A Scalable, Non-blocking Approach to Transactional Memory

Hassan Chafi
Jared Casper
Brian D. Carlstrom
Austen McDonald
Chi Cao Minh
Woongki Baek
Christos Kozyrakis
Kunle Olukotun

Computer System Laboratory
Stanford University
http://tcc.stanford.edu

Transactional Memory

- Problem: Parallel Programming is hard and expensive.
  - Correctness vs. performance
- Solution: Transactional Memory
  - Programmer-defined isolated, atomic regions
  - Easy to program, comparable performance to fine-grained locking
  - Done in software (STM), hardware (HTM), or both (Hybrid)
- Conflict Detection
  - Optimistic: Detect conflicts at transaction boundaries
  - Pessimistic: Detect conflicts during execution
- Version management
  - Lazy: Speculative writes kept in cache until end of transaction
  - Eager: Speculatively write “in place”, roll back on abort

So what's the problem? (Haven't we figured this out already?)

- Cores are the new GHz
  - Trend is 2x cores / 2 years: 2 in '05, 4 in '07, > 16 not far away
  - Sun: N2 has 8 cores with 8 threads = 64 threads
- It takes a lot to adopt a new programming model
  - Must last tens of years without much tweaking
  - Transactional Memory must (eventually) scale to 100s of processors
- TM studies so far use a small number of cores!
  - Assume broadcast snooping protocol
- If it does not scale, it does not matter

Lazy optimistic vs. Eager pessimistic

High contention

– Eager pessimistic
  - Serializes due to blocking
  - Slower aborts (result of undo log)

- Lazy optimistic
  - Optimistic parallelism
  - Fast aborts

Low contention

– Eager pessimistic
  - Fast commits

- Lazy optimistic
  - Slower commits… good enough??

What are we going to do about it?

- Serial commit $\Rightarrow$ Parallel commit
  - At 256 proc, if 5% of the work is serial, maximum speedup is 18.6x
  - Two-phase commit using directories
- Write-through $\Rightarrow$ write-back
  - Bandwidth requirements must scale nicely
  - Again, using directories
- Rest of talk:
  - Augmenting TCC with directories
  - Does it work?

Protocol Overview

- During the transaction
  - Track read and write sets in the cache
  - Track sharers of a line in the directory
- Two-phase commit
  - Validation: Mark all lines in write-set in directories
    - Locks line from being written by another transaction
  - Commit: Invalidate all sharers of marked lines
    - Dirty lines become “owned” in directory
- Require global ordering of transactions
  - Use a Global Transaction ID (TID) Vendor

Directory Structure

Directory

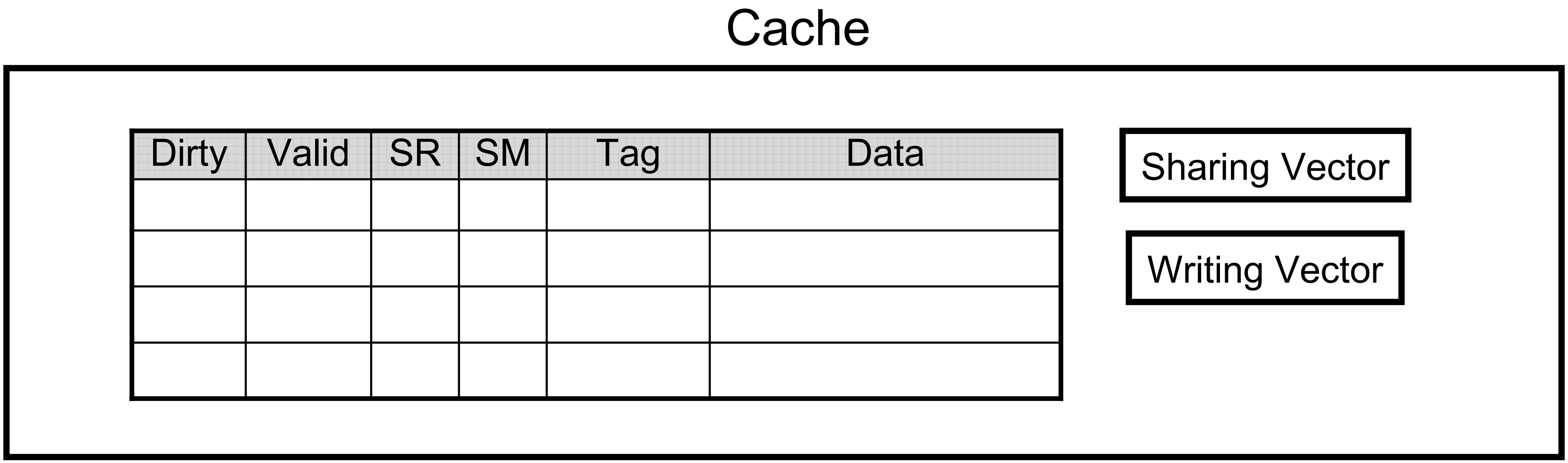
Address	Sharers List				Marked	Owned
	P0	P1	…	PN		
0x0000						
0x0004						
…..						
0x1000						

Now Serving TID (NSTID)

Skip Vector

- Directory tracks sharers of each line at home node
  - Marked bit is used in the protocol
- Now serving TID: transaction currently being serviced by directory
  - Used to ensure a global ordering of transactions
  - Skip vector used to help manage NSTID (see paper)

Cache Structure



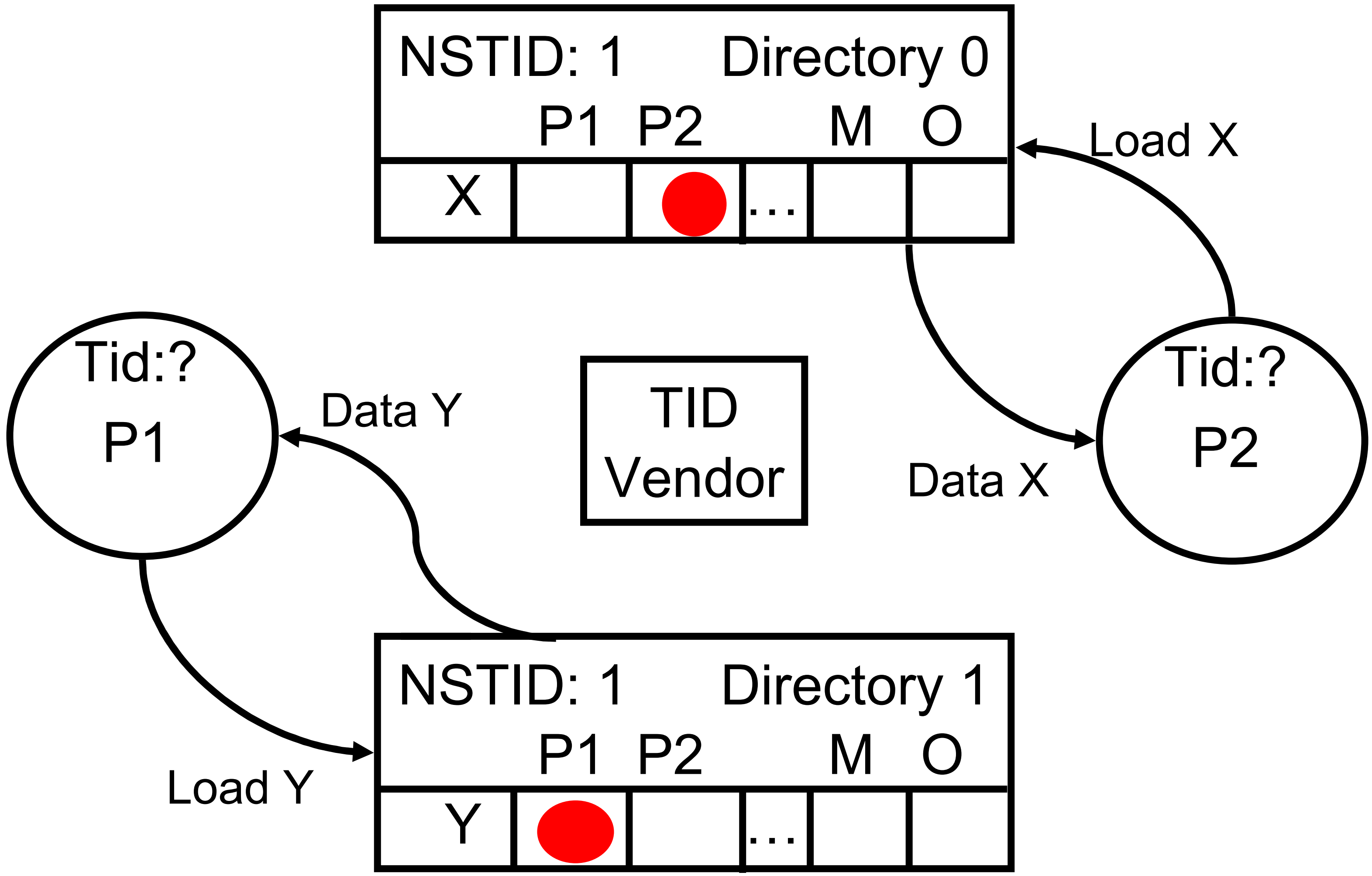
- Each cache line tracks if it was speculatively read (SR) or modified (SM)
  - Meaning that line was read or written in the current transaction
- Sharing and Writing vectors remember directories read from or written to
  - Simple bit vector

Commit procedure

- Validation
  - Request TID
  - Inform all directories not in writing vector we will not be writing to them (Skip)
  - Request NSTID of all directories in writing vector
    - Wait until all NSTIDs ≥ our TID
  - Mark all lines that we have modified
    - Can happen in parallel to getting NSTIDs
  - Request NSTID of all directories in sharing vector
    - Wait until all NSTIDs ≥ our TID
- Commit
  - Inform all directories in writing vector of commit
  - Directory invalidates all other copies of written line, and marks line owned
    - Invalidation may violate other transaction

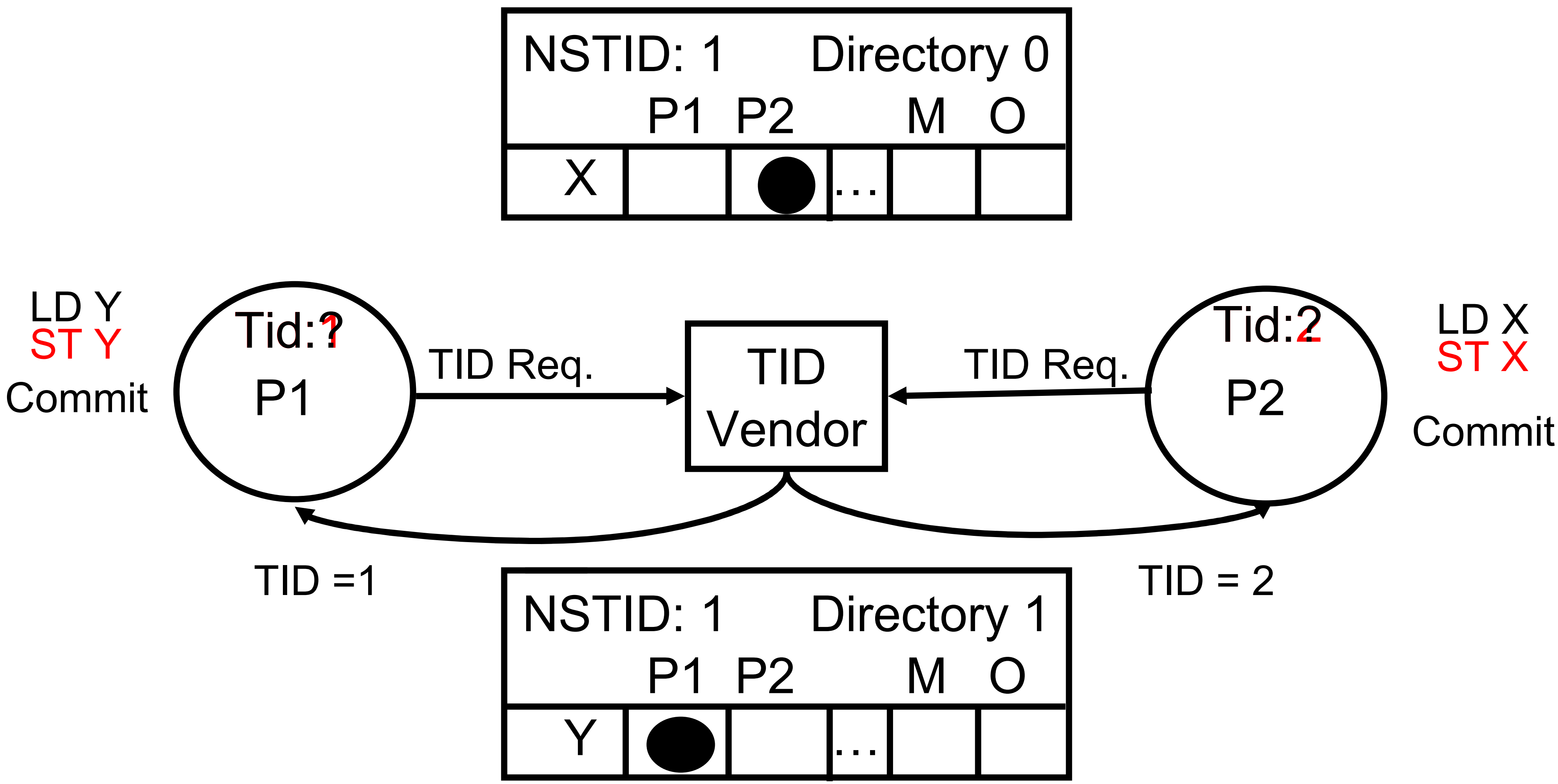
Parallel Commit Example

LD Y
ST Y
Commit

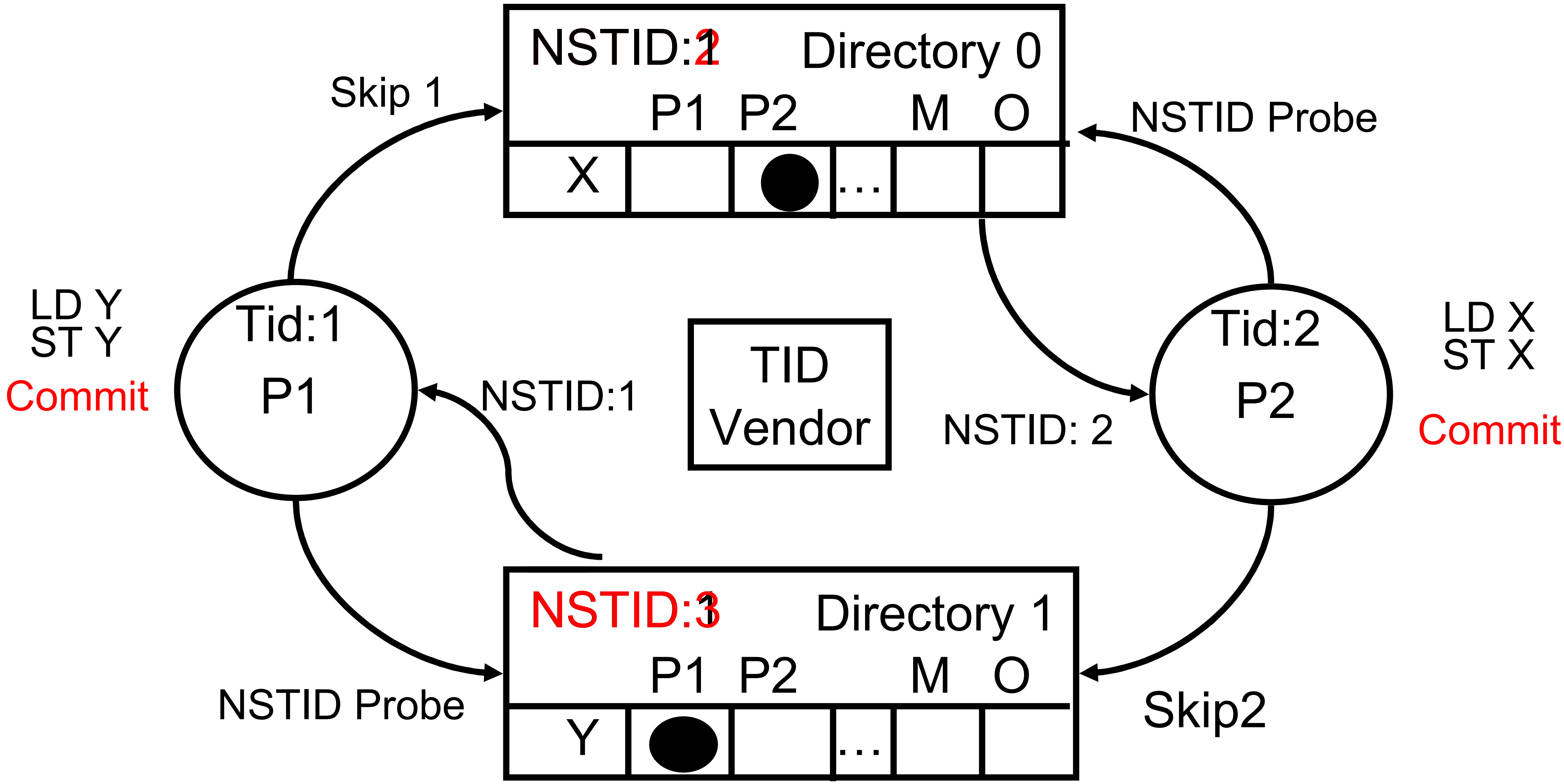


LD X
ST X
Commit

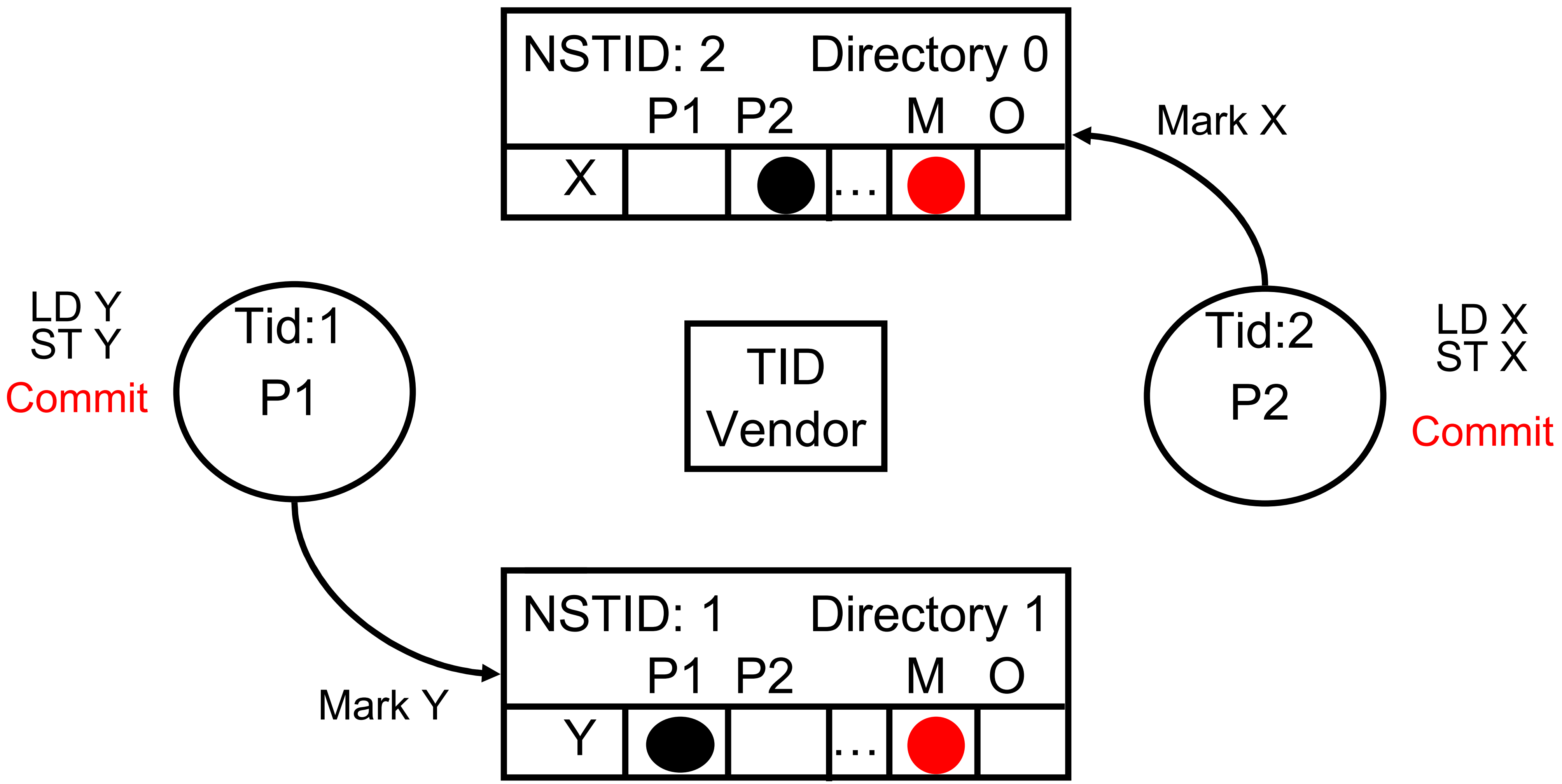
Parallel Commit Example



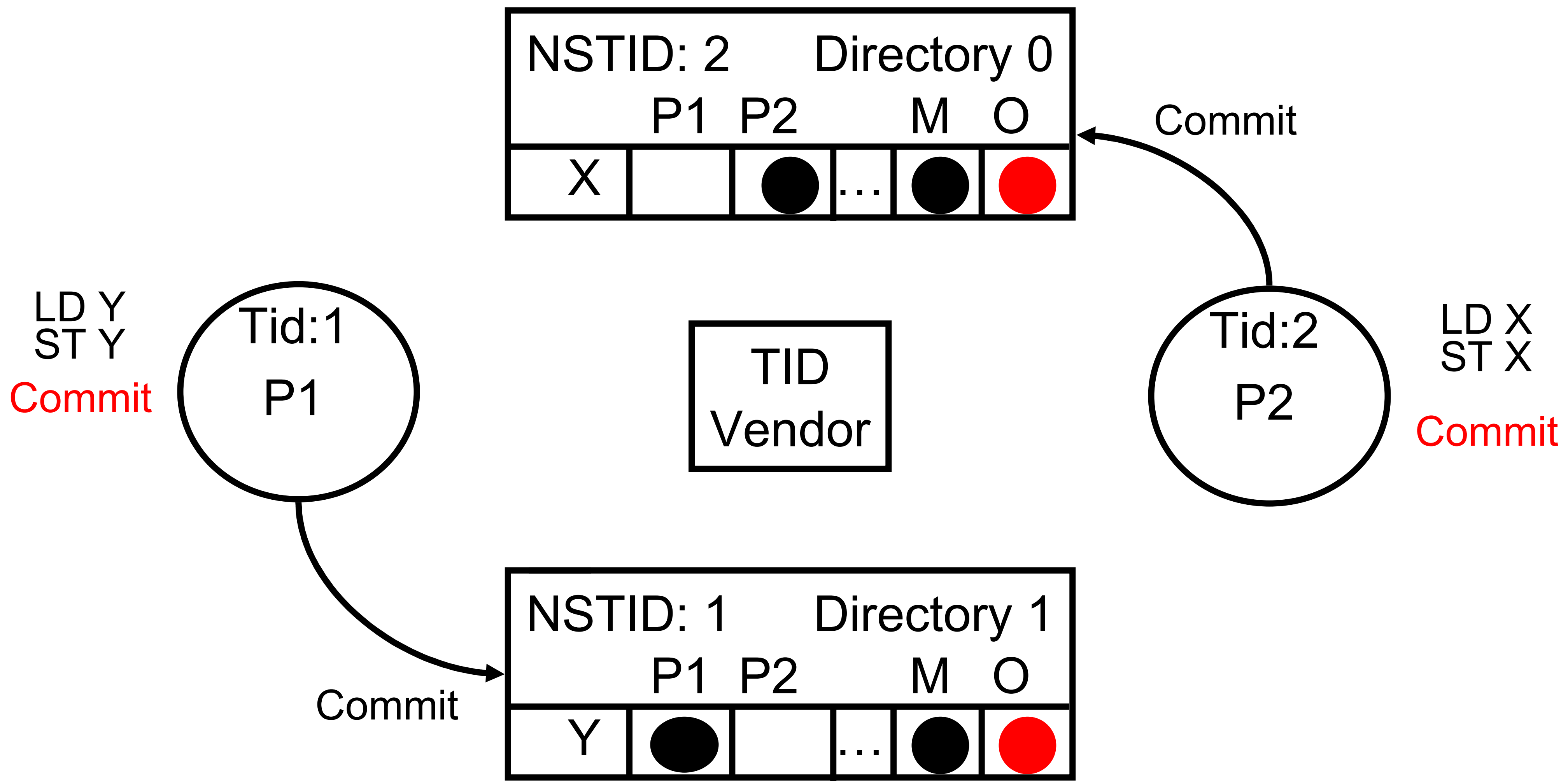
Parallel Commit Example



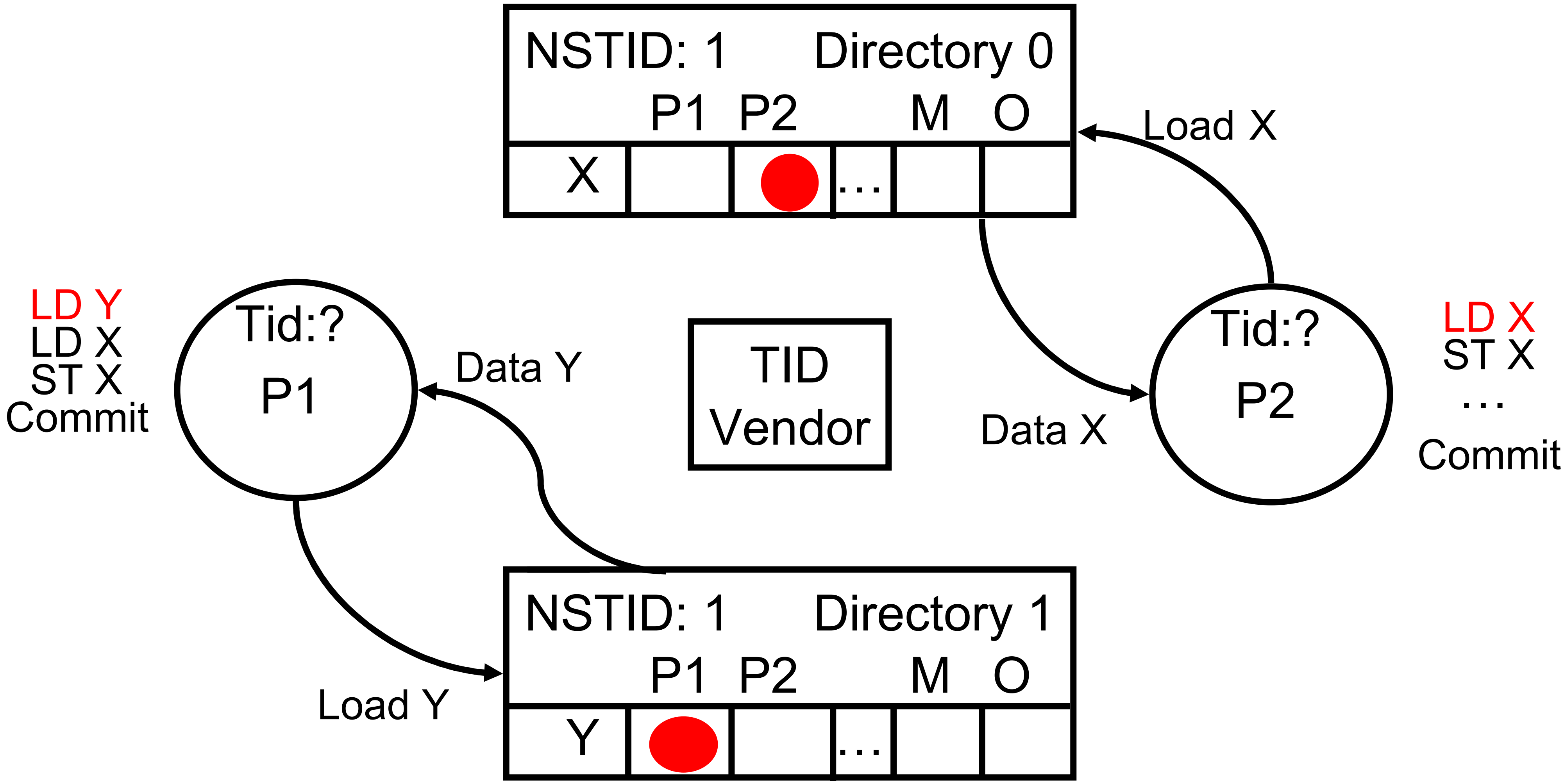
Parallel Commit Example



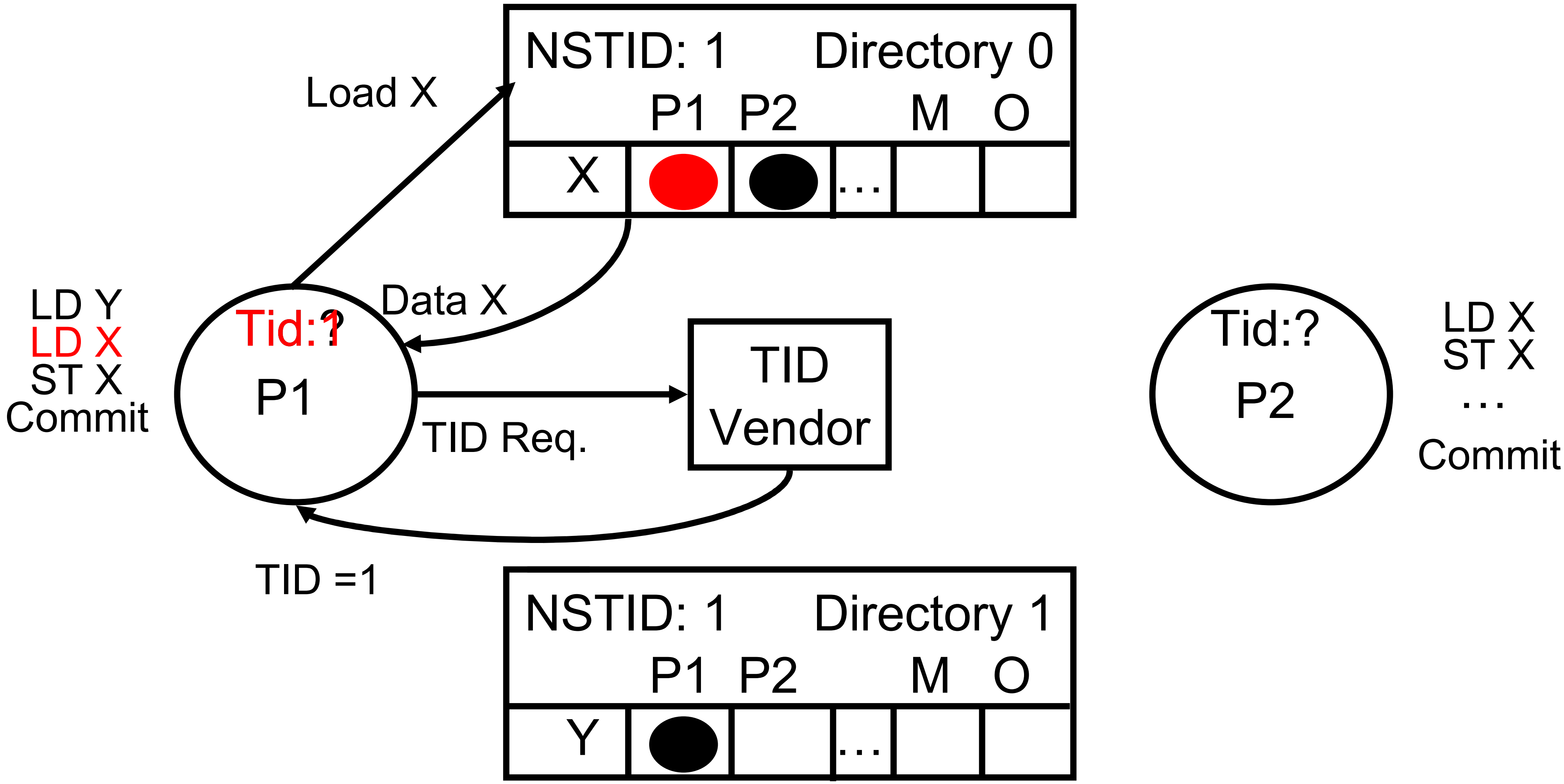
Parallel Commit Example



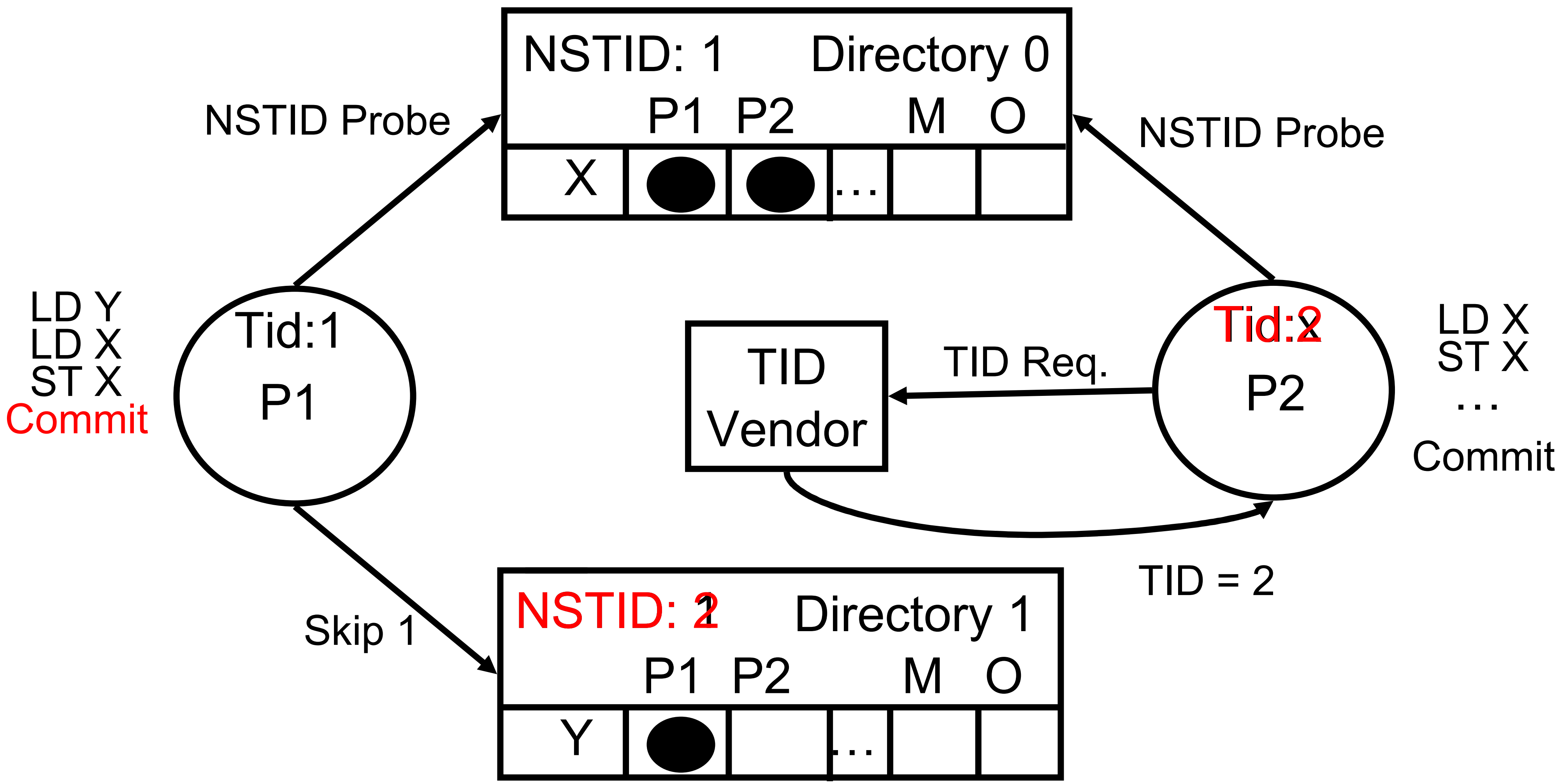
Conflict Resolution Example



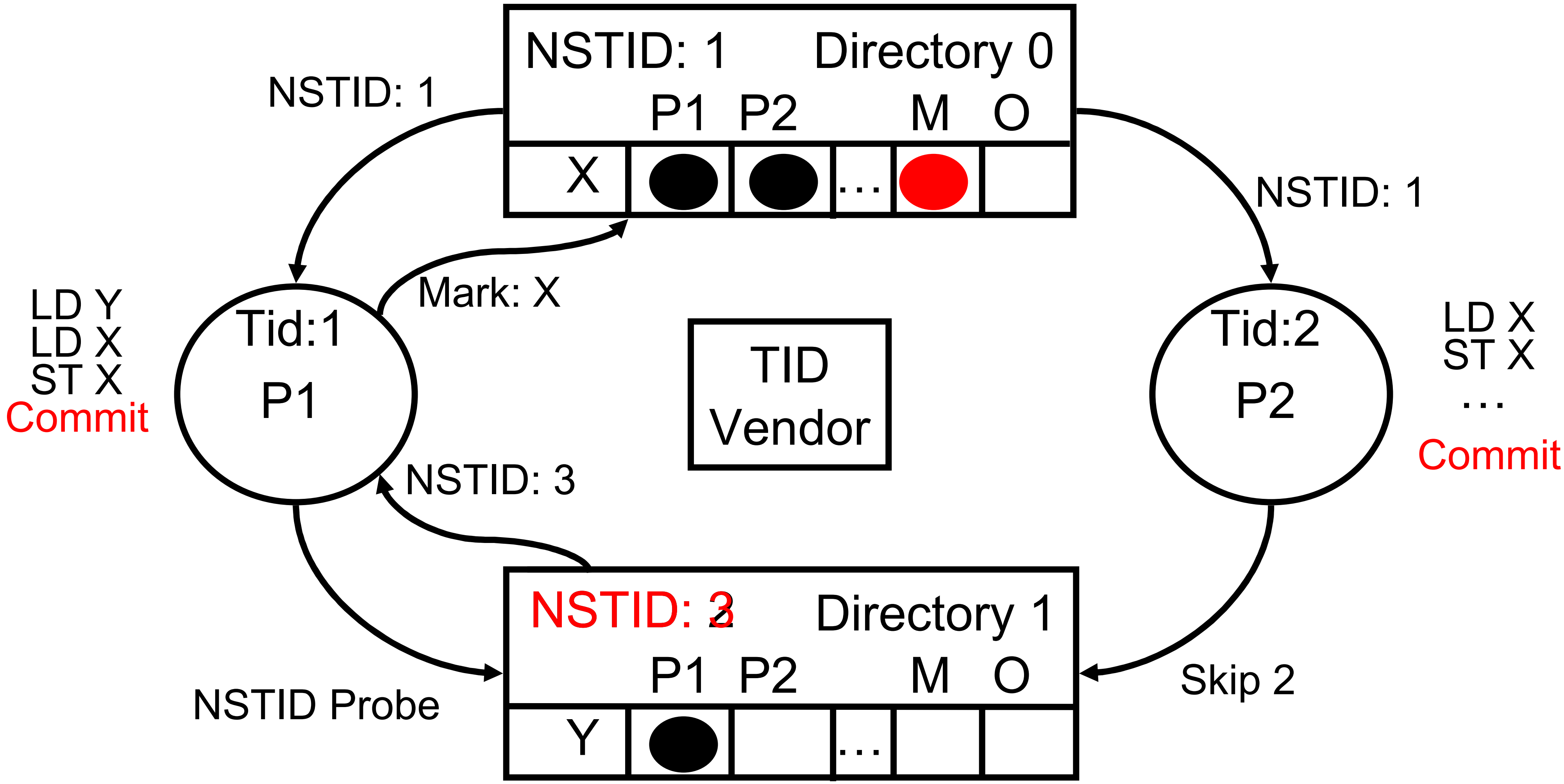
Conflict Resolution Example



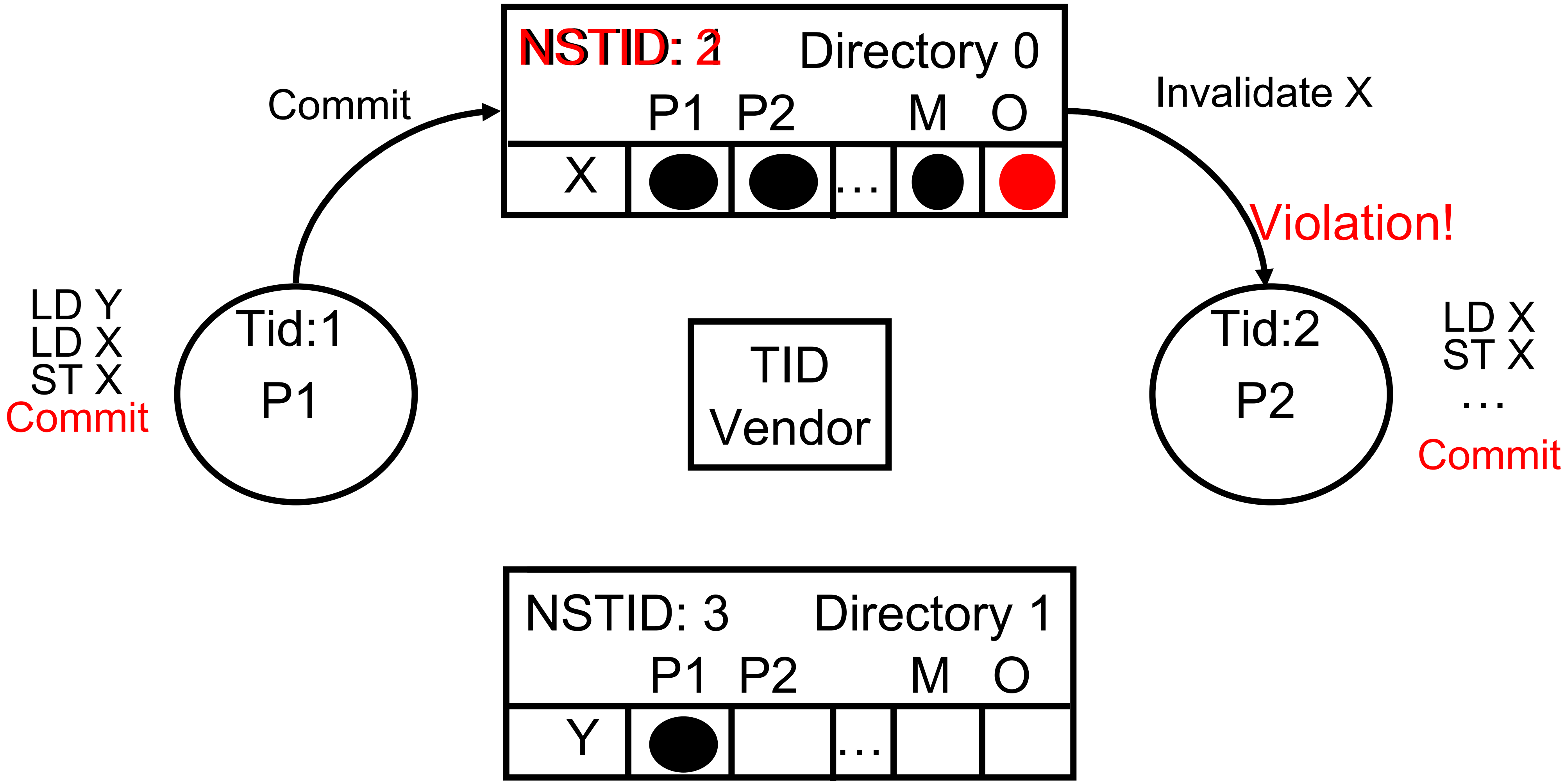
Conflict Resolution Example



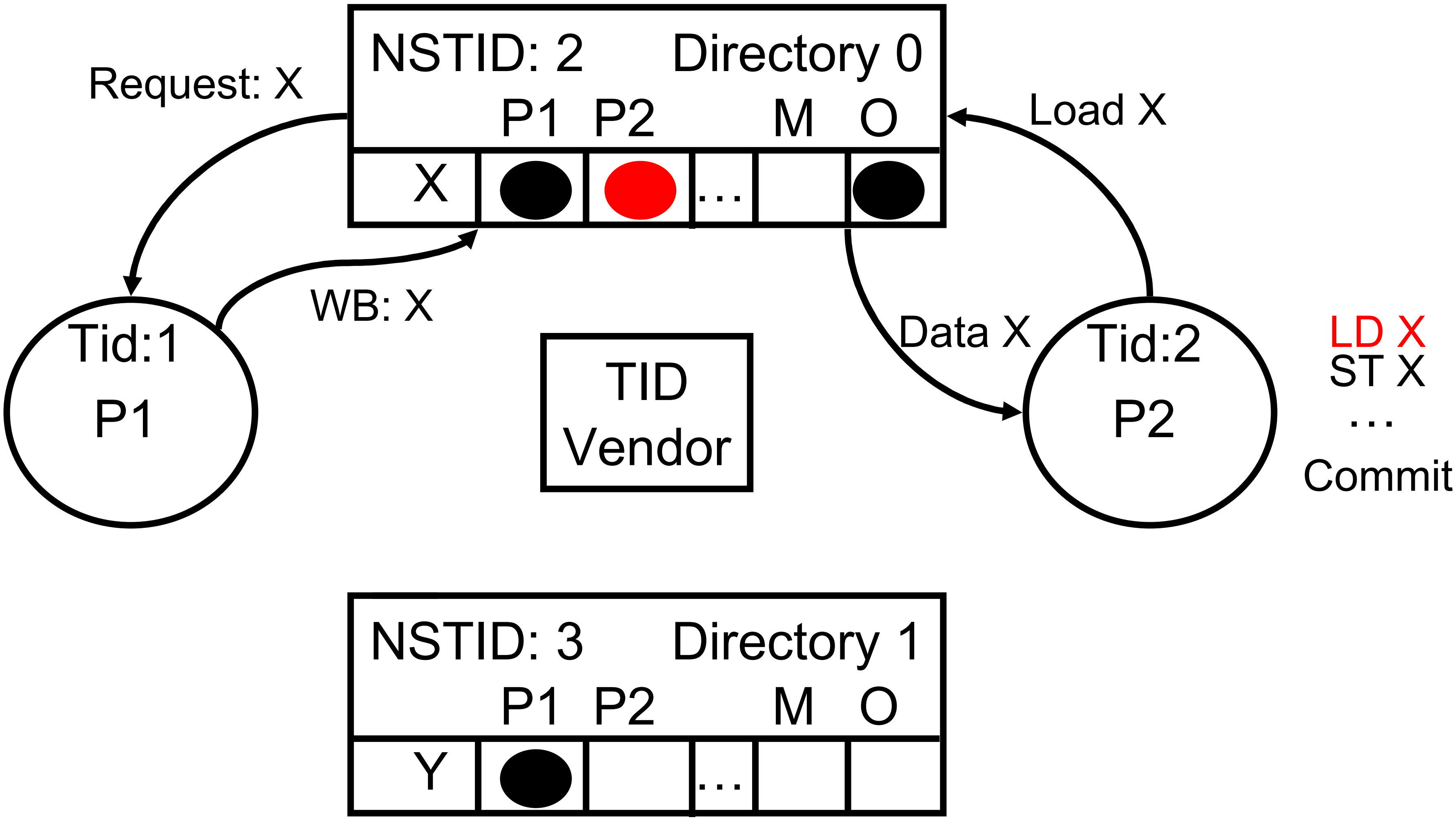
Conflict Resolution Example



Conflict Resolution Example



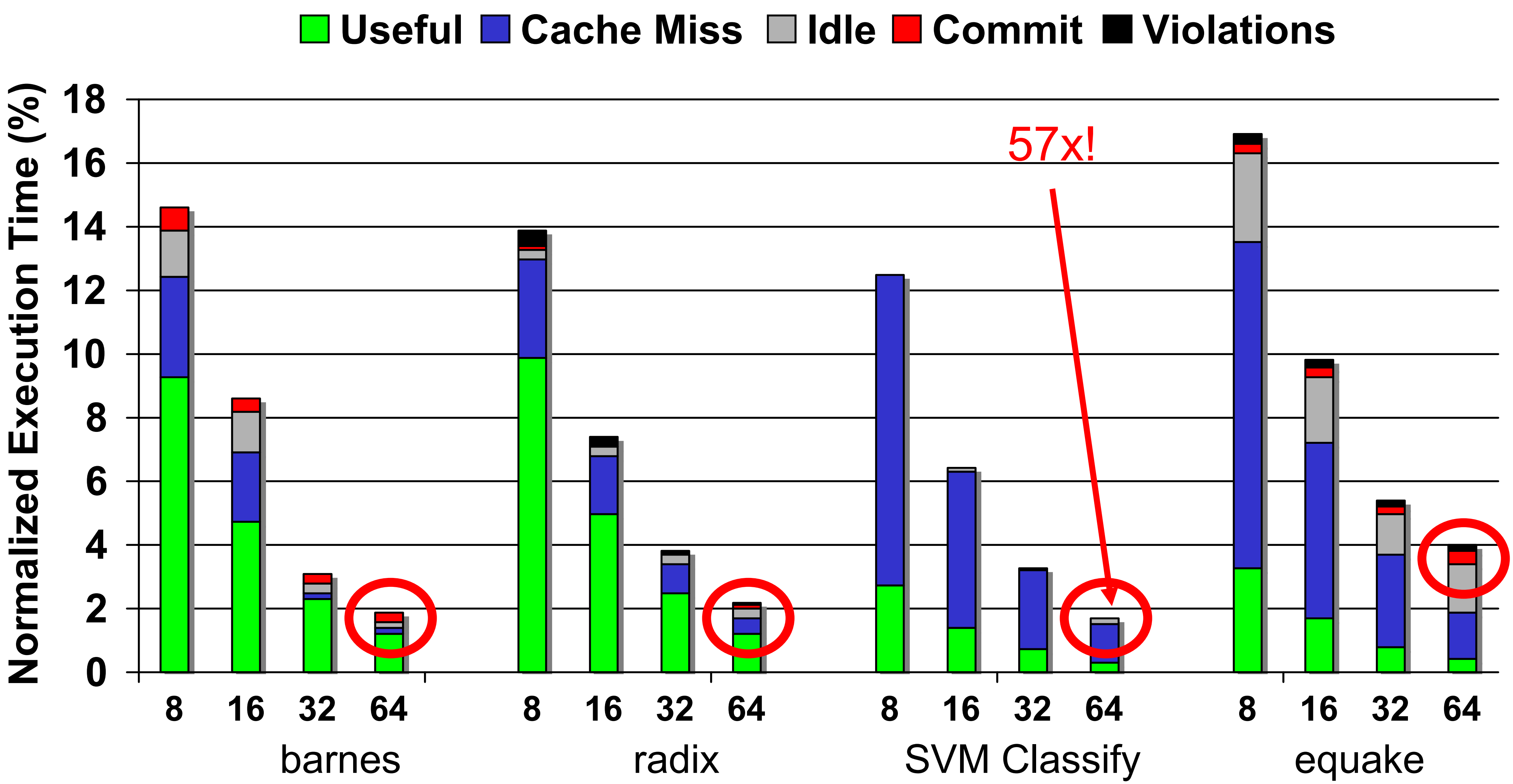
Conflict Resolution Example (Write-back)



Evaluation environment

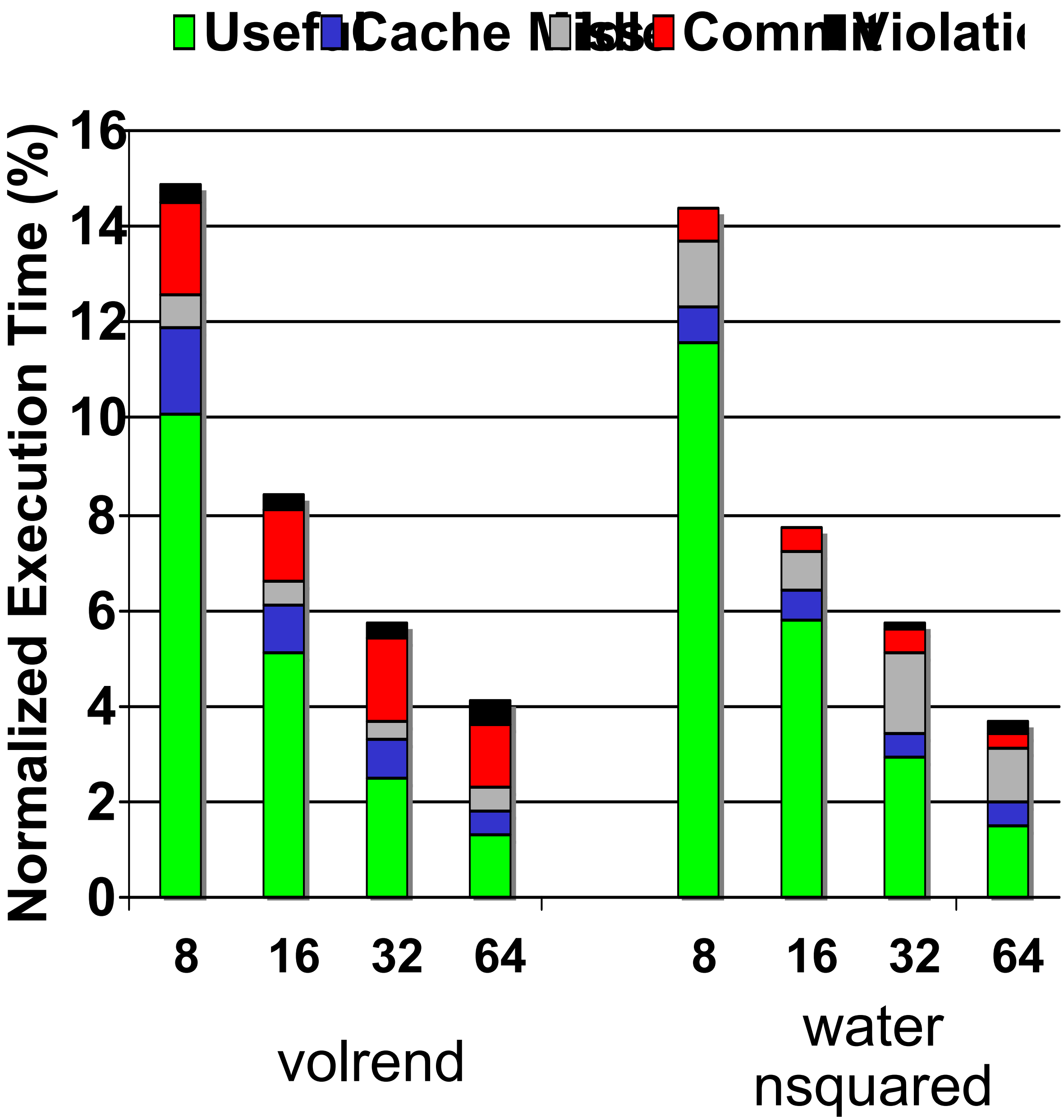
CPU	1 - 64 single-issue PowerPC cores
L1	32 KB, 32 byte cache line, 4-way, 1 cycle latency
L2	512 KB, 32 byte cache line, 8-way, 16 cycle latency
Interconnection	2D grid topology, 14 cycle link latency
Main Memory	100 cycle latency
Directory	1 per node, 10 cycle latency

It Scales!



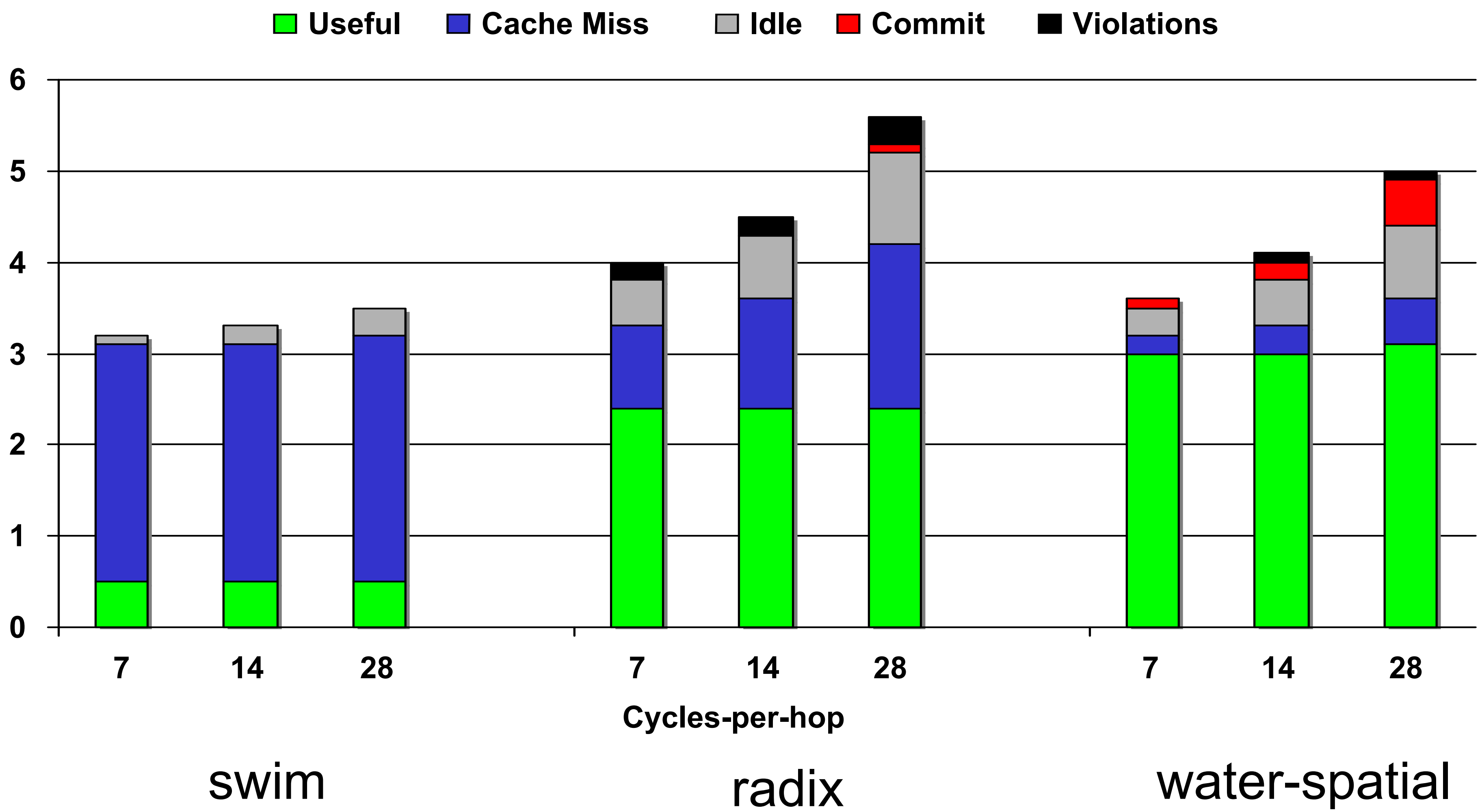
- Commit time (red) is small and decreasing, or non-existent

Results for small transactions



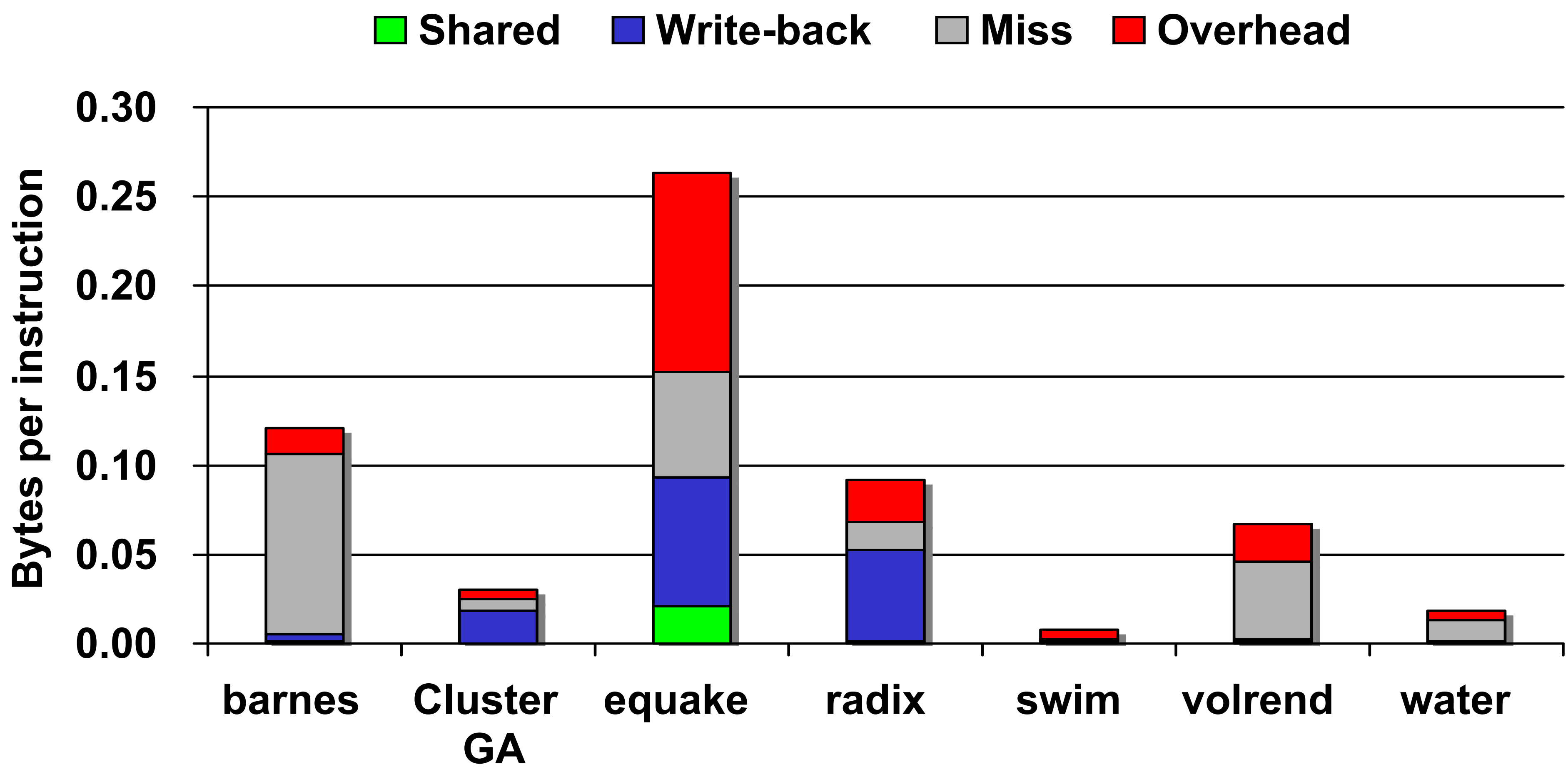
- Small transactions with a lot of communication magnifies commit latency
- Commit overhead does not grow with processor count, even in the worst case

Latency Tolerance



- 32 Processor system

Remote traffic bandwidth



- Comparable to published SPLASH-2
- Total bandwidth needed (at 2 GHz) between 2.5 MBps and 160 MBps

Take home

- Transactional Memory systems must scale for TM to be useful
- Lazy optimistic TM systems have inherent benefits
  - Non-blocking
  - Fast abort
- Lazy optimistic TM system scale
  - Fast parallel commit
  - Bandwidth efficiency through write-back commit

Questions?

Jared Casper
jaredc@stanford.edu

Computer Systems Lab
Stanford University
http://tcc.stanford.edu