Agenda

- Transactional Memory (TM)
  - TM Introduction
  - TM Implementation Overview
  - Hardware TM Techniques
  - Software TM Techniques
- Q&A

Transactional Memory Implementation Overview

Christos Kozyrakis

Computer Systems Laboratory
Stanford University
http://csl.stanford.edu/~christos

TM Implementation Requirements

- TM implementation 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

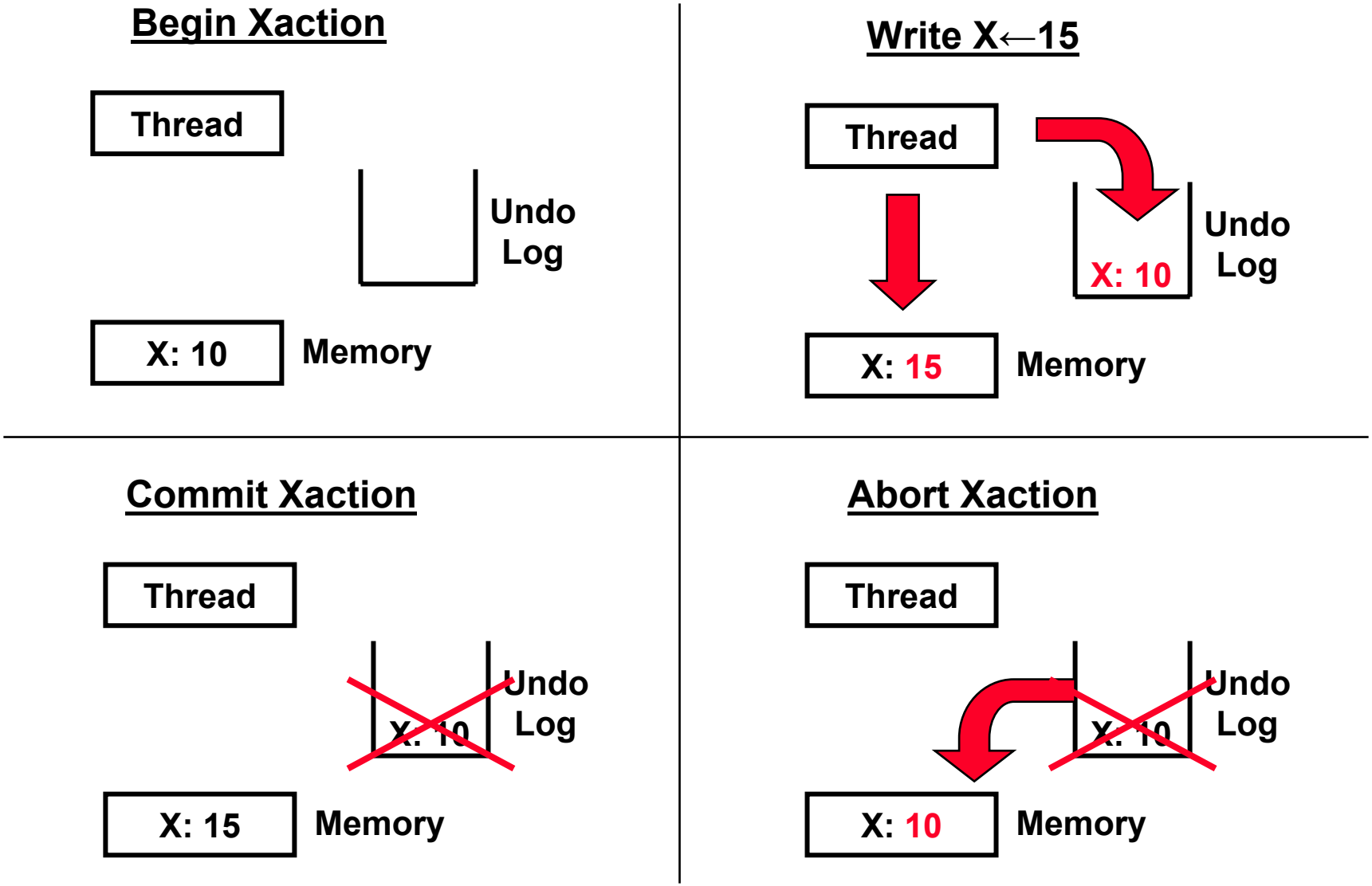
Data Versioning

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

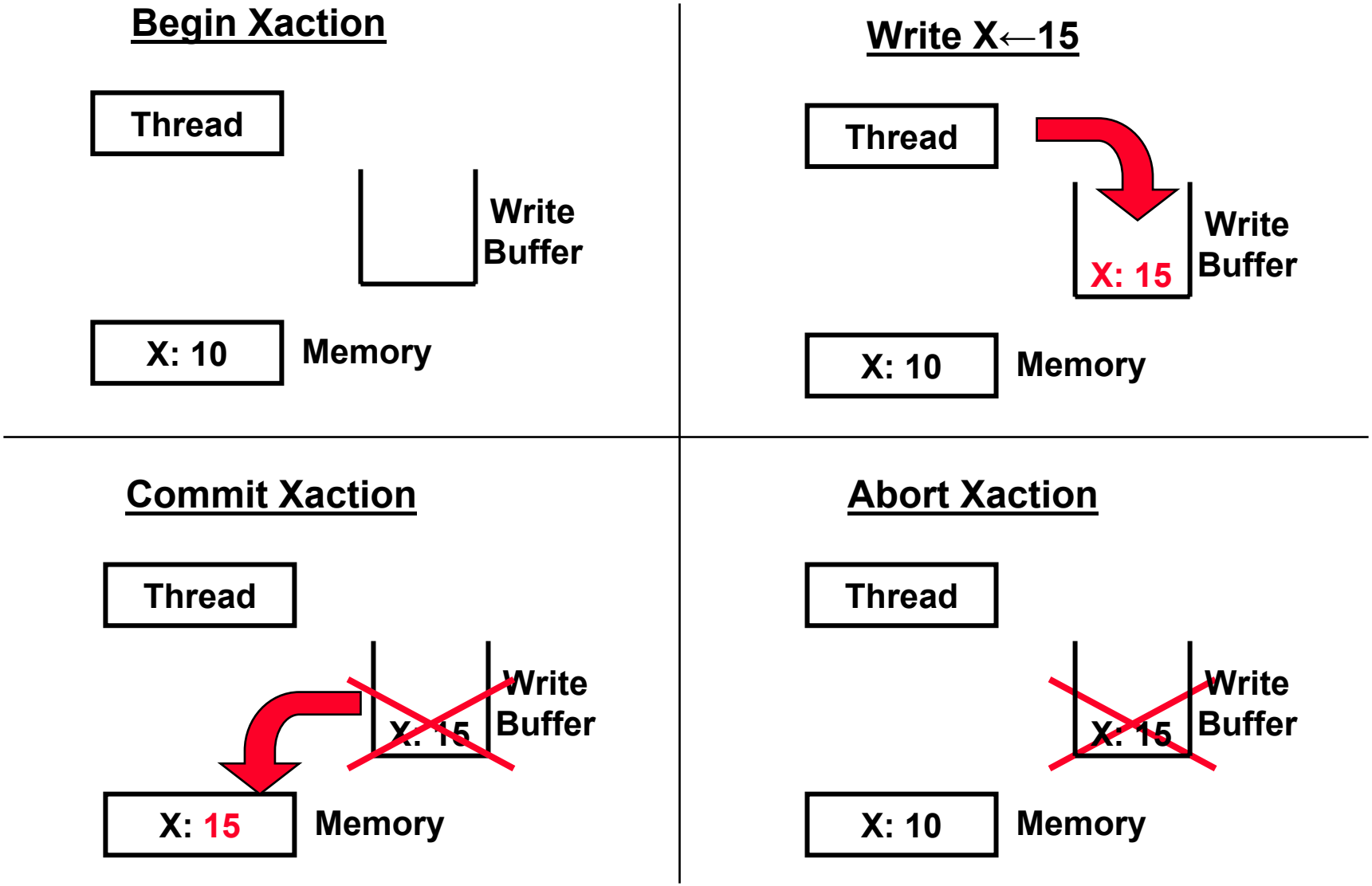
1. Eager (undo-log based)
   - Update memory location directly; maintain undo info in a log
   + Faster commit, direct reads (SW)
   – Slower aborts, no fault tolerance, weak atomicity (SW)

2. Lazy (write-buffer based)
   - Buffer writes until commit; update memory location on commit
   + Faster abort, fault tolerance, strong atomicity (SW)
   – Slower commits, indirect reads (SW)

Eager Versioning Illustration



Lazy Versioning Illustration



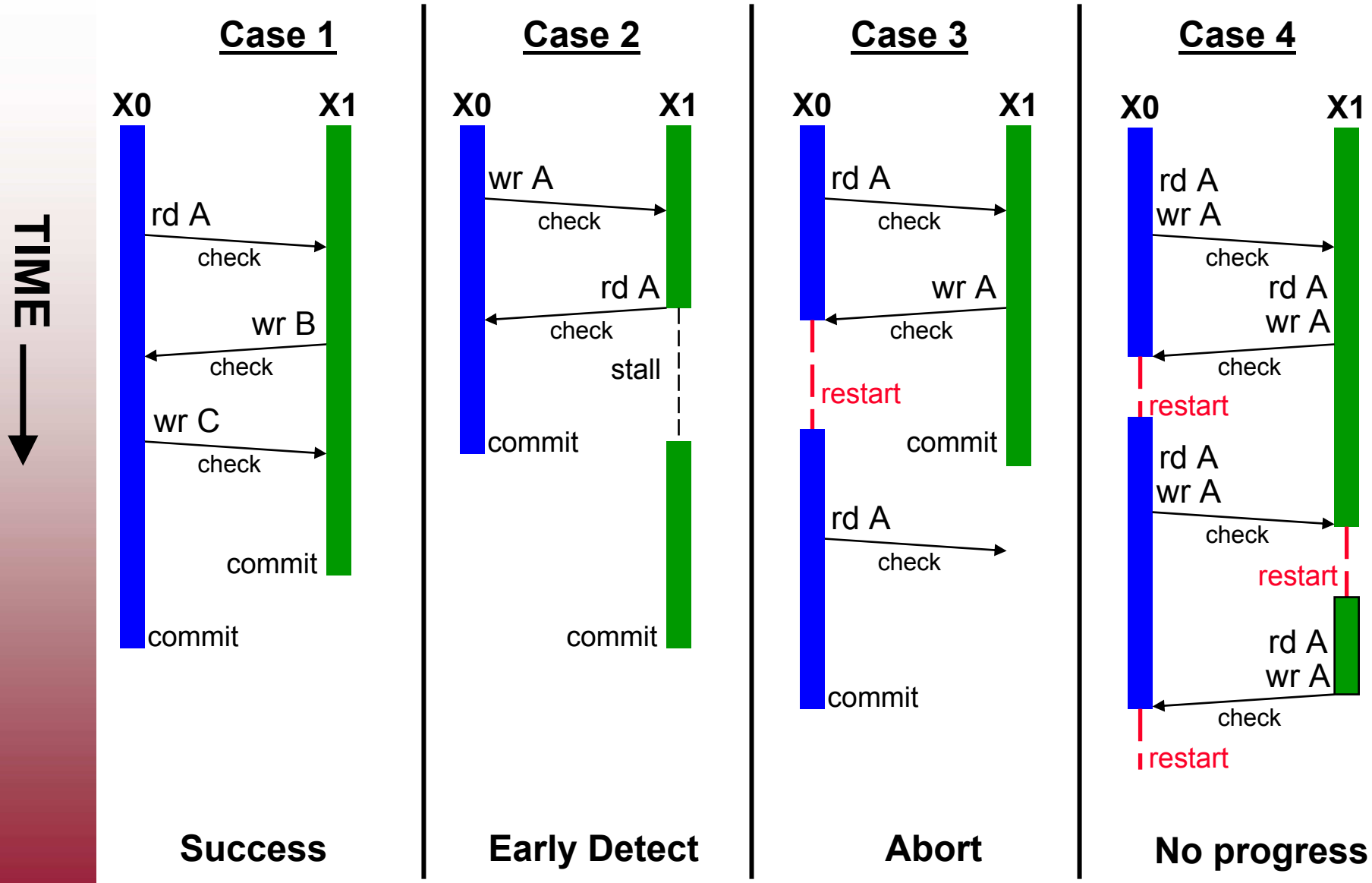
Conflict Detection

- Detect and handle conflicts between transaction
  - Read-Write and (often) Write-Write conflicts
  - For detection, a transactions tracks its read-set and write-set

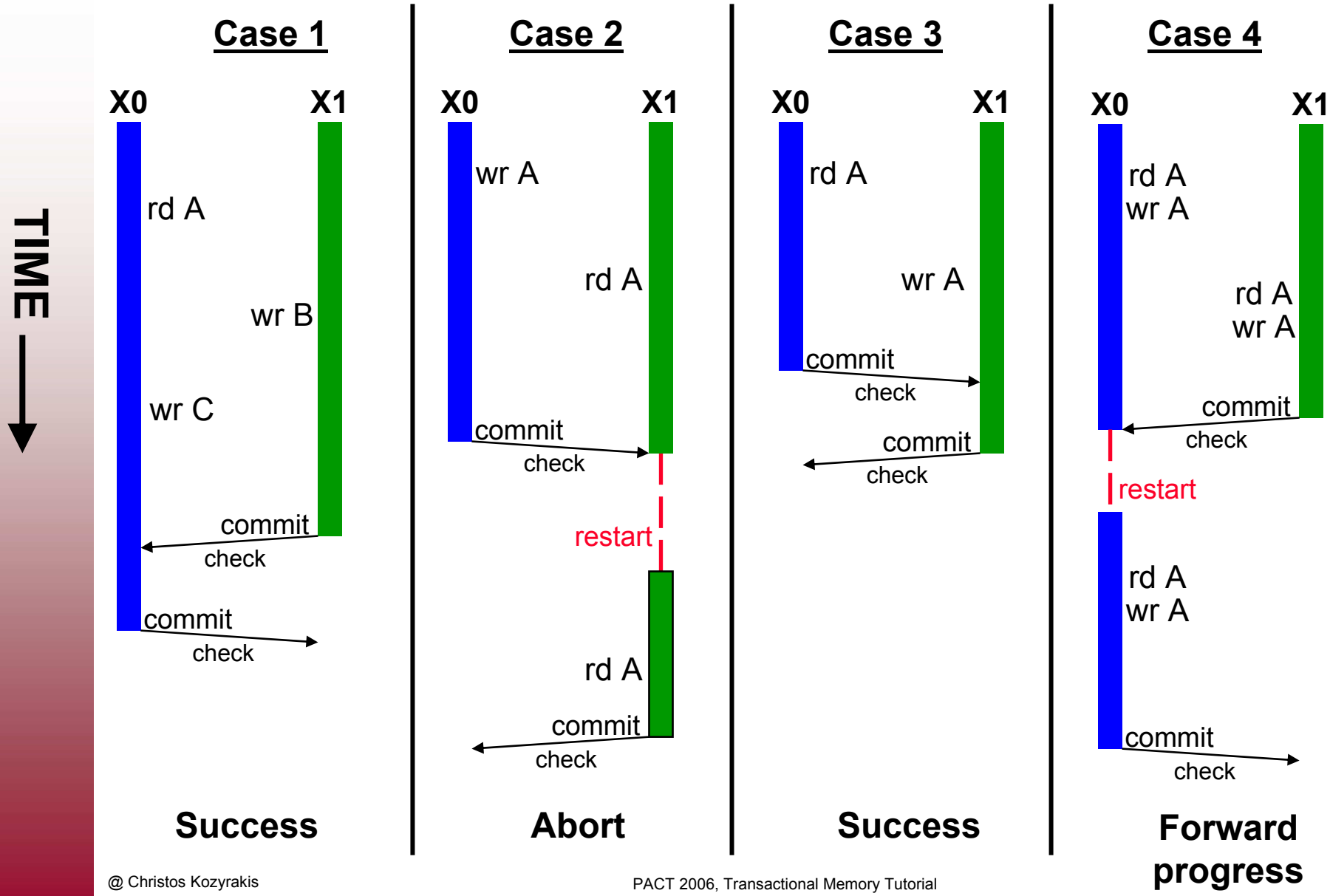
1. Pessimistic detection
   - Check for conflicts during loads or stores
     - HW: check through coherence lookups
     - SW: checks through locks and/or version numbers
   - Use contention manager to decide to stall or abort
     - Various priority policies to handle common case fast
2. Optimistic detection
   - Detect conflicts when a transaction attempts to commit
     - HW: write-set of committing transaction compared to read-set of others
       - Committing transaction succeeds; others may abort
     - SW: validate write-set and read-set using locks and version numbers

- Can use separate mechanism for loads & stores (SW)

Pessimistic Detection Illustration



Optimistic Detection Illustration



Conflict Detection Tradeoffs

1. 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)

2. Optimistic conflict detection (aka commit or lazy)
   - \+ Forward progress guarantees
   - \+ Potentially less conflicts, no locking (SW), bulk communication (HW)
   - – Detects conflicts late

Implementation Space

Conflict Detection	Version Management: Eager	Version Management: Lazy
Pessimistic	HW: UW LogTM SW: Intel McRT, MS-STM	HW: MIT LTM, Intel VTM SW: MS-OSTM
Optimistic		HW: Stanford TCC SW: Sun TL/2

[This is just a subset of proposed implementations]

- No convergence yet
- Decision will depend on
  - Application characteristics
  - Importance of fault tolerance, strong atomicity, complexity
  - Success of contention managers
- May have different approaches for HW, SW, and hybrid
  - It may not even matter

Conflict Detection Granularity

- Object granularity (SW/hybrid)
  - \+ Reduced overhead (time/space)
  - \+ Close to programmer's reasoning
  - – False sharing on large objects (e.g. arrays)
    - – Unnecessary aborts
- 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, …

Atomicity to Non-Transactional Code

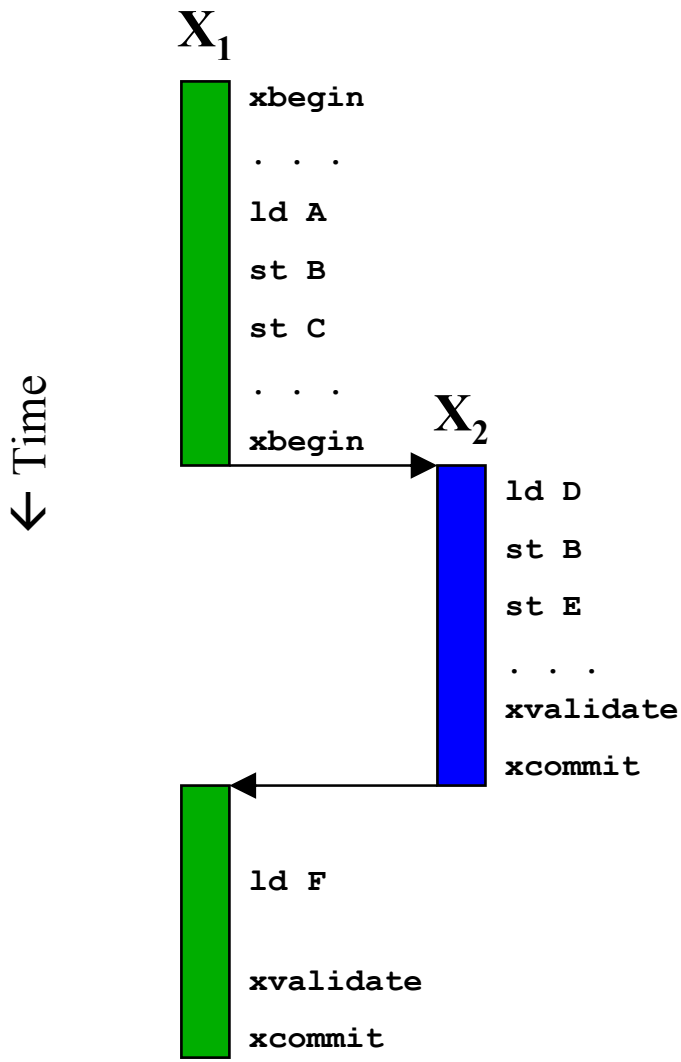
```
T1  . . .                    T2  . . .
    atomic {                     . . .
       write X;                  . . .
      . . .                      read X;
      . . .                      . . .
    }                            . . .
```

- Can non-transactional code read non-committed updates?
  - Yes ➔ weak atomicity
  - No ➔ strong atomicity
- Strong atomicity is generally preferred
  - Otherwise there can be consistency and correctness issues
  - Difficult to provide in SW with eager version management
    - But static or dynamic analysis may be able to help…

Interactions with PL & OS

- Challenging issues
  - Interaction with library-based software, I/O, exceptions, & system calls within transactions, error handling, schedulers, conditional synchronization, memory allocators, new language features, …
- Necessary TM semantics
  1. Two-phase commit
     - Separate validation from commit
  2. Transactional handlers for commit/abort/conflict
     - All interesting events switch to software handlers
     - Mechanisms for registering software handlers
  3. Support for nested transactions
     - Closed: independent rollback & restart for nested transactions
     - Open: independent atomicity and isolation for nested transactions
- See McDonald's paper in [ISCA'06]

Closed Nested Transactions



X_1 State

Read-set	A, D, F
Write-Set	B_2, C_1 , E_2

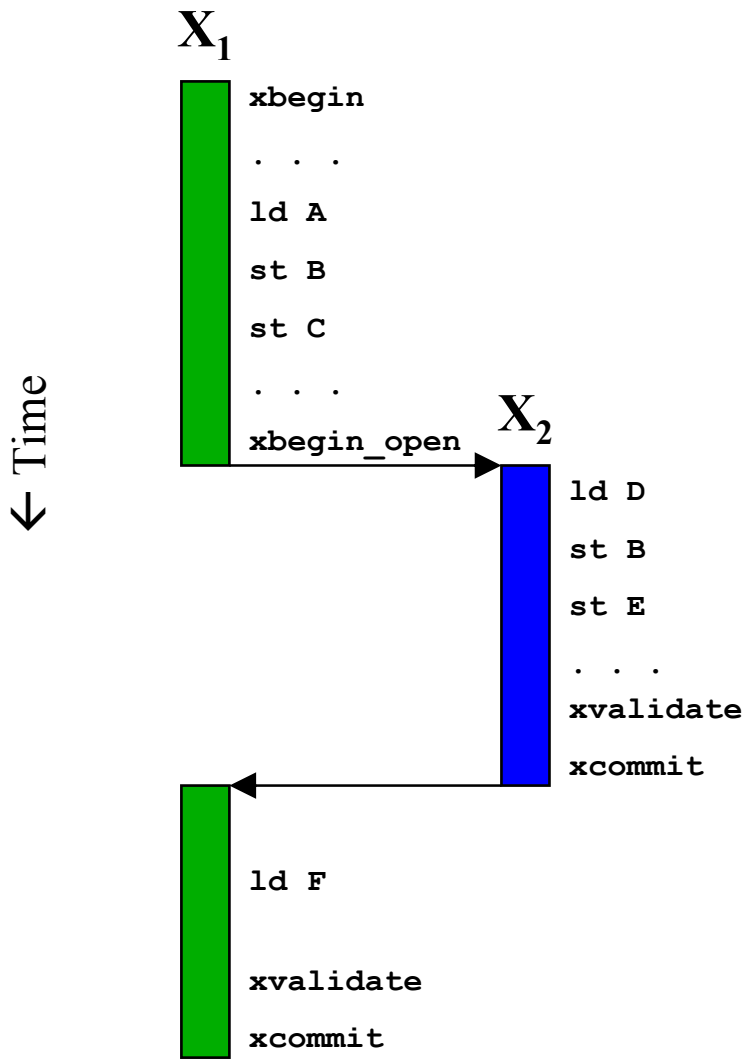
X_2 State

Read-set	D
Write-Set	B_2, E_2

Shared Memory

Address	A	B	C	D	E	F
Value	A_0	B_2	C_1	D_0	E_2	F_0

Open Nested Transactions



X_1 State

Read-set	A, F
Write-Set	B_2, C_1

X_2 State

Read-set	D
Write-Set	B_2, E_2

Shared Memory

Address	A	B	C	D	E	F
Value	A_0	B_2	C_1	D_0	E_2	F_0

Nested Transactions Summary

- Closed nesting
  - Independent rollback and restart
    - Read-set and write-set tracked independently from parent
    - On inner conflict, abort inner transaction but not outer
    - On inner commit, merge with parent's read-set and write-set
  - Uses: reduce cost of conflict, allow alternate execution paths
- Open nesting
  - Independent atomicity and isolation for nested transactions
    - On inner commit, shared memory is updated immediately
    - Independent rollback similar to closed nesting
  - Uses: system and runtime code, reduce frequency of conflicts
    - But, may be too tricky for end programmers

Questions?