CS510 Concurrent Systems

Jonathan Walpole

Portland State
UNIVERSITY

Transactional Memory: Architectural Support for Lock-Free Data Structures

By Maurice Herlihy and J. Eliot B. Moss

1993

Outline

Review of synchronization approaches

Transactional memory (TM) concept

A hardware-based TM implementation

- Core additions
- ISA additions
- Transactional cache
- Cache coherence protocol changes

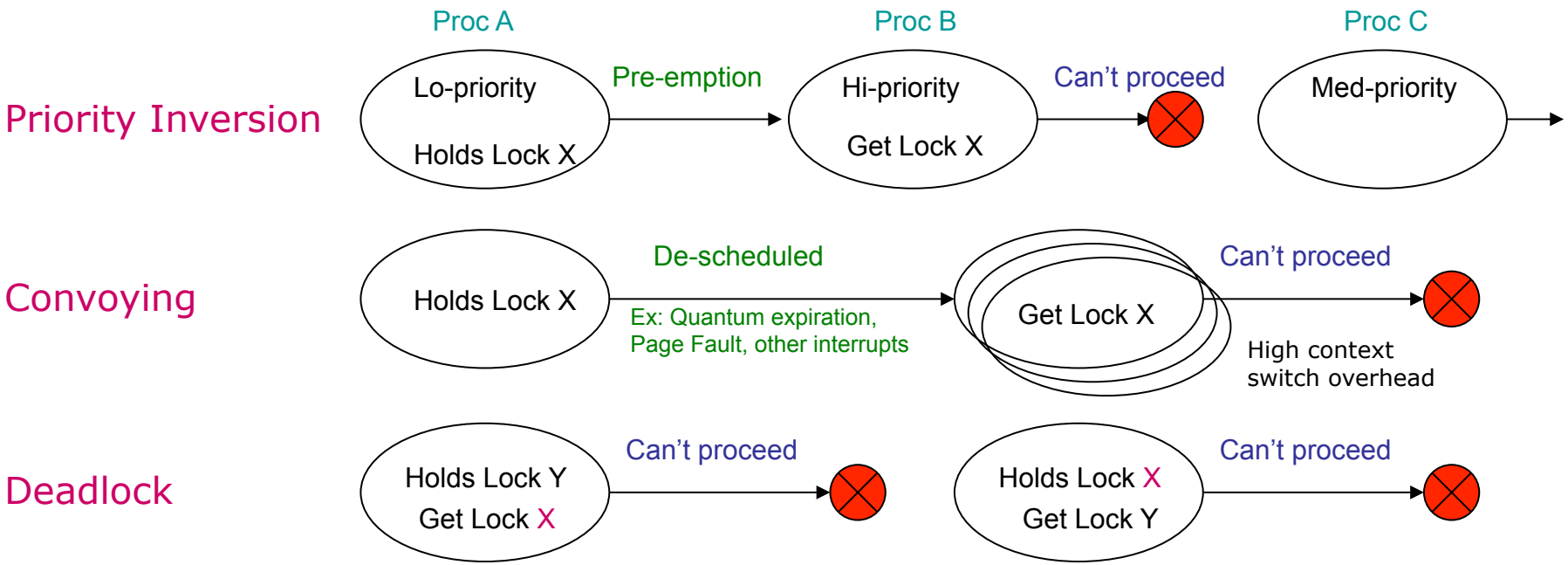
Validation and results

Conclusions

Locking

A pessimistic approach based on mutual exclusion and blocking

- Easy to understand and use
- Does not scale well
- Not composable



Lock-Free Synchronization

Non-Blocking, optimistic approach
- Uses atomic operations such as CAS, LL&SC
- Limited to operations on single-word or double-words

Avoids common problems seen with locking
- no priority inversion, no convoying, no deadlock

Difficult programming logic makes it hard to use

Wish List

Simple programming logic (easy to use)

No priority inversion, convoying or deadlock

Equivalent or better performance than locking

- Less data copying

No restrictions on data set size or contiguity

Composable

Wait-free

. . .

. . . Enter Transactional Memory (TM) …

What is a Transaction?

Transaction: a finite sequence of revocable operations, executed by a process, that satisfies two properties:

1. Serializability
   - the steps of one transaction are not seen to be interleaved with the steps of another transaction
   - all processes agree on the order of transaction execution
2. Atomicity
   - Each transaction either aborts or commits
   - Abort: all tentative changes of the transaction are discarded
   - Commit: all tentative changes of the transaction take effect immediately

This paper assumes that a process executes one transaction at a time

- Transactions do not nest (ie. not composable)
- Transactions do not overlap

What is Transactional Memory?

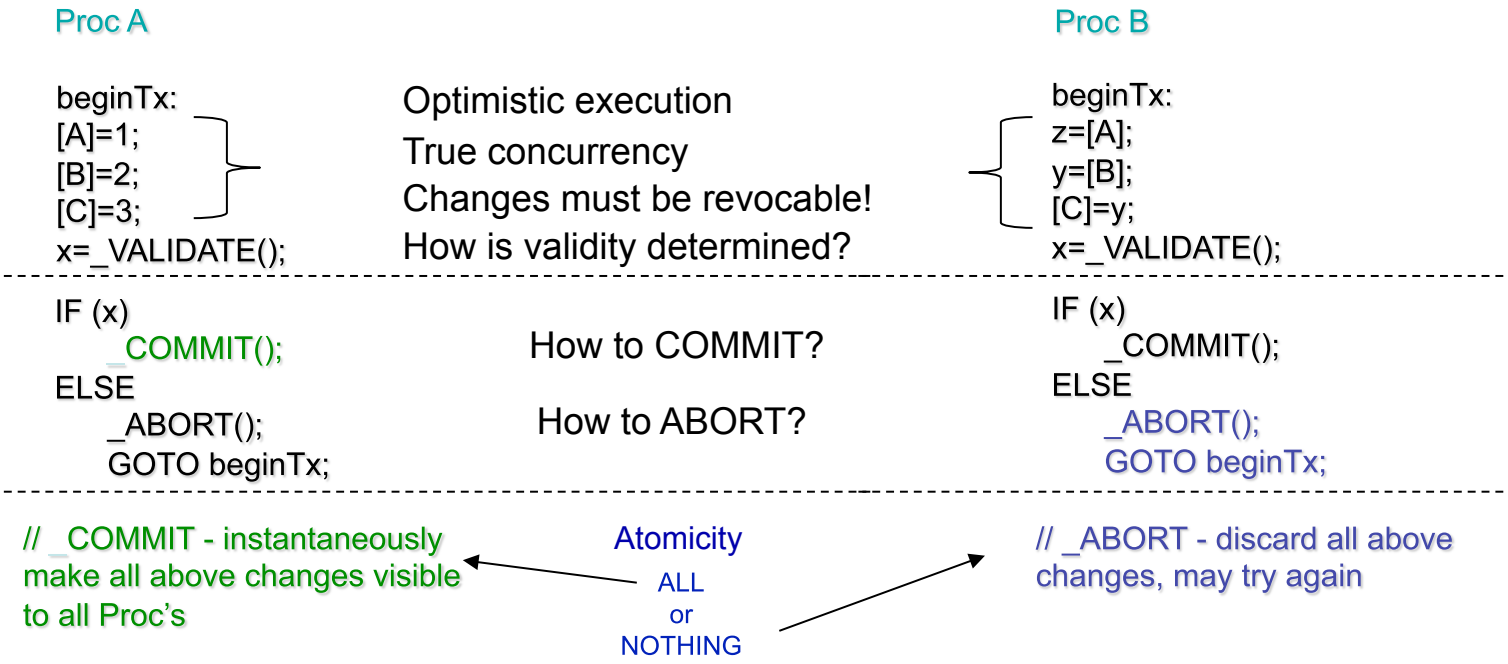
Transactional Memory (TM) is a lock-free, non-blocking concurrency control mechanism based on transactions

TM allows programmers to define customized atomic operations that apply to multiple, independently chosen memory locations

TM is non-blocking

- Multiple transactions execute optimistically, in parallel, on different CPU's
- If a conflict occurs only one can succeed, others can retry

Basic Transaction Concept

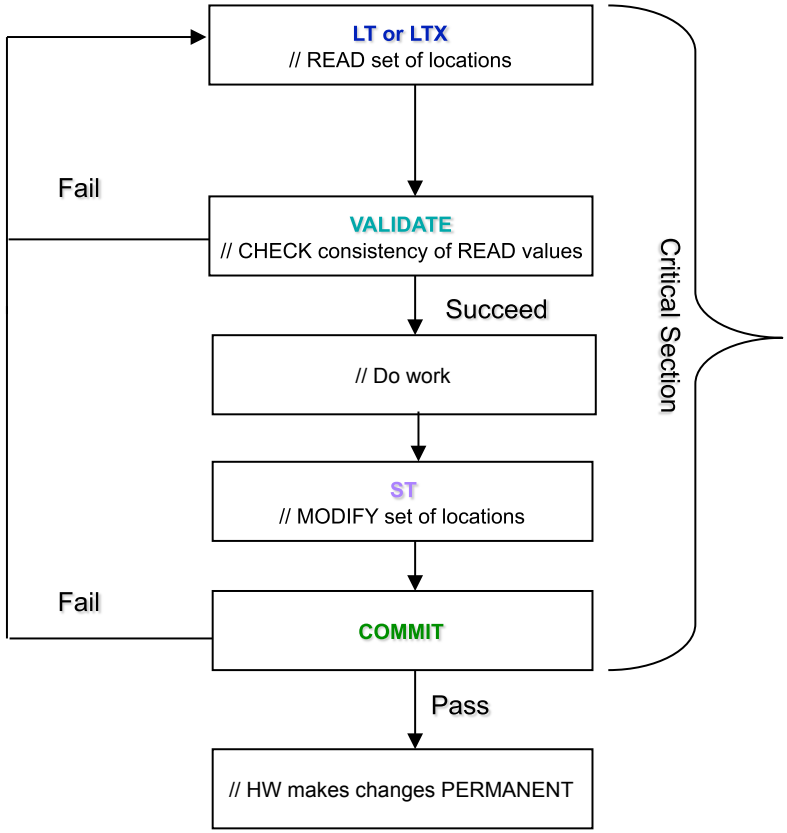


Serialization is ensured if only one transaction commits and others abort
Linearization is ensured by atomicity of validate, commit, and abort operations

TM Primitives

LT	- Load Transactional
LTX	- Load Transactional Exclusive
ST	- Store Transactional
Validate	
Commit	
Abort	

TM vs. Non-Blocking Algorithm




```
// TM
While (1) {

  curCnt = LTX(&cnt);


  if (Validate()) {

    int c = curCnt+1;


    ST(&cnt, c);

    if (Commit())
        return;
  }


}
```

```
// Non-Block
While (1) {

  curCnt = LL(&cnt);

  // Copy object

  if (Validate(&cnt)) {

    int c = curCnt + 1;


    if (SC(&cnt, c))
        return;
  }


}
```

Hardware or Software TM?

TM may be implemented in hardware or software

- Many SW TM library packages exist (C++, C#, Java, Haskell, Python, etc.)
- 2-3 orders of magnitude slower than other synchronization approaches

This paper focuses on a hardware implementation of TM

- better performance than SW for reasonable cost
- *Minor* tweaks to CPUs
  - core, instruction set, caches, bus and cache coherency protocols ...
- Uses cache-coherency protocol to maintain transactional memory consistency
- Hardware implementation without software cache overflow handling is problematical

Core Changes

Each CPU maintains two additional Status Register bits

TACTIVE

- flag set if a transaction is in progress on this CPU

TSTATUS

- flag set if the active transaction has conflicted with another transaction

TACTIVE	TSTATUS	MEANING
False	DC	No tx active
True	False	Orphan tx - executing, conflict detected, will abort
True	True	Active tx - executing, no conflict yet detected

Instruction Set Changes

LT: load from shared memory to register

ST: tentative write of register to shared memory
becomes visible upon successful COMMIT

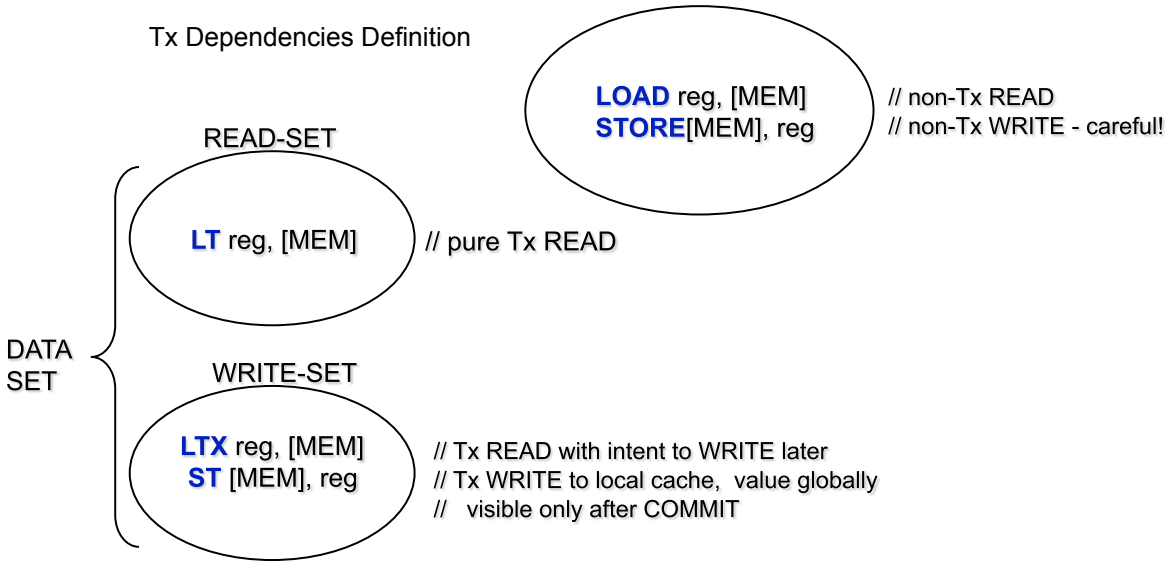
LTX: LT + intent to write to same location later
A performance optimization for early conflict detection

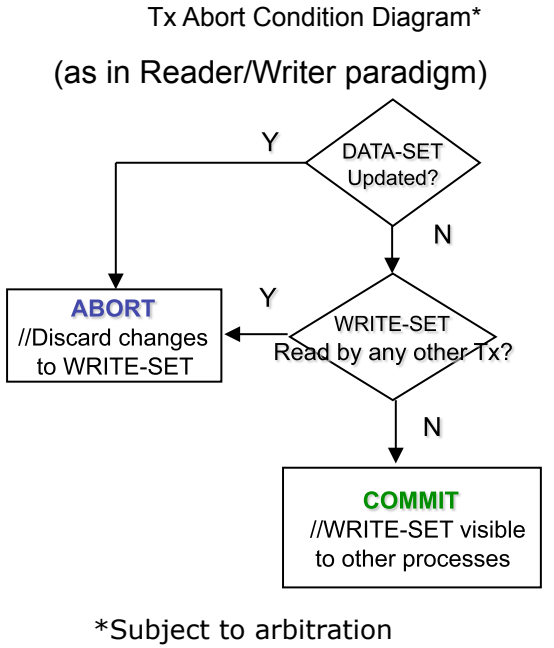
VALIDATE: Verify consistency of read set

ABORT: Unconditionally discard all updates

COMMIT: Attempt to make tentative updates permanent

TM Conflict Detection



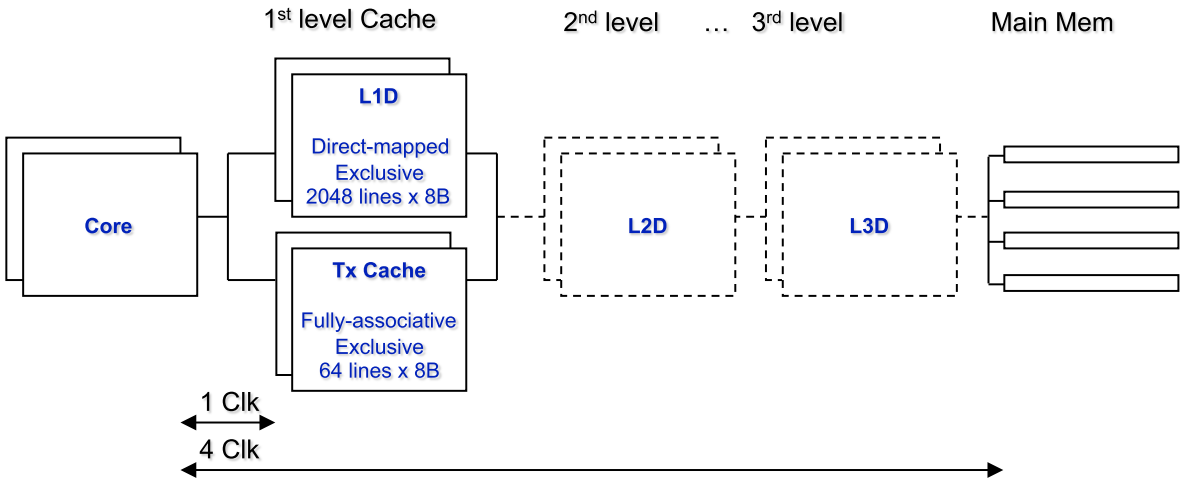


*Subject to arbitration

LOAD and STORE are supported but do not affect tx's READ or WRITE set

- Semantics are left to implementation
- Interaction between tx and non-tx operations to same address is generally a programming error
- If LOAD/STORE are not viewed as committed transactions, with conflict potential, non-linearizable outcomes are possible.

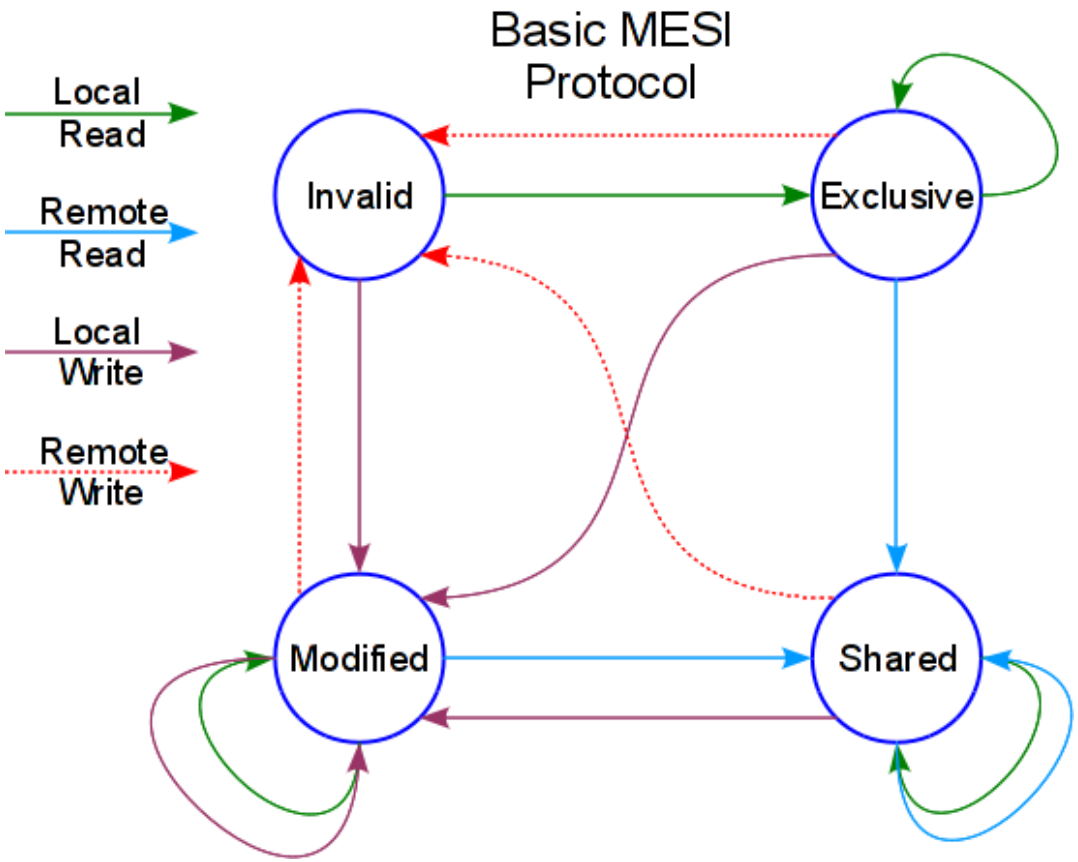
TM Cache Architecture



Tx Cache

- Fully-associative, otherwise how would address collisions be handled?
- Single-cycle COMMIT and ABORT
- Small - size is implementation dependent
- Holds all tentative writes without propagating to other caches or memory
- Upon ABORT modified lines set to INVALID state
- Upon COMMIT, lines can be snooped by other processors, lines written back to memory upon replacement

HW TM Leverages Cache Protocol



M - cache line only in current cache and is modified

E - cache line only in current cache and is not modified

S - cache line in current cache and possibly other caches and is not modified

I - invalid cache line

Tx commit logic detects the following events (akin to R/W locking):

Local read, remote write

- S -> I
- E -> I

Local write, remote write

- M -> I

Local write, remote read

M, snoop read, write back -> S

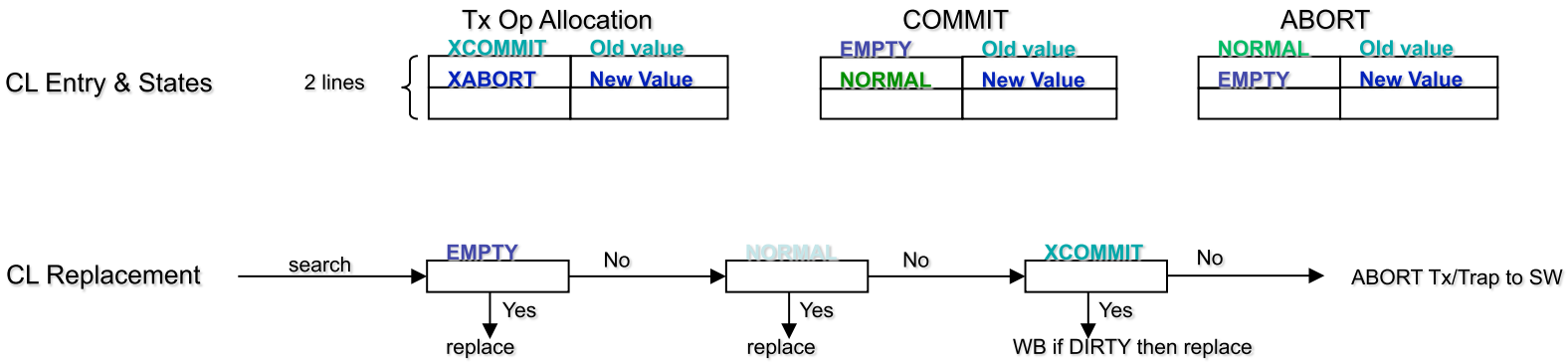
Works with bus-based (snoopy cache) or network-based (directory) architectures

Protocol Changes: Cache States

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

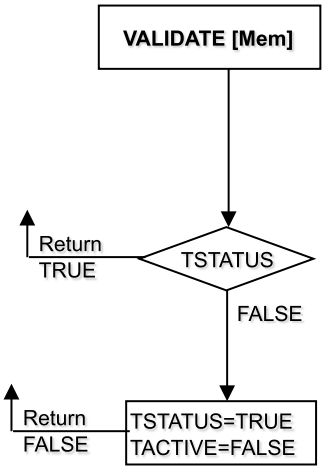
Tx Tags	
Name	**Meaning**
EMPTY	contains no data
NORMAL	contains committed data
XCOMMIT	discard on commit
XABORT	discard on abort

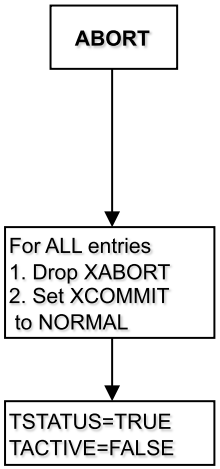
Tx Cache Line Entry, States and Replacement

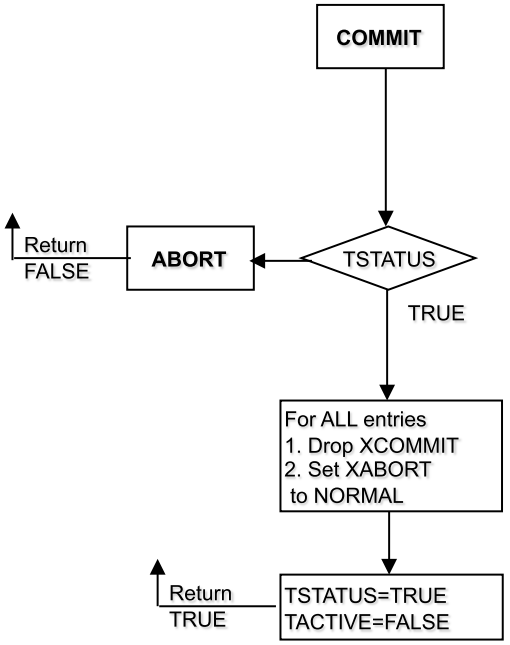


Every transactional operation allocates 2 cache line entries

ISA: TM Verification Operations







Orphan = TACTIVE==TRUE && TSTATUS==FALSE
- Tx continues to execute, but will fail at commit

Commit does not force write back to memory
- Memory written only when CL is evicted or invalidated

Conditions for calling ABORT
- Interrupts
- Tx cache overflow

TM Memory Access

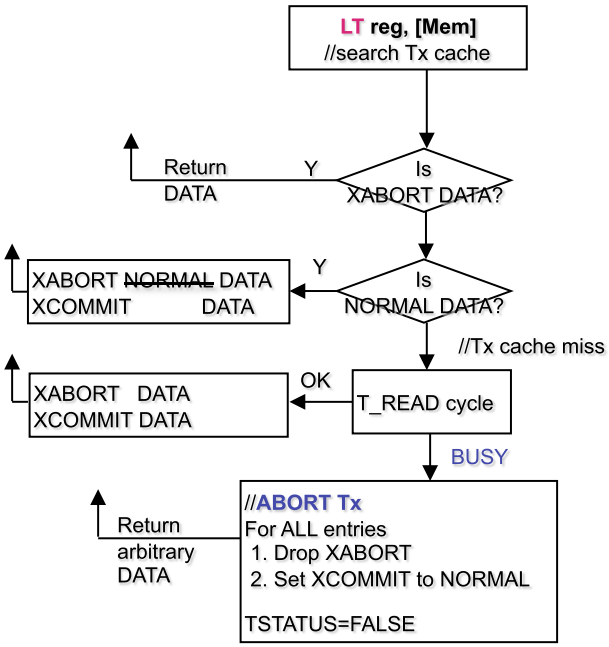
Bus Cycles			
Name	**Kind**	**Meaning**	**New Access**
READ	regular	read value	shared
RFO	regular	read value	exclusive
WRITE	**both**	write back	exclusive
T-READ	Tx	read value	shared
T-RFO	Tx	read value	exclusive
BUSY	Tx	refuse access	unchanged

- Tx requests REFUSED by BUSY response
  - Tx aborts and retries (after exponential backoff?)
  - Prevents deadlock or continual mutual aborts
- Exponential backoff not implemented in HW
  - Performance is parameter sensitive
  - Benchmarks appear not to be optimized

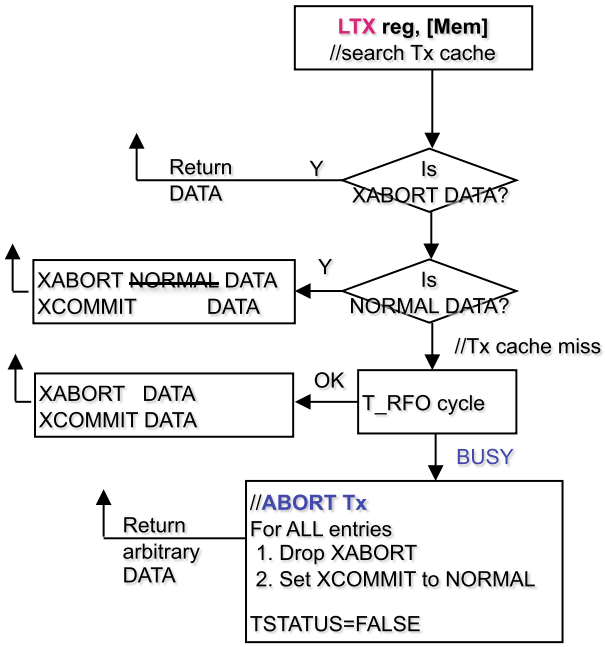
For reference

Tx Op Allocation

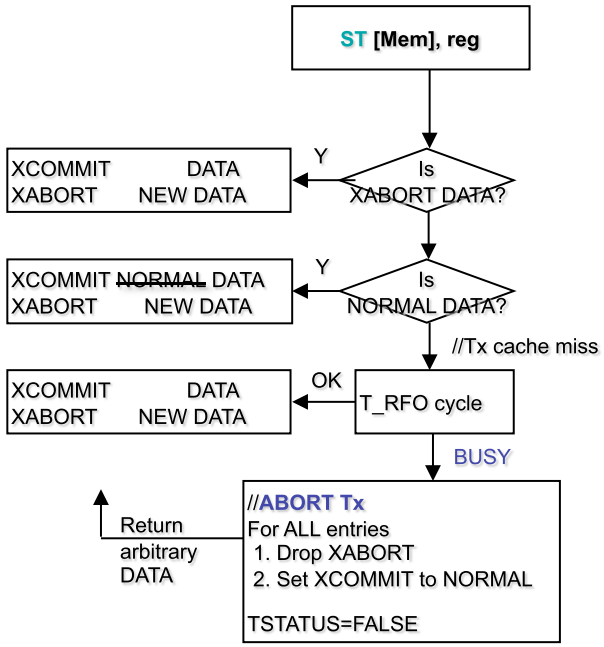
XCOMMIT	Old value
XABORT	New Value



CL State as Goodman's protocol for LOAD (Valid)



CL State as Goodman's protocol for LOAD (Reserved)



CL State as Goodman's protocol for STORE (Dirty)

ST to Tx cache only!!!

TM – Snoopy Cache Actions

Response to SNOOP Actions - Regular $		
Bus Request	**Current State**	**Next State/Action**
Regular Request		
Read	VALID/DIRTY/RESV.	VALID/Return Data
RFO	VALID/DIRTY/RESV.	INVALID/Return Data
Tx Request		
T_Read	VALID	VALID/Return Data
T_Read	DIRTY/RESV.	VALID/Return Data
T_RFO	VALID/DIRTY/RESV.	INVALID/Return Data

Response to SNOOP Actions - Tx $			
Bus Request	**Tx Tag**	**Current State**	**Next State/Action**
Regular Request			
Read	Normal	VALID/DIRTY/RESV.	VALID/Return Data
RFO	Normal	VALID/DIRTY/RESV.	INVALID/Return Data
Tx Request			
T_Read	Normal/Xcommit/Xabort	VALID	VALID/Return Data
T_Read	Normal/Xcommit/Xabort	DIRTY/RESV.	NA/BUSY SIGNAL
T_RFO	Normal/Xcommit/Xabort	DIRTY/RESV.	NA/BUSY SIGNAL

Line Replacement Action - Regular or Tx $	
Bus Request	**Action**
WRITE	write data on Bus, written to Memory

Both Regular and Tx Cache snoop on the bus
Main memory responds to all L1 read misses
Main memory responds to cache line replacement WRITE `s
If TSTATUS==FALSE, Tx cache acts as Regular cache (for NORMAL entries)

Experimental Platform

Implemented in Proetus - execution driven simulator from MIT

- Two versions of TM implemented
  - Goodman's snoopy protocol for bus-based architectures
  - Chaiken directory protocol for (simulated) Alewife machine
- 32 Processors
  - memory latency of 4 clock cycles
  - 1st level cache latency of 1 clock cycles
    2048x8B Direct-mapped regular cache
    64x8B fully-associative Tx cache
- Strong Memory Consistency Model

Comparisons

Compare TM to 4 alternative approaches

- Software
  - TTS (test-and-test-and-set) spinlock with exponential backoff [TTS Lock]
  - SW queuing [MCS Lock]
    Process unable to lock puts itself in the queue, eliminating poll time
- Hardware
  - LOAD_LINKED/STORE_COND with exponential backoff [LL/SC Direct/ Lock]
  - Hardware queuing [QOSB]
    Queue maintenance incorporated into cache-coherency protocol
    Goodman's QOSB protocol - head in memory elements in unused CL's

Test Methodology

Benchmarks

- Counting
  - LL/SC directly used on the single-word counter variable
- Producer & Consumer
- Doubly-Linked List

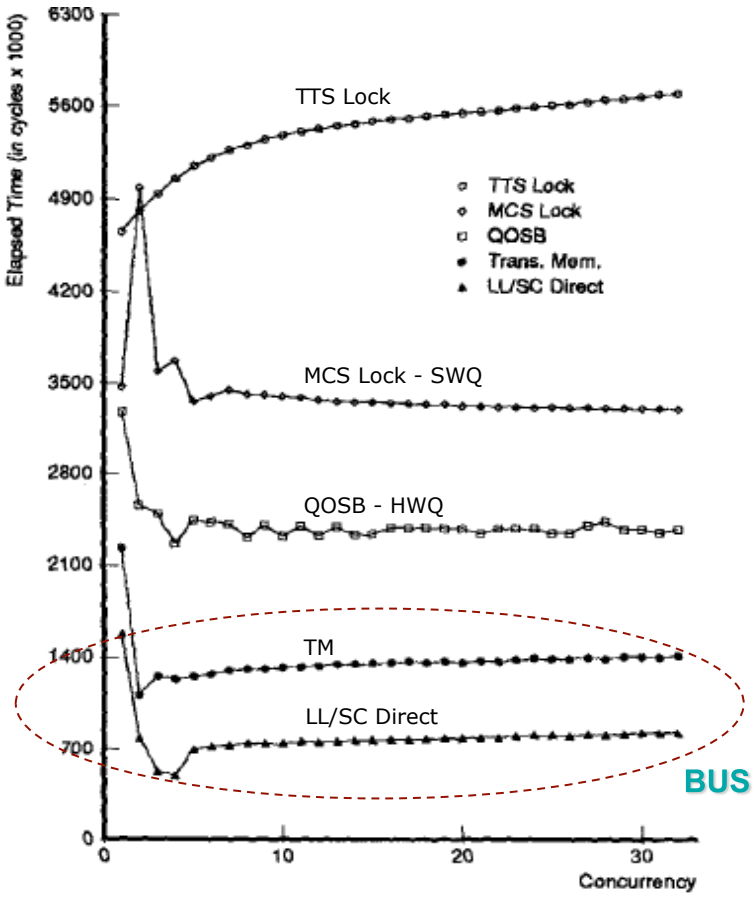
All benchmarks do a fixed amount of work

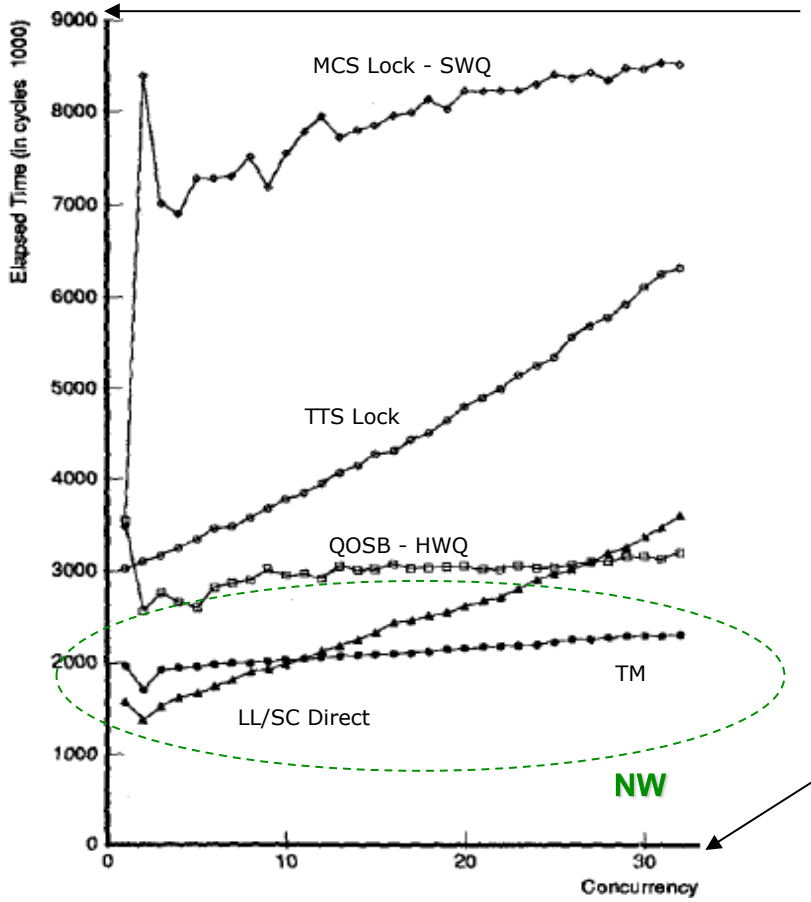
Counting Benchmark

```
void process (int work)
{
    int success = O, backof f = BACKOFF_MIN;
    unsigned wait;
    while (success < work) {
        ST (&counter, LTX (&counter) + 1) ;
        if (COMMIT()) {
            success++;
            backof f = BACKOFF_MIN;
        } else {
            wait = randomo % (01 << backoff) ;
            while (wait-- ) ;
            if (backof f < BACKOFF_MAX )
            backof f ++;
        }
    }
}
```

N processes increment shared counter 2^16/n times, n=1 to 32
Short CS with 2 shared-mem accesses, high contention
In absence of contention, TTS makes 5 references to mem for each increment
    RD + test-and-set to acquire lock + RD and WR in CS + release lock
TM requires only 3 mem accesses
    RD & WR to counter and then COMMIT (no bus traffic)

Counting Benchmark Results





Total cycles needed to complete the benchmark

SOURCE: Figure copied from paper

Concurrent Processes

LL/SC outperforms TM
- LL/SC applied directly to counter variable, no explicit commit required
- For other benchmarks, adv lost as shared object spans multiple words – only way to use LL/SC is as a spin lock

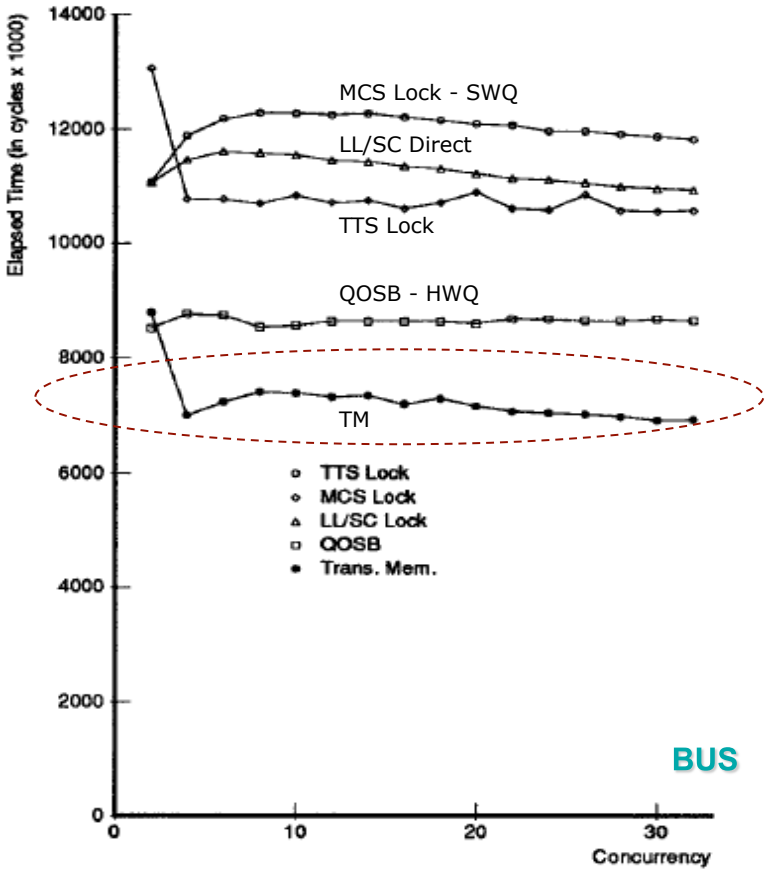
TM has higher throughput than all other mechanisms at most levels of concurrency
- TM uses no explicit locks and so fewer accesses to memory (LL/SC -2, TM-3,TTS-5)

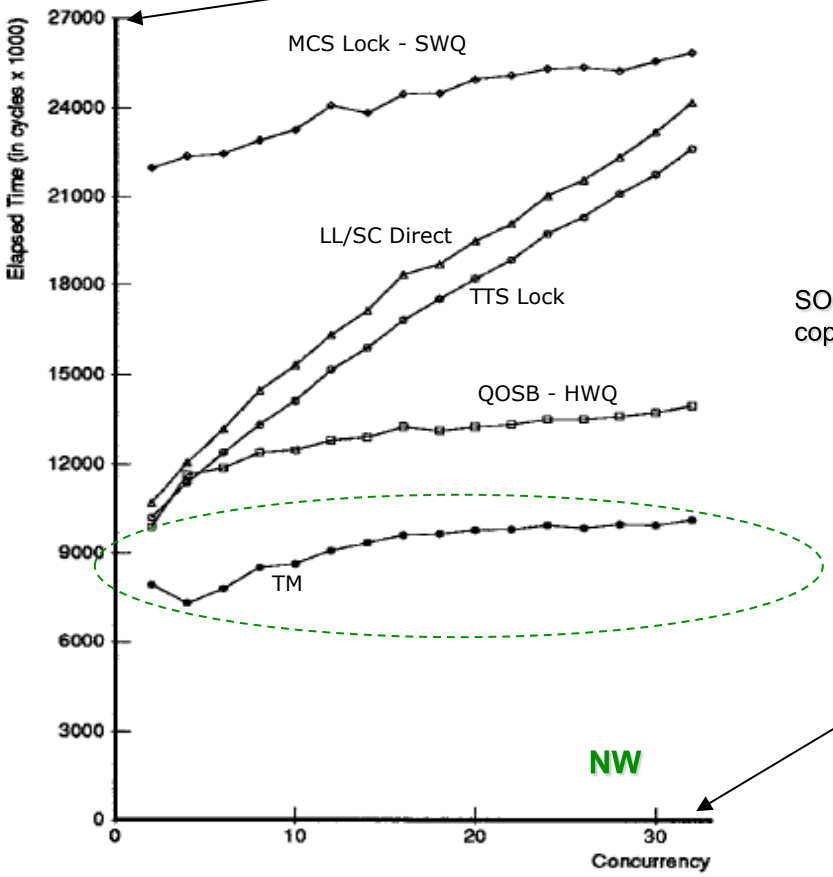
Producer/Consumer Benchmark

```
typedef struct { Word deqs; Word enqs; Word items [QUEUE_SI ZE ] ; } queue;
unsigned queue_deq (queue *q) {
    unsigned head, tail, result, wait;
    unsigned backoff = BACKOFF_MIN
    while (1) {
        result = QUEUE_EMPTY ;
        tail = LTX (&q-> enqs) ;
        head = LTX ( &q->deqs ) ;
        if (head ! = tail) {   /* queue not empty? */
            result = LT (&q->items [head % QUEUE_SIZE] ) ;
            ST (&q->deqs, head + 1) ; /* advance counter */
        }
        if (COMMIT() ) break; .
        wait = randomo % (01 << backoff) ; /* abort => backof f */
        while (wait--) ;
        if (backoff < BACKOFF_MAX) backof f ++;
    }
    return result;
}
```

N/2 producers and N/2 consumers share a bounded buffer, initially empty
Benchmark finishes when 2^16 operations have completed

Producer/Consumer Results





Cycles needed to complete the Benchmark

SOURCE: Figure copied from paper

Concurrent Processes

Doubly-Linked List Benchmark

N processes share a doubly linked list

- Head & Tail pointers anchor the list
- Process Dequeues an item by removing the item pointed by tail and then Enqueues it by adding it at head
- Removing the last item sets both Head & Tail to NULL
- Inserting the first item into an empty list set's both Head & Tail to point to the new item

Benchmark finishes when 2^16 operations have completed

Doubly-Linked List Benchmark

Concurrency difficult to exploit by conventional means

State dependent concurrency is not simple to recognize using locks

- Enquerers don't know if they must lock tail-ptr until after they have locked head-ptr & vice-versa for dequeuers
- Queue non-empty: each Tx modifies head or tail but not both, so enqueuers can (in principle) execute without interference from dequeuers and vice-versa
- Queue Empty: Tx must modify both pointers and enqueuers and dequeuers conflict

Locking techniques use only single lock

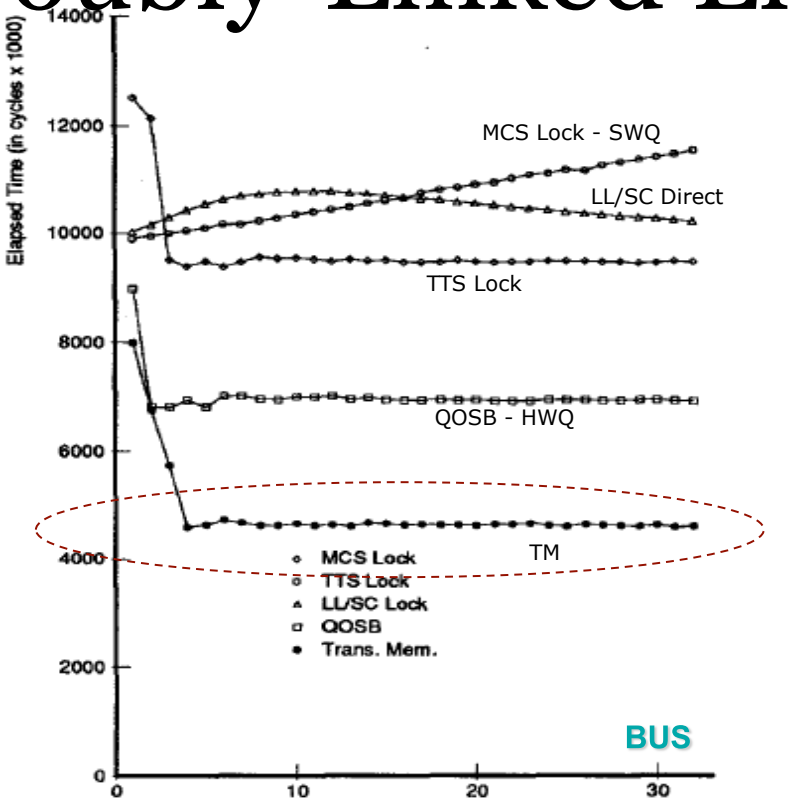
- Lower throughput as single lock prohibits overlapping of enqueues and dequeues

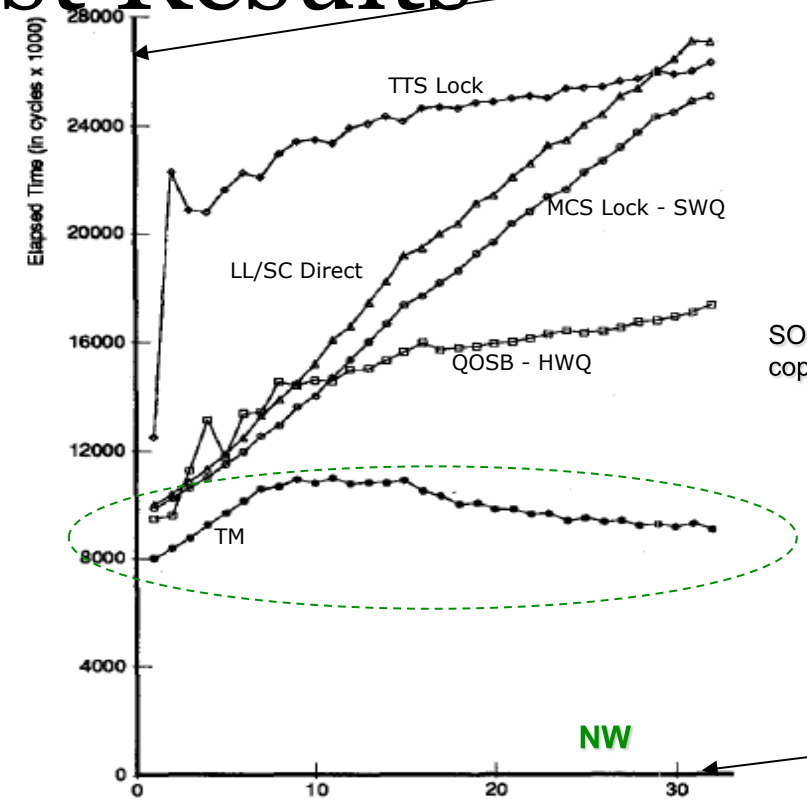
TM naturally permits this kind of parallelism

Doubly-Linked List Benchmark

```
typedef struct list_elem {
struct list elem *next; /* next to dequeue */
struct list elem *prev; /* previously enqueued */
int value;
} entry;
shared entry *Head, *Tail;
void list_enq(entry* new) {
    entry *old tail;
    unsigned backoff = BACKOFF_MIN;
    unsigned wait;
    new->next = new->prev = NULL;
    while (TRUE) {
        old_tail = (entry*) LTX(&Tail);
        if (VALIDATE()) {
            ST(&new->prev, old tail);
            if (old_tail == NULL) ST(&Head, new);
            else ST(&old —tail->next, new);
            ST(&Tail, new);
            if (COMMIT()) return;
        }
        wait = randomo % (01 << backoff);
        while (wait--);
        if (backoff < BACKOFF_MAX) backoff++;
    }
}
```

Doubly-Linked List Results





Cycles needed to complete the Benchmark

SOURCE: Figure copied from paper

Concurrent Processes

Concurrency difficult to exploit by conventional means

- State dependent concurrency is not simple to recognize using locks
  - Enquerers don't know if it must lock tail-ptr until after it has locked head-ptr & vice-versa for dequeuers
  - Queue non-empty: each Tx modifies head or tail but not both, so enqueuers can (in principle) execute without interference from dequeuers and vice-versa
  - Queue Empty: Tx must modify both pointers and enqueuers and dequeuers conflict
- Locking techniques uses only single lock
  - Lower thruput as single lock prohibits overlapping of enqueues and dequeues

TM naturally permits this kind of parallelism

Advantages of TM

TM matches or outperforms atomic update locking techniques for simple benchmarks

TM uses no locks and thus has fewer memory accesses

TM avoids priority inversion, convoying and deadlock

TM's programming semantics are fairly easy

Complex non-blocking algorithms, such as doubly-linked list, are more realizable usingTM

Allows true concurrency?

- It allows disjoint access concurrency
- Should be scalable (for small transaction sizes)

Disadvantages of TM

TM can not perform undoable operations such as I/O

Single cycle commit and abort restrict size of 1st level cache and hence Tx size

Portability is restricted by transactional cache size

Algorithm tuning benefits from SW based adaptive backoff and transactional cache overflow handling

Long transactions have high risk of being aborted by an interrupt, scheduling conflict or transactional conflict

Weaker consistency models require explicit barriers at start and end, impacting performance

Disadvantages of TM

Complications that make it more difficult to implement in hardware:

- Multi-level caches
- Nested Transactions (required for composability)
- Cache coherency complexity on many-core SMP and NUMA architectures

Theoretically subject to starvation

- Adaptive backoff strategy suggested fix - authors used exponential backoff

Poor debugger support

Summary

Wish	Granted?	Comment
Simple programming model	Yes	Very elegant
Avoids priority inversion, convoying and deadlock	Yes	Inherent in NB approach
Equivalent or better performance than lock-based approach	Yes	Only for very small and short tx's
No restrictions on data set size or contiguity	No	Limited by practical considerations
Composable	No	Possible but would add significant HW complexity
Wait-free	No	Possible but would add significant HW complexity

Summary

TM is a novel multi-processor architecture which allows easy lock-free multi-word synchronization in hardware

TM is losely based on the concept of Database Transactions

TM overcomes the single/double-word limitation of CAS and LL/SC

TM implementation exploits cache-coherency mechanisms