Introduction to Multicore Scalability Problems
-A Scalable Lock Manager for Multicores-

Hyungsoo Jung

This talk will discuss …

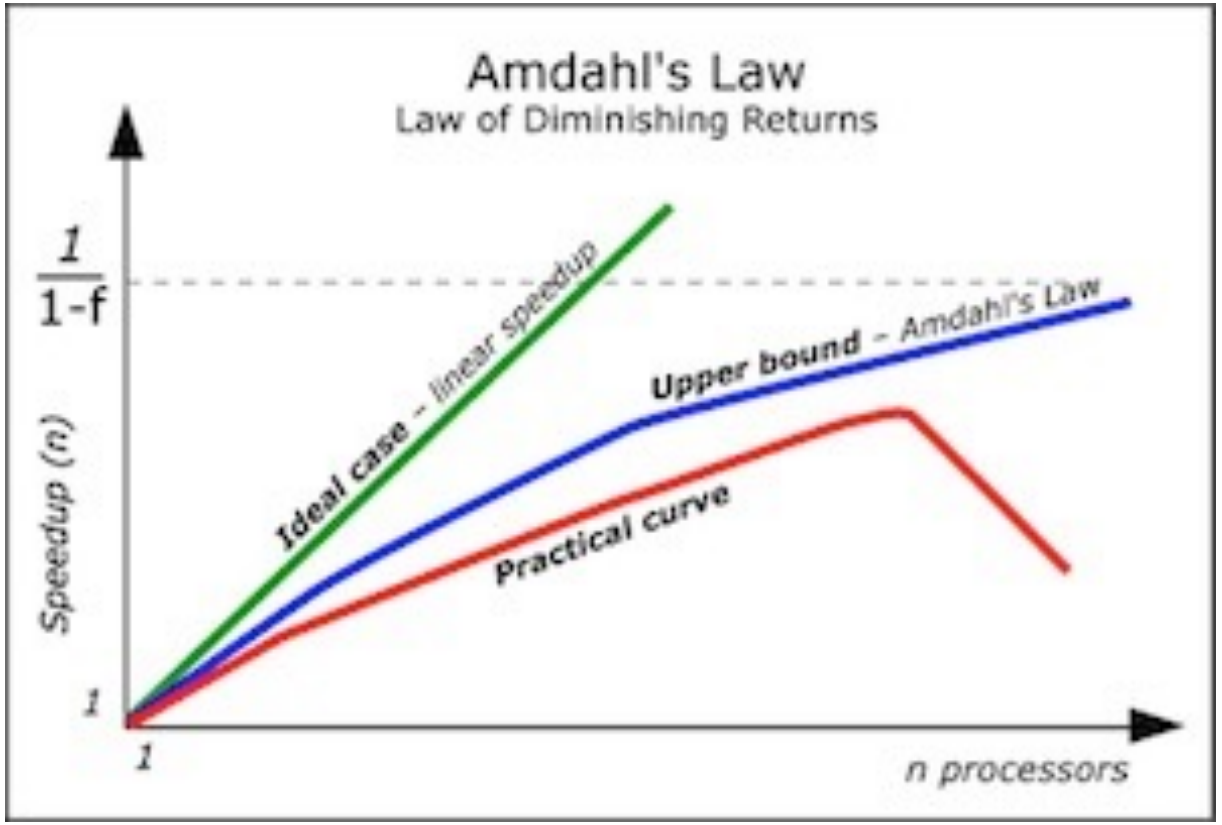
- **A Scalable Lock Manager For Multicores**

- **Agenda**
  - **Hardware Development Trend**
  - **Scalability Problems in Database Systems**
  - **Future Research Direction**

Why is scalability so important ?

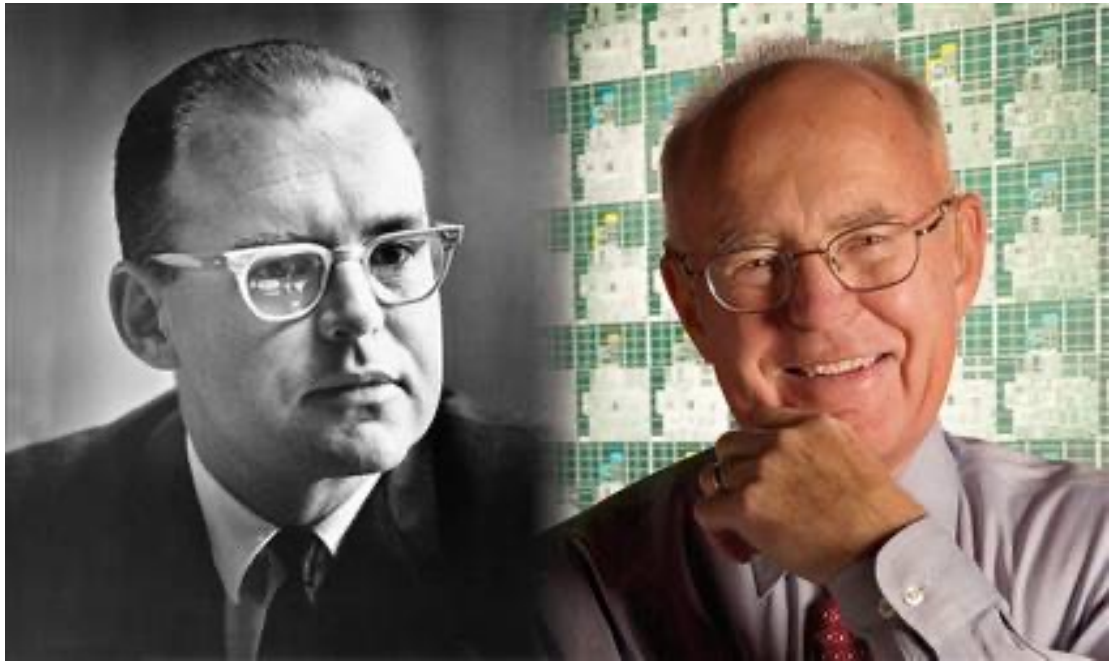
- See Amdahl's Law

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$



Hardware Development Trend

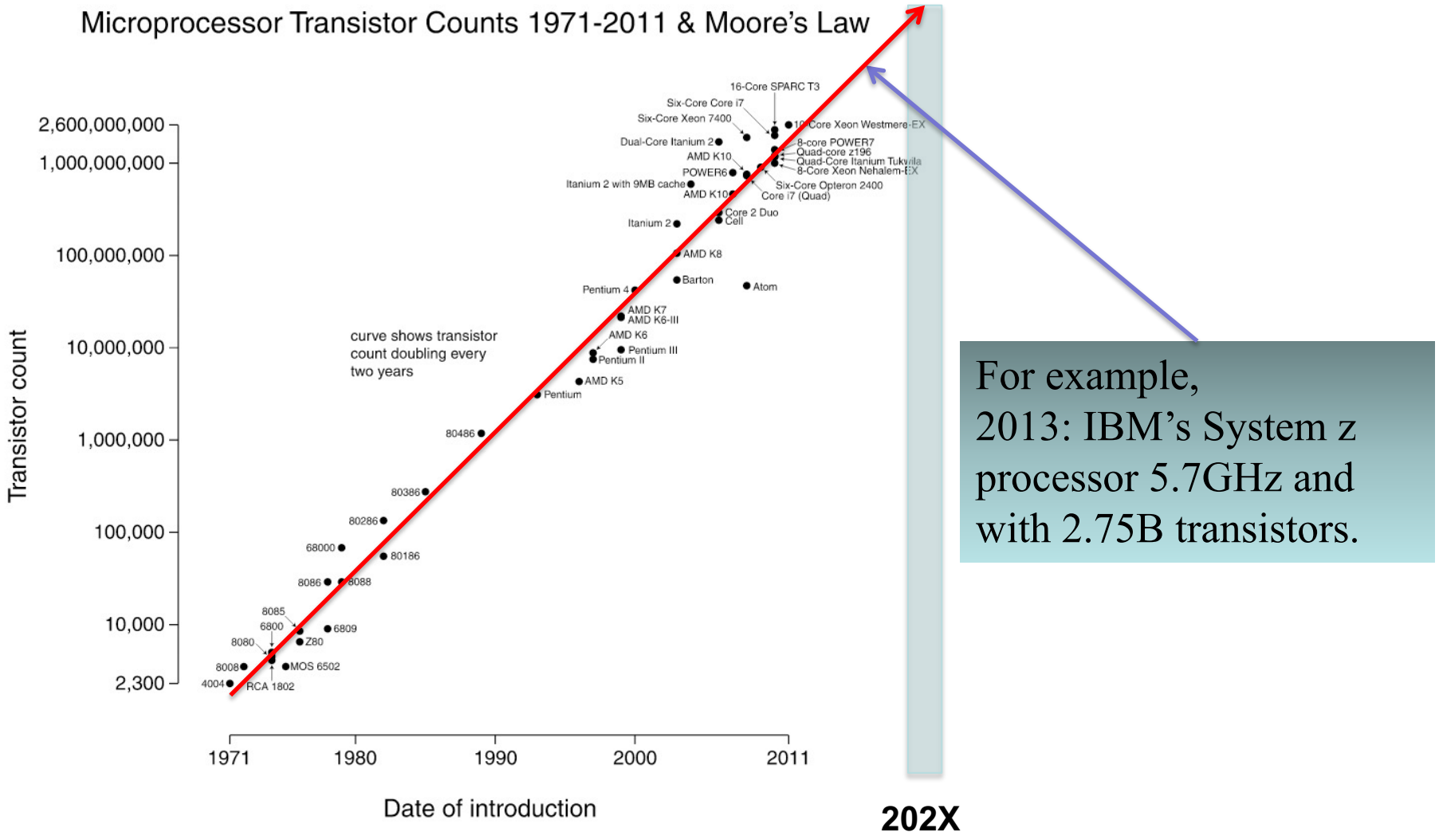
Moore's Law (processor cores)



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

--Gordon Moore, Intel co-founder

Moore's Law (processor cores)



For example,
2013: IBM's System z processor 5.7GHz and with 2.75B transistors.

202X

Trend Summary

		Annualized Growth Rate	Compound Growth Over 10 Years
Nielsen's Law	Internet bandwidth	**50%**	57 ×
Moore's Law	Computer power	**60%**	100 ×

Scalability Problems in Database Systems

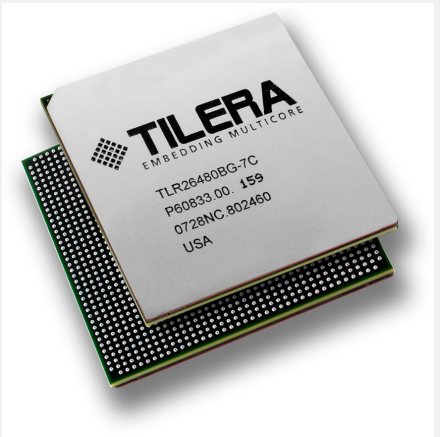
Moore’s Law goes Multicores

But what about the software?

Database Management Systems: this is the focus !!!!

Enterprise Software Systems (Not explored completely)

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



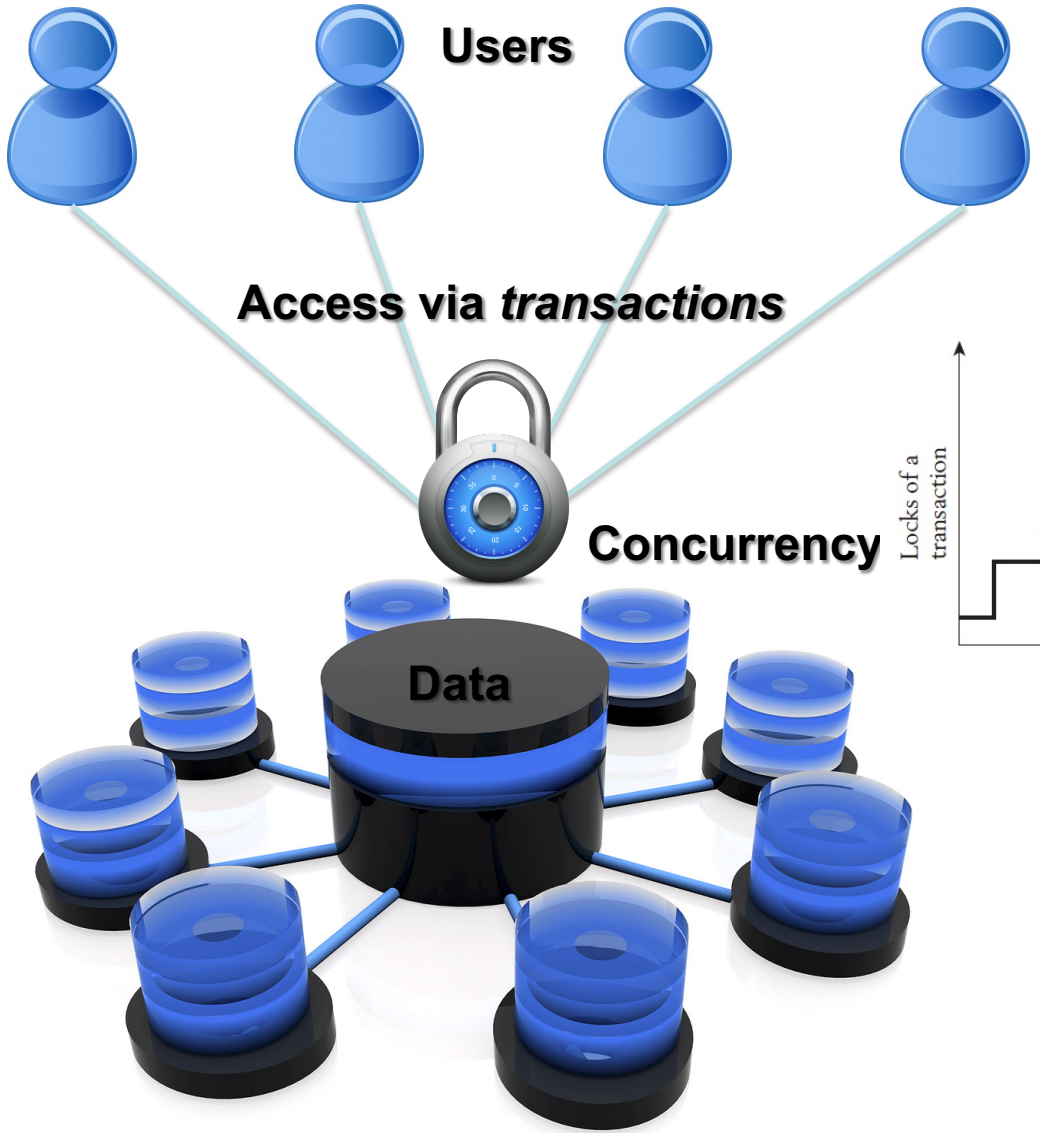
MULTICORE MACHINES

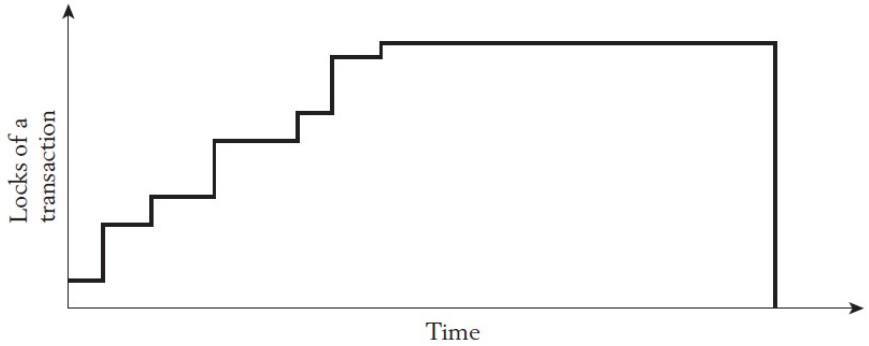
The main goal is to solve ..

- Multi-core scalability problems of DBMS by eliminating **scalability bottleneck**, especially in **the 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.

Lock vs. Latch : Database Lock



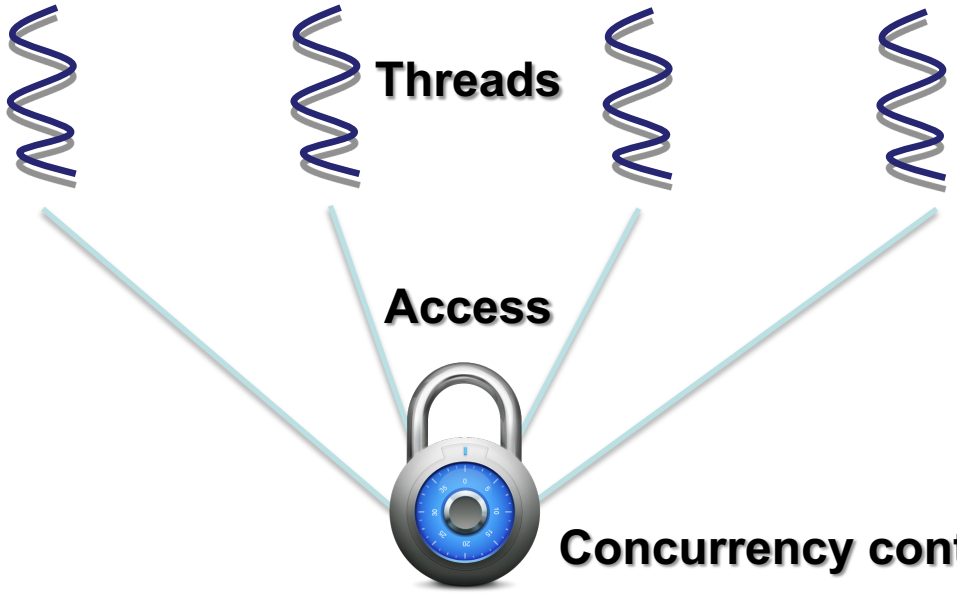


Duration is long (S2PL)

Sleeping when locks conflict

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

Lock vs. Latch : Latch



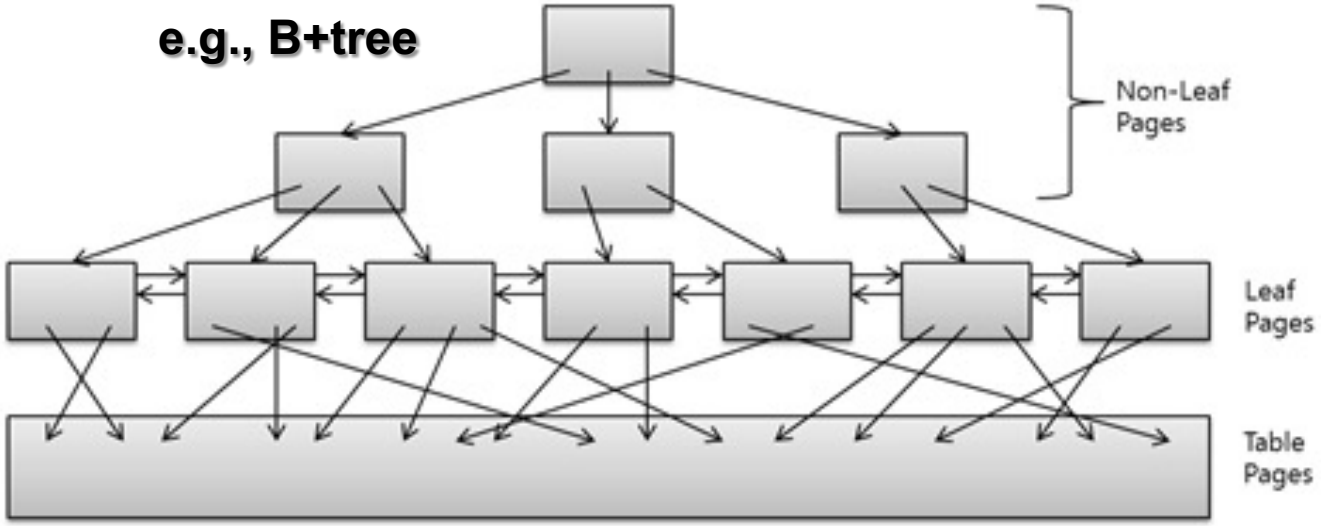
Duration is usually very short

Spin-waiting on contention

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

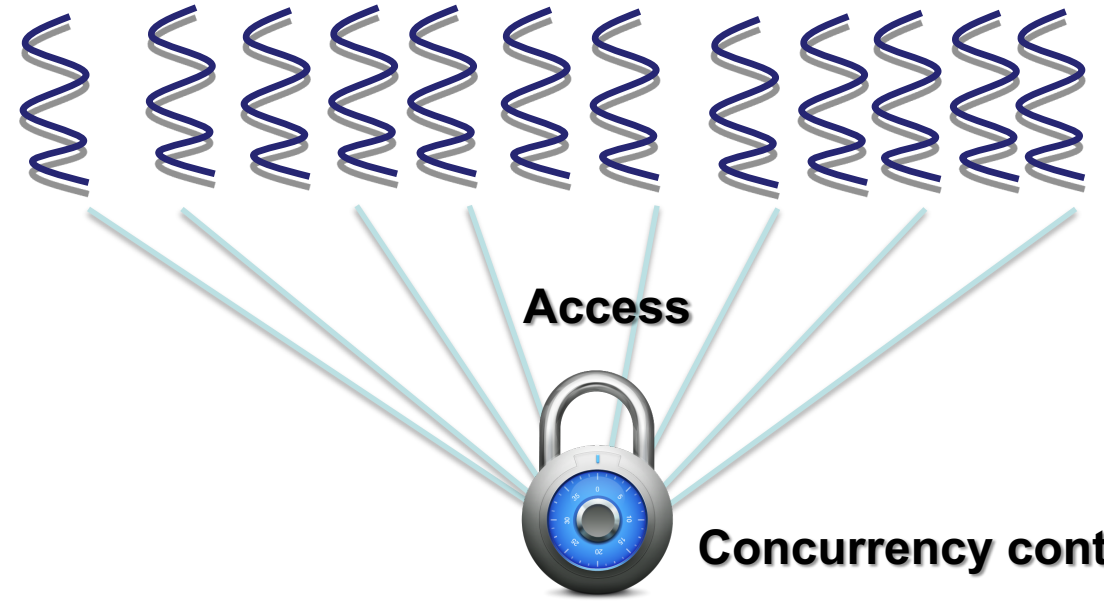
Concurrency control by latches

e.g., B+tree



In-memory Data Structures

Lock vs. Latch : High latch contention

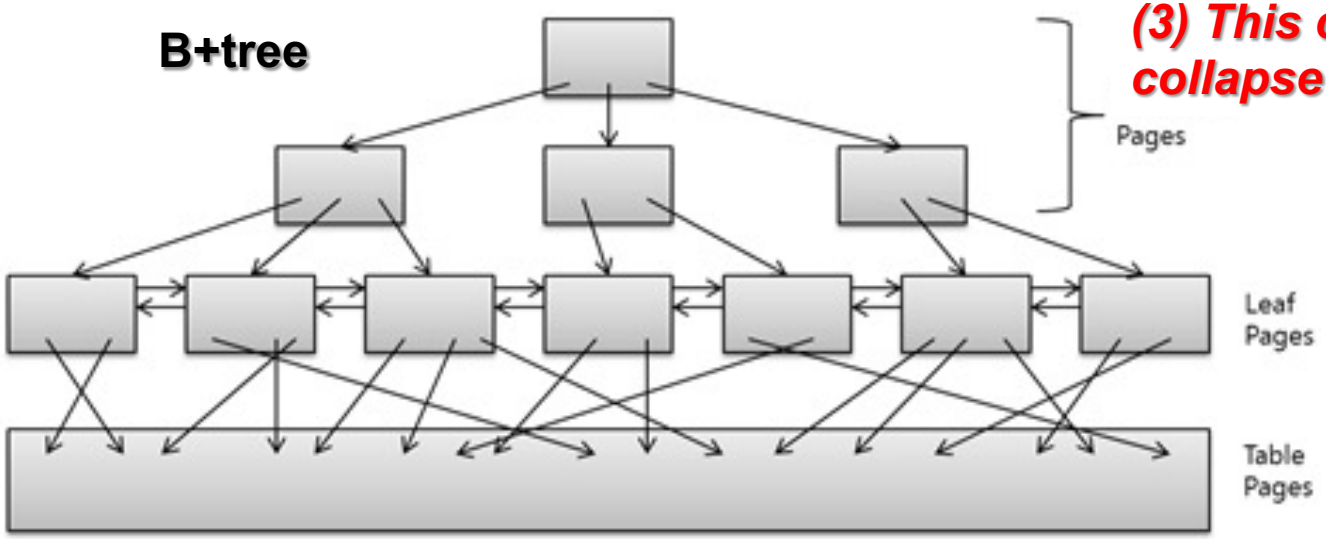


In high contention :

(1) latch duration gets longer

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

(3) This causes performance collapse !!!!



In-memory Data Structures

Latch protecting **Lock** table (MySQL-5.6)

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
```

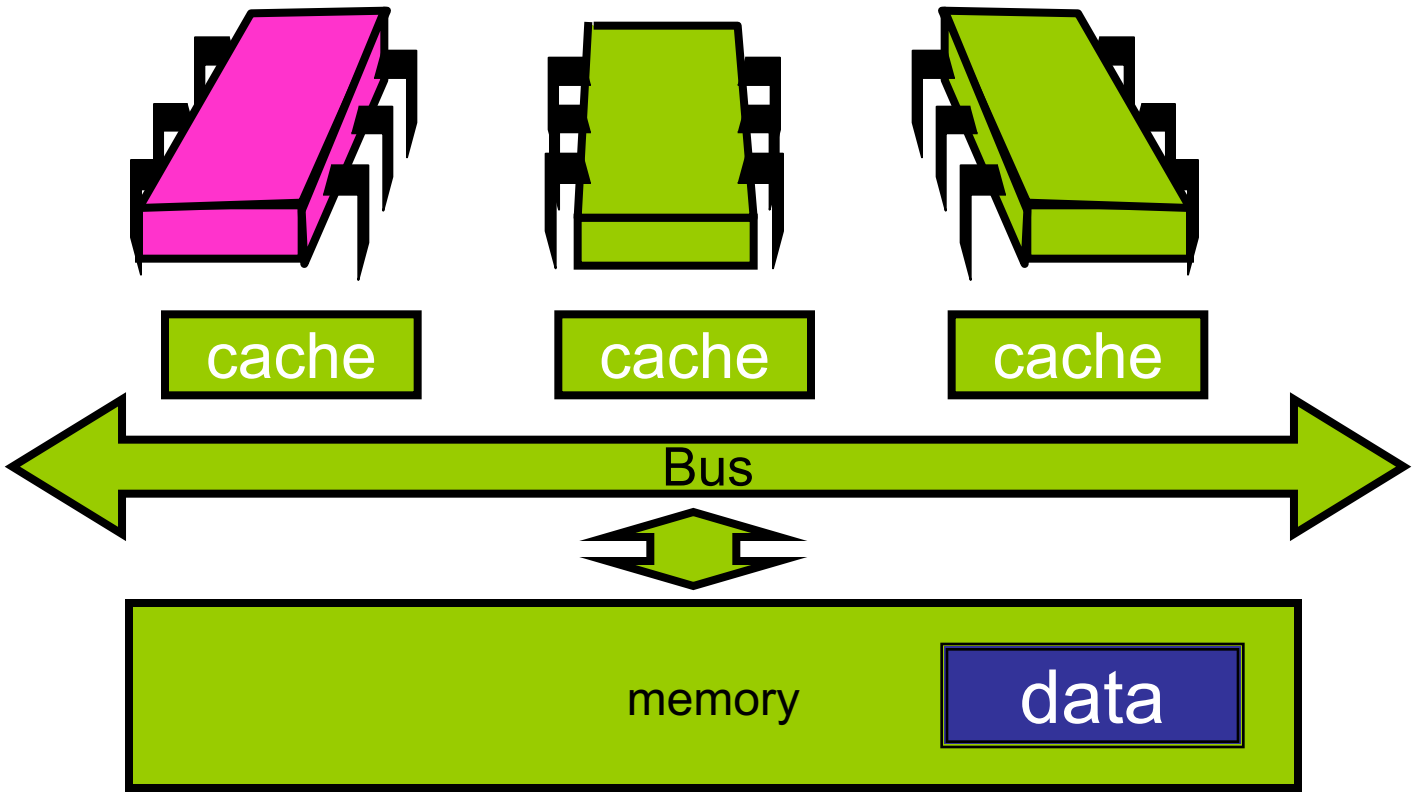
Lock Release in Shrinking Phase

```
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);
```

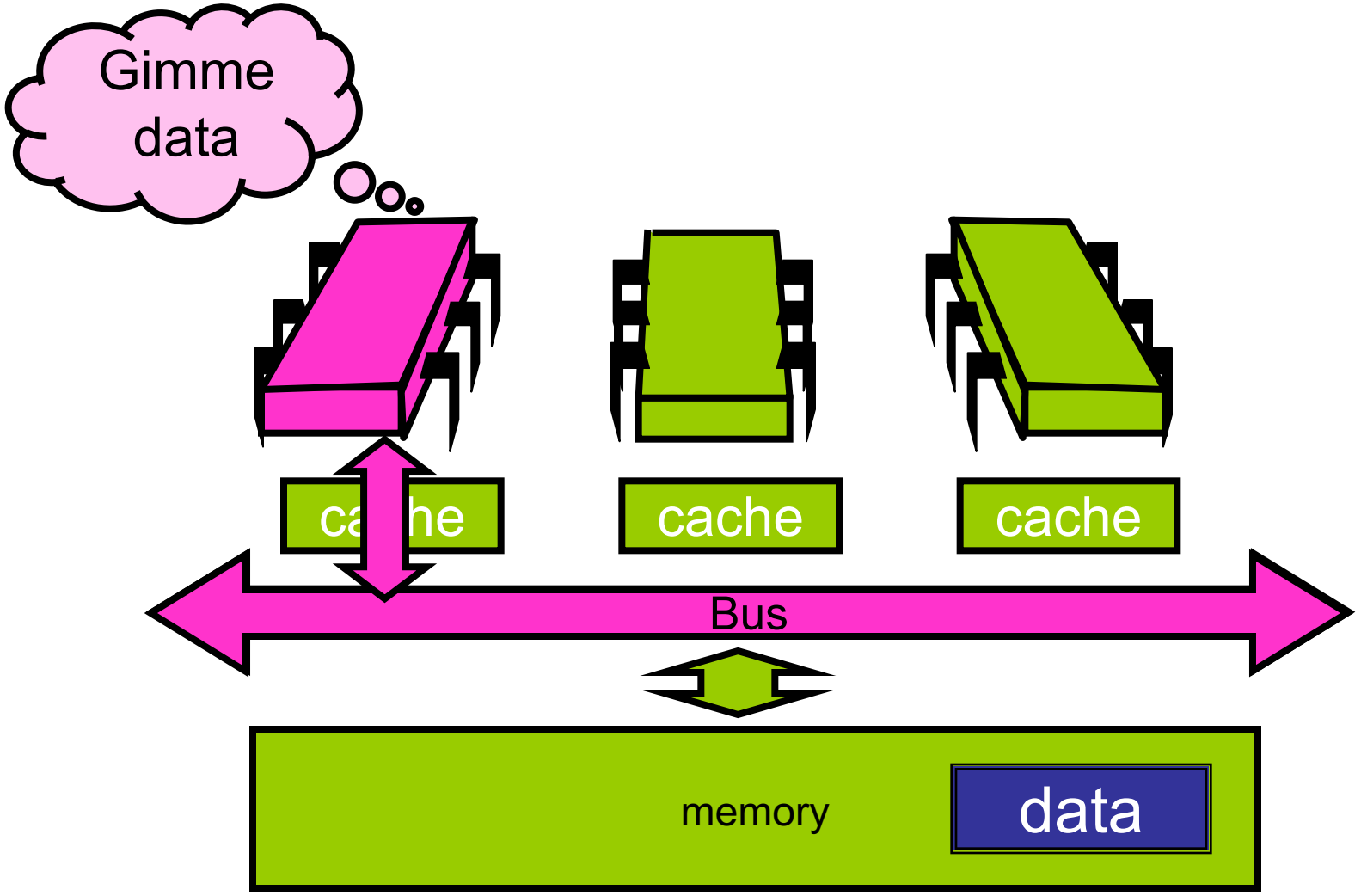
Lock Table Mutex (or Latch)

Cache Invalidation Storm

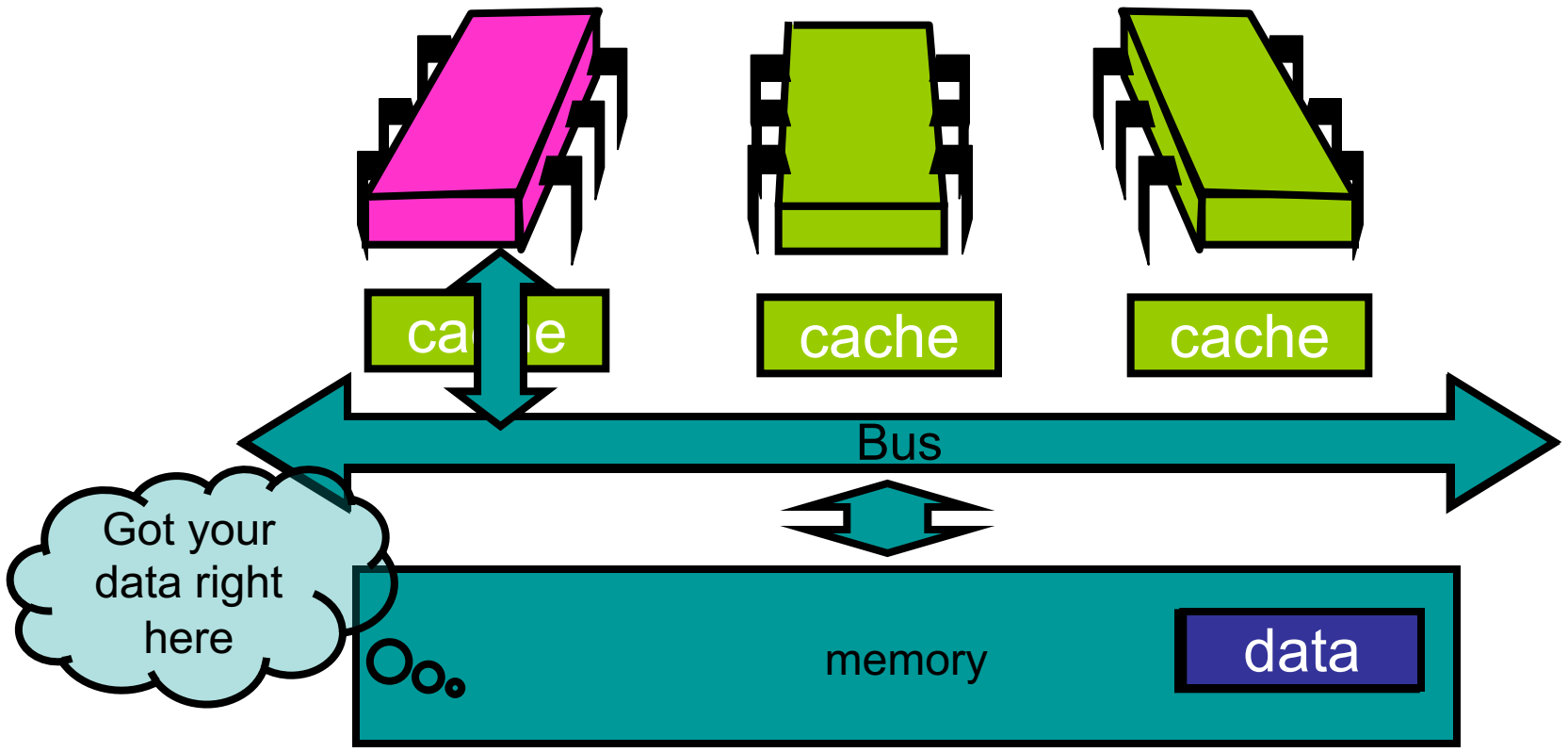
Processor Issues Load Request



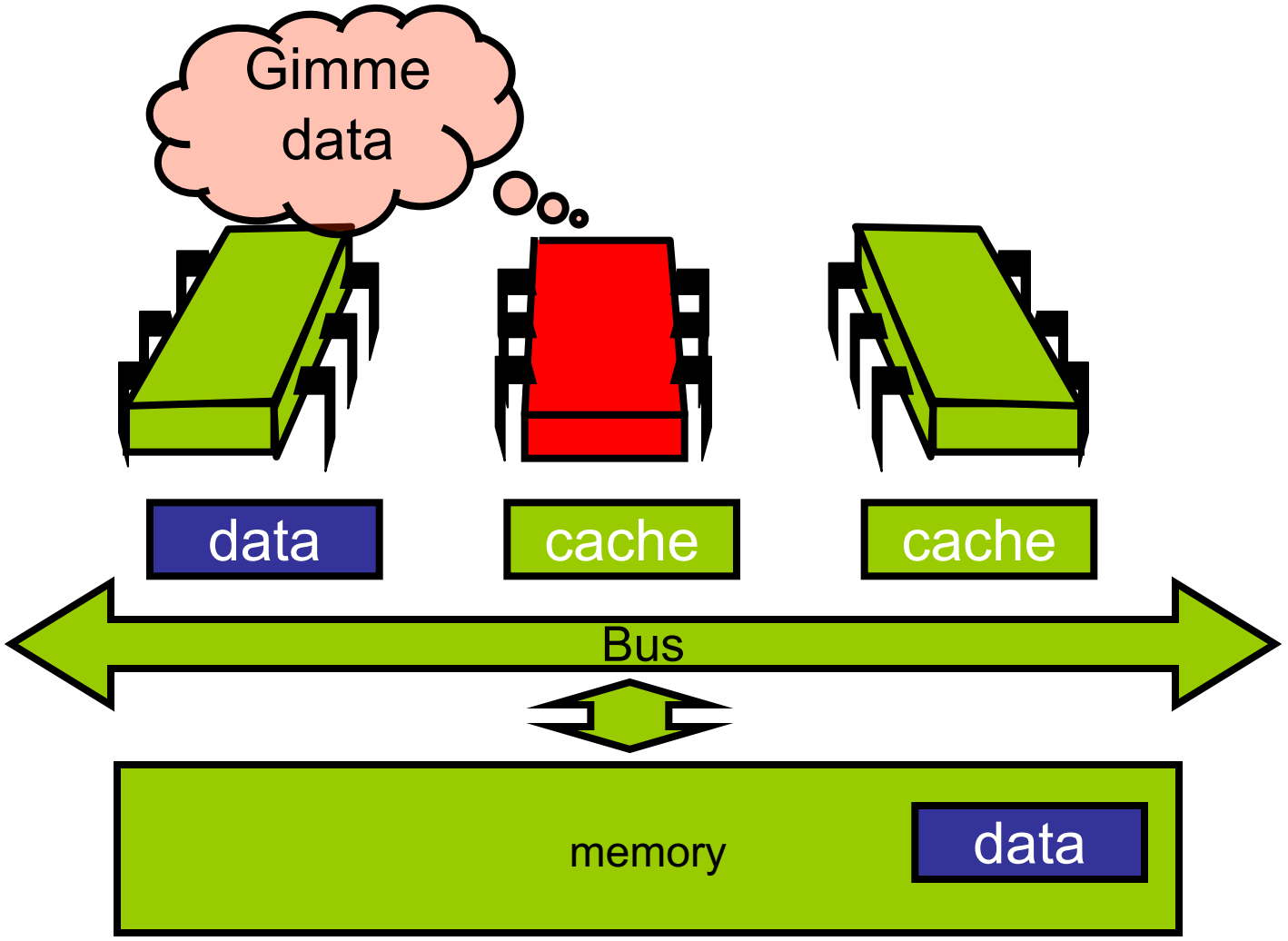
Processor Issues Load Request



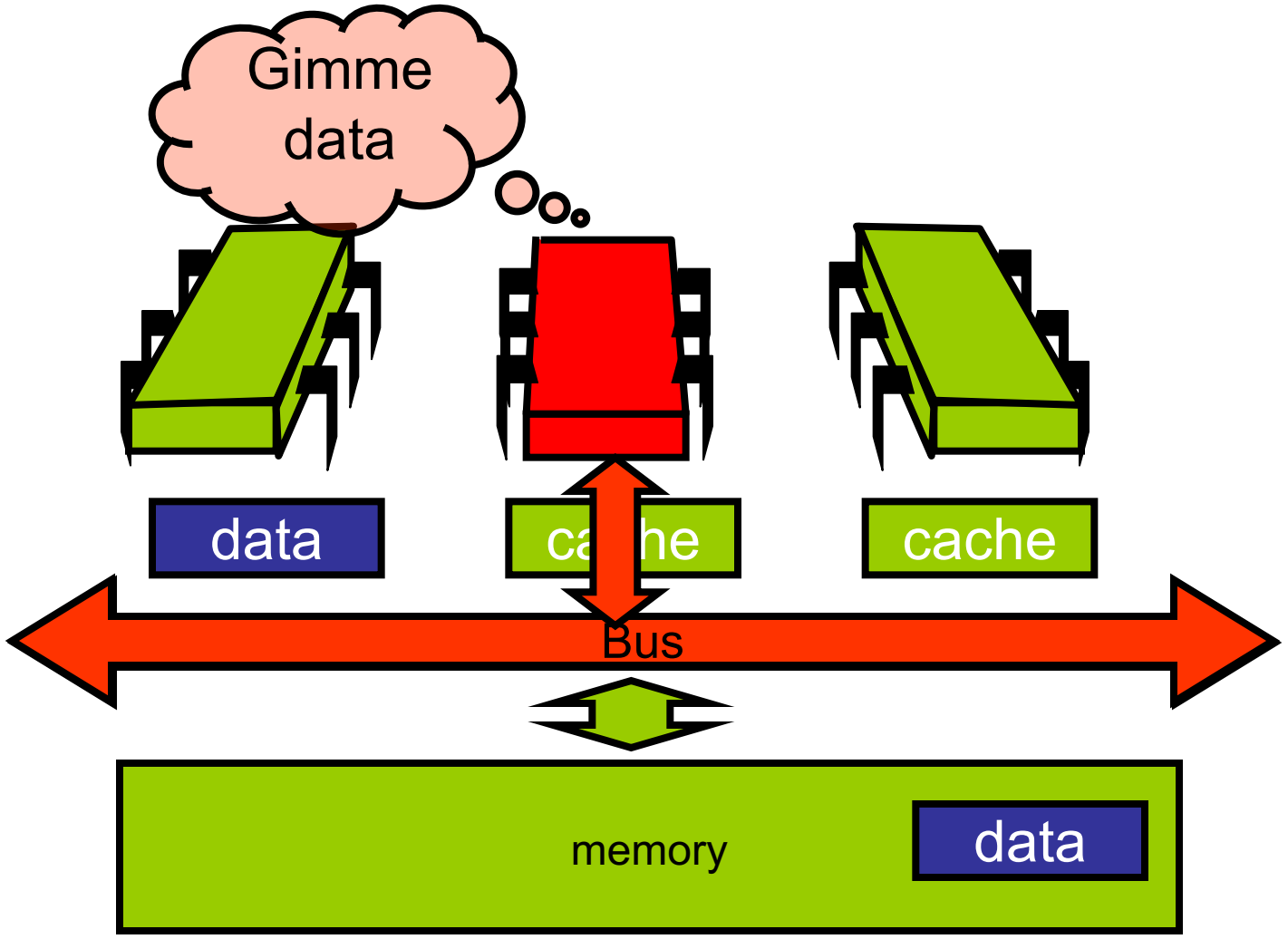
Memory Responds



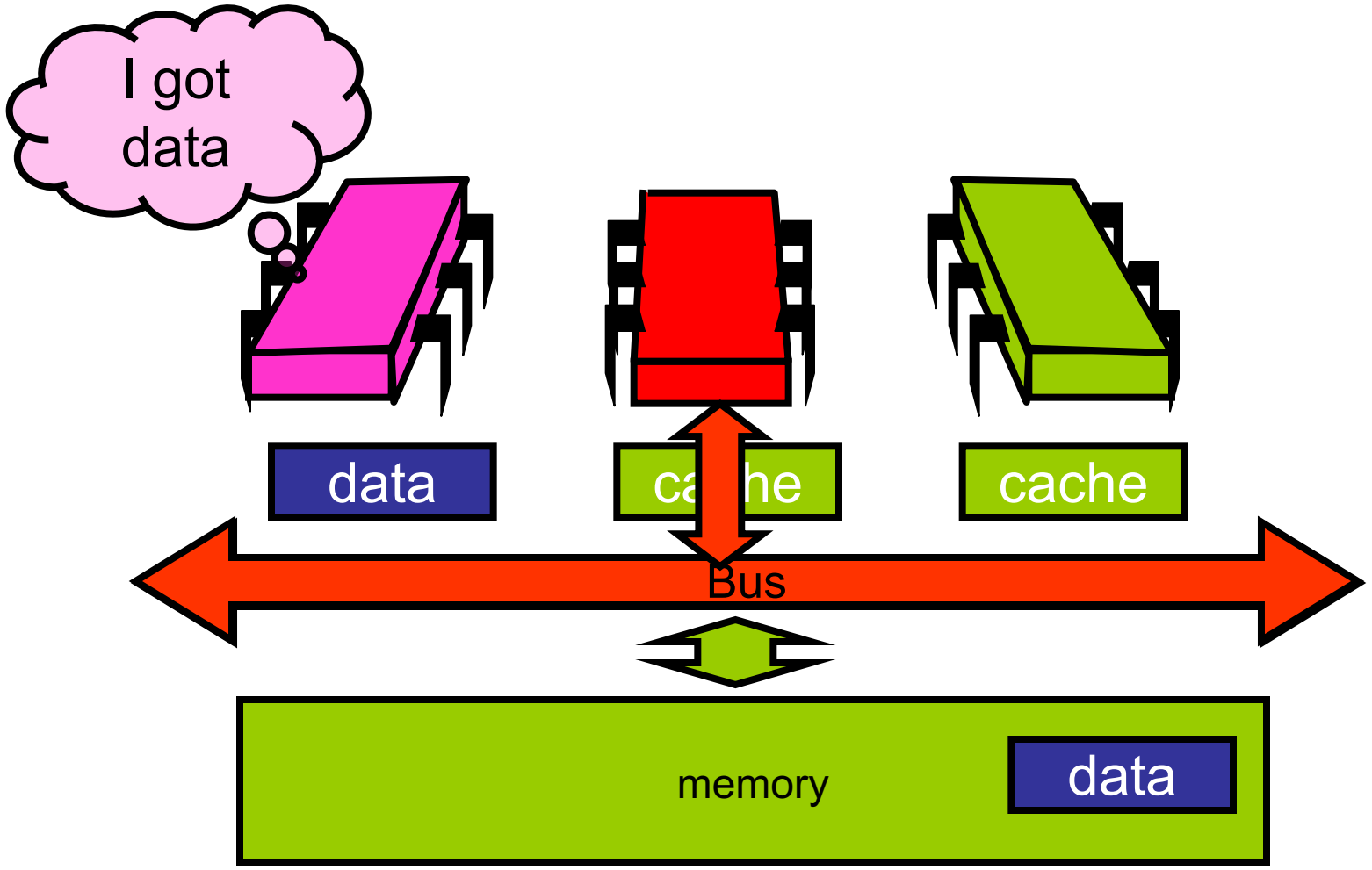
Processor Issues Load Request



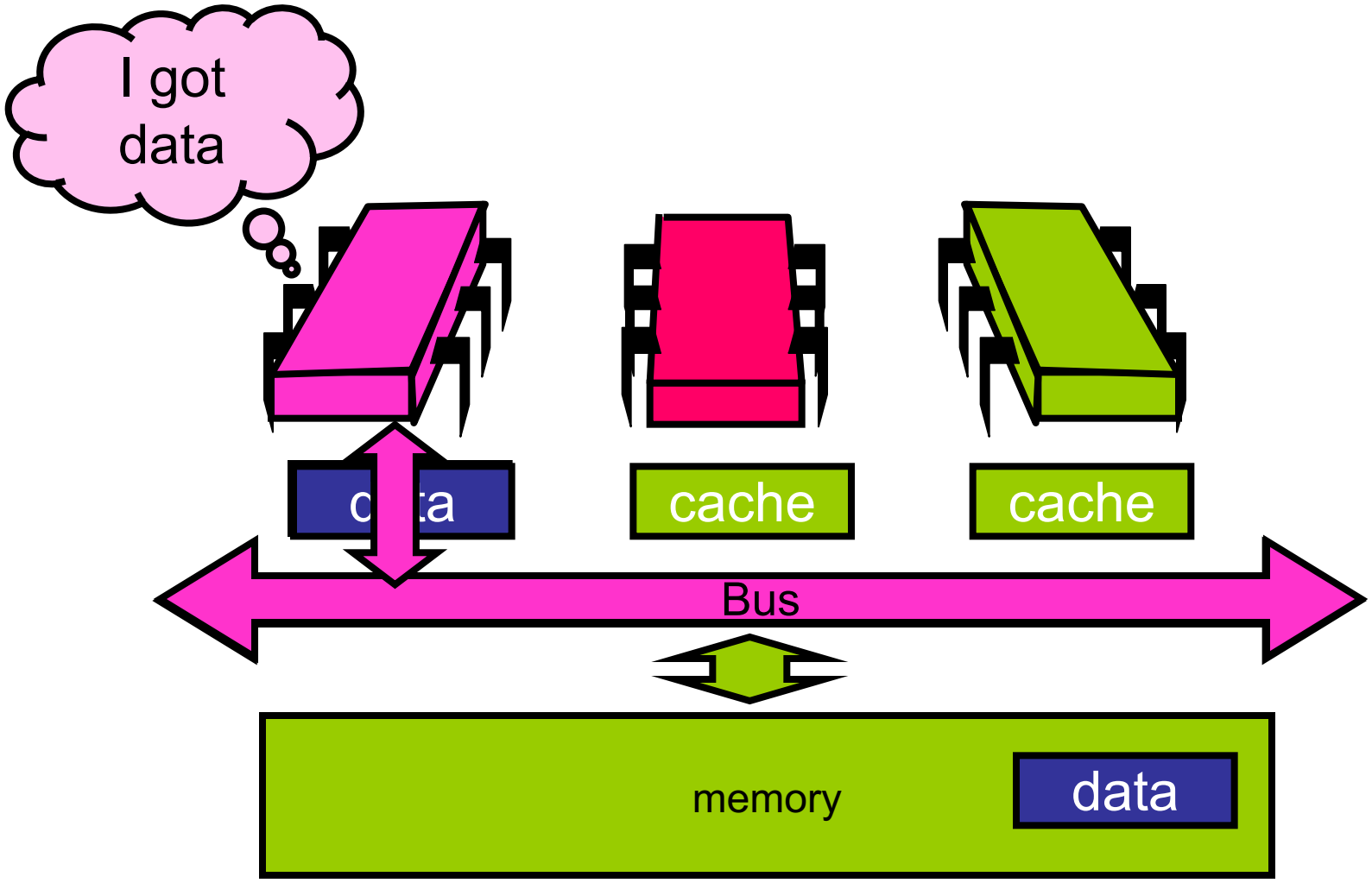
Processor Issues Load Request



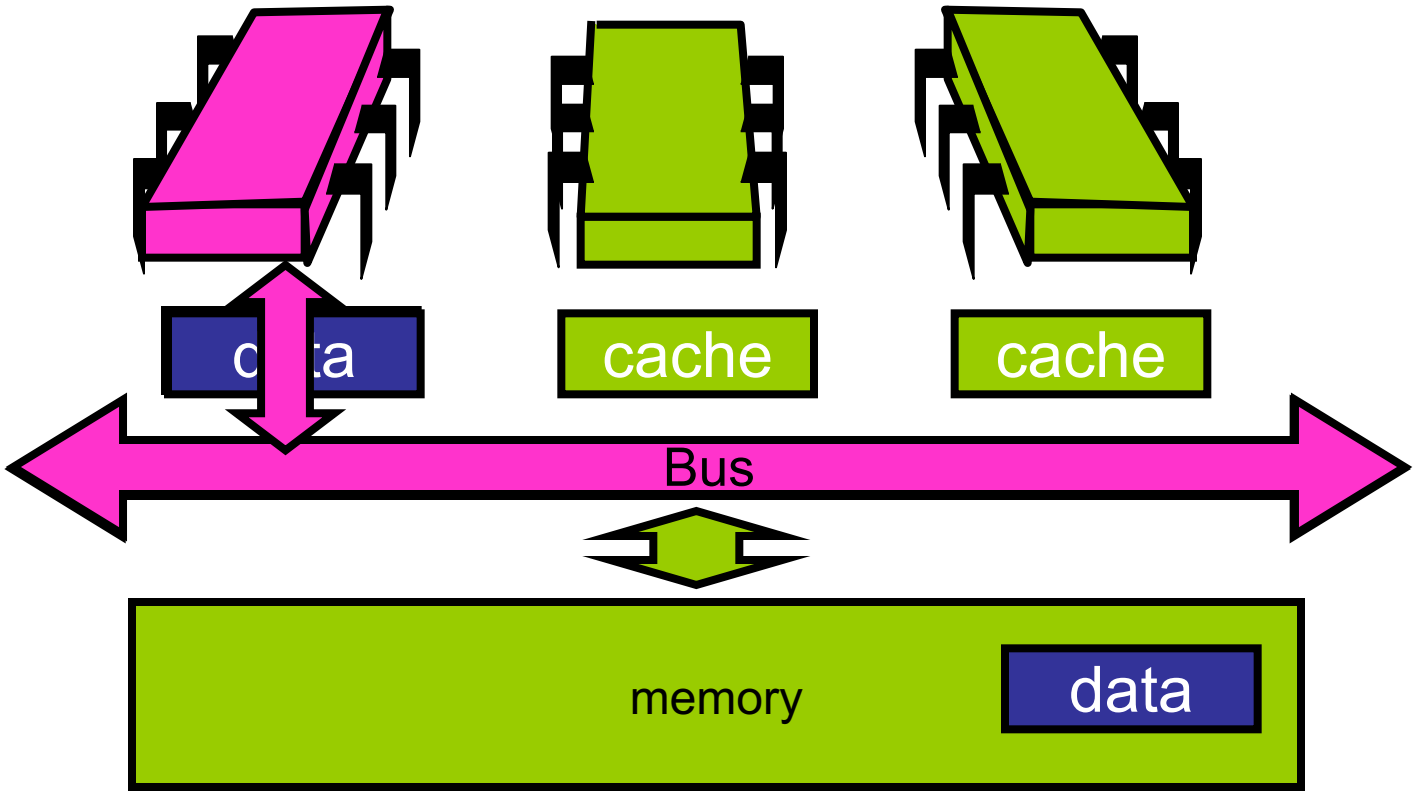
Processor Issues Load Request



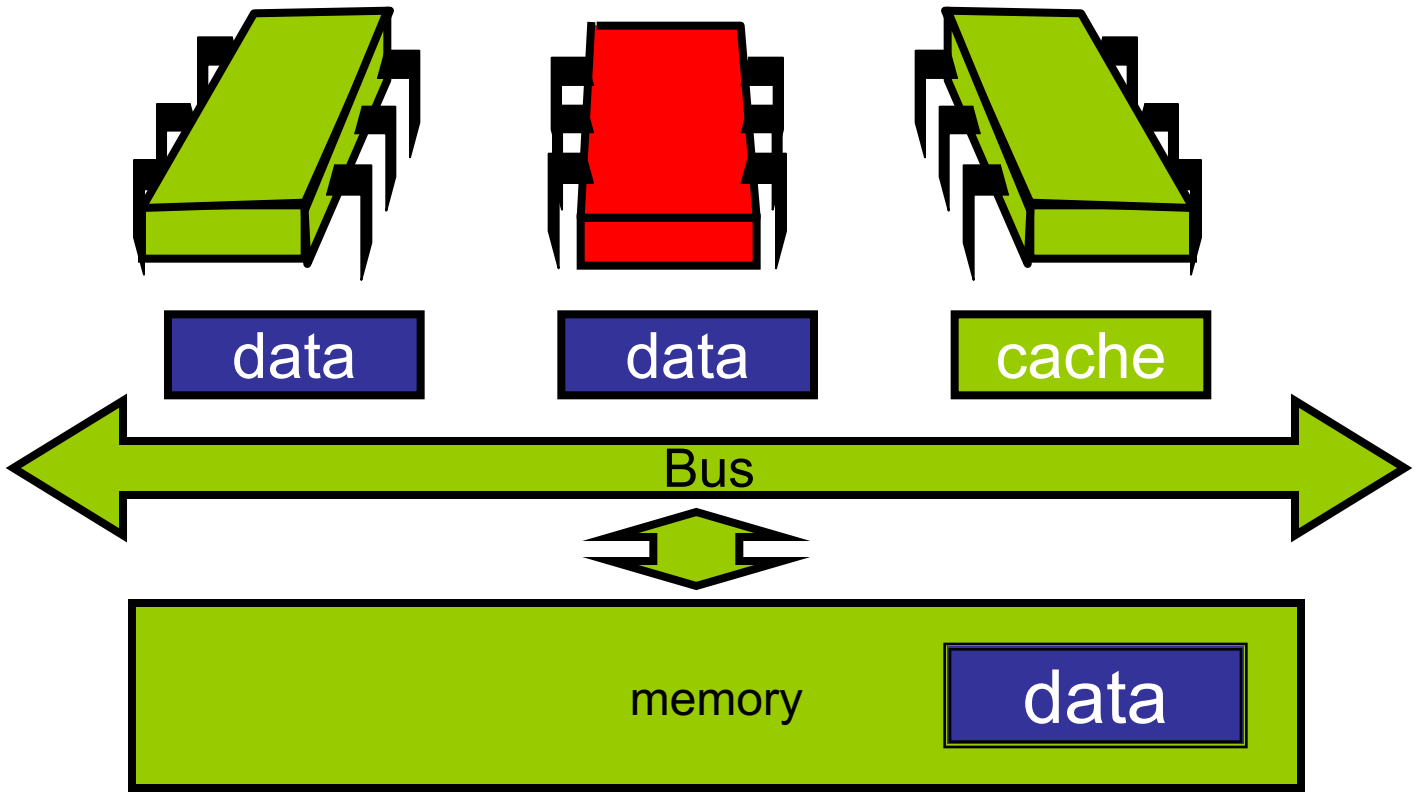
Other Processor Responds



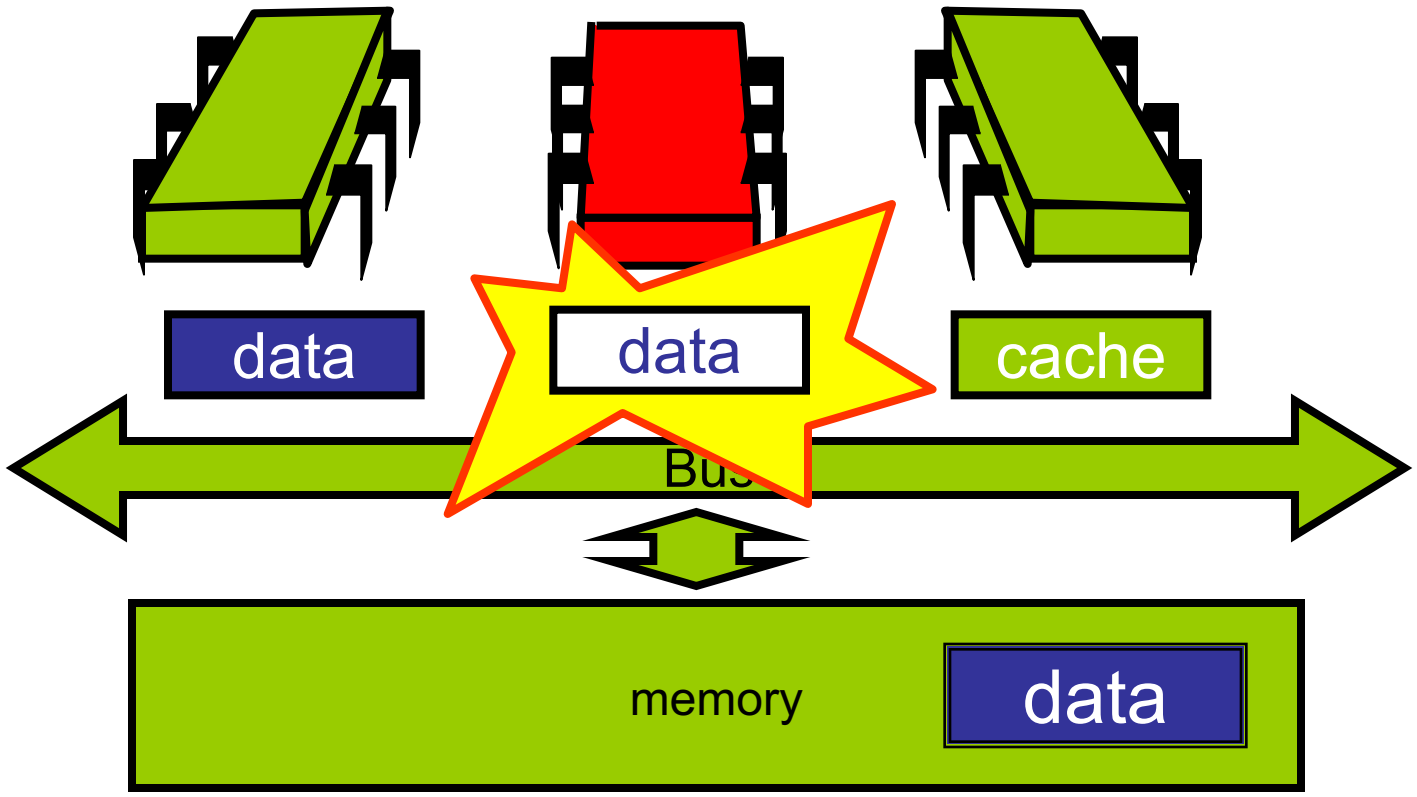
Other Processor Responds



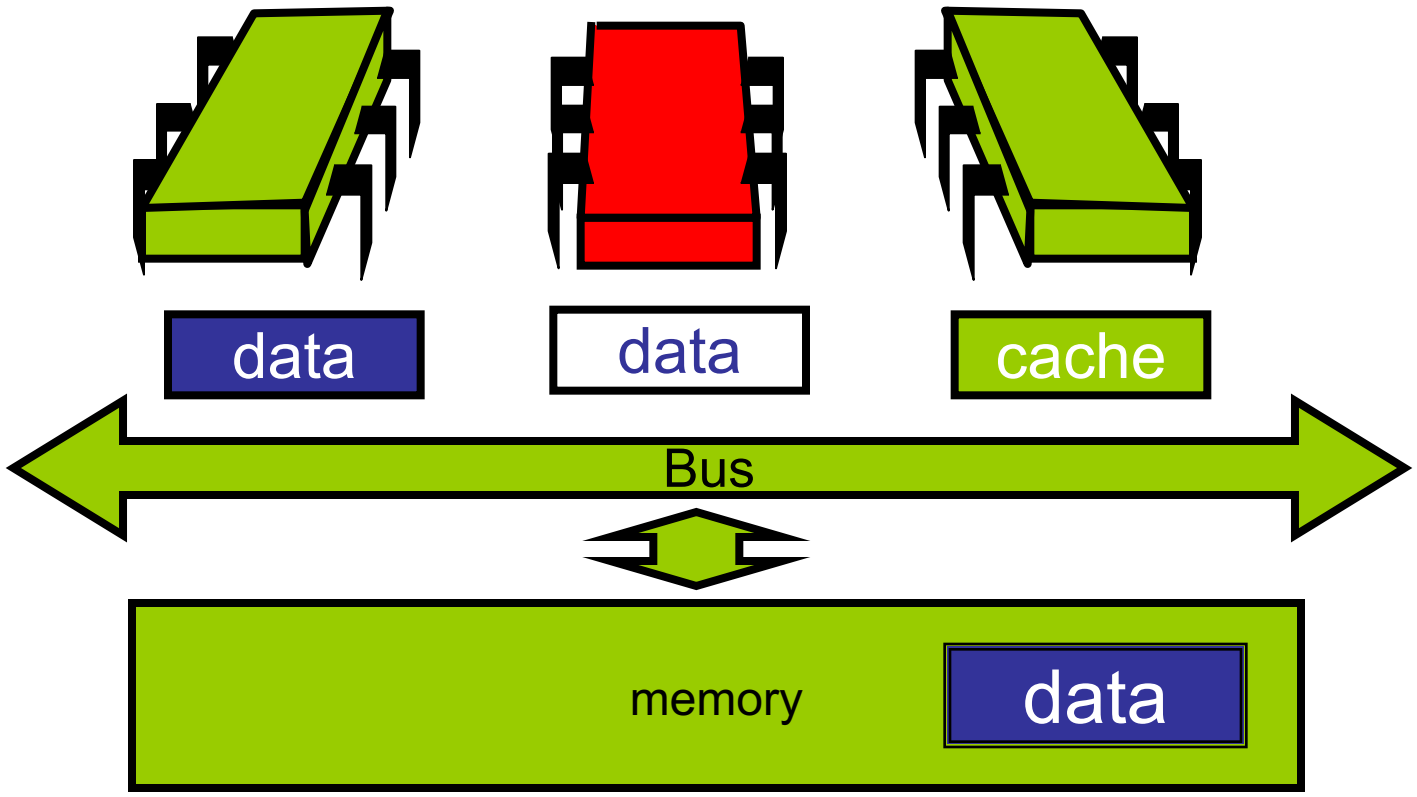
Modify Cached Data



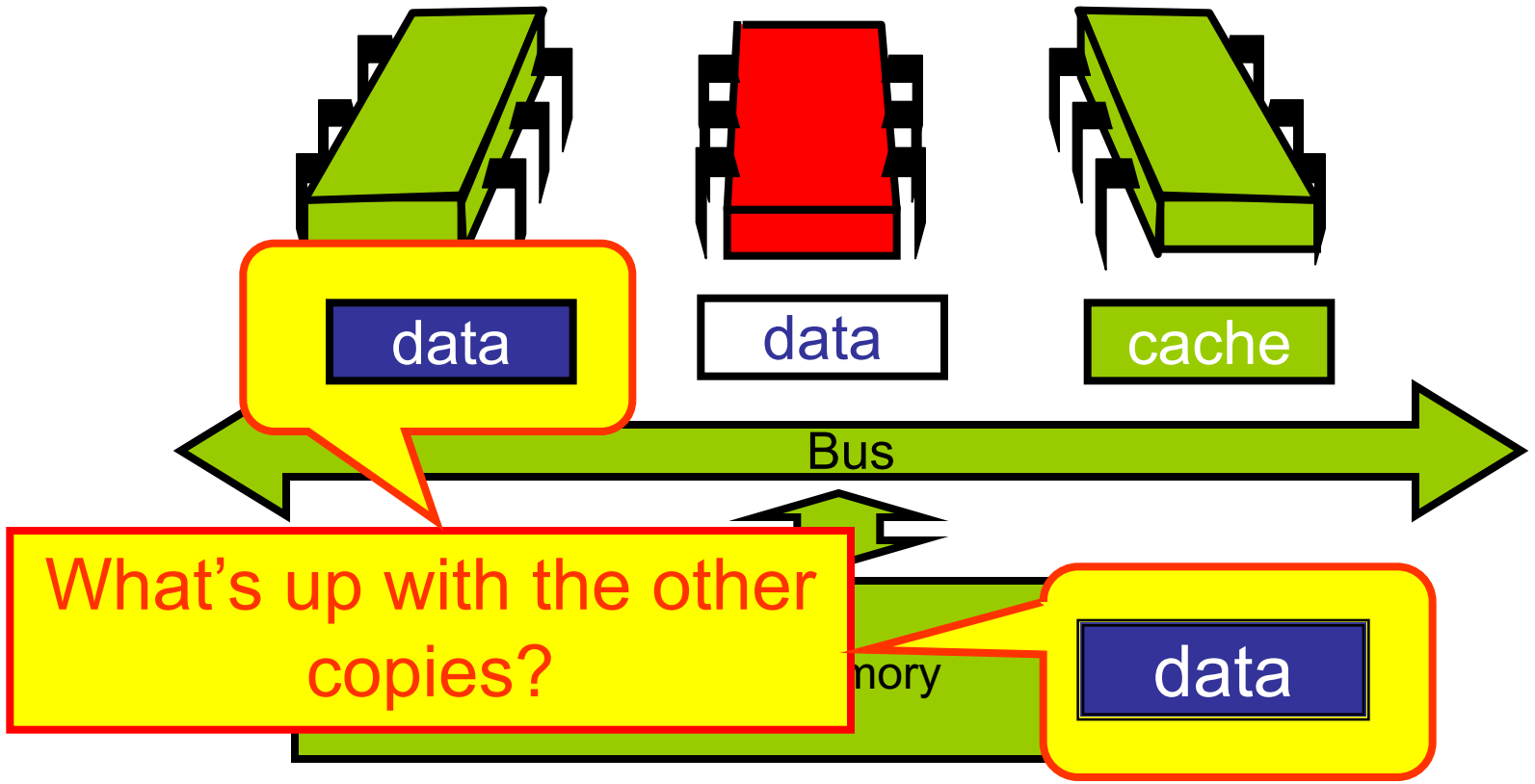
Modify Cached Data



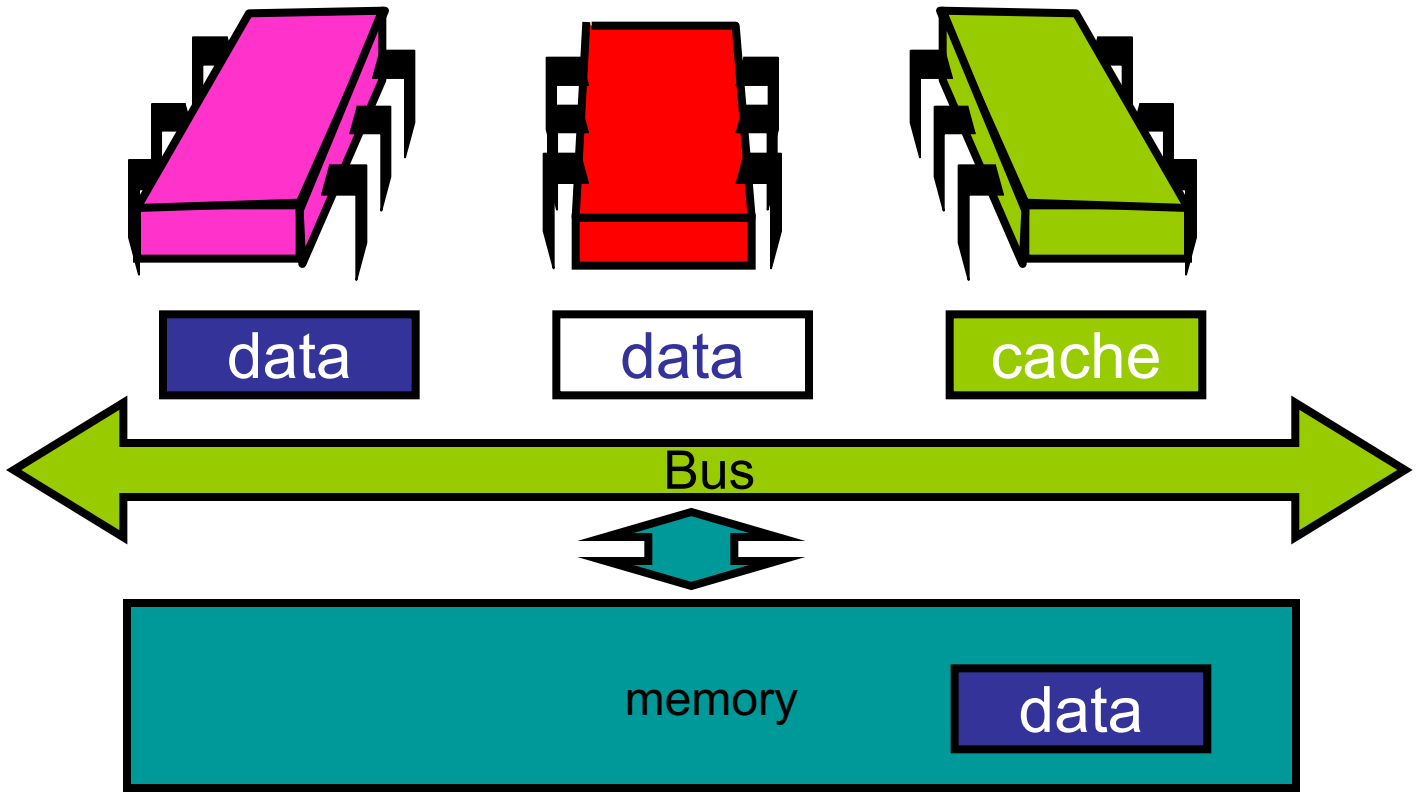
Modify Cached Data



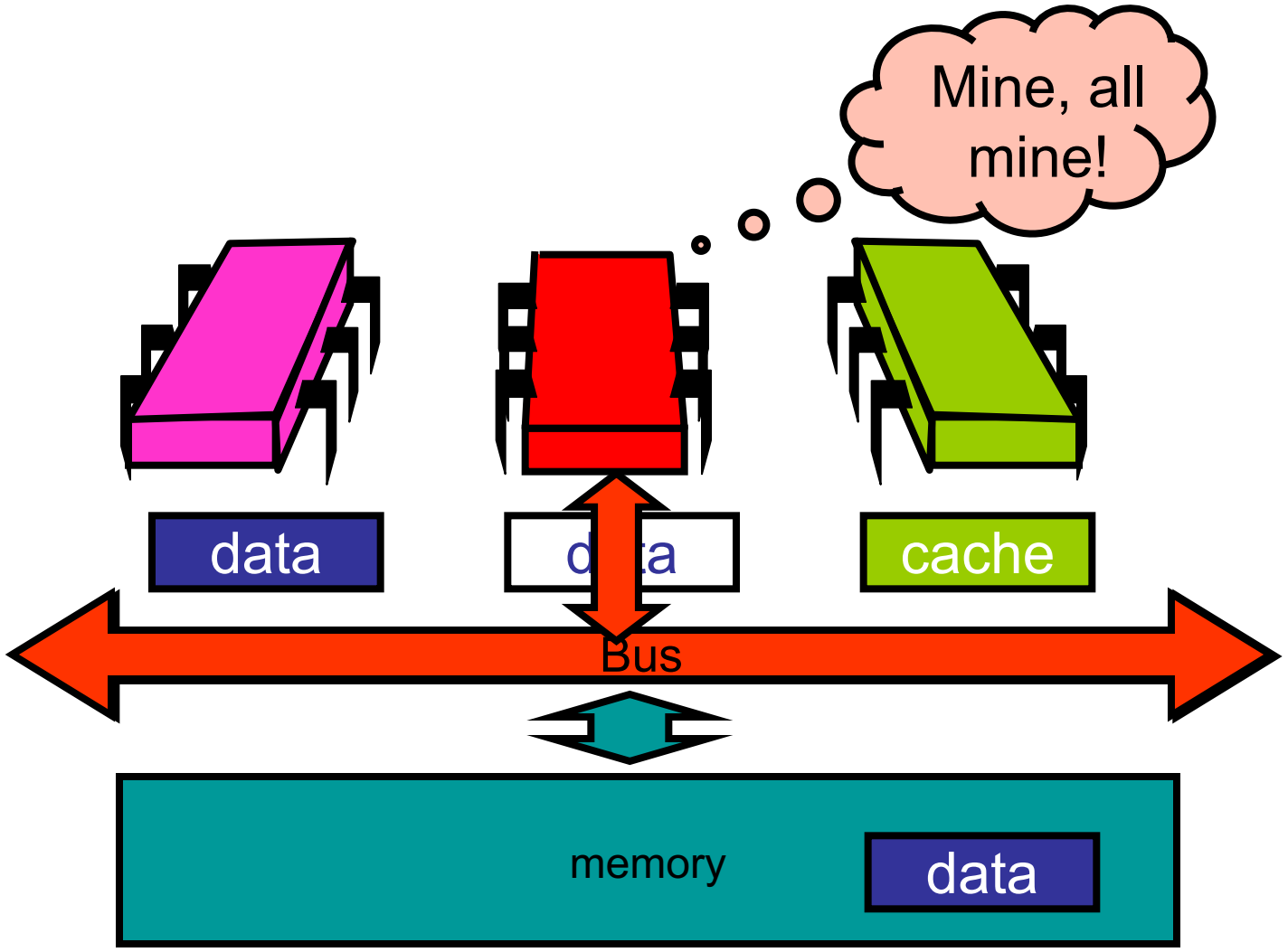
Modify Cached Data



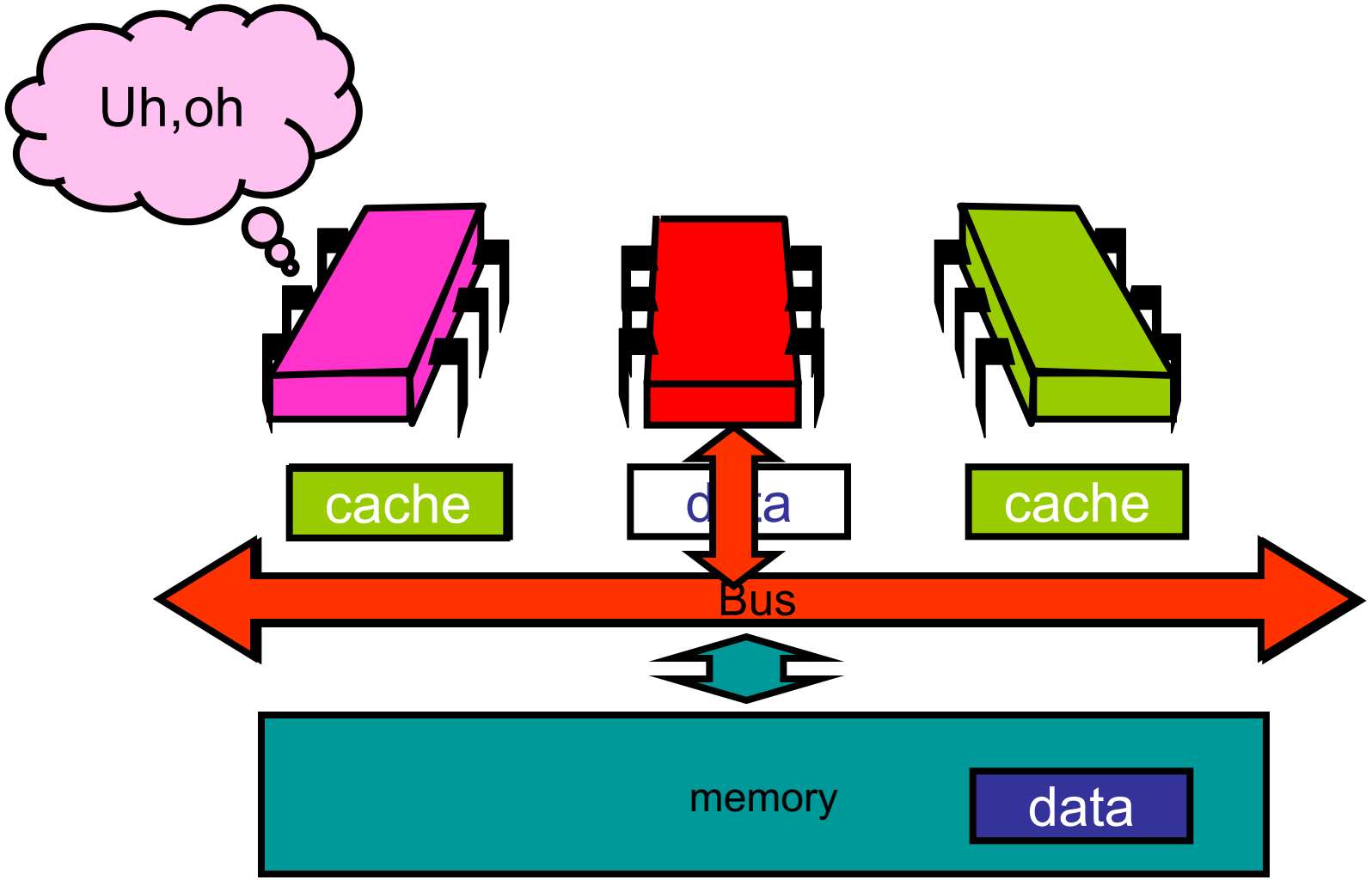
Invalidate



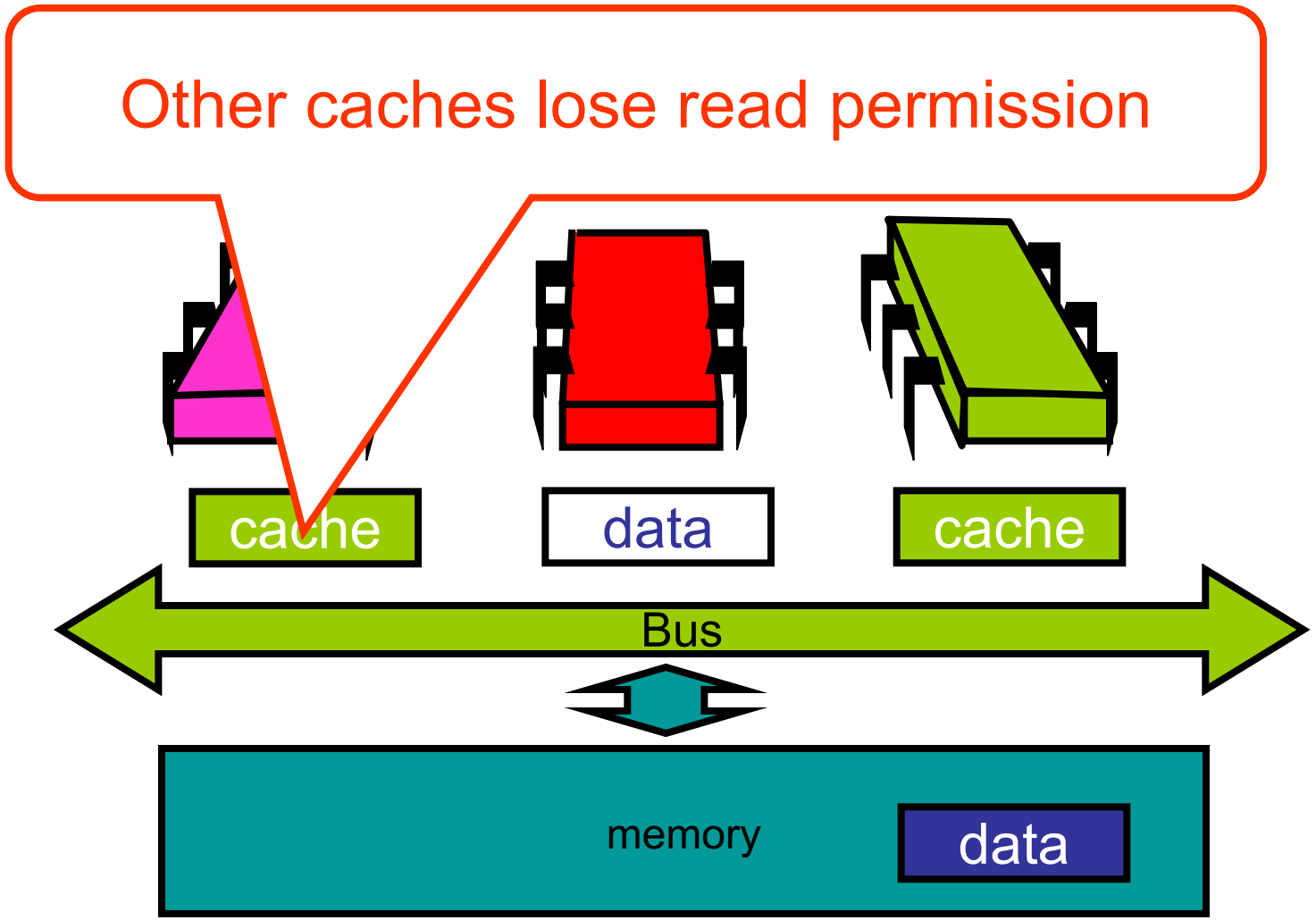
Invalidate



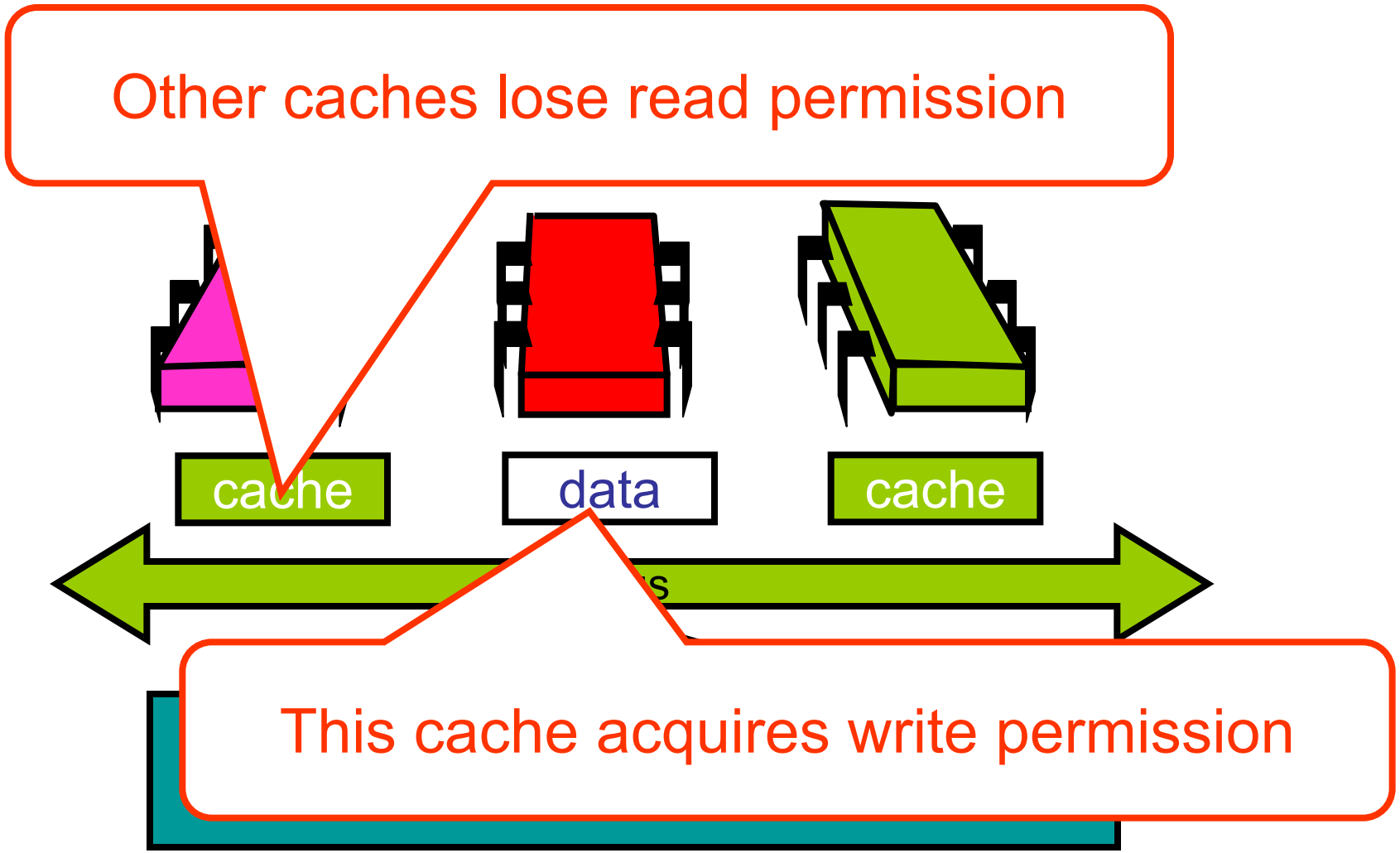
Invalidate



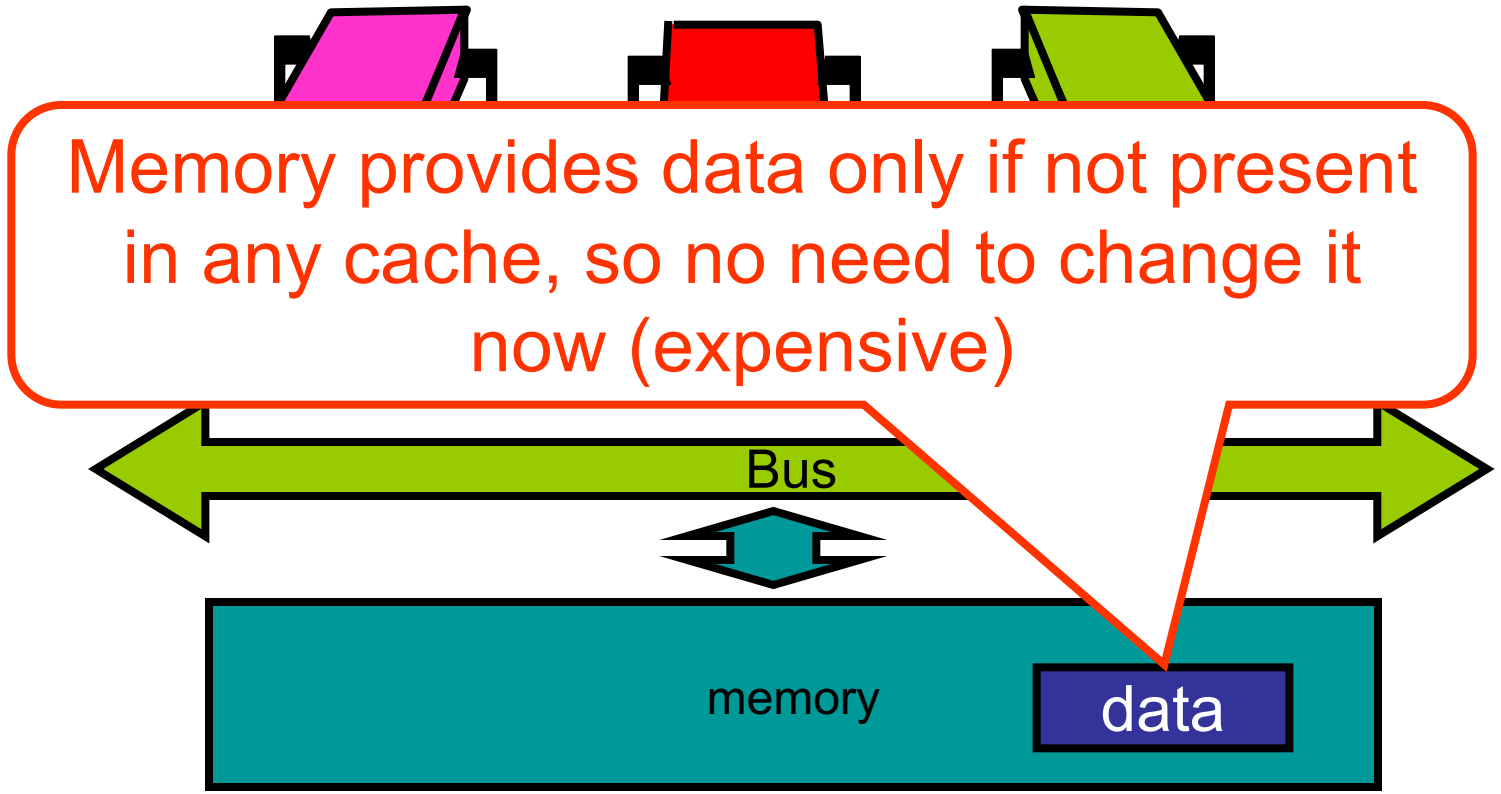
Invalidate


Other caches lose read permission
cache
data
cache
Bus
memory
data

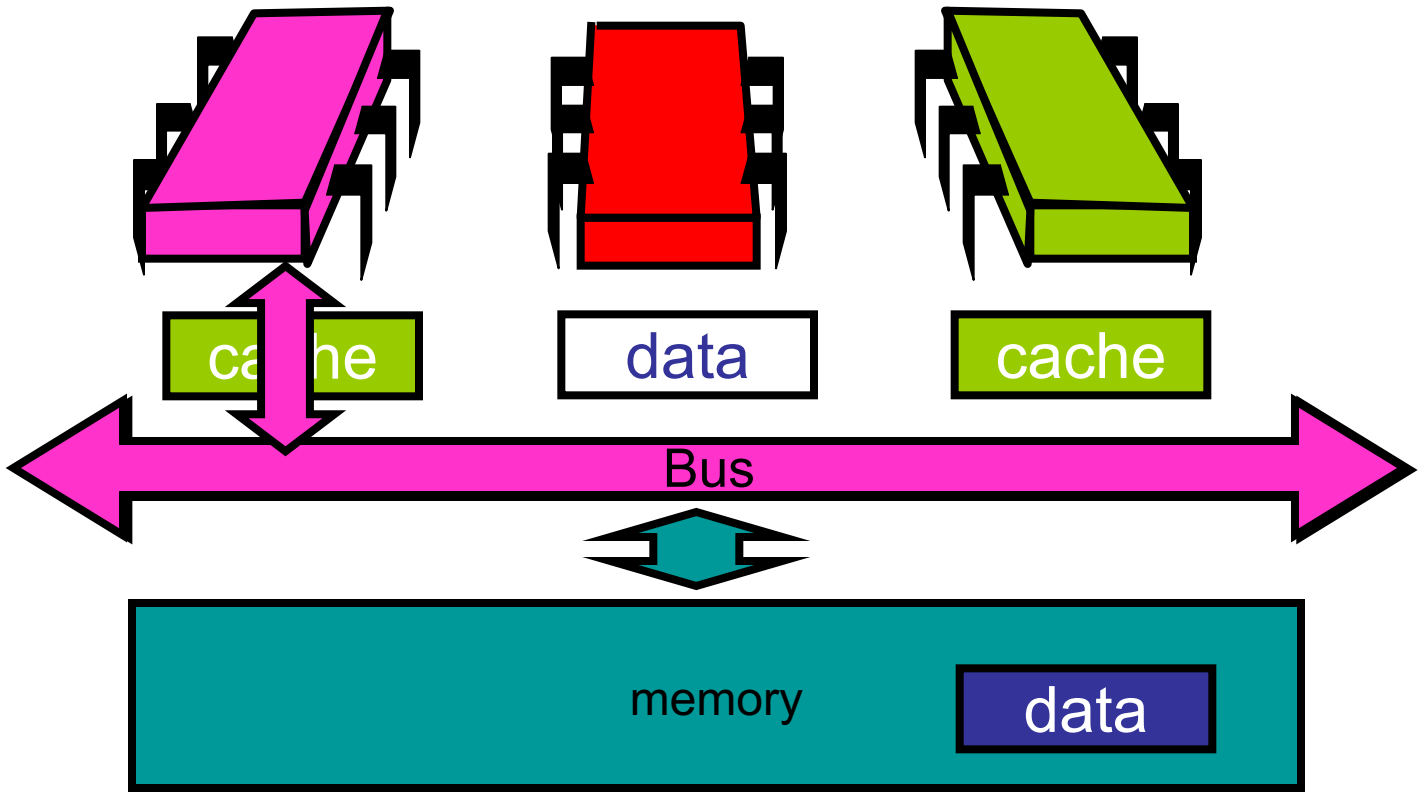
Invalidate


Other caches lose read permission
cache
data
cache
This cache acquires write permission

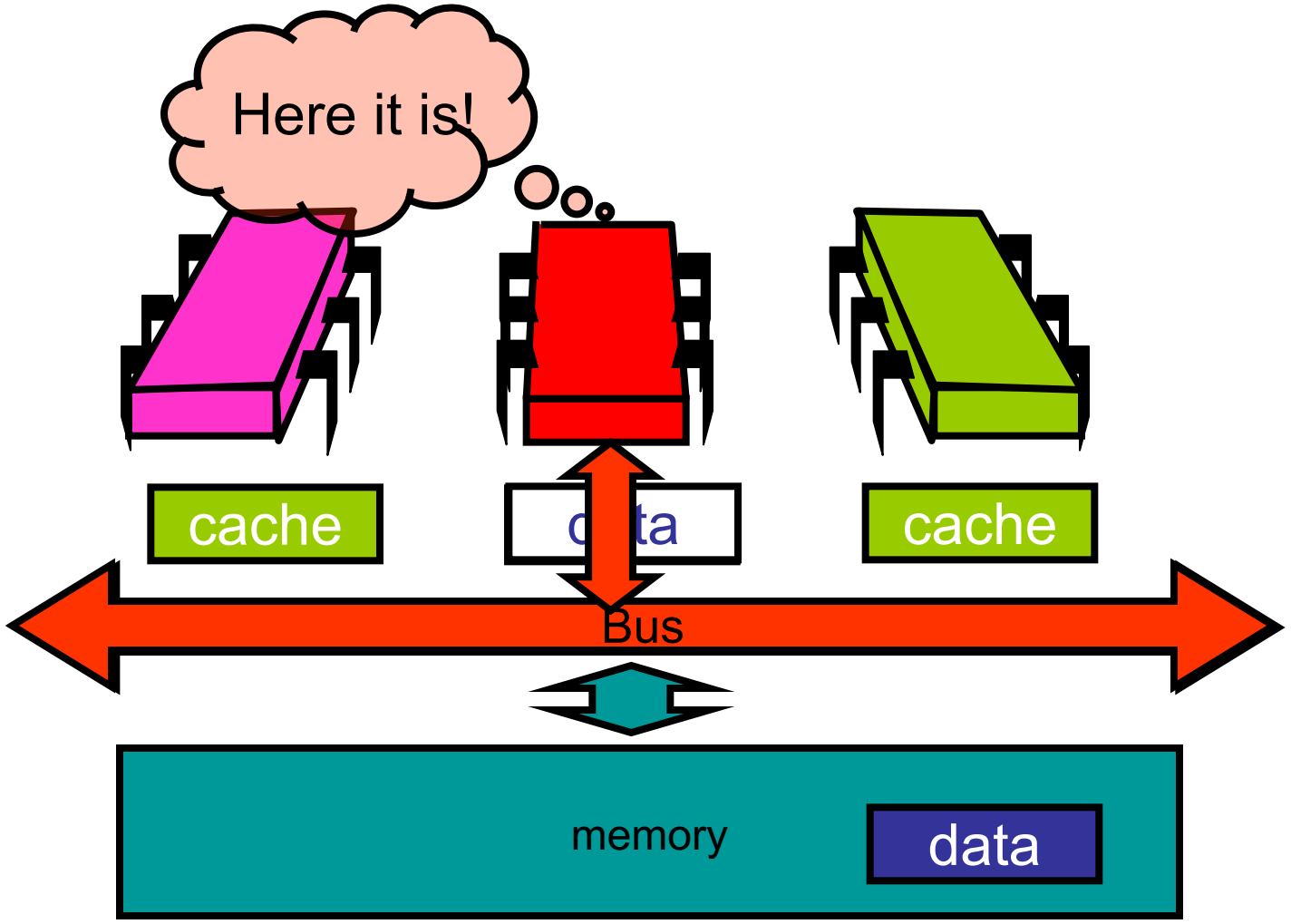
Invalidate


Memory provides data only if not present in any cache, so no need to change it now (expensive)
Bus
memory
data

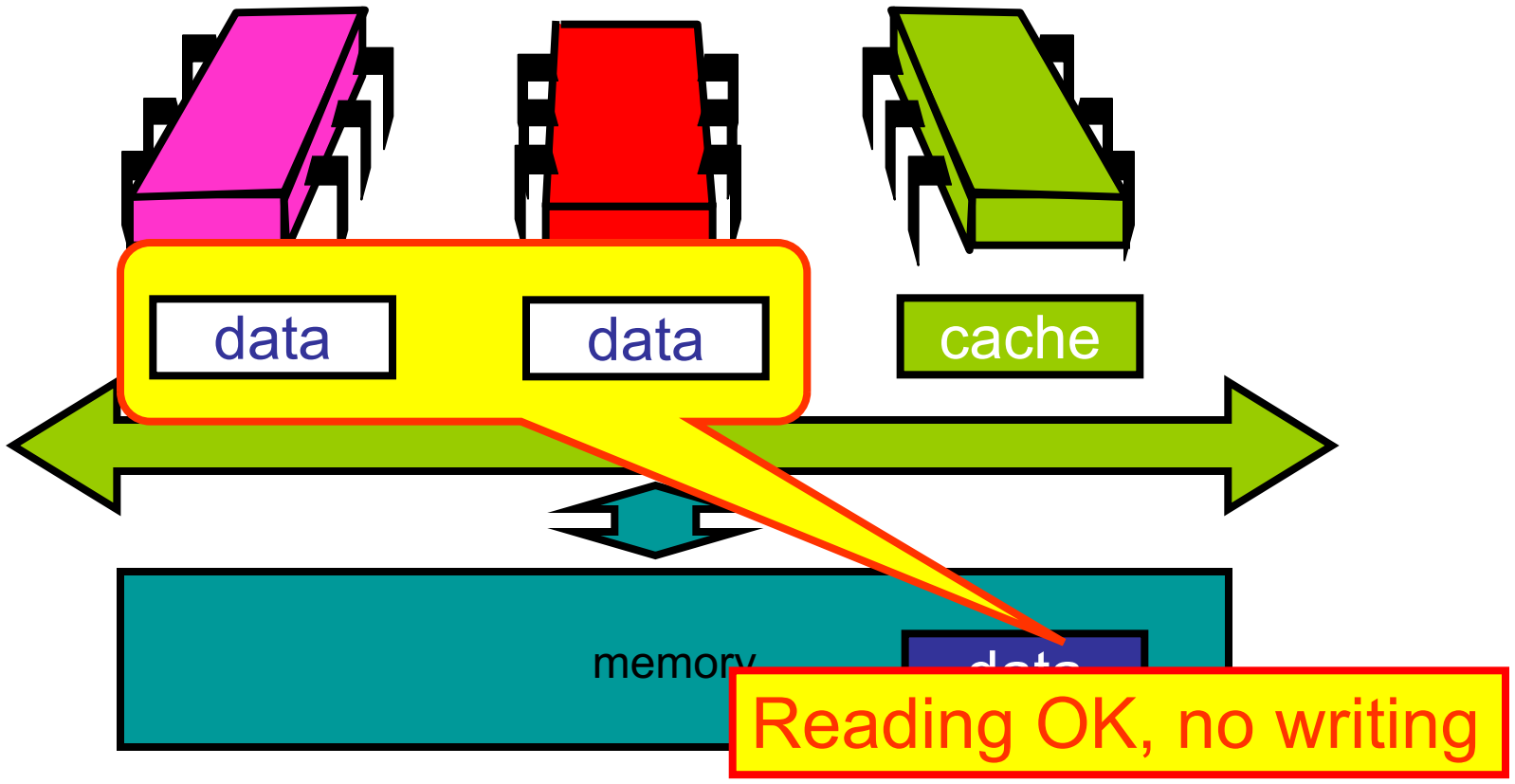
Another Processor Asks for Data



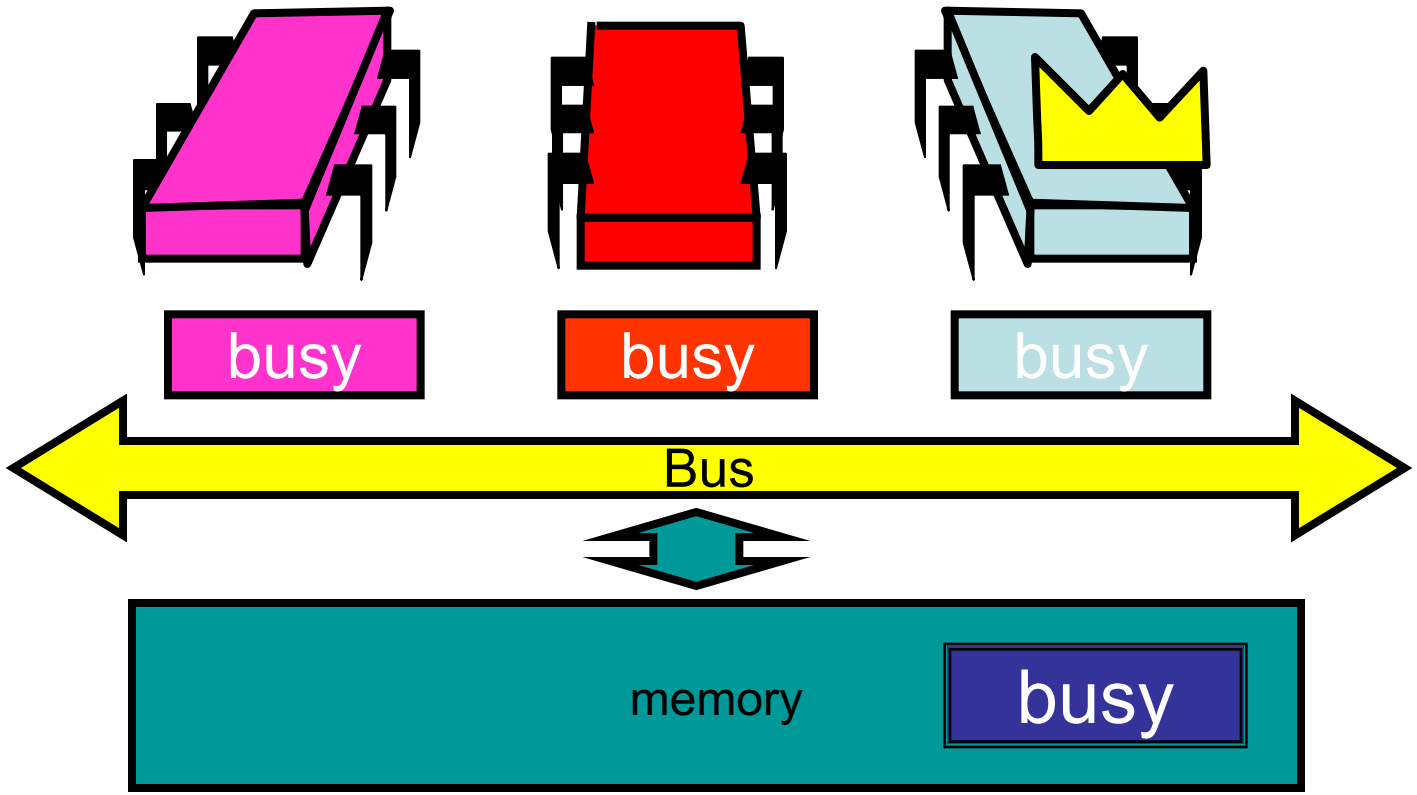
Owner Responds



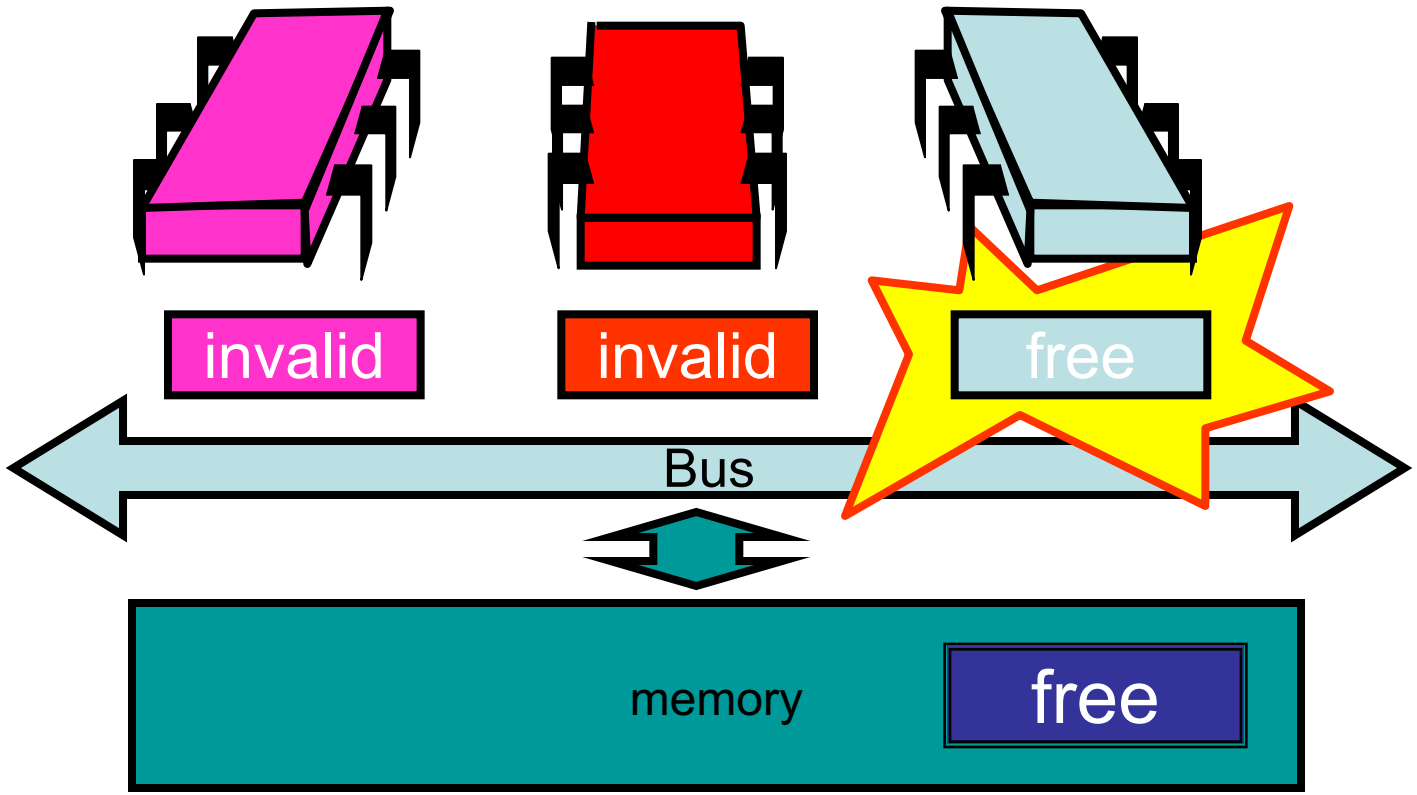
End of the Day …



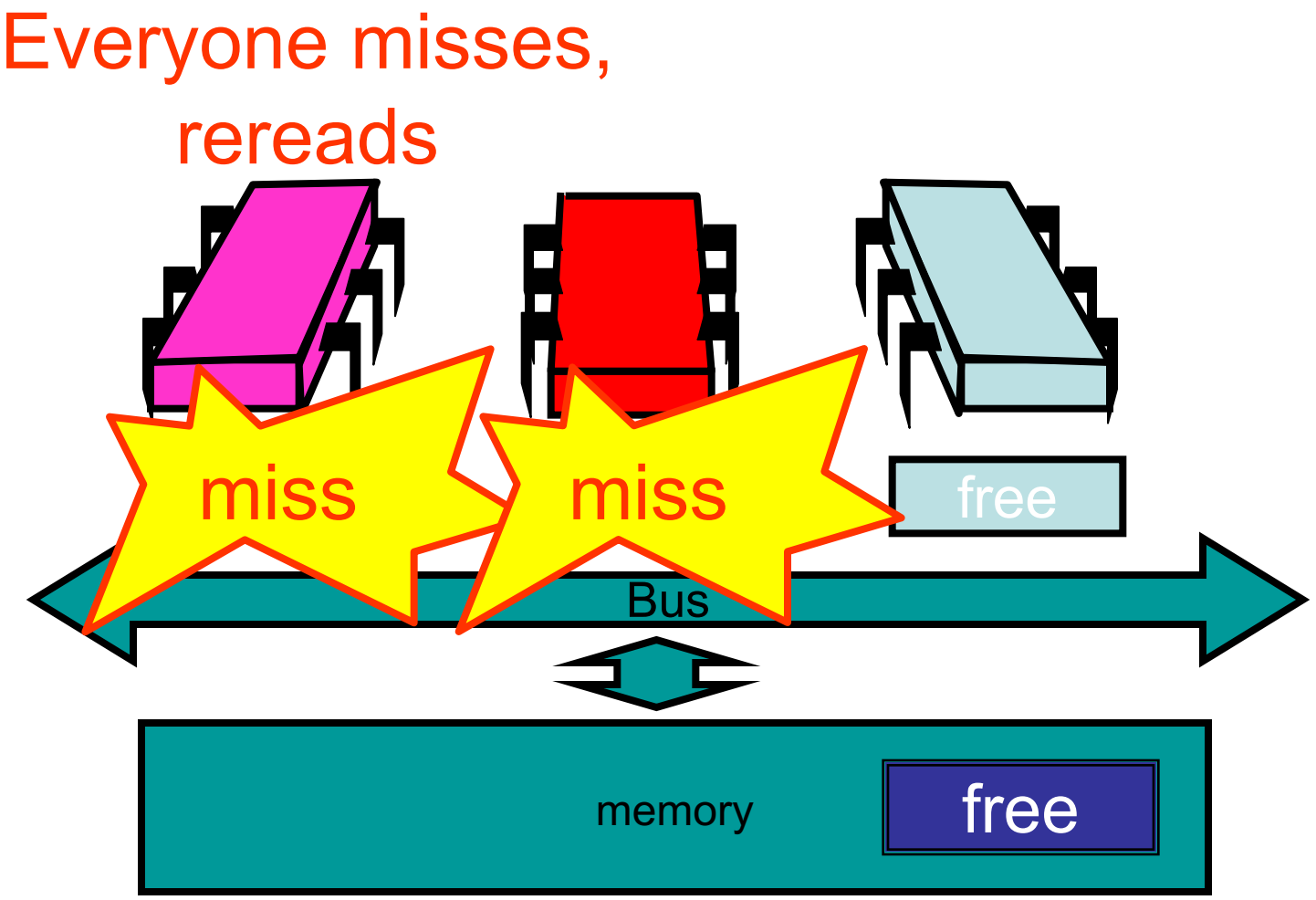
Local Spinning while Lock is Busy



On Release

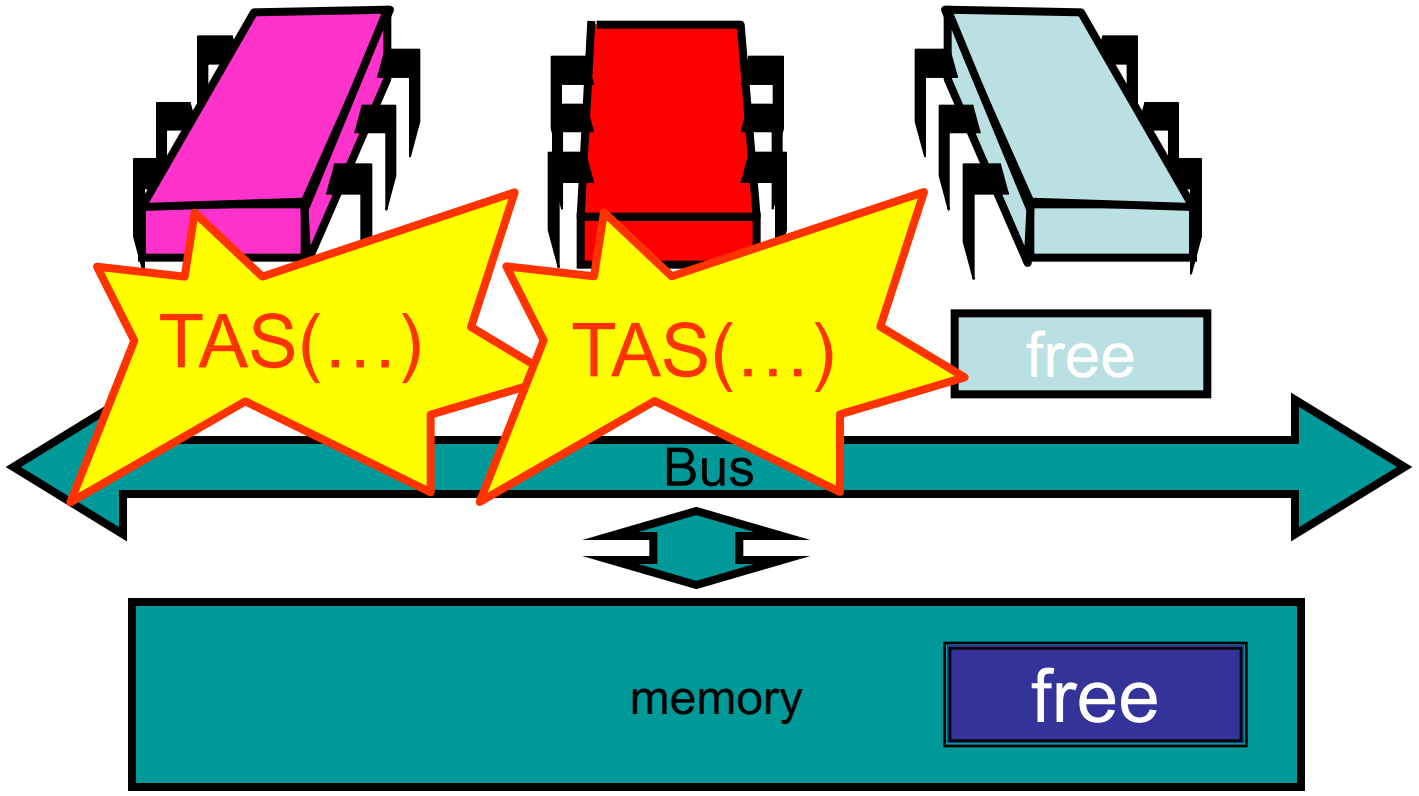


On Release

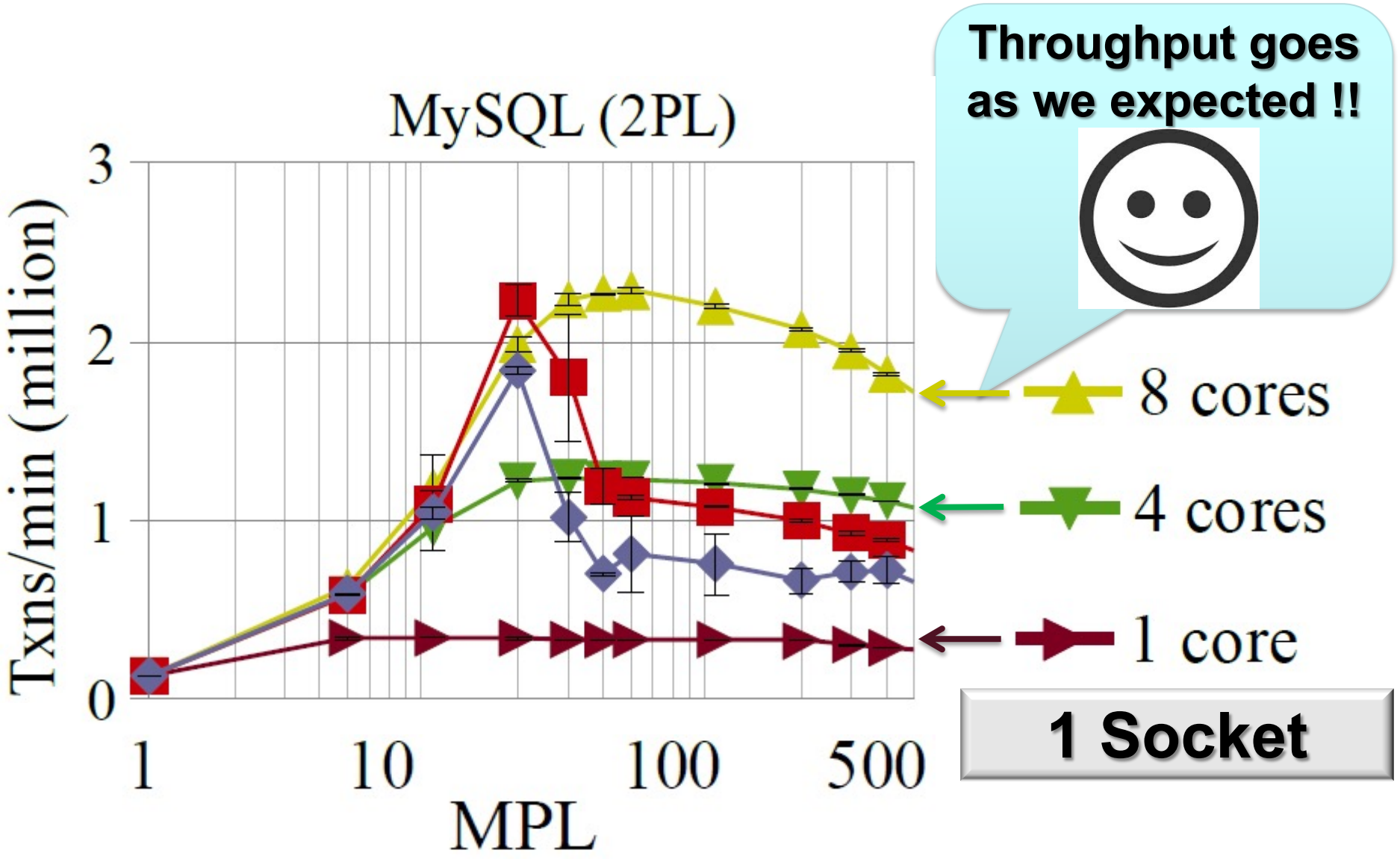


On Release

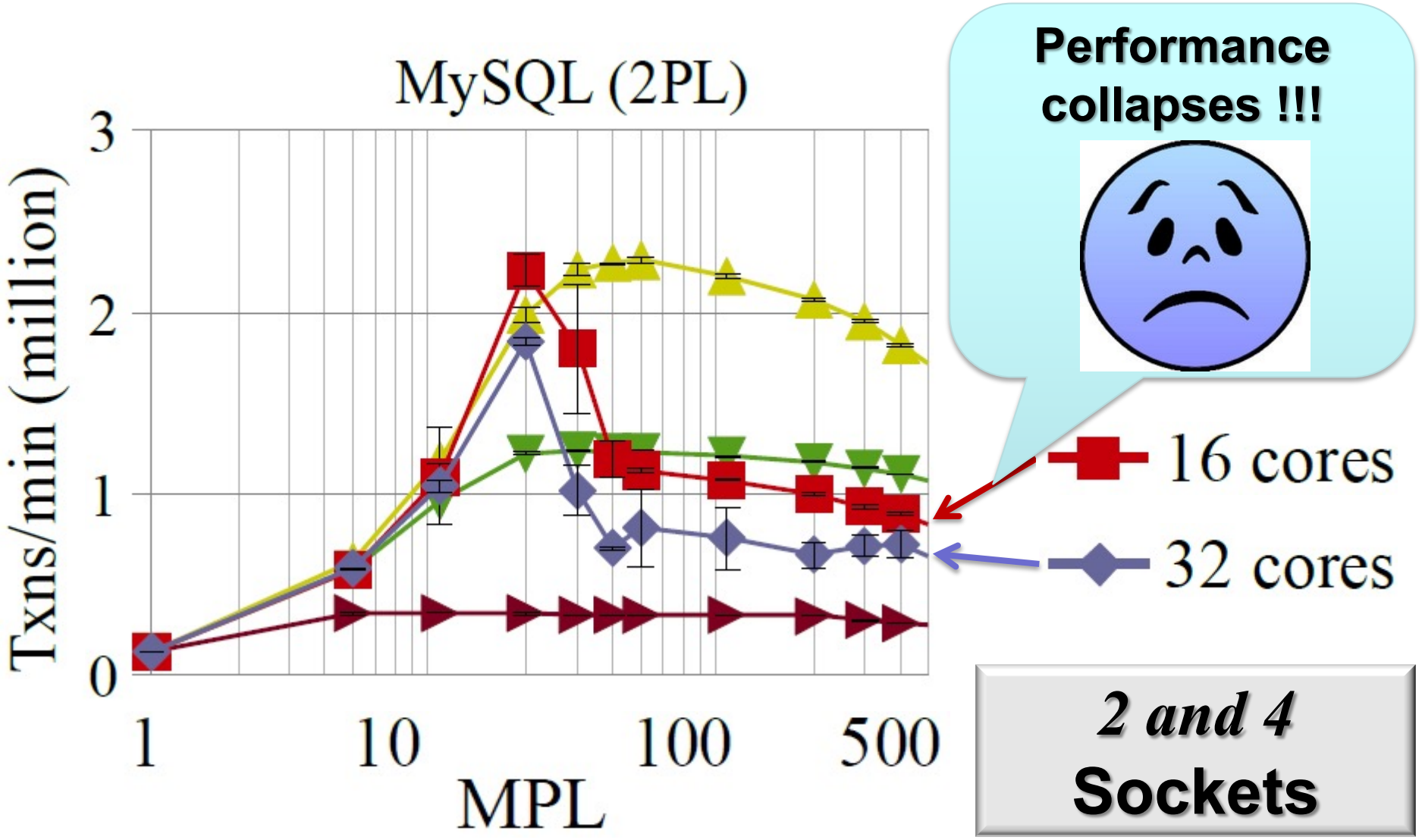
Everyone tries TAS (Test-And-Set)



How bad is the performance collapse?



How bad is the performance collapse?

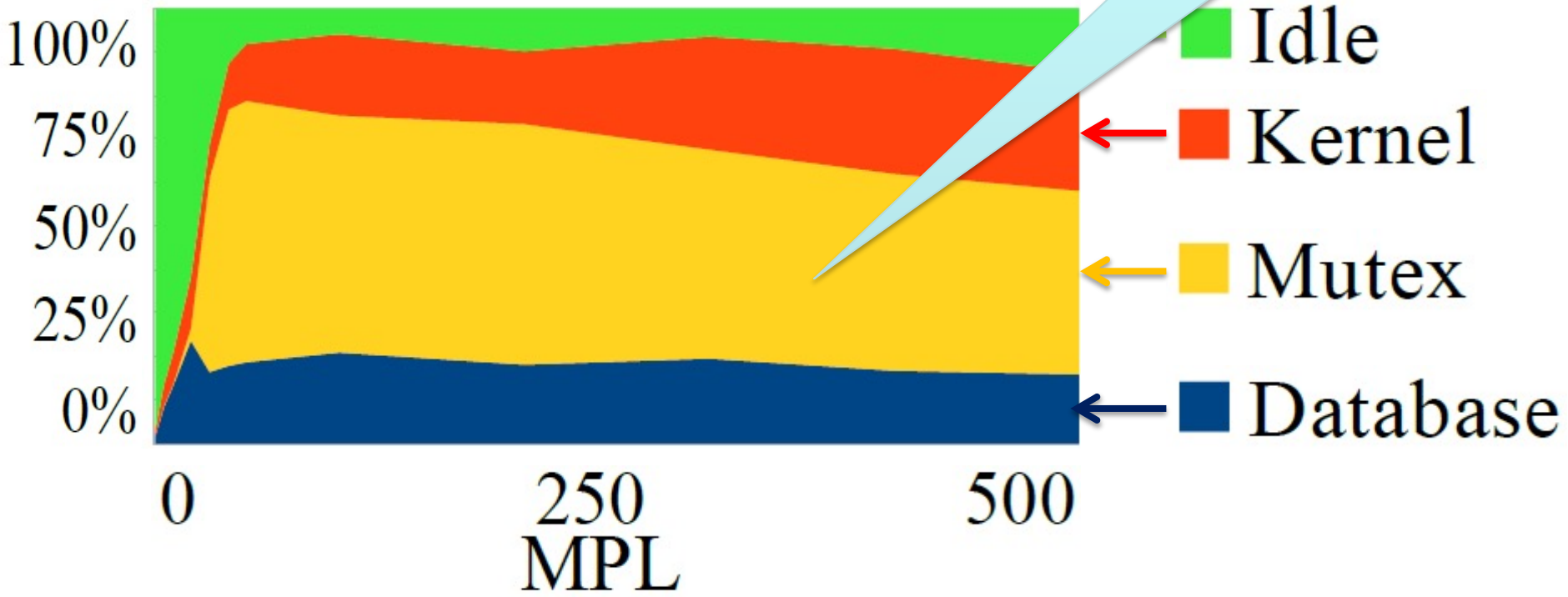


What causes this collapse ?

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.**

What I proposed to resolve this …

- 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)

Lock Acquire in Growing Phase

```
A1:  n_lock = lock_create();
A2:  n_lock->state = ACTIVE;                          S1
A3:  atomic_lock_insert(n_lock);
A4:  for all locks (lock) in hash_bucket
A5:    if (lock is incompatible with n_lock)
A6:      n_lock->state = WAIT;                        S2
A7:      atomic_synchronize();
A8:      if (lock->state==OBSOLETE)
A9:        n_lock->state=ACTIVE;                      S3
A10:       atomic_synchronize();
A11:       continue;
A12:     if (new_deadlock()==TRUE)
A13:       abort Tx;
A14:     break;
A15:   end if
A16: end for
A17: if (n_lock->state == WAIT)
A18:   mutex_enter(Tx->mutex);                         S4
A19:   atomic_synchronize();
A20:   if ( n_lock has to wait )
A21:     Tx->state = WAIT;
A22:     os_cond_wait(Tx->mutex);
A23:   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)

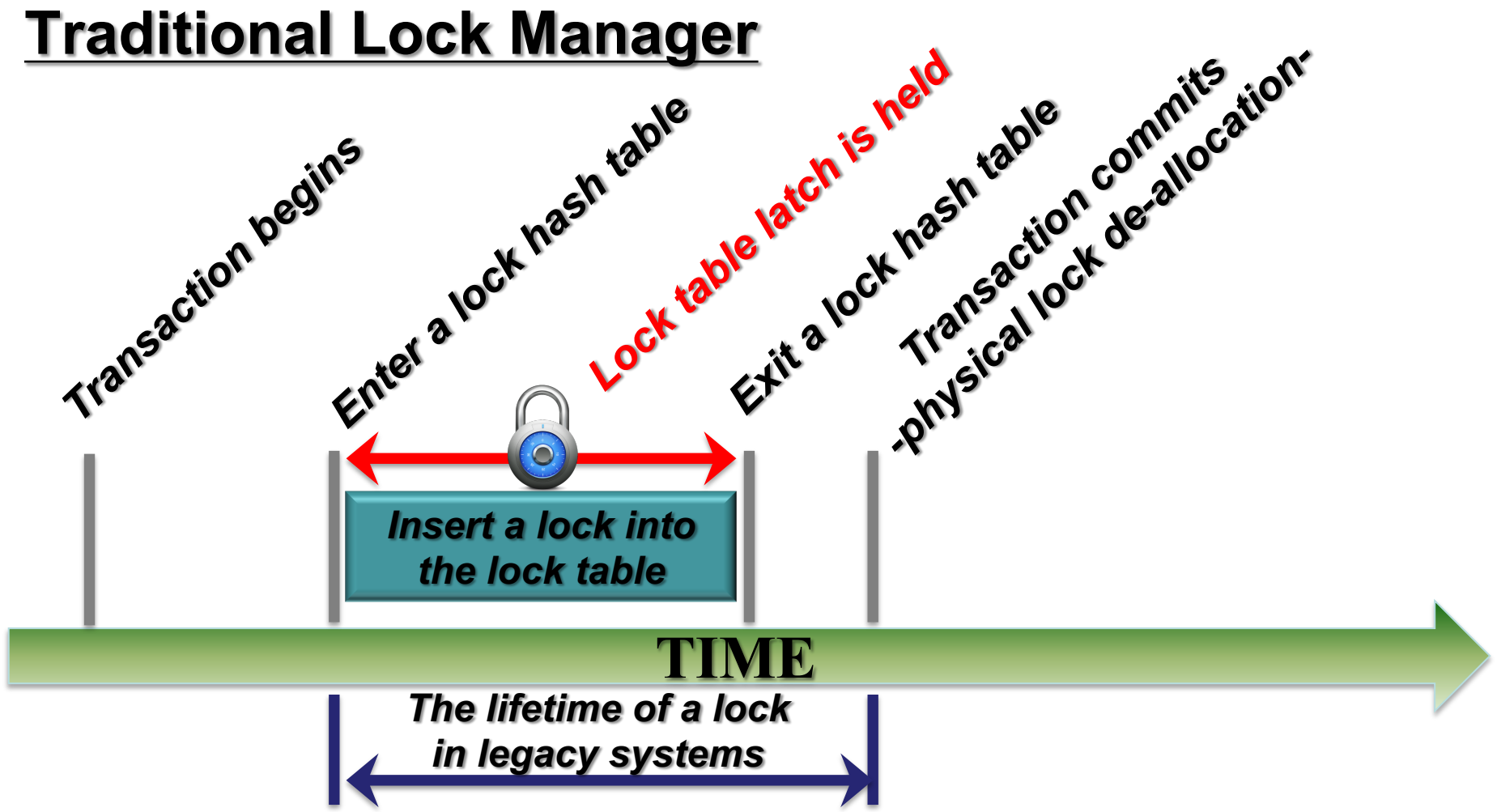
Lock Release in Shrinking Phase

```
R1:  for all locks (lock1) in Tx
R2:     lock1->state = OBSOLETE;                    S6
R3:     atomic_synchronize();
R4:     for all locks (lock2) that follow lock1
R5:       mutex_enter(lock2->Tx->mutex);
R6:       if ( lock2->Tx->state==WAIT &&
R7:            lock2 does not have to wait )
R8:              lock2->Tx->state=ACTIVE;
R9:              lock2->state=ACTIVE;               S7
R10:             atomic_synchronize();
R11:             os_cond_signal(lock2->Tx);
R12:      end if
R13:      mutex_exit(lock2->Tx->mutex);
R14:   end for
R15:end for
```

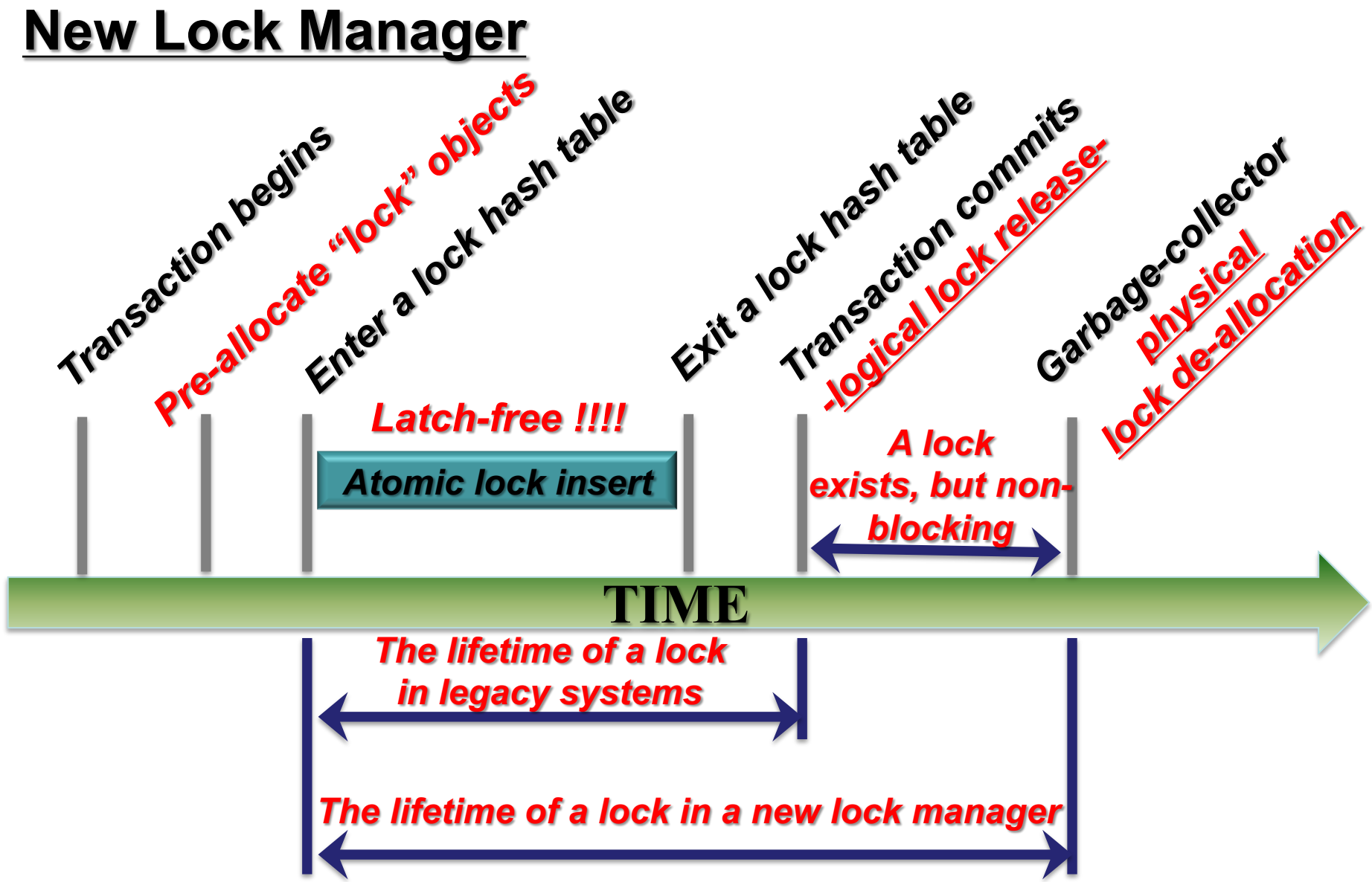
<-Write
<-Barrier
<-Read

<-Write
<-Barrier
<-Read

Staged allocation and de-allocation



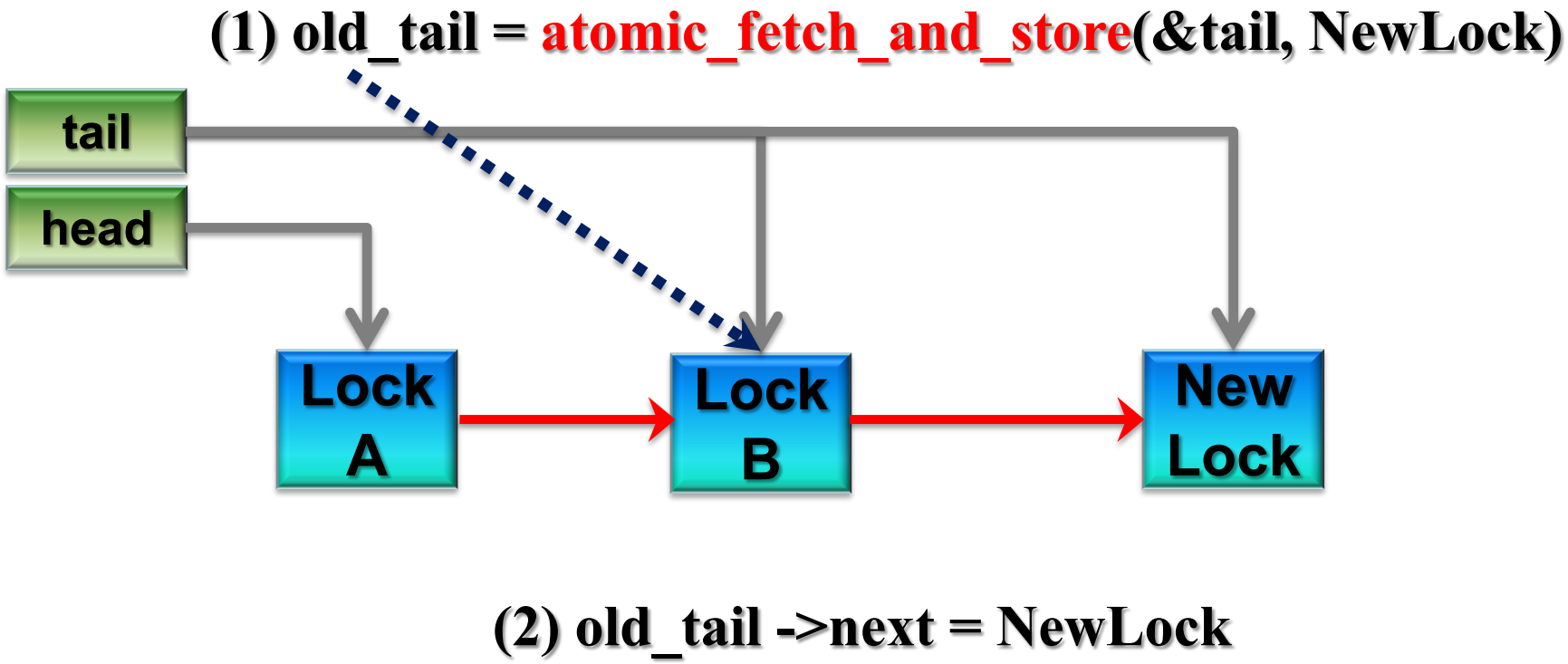
Staged allocation and de-allocation



Two important operations

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

Atomic lock insert (1/2)



Atomic lock insert (2/2)

```
list = getBucketList( hash_code );
old_tail = atomic_fetch_and_store(list->tail, new_lock); //(1)
old_tail->next = new_lock; // (2)
atomic_synchronize();
next_pointer_update() ; // (3)
atomic_synchronize();
```

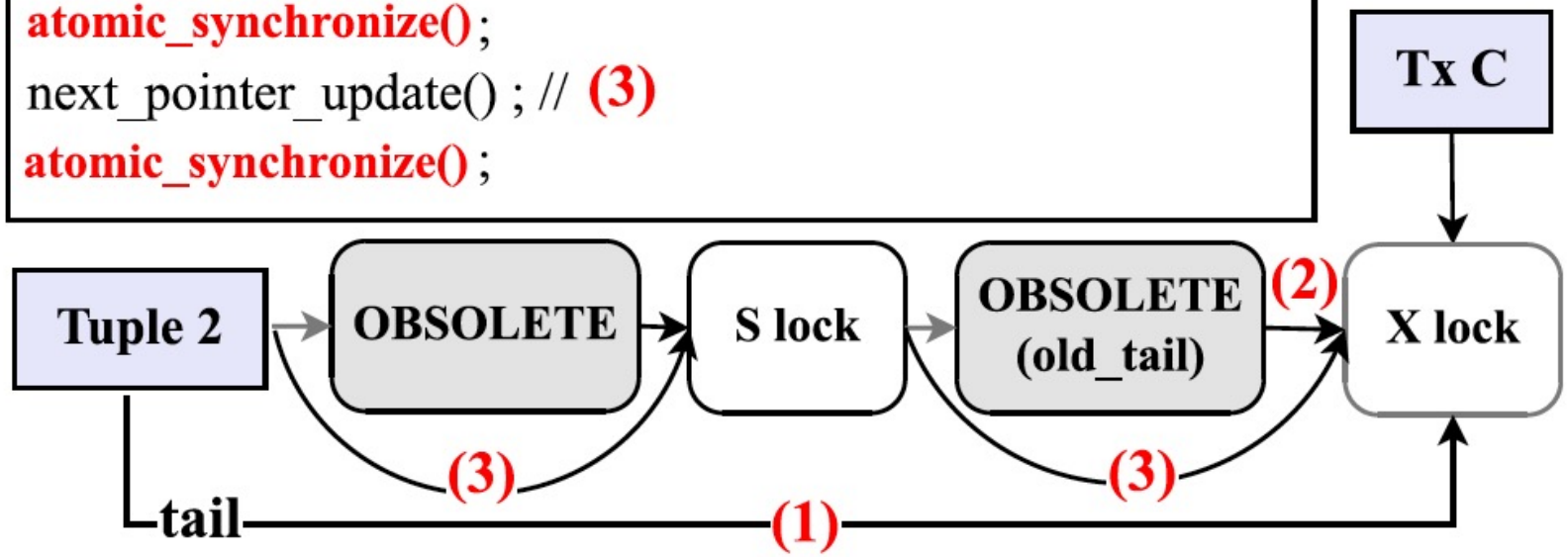
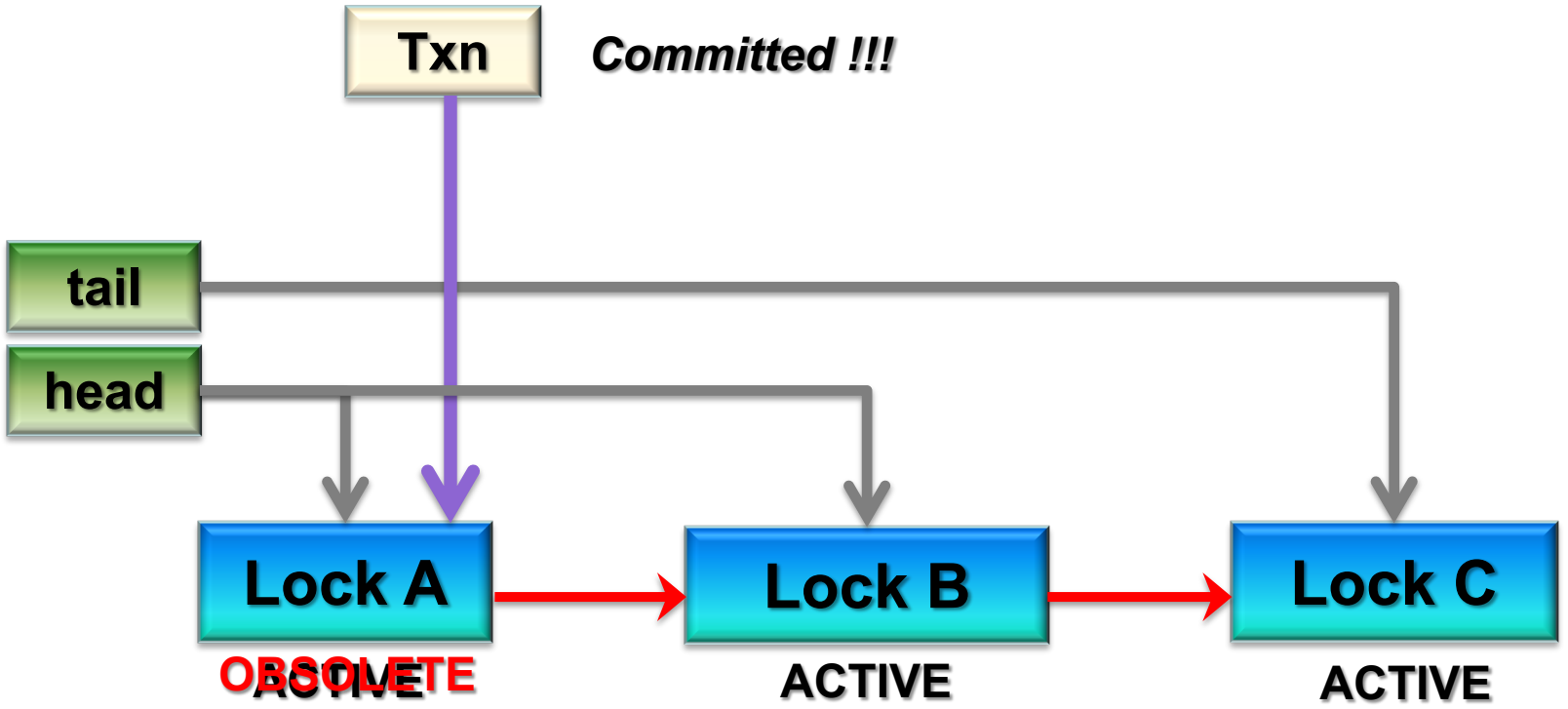


Fig. 4. Operation of `atomic_lock_insert()` on a lock hash chain.

Garage-collection (1/2)



(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.

Garage-collection (2/2)

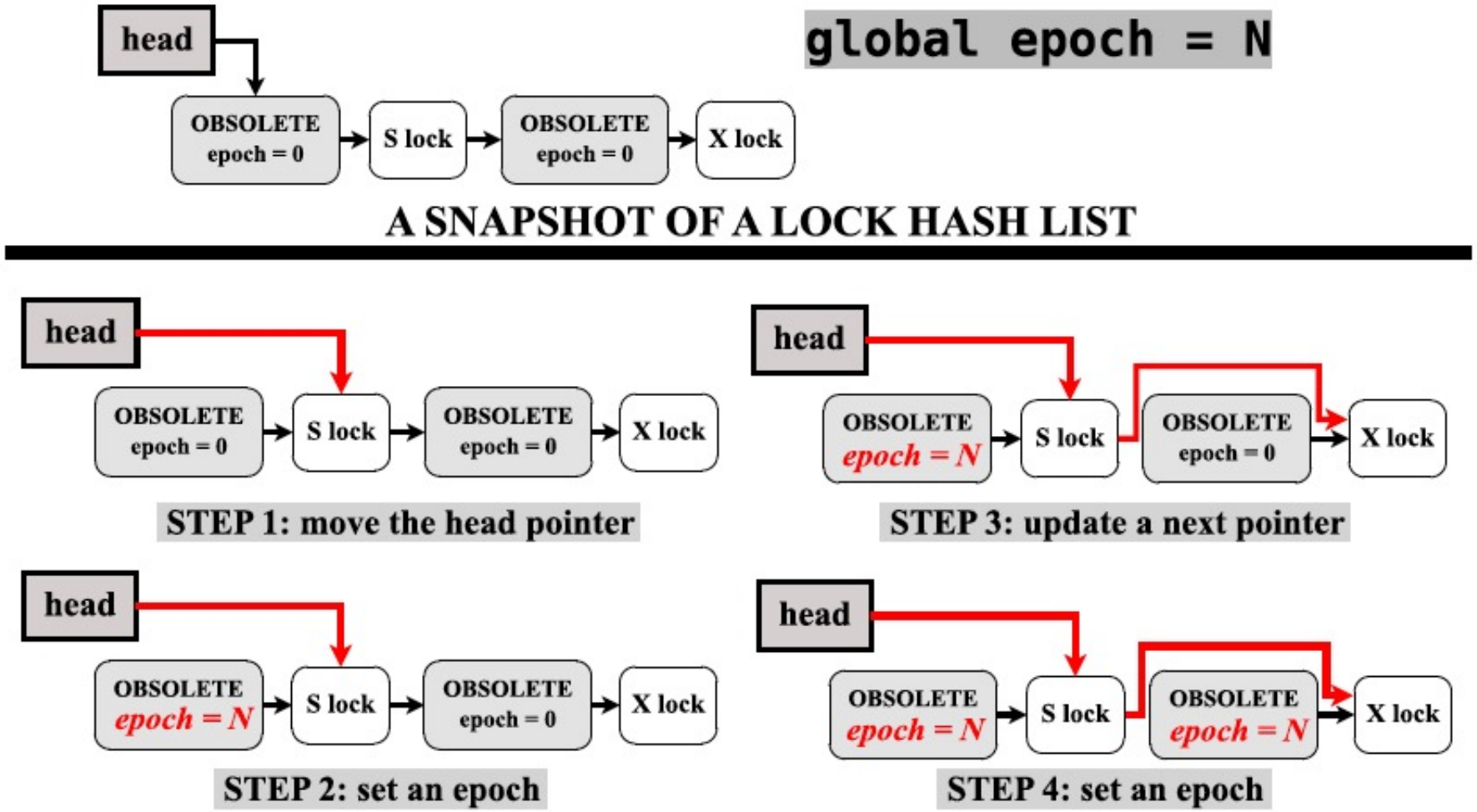
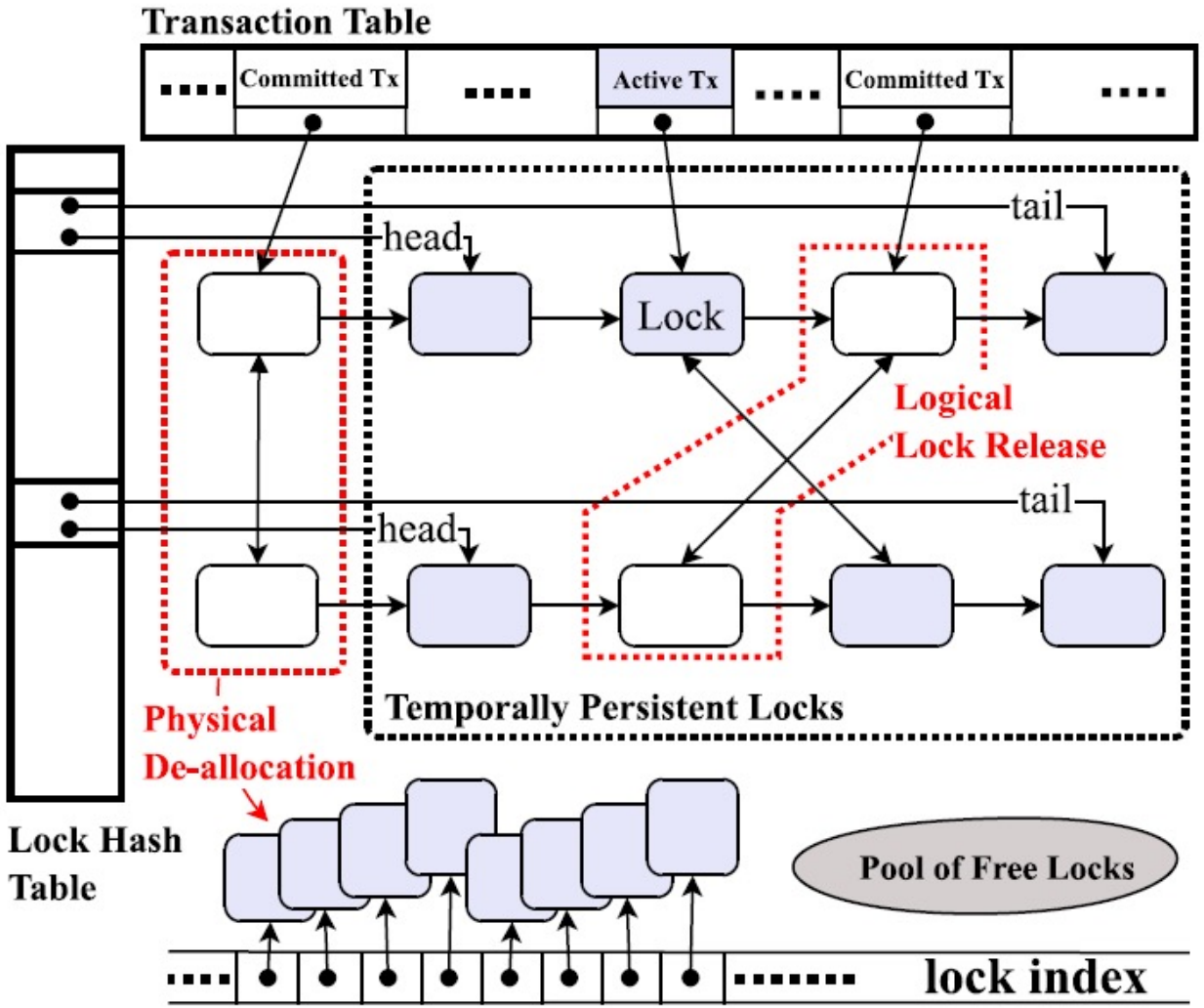


Fig. 7. Update on pointers and epochs.

The Architecture of New Lock Manager



Experimental Setup

- 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
  - Update

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

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

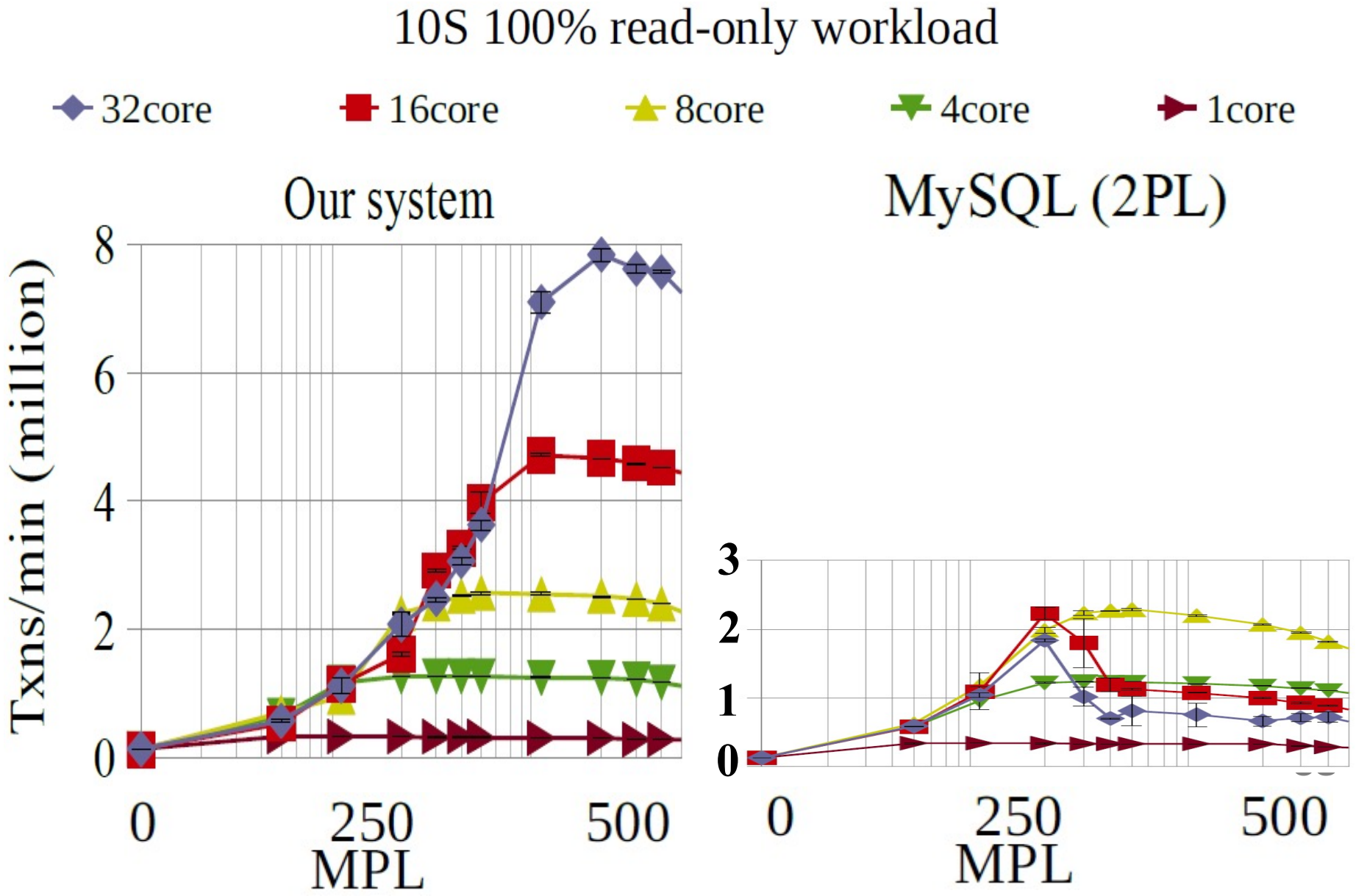
Experimental Setup (cont.)

- Multicore machines

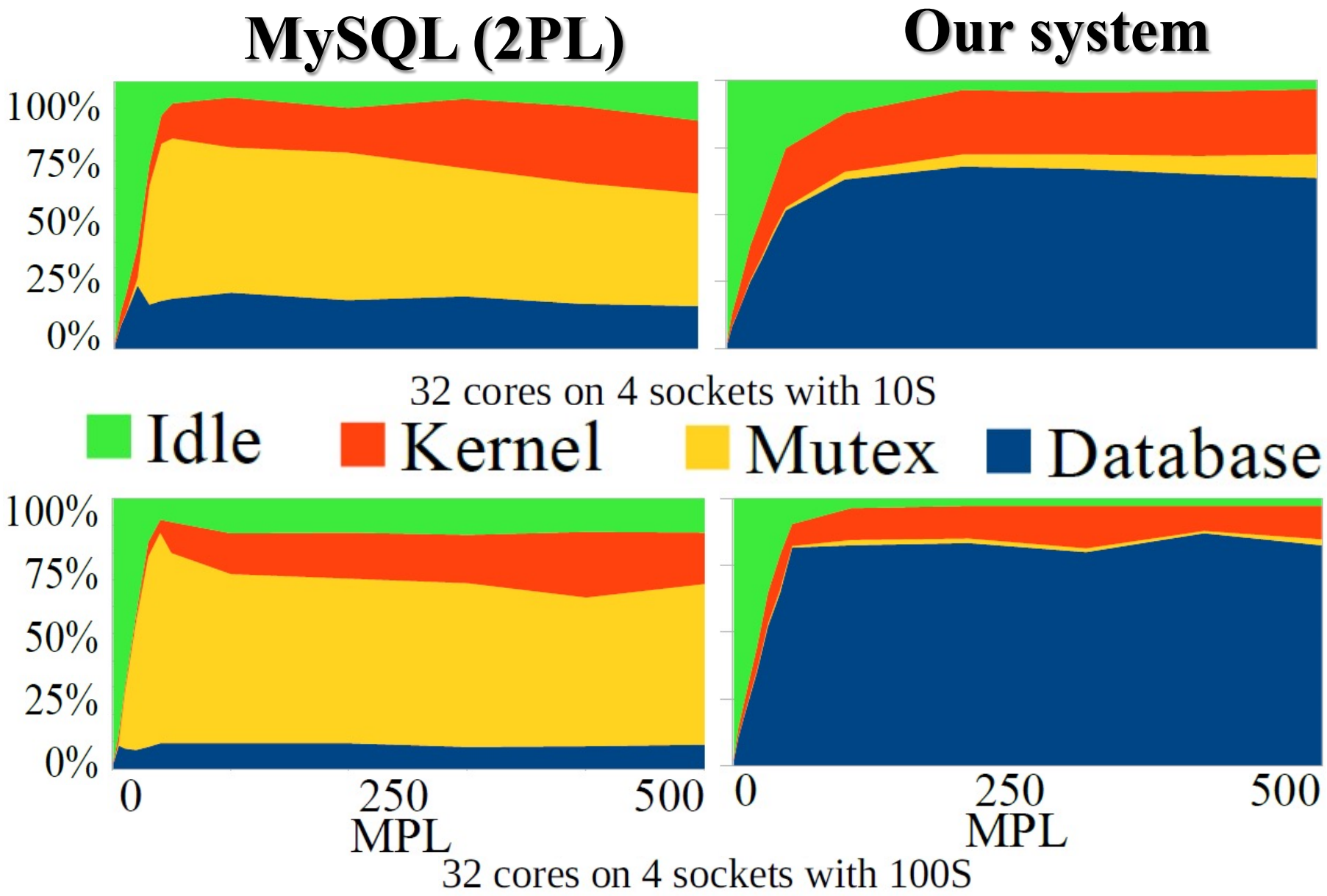
Component	Specification
Processors	8-Core Intel Xeon CPU E7-8837
Processor Sockets	4 Sockets
Hardware Threads	32 (No HyperThreading Support)
Clock Speed	2.66 GHz
L1 D-Cache	32 KiB (per core)
L1 I-Cache	32 KiB (per core)
L2 Cache	256 KiB (per core)
L3 Cache	24 MiB (per socket)
Memory	128 GiB DDR3 1066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

- Isolation : “SERIALIZABLE”

Performance Evaluation (throughput)



Performance Evaluation (profiled)



Research results

- We published the results:
  - ***ACM SIGMOD'13***
  - ***IEEE ICDE'14***
  - ***ACM Transactions on Database Systems 2014***
- Researchers at ***HP Lab*** implemented our design and verified the effectiveness of our proposal in their ***VLDB'14*** paper.

Future research direction

- We have lots of research opportunities in system software designs that are still unable to fully utilize hardware resources.

- **Collaboration between software and hardware researchers can make a huge impact on this research area.**

Thank You !!