhead



- Parallel GC: stop app threads & run GC threads
  - 20% to 30% overhead for memory intensive apps
- Snapshot GC ⇒GC is essentially free
  - Stop app, take snapshot, then run GC & app concurrently
- Snapshot GC ⇒fast & simple
  - +100 lines over simple sequential GC by Boehm
  - Fundamentally simpler than any other concurrent GC

Example Use : Dynamic Binary Translation

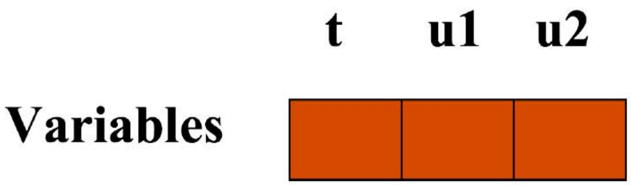
- DBT
  - Short code sequence is translated in run-time
  - PIN, Valgrind, DynamoRIO, StarDBT, etc



- DBT use cases
  - Translation on new target architecture
  - JIT optimizations in virtual machines
  - Binary instrumentation
    - Profiling, security, debugging, …

DBT Use: Dynamic Information Flow Tracking (DIFT)

```
t = XX ; // untrusted data from network

…….

swap t, u1;

u2 = u1;
```



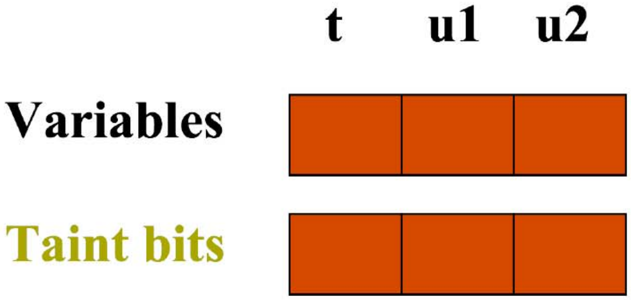
t u1 u2

Variables

- Untrusted data are tracked throughout execution
  - A taint bit per memory byte is used to track untrusted data.
  - Security policy uses the taint bit.
    - E.g. untrusted data should not be used as syscall argument.
- Dynamic instrumentation to propagates and checks taint bits

DBT Use: Dynamic Information Flow Tracking (DIFT)

```
t = XX ; // untrusted data from network
taint(t) = 1;
…….

swap t, u1;
swap taint(t), taint(u1);
u2 = u1;
taint(u2) = taint(u1);
```



- Untrusted data are tracked throughout execution
  - A taint bit per memory byte is used to track untrusted data.
  - Security policy uses the taint bit.
    - E.g. untrusted data should not be used as syscall argument.
- Dynamic instrumentation to propagates and checks taint bits

DBT Use: Dynamic Information Flow Tracking (DIFT)

```
t = XX ; // untrusted data from network
taint(t) = 1;
.......

swap t, u1;
swap taint(t), taint(u1);
u2 = u1;
taint(u2) = taint(u1);
```

	t	u1	u2
Variables	XX		
Taint bits			

- Untrusted data are tracked throughout execution
  - A taint bit per memory byte is used to track untrusted data.
  - Security policy uses the taint bit.
    - E.g. untrusted data should not be used as syscall argument.
- Dynamic instrumentation to propagates and checks taint bits

DBT Use: Dynamic Information Flow Tracking (DIFT)

```
   t = XX ; // untrusted data from network
➡ taint(t) = 1;
   …….

   swap t, u1;
   swap taint(t), taint(u1);
   u2 = u1;
   taint(u2) = taint(u1);
```

	t	u1	u2
Variables	XX		
Taint bits	1		

- Untrusted data are tracked throughout execution
  - A taint bit per memory byte is used to track untrusted data.
  - Security policy uses the taint bit.
    - E.g. untrusted data should not be used as syscall argument.
- Dynamic instrumentation to propagates and checks taint bits

DBT Use: Dynamic Information Flow Tracking (DIFT)

```
t = XX ; // untrusted data from network
taint(t) = 1;
…….

swap t, u1;
swap taint(t), taint(u1);
u2 = u1;
taint(u2) = taint(u1);
```

	t	u1	u2
Variables		XX	
Taint bits	1		

- Untrusted data are tracked throughout execution
  - A taint bit per memory byte is used to track untrusted data.
  - Security policy uses the taint bit.
    - E.g. untrusted data should not be used as syscall argument.
- Dynamic instrumentation to propagates and checks taint bits

DBT Use: Dynamic Information Flow Tracking (DIFT)

```
   t = XX ; // untrusted data from network
   taint(t) = 1;
   …….

   swap t, u1;
➡  swap taint(t), taint(u1);
   u2 = u1;
   taint(u2) = taint(u1);
```

	t	u1	u2
Variables		XX	
Taint bits		1	

- Untrusted data are tracked throughout execution
  - A taint bit per memory byte is used to track untrusted data.
  - Security policy uses the taint bit.
    - E.g. untrusted data should not be used as syscall argument.
- Dynamic instrumentation to propagates and checks taint bits

DBT Use: Dynamic Information Flow Tracking (DIFT)

```
t = XX ; // untrusted data from network
taint(t) = 1;
…….

swap t, u1;
swap taint(t), taint(u1);
→ u2 = u1;
taint(u2) = taint(u1);
```

	t	u1	u2
Variables		XX	XX
Taint bits		1	

- Untrusted data are tracked throughout execution
  - A taint bit per memory byte is used to track untrusted data.
  - Security policy uses the taint bit.
    - E.g. untrusted data should not be used as syscall argument.
- Dynamic instrumentation to propagates and checks taint bits

DBT Use: Dynamic Information Flow Tracking (DIFT)

```
    t = XX ; // untrusted data from network
    taint(t) = 1;
    .......

    swap t, u1;
    swap taint(t), taint(u1);
    u2 = u1;
➡   taint(u2) = taint(u1);
```

	t	u1	u2
Variables		XX	XX
Taint bits		1	1

- Untrusted data are tracked throughout execution
  - A taint bit per memory byte is used to track untrusted data.
  - Security policy uses the taint bit.
    - E.g. untrusted data should not be used as syscall argument.
- Dynamic instrumentation to propagates and checks taint bits

DBT & Multithreading

- DBT with multithreaded executables as input

- Challenges
  - Atomicity of target instructions
    - E.g. compare-and-exchange
  - Atomicity of additional instrumentation
    - Races in accesses to application data & DBT metadata

- Easy but unsatisfactory solutions
  - Do not allow multithreaded programs (StarDBT)
  - Serialize multithreaded execution (Valgrind)

Example MetaData Race ⇒Security Breach

- User code uses atomic instructions
    - After instrumentation, there are races on taint bits

```
Thread 1                 Thread2

swap t, u1;

                         u2 = u1;
```

	t	u1	u2
Variables	XX		

Example MetaData Race ⇒Security Breach

- User code uses atomic instructions
  - After instrumentation, there are races on taint bits

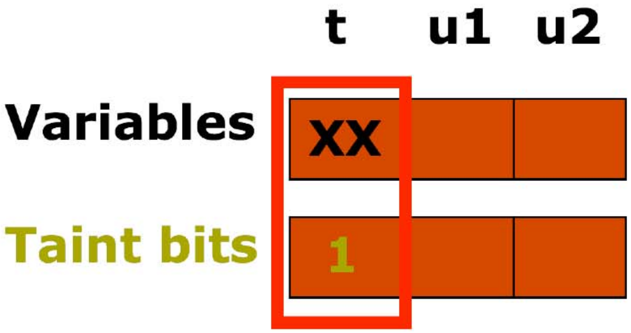
```
Thread 1                        Thread2

swap t, u1;
                                u2 = u1;

                                taint(u2) = taint(u1);

swap taint(t), taint(u1);
```

	t	u1	u2
Variables	XX		
Taint bits	1		

Example MetaData Race ⇒Security Breach

- User code uses atomic instructions
  - After instrumentation, there are races on taint bits

```
Thread 1                      Thread2

swap t, u1;
                              u2 = u1;

                              taint(u2) = taint(u1);

swap taint(t), taint(u1);
```



	t	u1	u2
Variables	XX		
Taint bits	1		

Example MetaData Race ⇒Security Breach

- User code uses atomic instructions
  - After instrumentation, there are races on taint bits

```
    Thread 1                          Thread2

  ➡ swap t, u1;
                                      u2 = u1;

                                      taint(u2) = taint(u1);

    swap taint(t), taint(u1);
```

	t	u1	u2
Variables		XX	
Taint bits	1		

Example MetaData Race ⇒Security Breach

- User code uses atomic instructions
  - After instrumentation, there are races on taint bits

```
Thread 1                          Thread2

swap t, u1;
                                  u2 = u1;

                                  taint(u2) = taint(u1);

swap taint(t), taint(u1);
```

	t	u1	u2
Variables		XX	XX
Taint bits	1		

Example MetaData Race ⇒Security Breach

- User code uses atomic instructions
  - After instrumentation, there are races on taint bits

```
Thread 1                          Thread2

swap t, u1;
                                  u2 = u1;

                                  taint(u2) = taint(u1); ⬅
swap taint(t), taint(u1);
```

	t	u1	u2
Variables		XX	XX
Taint bits	1		

Example MetaData Race ⇒Security Breach

- User code uses atomic instructions
  - After instrumentation, there are races on taint bits

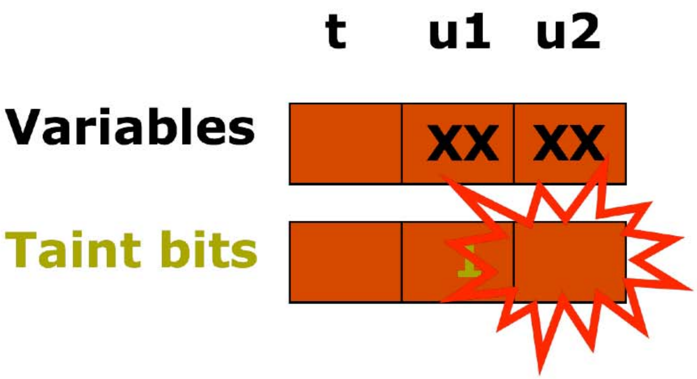
```
     Thread 1                           Thread2

     swap t, u1;
                                        u2 = u1;

                                        taint(u2) = taint(u1);

➡    swap taint(t), taint(u1);
```

	t	u1	u2
Variables		XX	XX
Taint bits		1	

Example MetaData Race ⇒Security Breach

- User code uses atomic instructions
  - After instrumentation, there are races on taint bits

```
Thread 1                          Thread2

swap t, u1;
                                  u2 = u1;

                                  taint(u2) = taint(u1);

➡ swap taint(t), taint(u1);
```



Can We Fix It with Locks?

- Idea
  - Enclose access to data & metadata within a locked region
- Problems
  - Coarse-grained locks
    - Performance degradation
  - Fine-grained locks
    - Locking overhead, convoying, limited scope of DBT optimizations
  - Lock nesting between application & DBT locks
    - Potential deadlock
  - Tool developers should be a feature + multithreading experts
    - Must know both security & multithreading to develop tool

TM for DBT

- Idea
  - DBT instruments a transaction to enclose accesses to (data, metadata) within the transaction boundary.

```
Thread 1                            Thread2



swap t, u1;                         u2 = u1;
swap taint(t), taint(u1);           taint(u2) = taint(u1);
```

- Advantages
  - Atomic execution
  - High performance through optimistic concurrency
  - Support for nested transactions
  - Hidden from the tool and application developers

TM for DBT

- Idea
  - DBT instruments a transaction to enclose accesses to (data, metadata) within the transaction boundary.

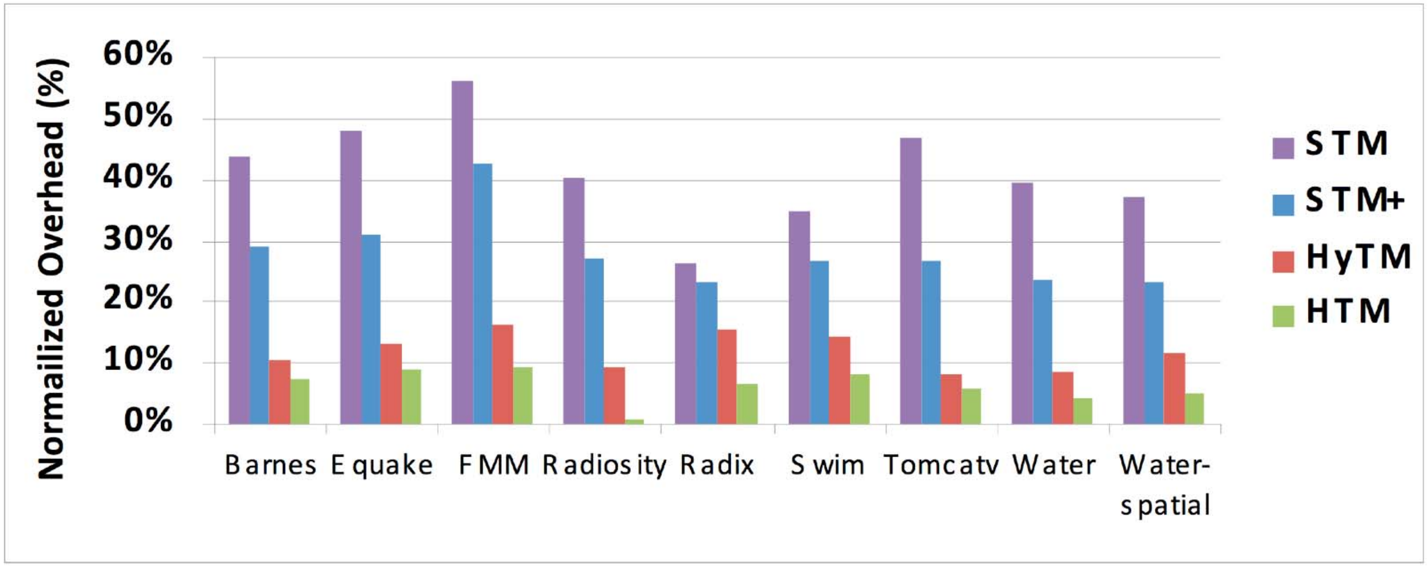
```
Thread 1                          Thread2

TX_Begin                          TX_Begin
swap t, u1;                       u2 = u1;
swap taint(t), taint(u1);         taint(u2) = taint(u1);
TX_End                            TX_End
```

- Advantages
  - Atomic execution
  - High performance through optimistic concurrency
  - Support for nested transactions
  - Hidden from the tool and application developers

Granularity of Transaction Instrumentation

- Per instruction
  - High overhead of executing TX_Begin and TX_End
  - Limited scope for DBT optimizations
- Per basic block
  - Amortizing the TX_Begin and TX_End overhead
  - Easy to match TX_Begin and TX_End
- Per trace
  - Further amortization of the overhead
  - Potentially high transaction conflict
- Profile-based sizing
  - Optimize transaction size based on transaction abort ratio

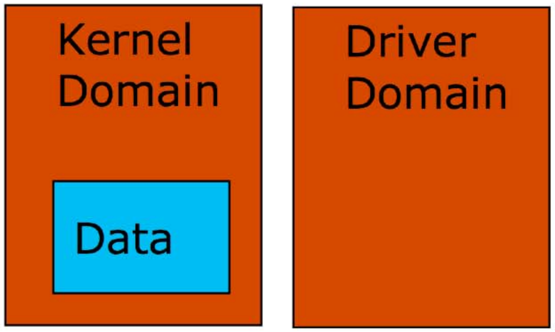
Performance Overheads



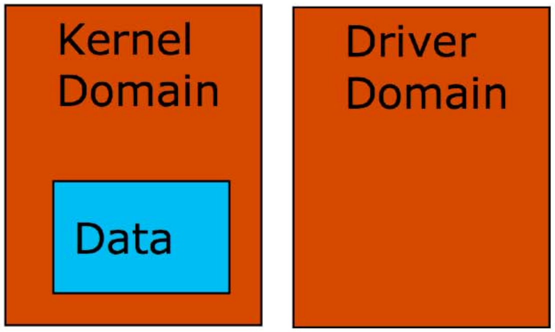
- TM systems evaluated
  - STM: software TM, STM+ = STM + HW checkpointing
  - HyTM: hardware-accelerated TM (similar to SigTM)
  - HTM: full hardware TM implementation

Example Use: Reliable Systems

- Kernel protection
  - Faulty drivers can corrupt kernel data
- Protection through domain isolation
  - Kernel data are copied to driver
    - RPC likes operation
  - If no fault occurs, modified data copied back to kernel space
- Use of TM
  - Replace copying with atomic block
  - If fault occurs, abort transaction



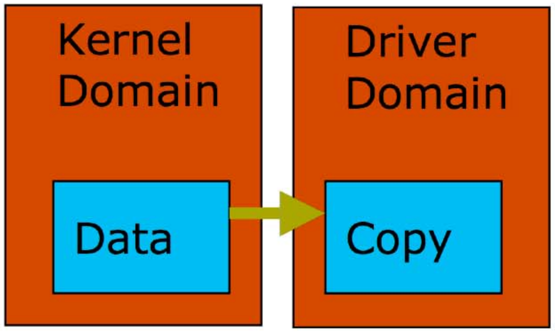
< RPC-based approach>



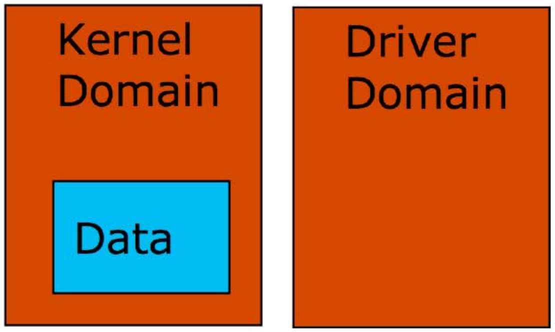
< TM-based style >

Example Use: Reliable Systems

- Kernel protection
  - Faulty drivers can corrupt kernel data
- Protection through domain isolation
  - Kernel data are copied to driver
    - RPC likes operation
  - If no fault occurs, modified data copied back to kernel space
- Use of TM
  - Replace copying with atomic block
  - If fault occurs, abort transaction



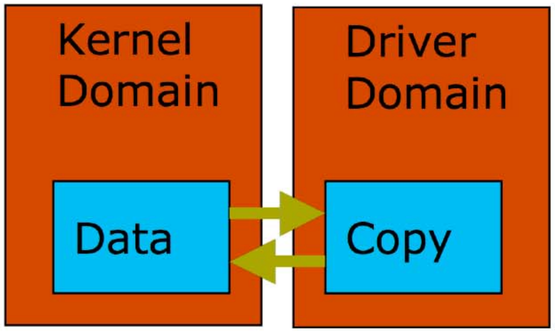
< RPC-based approach>



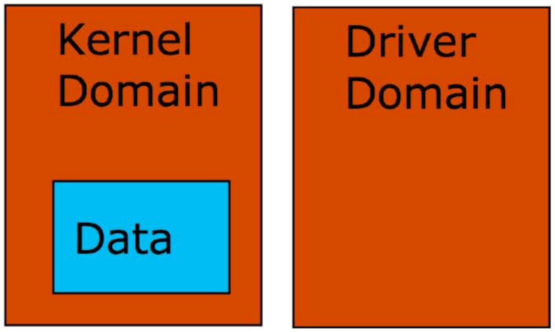
< TM-based style >

Example Use: Reliable Systems

- Kernel protection
  - Faulty drivers can corrupt kernel data
- Protection through domain isolation
  - Kernel data are copied to driver
    - RPC likes operation
  - If no fault occurs, modified data copied back to kernel space
- Use of TM
  - Replace copying with atomic block
  - If fault occurs, abort transaction



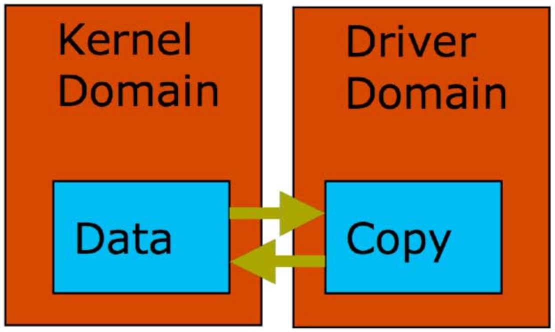
< RPC-based approach>



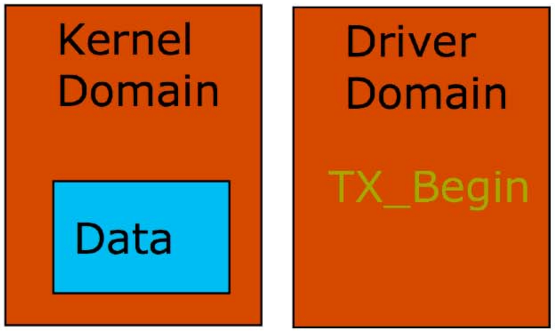
< TM-based style >

Example Use: Reliable Systems

- Kernel protection
  - Faulty drivers can corrupt kernel data
- Protection through domain isolation
  - Kernel data are copied to driver
    - RPC likes operation
  - If no fault occurs, modified data copied back to kernel space
- Use of TM
  - Replace copying with atomic block
  - If fault occurs, abort transaction



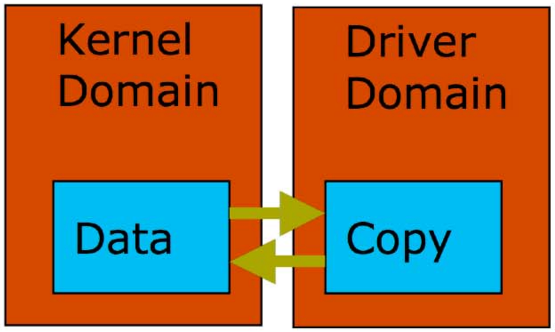
< RPC-based approach>



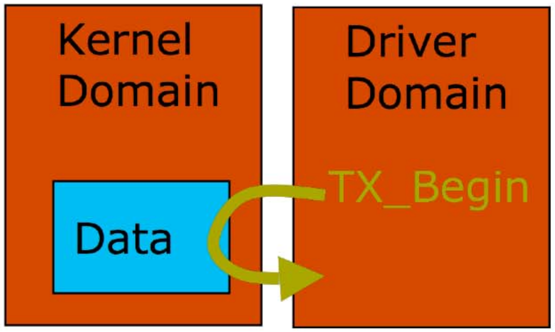
< TM-based style >

Example Use: Reliable Systems

- Kernel protection
  - Faulty drivers can corrupt kernel data
- Protection through domain isolation
  - Kernel data are copied to driver
    - RPC likes operation
  - If no fault occurs, modified data copied back to kernel space
- Use of TM
  - Replace copying with atomic block
  - If fault occurs, abort transaction



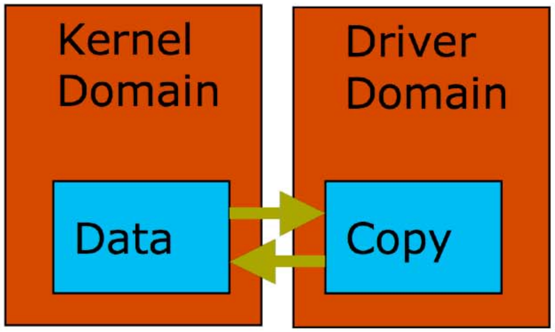
< RPC-based approach>



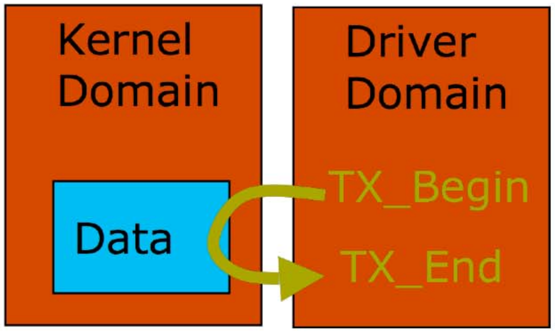
< TM-based style >

Example Use: Reliable Systems

- Kernel protection
  - Faulty drivers can corrupt kernel data
- Protection through domain isolation
  - Kernel data are copied to driver
    - RPC likes operation
  - If no fault occurs, modified data copied back to kernel space
- Use of TM
  - Replace copying with atomic block
  - If fault occurs, abort transaction



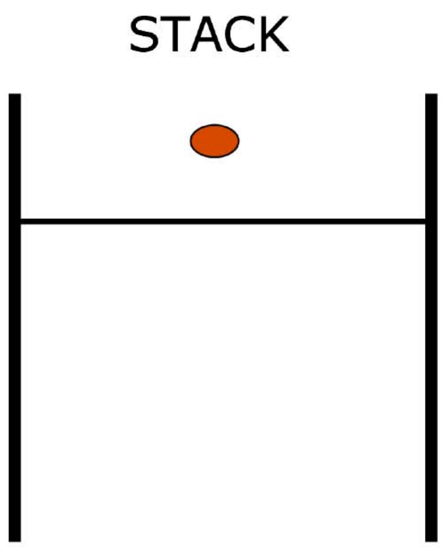
< RPC-based approach>



< TM-based style >

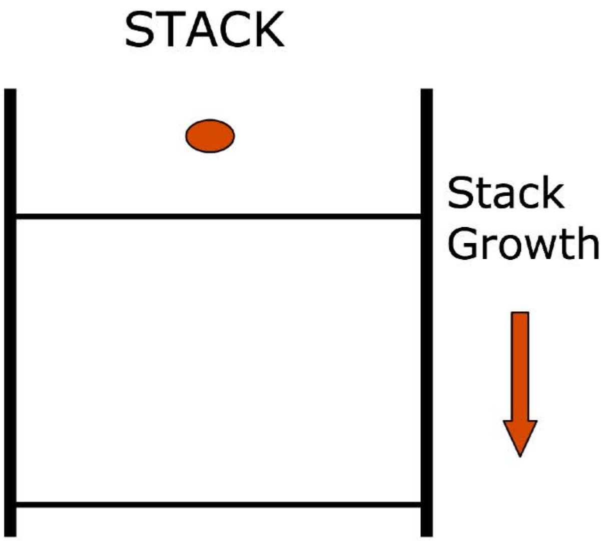
Exampled Use: Security

- Stack smashing
  - Overwrite return address using a buffer overflow
  - Can jump to arbitrary code
- Protection through canary
  - Place a special value next to the return address.
  - If the value is modified at the end of function, the return address is compromised
- Use of TM
  - Use address tracking to detect overwrites of return address
  - Lower time & space overhead



Exampled Use: Security

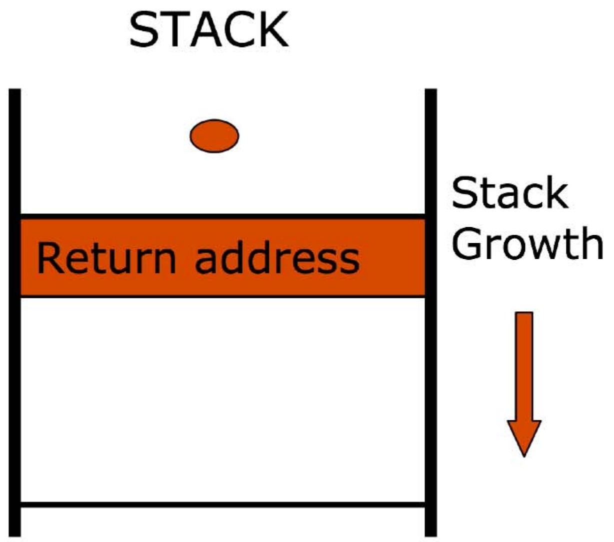
- Stack smashing
  - Overwrite return address using a buffer overflow
  - Can jump to arbitrary code



- Protection through canary
  - Place a special value next to the return address.
  - If the value is modified at the end of function, the return address is compromised
- Use of TM
  - Use address tracking to detect overwrites of return address
  - Lower time & space overhead

Exampled Use: Security

- Stack smashing
  - Overwrite return address using a buffer overflow
  - Can jump to arbitrary code
- Protection through canary
  - Place a special value next to the return address.
  - If the value is modified at the end of function, the return address is compromised
- Use of TM
  - Use address tracking to detect overwrites of return address
  - Lower time & space overhead

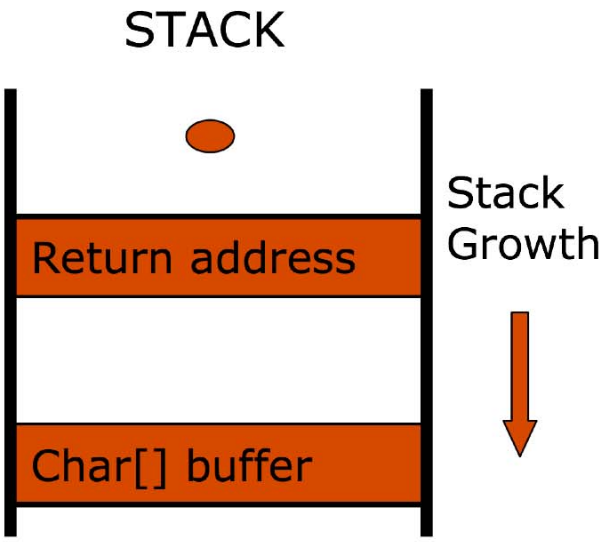


Exampled Use: Security

- Stack smashing
  - Overwrite return address using a buffer overflow
  - Can jump to arbitrary code

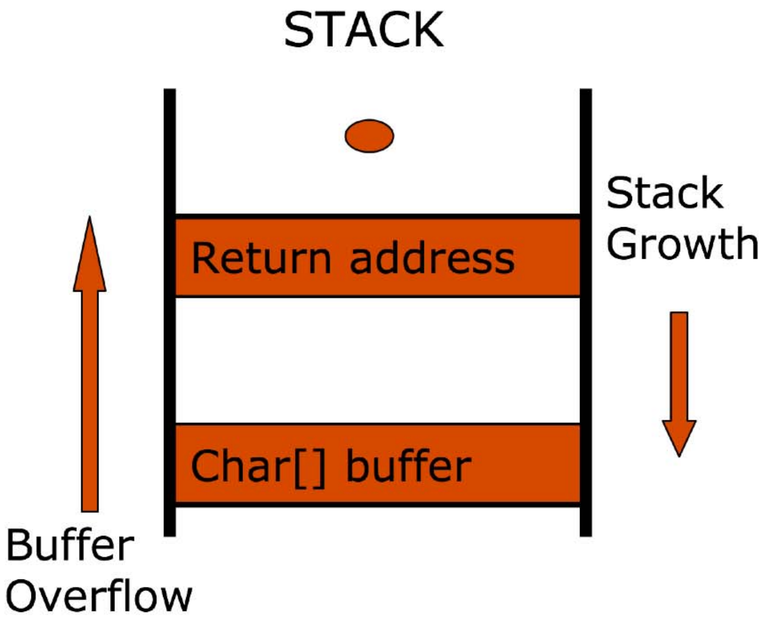
- Protection through canary
  - Place a special value next to the return address.
  - If the value is modified at the end of function, the return address is compromised

- Use of TM
  - Use address tracking to detect overwrites of return address
  - Lower time & space overhead



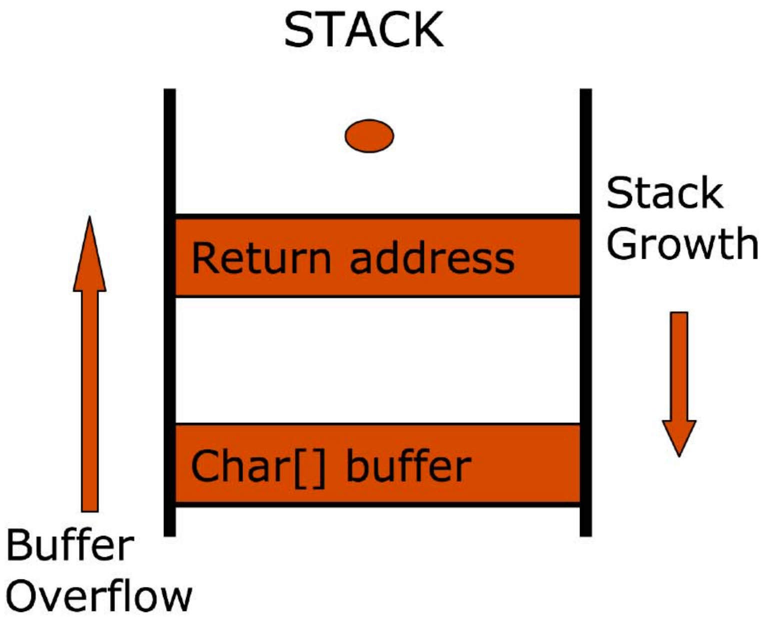
Exampled Use: Security

- Stack smashing
  - Overwrite return address using a buffer overflow
  - Can jump to arbitrary code
- Protection through canary
  - Place a special value next to the return address.
  - If the value is modified at the end of function, the return address is compromised
- Use of TM
  - Use address tracking to detect overwrites of return address
  - Lower time & space overhead



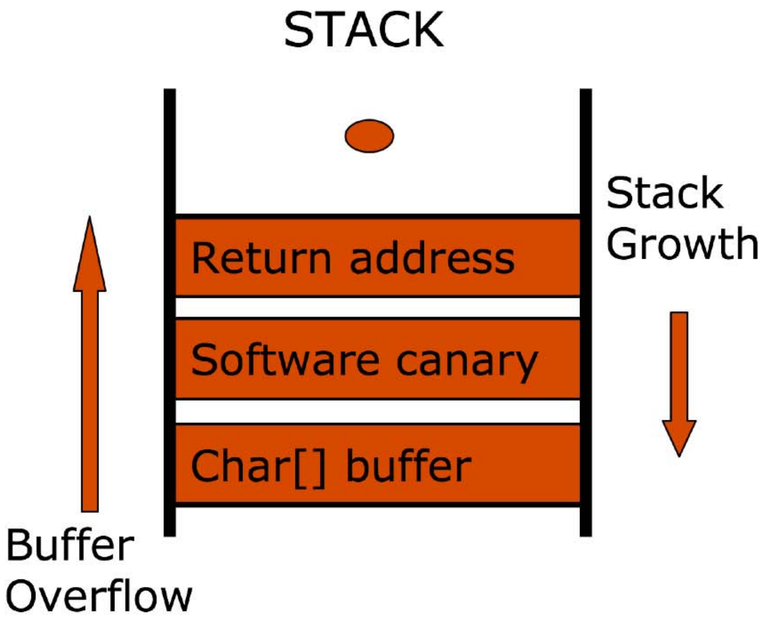
Exampled Use: Security

- Stack smashing
  - Overwrite return address using a buffer overflow
  - Can jump to arbitrary code
- Protection through canary
  - Place a special value next to the return address.
  - If the value is modified at the end of function, the return address is compromised
- Use of TM
  - Use address tracking to detect overwrites of return address
  - Lower time & space overhead



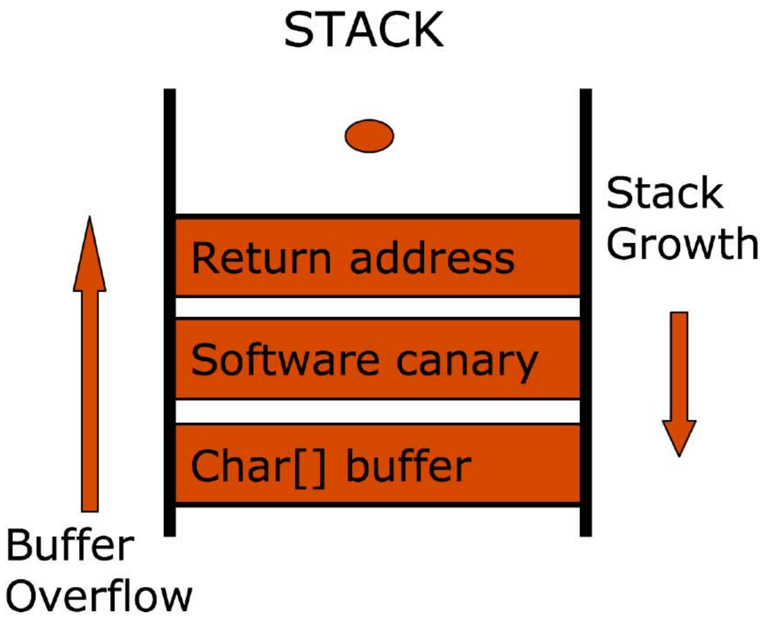
Exampled Use: Security

- Stack smashing
  - Overwrite return address using a buffer overflow
  - Can jump to arbitrary code
- Protection through canary
  - Place a special value next to the return address.
  - If the value is modified at the end of function, the return address is compromised
- Use of TM
  - Use address tracking to detect overwrites of return address
  - Lower time & space overhead



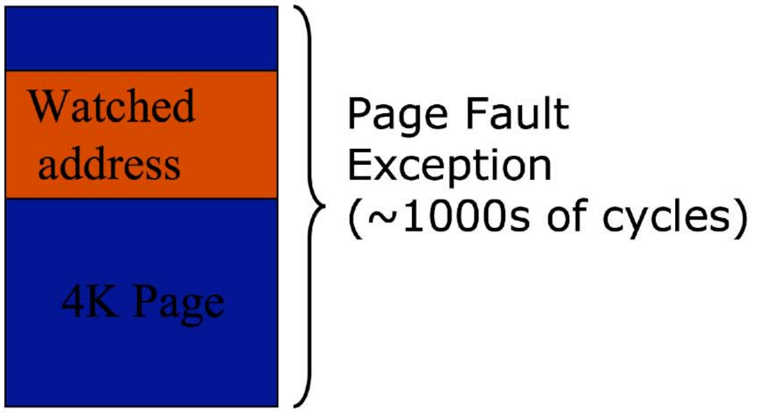
Exampled Use: Security

- Stack smashing
  - Overwrite return address using a buffer overflow
  - Can jump to arbitrary code
- Protection through canary
  - Place a special value next to the return address.
  - If the value is modified at the end of function, the return address is compromised
- Use of TM
  - Use address tracking to detect overwrites of return address
  - Lower time & space overhead



Example Use: Debugging

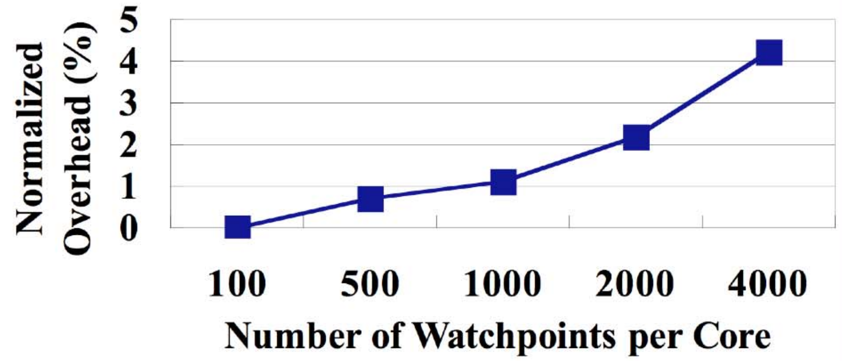
- Data watchpoint
  - Detects memory accesses
  - Triggers software handler
- Current approaches
  - Up to 4 HW watchpoints
  - Infinite watchpoints with VM
    - OS overheads
    - False positivies
- Use of TM
  - Use access tracking for watchpoints
  - Fine granularity
  - User-level overheads



Watched address

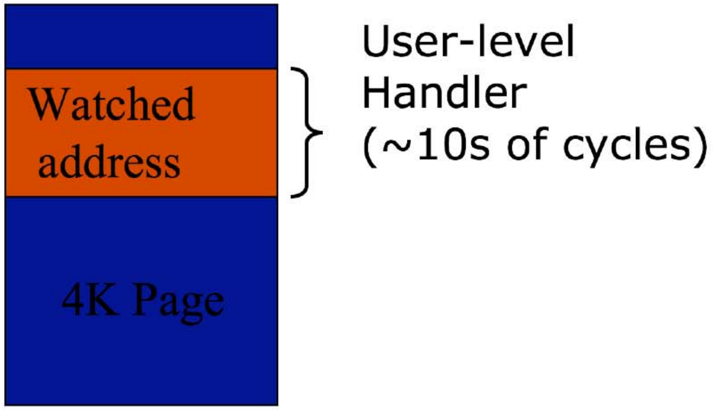
4K Page

Page Fault Exception (~1000s of cycles)



Example Use: Debugging

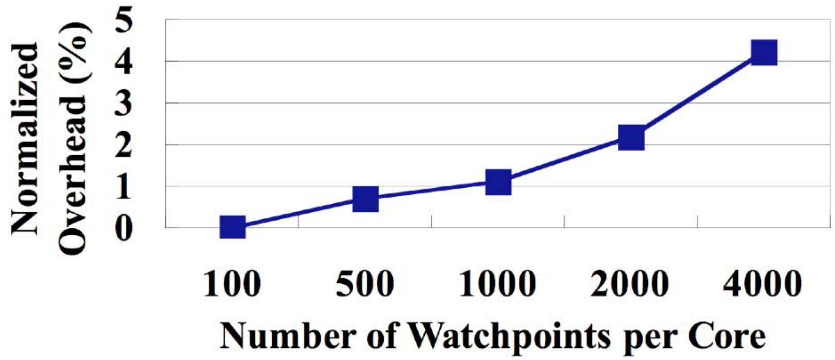
- Data watchpoint
  - Detects memory accesses
  - Triggers software handler
- Current approaches
  - Up to 4 HW watchpoints
  - Infinite watchpoints with VM
    - OS overheads
    - False positivies
- Use of TM
  - Use access tracking for watchpoints
  - Fine granularity
  - User-level overheads



Watched address

4K Page

User-level Handler (~10s of cycles)



Lecture 4: Select References

Overview

- Adl-Tabatabai. Unlocking Concurrency: Multi-core Programming with Transactional Memory, ACM Queue, 2006
- Larus & Kozyrakis. Transactional Memory, CACM, 2008

Hardware/Software Interface

- McDonald et al. Architectural Semantics for Practical Transactional Memory, ISCA, 2006
- Carlstrom et al. The Atomos Transactional Programming Language, PLDI, 2006
- Moravan et al. Supporting Nested Transactions in LogTM, ASPLOS, 2006
- Carlstrom et al. Transactional Collection Classes, PPoPP, 2007
- Ni et al. Open Nesting in Software Transactional Memory, PPoPP, 2007
- Sriraman et al. An Integrated Hardware-Software Approach to Flexible Transactional Memory, ISCA, 2007
- Baugh et al. An Analysis of I/O and Syscalls in Critical Sections and their Implications to Transactional Memory, Transact 2007

- TM uses Beyond Concurrency Control
  - Chung et al. Thread-safe Dynamic Binary Translation Using Transactional Memory, HPCA, 2008
  - Chung, System Challenges and Opportunities for Transactional Memory, PhD Thesis, 2008

Questions?

- Thank you for your attention
- For further questions or comments contact me at

  christos@ee.stanford.edu