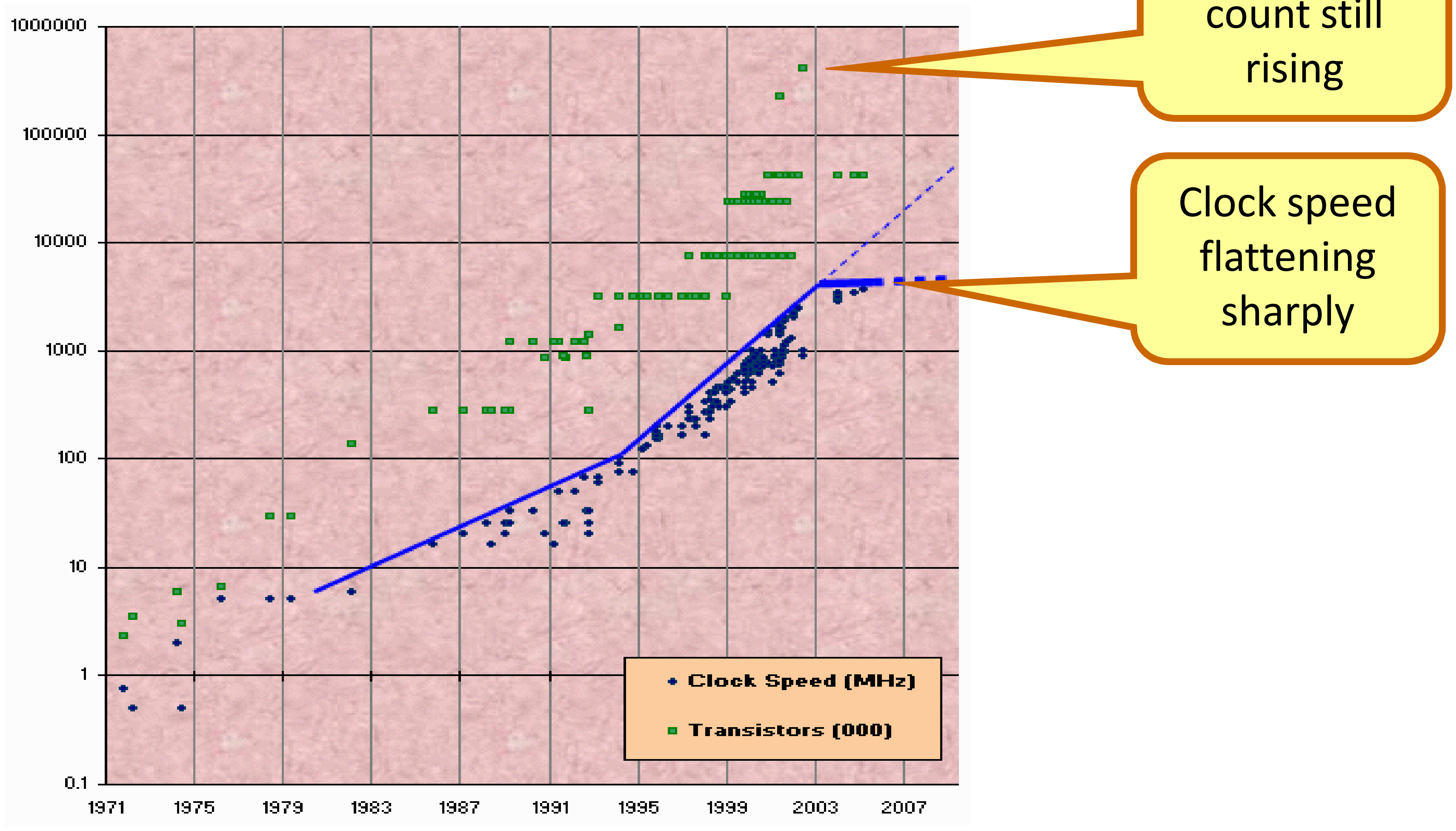
Hardware Transactional Memory

Moore's Law

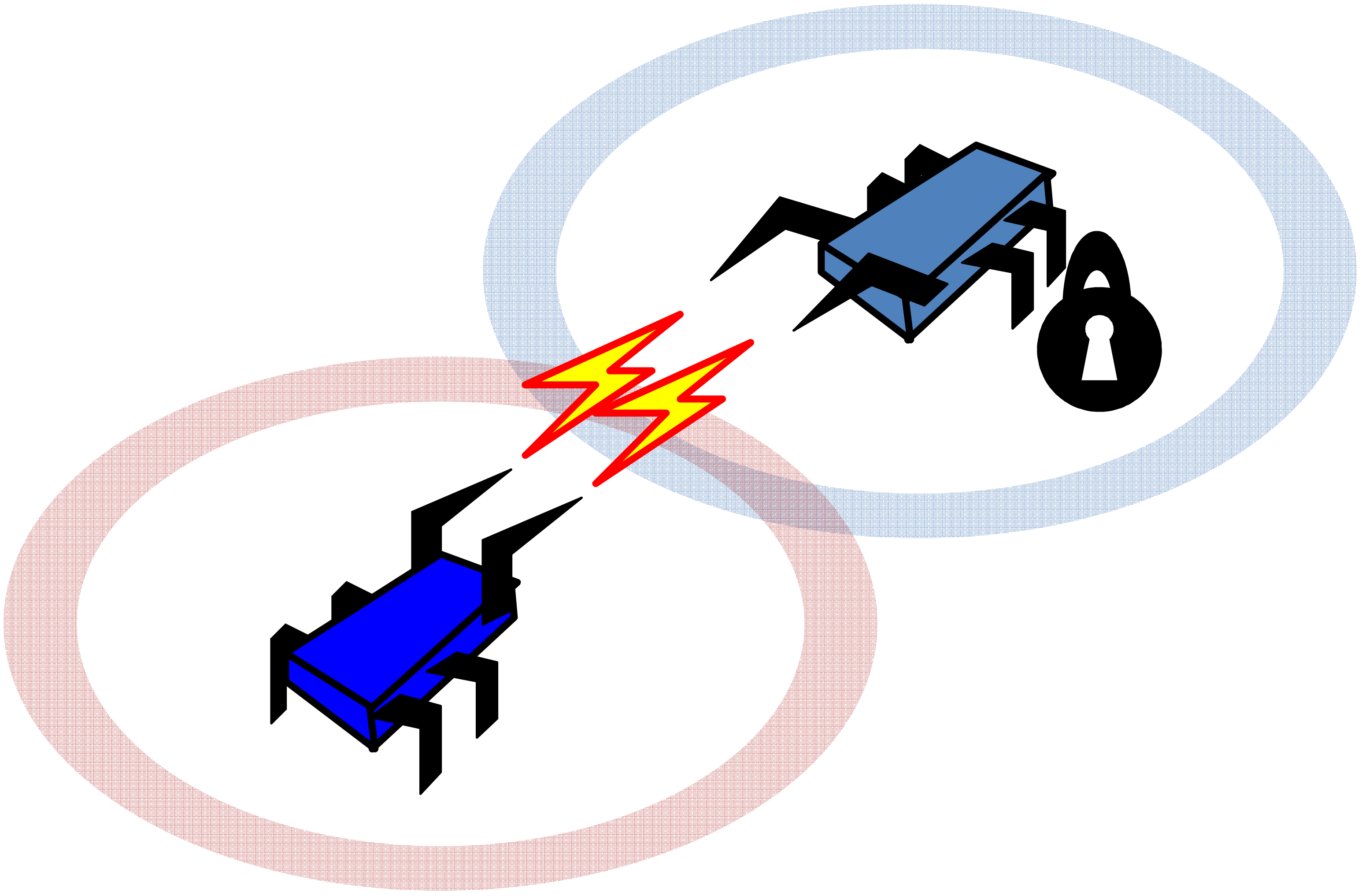

1000000
100000
10000
1000
100
10
1
0.1
1971
1975
1979
1983
1987
1991
1995
1999
2003
2007
● Clock Speed (MHz)
🟩 Transistors (000)
Transistor count still rising
Clock speed flattening sharply

Two communication models

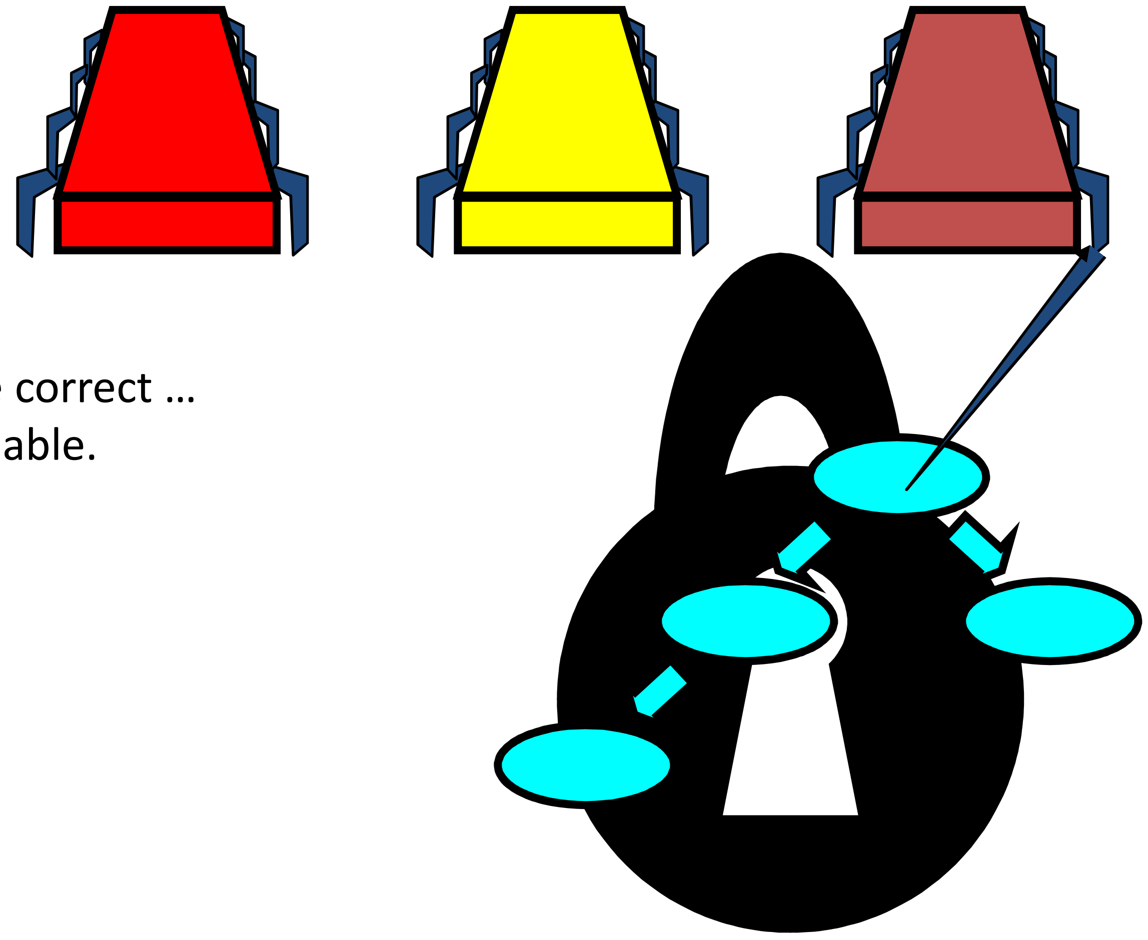
- Message Passing (e.g. MPI)
- Shared Memory

- **This talk:** Shared memory
- **Ultimate question:** How can we write code that needs to run “atomically” ?

Locking

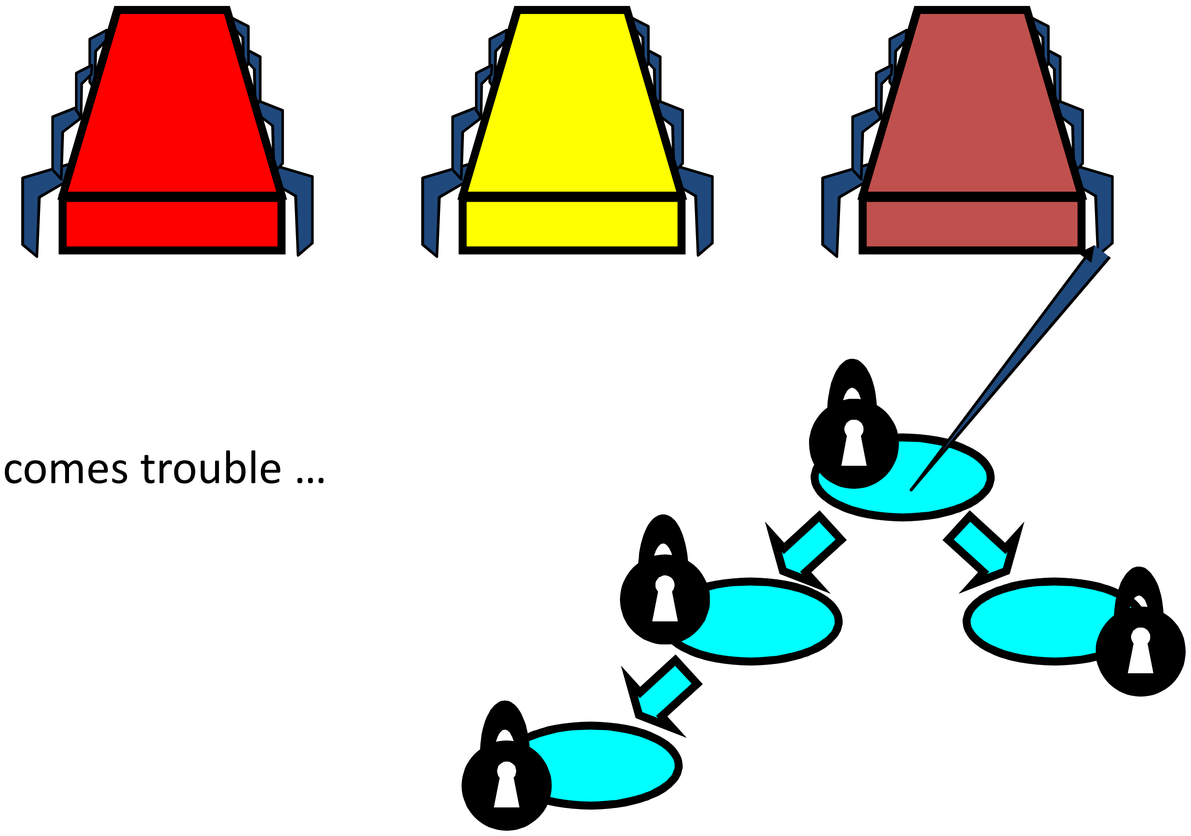


Coarse-Grained Locking



Easily made correct …
But not scalable.

Fine-Grained Locking



Here comes trouble …

Why Locking Doesn't Scale

- Relies on conventions
- Hard to Use
  - Conservative
  - Deadlocks
  - Lost wake-ups: Miss a conditional wake-up when mutex_lock has not been acquired
- Not Composable

Why Locking Doesn't Scale

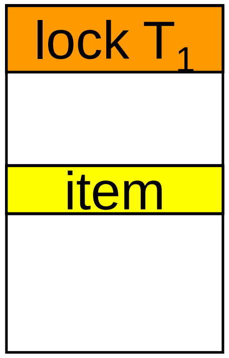
- Relies on conventions

- Hard to Use
  - Conservative
  - Deadlocks, Priority inversion
  - Lost wake-ups: Miss a conditional wake-up when mutex_lock has not been acquired
- Not Composable

Locks do not compose

Hash Table

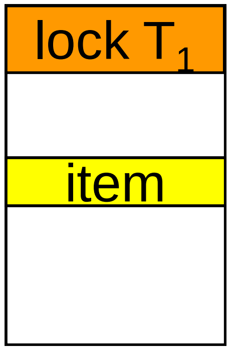
add(T_1, item)

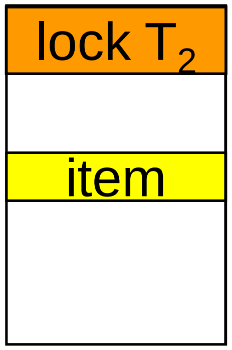


Must lock T_1 before adding item

Move from T_1 to T_2

delete(T_1, item)
add(T_2, item)





Must lock T_2 before deleting from T_1

Exposing lock internals breaks abstraction

Solution: Don’t use locks

“atomic” abstraction

```
Public void LeftEnq(item x) {
 atomic {
  Qnode q = new Qnode(x);
  q.left = this.left;
  this.left.right = q;
  this.left = q;
 }
}
```

- Composable
- Easy to reason about
- No need to know implementation specific details
- Main disadvantage: **“Scoped”**

Atomic

- Can still be implemented with locks
- "Lockset" inference algorithms
- Can still cause deadlocks , starvation
- Blocking methods reduce parallelism – e.g.

```
 lock lk;
 receive buffer from inet ;
*ptr = buffer ;
  unlock lk
```

Solution: Don’t use blocking methods

Non-blocking methods:

- Transactional Memory
- Proofs: Temporal Logic ➔
  Atomic:Thread1@(t1,t2) ^
  Atomic:Thread2@(t3,t4) ^
  overlap(t1,t2,t3,t4) = false
- Lock-free algorithms … error prone
- … other ?

HTM vs STM

- Two schools of thought
  1. Transactions can be done entirely in software
  2. No they can't.

- STM:
  - Slow but no special H/W required

- HTM
  - Fast but special H/W required

Both methods suffer from irrevocable "side-effects":
Transactions are based on **cancelation.** An I/O events (e.g. launch missile) are usually irrevocable

Transactions = ACID

- Atomicity
  - All (**at once**)or nothing
- Consistency
  - Correct state change, legal state **before** and **after**
- Isolation
  - Acts as if alone --- Non-interference, “single step”
- Durability
  - Survives failures, complete system restoration

TM = ACI

- Atomicity
  - All (**at once**)or nothing
- Consistency
  - Correct state change, legal state **before** and **after**
- Isolation
  - Acts as if alone --- Non-interference, “single step”
- ~~Durability~~
  - ~~Survives failures~~, complete system restoration

Herlihy & Moss [93]

- Idea: Hardware provides a direct transactional abstraction in the primary memory of the computer, not just on disk, as with database systems.

- **“Abstract” Invariant at commit:**

  $\mathbf{Valid(T_j)} \equiv_{\alpha} \forall \mathbf{i \neq j.} \; \mathbf{(}RS(T_j) \cup WS(T_j)) \cap WS(T_i) = \emptyset) \wedge (RS(T_i) \cap WS(T_j) = \emptyset)$

  - **If valid(T_j) then commit else abort**
  - **On abort, restore state**

Herlihy & Moss [93]

- RAM not involved.
- **Transaction** = R/W cache lines

- **Log** *original values* in cache lines for T1
- **Abort** = clear T1 cache lines, restore original values modified by T1
- **Commit** = clear T1 cache lines, commit

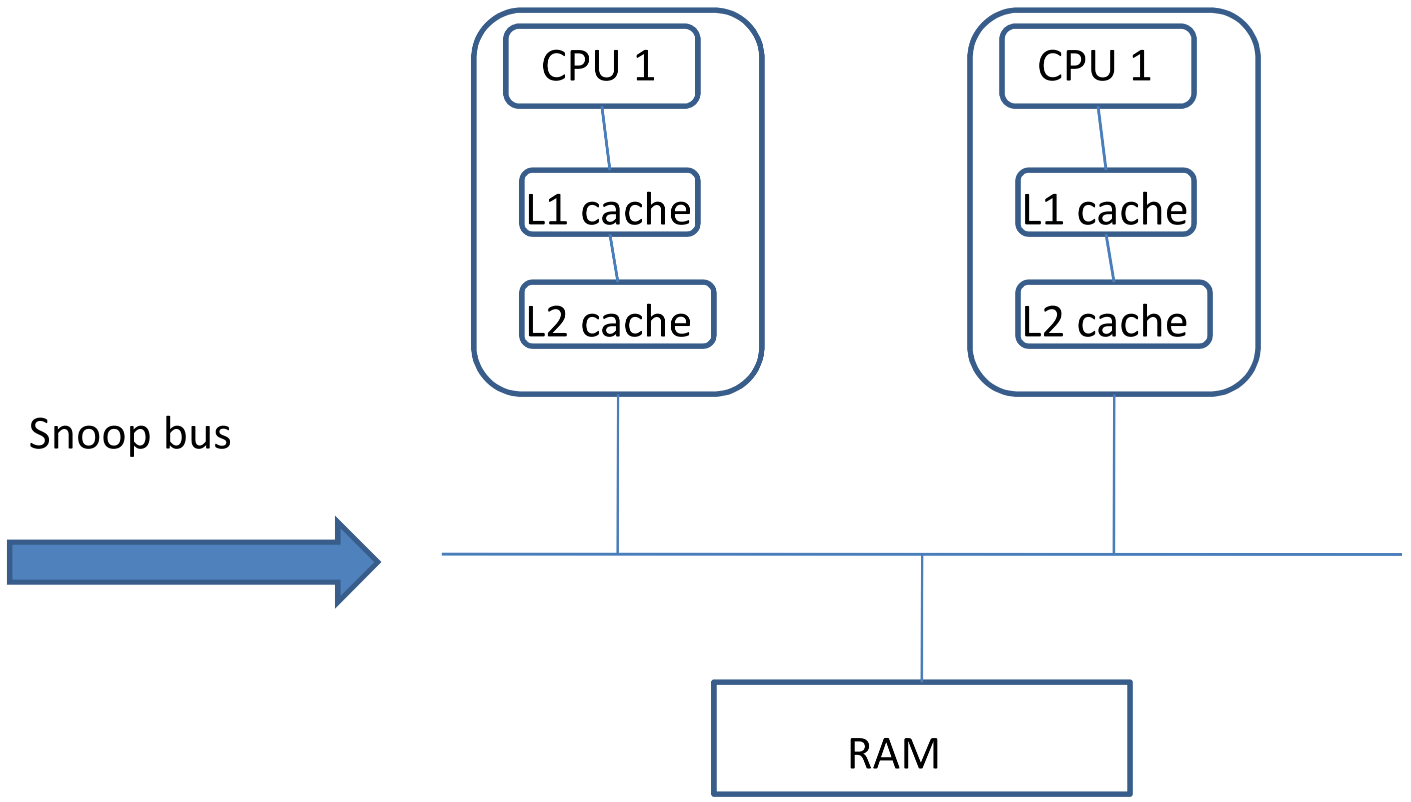
Transactional Instructions

- Memory access:
  - LT or LTX, , ST
- Transaction state:
  - Validate, Commit , Abort
- **Note:** There is no “begin Tx” instruction

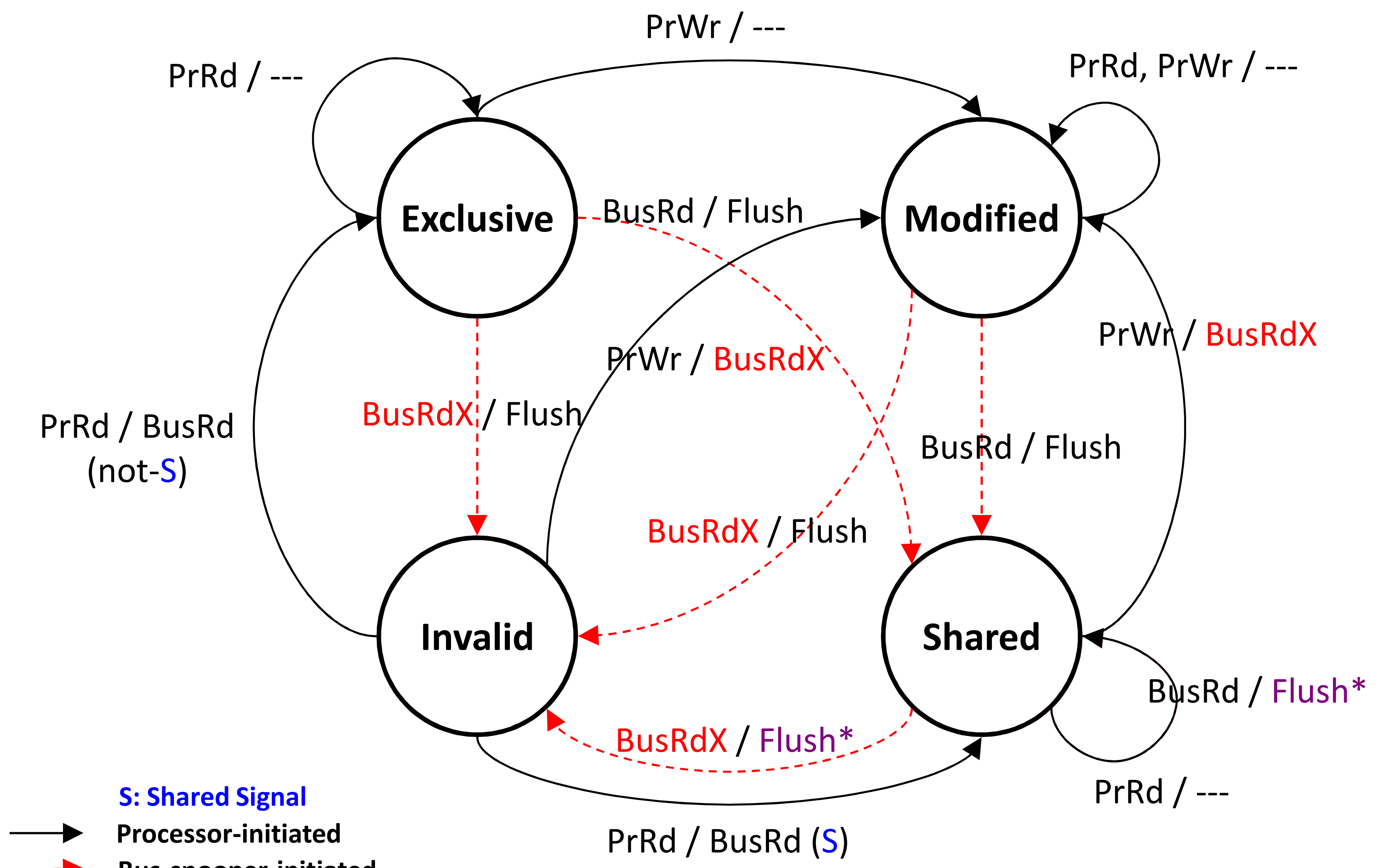
Intended use

```
loop:
 LT  0x5000
 mov eax, word ptr[0x5000]
add  eax,ecx
 VALIDATE
jnz loop
mov word ptr[0x5000],eax
ST 0x5000
COMMIT
jnz loop
halt
```

Snoopy caches



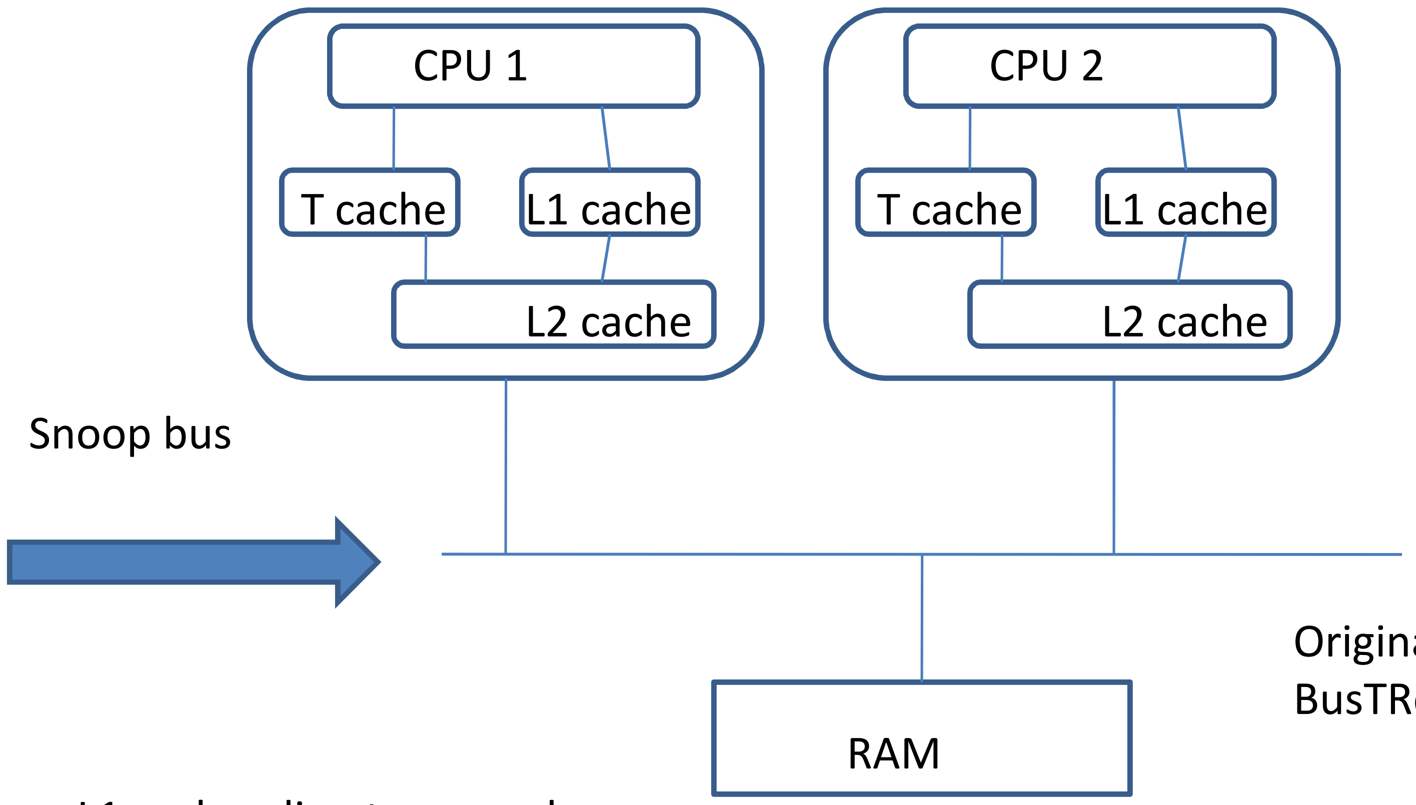
MESI Writeback Invalidation Protocol
(**Messy** Protocol ?)



S: Shared Signal

Flush*: Flush for data supplier; no action for other sharers

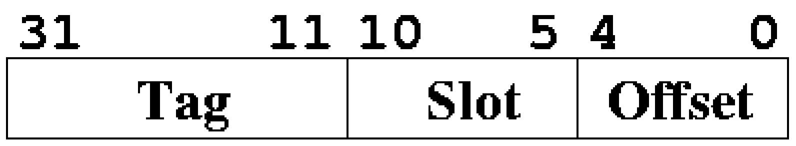
HTM: Snoopy caches



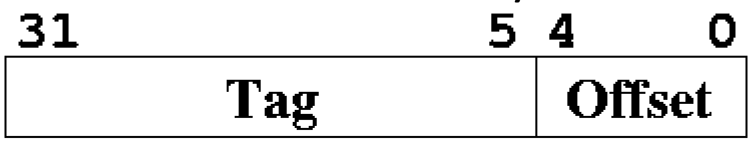
Exclusive memory locations in T and L1 cache

Original Bus cycles +
BusTRd,BusBusy,BusTRdX

L1 cache: direct-mapped



T Cache: fully assoc



T Cache

- Key point – new transactional **cache tags**

- Make **two copies** of data during a transaction
  – XCOMMIT, XABORT (=**log,** kept in Tcache)

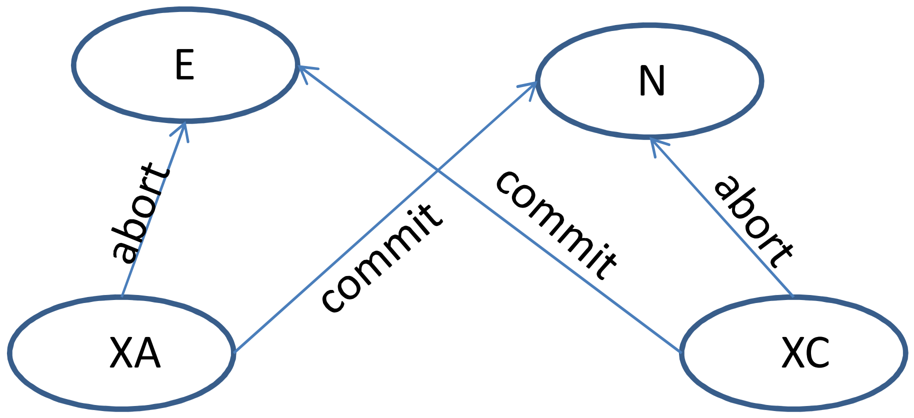
- Updates to XABORT only
- XCOMMIT readonly

T cache

- Two states per **line**:
  - Usual state = {M,E,S,I}
  - T-state = { Empty, Normal, Xcommit ,Xabort }

performance

Not all combs valid

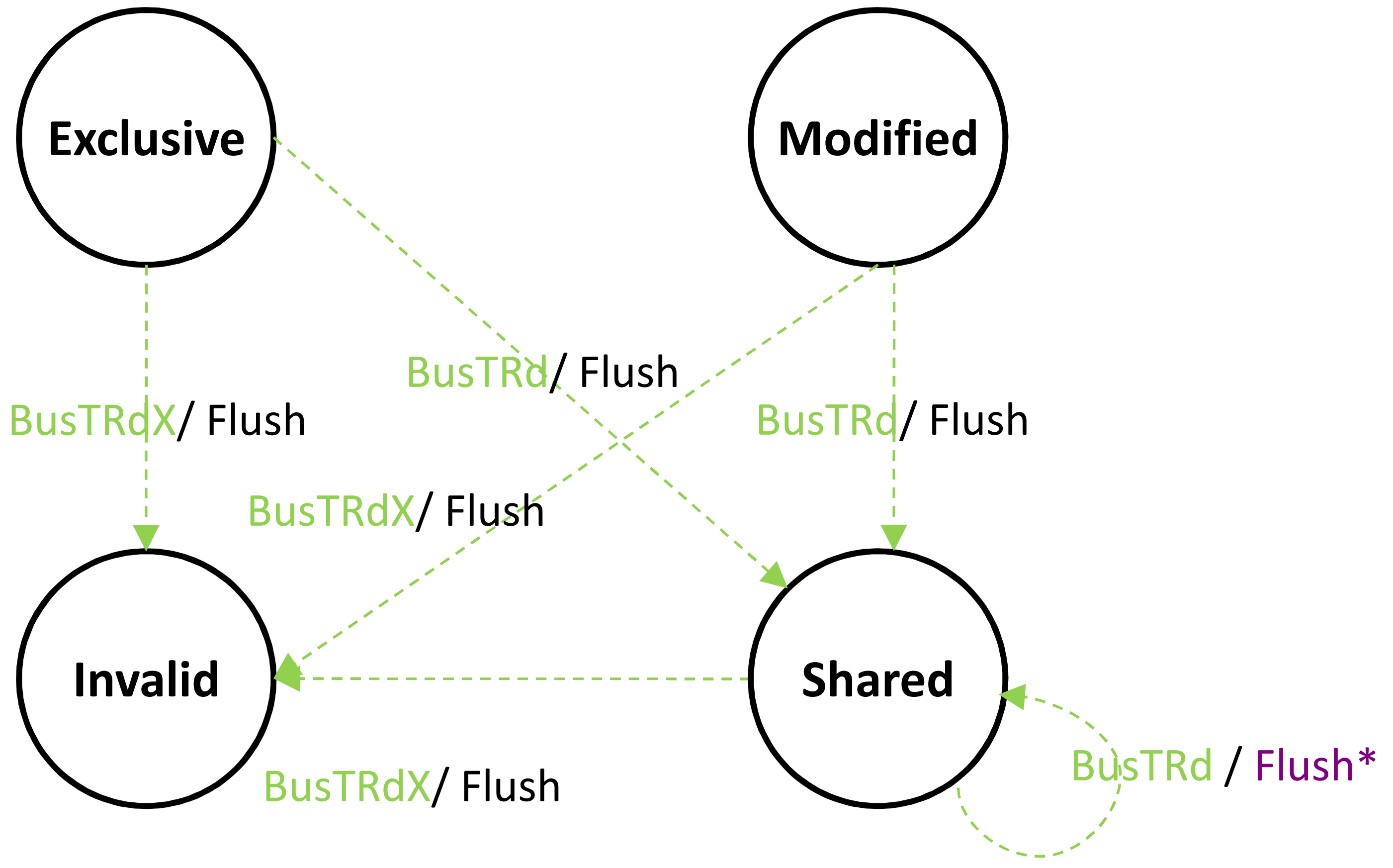


Replacement strategy:
- Empty first
- Normal then
- Xcommit (if dirty write-back)

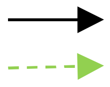
Question 1: What happens when all tags are **XA** ?

Question 2: What happens when an interrupt occurs ?

L1 Cache



S: Shared Signal



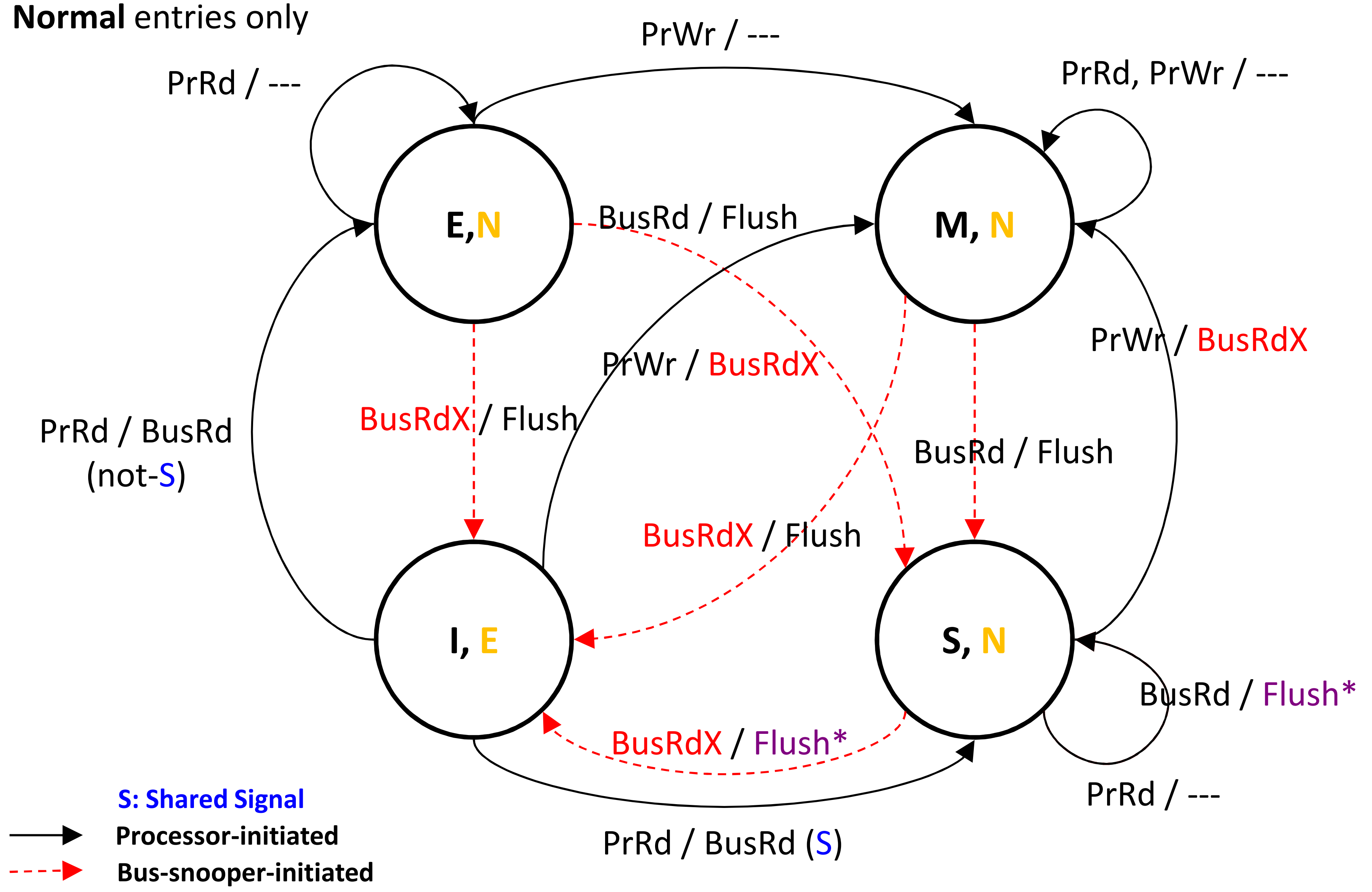
Processor-initiated

Bus-snooper-initiated

Flush*: Flush for data supplier; no action for other sharers

T Cache – Normal Mode

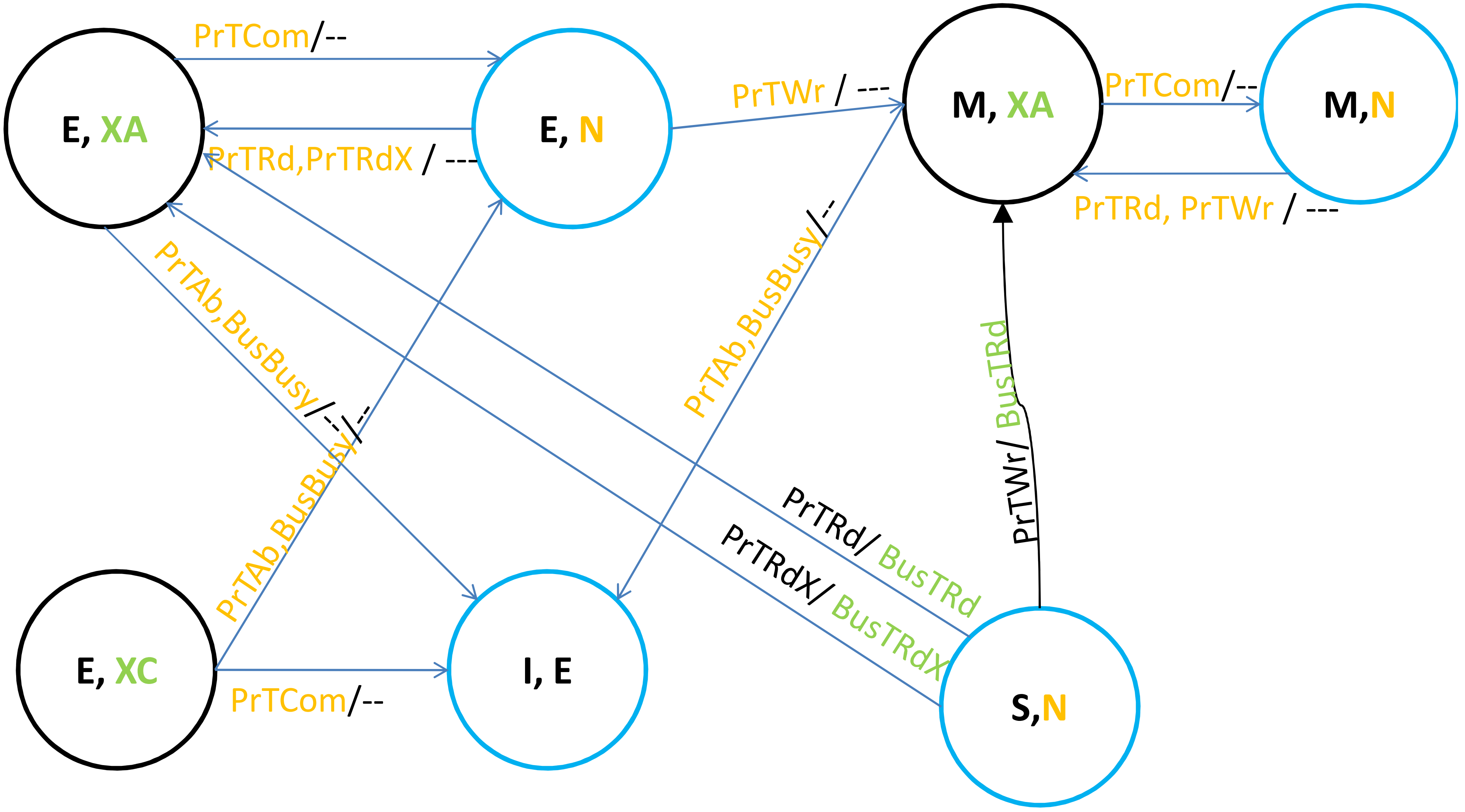
Idea: Act as L1 Cache for **Normal** entries only



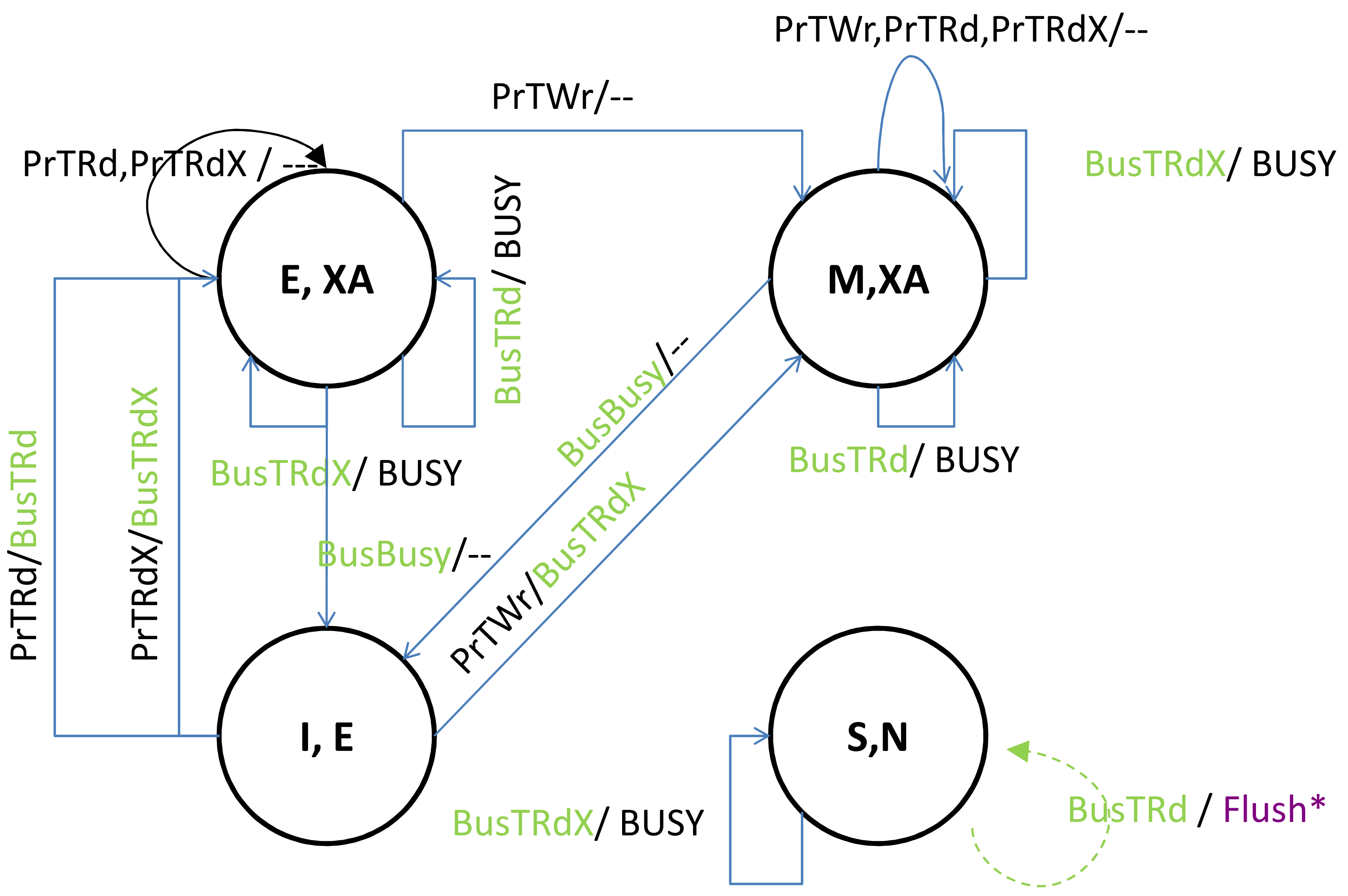
S: Shared Signal

Flush*: Flush for data supplier; no action for other sharers

T Cache: Normal ⬄ Transactional



T Cache: Transactional Mode



→ **Processor-initiated**

⇢ **Bus-snooper-initiated**

Flush*: Flush for data supplier; no action for other sharers

Question:
What happens when data resides on L1 Cache and we need to perform a transaction ?

Can we classify hardware transactional memory?

- **Version management** – at most one copy of data can be held in place, others must be in some other buffer etc.
  - Lazy: new values stored elsewhere
  - Eager: new values stored in place ← Fast commit
- **Conflict management**
  - Lazy: At commit
  - Eager: As soon as there is a conflict ← Less wasted work

(these classifications given by Moore et al)

Can we classify hardware transactional memory?

- In databases transactions usually reduce isolation to improve performance
  - Communication from within uncommitted transactions!
- **Closed-nested transaction:** Results of children visible only to parent
- **Open-nested Transaction:** Results of children visible globally
- **Linear/Flat nesting:** One top-level transaction
- **No nesting:** Transactions are not allowed to nest

Can we classify hardware transactional memory?

- ***Weak atomicity:*** *transactions are not atomic with* respect to non-transactional accesses

- ***Strong atomicity:*** *transactions are atomic with r*espect to non-transactional accesses

Herlihy & Moss [93]

- **Version Management:** Lazy
- **Conflict Management:** Eager
- **Nesting:** Flat
- **Atomicity:** Weak

Resource Limitations

- Space
  - Eviction: cache is full
  - Collision: too many lines map to same set
  - Transactional cache overflow
- Time
  - Thread preempted by timer interrupt, I/O, etc.

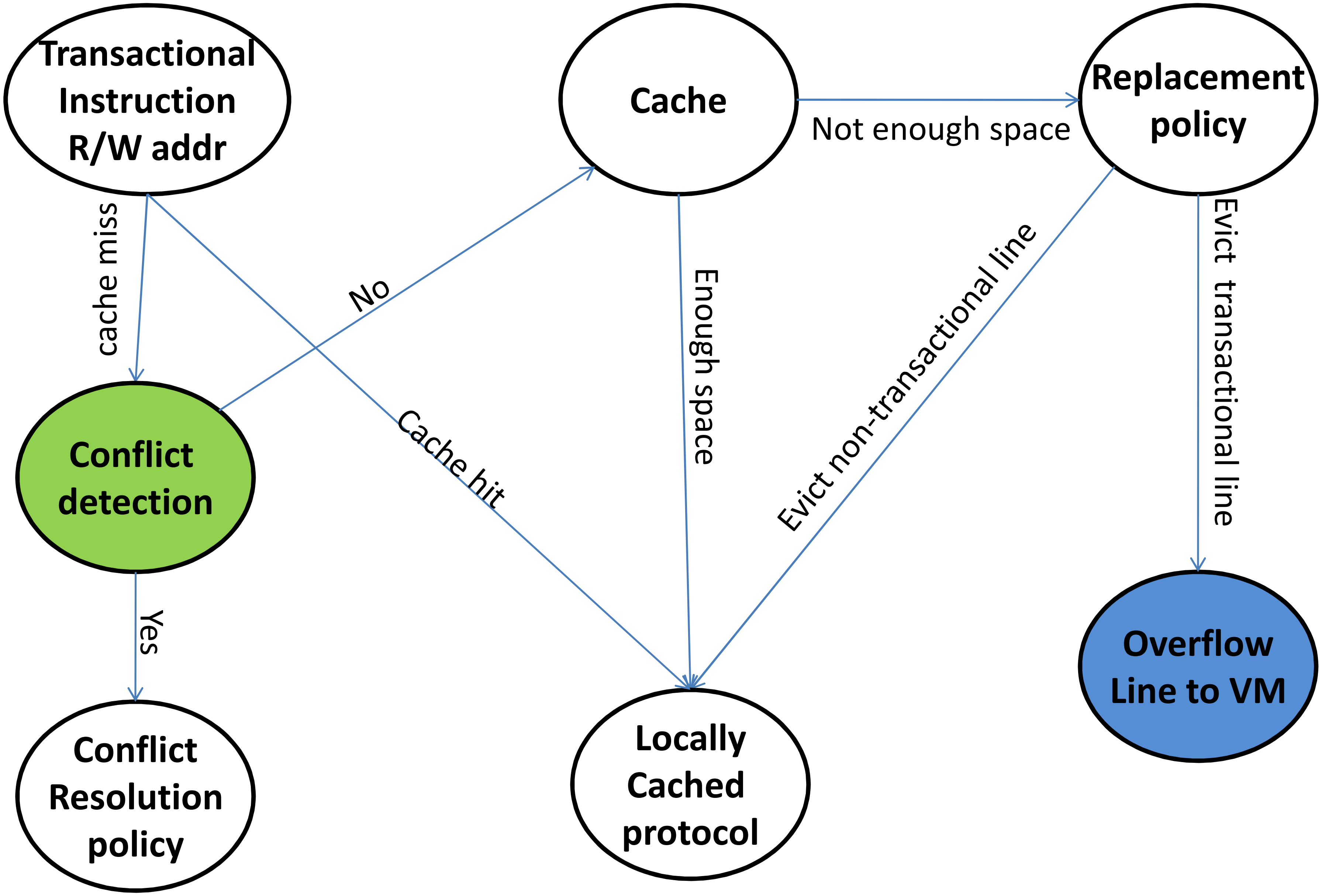
Virtualizing Transactional Memory Herlihy

- Hardware **+** Software architecture
- Use pure HTM while “resources” are not exhausted
- Use Software to hide resource exhaustion
  - In **space**
  - and **time**

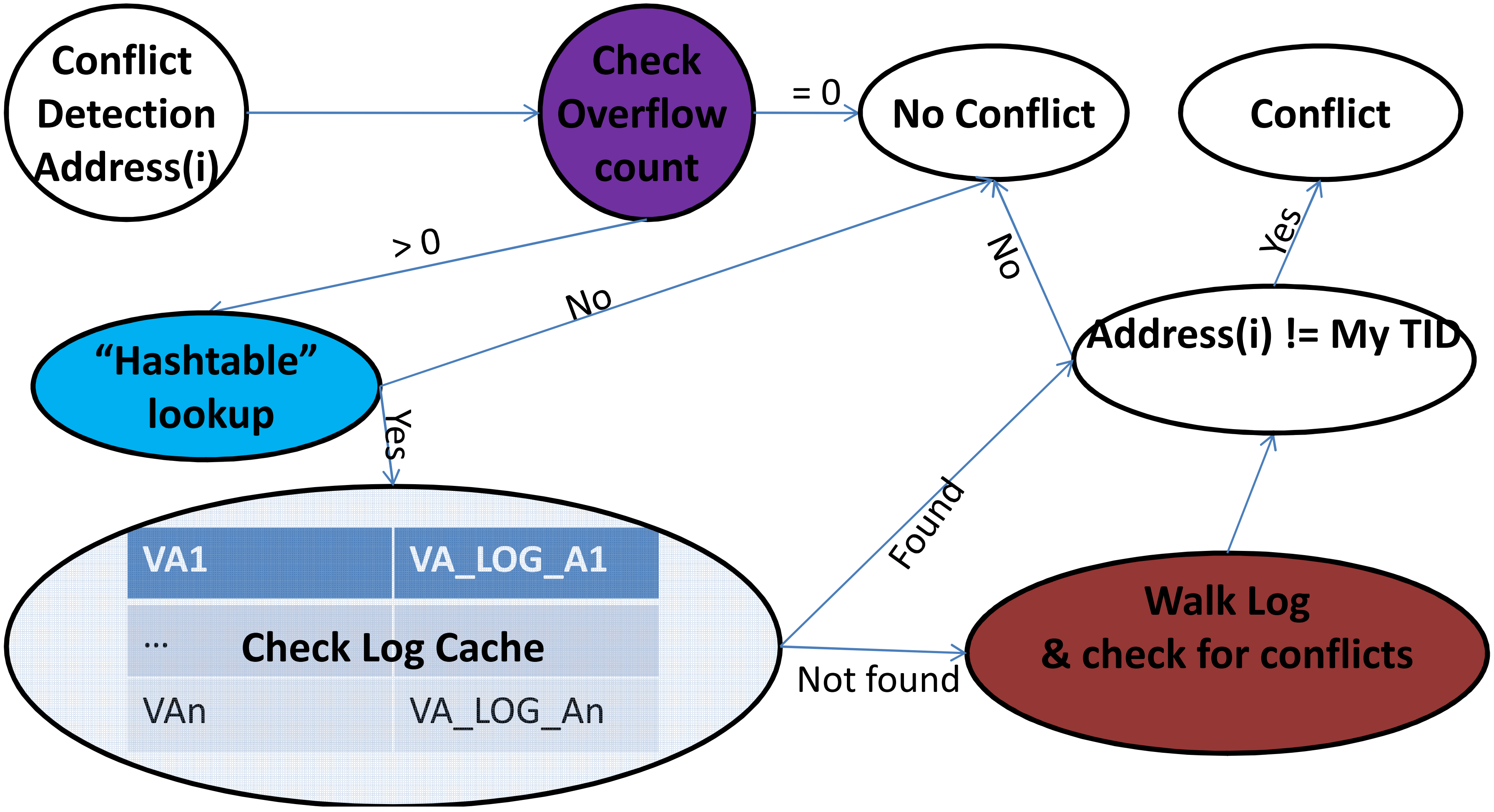
VTM States

- **Locally cached state:** Use CPU Tcache only
- **Overflowed state:** Use VM user-space to temporarily store modified locations
- Pure HTM detected conflicts via an **extended** cache coherence protocol
- **Challenge:** Detect conflicts in the o**verflowed state when …**
  - A process gets swapped out
  - Transactional data (logs) has overflowed to **VM**

Key idea


Transactional Instruction R/W addr
Cache
Replacement policy
Not enough space
cache miss
No
Cache hit
Enough space
Evict non-transactional line
Evict transactional line
Conflict detection
Yes
Conflict Resolution policy
Locally Cached protocol
Overflow Line to VM

Conflict detection: Lazy



Key Components

- Overflow counter – Is there an oveflow ?
- "Hashtable" -- Fast check if **addr** belongs in **log**
- Log cache – Fast translation between **addr** and **log addr**
- **Log:** Resides in VM and performs
  Lookup(addr) , Add(addr), Commit , Abort,
  Save state on swap

Key components

- **Overflow counter:** Implemented as a software counter which resides in the VM
- **Log cache (ADC) :** Implemented in h/w . Update on **overflow , swap** and **restore**
- **The Log** has a S/W component which resides in VM and a h/w component the **log walker**

Key components: “Hash table” (XF)

- Implemented as a **Counting Bloom filter:** add(x), member(x) ➔ false positives, remove(x)

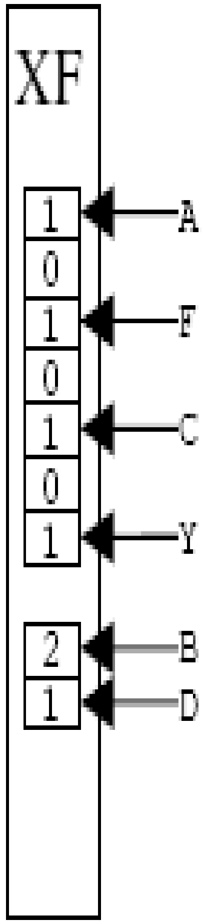
- **add(X) ➔**

{for i<k:let offset = hash[i](X) in bucket[offset]++}

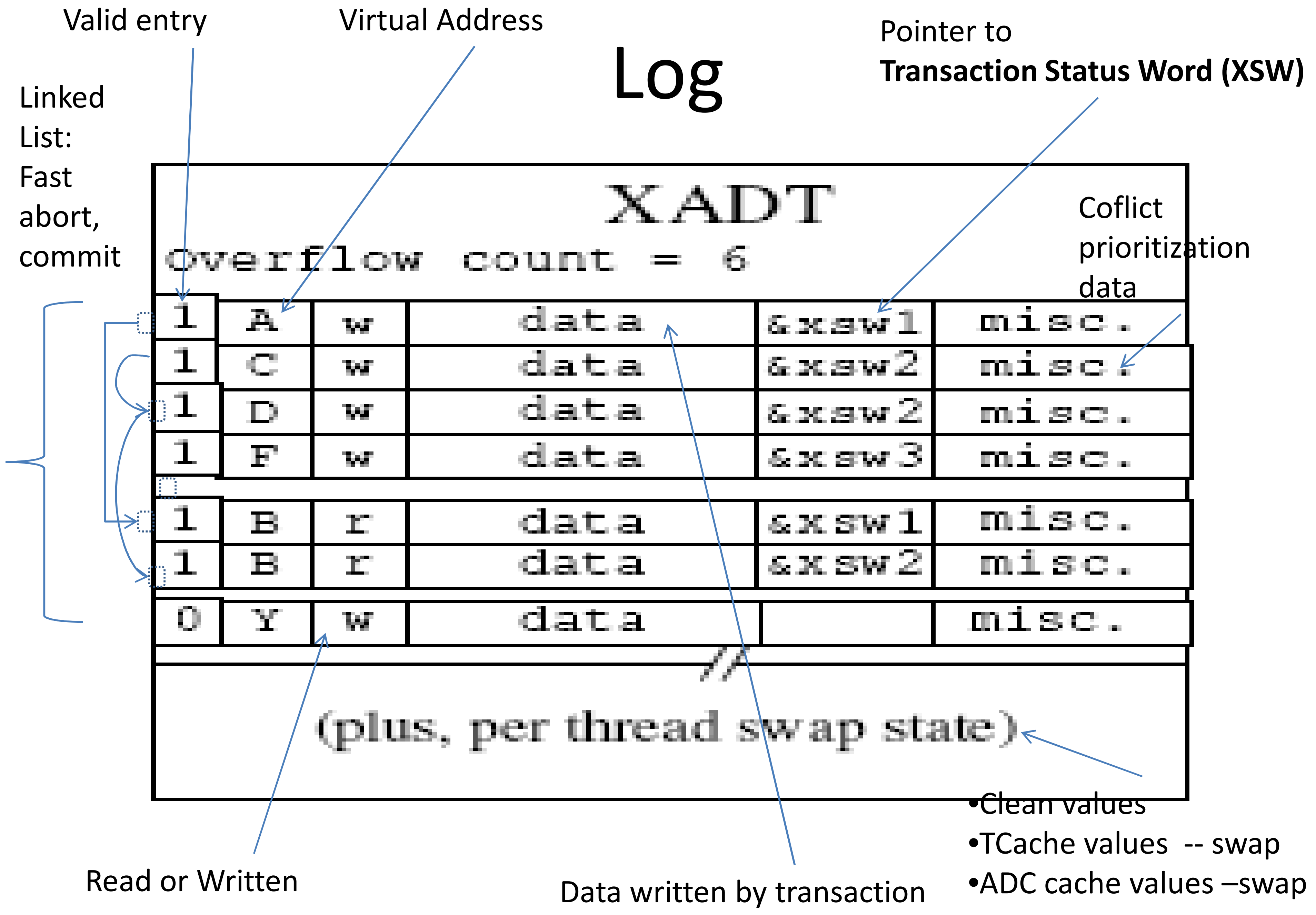
- **remove(x)** same as add but --

- **member(x) ➔**

{for i<k:let offset = hash[i](X) in
If **!**bucket[offset] then ret false} ; ret true;



Log


Valid entry
Virtual Address
Pointer to **Transaction Status Word (XSW)**
Linked List: Fast abort, commit
Coflict prioritization data
Read or Written
Data written by transaction
•Clean values
•TCache values -- swap
•ADC cache values –swap

XADT

Overflow count = 6

1	A	w	data	&xsw1	misc.
1	C	w	data	&xsw2	misc.
1	D	w	data	&xsw2	misc.
1	F	w	data	&xsw3	misc.
1	B	r	data	&xsw1	misc.
1	B	r	data	&xsw2	misc.
0	Y	w	data		misc.

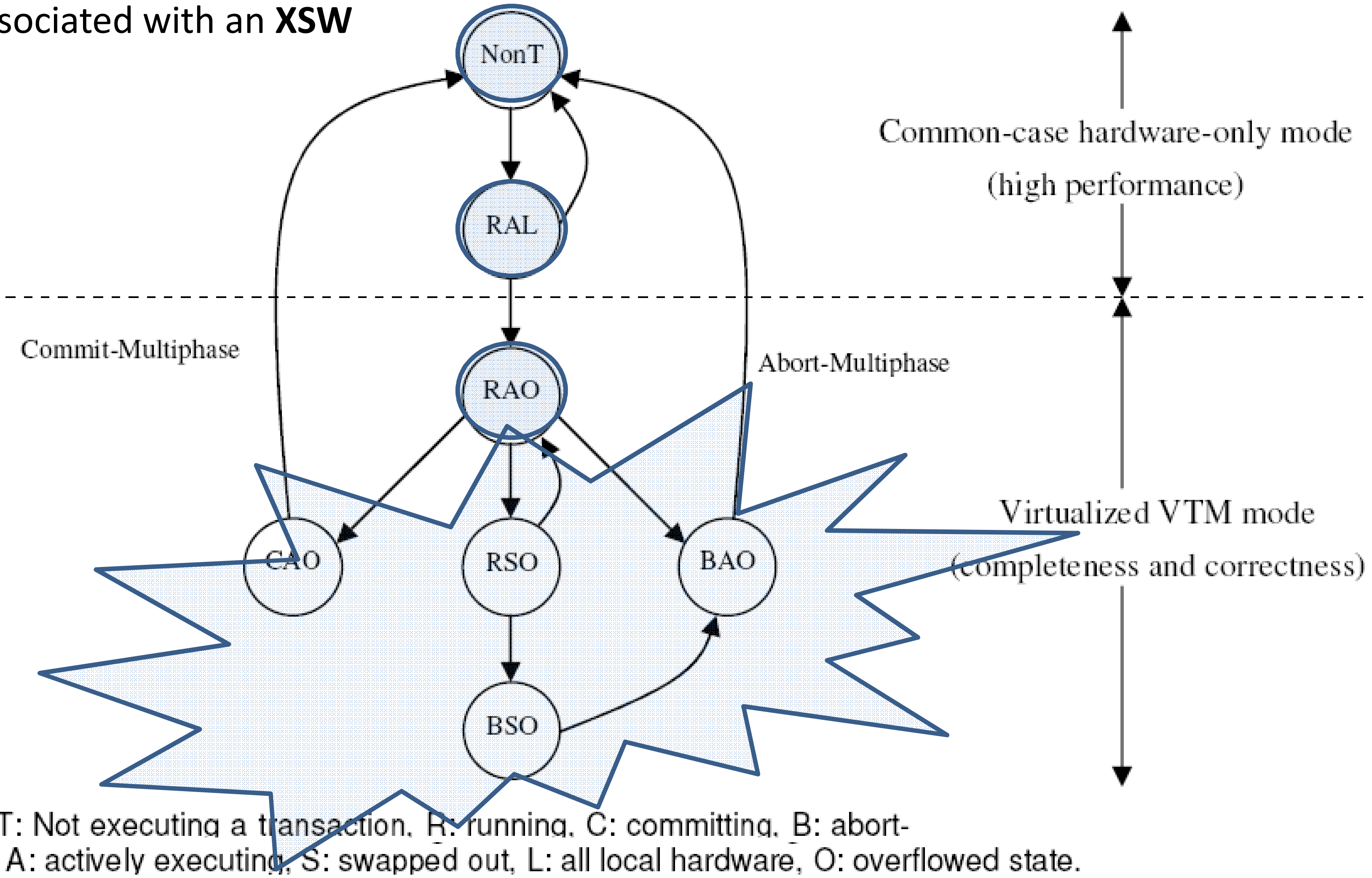
(plus, per thread swap state)

Overflow Algorithm

1. Allocate new XADT entry
2. Update counter
3. Update XF
4. Cache new entry addr translation into ADC
5. Send **BusTRdX(addr)** so that other CPUs with **locally cached addr** (not overflowed) are notified

Transaction States

Each thread is associated with an **XSW**



NonT: Not executing a transaction, R: running, C: committing, B: aborting, A: actively executing, S: swapped out, L: all local hardware, O: overflowed state.

Transaction Swapping & Restore Algorithm

On swap:

1. Overflow Tcache to Log (VM) --- Clean and tenant values
2. Change state (XSW=RAO ➔ XSW=RSO)

On restore:

1. Repopulate
   - ✓ Tcache
   - ✓ ADC
2. Check if state (XSW) has changed
   1. If XSW == BSO then XSW = BAO ; Abort()
   2. else XSW = RAO;

Physical & Logical Commits & Aborts

- Idea: Other transaction only care about "logical" commits/aborts i.e. XSW

- To logically commit/abort use CAS to modify XSW
- **Physical abort:** Traverse transactions' **linked list** one by one and commit

- Other transactions must **wait** for physical commit – Especially slow on **page fault**

- An active transaction may abort a swapped transaction on conflict by modifying XSW with CAS

Strong atomicity Implications

- On commit the CPU needs to <u>compare</u> **clean values of log** to original values
  - Abort if differ
- On resume from swap the CPU performs the same task as above

- On physical commit **non-transactional instructions** need to access the XADT – cannot read directly from VM --- slow

VTM Characteristics

- **Version Management:** Lazy
- **Conflict Management:** Eager
- **Nesting:** Flat
- **Atomicity:** Strong

Advantages

- Faster than STM
- Resolves the resource exhaustion issue
  - Time
  - Space

Disadvantages

- Lots of overhead
- Swaps are extremely expensive
- VA-based approach slow
- Physical aborts/commit : Other transactions including non-transactions must wait for a physical abort to end
- It does not work with OS which map the same PA to different VA (mmap)

Point to ponder:

Would it be faster to implement the VTM in “kernel space”?

- Pin down Log in processes’ **working set**

Unbounded Transactional Memory

Scot Ananian, Krste Asanovic, Bradley Kuszmaul, Charles Leiserson, Sean Lie

Prior work compared to **VTM**

Key idea: Use hardware modifications and software support to resolve the resource exhaustion issue

Two models:

- **Unbounded Transactional Memory:** Key idea ➔ Use VM, requires more hardware modifications and complicated software
- **Large Transactional Memory:** Key idea ➔ Use PM with less…

Key differences with VTM

- Cache coherence protocol remains intact
- Weak atomicity
- Explicit instruction to initiate xaction (XBEGIN)
- Explicit abort handler: if(**b**) then **xaction** else **abort** ➔ speculatively execute xaction otherwise execute abort
- Eager version management

Key similarities with VTM

- XEND = Commit
- A **log** maintained in VM
- CPU requires additional instructions
- Modified cache h/w
- Flat transactions
- Eager conflict management

Hardware modifications

1. Take snapshot of RF
2. Preserve the "right" physical registers

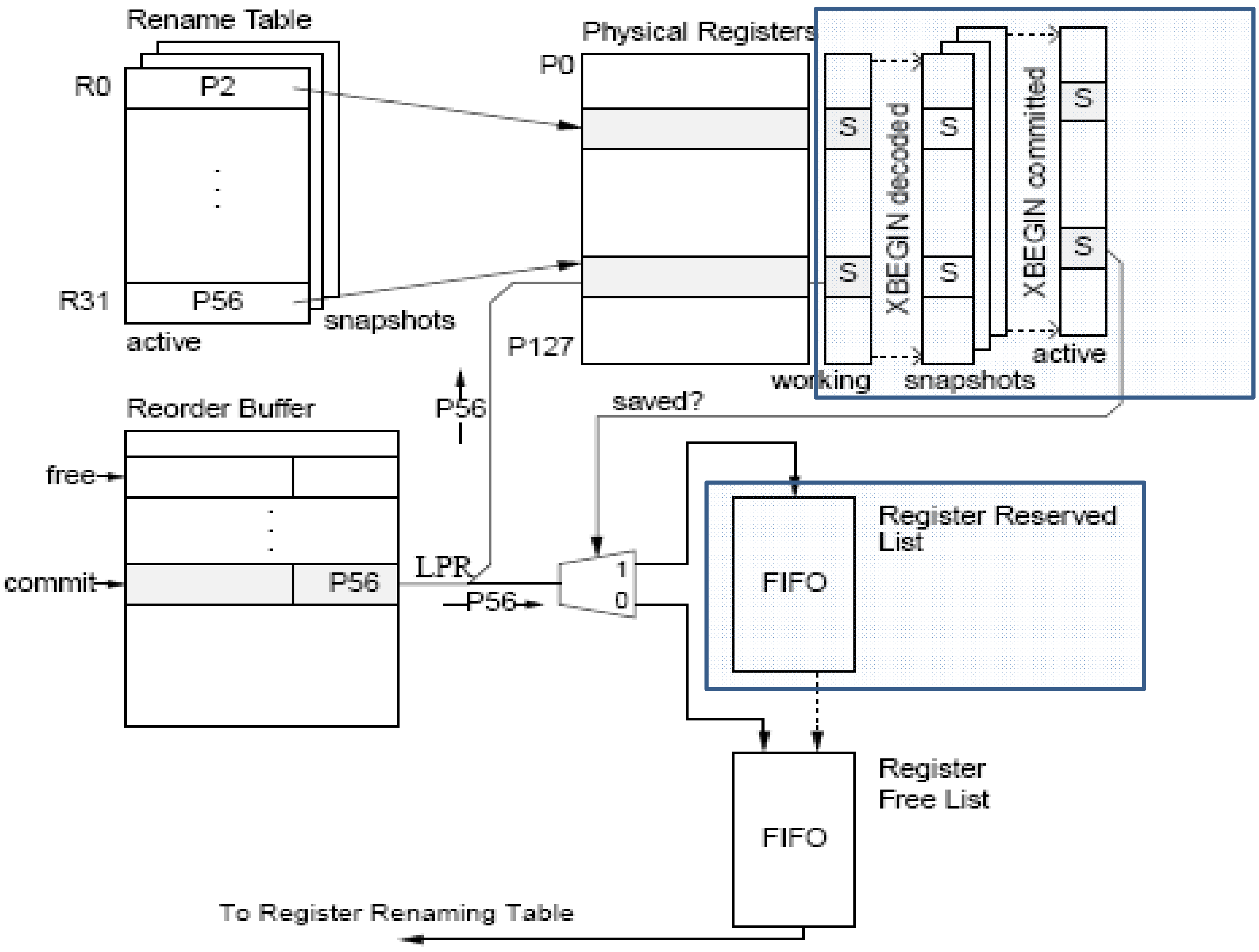
- **Abort handler** acts as an "else" branch. Need to recover **architectural registers** if "abort" branch taken

- As in speculative execution take a **register-renaming table** snapshot when XBegin graduates ROB

- Ensure that **physical** regs not **overwritten** by adding an "saved" bit. Keep "saved" bit on until Xend.

- **On instr graduation:** "saved" physical regs ➔ **Reserved list**, otherwise move physical reg to **Freed list**. If(instr = XEND) move RL to FL

- On "misprediction" (i.e. abort) restore snapshot

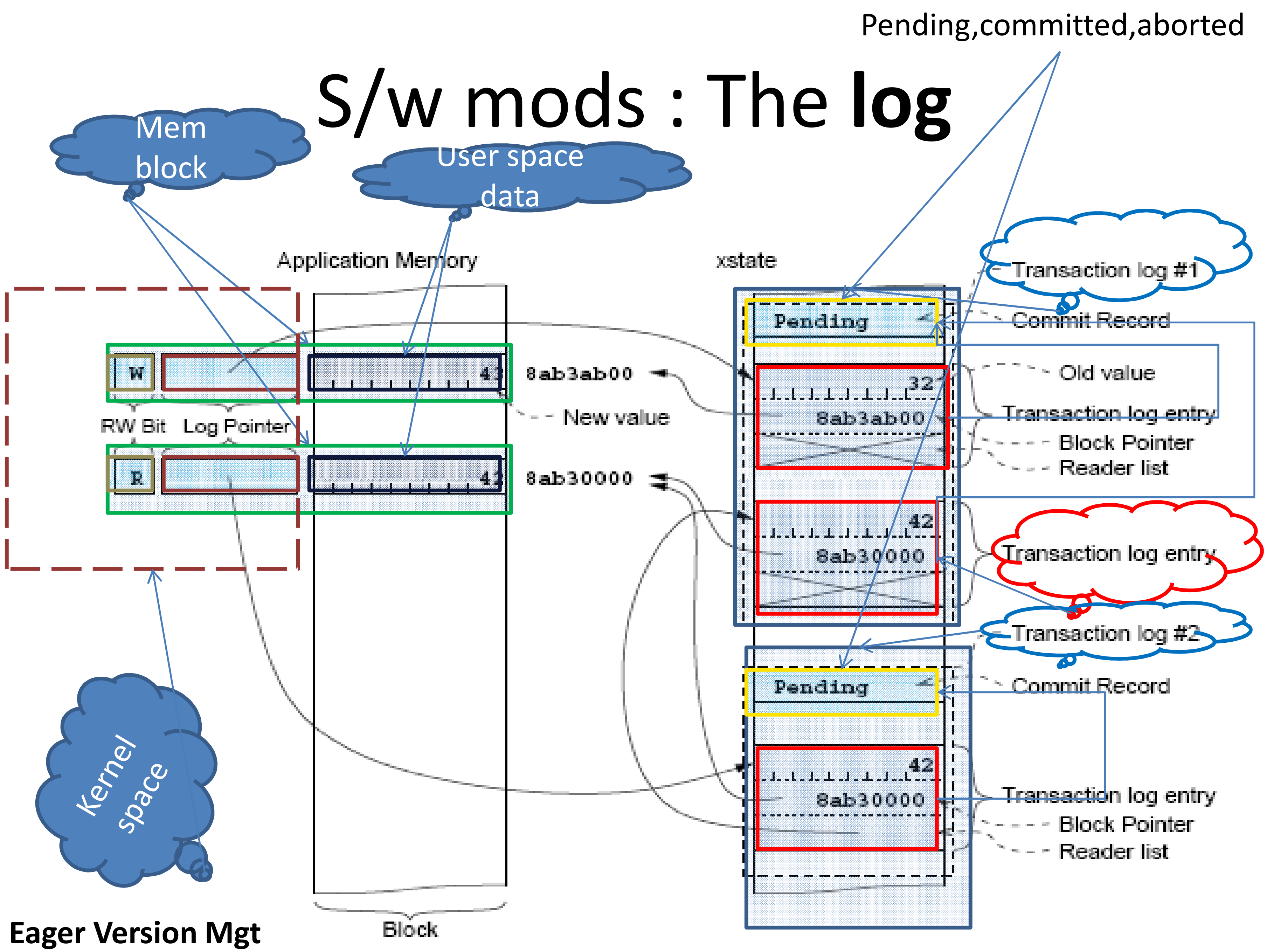


Rename Table
R0
P2
R31
P56
active
snapshots
Physical Registers
P0
P127
S
S
working
XBEGIN decoded
S
S
snapshots
XBEGIN commited
S
S
active
Reorder Buffer
free
commit
P56
P56
LPR
P56
saved?
1
0
FIFO
Register Reserved List
FIFO
Register Free List
To Register Renaming Table

More h/w modifications

- Add **log ptr** to **CPU state** so that a process can migrate within CPUs.
- Two additional **control registers = (base,size) thread's log**
- **Note:** No cache modifications needed

S/w mods : The **log**



Eager Version Mgt

XBegin , XEnd

- XBegin generates a **new log**
- Xend commits modifications and deallocates log
- A fault occurs when an I/O instruction gets executed within a transaction

Conflict detection

- **Conflict:** Multiple accesses in a single block and at least one **write**
- Conflict detection on **read instruction**:
  - Mem_blk.status = W & Mem_blk.ptr != this xaction
- Conflict detection on **write instruction:**
  - Mem_blk.ptr != NULL &Mem_blk.ptr != this xaction
- **On conflict** use timestamps to determine which xaction has priority. Others **abort**
- **Abort easy:** traverse the log entries for this mem block.

Write/Load

- Check conflict
- If !(exists log entry for this mem block)
  - If log not large enough, enlarge log and abort (**retry**)
  - allocate a new log entry (save contents etc.)

- Set mem_block.ptr = log entry of my xaction

- On **write** perform in-place update

- **Commit:** *mem_block.ptr->status = COMMITED

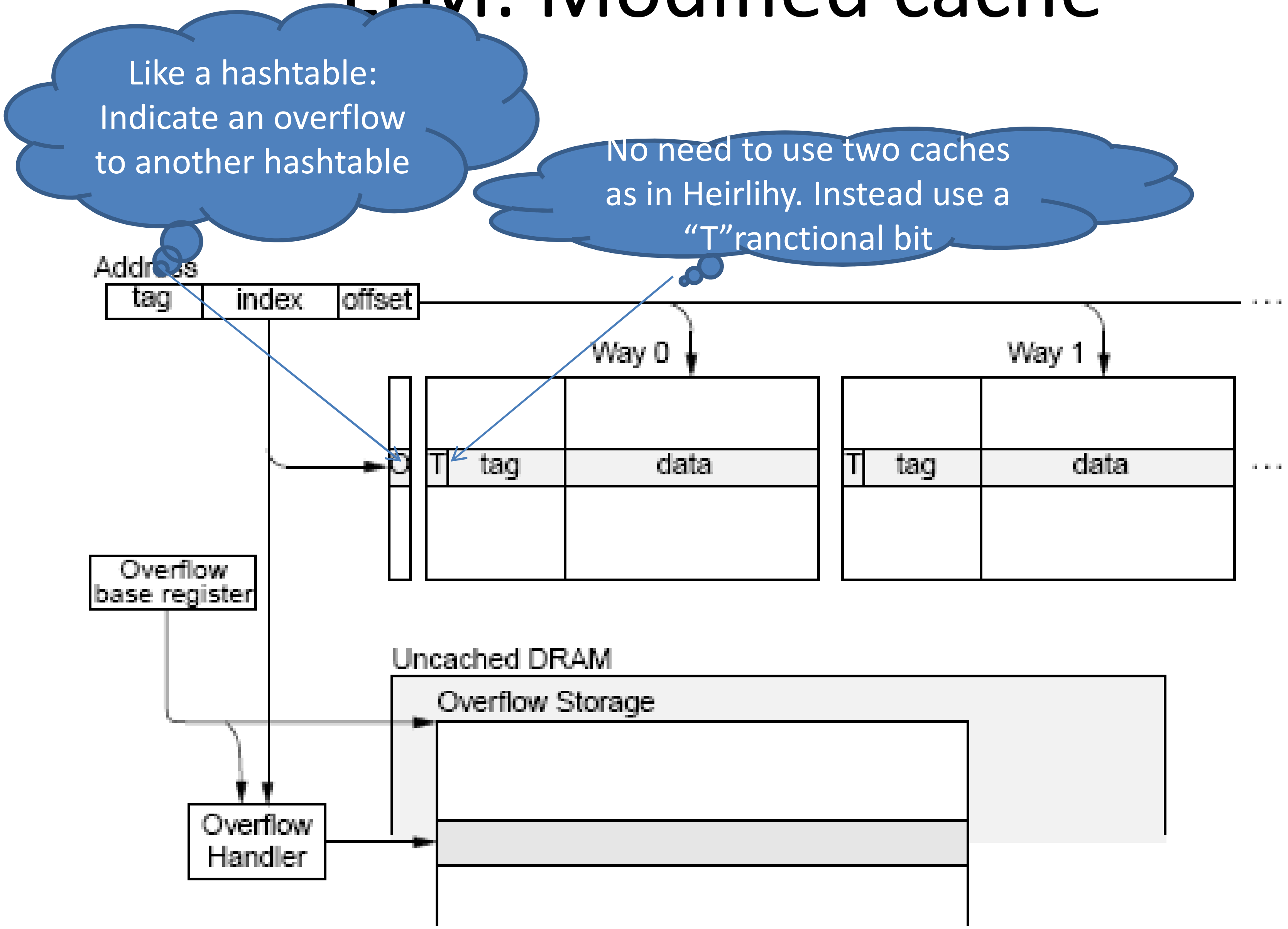
UTM Characteristics

- **Version Management:** Eager
- **Conflict Management:** Eager
- **Nesting:** Flat
- **Atomicity:** Weak

LTM

- Lightweight version of UTM
- Large but not Unbounded
- **Limitations:**
  - Size <= Physical Memory
  - Lifetime <= 1 time-slice
  - Transactions cannot migrate within CPUs
- Similarities with Heirlihy's original HTM:
  - Use cache-coherence protocol
- **Overflows are allowed:** Use a hash table in PM
- **Advantages:** Few h/w mods

LTM: Modified cache


Like a hashtable: Indicate an overflow to another hashtable
No need to use two caches as in Heirlihy. Instead use a “T”ranctional bit
Address
tag
index
offset
Way 0
Way 1
O
T
tag
data
T
tag
data
Overflow base register
Uncached DRAM
Overflow Storage
Overflow Handler

Modified cache

- If tag = O then compare cache tag to addr tag
  - If not equal (**miss**)search overflow table
  - Swap the overflow table line with old cache line

- Conflict mgt: Similar to HTM. If lines overflow search hash table bucket linearly to determine conflict

LTM Characteristics

- **Version Management:** Lazy
  - Mods in cache and overflow table
- **Conflict Management:** Eager
- **Nesting:** Flat
- **Atomicity:** Weak

Related work

- **TRANSACTIONAL COHERENCE AND CONSISTENCY: SIMPLIFYING PARALLEL HARDWARE AND SOFTWARE**
  - Overflow: stall other threads (transactional [overflowed or not]or not) on overflow. No need for complicated conflict protocol.
- **Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory**
  - Can abort an overflowed transaction. Stalls **only** overflowed threads/transactions on oveflow. Non transactional threads always executed in parallel.

Related Work

- **Hybrid Transactional Memory**
  - Execute xactions entirely in **h/w**
  - Executed entirely in software on overflow
  - Much simpler approach, less h/w mods

- **PhTM: Phased Transactional Memory**
  - Use multiple approaches and adapt according to current environment and workload

Related Work

- **LogTM-SE: Decoupling Hardware Transactional Memory from Caches**
  - Use signatures to summarize read/write sets
  - Accelerate conflict detection
- **An effective hybrid transactional memory system with strong isolation guarantees**
  - HyTM
  - Signatures
- **Unrestricted Transactional Memory:**
  Supporting I/O and System Calls within Transactions
  - Deal with I/O

Thanks for your attention !