Transactional Memory

Zhaoguo Wang

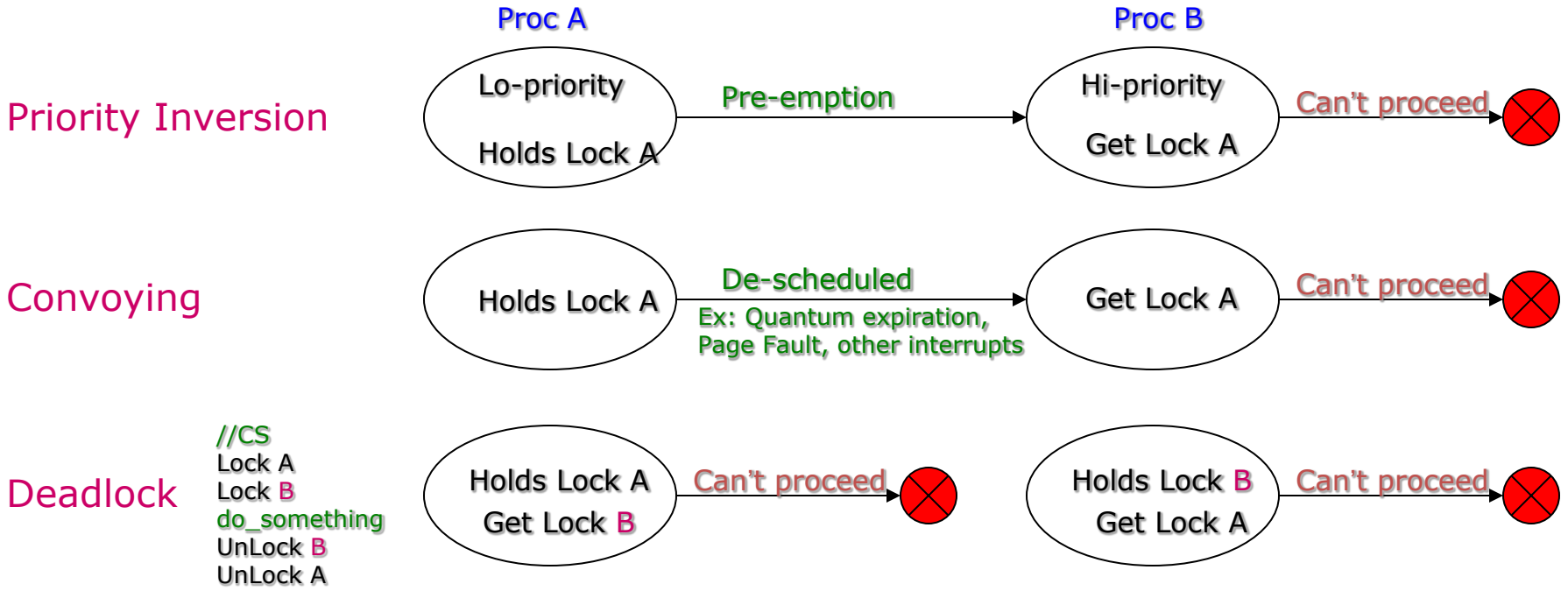
TM paper slides adapted form Ashish Jha PSU CS-510

Outline

Original transaction proposal

A study on Intel's TSX implementation

Conventional Synch Technique



- Uses Mutual Exclusion
  - Blocking i.e. only ONE process/thread can execute at a time
- Easy to use
- Typical problems as seen in highly concurrent systems makes it less acceptable

Lock-Free Synchronization

SW=Software, HW=Hardware, RMW=Read-Modify-Write

Non-Blocking as doesn't use mutual exclusion

Lots of research and implementations in SW

- Uses RMW operations such as CAS, LL&SC
  - limited to operations on single-word or double-words (not supported on all CPU arch)
- Difficult programming logic
- Avoids common problems seen with conventional Techniques such as Priority inversion, Convoying and Deadlock

Experimental evidence cited in various papers, suggests

- in absence of above problems and as implemented in SW, lock-free does not perform as well as their locking-based ones

Basic Idea of Transactional Memory

Leverage existing ideas

HW - LL/SC

- Serves as atomic RMW
- MIPS II and Digital Alpha
  - Restricted to single-word

SW - Database

- Transactions
  - How about expand LL&SC to multiple words
- Apply ATOMICITY
  - COMMIT or ABORT

TM - Transactional Memory

Allows Lock-free synchronization and implemented in HW

- Provides mutual exclusion and easy to use as in conventional techniques

Why a Transaction?

- Concept based on Database Transactions
  - EXECUTE multiple operations (i.e. DB records) and
  - ATOMIC i.e. for "all operations executed", finally
    - COMMIT – if "no" conflict
    - ABORT – if "any" conflict
- Replaces the Critical Section

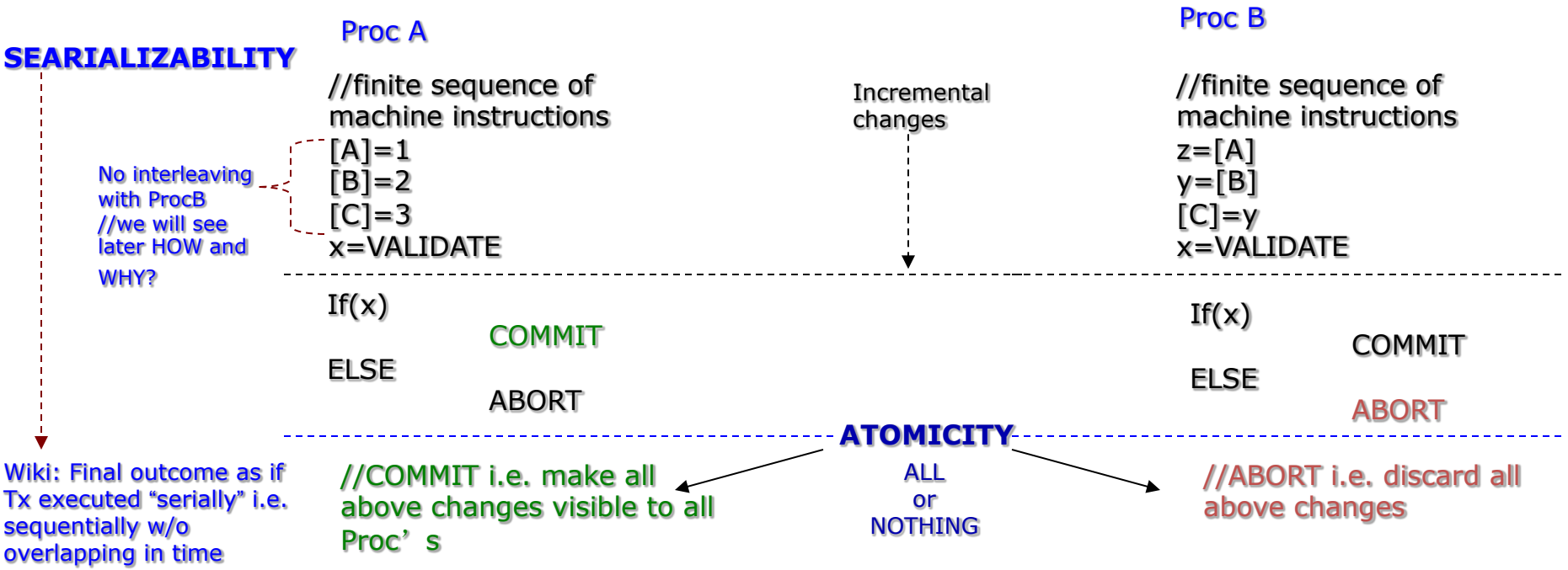
How Lock-Free?

- allows RMW operations on multiple, independently chosen words of memory
  - concept based on LL & SC implementations as in MIPS, DEC Alpha
    - But not limited to just 1 or 2 words
- Is non-blocking
  - Multiple transactions for that CS could execute in Parallel (on diff CPU's) but
    - Only ONE would SUCCEED, thus maintaining memory consistency

Why implemented in HW?

- Leverage existing cache-coherency protocol to maintain memory consistency
- Lower cost, minor tweaks to CPU's
  - Core
  - Caches
  - ISA
  - Bus and Cache coherency protocols

A Transaction & its Properties

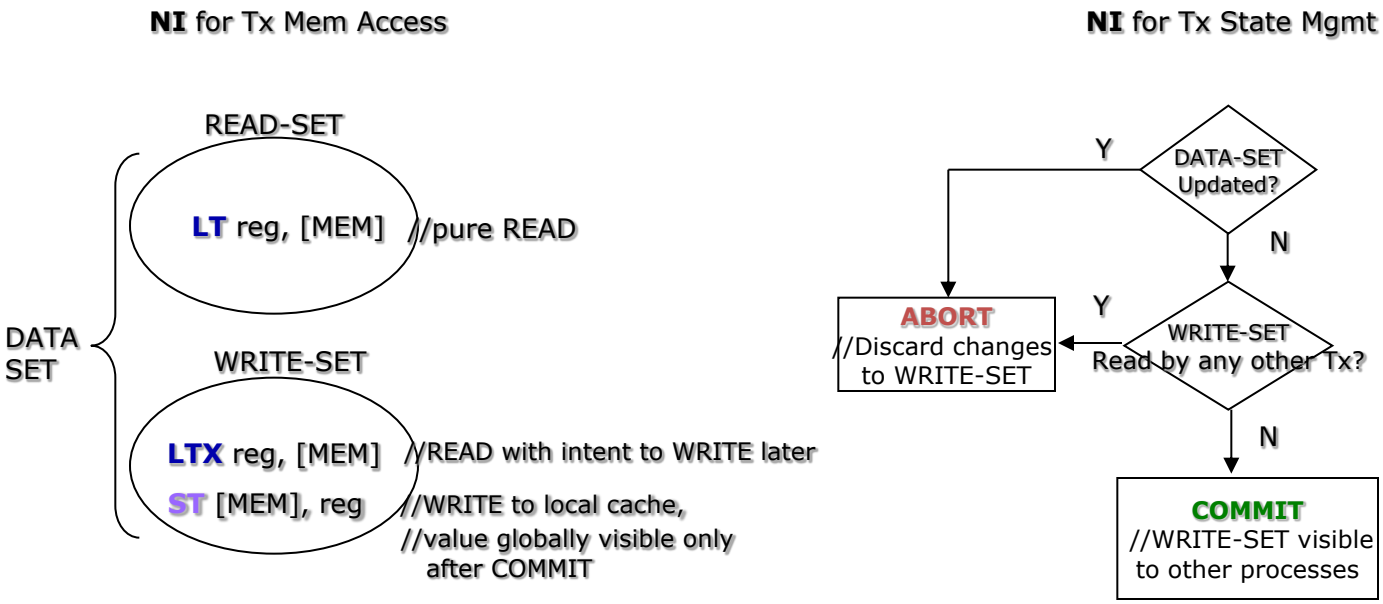


guaranteed Atomic and Serializable behavior

Assumption

a process executes only one Tx at a time

ISA for TM



A Transaction consist of above NI's

allows programmer to define

customized RMW operations

operating on "independent arbitrary region of memory" not just single word

NOTE

Non-transactional operations such as LOAD and STORE are supported but does not affect Tx's READ or WRITE set

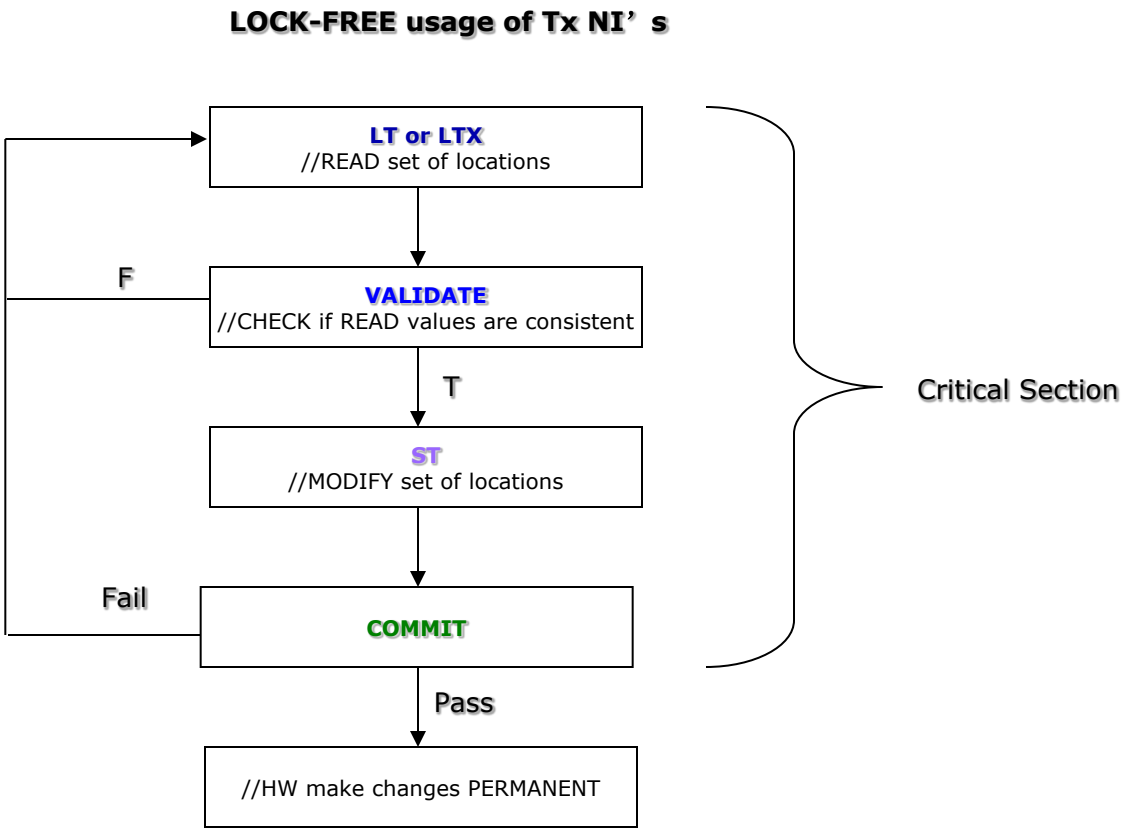
Left to implementation

Interaction between Tx and non-Tx operations

actions on ABORT under circumstances such as Context Switch or interrupts (page faults, quantum expiration)

Avoid or resolve serialization conflicts

TM NI Usage



Tx's intended to replace "short" critical section
- No more acquire_lock, execute CS, release_lock

Tx satisfies Atomicity and Serializability properties

Ideal Size and duration of Tx's implementation dependent, though
- Should complete within single scheduling quantum
- Number of locations accessed not to exceed architecturally specified limit
  - What is that limit and why only short CS? – later…

TM – Implementation

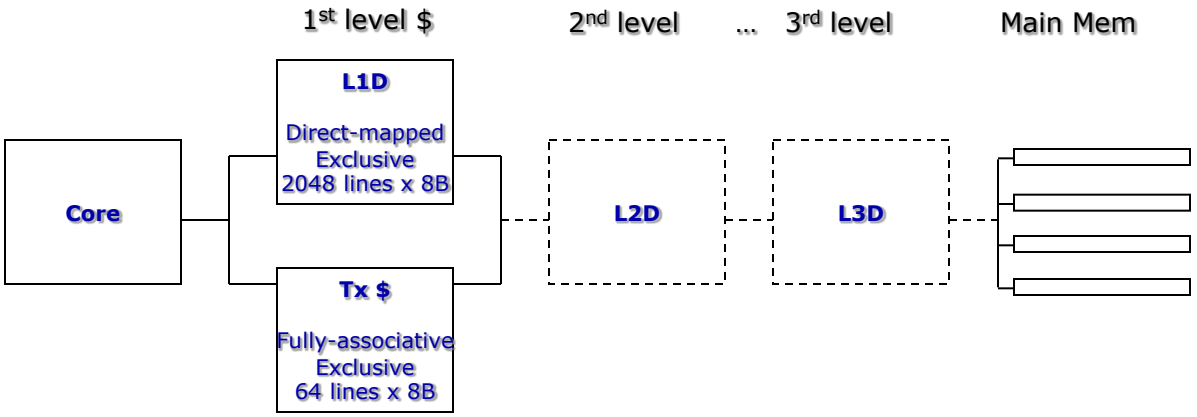
- Design satisfies following criteria
  - In absence of TM, Non-Tx ops uses same caches, its control logic and coherency protocols
  - Committing or Aborting a Tx is an operation "local" to the cache
    - Does not require communicating with other CPU's or writing data back to memory

- TM exploits cache states associated with the cache coherency protocol
  - Available on Bus based (snoopy cache) or network-based (directory) architectures
  - Cache State could be in one of the forms - think MESI or MOESI
    - SHARED
      - Permitting READS, where memory is shared between ALL CPUs
    - EXCLUSIVE
      - Permitting WRITES but exclusive to only ONE CPU
    - INVALID
      - Not available to ANY CPU (i.e. in memory)

- BASIC IDEA
  - The cache coherency protocol detects "any access" CONFLICTS
    - Apply this state logic with Transaction counterparts
      - At no extra cost
  - If any Tx conflict detected
    - ABORT the transaction
  - If Tx stalled
    - Use a timer or other sort of interrupt to abort the Tx

TM – Paper's Implementation

Example Implementation in the Paper



Proteus Simulator
- 32 CPUs
- Two versions of TM implementation
  - Goodmans's snoopy protocol for bus-based arch
  - Chaiken directory protocol for Alewife machine

Two Primary caches
- To isolate traffic for non-transactional operations

Tx $
- Small – note "small" – implementation dependent
- Exclusive
- Fully-associative
  - Avoids conflict misses
- single-cycle COMMIT and ABORT
- Similar to Victim Cache
  - Holds all tentative writes w/o propagating to other caches or memory
- ABORT
  - Lines holding tentative writes dropped (INVALID state)
- COMMIT
  - Lines could be snooped by other processors
  - Lines WB to mem upon replacement

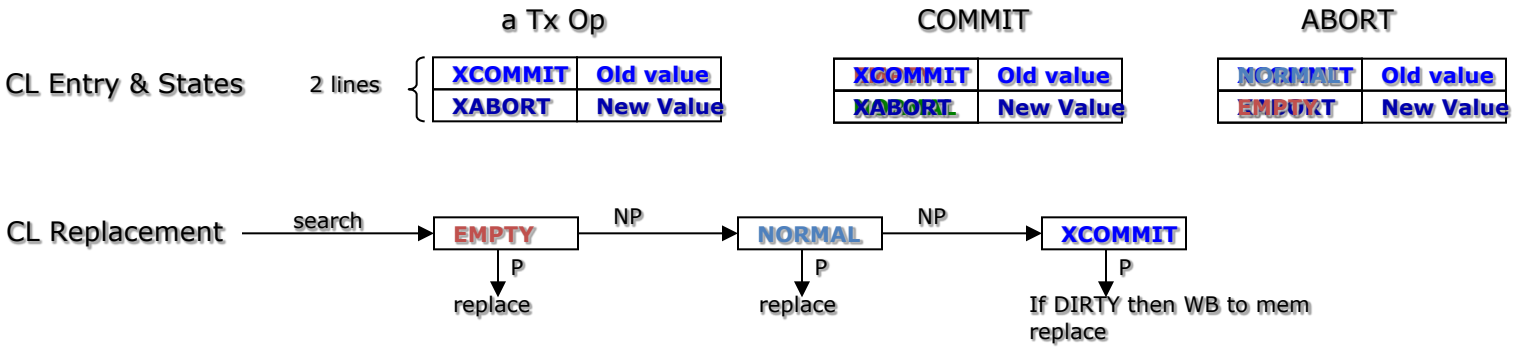
TM Impl. – Cache States & Bus Cycles

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

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 Cache Line Entry, States and Replacement

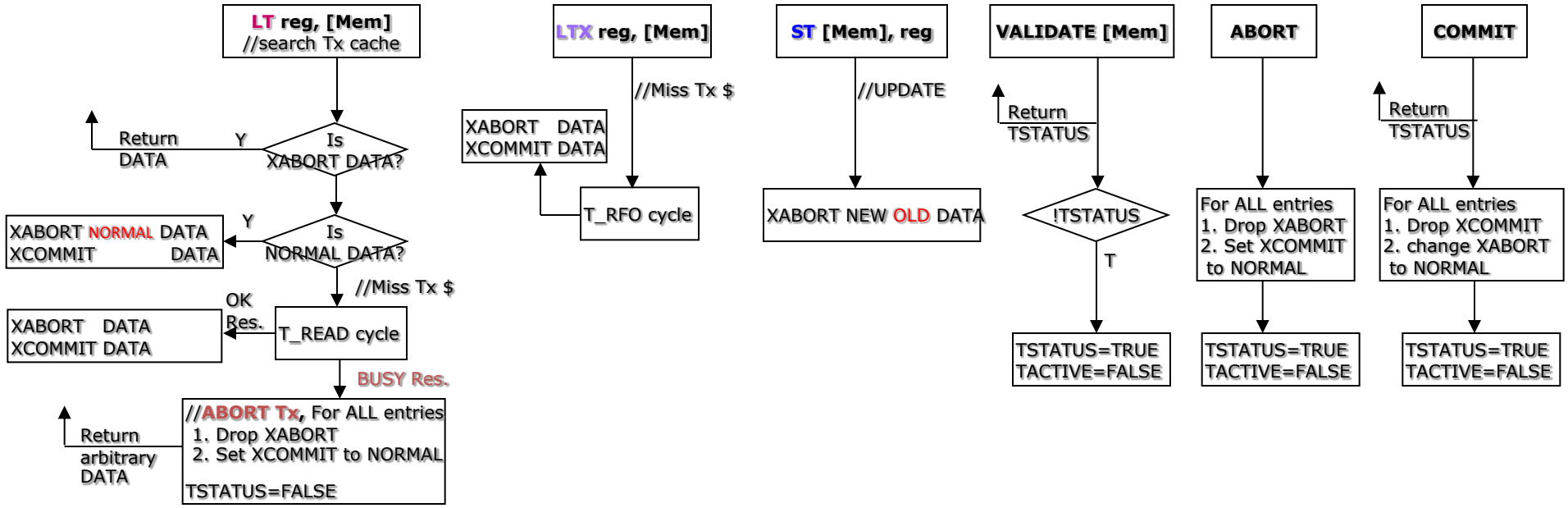


- A dirty value "originally read" must either be
  - WB to mem, or
  - Allocated to XCOMMIT entry as its an "old" entry
    - avoids continuous WB's to memory and improves performance
- Tx requests REFUSED by BUSY response
  - Tx aborts and retries
    - Prevents deadlock or continual mutual aborts
    - Theoretically subject to starvation
      - Could be augmented with a queuing mechanism
- Every Tx Op takes 2 CL entries
  - Tx cache cannot be big due to perf considerations - single cycle abort/commit + cache management
    - Hence, only short Tx size supported

TM Impl. – CPU Actions

CPU Flags
- **TACTIVE** – is Tx in Progress? Implicitly set when Tx executes its first Op – meaning start of CS
- **TSTATUS** – TRUE if Tx is Active, FALSE if Aborted

TSTATUS=TRUE



- Other conditions for ABORT
  - Interrupts
  - Tx $ overflow
- Commit does not force changes to memory
  - Taken care (i.e. mem written) only when CL is evicted or invalidated by cache coherence protocol

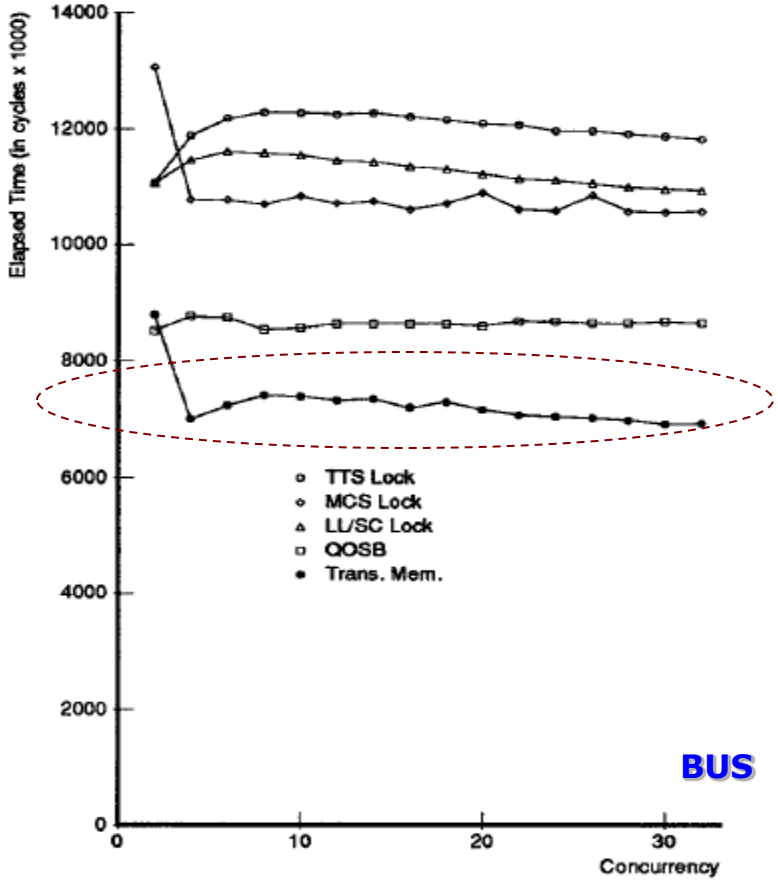
Test - Methodology

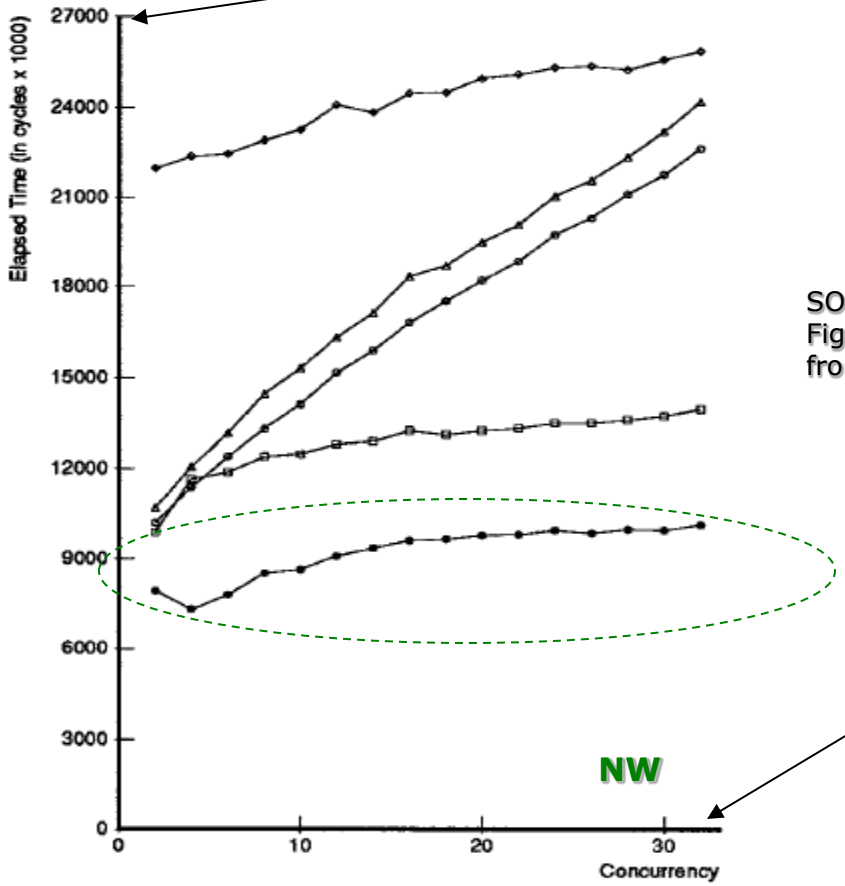
- TM implemented in Proetus sim - execution driven simulator from MIT
  - Two versions of TM implementation
    - Goodman's snoopy protocol for bus-based arch
    - Chaiken directory protocol for (simulated) Alewife machine
  - 32 Processors
    - mem latency of 4 clks
    - 1st level $ latency of 1 clk
      - 2048x8B Direct-mapped Regular $
      - 64x8B fully-associative Tx $
  - Strong Memory Consistency Model

- Compare TM to 4 different implementation Techniques
  - SW
    - TTS (test-and-test-and-set) spinlock with exponential backoff
    - SW queuing
      - Process unable to lock puts itself in the queue, eliminating poll time
  - HW
    - LL/SC (LOAD_LINKED/STORE_COND) with exponential backoff
    - HW queuing
      - Queue maintenance incorporated into cache-coherency protocol
        - » Goodman's QOSB protocol - head in mem, elements in unused CL's

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

Test – Prod/Cons Results





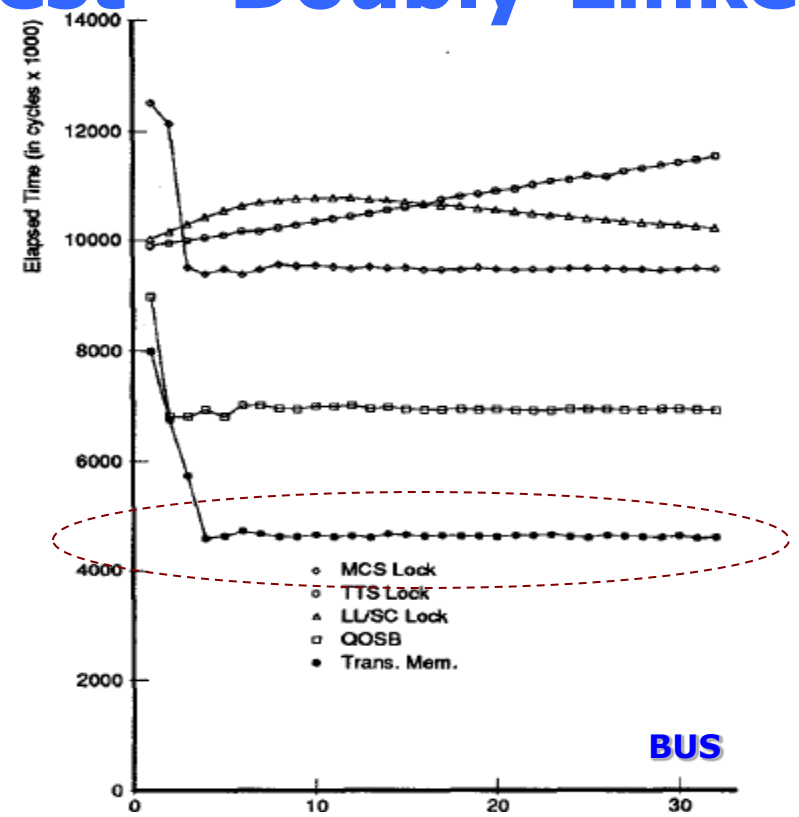
Cycles needed to complete the Benchmark

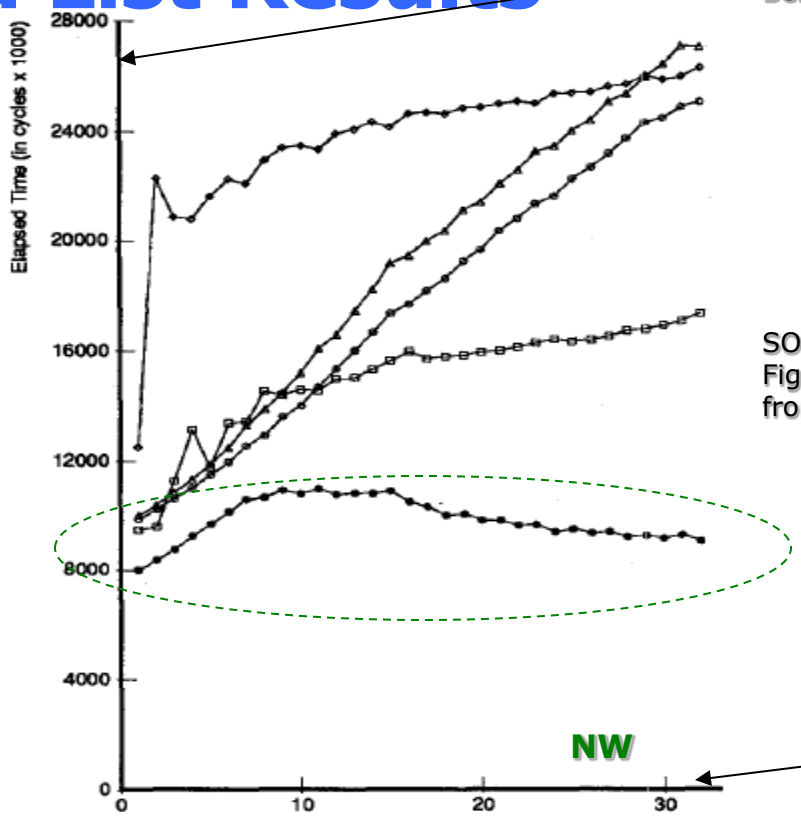
SOURCE: Figure copied from paper

Concurrent Processes

- In Bus arch, almost flat TPT
  - TM yields higher TPT but not as dramatic as counting benchmark
- In NW arch, all TPT suffers as contention increases
  - TM suffers the least and wins

Test – Doubly-Linked List Results





Cycles needed to complete the Benchmark

SOURCE: Figure copied from paper

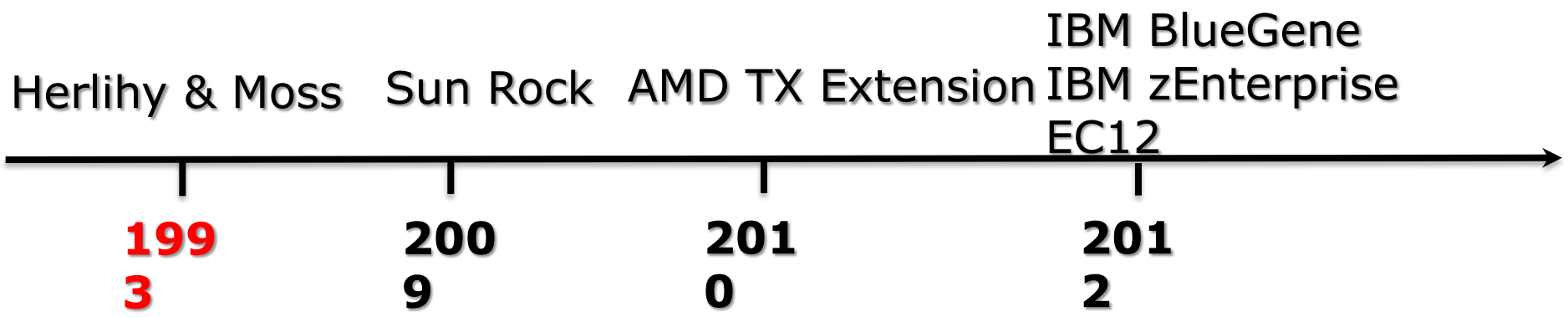
Concurrent Processes

- 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 TPT as they don't allow overlapping of enqueues and dequeues
- TM naturally permits this kind of parallelism

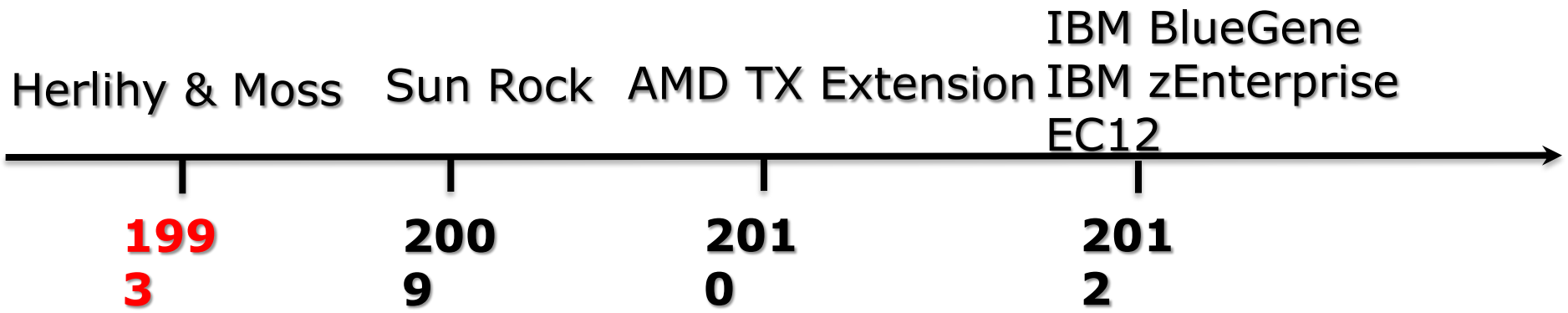
Summary of Original Proposal

- TM is direct generalization from LL&SC of MIPS II & Digital Alpha
  - Overcoming single-word limitation long realized
    - Motorola 68000 implemented CAS2 – limited to double-word
- TM is a multi-processor architecture which allows easy lock-free multi-word synchronization in HW
  - exploiting cache-coherency mechanisms, and
  - leveraging concept of Database Transactions
- TM matches or outperforms atomic update locking techniques (shown earlier) for simple benchmarks
  - Even in absence of priority inversion, convoying & deadlock
    - uses no locks and thus has fewer memory accesses

Hardware Transactional Memory

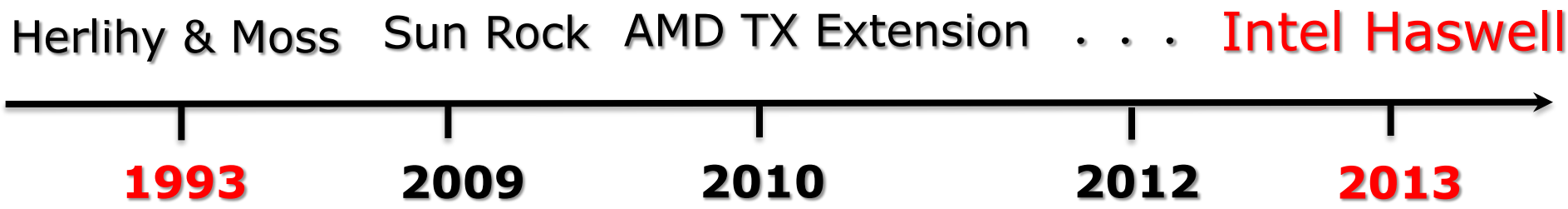


Hardware Transactional Memory



High end server or research prototype

Hardware Transactional Memory



Massively available

Intel Haswell

Restricted Transactional Memory (RTM)

- Hardware transactional memory with limitations

Major limitations

- Working set is limited
- Some system events abort the TX

New instruction set

- Xbegin, Xend, Xabort

Programming With RTM

```
RTM Usage
    if _xbegin() == _XBEGIN_STARTED
            do some critical work
        _xend()
    else
        fallback routine
```

Handle the abort event

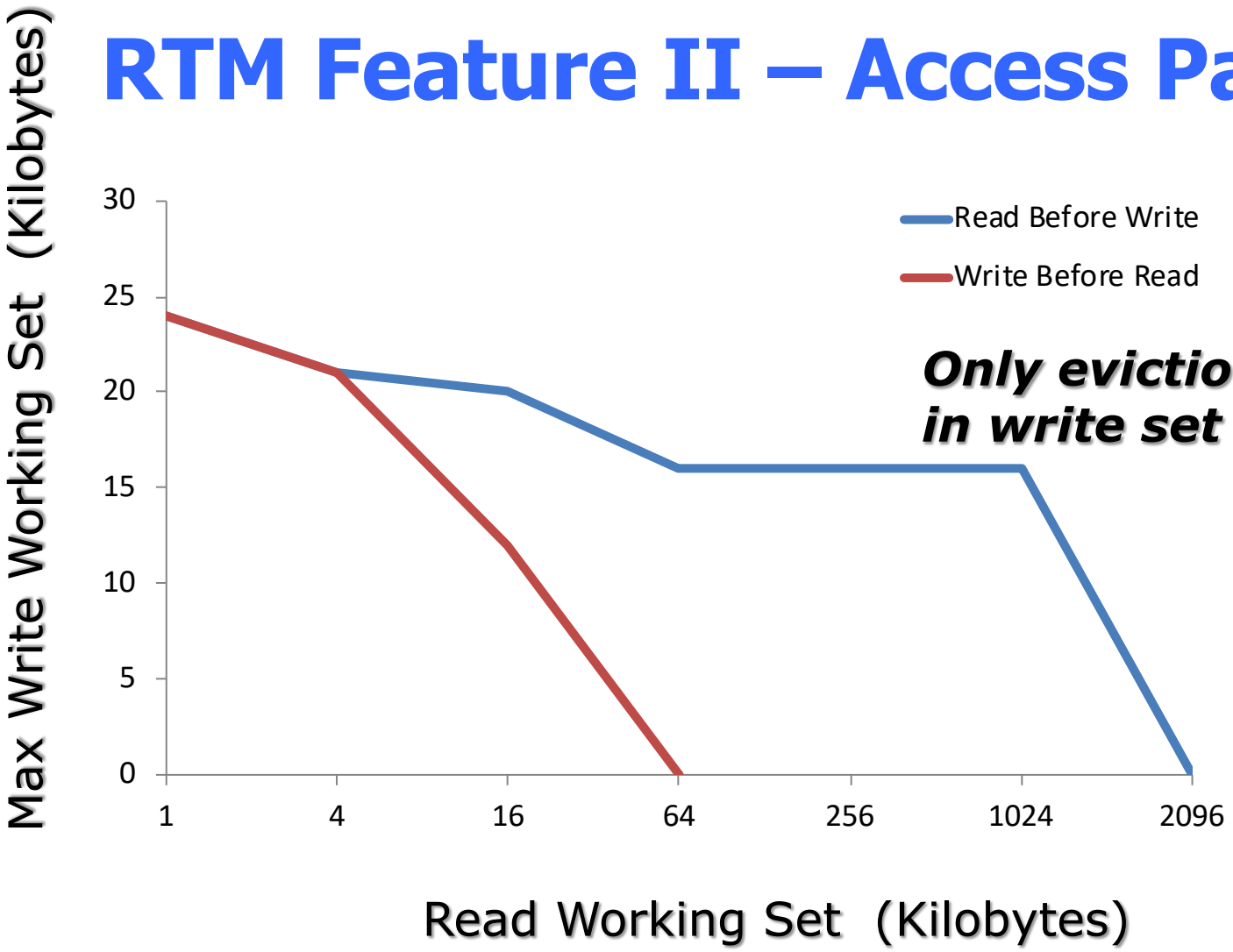
RTM Feature I – Asymmetric Read/Write Limits

	Max Read Set	Max Write Set
No HT	4MB	24KB
With HT	1.8MB	24KB

L1-Cache tracks writes
An implementation specific structure tracks reads

RTM prefers reads than writes

RTM Feature II – Access Patterns



Only eviction of cachelines in write set will abort TX

RTM prefers read before write

Case Study I – Data Structure

Commonly used data structures

- B+ tree, Hashtable, Cuckoo Hash, Skiplist

Operation feature

- First read (find the position) then write (update)

Study on B+ tree

Use RTM to Scale Up B+ Tree

Insert(*key, value*):

RTM Begin

Traverse the tree from root

If need

do split

Insert the record

RTM End

Get(*key*):

RTM Begin

Traverse the tree from root

Return the value

RTM End

Use RTM to protect the entire operation

Experiment Setup

Hardware

- Quad core
- 32KB , 8-way, L1 cache
- 32GB memory

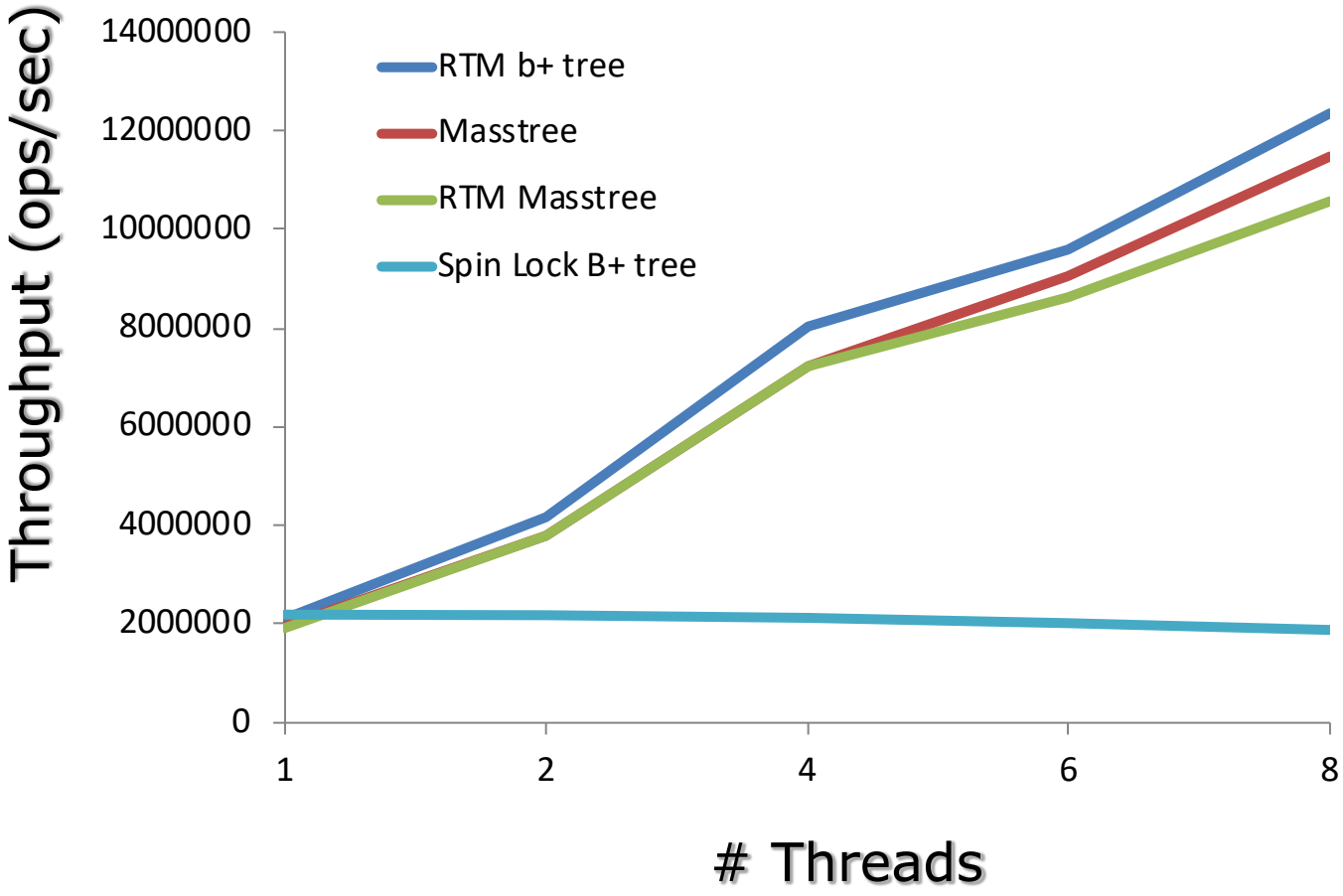
Tree Structure

- Spanout: 15
- Key length: 64 bit

Masstree

- Eurosys 2012
- A highly tuned B+ tree for multicore when key length is 64 bit

RTM B+ Tree vs. Masstree – YCSB



RTM B+ tree outperforms masstree due to its simple