ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Transactional Memory

Architectural Support for Lock-Free Data Structures

Maurice Herlihy and J. Eliot B. Moss

1993

Presenter: Giulio Valente

Lock-Free

A shared data structure is *lock-free* if its operations do not require mutual exclusion

Motivation

Problem: Multi-core systems are hard to program

Motivation

Problem: Multi-core systems are hard to program

Transactional Memory solves the problem by:

- Makes parallel programming easier by simplifying coordination
- Exploit hardware support for performance
- Implements *lock-free* data structures

Common problems with locking techniques

- **Priority Inversion**

  when a lower-priority process is preempted while holding a lock needed by higher-priority processes

- **Convoying**

  a process holding a lock is descheduled

- **Deadlock**

  processes attempt to lock the same set of objects in different orders

Transaction

- **Assumption**: a process executes only one transaction at a time

- Finite sequence of instructions
- Executed by a single process
- Satisfies *serializability* and *atomicity*

TM primitive instructions

- *Load-transactional* (LT): reads value of a shared memory location into a private register
- *Load-transactional-exclusive* (LTX): similar to LT, but indicates that is likely to be updated
- *Store-transactional* (ST): tentatively writes a value to a shared memory location, but not visible until a successful commit.

Transaction's sets

- **Read set**: set of locations read by LT
- **Write set**: set of locations accessed by LTX or ST
- **Data set**: Read set U Write set

TM instructions for transaction state

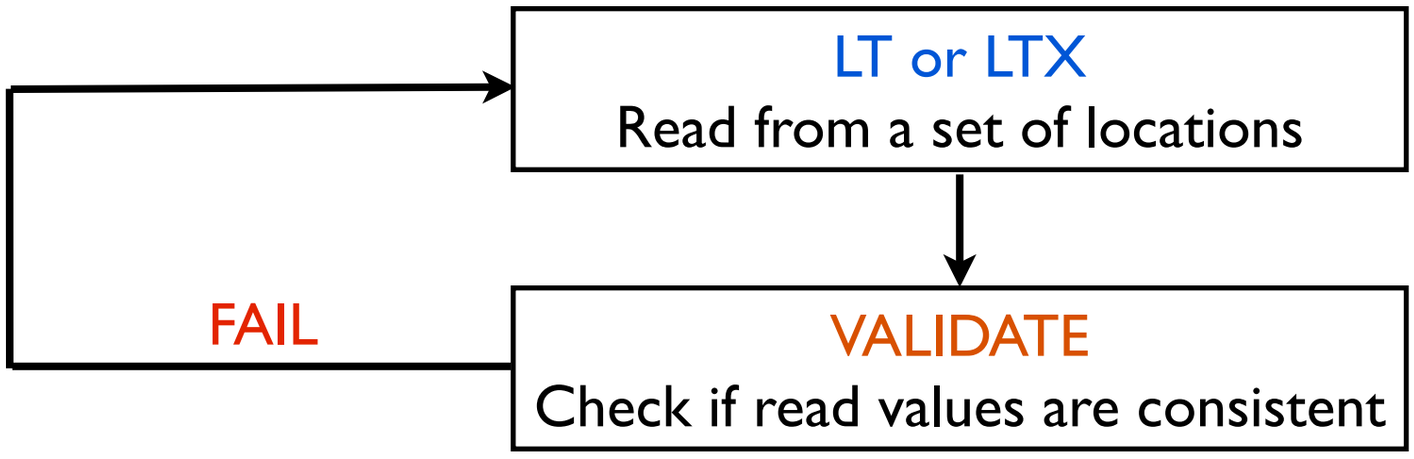
- Commit (COMMIT) attempts to make the transaction's tentative changes permanent.
- Abort (ABORT) discards all updates
- Validate (VALIDATE) tests the current transaction status. Return TRUE or FALSE.

Intended Use

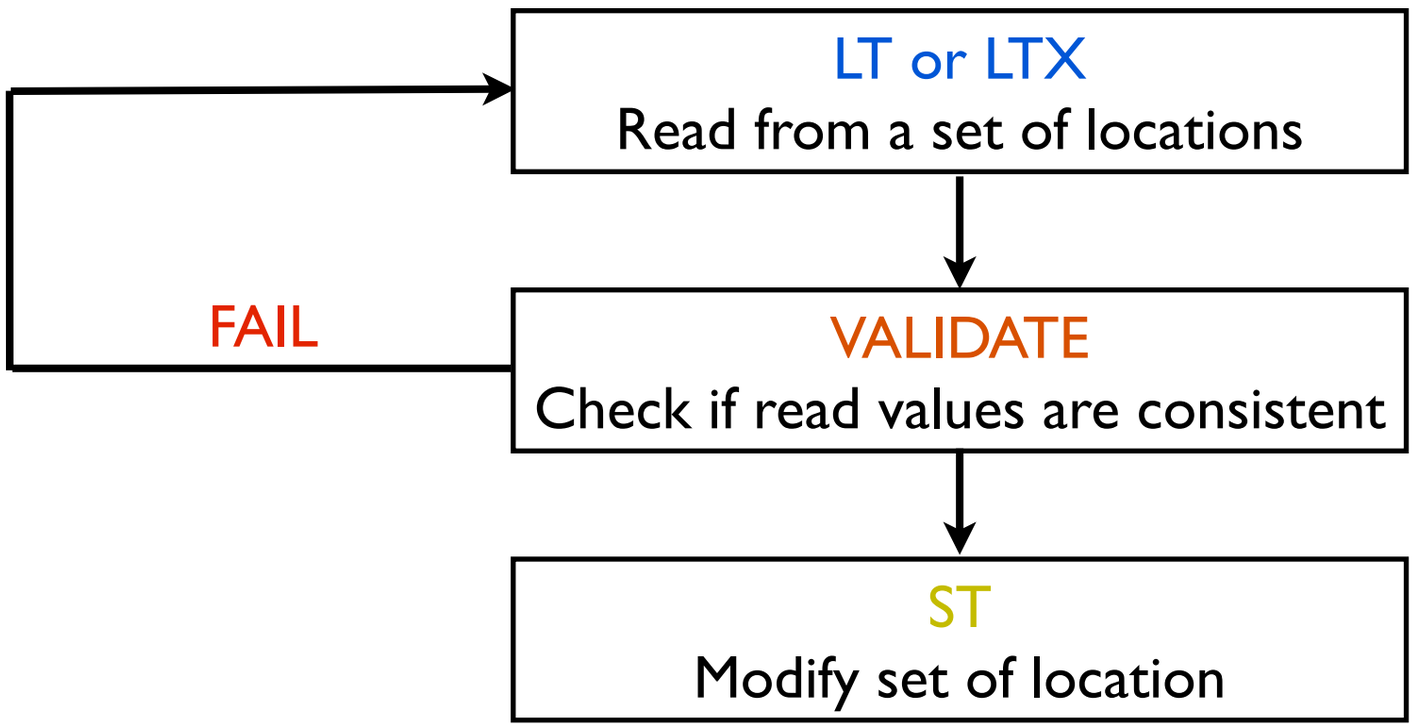
LT or LTX

Read from a set of locations

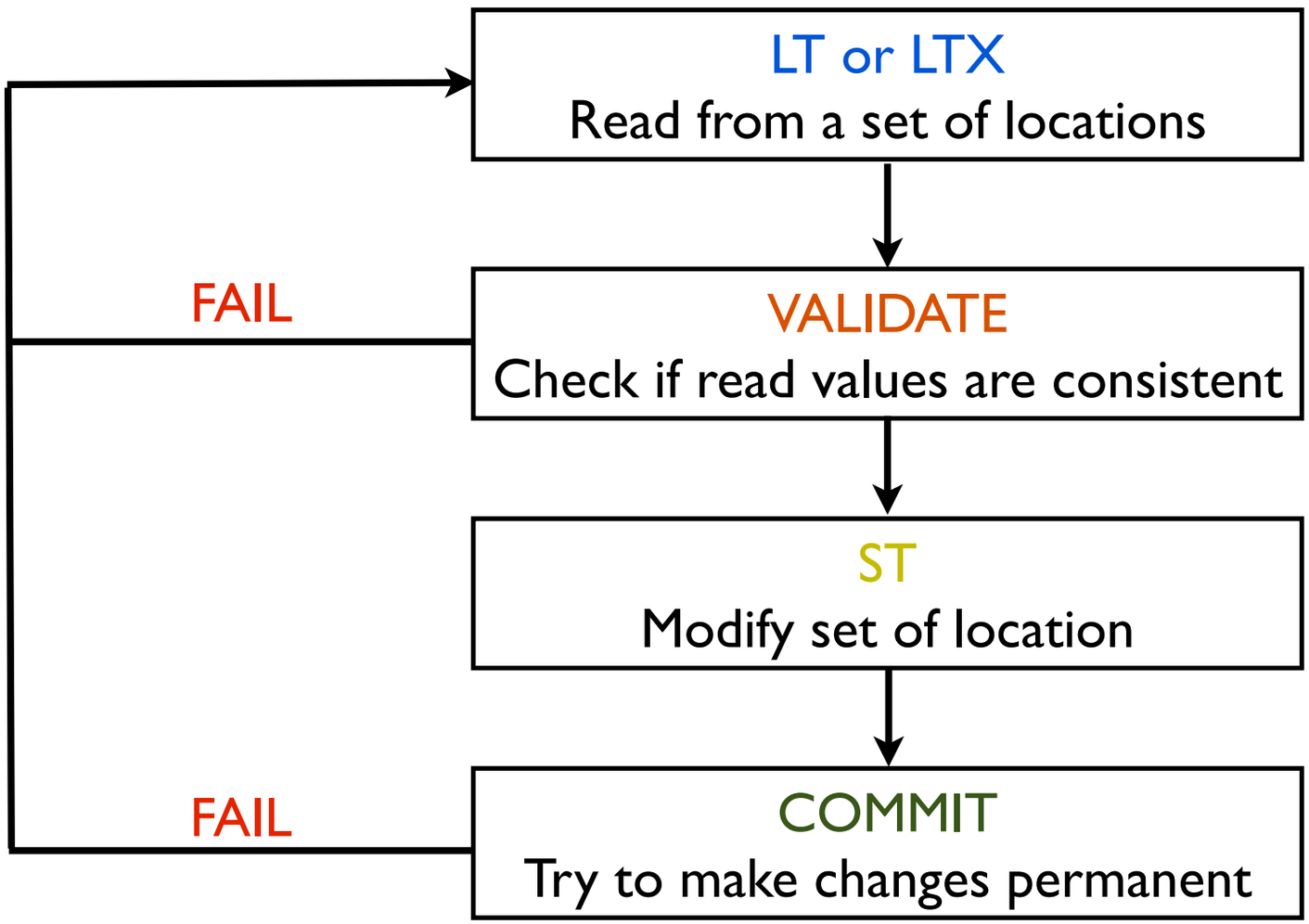
Intended Use



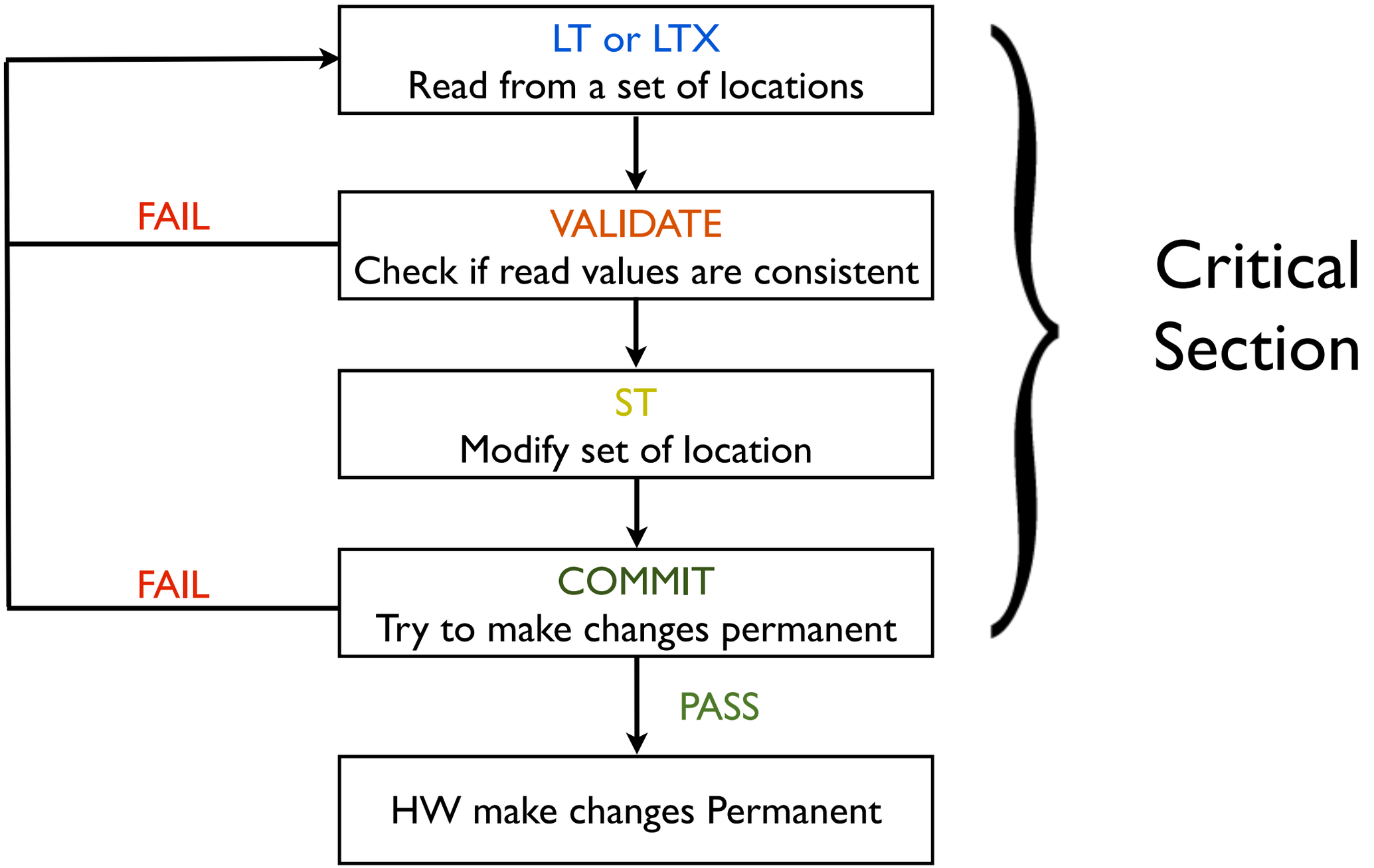
Intended Use



Intended Use



Intended Use



TM implementation

- Implemented by modifying standard multiprocessor cache coherence protocols
- Basic idea: any protocol capable of detecting accessibility conflicts can also detect transaction conflict at no extra cost
- Transactional cache holds all the tentative writes, without propagating them to other processors or to main memory unless the transaction commits

Cache-Coherence Protocol

- Intended for managing caches of a multiprocessor system
- Each processor may have its own memory cache, which is separated from the shared memory
- Cache coherence is intended to manage conflicts and mantain consistency between cache and memory

Caches in TM

- Each processor maintains two caches:
  - Regular cache for non-transactional operations
  - Transactional cache for transactional operations

Cache line states

Each cache line has one of the following states

Name	Access	Shared?	Modified?	Meaning
INVALID	none	-	-	Incoherent copy of the memory
VALID	R	Yes	No	Coherent copy of the memory
DIRTY	R, W	No	Yes	Incoherent and only copy
RESERVED	R, W	No	No	Coherent copy, but is the only copy

Transactional Cache States

The transactional cache expends these states with transactional tags

Name	Meaning
EMPTY	contains no data
NORMAL	contains committed data
XCOMMIT	discard on commit
XABORT	discard on abort

Bus Cycle

Name	Kind	Meaning	New Access
READ	regular	read value	shared
RFO	regular	read value	exclusive
WRITE	both	write back	exclusive
T_READ	transactional	read value	shared
T_RFO	transactional	read value	exclusive
BUSY	transactional	refuse access	unchanged

Original in Goodman's protocol

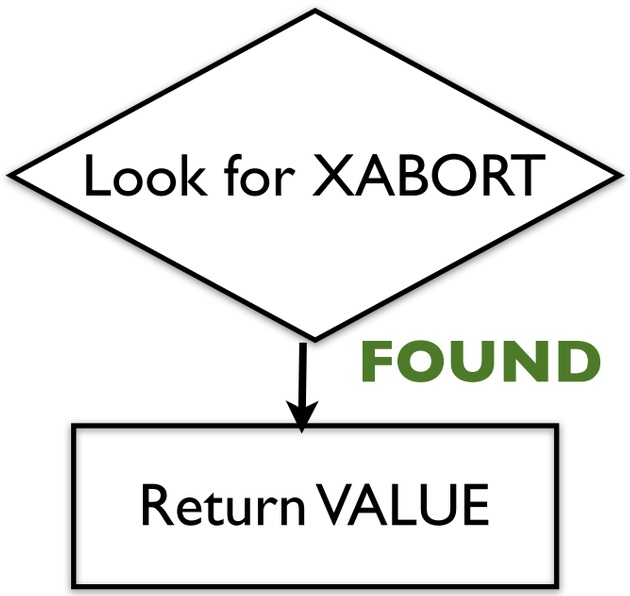
Added by TM

LT Instruction

Operations for an active transaction

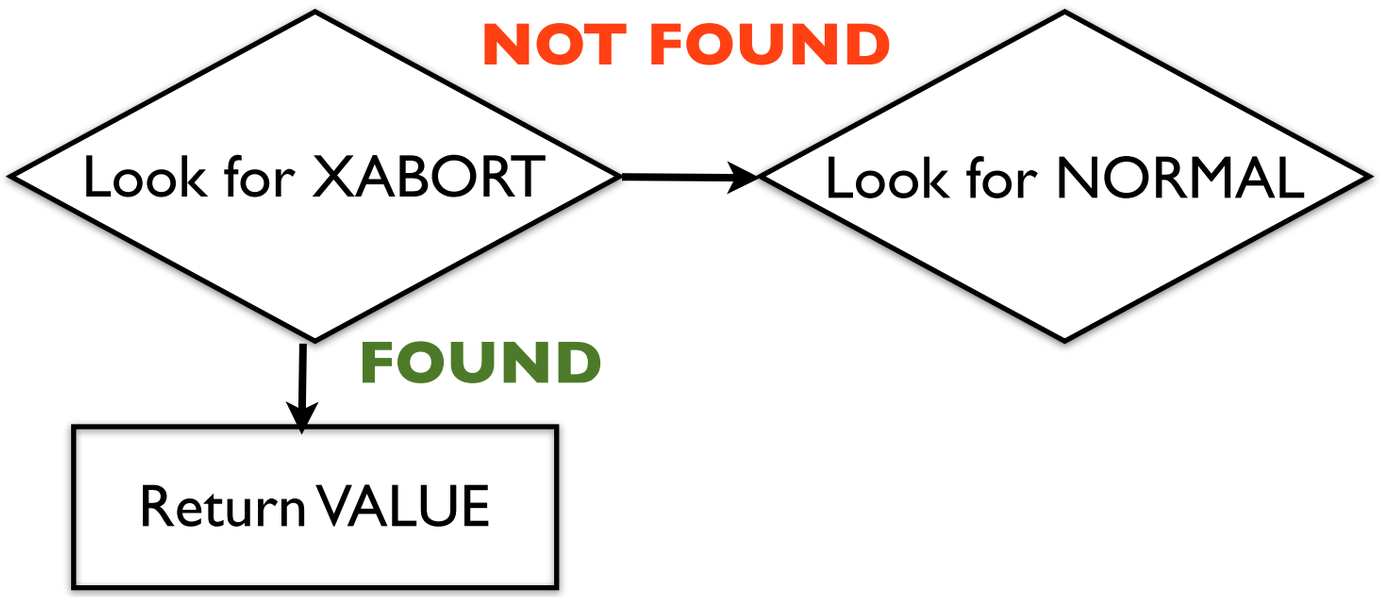
LT Instruction

Operations for an active transaction



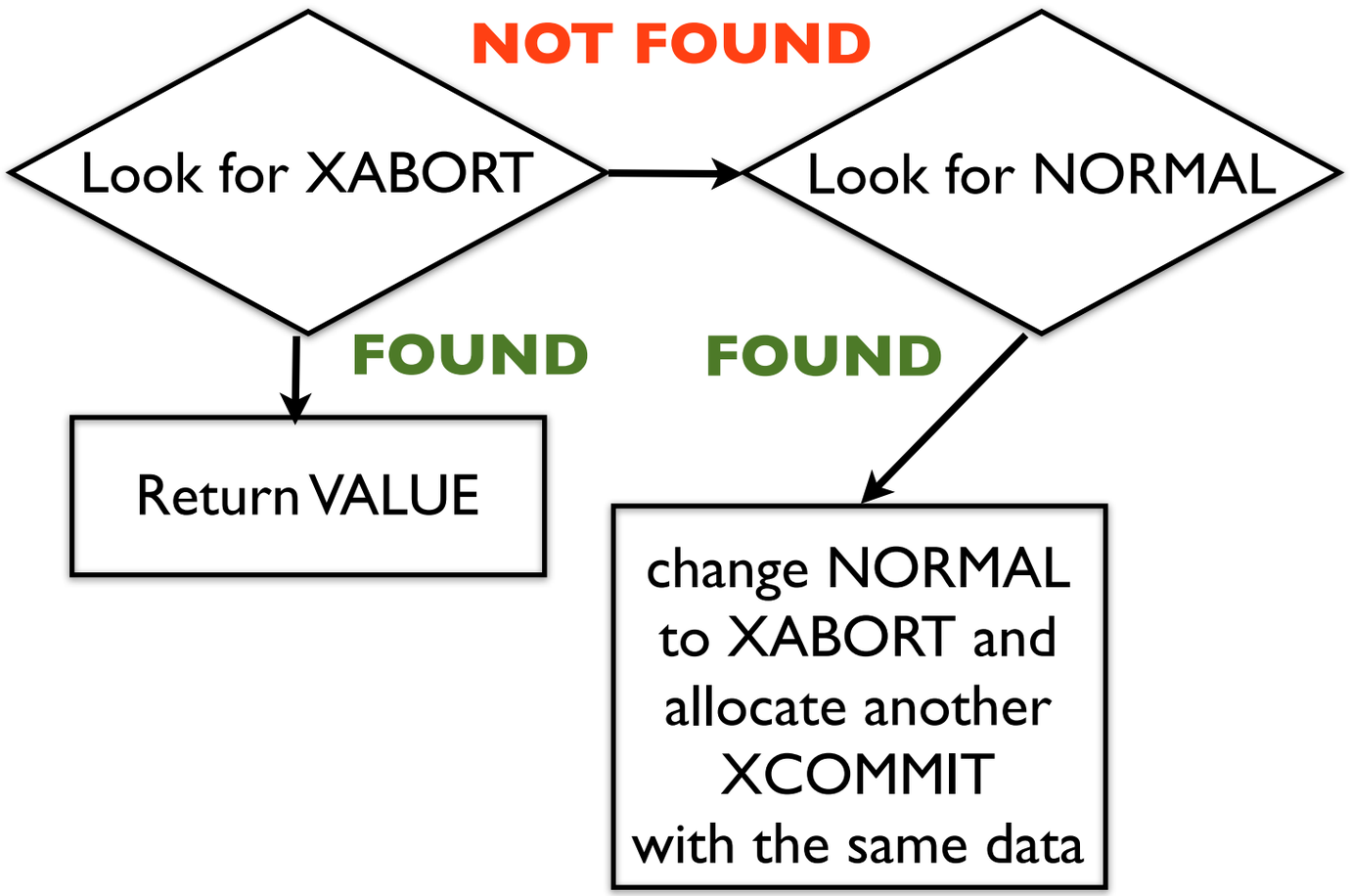
LT Instruction

Operations for an active transaction



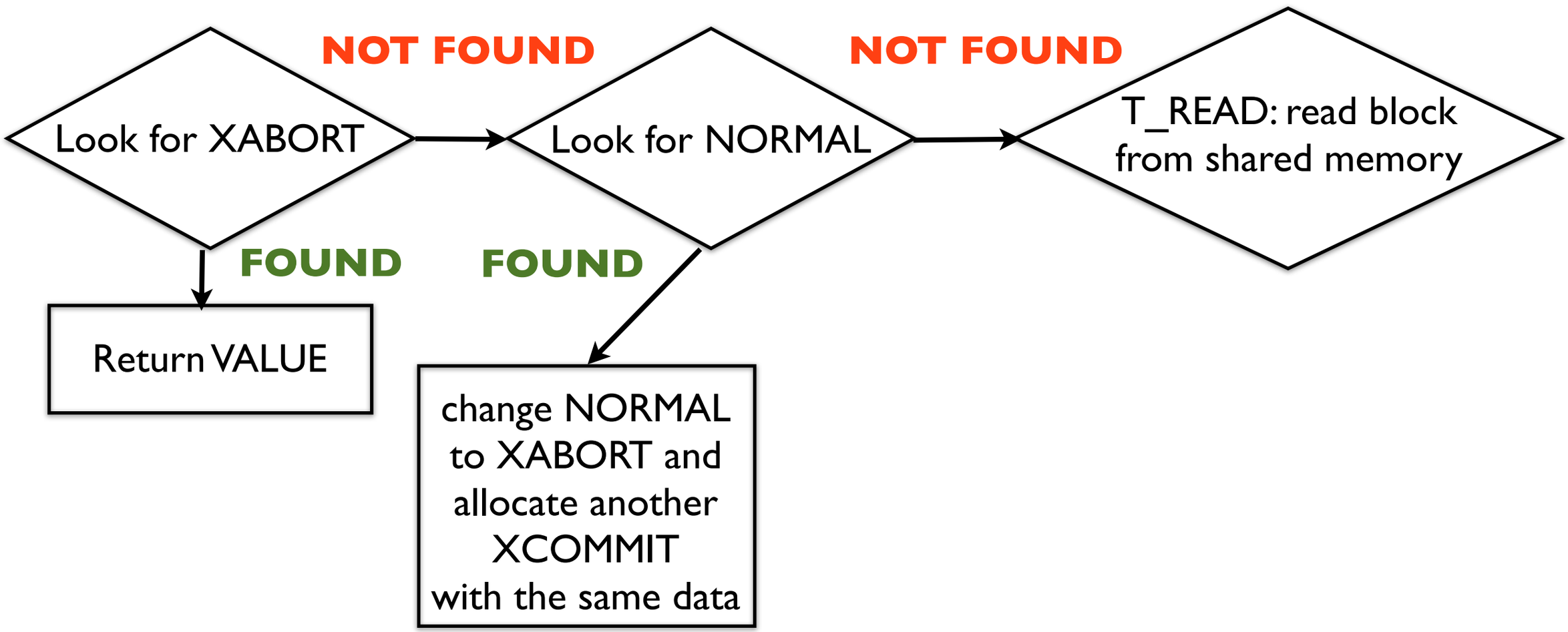
LT Instruction

Operations for an active transaction



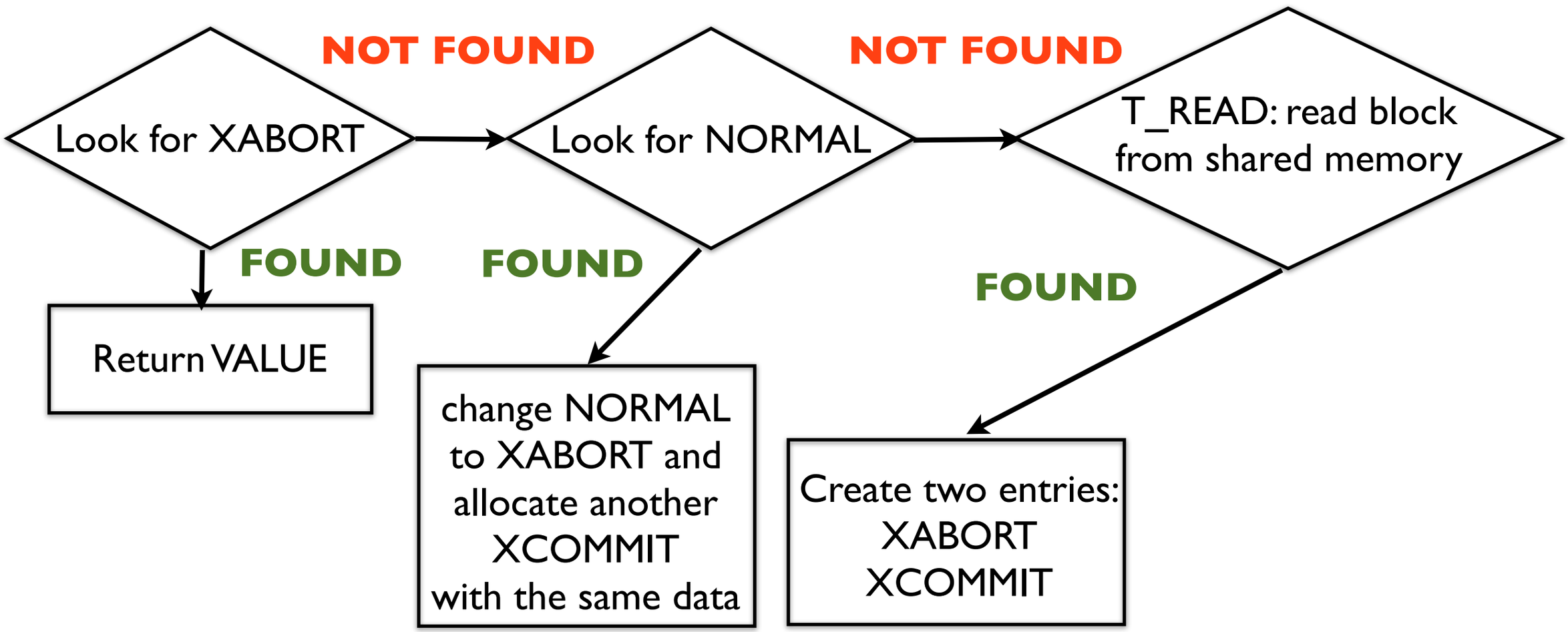
LT Instruction

Operations for an active transaction



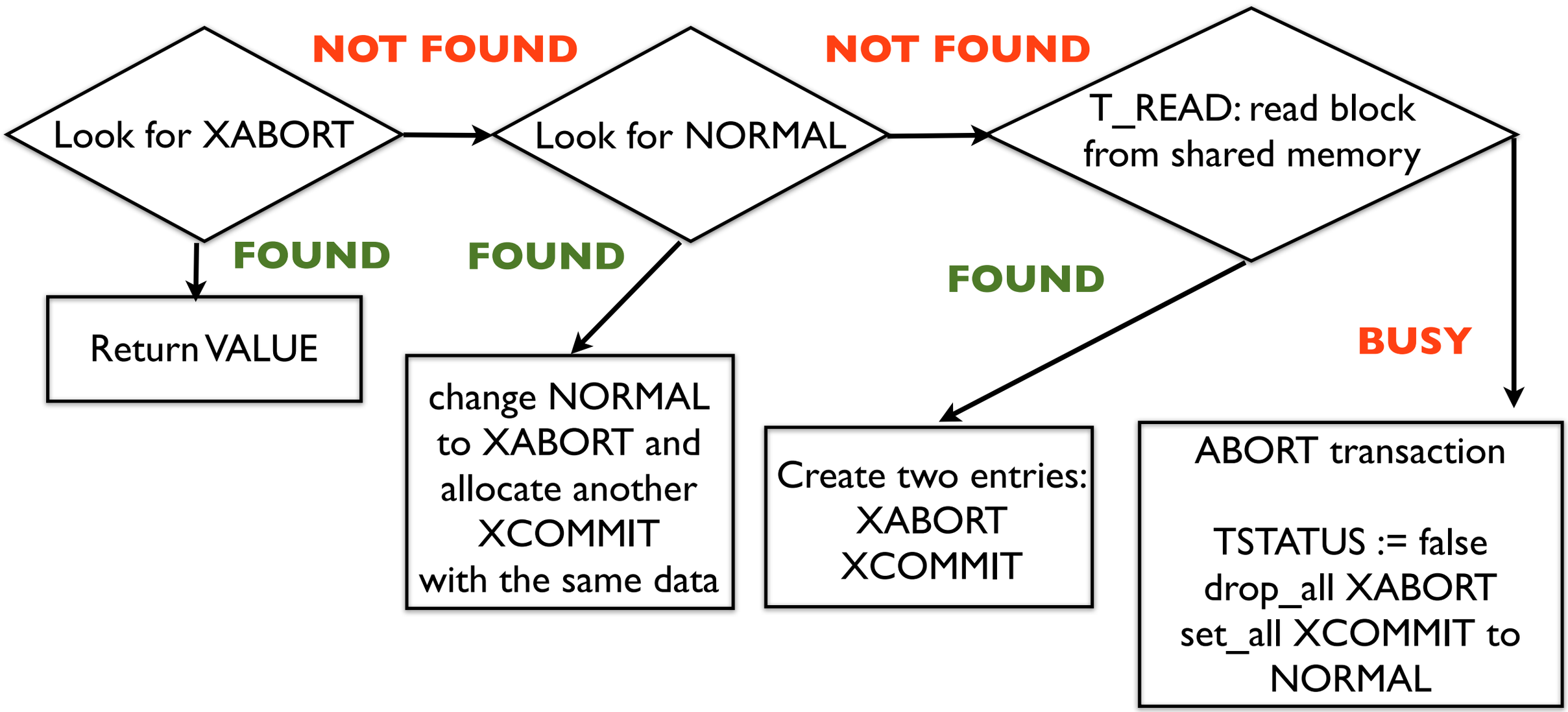
LT Instruction

Operations for an active transaction



LT Instruction

Operations for an active transaction



Simulation

- Proteus: execution driven simulator system for multiprocessor developed at MIT
- Two version of TM:
  - Goodman's snoopy protocol for bus based architecture
  - Chaicken directory protocol for a simulated Alewife machine

Simulated Architecture

- 32 processors
- Regular cache is direct mapped with 2048 lines of 8 bytes
- Transactional cache has 64 lines of 8 bytes
- Memory access requires 4 cycle

Counter Benchmark

- N processes increment shared counter 2^{16} / N times ($1 \leq N \leq 32$)
- Short critical section (only **two** shared memory accesses)

```
shared int counter;

void process(int work)
{
  int success = 0, backoff = BACKOFF_MIN;
  unsigned wait;

  while (success < work) {
    ST(&counter, LTX(&counter) + 1);
    if (COMMIT()) {
      success++;
      backoff = BACKOFF_MIN;
    }
    else {
      wait = random() % (01 << backoff);
      while (wait--);
      if (backoff < BACKOFF_MAX)
        backoff++;
    }
  }
}
```

Counter Benchmark

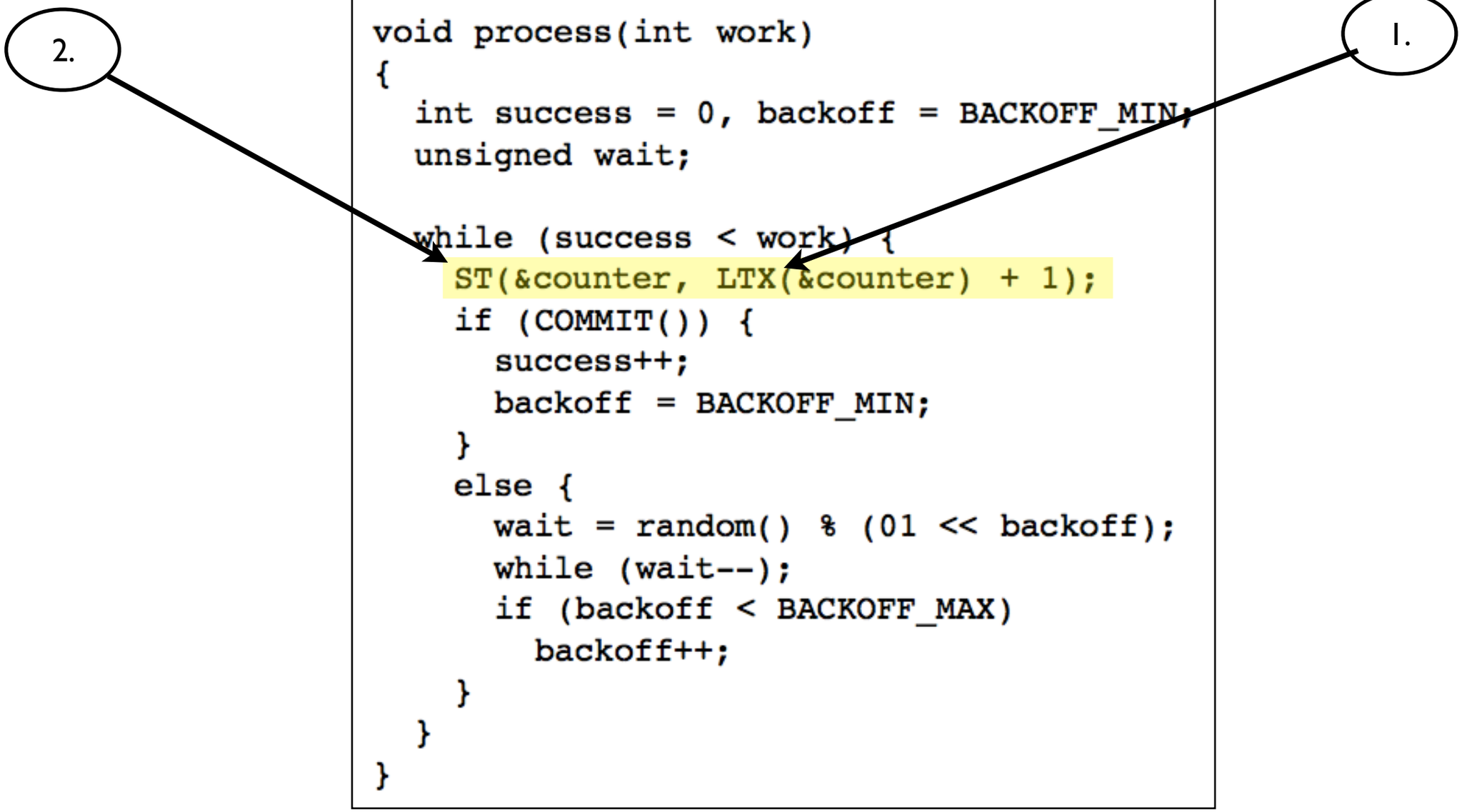
- N processes increment shared counter 2^{16} / N times ($1 \leq N \leq 32$)
- Short critical section (only **two** shared memory accesses)

Shared variable

```
shared int counter;

void process(int work)
{
  int success = 0, backoff = BACKOFF_MIN;
  unsigned wait;

  while (success < work) {
    ST(&counter, LTX(&counter) + 1);
    if (COMMIT()) {
      success++;
      backoff = BACKOFF_MIN;
    }
    else {
      wait = random() % (01 << backoff);
      while (wait--);
      if (backoff < BACKOFF_MAX)
        backoff++;
    }
  }
}
```

Counter Benchmark

- N processes increment shared counter 2^{16} / N times ($1 \leq N \leq 32$)
- Short critical section (only **two** shared memory accesses)



Counter Benchmark

- N processes increment shared counter 2^{16} / N times ($1 \leq N \leq 32$)
- Short critical section (only **two** shared memory accesses)

```
shared int counter;

void process(int work)
{
  int success = 0, backoff = BACKOFF_MIN;
  unsigned wait;

  while (success < work) {
    ST(&counter, LTX(&counter) + 1);
    if (COMMIT()) {
      success++;
      backoff = BACKOFF_MIN;
    }
    else {
      wait = random() % (01 << backoff);
      while (wait--);
      if (backoff < BACKOFF_MAX)
        backoff++;
    }
  }
}
```

Critical Section

Reads the value "hinting" that the variable is likely to be update.
Then store the new one

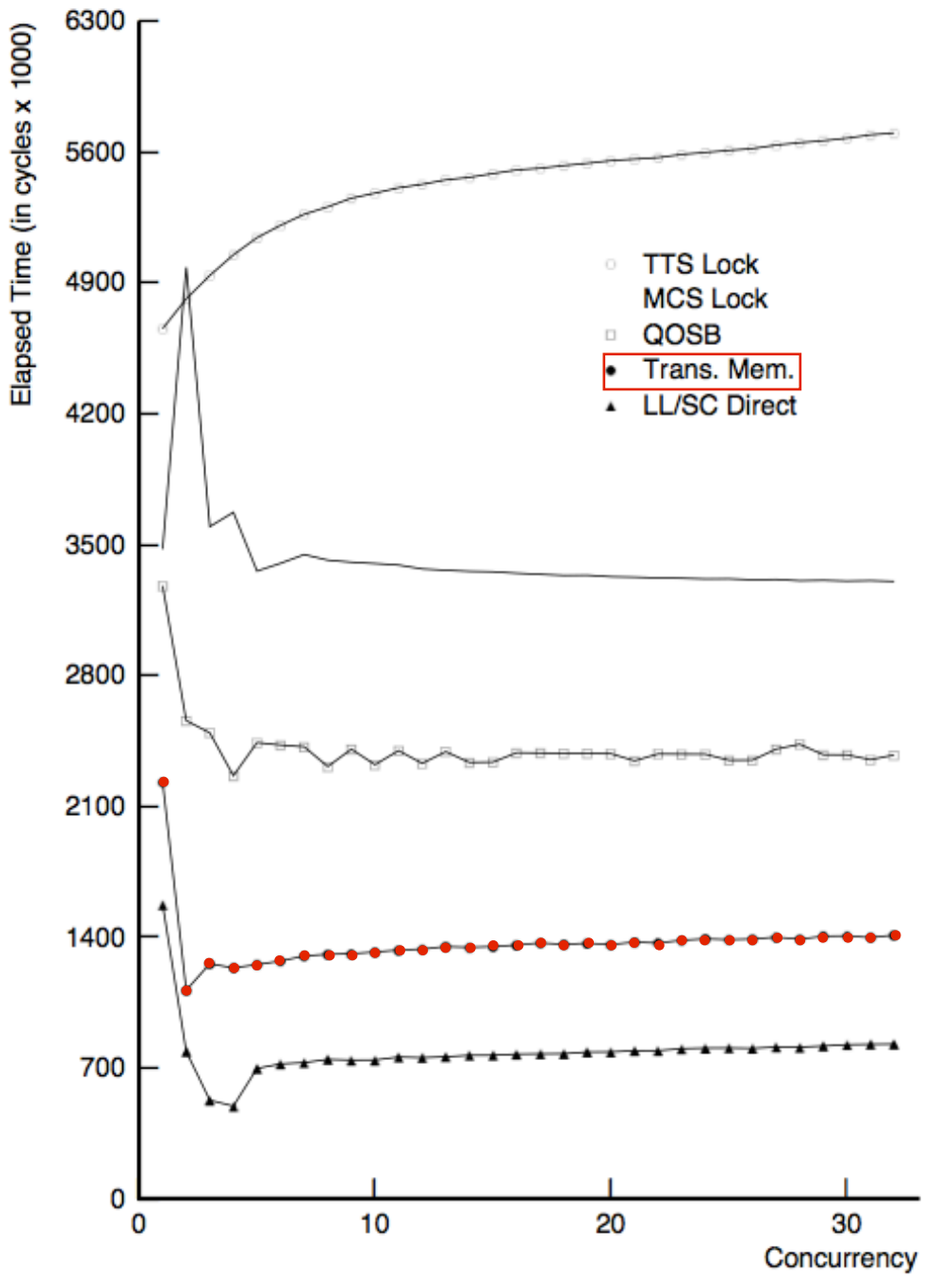
Counter Benchmark

- N processes increment shared counter 2^{16} / N times ($1 \leq N \leq 32$)
- Short critical section (only **two** shared memory accesses)

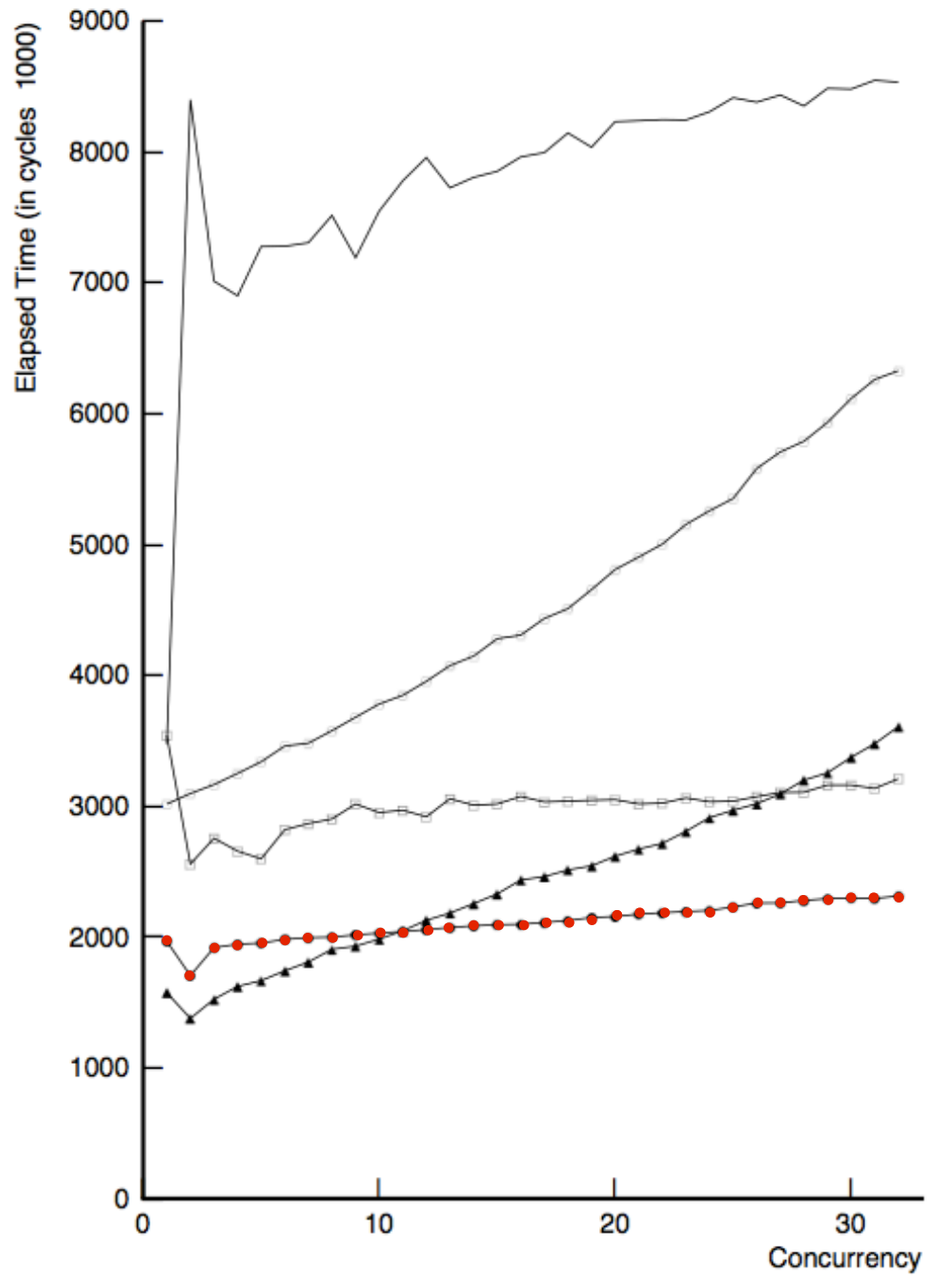
```
shared int counter;

void process(int work)
{
  int success = 0, backoff = BACKOFF_MIN;
  unsigned wait;

  while (success < work) {
    ST(&counter, LTX(&counter) + 1);
    if (COMMIT()) {
      success++;
      backoff = BACKOFF_MIN;
    }
    else {
      wait = random() % (01 << backoff);
      while (wait--);
      if (backoff < BACKOFF_MAX)
        backoff++;
    }
  }
}
```

If counter has been committed successfully, then we can proceed otherwise we wait

Counting Result



Bus-Based



Directory-Based

In Conclusion TM...

- overcome single-word limitation
- is a multi-processor architecture which allows easy lock-free multi-word synchronization in HW
- matches or outperforms atomic update locking techniques for simple benchmarks

Pros

- Easy programming semantics
- More parallelism and hence highly scalable for transactions of small size

Pros

- Easy programming semantics
- More parallelism and hence highly scalable for transactions of small size

Cons

- Small transaction size limit usage for applications locking large object
- Limited only to primary cache

Questions?