NICTA

A Scalable Lock Manager for Multicores

<u>Hyungsoo Jung</u> — NICTA

Hyuck Han — Samsung Electronics

Alan Fekete — University of Sydney

Gernot Heiser — NICTA

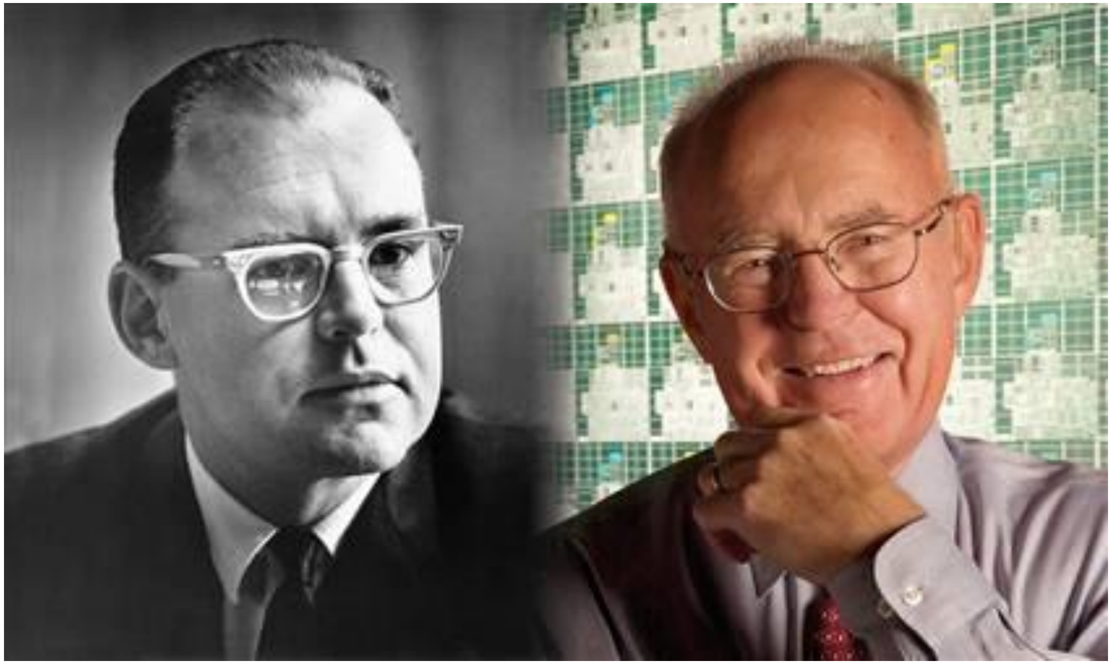
Heon Y. Yeom — Seoul National University

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

@University of Sydney

Moore's Law

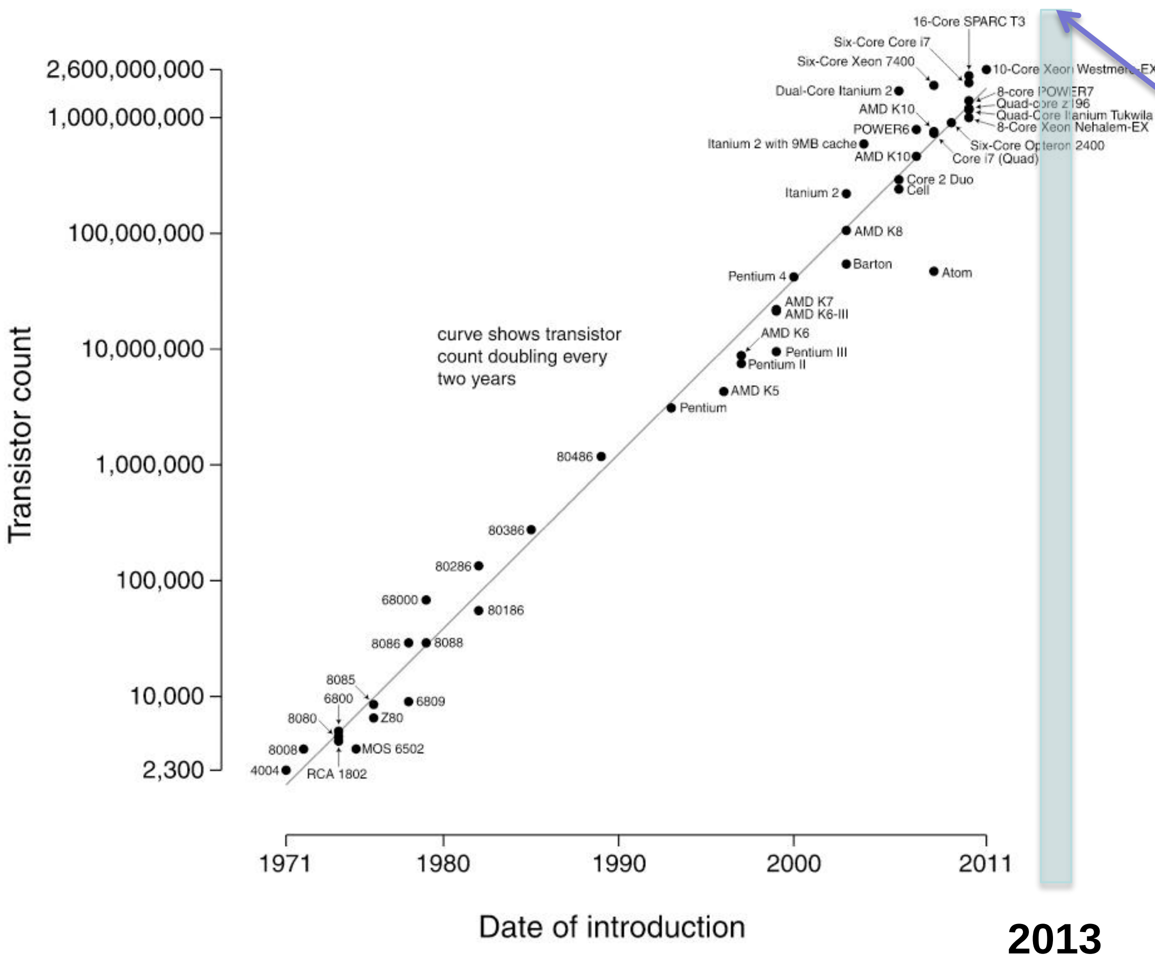


"The number of transistors incorporated in a chip will approximately ***double*** *every 24 months."*

--Gordon Moore, Intel co-founder

Moore’s Law

Microprocessor Transistor Counts 1971-2011 & Moore’s Law



2013: IBM’s System z processor
5.7GHz and with 2.75B transistors.

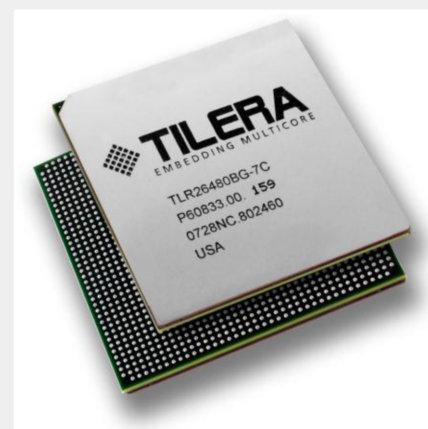
Moore's Law goes Multicores

But what about the software?

Database Management Systems : our focus !!!!

Enterprise Software Systems (Not explored completely)

Operating System
"Linux is not scalable,
See [OSDI 2010, EuroSys2012, ASPLOS 2012]"



MULTICORE MACHINES

This research tries to solve ..

- Multi-core scalability problems of DBMS engines (running at SERIALIZABLE isolation) by eliminating latching overhead in a lock manager.
  - Keep overall architecture the same
  - Unlike larger redesigns proposed by Johnson et al. and Thomson et al.

- Now let's see some background.

Latch protecting **Lock** table (MySQL)

NICTA

Lock Acquire in Growing Phase

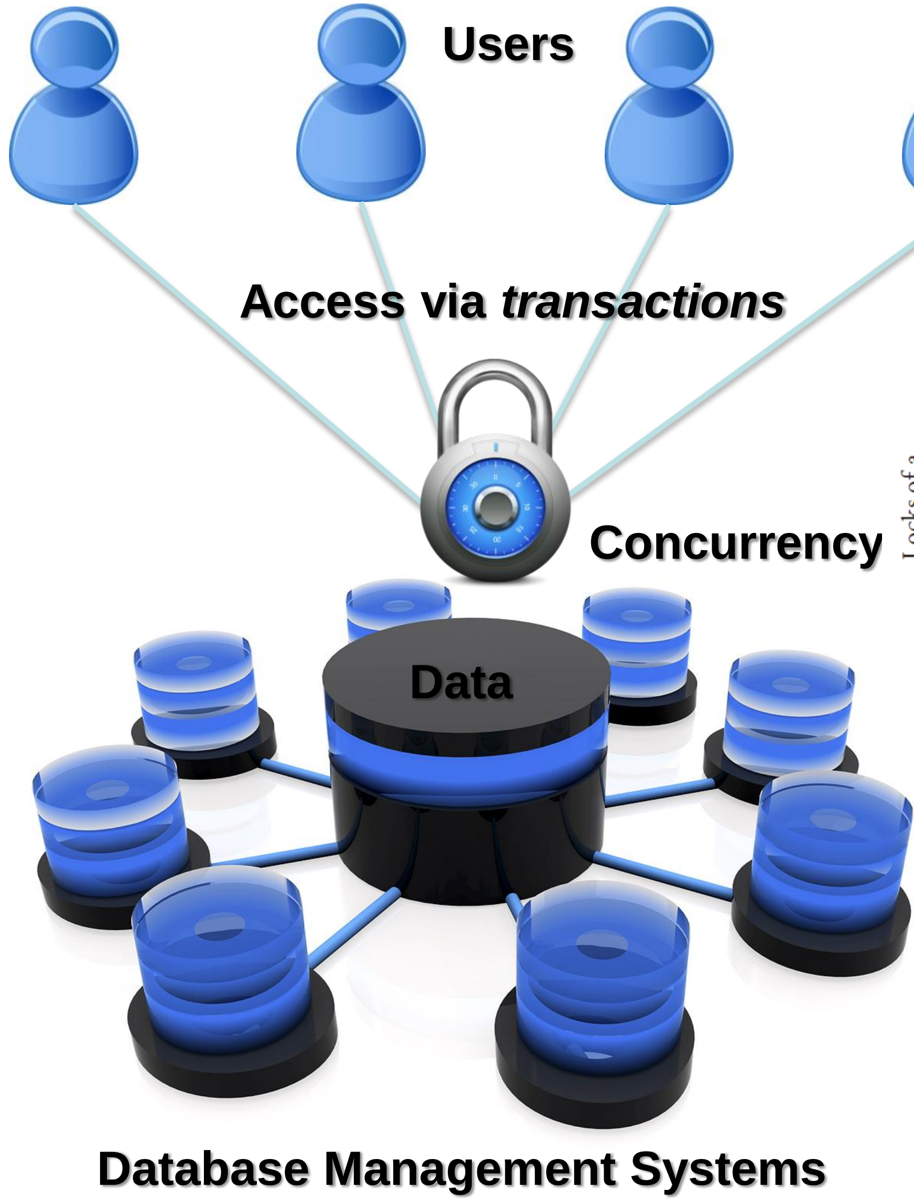
```
mutex_enter(lock_table->mutex);
n_lock = lock_create();
n_lock->state = ACTIVE;
lock_insert(n_lock);
for all locks (lock) in hash_bucket
  if (lock is incompatible with n_ lock)
        n_lock->state = WAIT;
        if (deadlock_check() ==TRUE)
              abort Tx;
         break;
  else
        continue;
  end if
end for
mutex_exit(lock_table->mutex);
if (n_lock->state==WAIT)
    mutex_enter(Tx->mutex);
    Tx->state = WAIT;
    os_cond_wait(Tx->mutex);
    mutex_exit(Tx->mutex);
end if
```

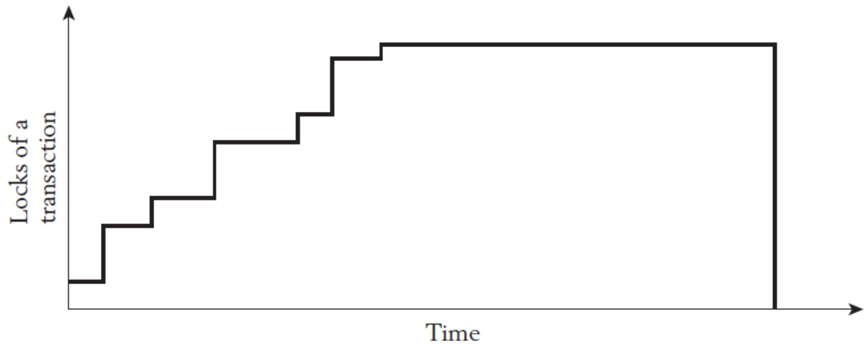
Lock Release in Shrinking Phase

```
mutex_enter(lock_table->mutex);
for all locks (lock1) in Tx
  lock_release(lock1);
  for all locks (lock2) following lock1
    if ( lock2 doesn't have to wait )
      lock_grant( lock2 );
      lock2->state=ACTIVE;
    end if
  end for
end for
mutex_exit(lock_table->mutex);
```

Lock Table Mutex (or Latch)

Lock vs. Latch : Database Lock



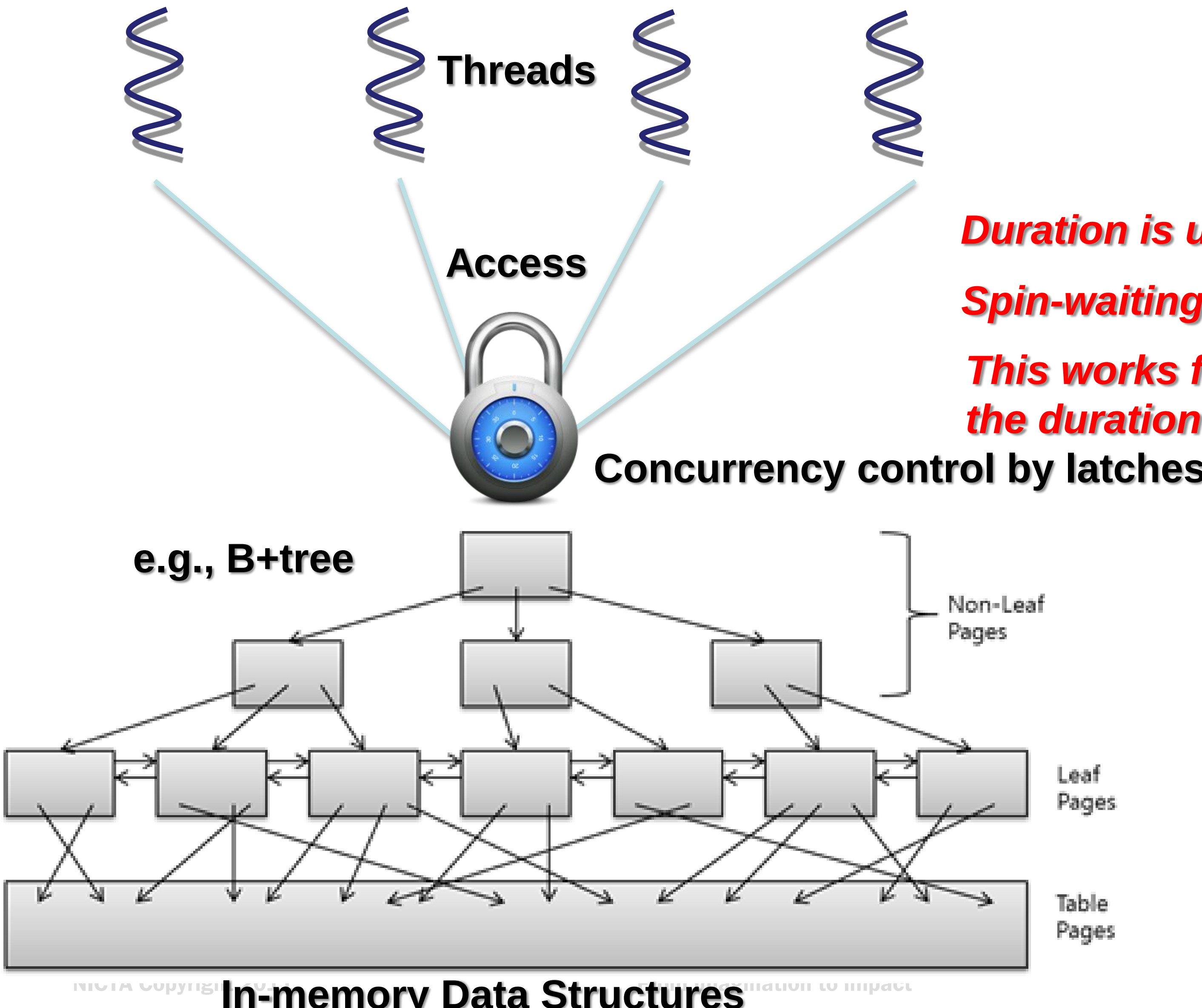


Duration is long (S2PL)

Sleeping when locks conflict

Lock conflicts don't cause the observed performance collapse !!!!

Lock vs. Latch : Latch

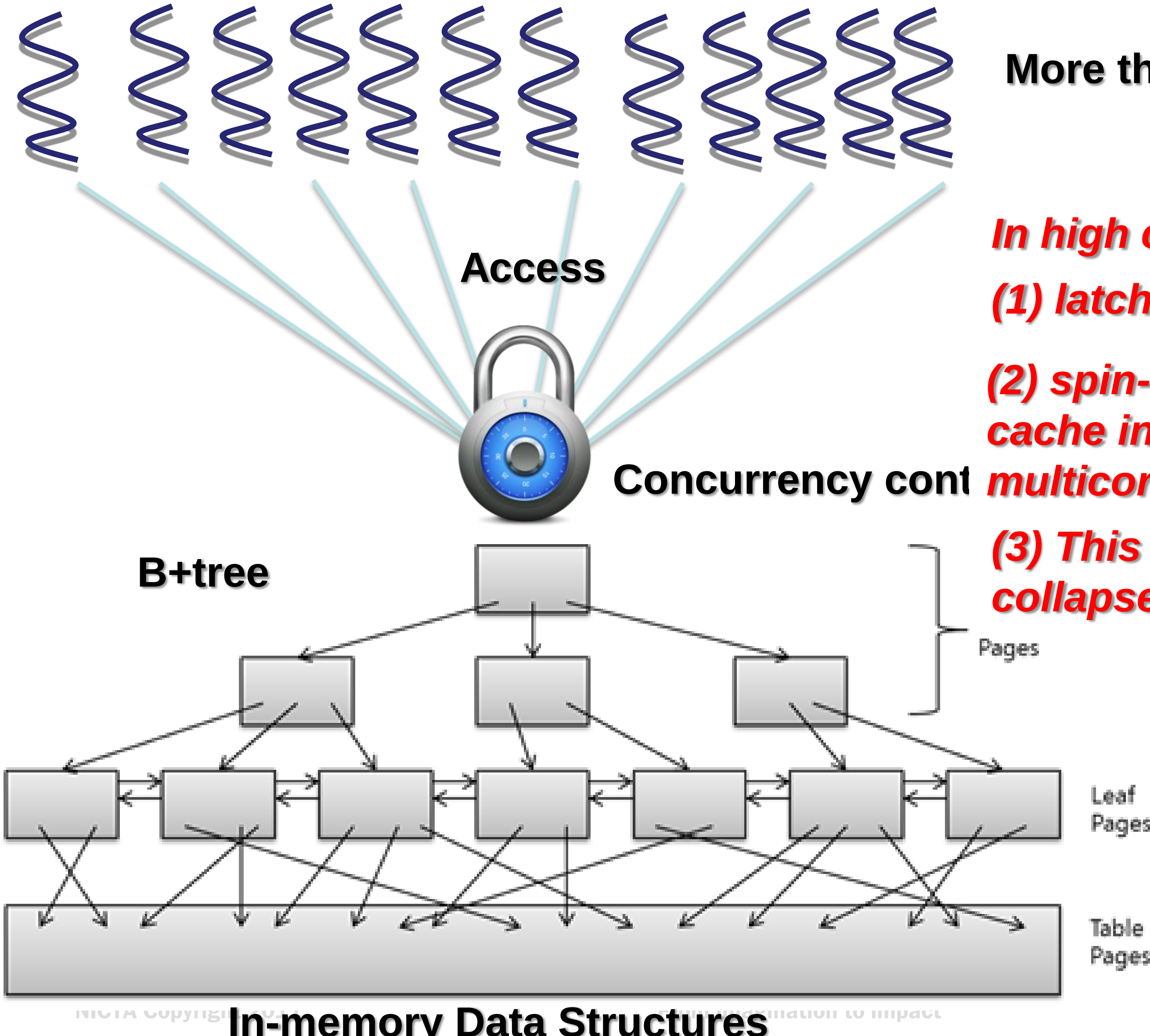


Duration is usually very short

Spin-waiting on contention

This works fine as long as the duration is really short.

Lock vs. Latch : High latch contention



In high contention :

(1) latch duration gets longer

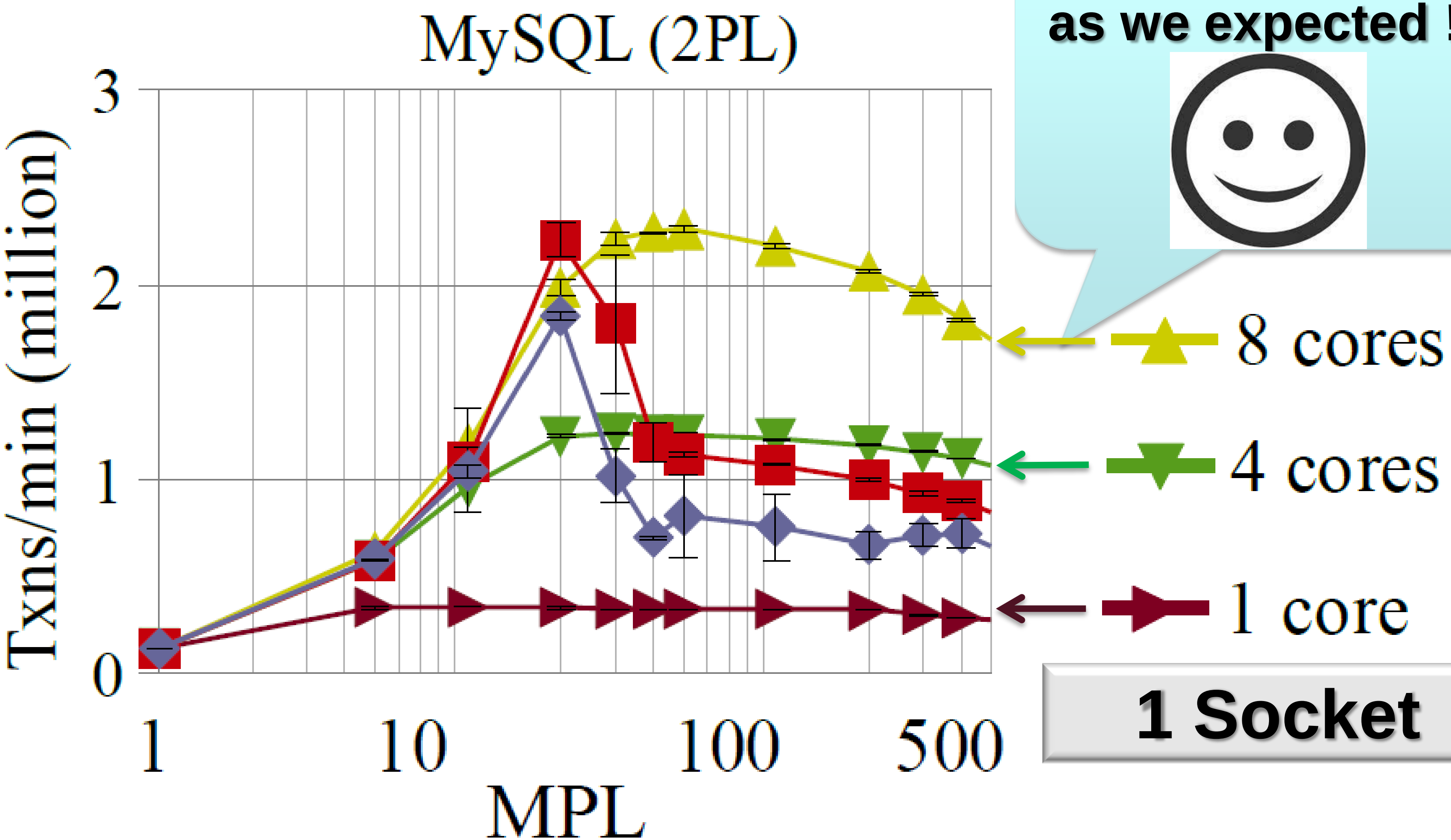
(2) spin-waiting incurs the cache invalidation storm on multicores!!!

(3) This causes performance collapse !!!!

How bad is the performance collapse?

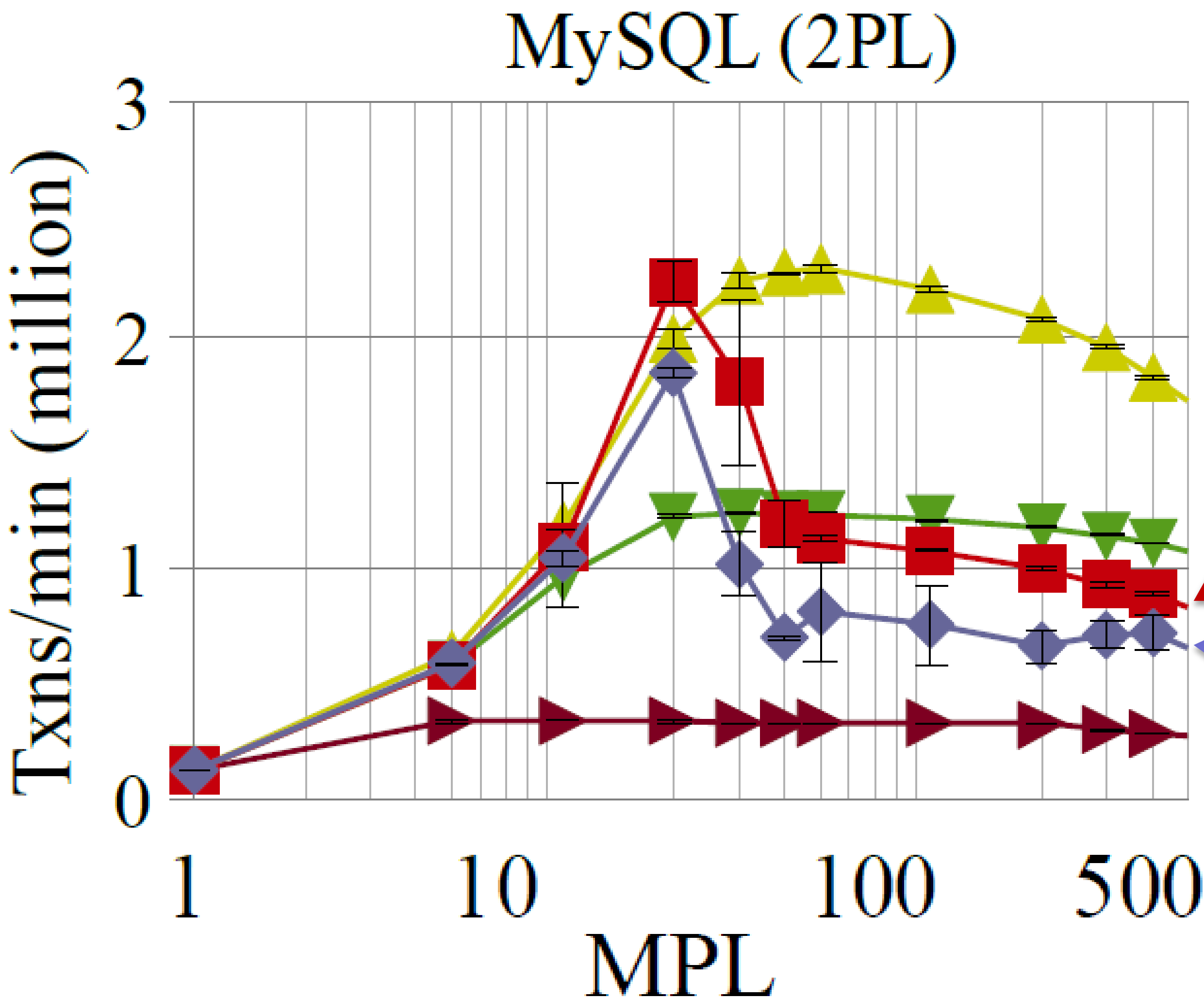
NICTA

Throughput goes as we expected !!



1 Socket

How bad is the performance collapse?



Performance collapses !!!

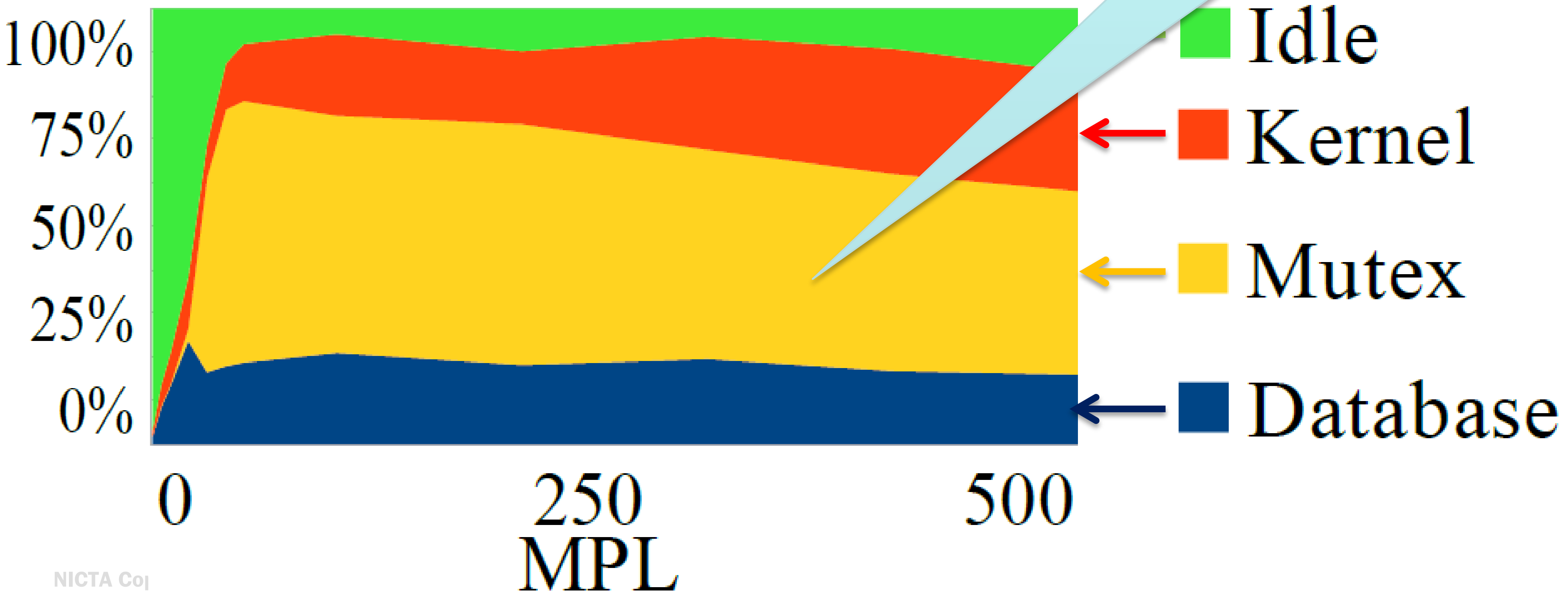
2 and 4
Sockets

What causes this collapse ?

Let's profile databases to peek a little bit deeper inside the system.

Profiling:
read-only queries under "SERIALIZABLE" isolation on 32 cores on 4 sockets

Latch contention is the cause !!!



Step back: why do we use latches ???

- Goal : mutual exclusion (ME) between threads
- Mutual Exclusion:
  - **(1) prevents data race errors**
  - **(2) synchronizes update made inside critical section.**
- Our intuition is:
  - **If we could achieve two objectives with an alternative paradigm, then we can avoid using latches.**

We propose

- a scalable lock manager with reduced latching.
- We achieved this by:
  - ***Read-After-Write (RAW)*** with memory barriers for fast synchronization
  - ***Staged allocation and de-allocation*** of locks for a lock hash table without dangling pointer dereferences

RAW-inspired Implementation (Acquire)

NICTA

Lock Acquire in Growing Phase

```
A1:  n_lock = lock_create();
A2:  n_lock->state = ACTIVE;
A3:  atomic_lock_insert(n_lock);
A4:  for all locks (lock) in hash_buc
A5:      if (lock is incompatible with n_lock)
A6:          n_lock->state = WAIT;
A7:          atomic_synchronize();
A8:          if (lock->state==OBSOLE
             n_lock->state=ACTIVE;          S3
             atomic_synchronize();
             continue;
A12:         if (new_deadlock()==TRUE)
A13:             abort Tx;
A14:         break;
A15:     end if
A16: end for

     if (n_lock->state == WAIT)
         mutex_enter(Tx->mutex);            S4
         atomic_synchronize();
A20:     if ( n_lock has to wait )
    :        Tx->state = WAIT;
    :        os_cond_wait(Tx->mutex);
    :    else
A24:         n_lock->state = ACTIVE;        S5
A25:         atomic_synchronize();
A26:     end if
A27:     mutex_exit(Tx->mutex);
A28: end if
```

<-Write
<-Barrier
<-Read

<-Write
<-Barrier
<-Read

Write->
Barrier->
Read->

RAW-inspired Implementation (Release)

Lock Release in Shrinking Phase

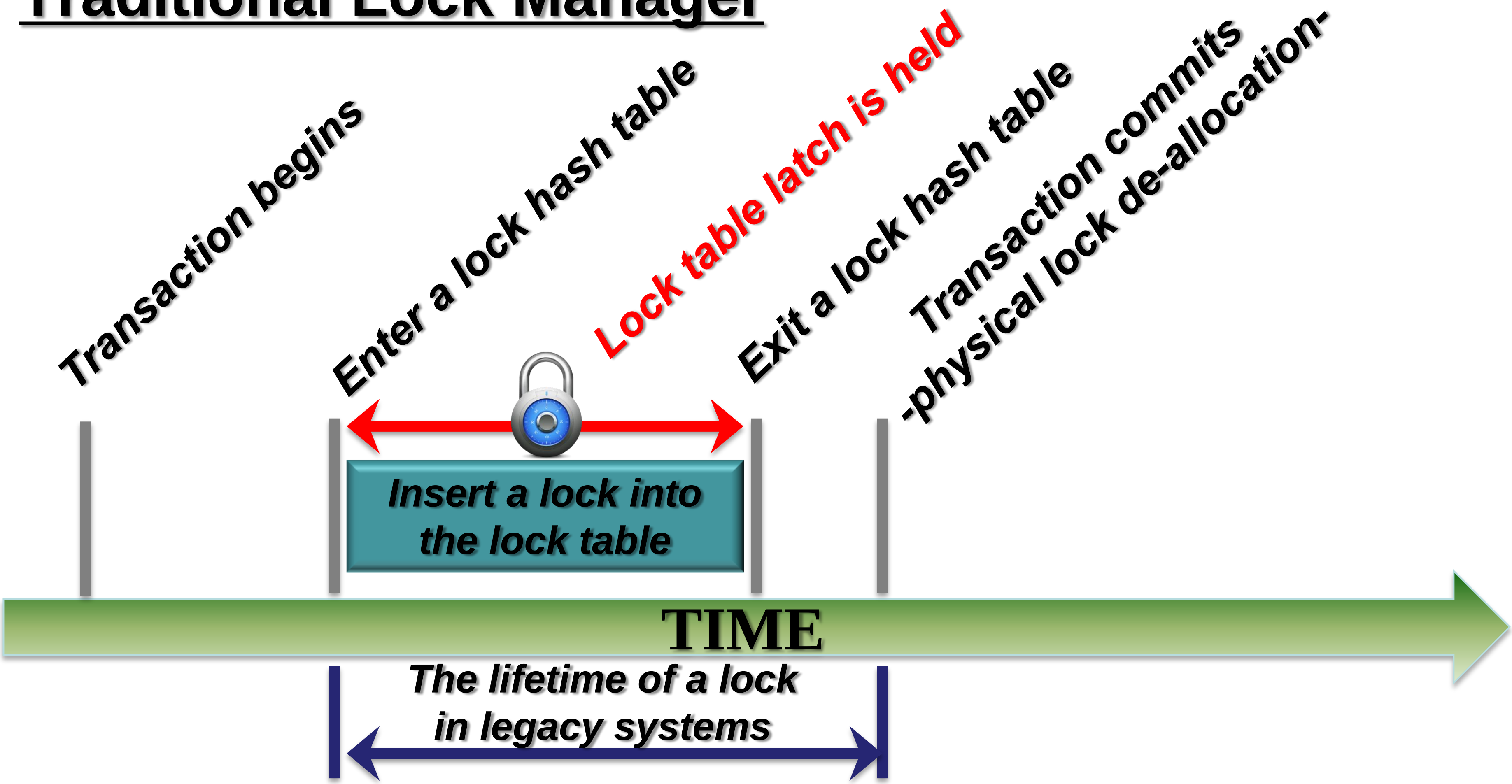
```
for all locks (lock1) in Tx
    lock1->state = OBSOLETE;                     S6
    atomic_synchronize();
    for all locks (lock2) that follow lock1
      mutex_enter(lock2->Tx->mutex);
      if ( lock2->Tx->state==WAIT &&
           lock2 does not have to wait )
             lock2->Tx->state=ACTIVE;
             lock2->state=ACTIVE;                S7
             atomic_synchronize();
             os_cond_signal(lock2->Tx);
      end if
      mutex_exit(lock2->Tx->mutex);
    end for
end for
```

<-Write
<-Barrier
<-Read

<-Write
<-Barrier
<-Read

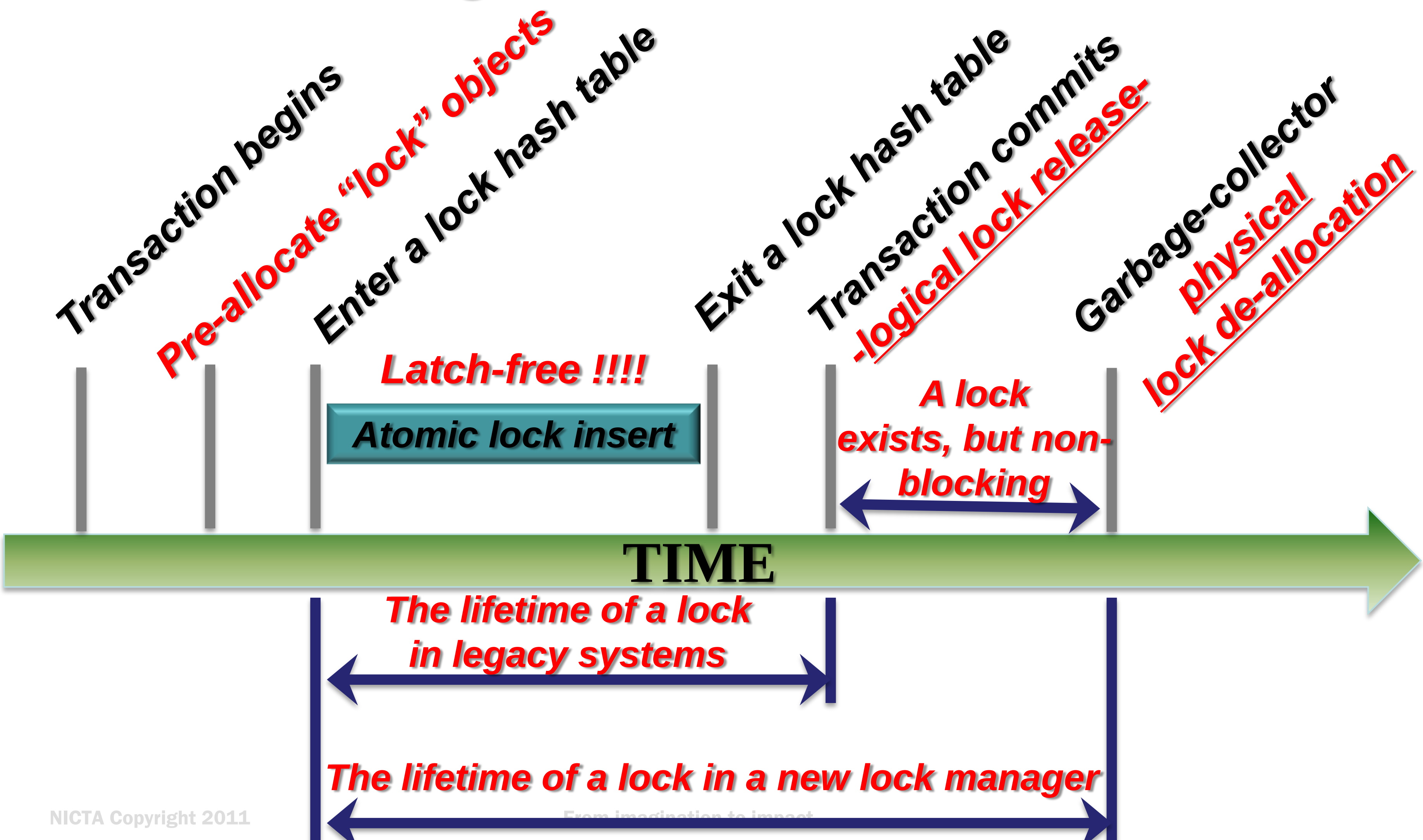
Staged allocation and de-allocation

Traditional Lock Manager



Staged allocation and de-allocation

New Lock Manager

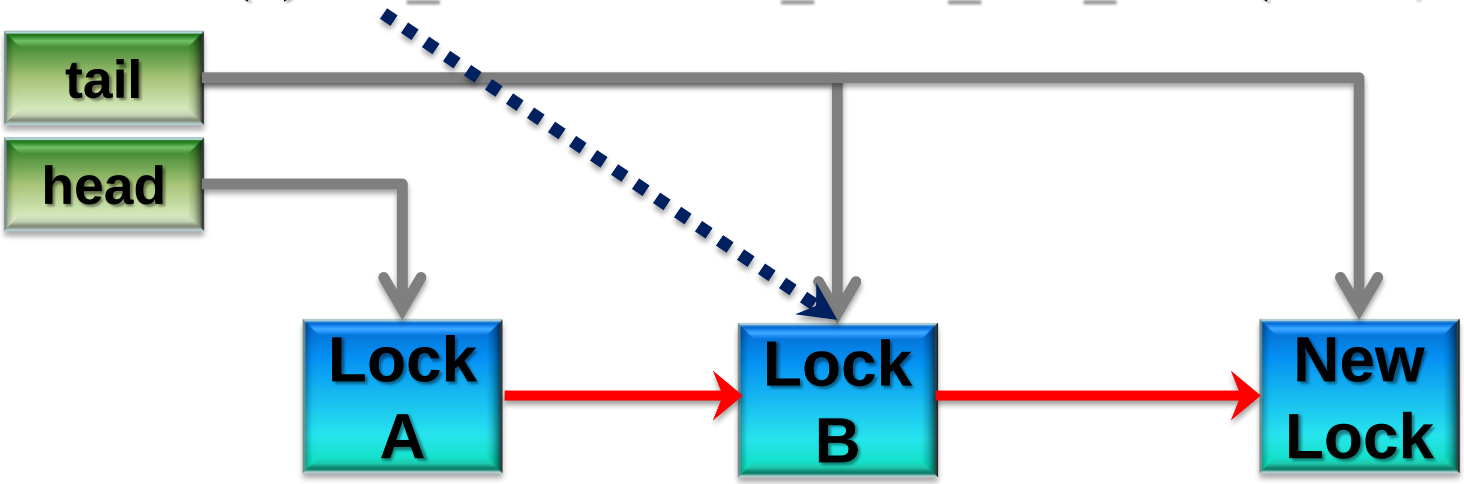


Two important operations

- ***Atomic lock insert***
  - Unique insert order must be ensured
- ***Garage-collection***
  - No dangling pointer dereference !!!

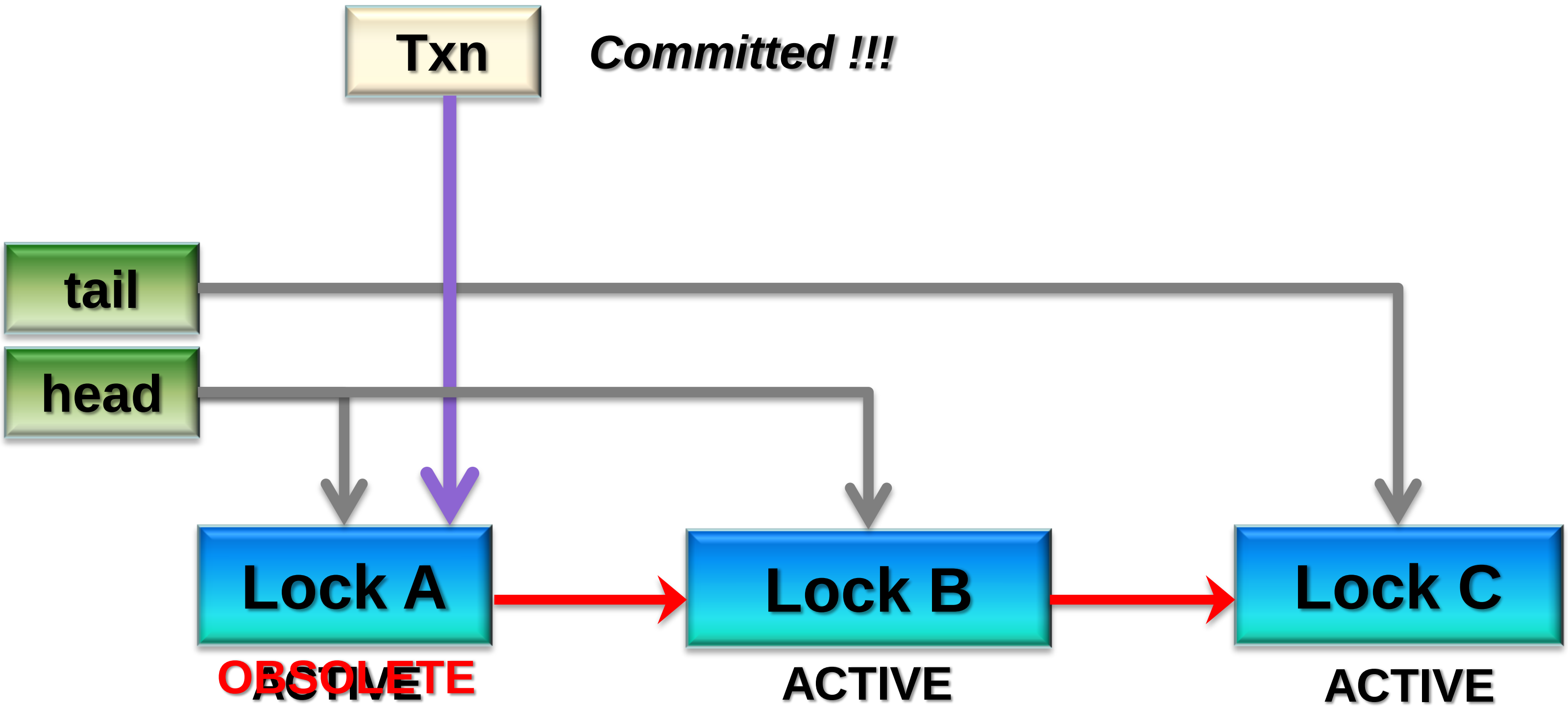
Atomic lock insert

(1) old_tail = atomic_fetch_and_store(&tail, NewLock)



(2) old_tail ->next = NewLock

Garage-collection



(1) Logical release by changing the state of a lock A

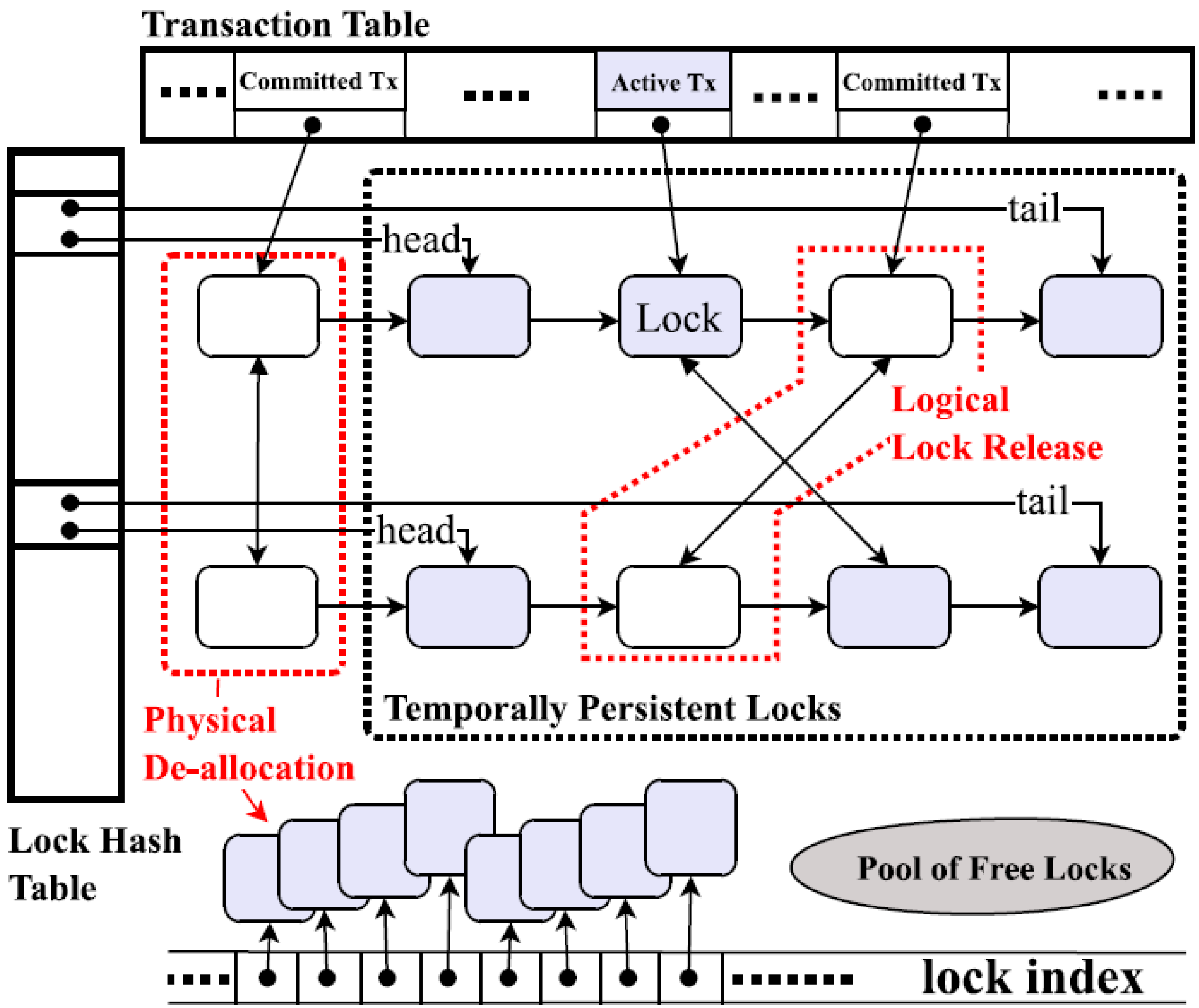
(2) Advance the head pointer

(3) Garbage-collect "OBSOLETE" locks

Correctness: transactions started after the head is advanced can NEVER see "Lock A" since it is INVISIBLE to him.

The Architecture of New Lock Manager

NICTA



Experimental Setup

NICTA

- Databases
  - MySQL-5.6.10, Our system (only the lock manager has been rewritten); also but not for comparison: Wisconsin Shore-MT and commercial DBMS X
- Micro-benchmark
  - Read-only

```
SELECT sum(b_int_value)*rand_number FROM txbench-i
 WHERE b_int_key > :id and b_int_key <= :id+S
```

  - Update

```
UPDATE txbench-((i+1)%3) SET b_value-k = rand_str
 WHERE b_int_key = :id1
       OR b_int_key = :id2
```

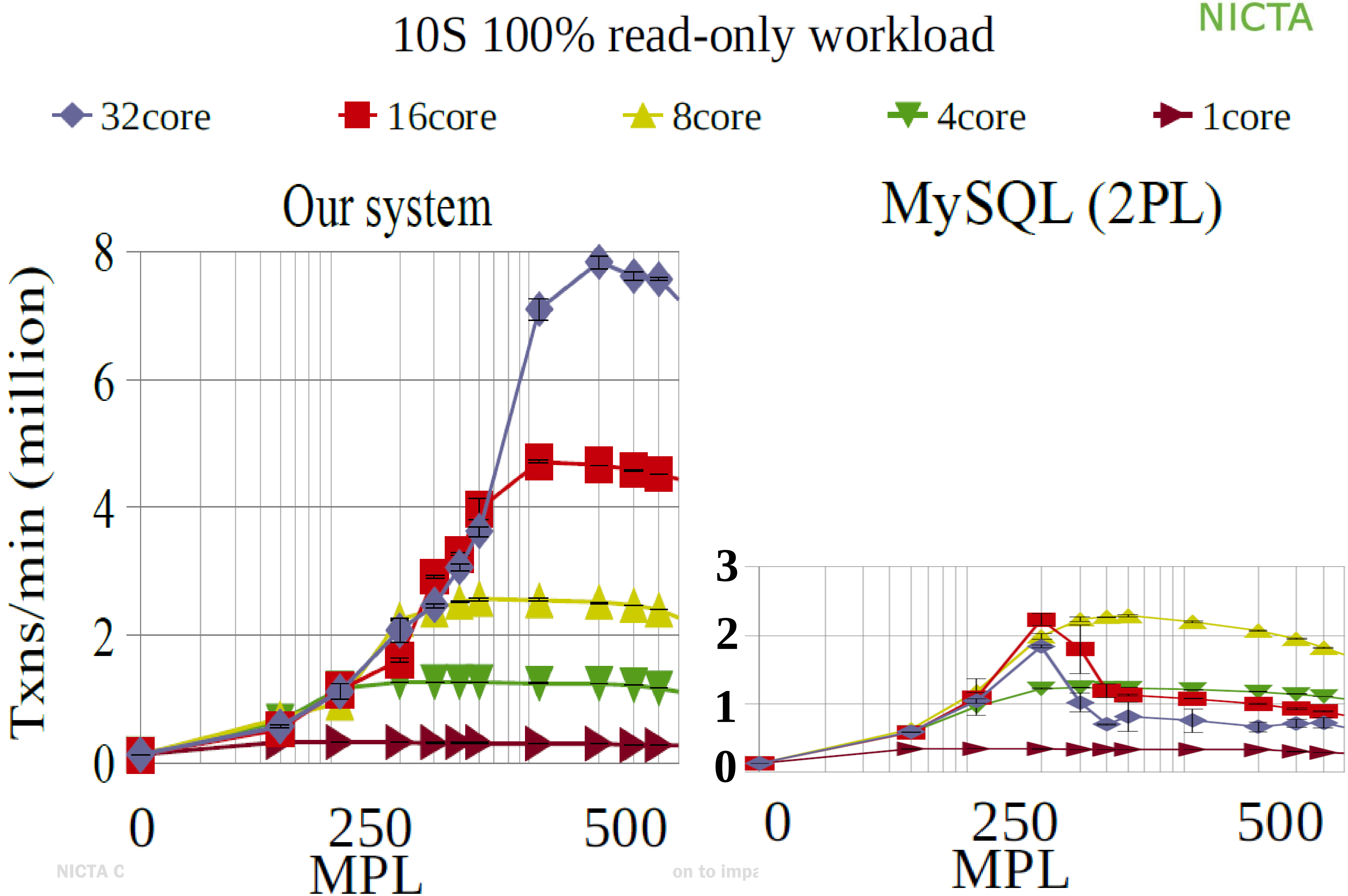
Experimental Setup (cont.)

- Multicore machines

Component	Specification
Processors	8-Core Intel Xeon CPU E7-8837
Processor Sockets	4 Sockets
Hardware Threads	32 (No HyperThreading Support)
Clock Speed	2.66 GHz
L1 D-Cache	32 KiB (per core)
L1 I-Cache	32 KiB (per core)
L2 Cache	256 KiB (per core)
L3 Cache	24 MiB (per socket)
Memory	128 GiB DDR3 1066 MHz
Network	Ethernet 1 Gbps

- Isolation : "SERIALIZABLE"

Performance Evaluation (throughput)

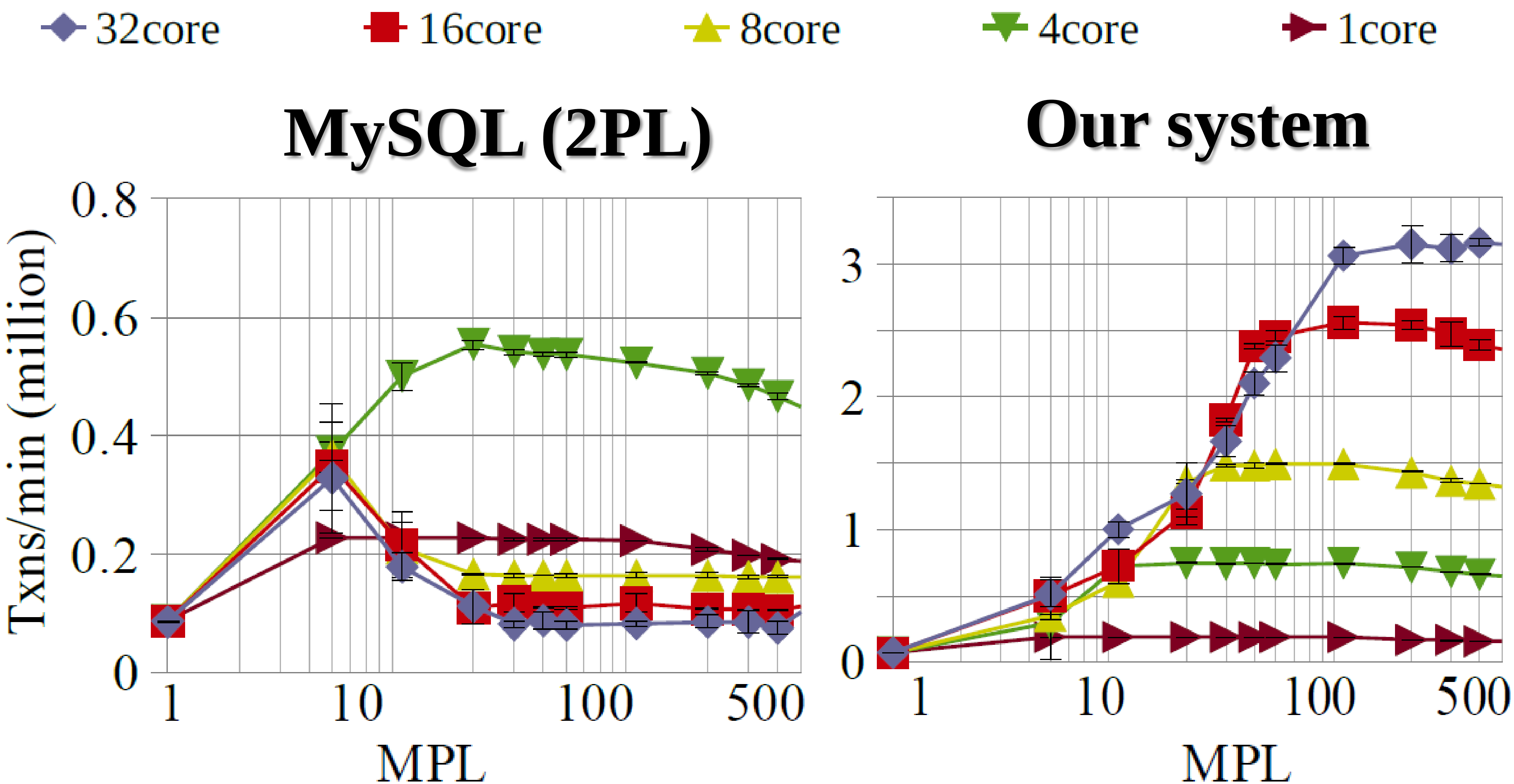


 on to impa

Performance Evaluation (throughput)

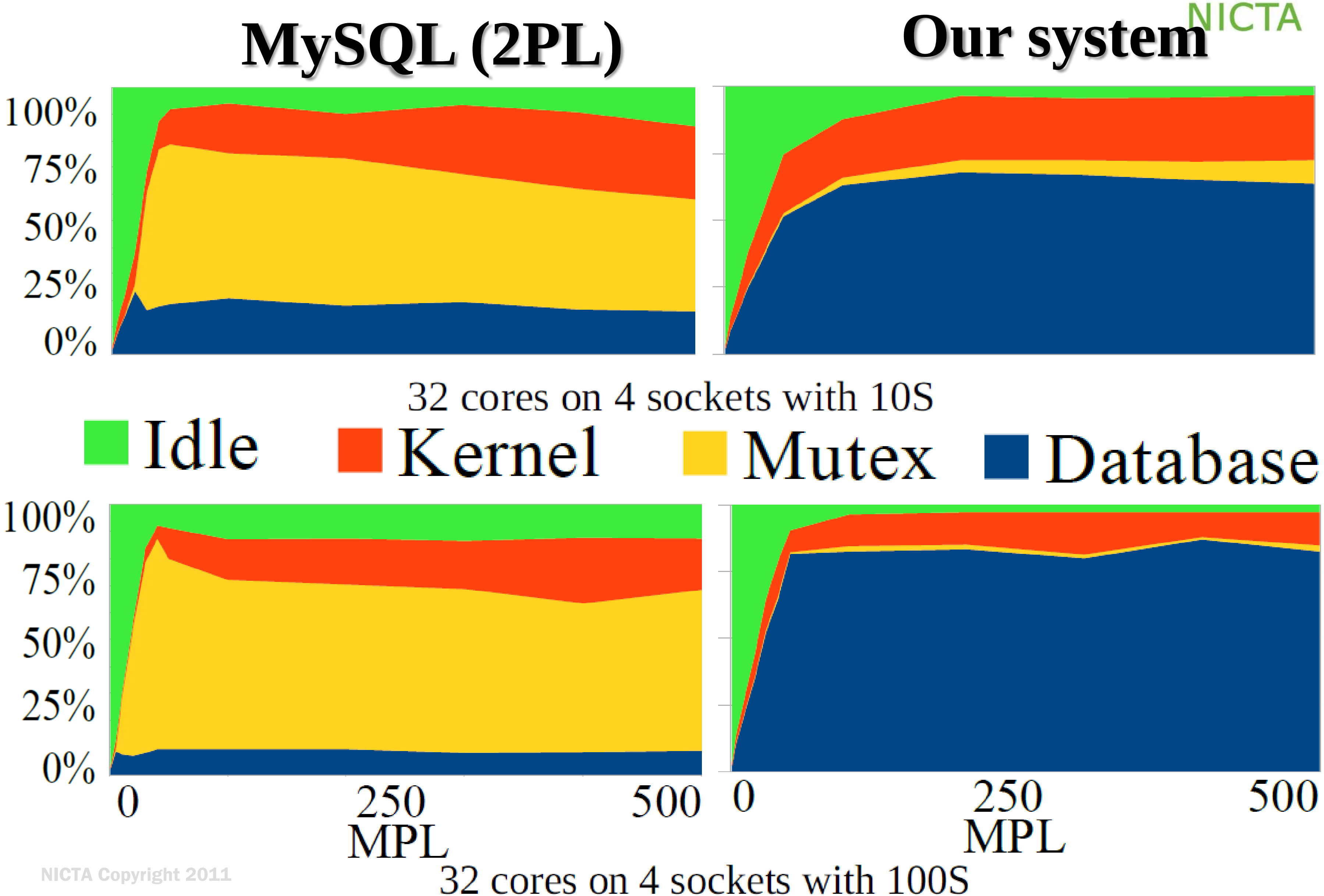
NICTA

100S 100% read-only workload



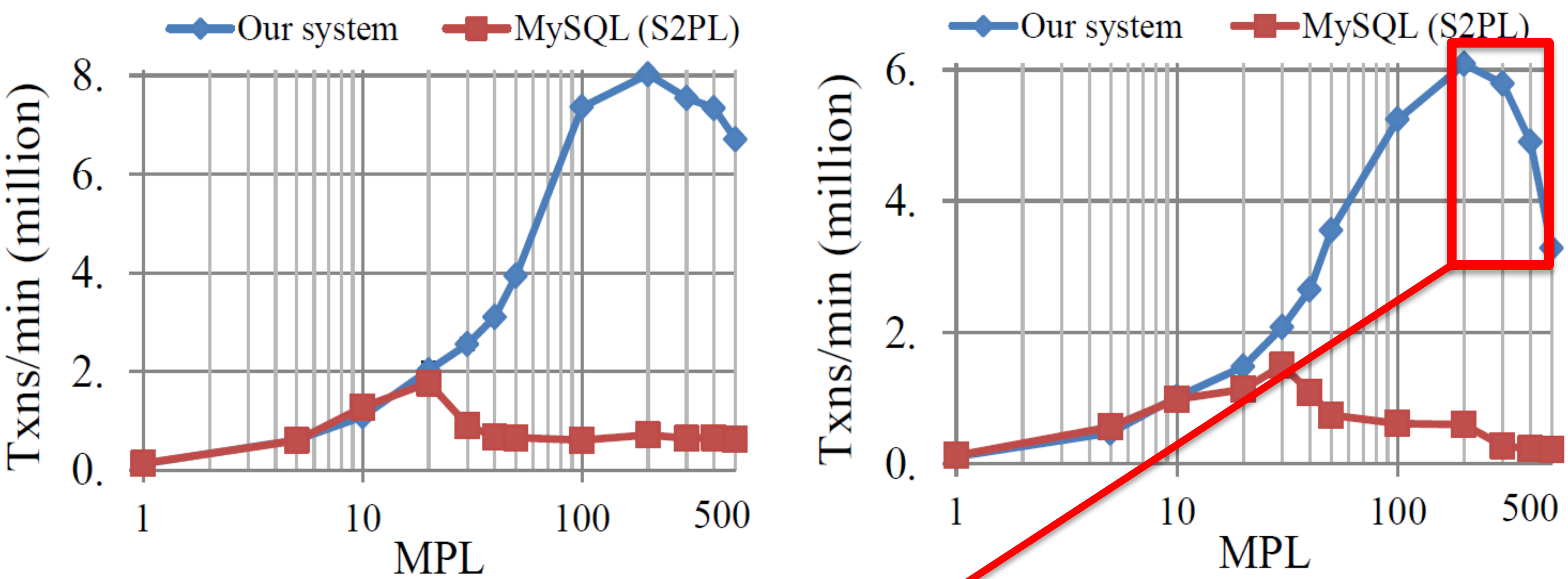
Note Y-axes differ

Performance Evaluation (profiled)



Performance Evaluation (update & hotspot)

NICTA



(a) Throughput, R/O workload (b) Throughput, 20% updates, hotspot

Degradation is due to lock conflicts, not latch contention.

Conclusion

- We identified that latch contention in the lock manager is a major cause for the performance collapse problems in multicore environments.
- We proposed a scalable lock manager with reduced latching, and demonstrated the performance.

Thank You & Questions?