Transactional Memory: Architectural Support for Lock-Free Data Structures (1993)

Maurice Herlihy J. Eliot B. Moss

Presented by Silvan Mosberger

Background: Transactional Memory

- Concurrency is hard! Locking is error-prone, transactional memory is easy
- Allows multiple operations, a transaction, to be executed atomically
- Can include loads/stores to arbitrary memory locations
- Transactions are isolated, all its changes are only visible once it commits
- When something went wrong, abort it and retry

Problem

- Problems with conventional locking techniques in highly concurrent systems
  - Priority Inversion
  - Lock convoy
  - Deadlock
- Software transactional memory is nice but slow

Goal

- Specify implementation for hardware transactional memory
- Make it fast in highly concurrent systems
- Consequently, committing/aborting transactions should be processor-local

Key Approach and Idea

Idea

- Snoopy cache coherency protocol can also detect conflicting transactions
- Abort a transaction upon conflict

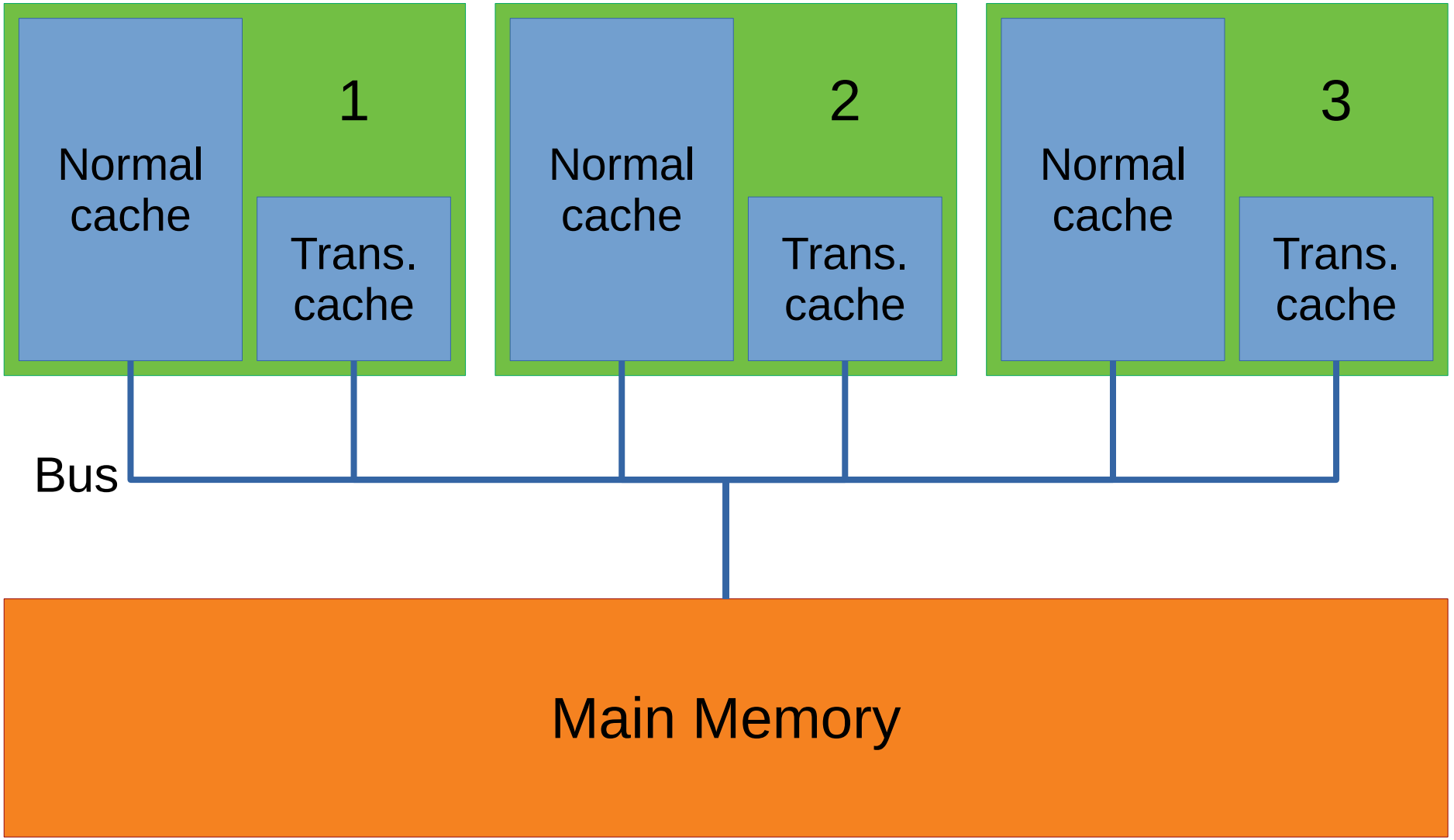
Key Approach

- Additional smaller transactional cache for memory locations participating in the transaction
- Use two cache entries, one in case of abort, one in case of commit
- Extend snoopy protocol for transactions

Mechanisms: Programmer Interface

- LT: Load-transactional, read a memory location
- LTX: Load-transactional-exclusive, read a memory location “hinting” it will be updated
- ST: Store-transactional, write a memory location
- COMMIT: attempt to commit the changes
- ABORT: discard all changes
- VALIDATE: Test for already aborted, guarantees consistency of previously read values

Mechanisms: Cache structure



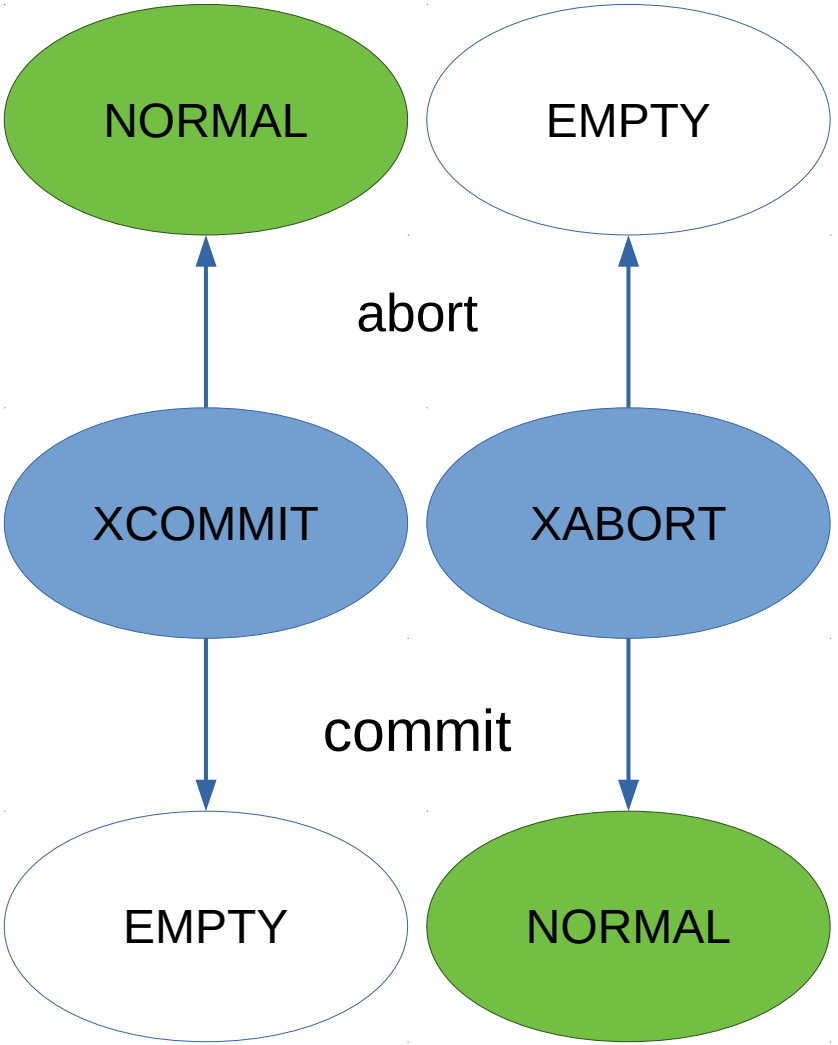
Mechanisms: Transactional cache

Name	Access	Shared?	Modified?
INVALID	none	—	—
VALID	R	Yes	No
DIRTY	R, W	No	Yes
RESERVED	R, W	No	No

Table 1: Cache line states

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

Table 2: Transactional tags



Mechanisms: Bus cycles

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

Standard bus cycles

- WRITE: Write back to main memory
- READ: Read for shared access
- RFO: Read for exclusive access

New transactional cycles

- T_READ: Same as READ but for transactional cache
- T_RFO: Same as RFO but for transactional cache
- BUSY: Used for refusing cache requests

Mechanisms: Processor Actions

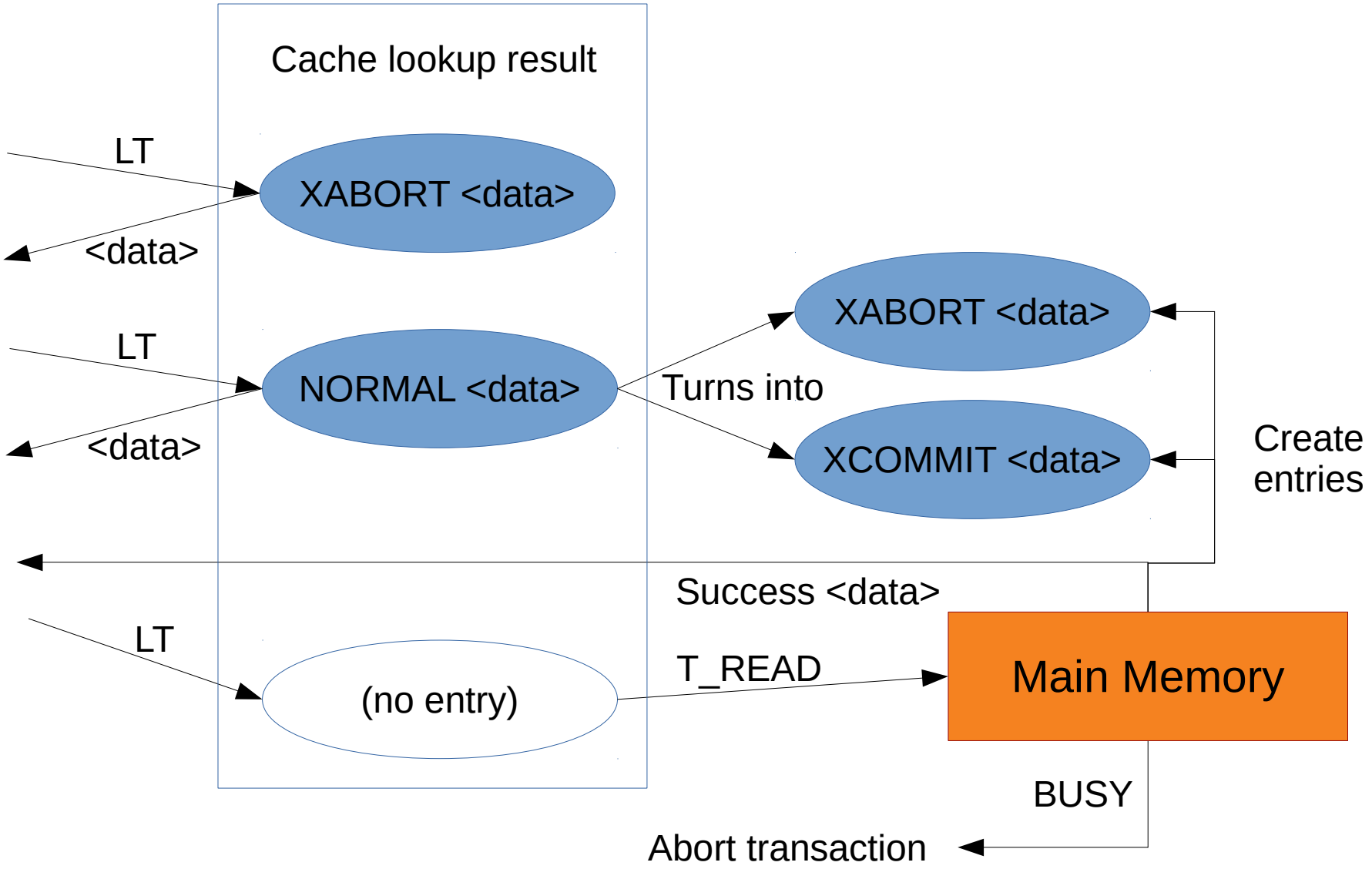
```
1 // Whether a transaction is
2 // in progress
3 bool TACTIVE;
4 // Whether the transactios is
5 // still active or aborted
6 bool TSTATUS;
```

```
8  void abort(bool internal) {
9    if (internal) {
10     TSTATUS = false;
11   } else {
12     TACTIVE = false;
13     TSTATUS = true;
14   }
15
16   set_all(XCOMMIT, NORMAL);
17   set_all(XABORT, EMPTY);
18 }
```

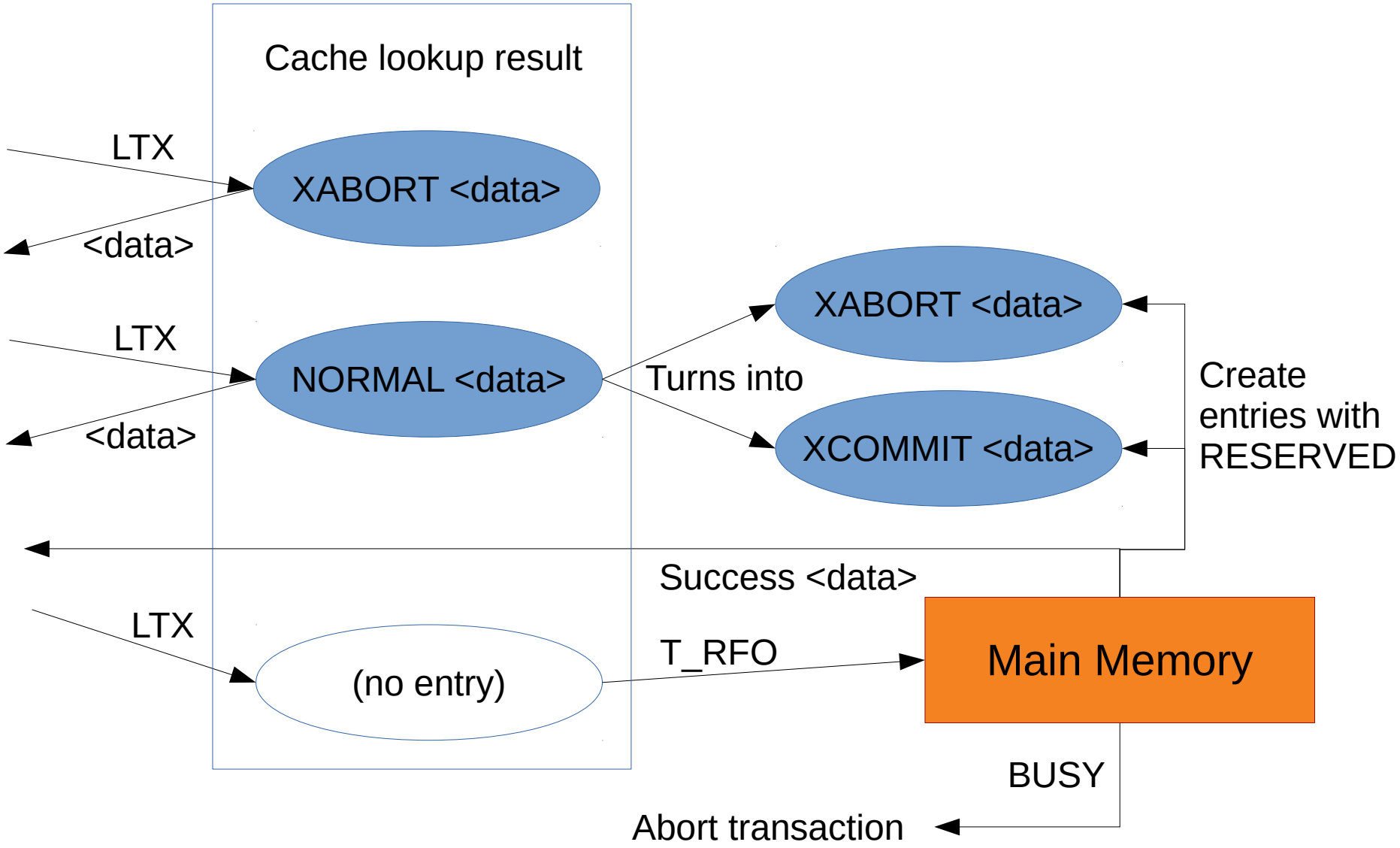
```
20 bool commit() {
21   bool oldStatus = TSTATUS;
22
23   TACTIVE = false;
24   TSTATUS = true;
25
26   set_all(XCOMMIT, EMPTY);
27   set_all(XABORT, NORMAL);
28
29   return oldStatus;
30 }
```

```
32 bool validate() {
33   bool oldStatus = TSTATUS;
34   if (!TSTATUS) {
35     TACTIVE = false;
36     TSTATUS = true;
37   }
38   return oldStatus;
39 }
```

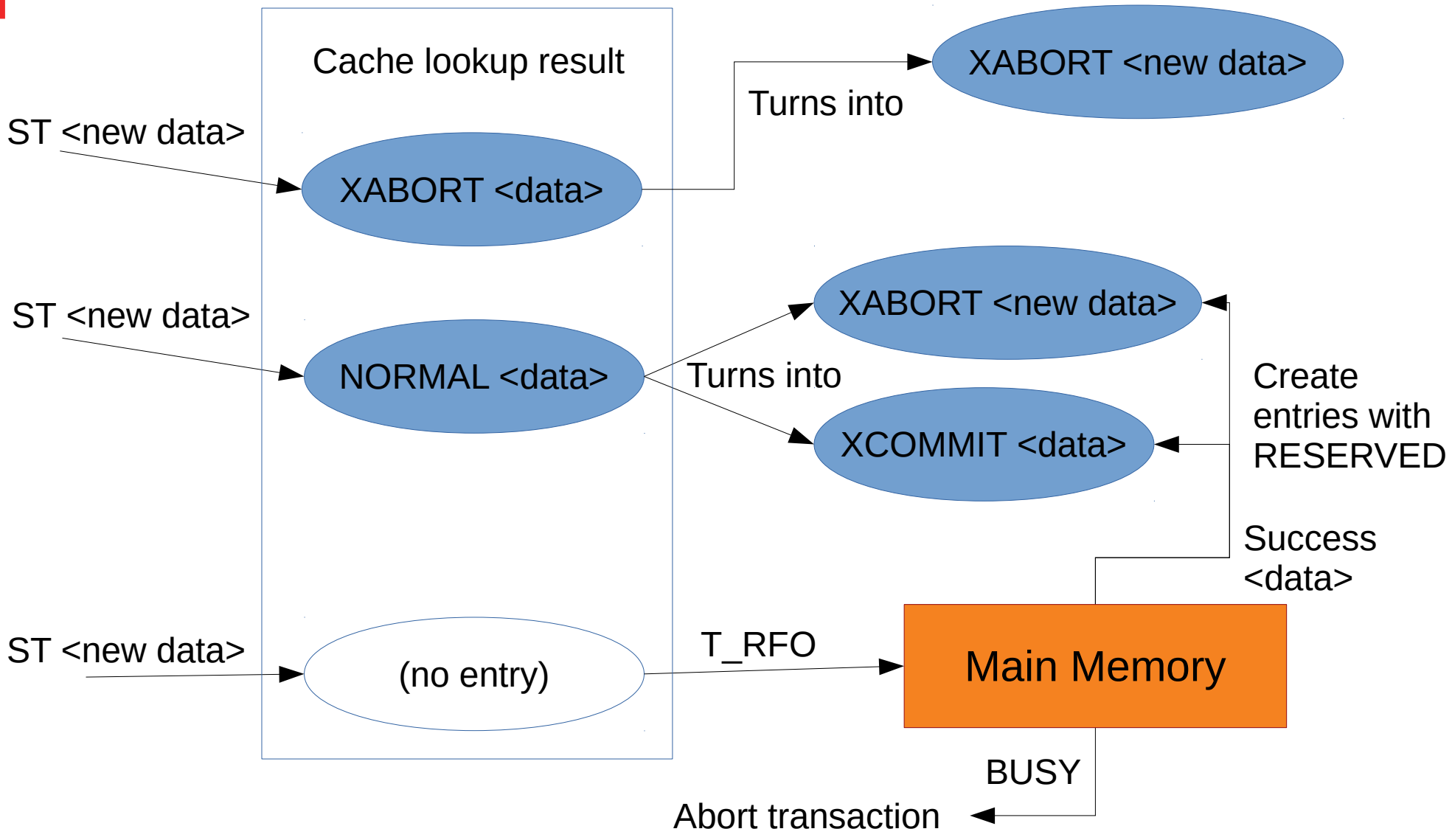
Mechanisms: Processor Action: LT



Mechanisms: Processor Action: LTX



Mechanisms: Processor Action: ST



Key Results: Methodology and Evaluation

Architectures

- **Bus:** Snoopy cache coherence for bus-based architecture
- **Network:** Chaiken directory protocol for network-based machine, discussed in technical report

Benchmarks:

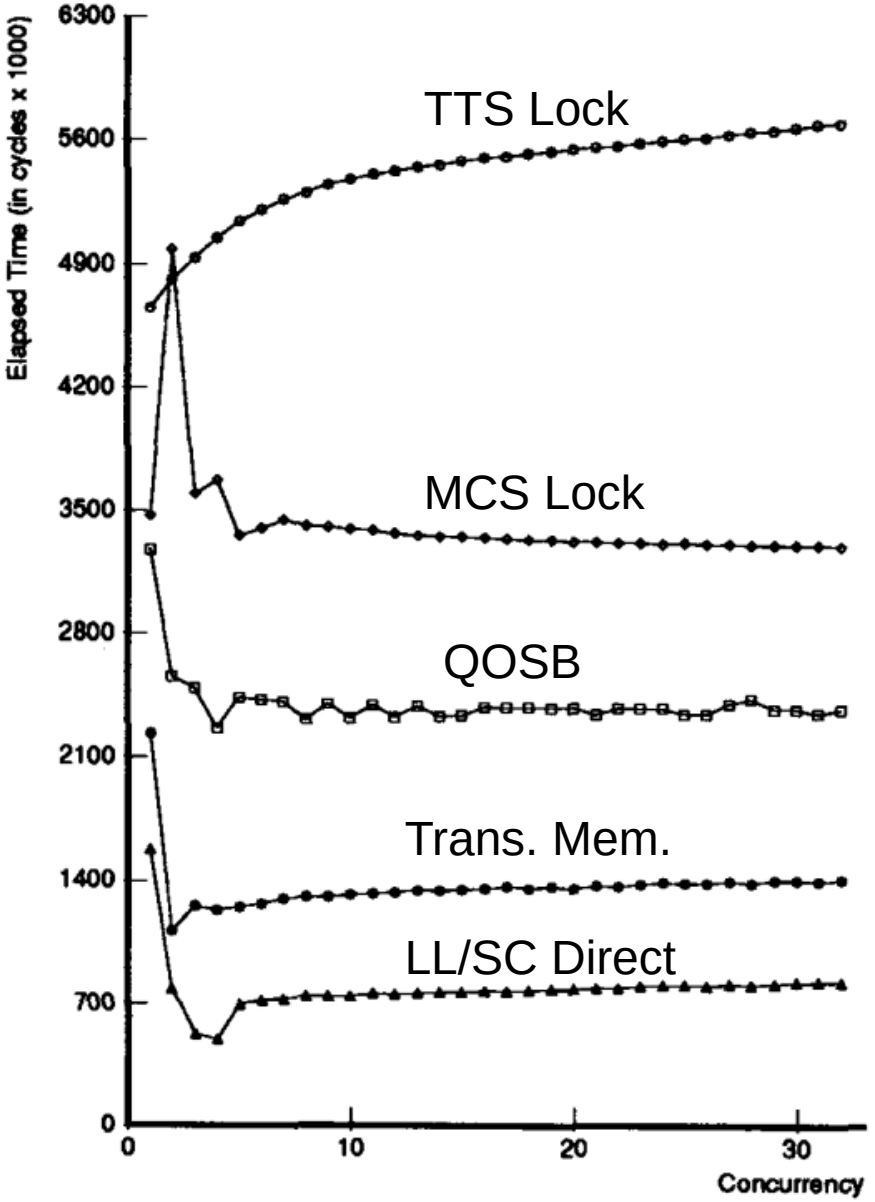
- **Counting:** Increment shared counter. short critical sections $\rightarrow$ contention high
- **Producer/Consumer:** Shared bounded FIFO buffer, half of the processors producers, half consumers
- **Doubly-Linked List:** Shared linked list, every process dequeues from tail, enqueues back to head. No easy concurrency for locks

Key Results: Methodology and Evaluation

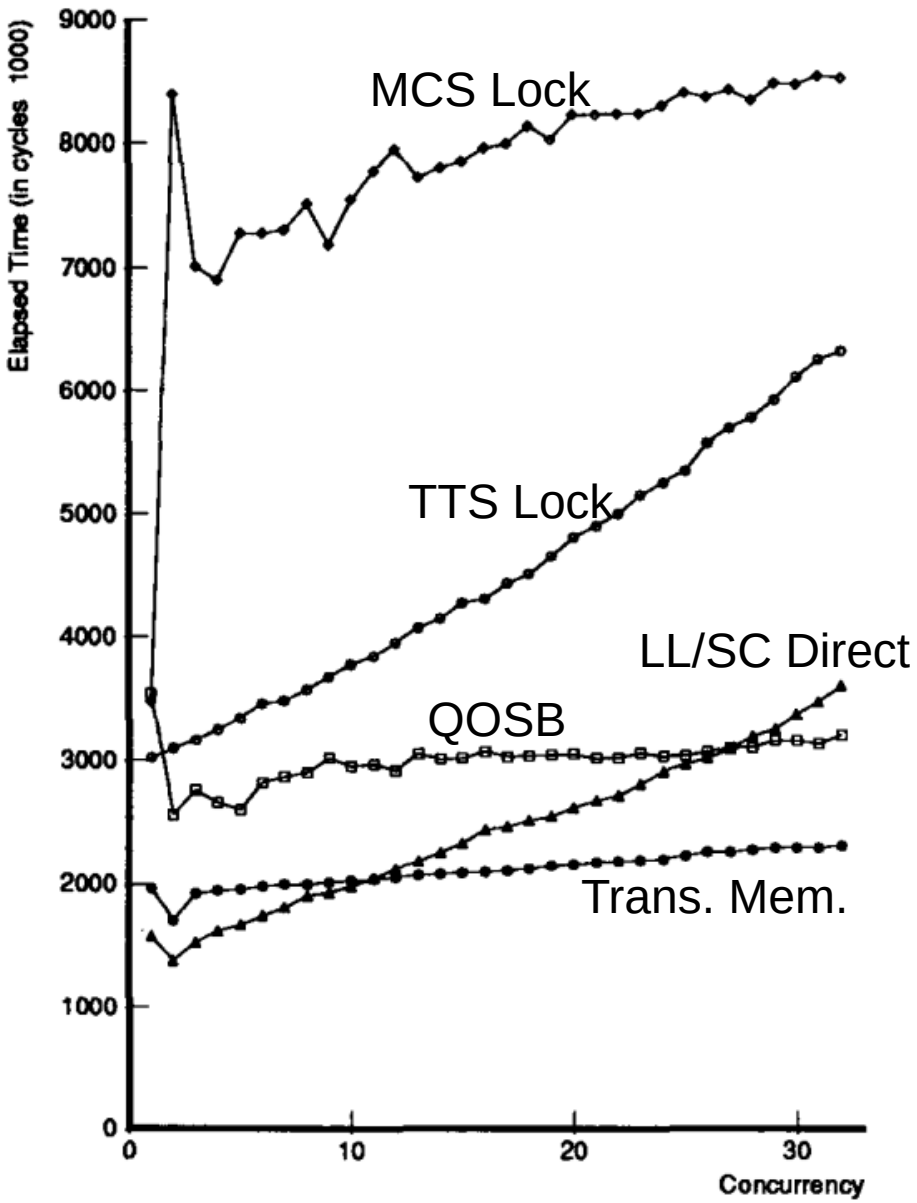
Other techniques for comparison:

- **TTS (test-and-test-and-set) Lock:** Read cached value until evicted, then do test-and-set in memory directly
- **LL/SC (load-linked/store-cond):** LL copies value to local variable, SC tries to change its value and succeeds if no other process has modified it
- **MCS Lock (software queueing):** Placed on queue if unable to acquire lock, eliminating lock polls
- **QOSB (hardware queueing):** Queue incorporated into cache coherence protocol via unused cache lines

Benchmark: Counting

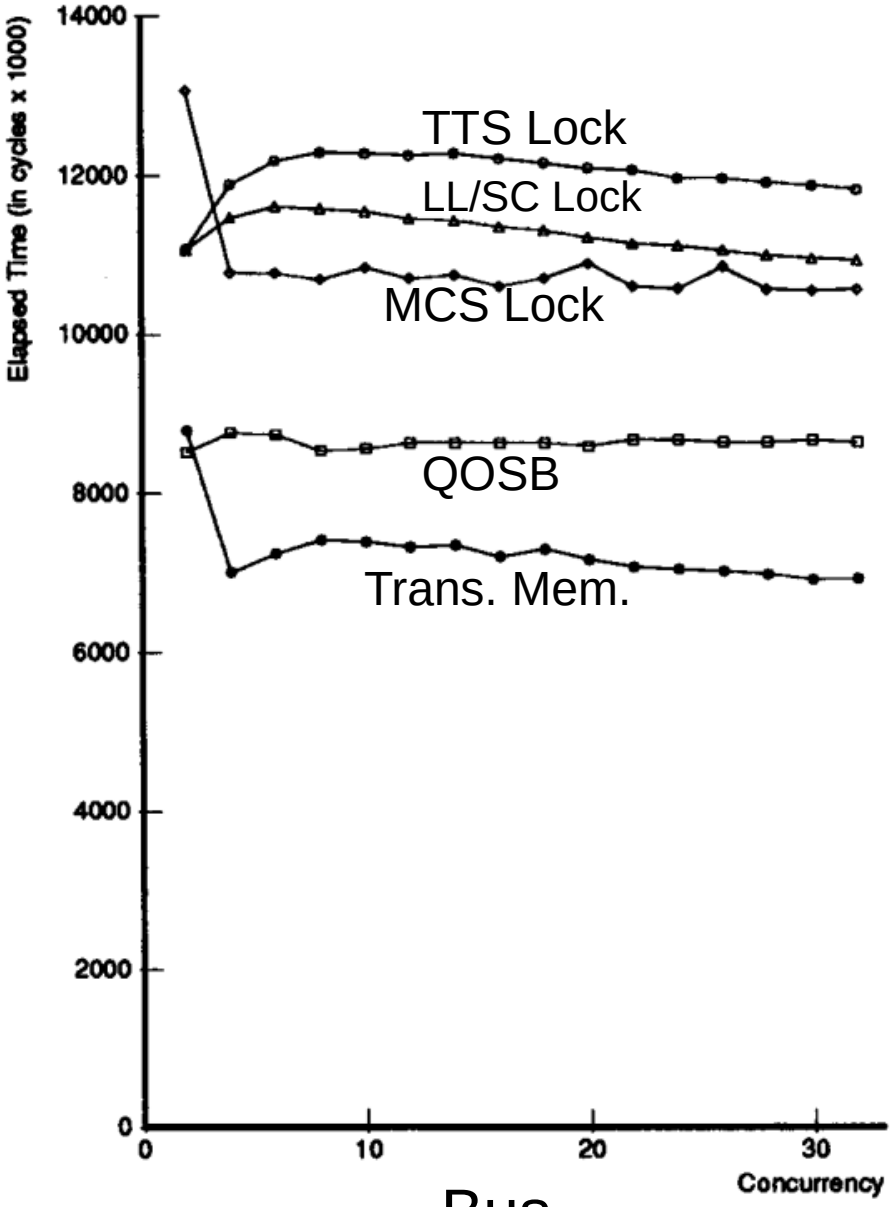


Bus

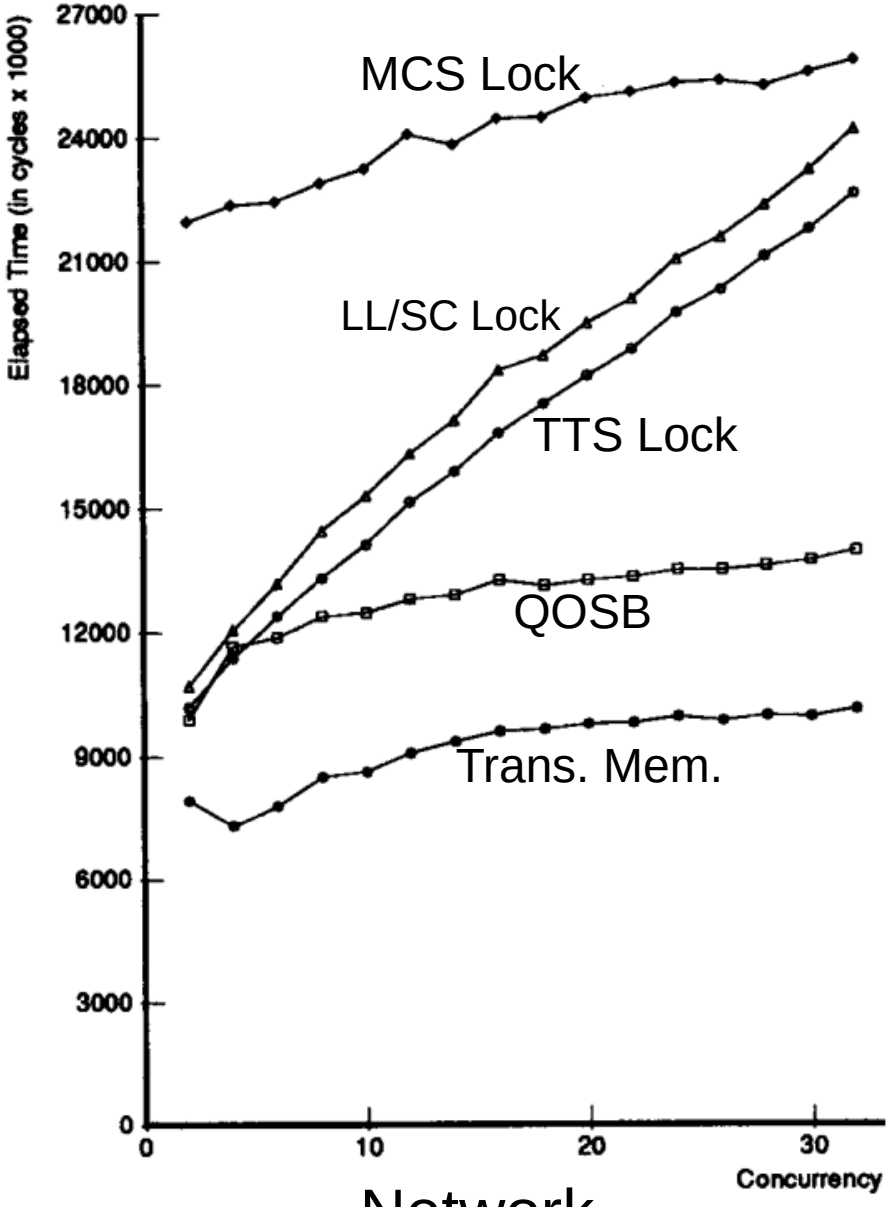


Network

Benchmark: Producer/Consumer

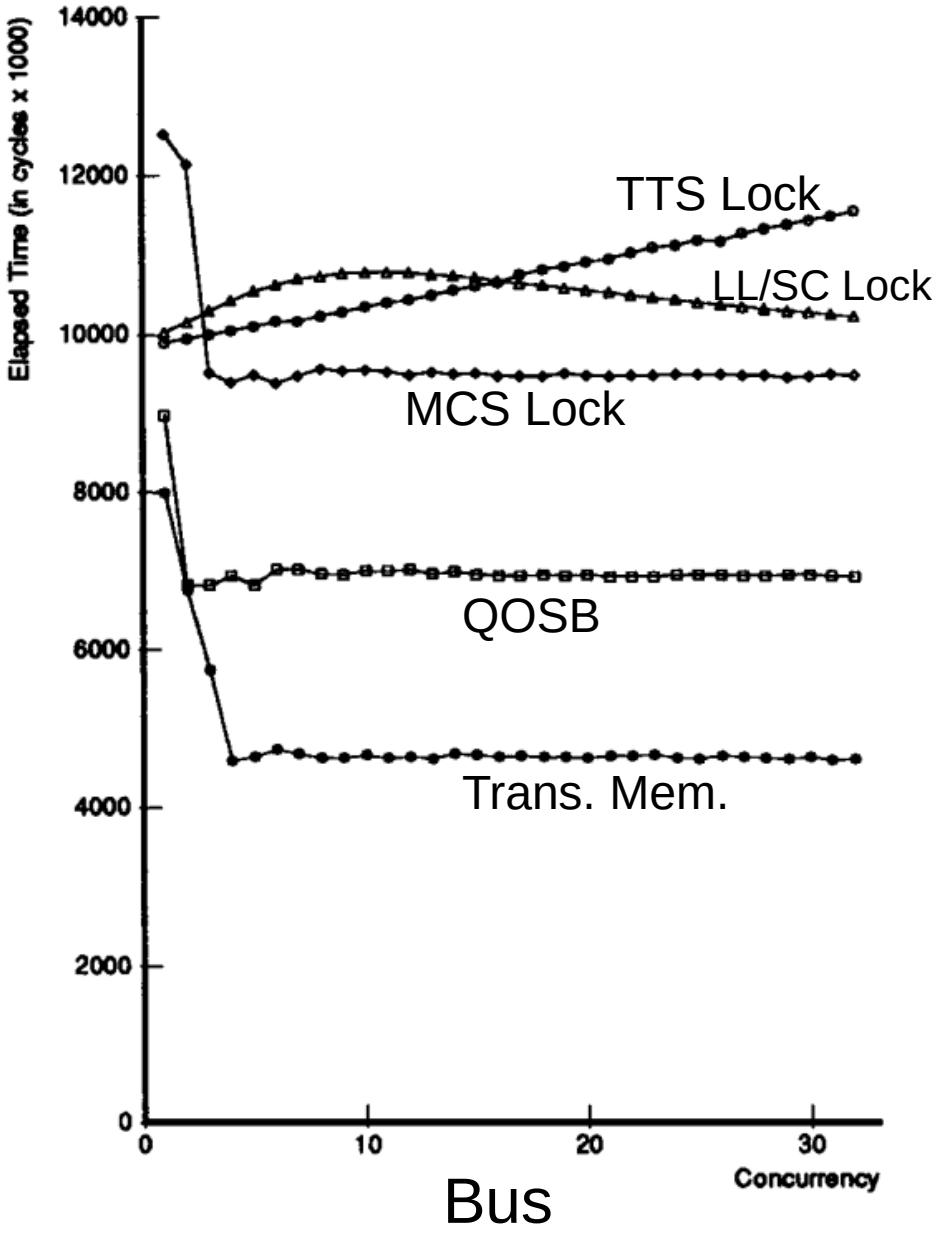


Bus

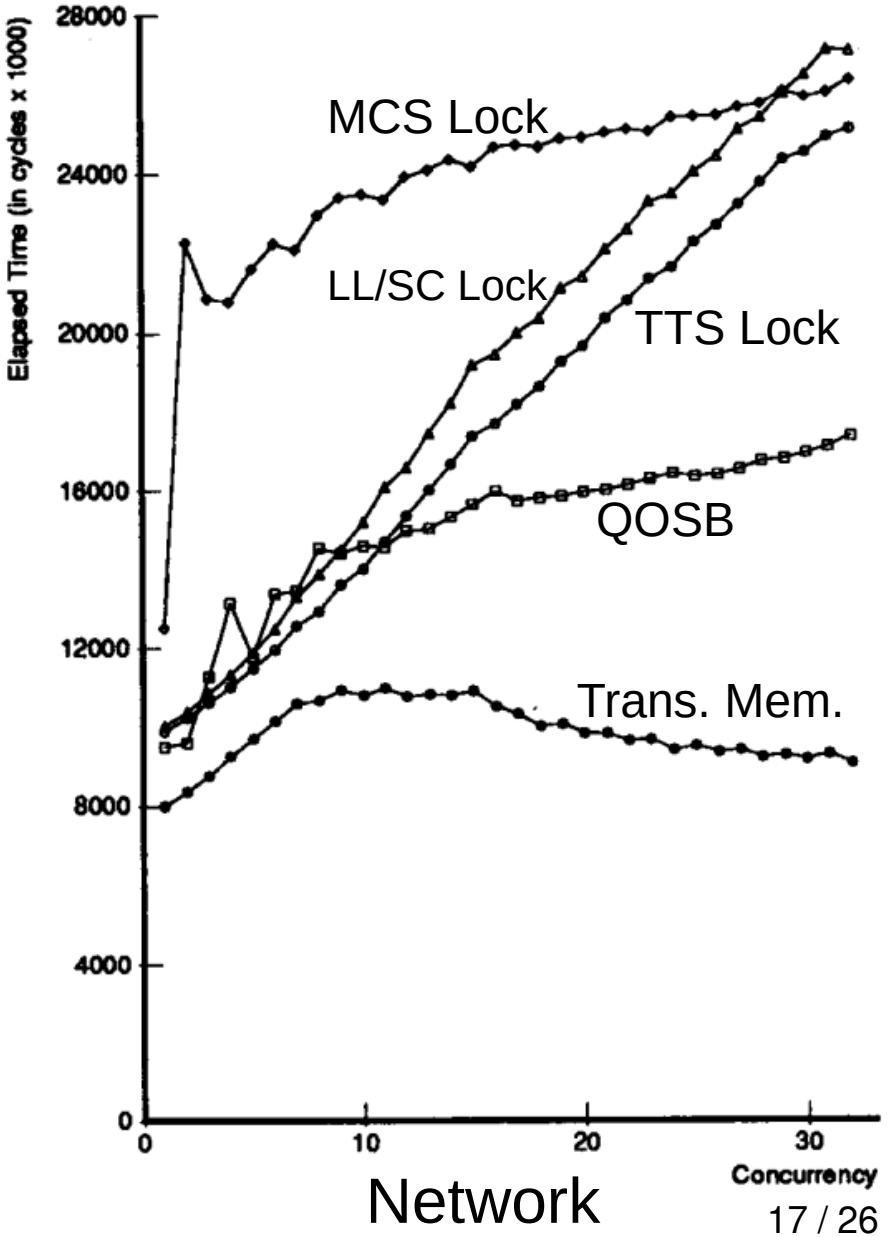


Network

Benchmark: Doubly-Linked List



Bus



Network

Summary

- **Problem:** Locks are fast but hard to use, software transactional memory is easy to use but slow
- **Goal:** Implement fast hardware-based transactional memory
- **Idea:** Use separate transactional cache for storing two entries for every memory location, one in case of commit, one in case of abort. Use cache coherency protocol to detect conflicting transactions.
- **Results:** Hardware transactional memory outperforms other techniques especially in highly concurrent systems.

Strengths

- Explains the limits of this approach and how to work around it
  - Starvation → exponential backoff
  - Too few cache lines → emulate in software
- First paper to fully explore hardware transactional memory
- No need to write back to memory on commit, happens over time when cache lines get replaced
- Extra technical paper explains everything in much more detail

Weaknesses

- No explanation as to why the mentioned protocol is correct or how it came to be
- No diagrams to visualize the protocol, not easy to follow with just text
- XABORT means “Discard on abort”, but it becomes valid on commit which is more understandable. Same with XCOMMIT. Naming is hard!
- LTX and XCOMMIT only there to make it faster, but no benchmarks for determining the difference they make and in which cases
- Doesn’t explain well how transactional and non-transactional memory locations interact

Takeaways

- Consider using transactional memory for your concurrency needs
- Ideas sometimes have applications you haven’t thought of initially
- Consider tradeoffs, it might be desirable to have more complexity for more performance

Questions and Open Discussion

Extra Questions

- Has anybody used transactional memory before?
- How could software transactional memory be implemented?
  - Write to shared memory
  - Log all read and writes
  - On commit, ensure all reads haven't changed
  - Abort and roll back changes if not
- What problem is there with not writing immediately back to main memory?
  - Values might not get written back for a while $\rightarrow$ more chance of losing updates on power loss
  - Polutes cache, creating new entries can require writeback to main memory

Related Papers

- First paper to suggest hardware transactional memory: Knight, T. An architecture for mostly functional languages. Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP ’86 doi:10.1145/319838.319854
- Associated technical report with implementation details: M.P. Herlihy, J. Eliot B. Moss. Transactional memory: Architectual support for lock-free data structures. Technical Report CRL-92-7, Digital Cambridge Research Lab, One Kendall Square, Cambridge MA 02139, December 1992.
- Paper introducing software transactional memory to Haskell: Tim Harris, Simon Marlow, Simon Peyton Jones. Composable Memory Transactions. PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, January 2005. doi:10.1145/1065944.1065952

Extra: Usage example

```
shared long[] accounts = ...;

// Try to transfer `amount` of money from account `from` to `to`
int transfer (int from, int to, long amount) {
  float frombalance = LTX(&accounts[from]);
  if (amount >= frombalance)
    ABORT();

  float tobalance = LTX(&accounts[to]);
  ST(&accounts[from], frombalance - amount);
  ST(&accounts[to], tobalance + amount);
  return COMMIT();
}
```

Extra: Snoopy actions

- Both caches snoop on the bus
- A cache ignores any cycles for lines not in that cache
- The regular cache
  - On READ/T_READ, if state is VALID, return value
  - If RESERVED or DIRTY, return value and reset to VALID
  - On RFO/T_RFO, return data and invalidate line
- The transactional cache
  - If TSTATUS is false, or if READ/RFO, behave just like normal cache, except it ignores entries with transactional tag != NORMAL
  - On T_READ and state VALID, return value
  - Otherwise return BUSY
- Either cache can do WRITE when line needs to be replaced
- Memory only responds to READ, T_READ, RFO and T_RFO that no cache responds to, and all WRITE requests