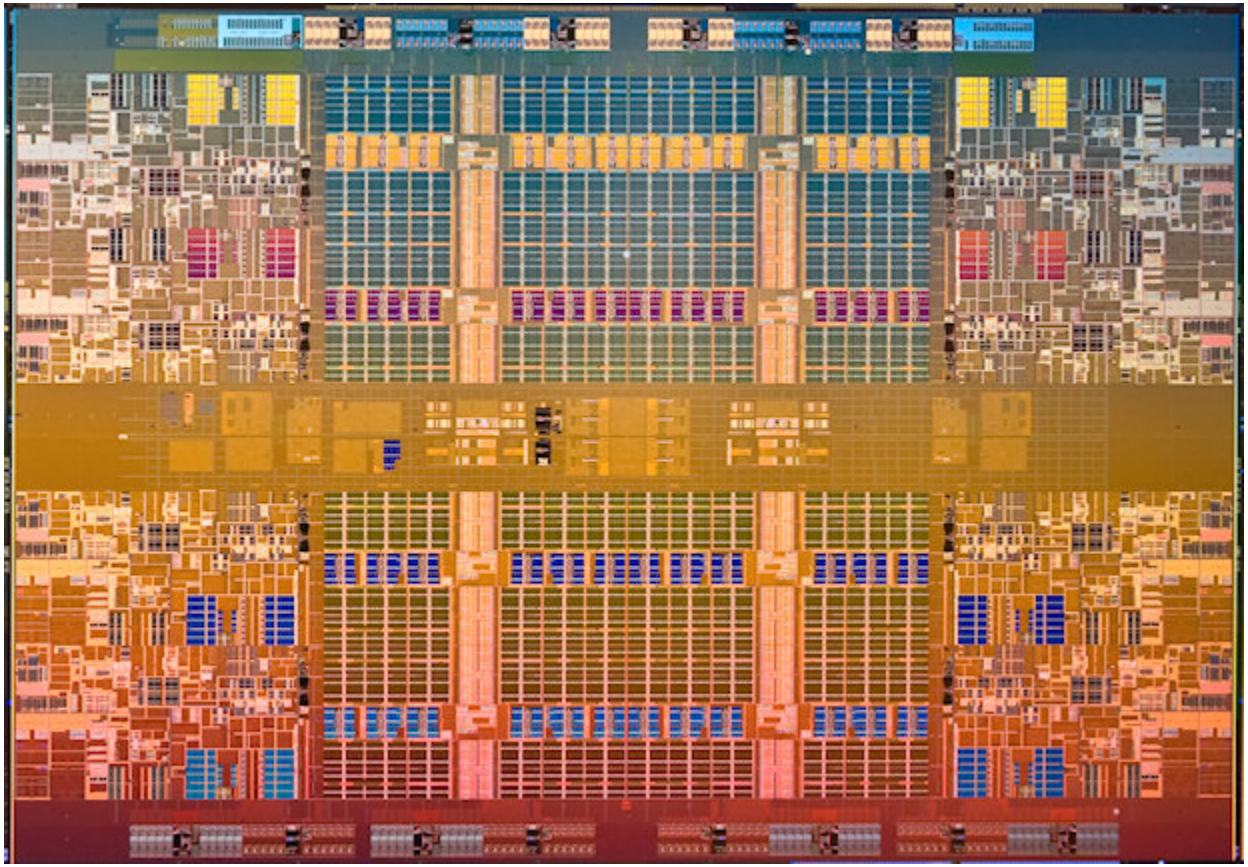
Multiprocessor: Scalable Locking

Zhaoguo Wang

Some slides adjusted from. Elsa L Gunter (UIUC), Jonathan Walpole (PSU)
Paul McKenney (IBM) Tom Hart (University of Toronto)

We are in Multi-core era



What is scalability?

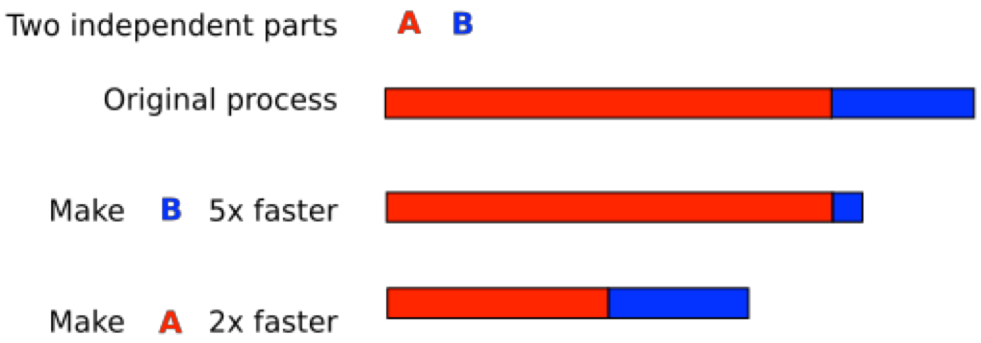
Application does N times as much work on N cores as it could on 1 core

Scalability may be limited by Amdahl's Law:

- Locks, shared data structures, ... Shared hardware (DRAM, NIC, ...)

Amdahl's Law

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



Locking

Why do kernels normally use locks?

Locks support a concurrent programming style based on mutual exclusion

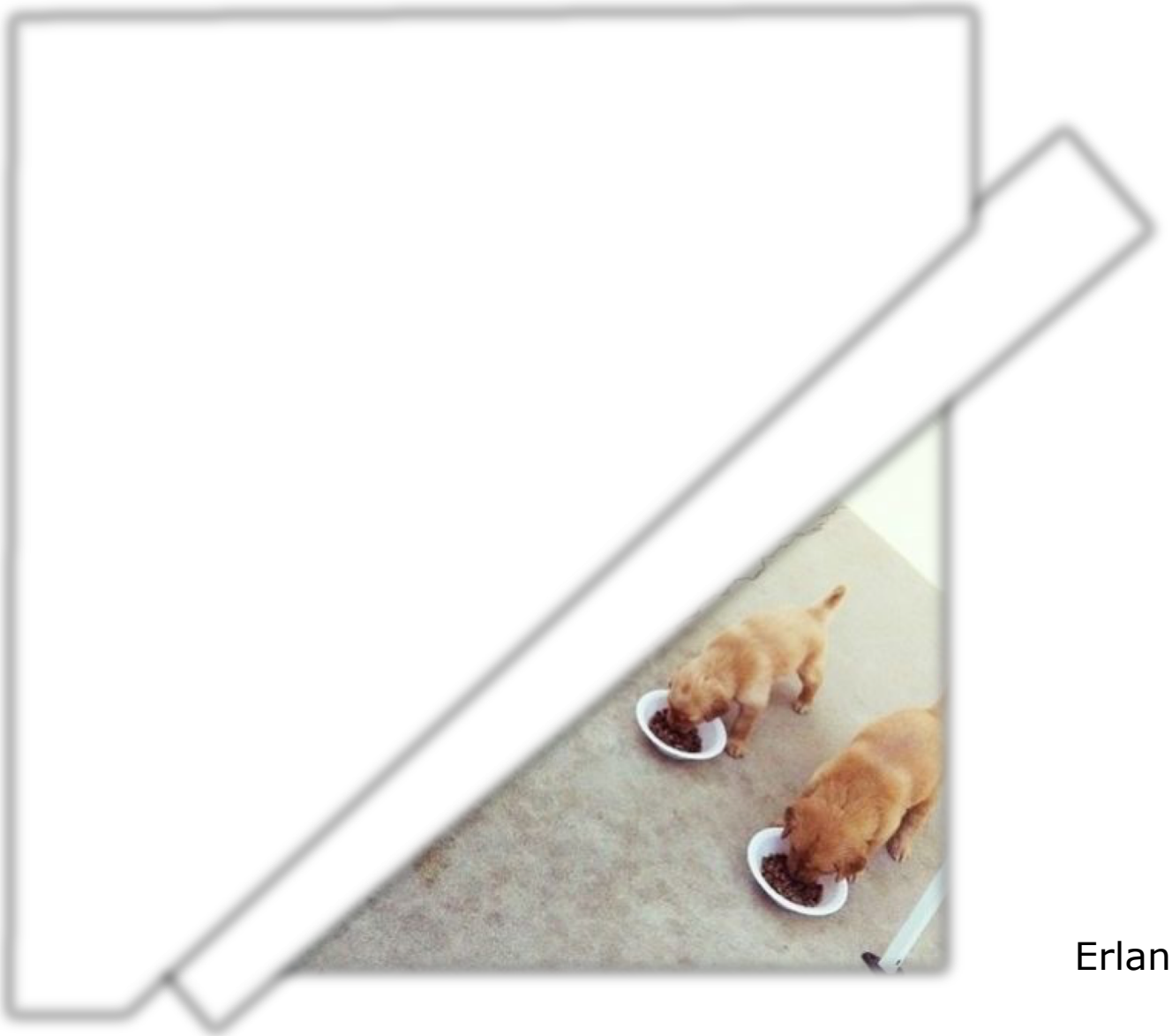
- Acquire lock on entry to critical sections
- Release lock on exit
- Block or spin if lock is held
- Only one thread at a time executes the critical section

Locks prevent concurrent access and enable sequential reasoning about critical section code

Ideal Multicore Scalability

Credit:
Erlang@Sina
Weibo

Ideal Multicore Scalability



Credit:
Erlang@Sina
Weibo

Ideal Multicore Scalability

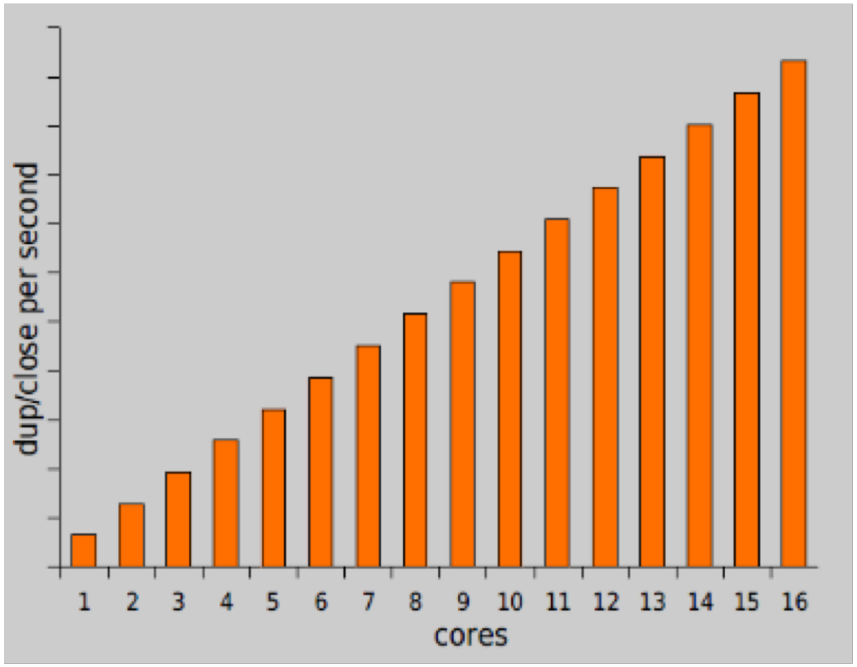
Credit:
Erlang@Sina
Weibo

Multicore Scalability in Reality

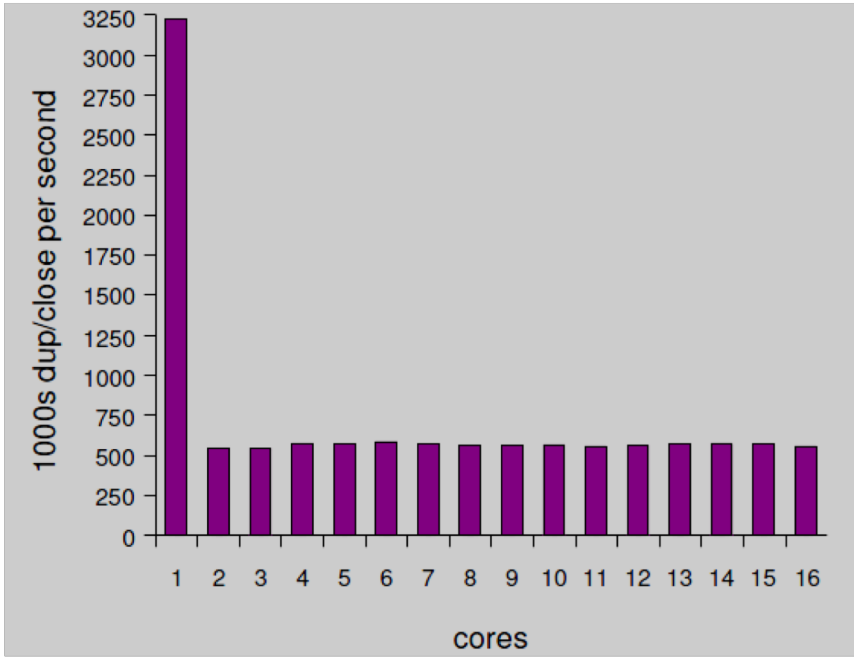
Credit:
Erlang@Sina
Weibo

Motivating example: file descriptors

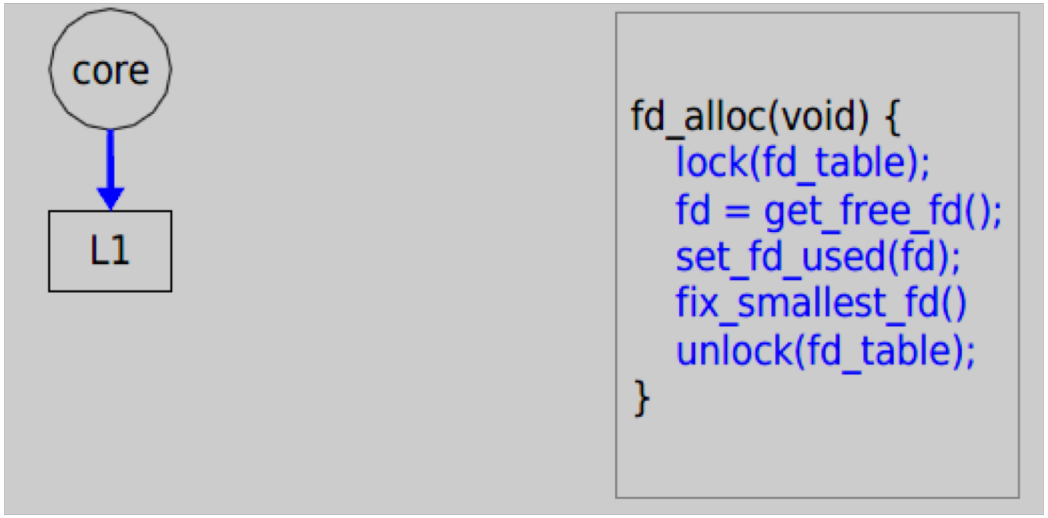
Ideal FD performance graph



Actual FD performance

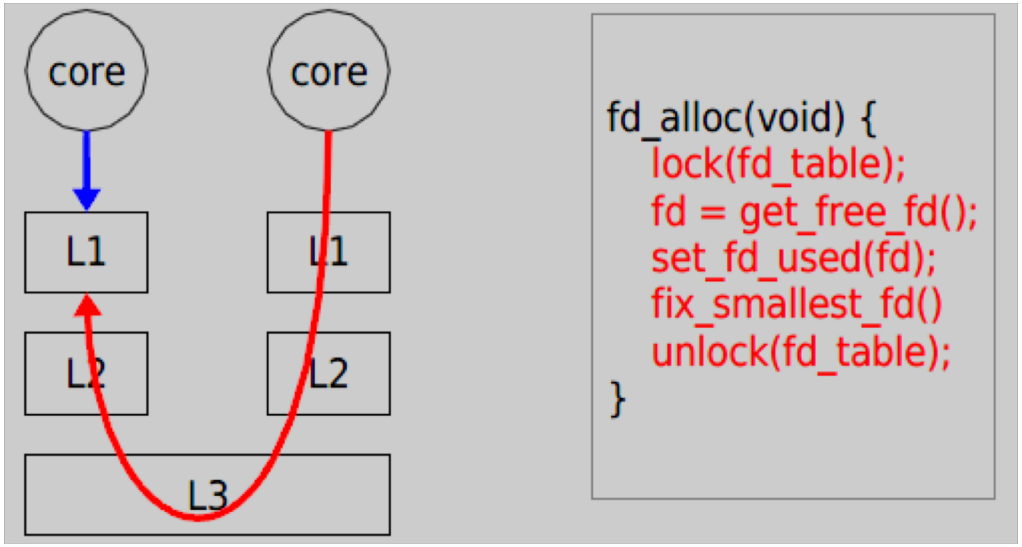


Recap: Why throughput drops?



Load fd_table data from L1 in 3 cycles.

Recap: Why throughput drops?



Now it takes 121 cycles!

Cause: Non-scalable locks

Non-scalable locks

- Such as spin locks
- Poor performance when highly contended

Many systems are using non-scalable locks

But they are dangerous

Why they are dangerous

Lead to performance collapse when adding a few more cores

Even tiny critical section will also lead to this performance collapse

Case study: ticket spinlock

Normal spinlock has extremely noticeable unfairness

- Even on a 8 core CPU

Ticket spinlock guarantees locks are granted to acquirers in order

- Used in Linux kernel

But it’s non scalable lock

Pseudo code for ticket lock

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}

void spin_lock(spinlock_t *l) {
    int t = atomic_xadd(&l->next_ticket);
    while (t != lock->current_ticket)
        ; // spin
}

void spin_unlock(spinlock_t *l) {
    l->current_ticket++;
}
```

Currently serving which one

Ticket for the next comer

Add and get original value

Benchmark setup

Hardware

- 48 core (8 x 6-core 2.4GHz AMD Opteron)

OS

- Linux 2.6.39

Warm file system cache

- Do not involve disk I/O during benchmarking

Benchmark: FOPS

Create a single file

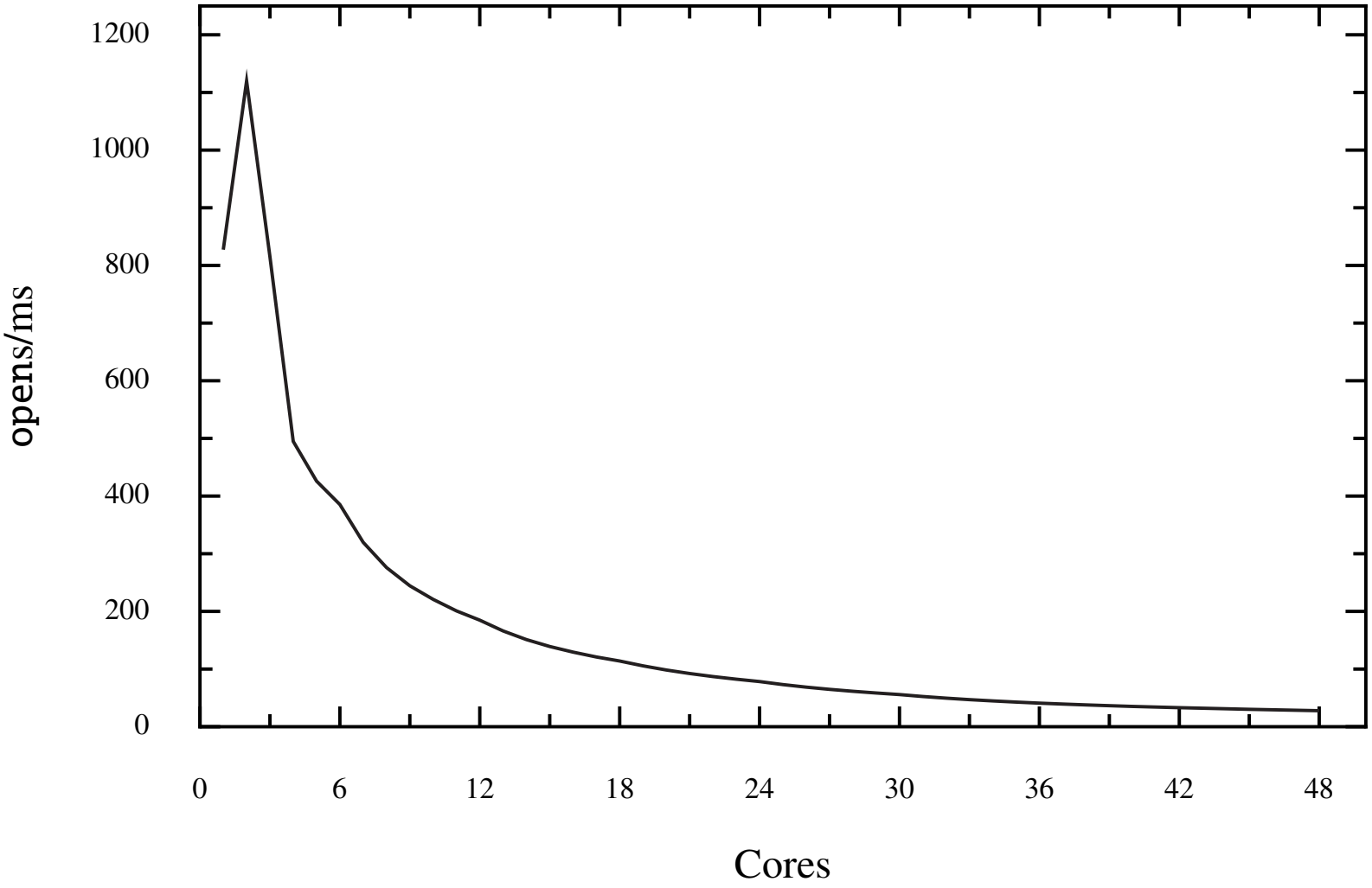
Starts one process on each core

Each process repeatedly opens and closes the file

Lock in kernel

- per-entry lock in the FS name/inode cache
- when closing a file, acquire the lock, decrease ref cnt

Benchmark: FOPS



Benchmark: MEMPOP

Create one process on each core

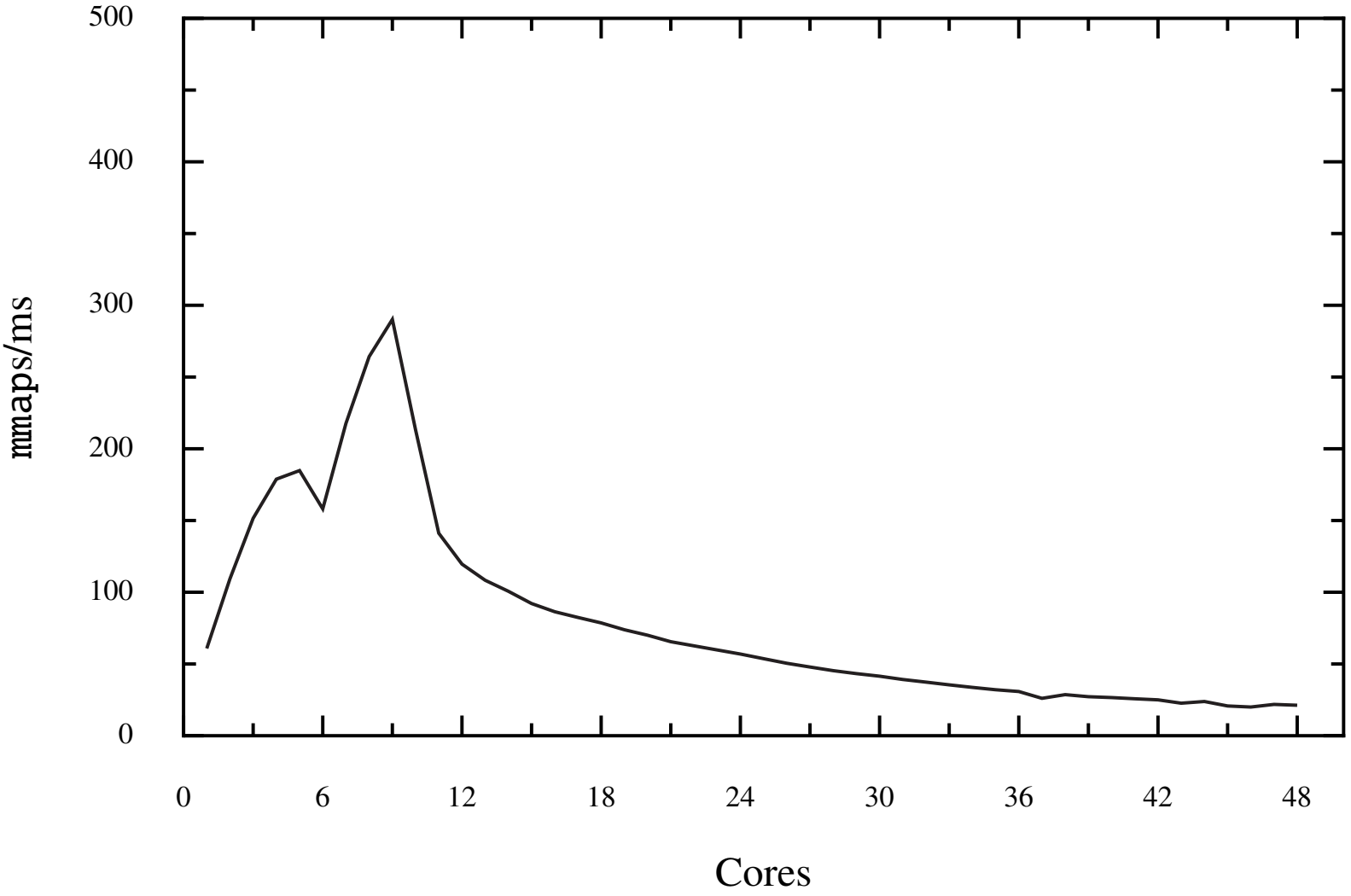
Repeat the following

- `mmap` 64KB of memory with `MAP_POPULATE` flag
  - Tell the kernel to allocate page and insert into page table immediately
- `munmap` the memory

Lock in kernel

- protects the data structure mapping physical pages to virtual memory regions

Benchmark: MEMPOP



Benchmark: PFIND

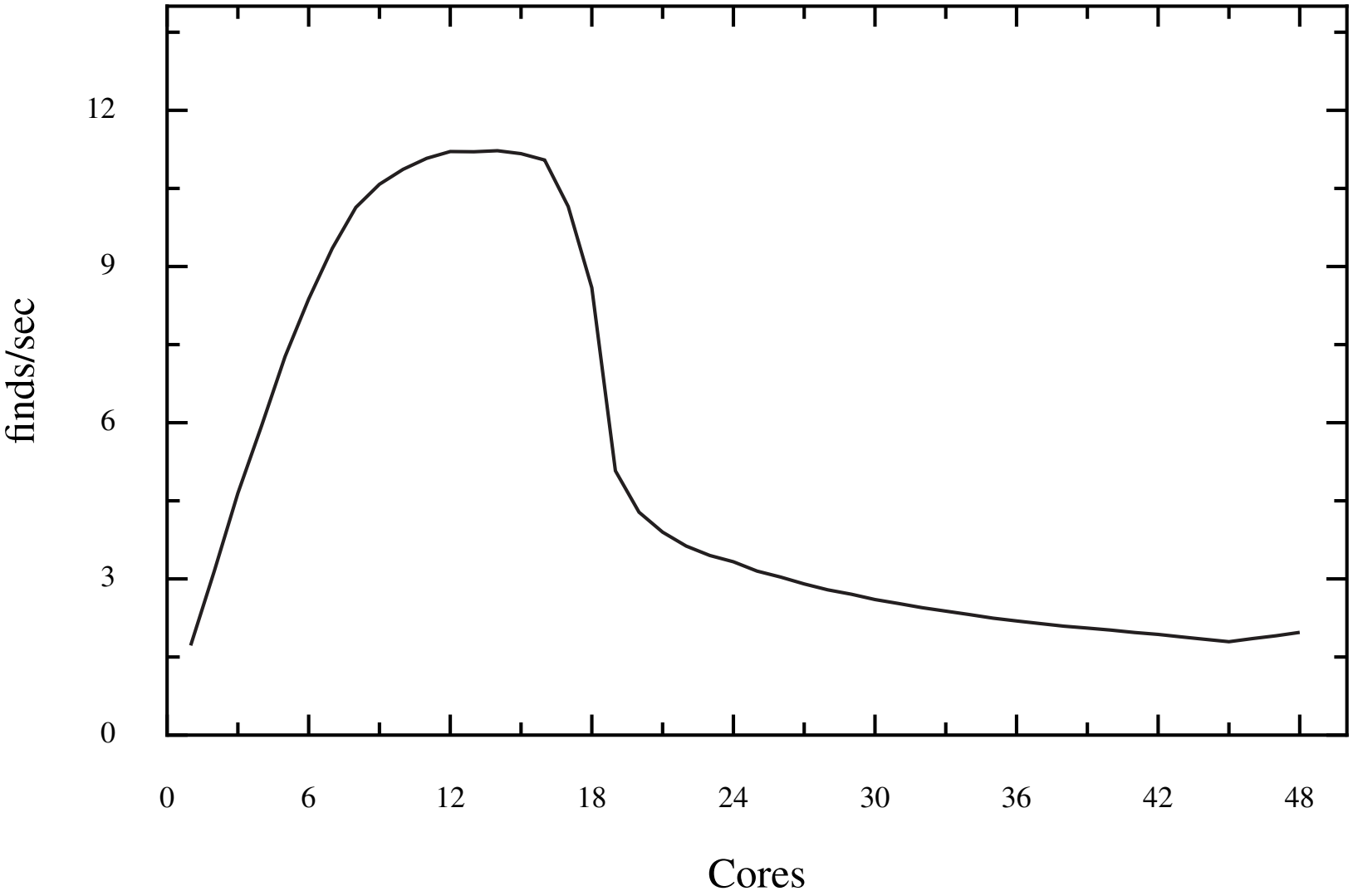
Search file in a directory

- evenly divide directories in the search directory as per-core inputs
- search the file using find command on each subdirectory simultaneously
- Input directory is balanced

Lock in kernel

- lock protecting block buffer cache

Benchmark: PFIND



Benchmark: EXIM

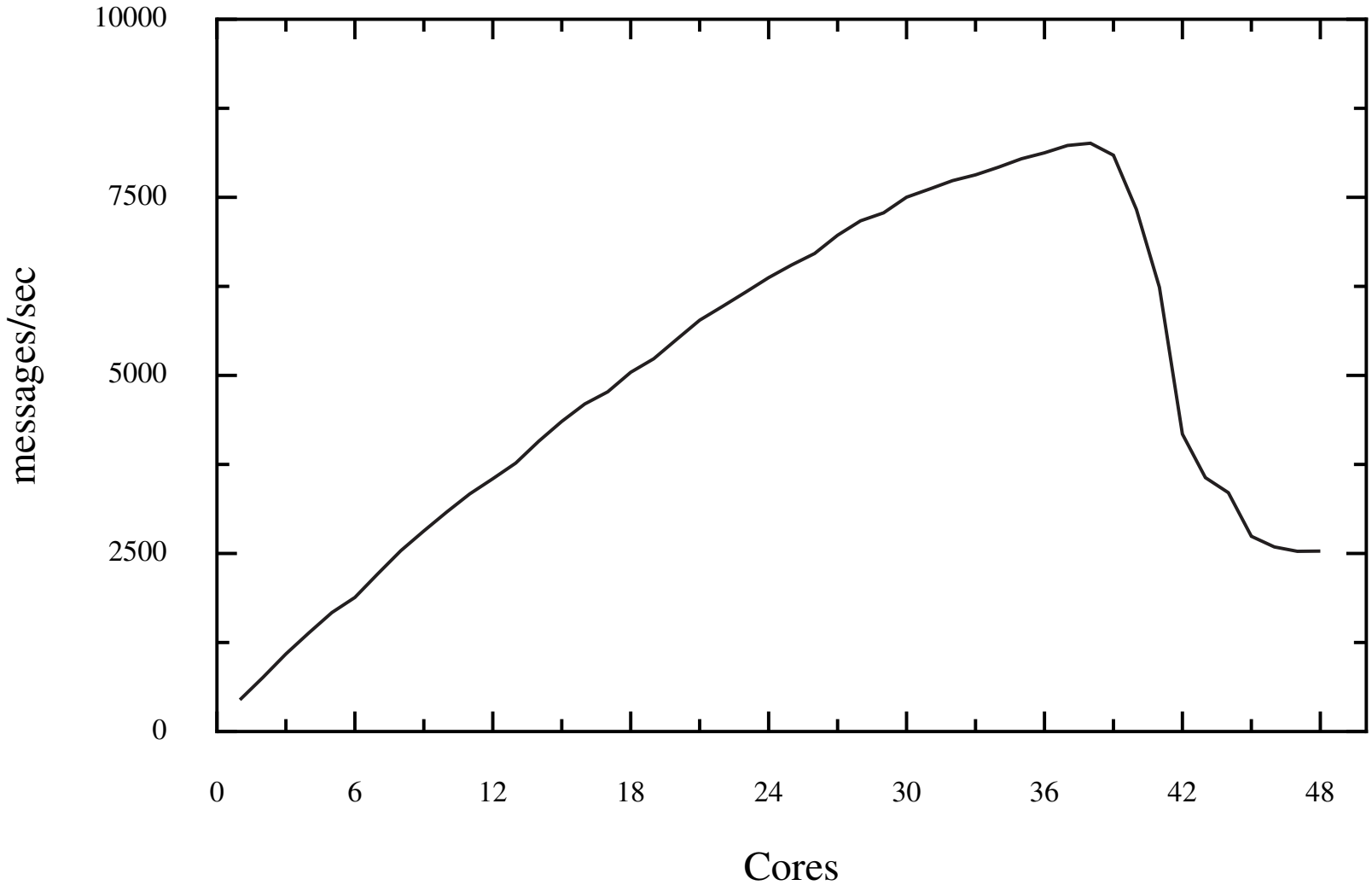
A commonly used mail server on Unix system

- Single master process listening for incoming SMTP connections
- Fork a new process for each connection and handles the incoming message

Lock in kernel

- protects the data structure mapping physical pages to virtual memory regions (same as MEMOP)

Benchmark: EXIM



Performance collapse

We expect to improve system performance by adding more CPU cores

But the performance actually becomes worse when we add more cores above the “threshold”

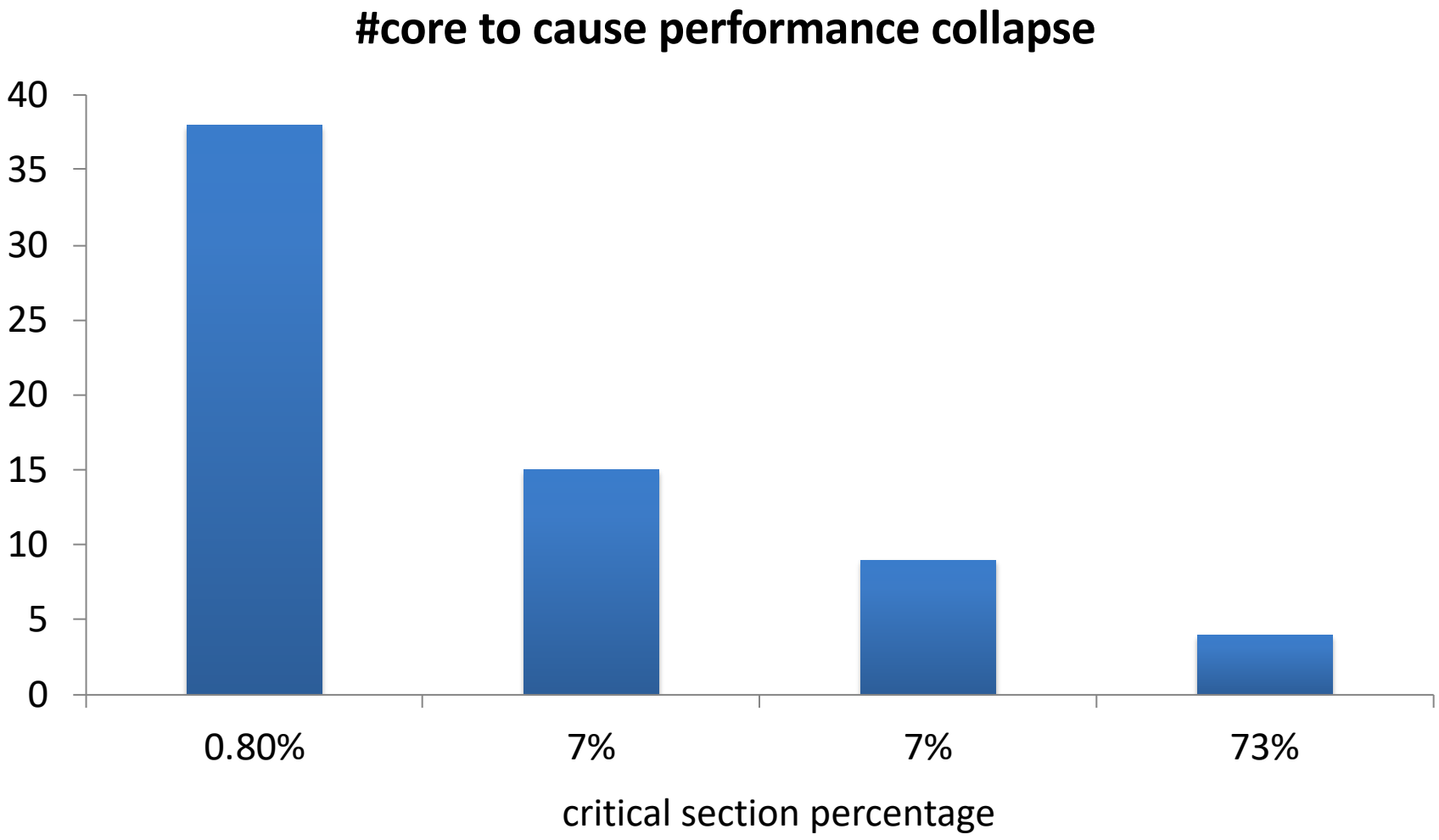
These are caused by critical section contention

Critical section details

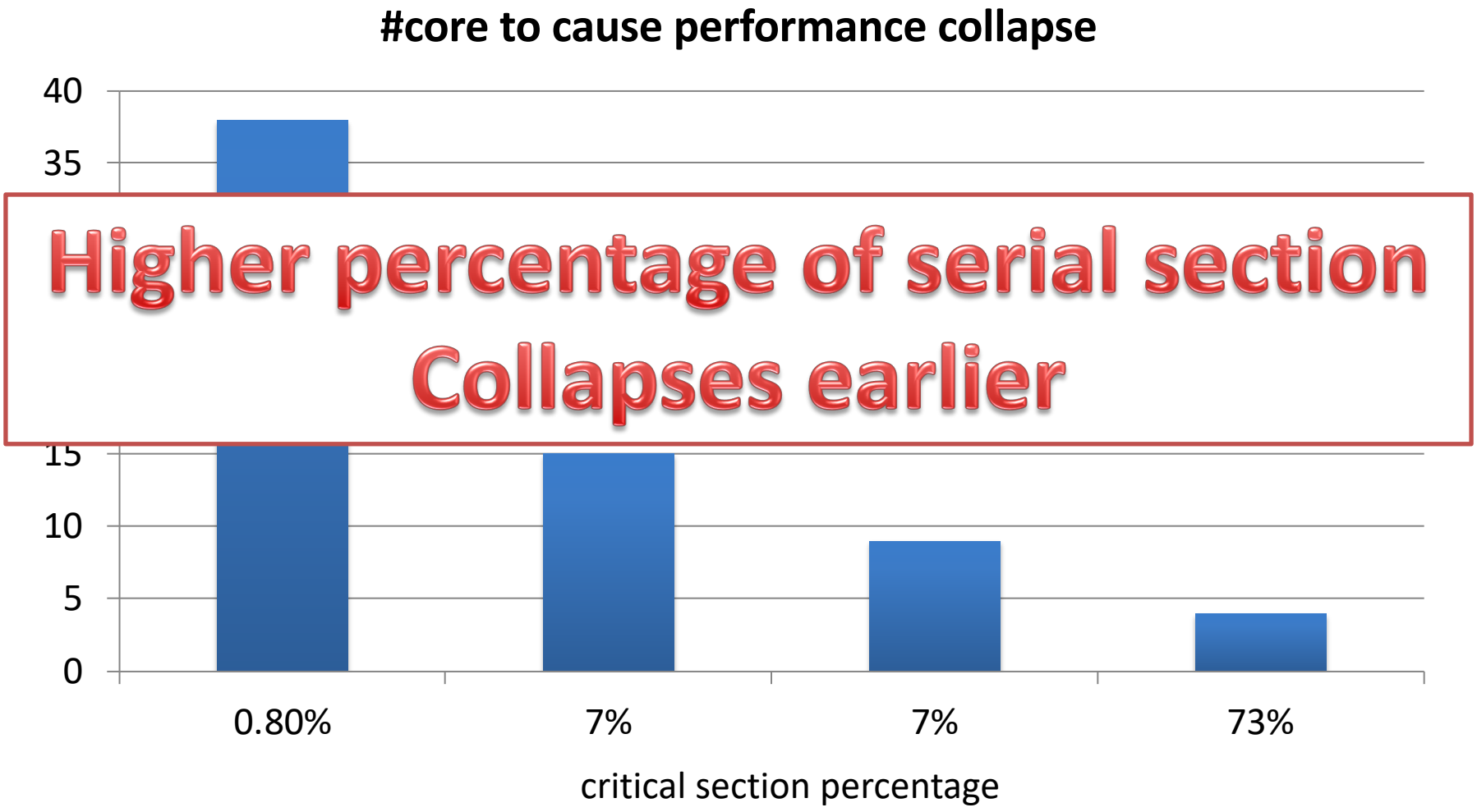
Single core measurement

Benchmark	cycle/op	#acquire/op	cycles in critical section	% in critical section
FOPS	503	4	92	73%
MEMPOP	6852	4	121	7%
PFIND	2099M	70K	350	7%
EXIM	1156K	58	165	0.8%

Critical section length and collapse



Critical section length and collapse



Higher percentage of serial section Collapses earlier

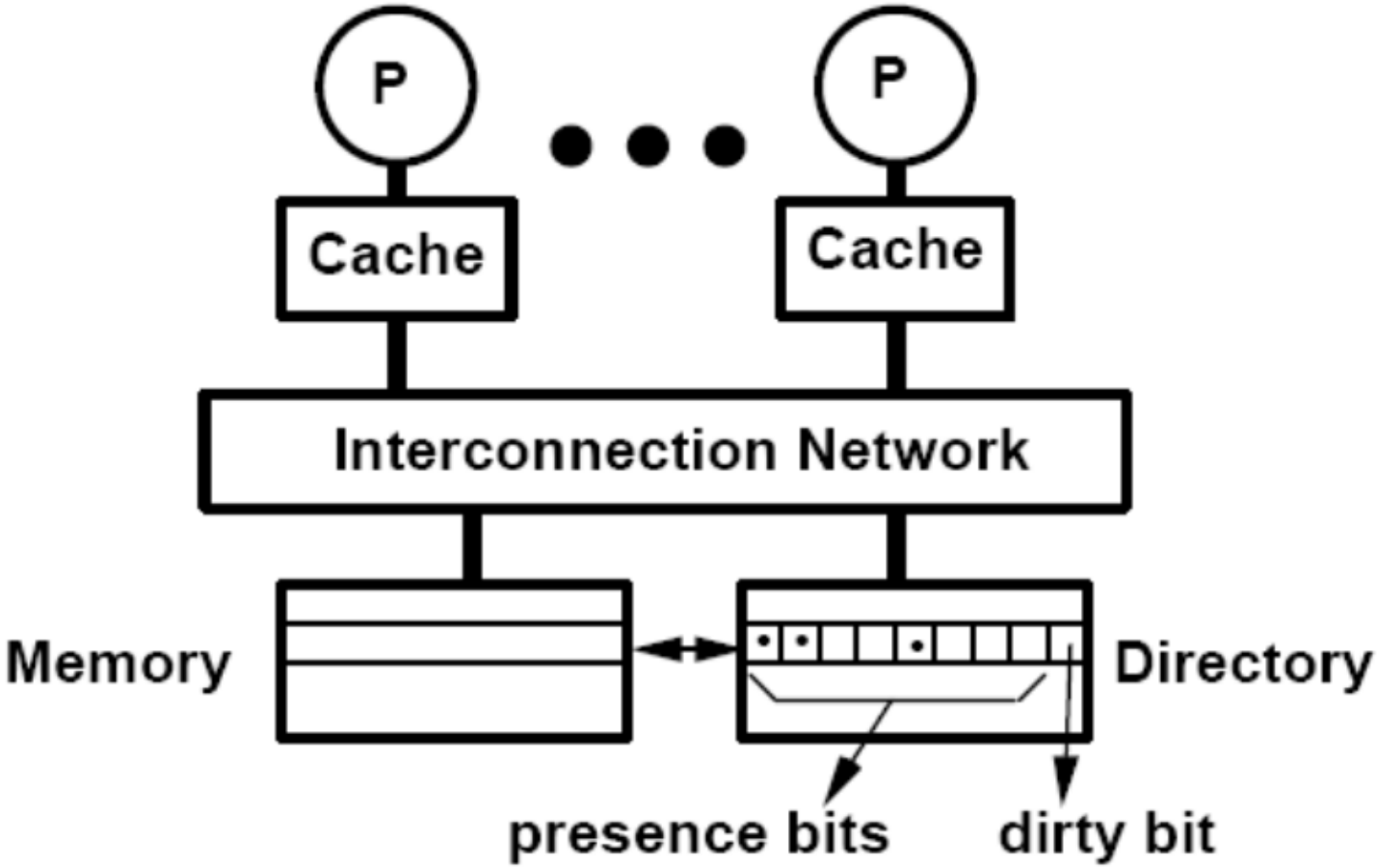
Background on cache coherence

On multi-core processors, each core has it's own private cache

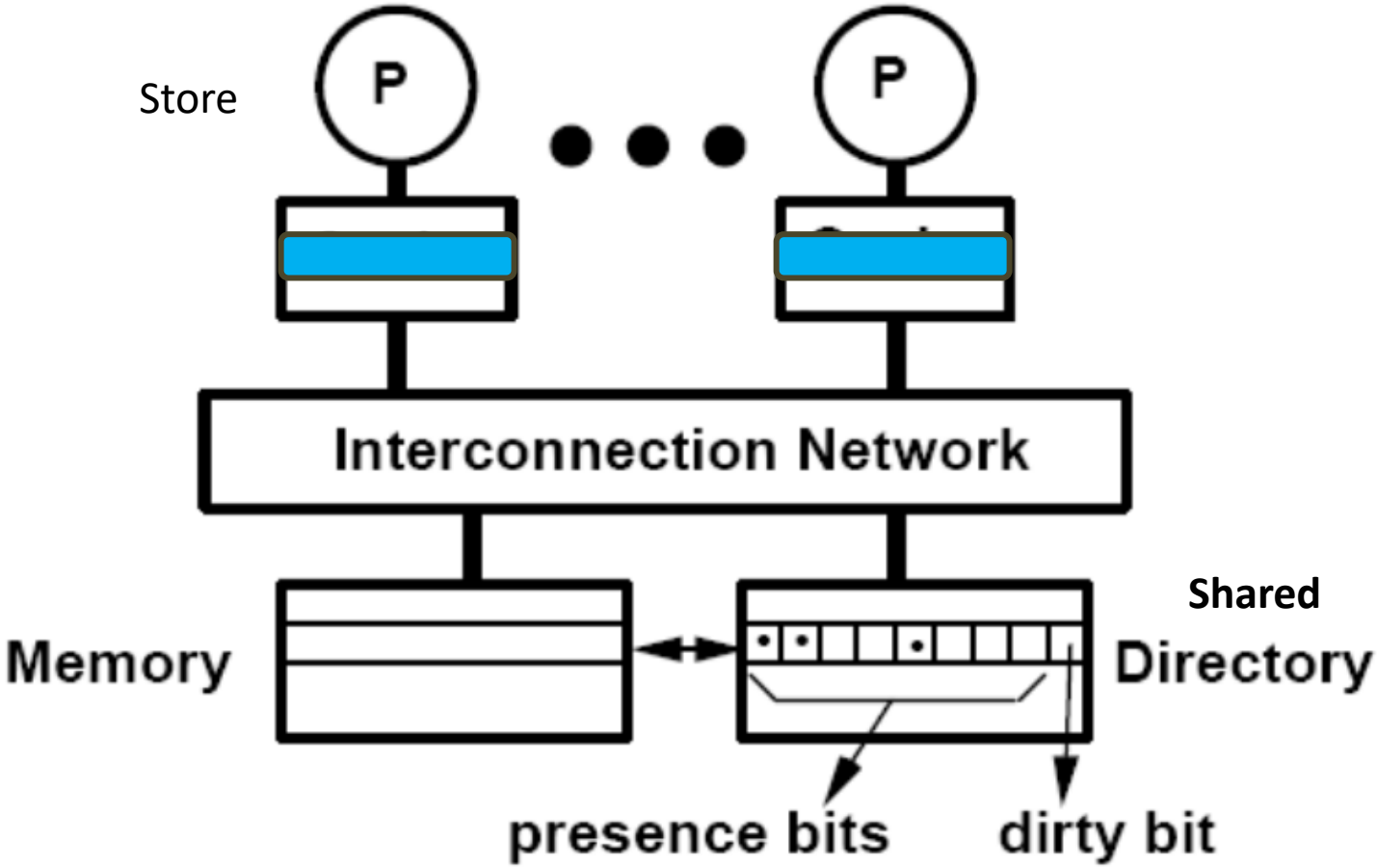
Need to keep the cache content in sync when some CPU modifies some memory

Accomplished by cache coherence protocol

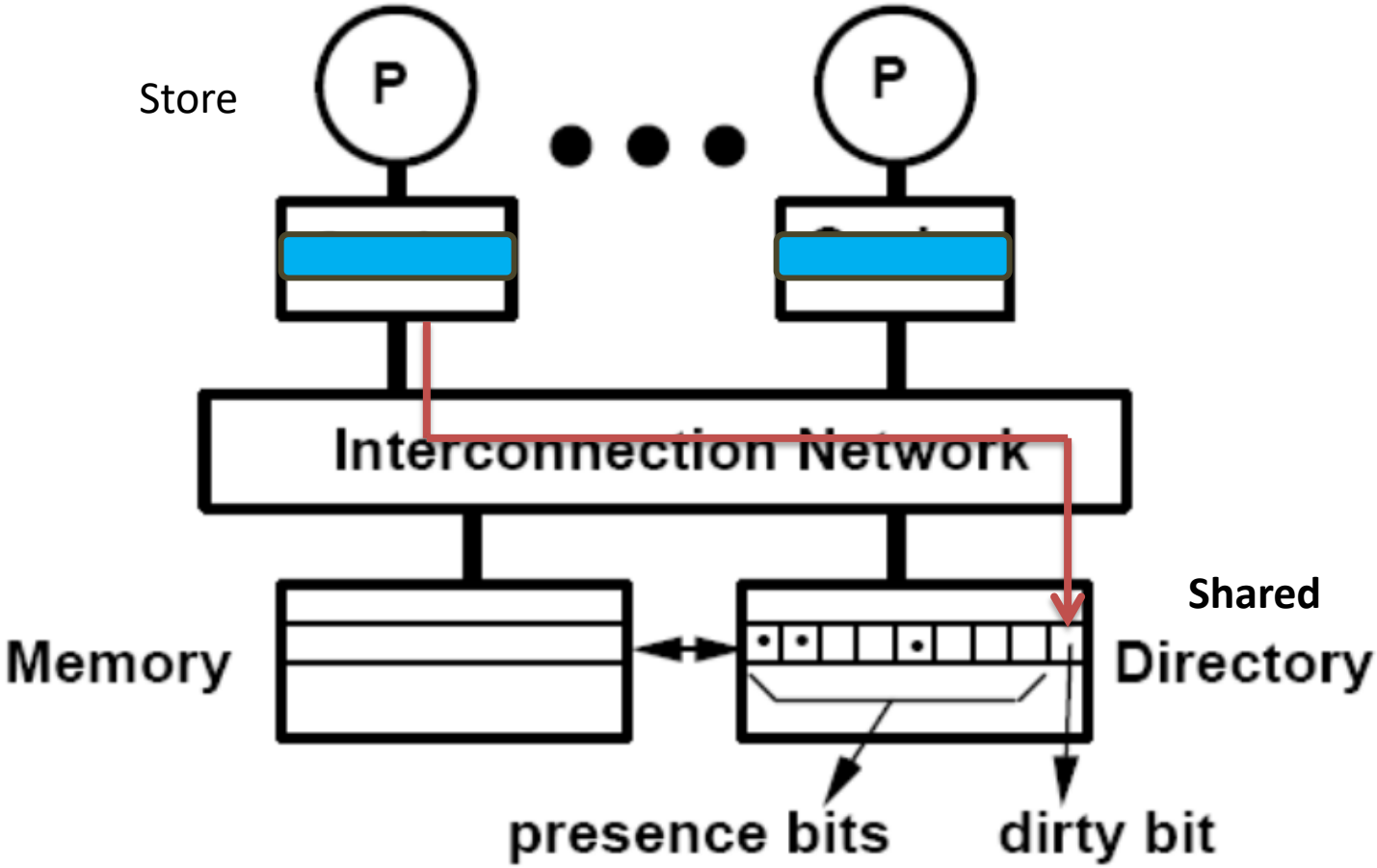
Directory-based cache coherence



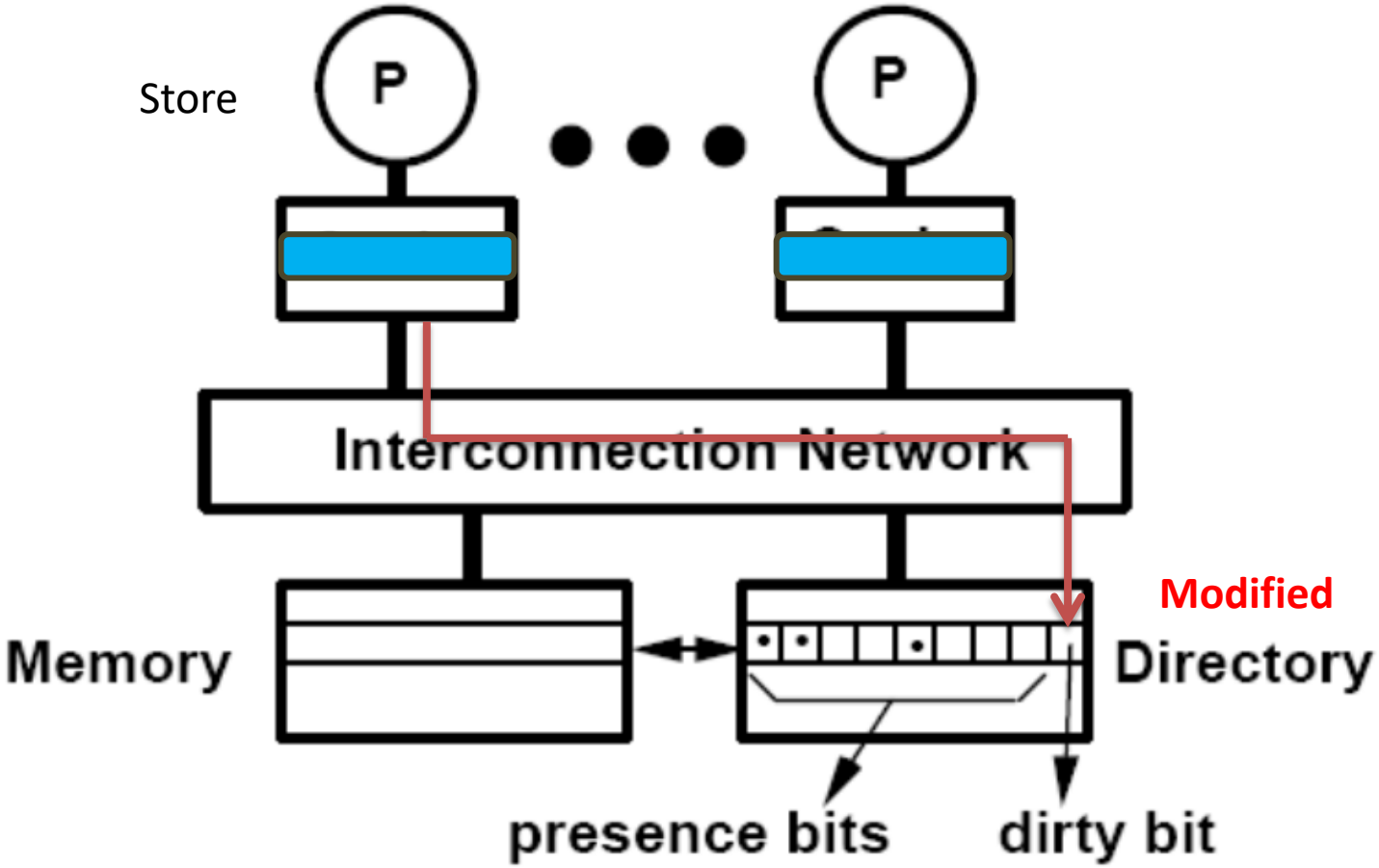
Directory-based cache coherence



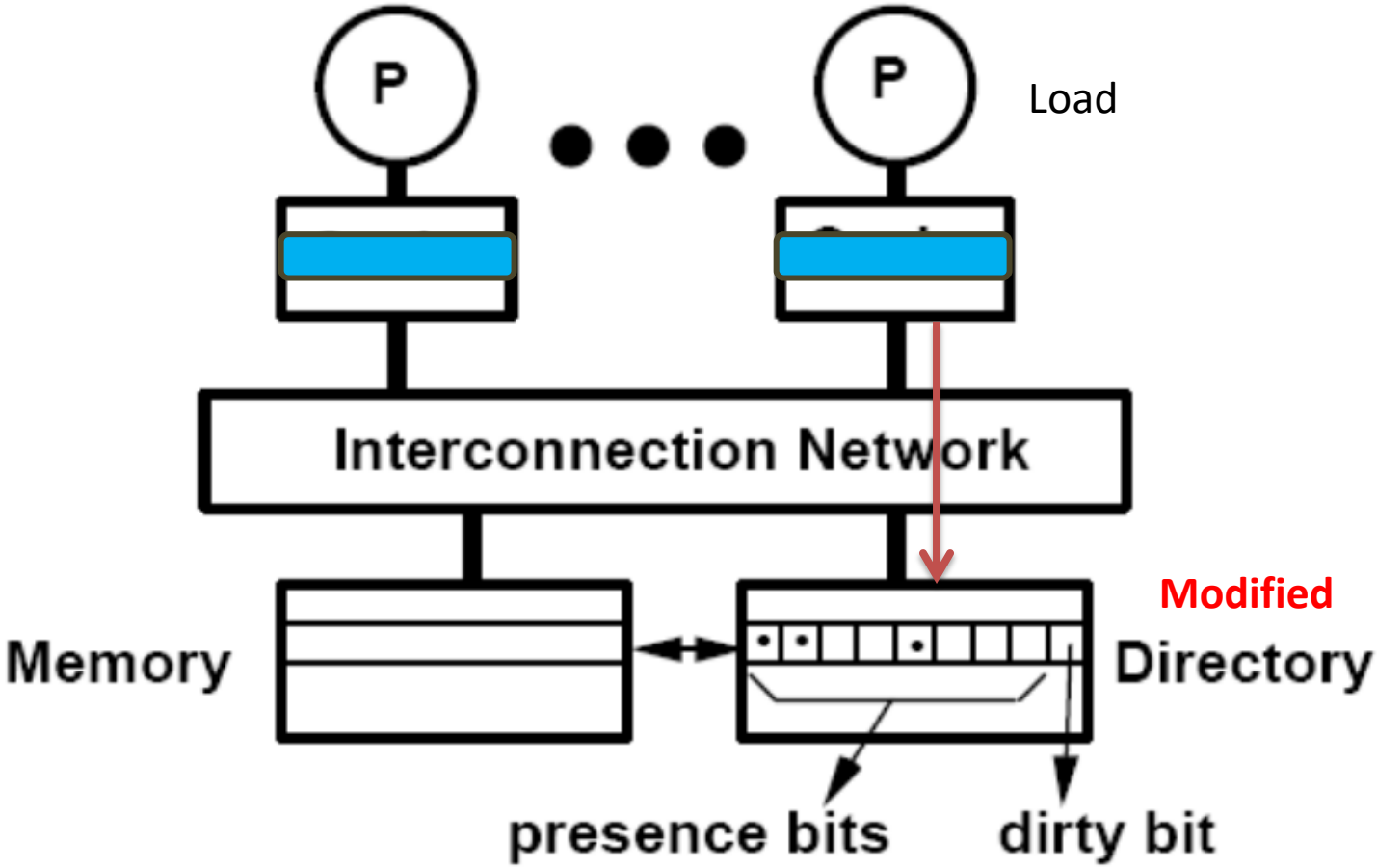
Directory-based cache coherence



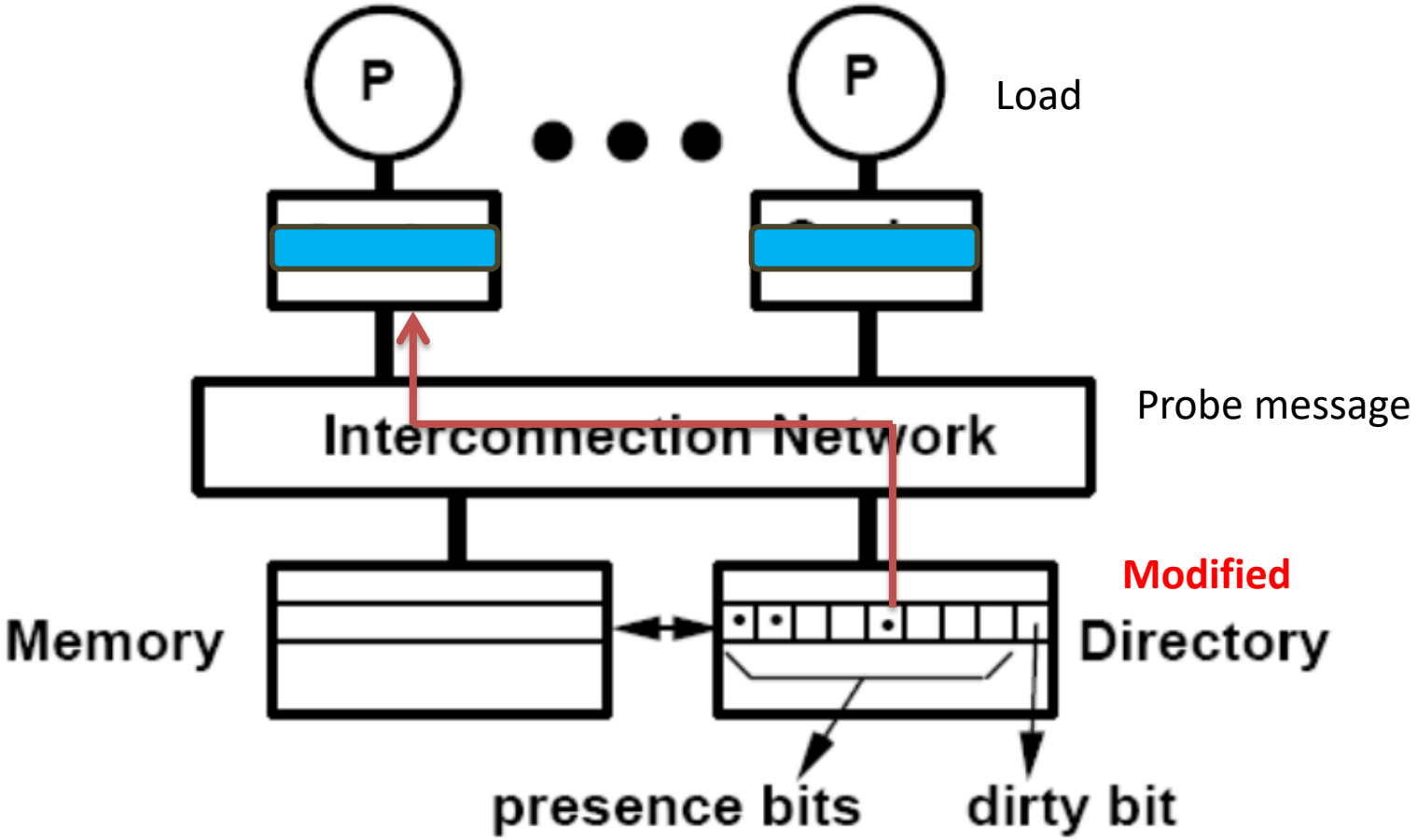
Directory-based cache coherence



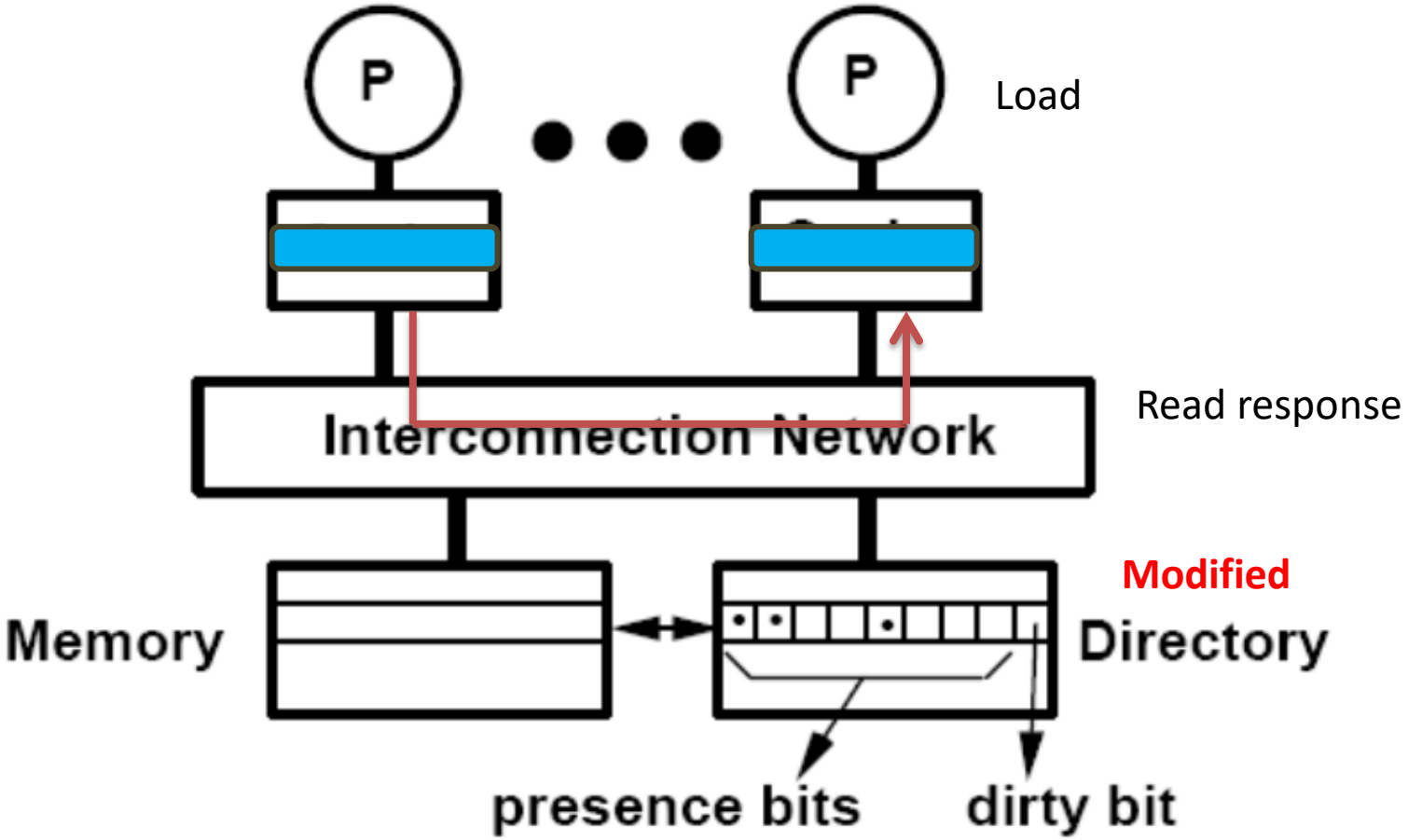
Directory-based cache coherence



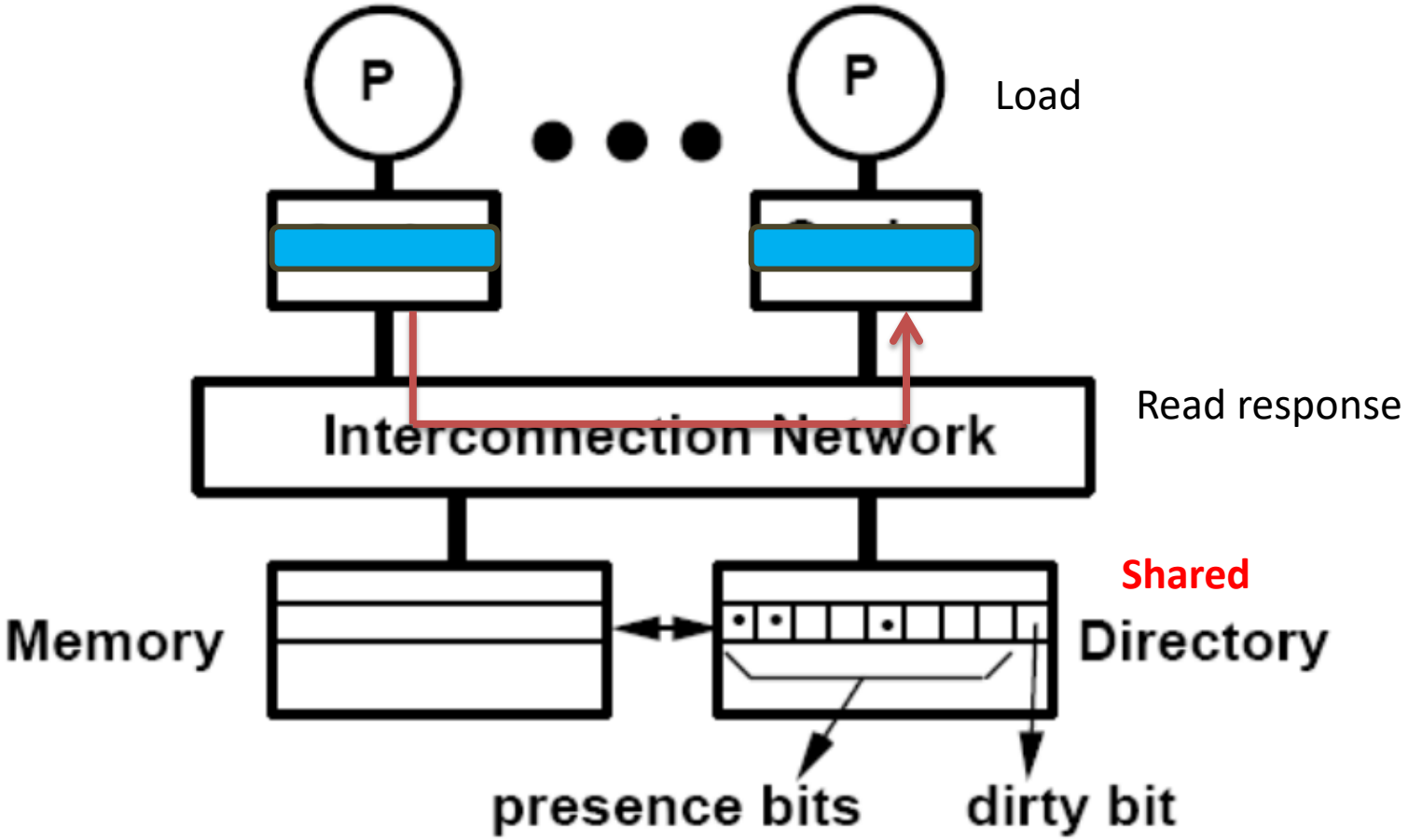
Directory-based cache coherence



Directory-based cache coherence



Directory-based cache coherence



A few notes cache coherence

There may be more than a single directory

- Especially for NUMA systems

Interconnect structure affects cache coherence performance

Directory is just one possible implementation

- Snooping is another commonly used approach

Intuition of the collapse

Key point: read with modified cache line have to get data back from the owner

- Coherence message are processed sequentially

Lock holder modifies cache holding the lock

Waiter is trying to read the lock

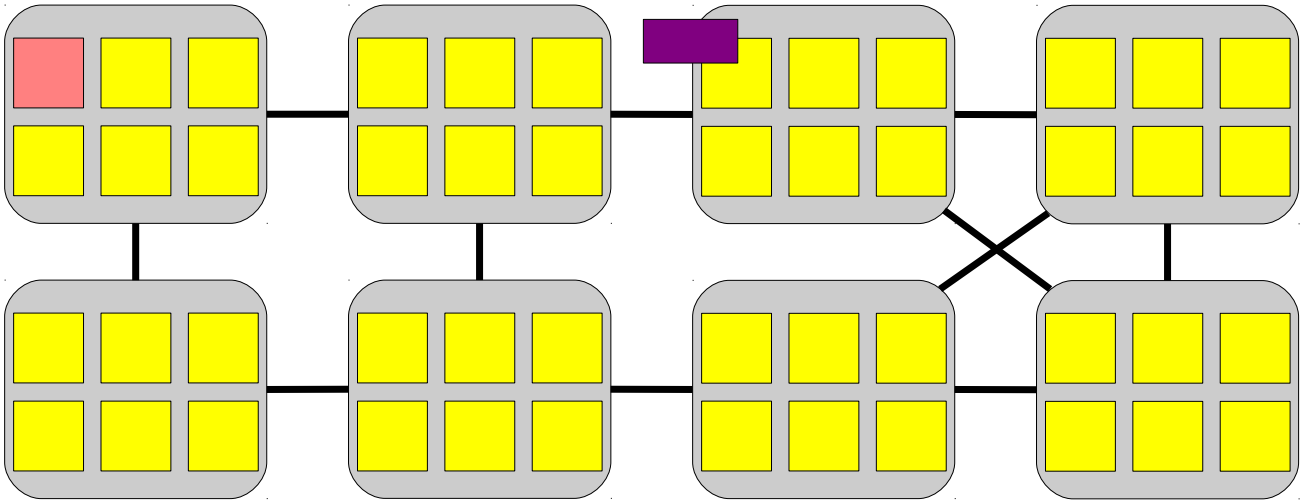
- They get value of the lock from the lock holder
- More readers means the next lock holder need to wait more time to get the lock

Allocate a ticket
read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



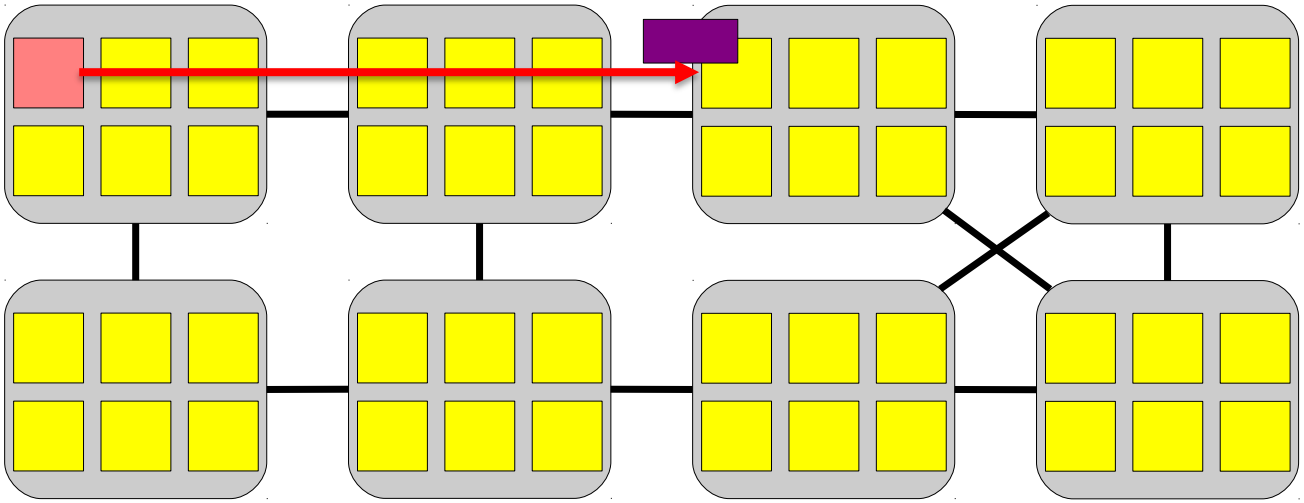
Allocate a ticket
read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

cache coherence message



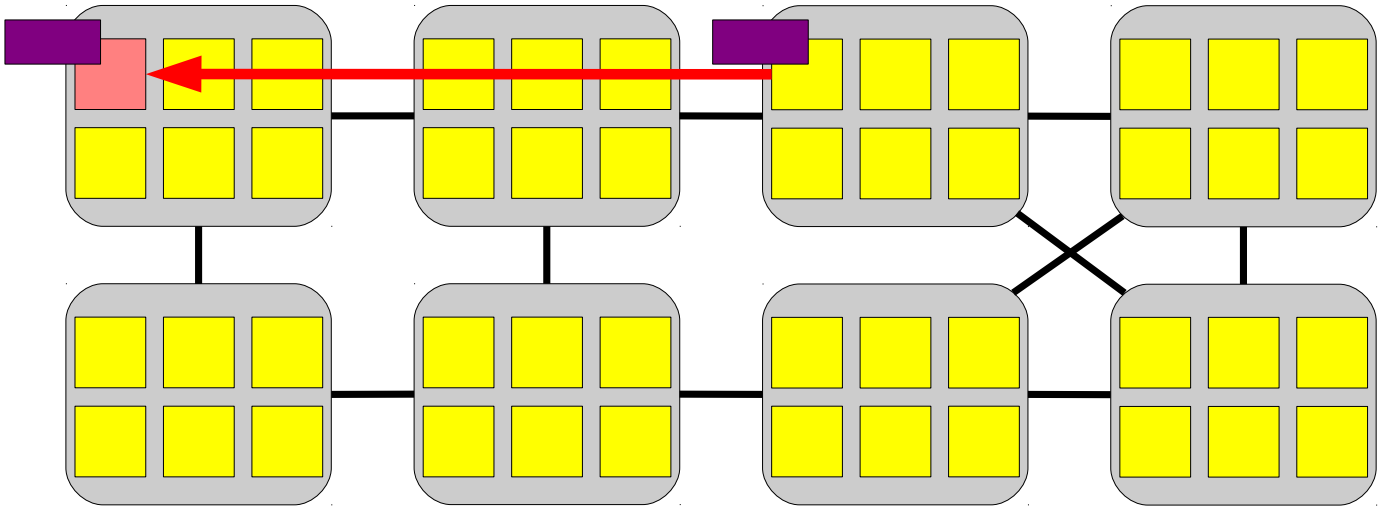
Allocate a ticket
read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

cache coherence message



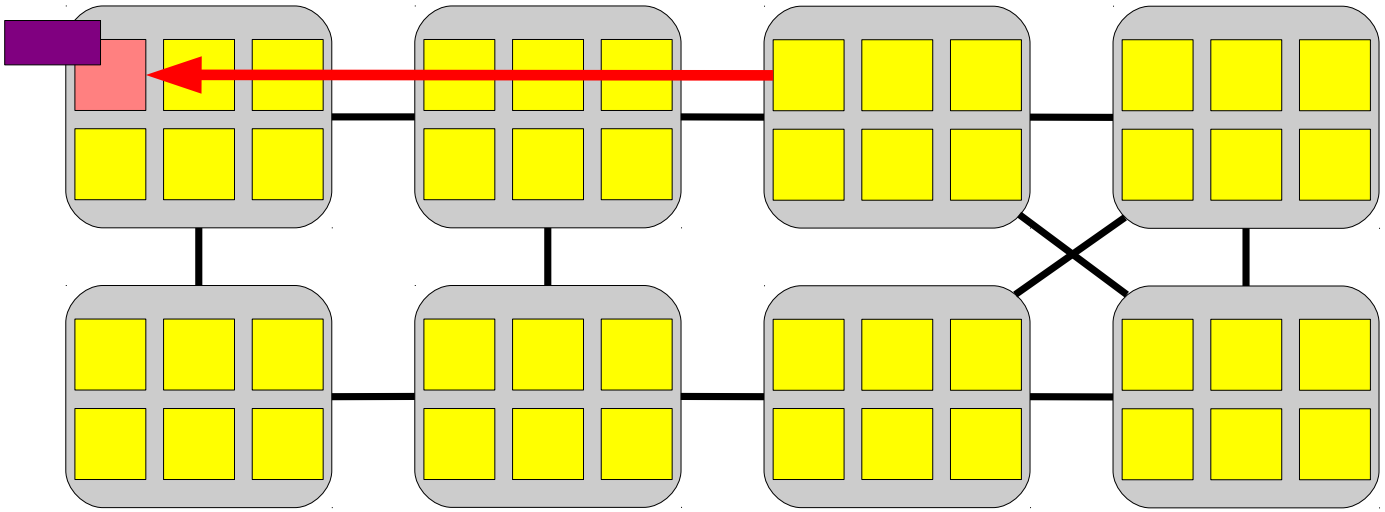
Allocate a ticket
read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

120 ~ 420 cycles

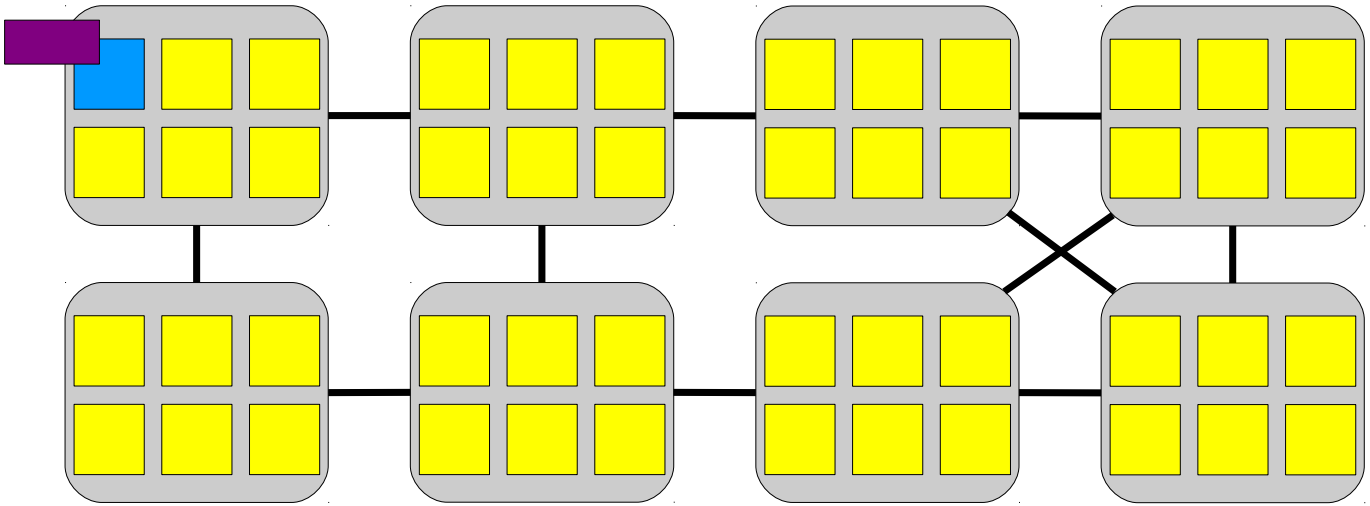


update the ticket value

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

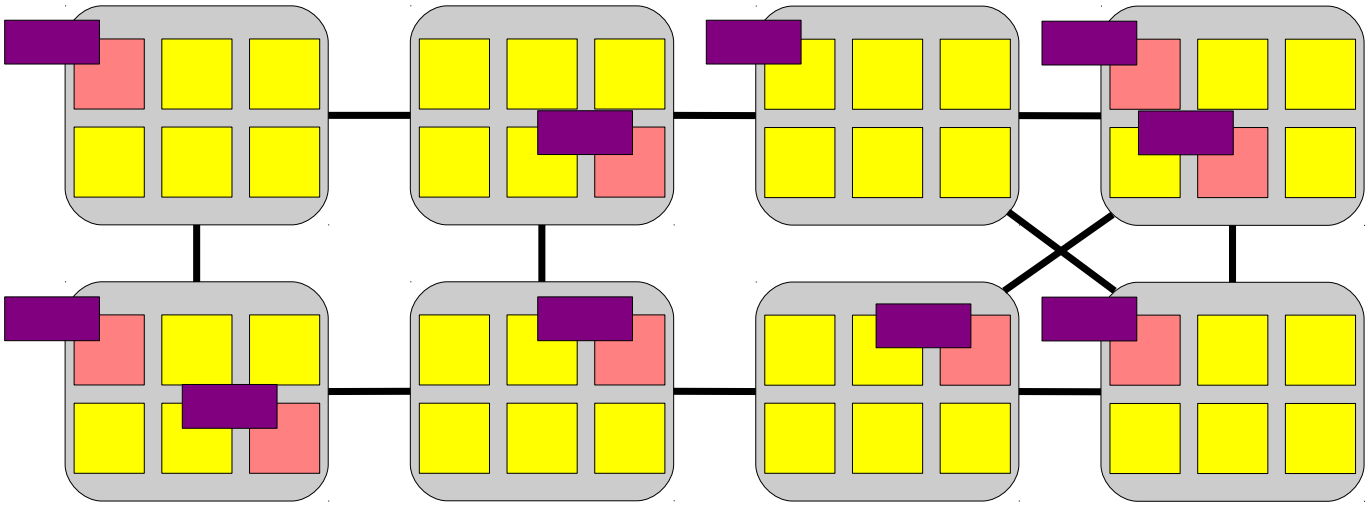
```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

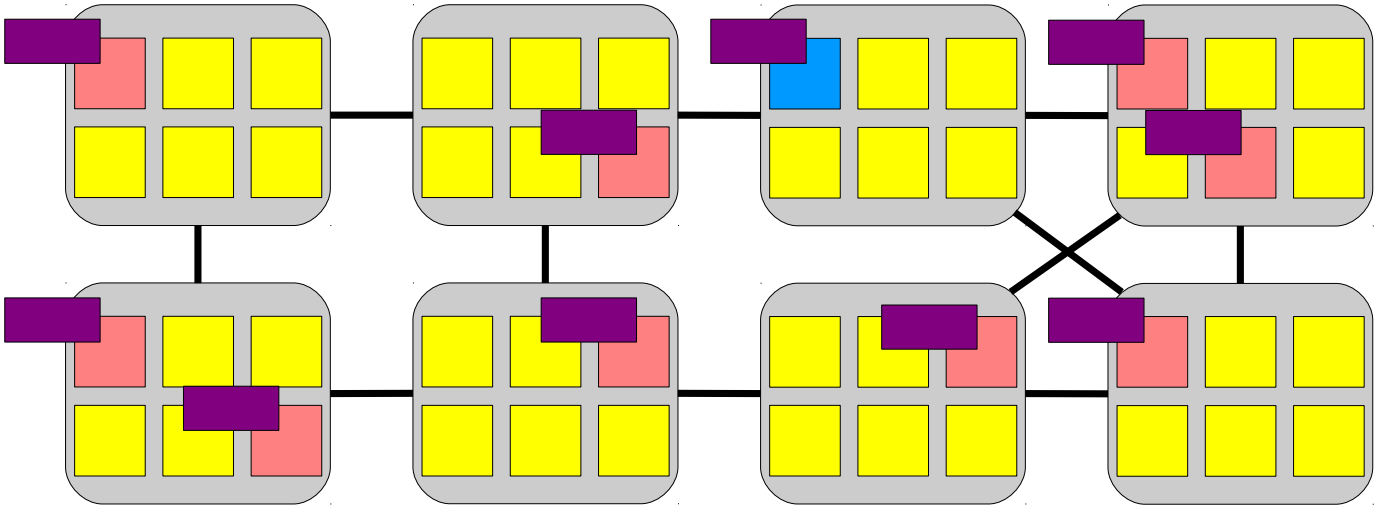


Lock shared by all core

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



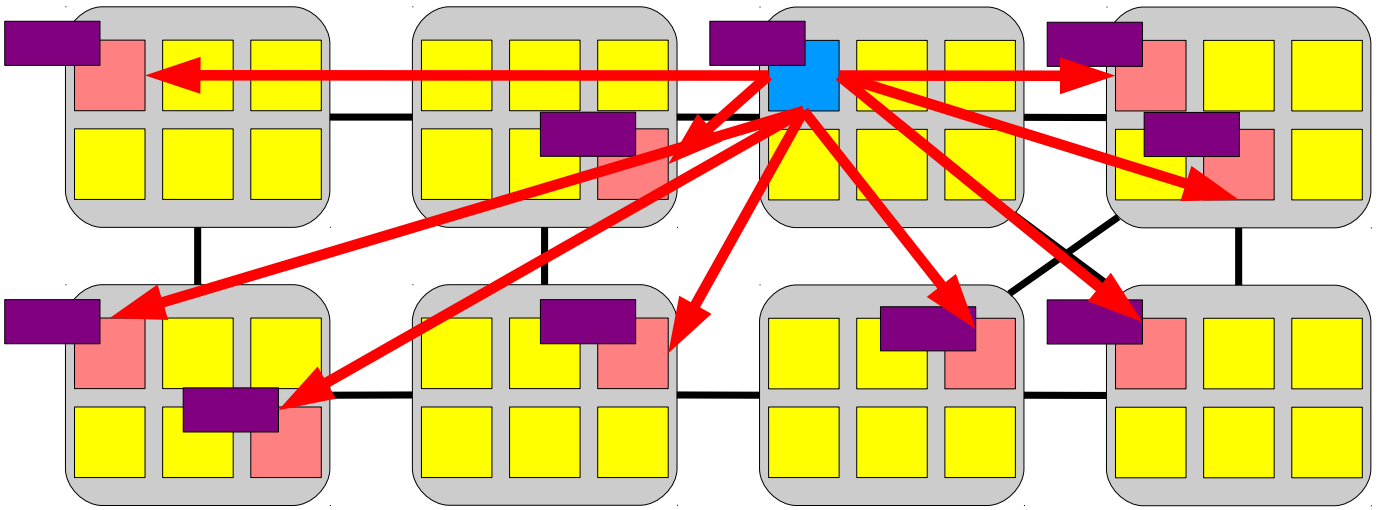
Lock holder update ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Invalidate message



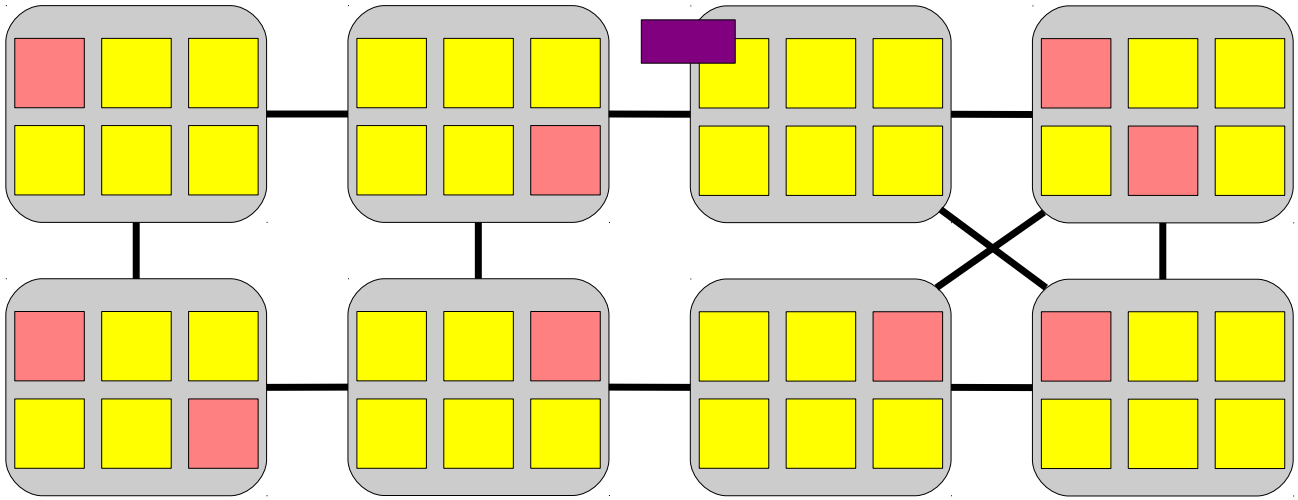
Lock holder update ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Only lock holder has lock in cache

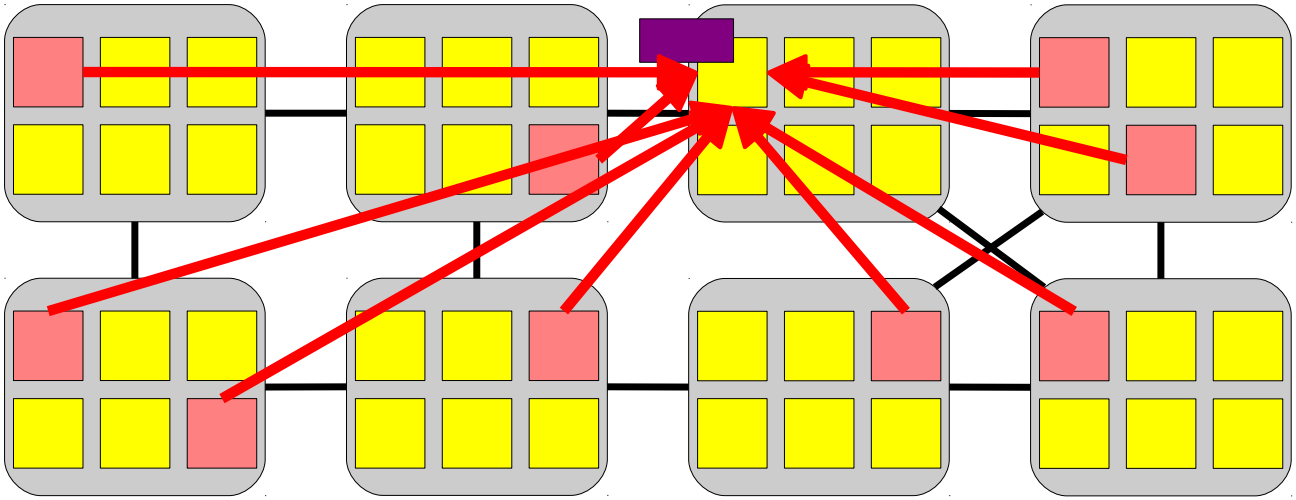


Lock holder update ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



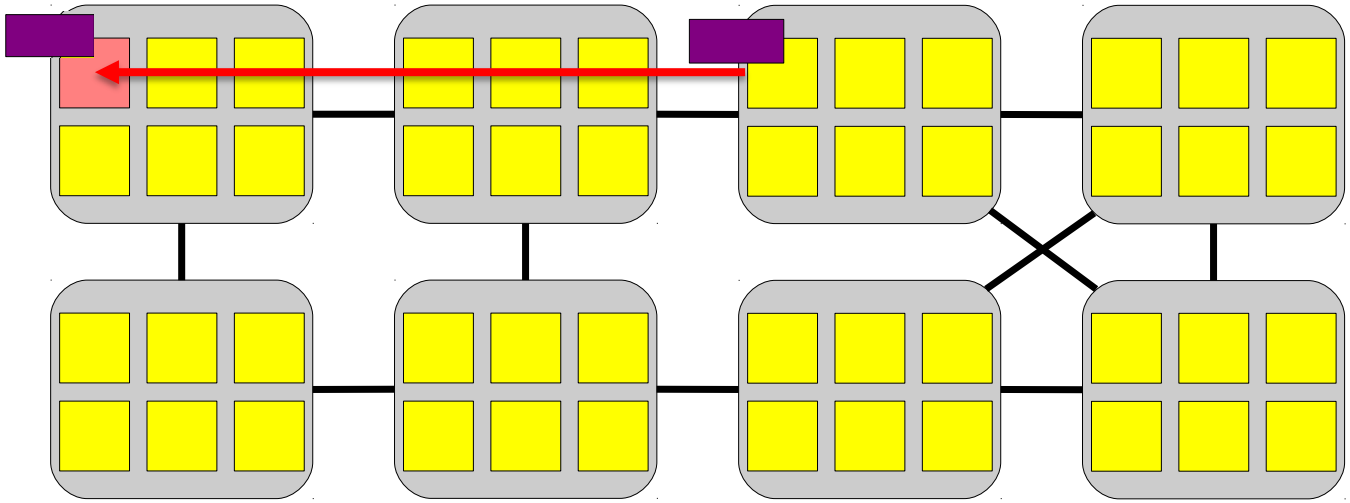
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

500 ~ 4000 cycles!



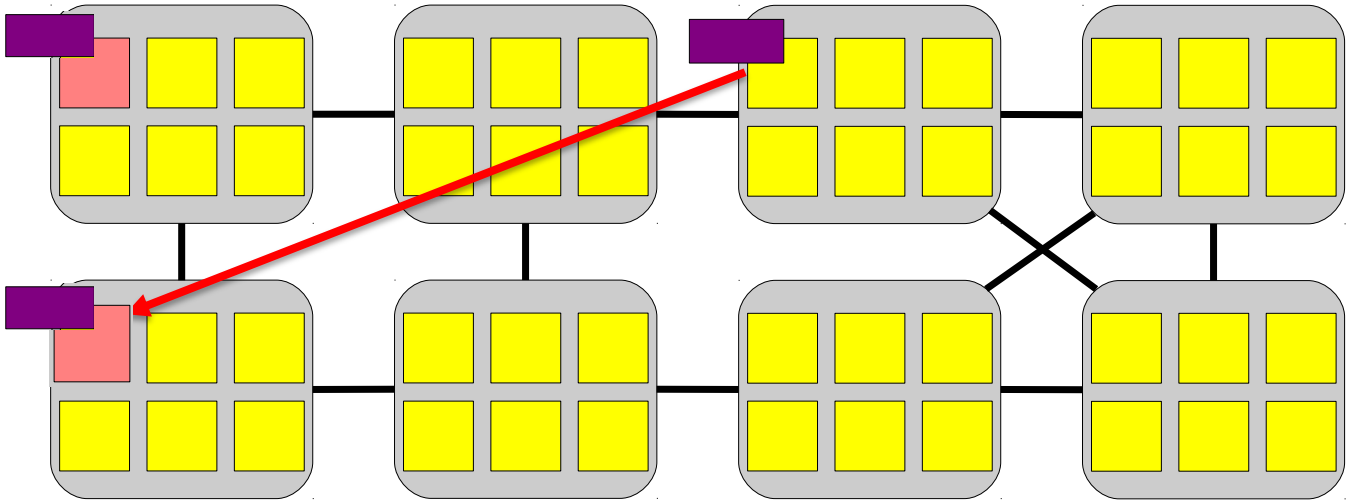
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Reply read request one by one



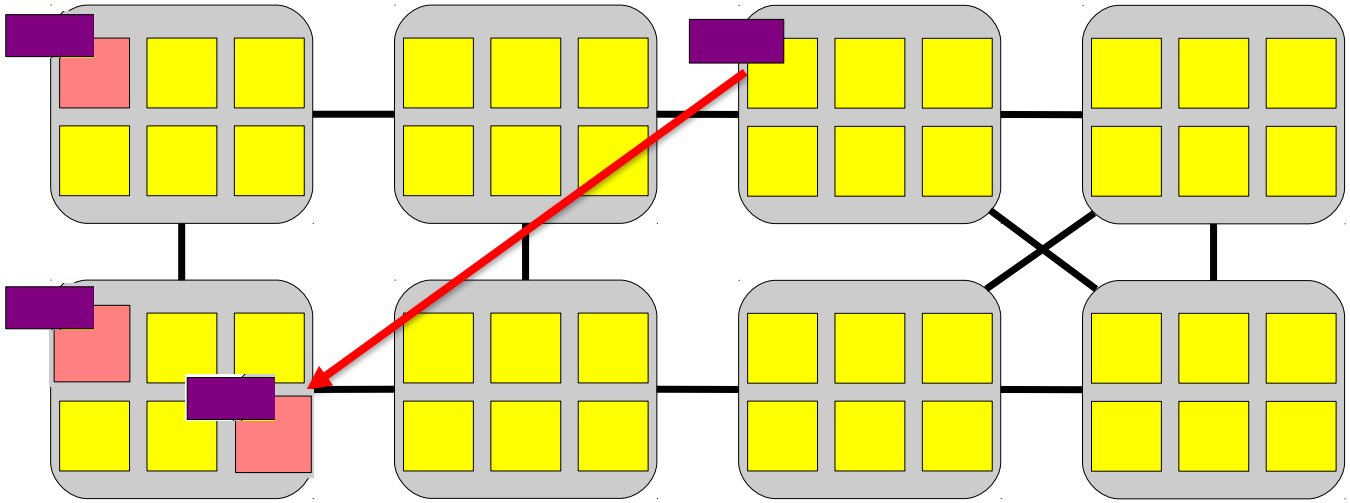
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Reply read request one by one



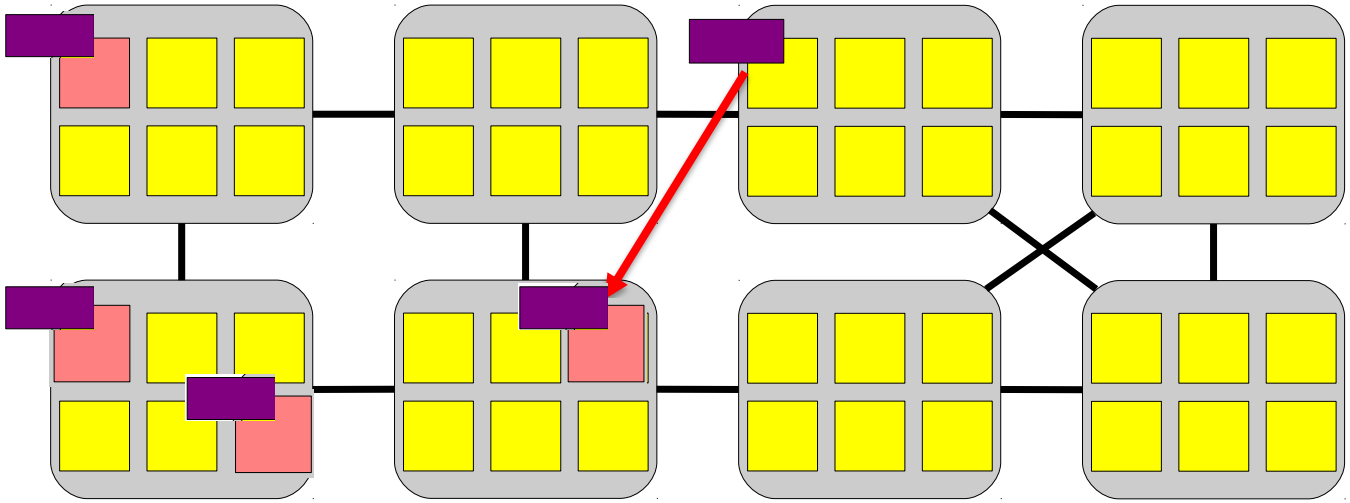
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Previous lock holder notifies next lock holder after sending out N/2 replies



All waiters read the lock

Service rate

Define

s: time spent in serial section

c: time taken by the directory to respond to a cache line request

Directory responds to each cache line request in turn

Service rate

k requests, time for winner to get the cache line is *ck/2*

Time to process the serial section and transfer the lock to the next holder: *s* + *ck/2*

Increases with more cores contending

Service rate: $s_k = 1/(s+ck/2)$

Decreases with more cores contending

Validating the model

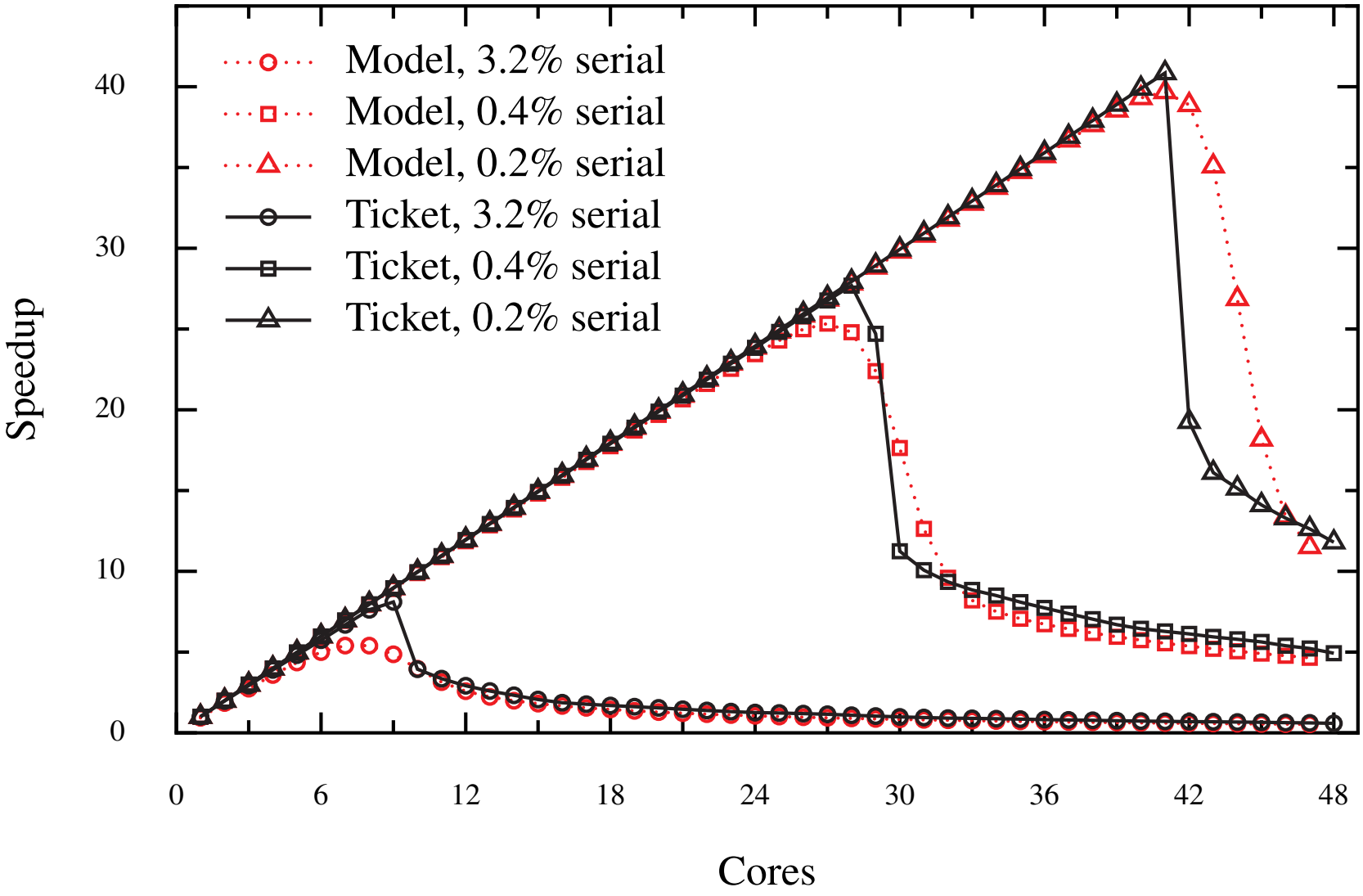
Use the model to predict speed up

- with different length of critical section

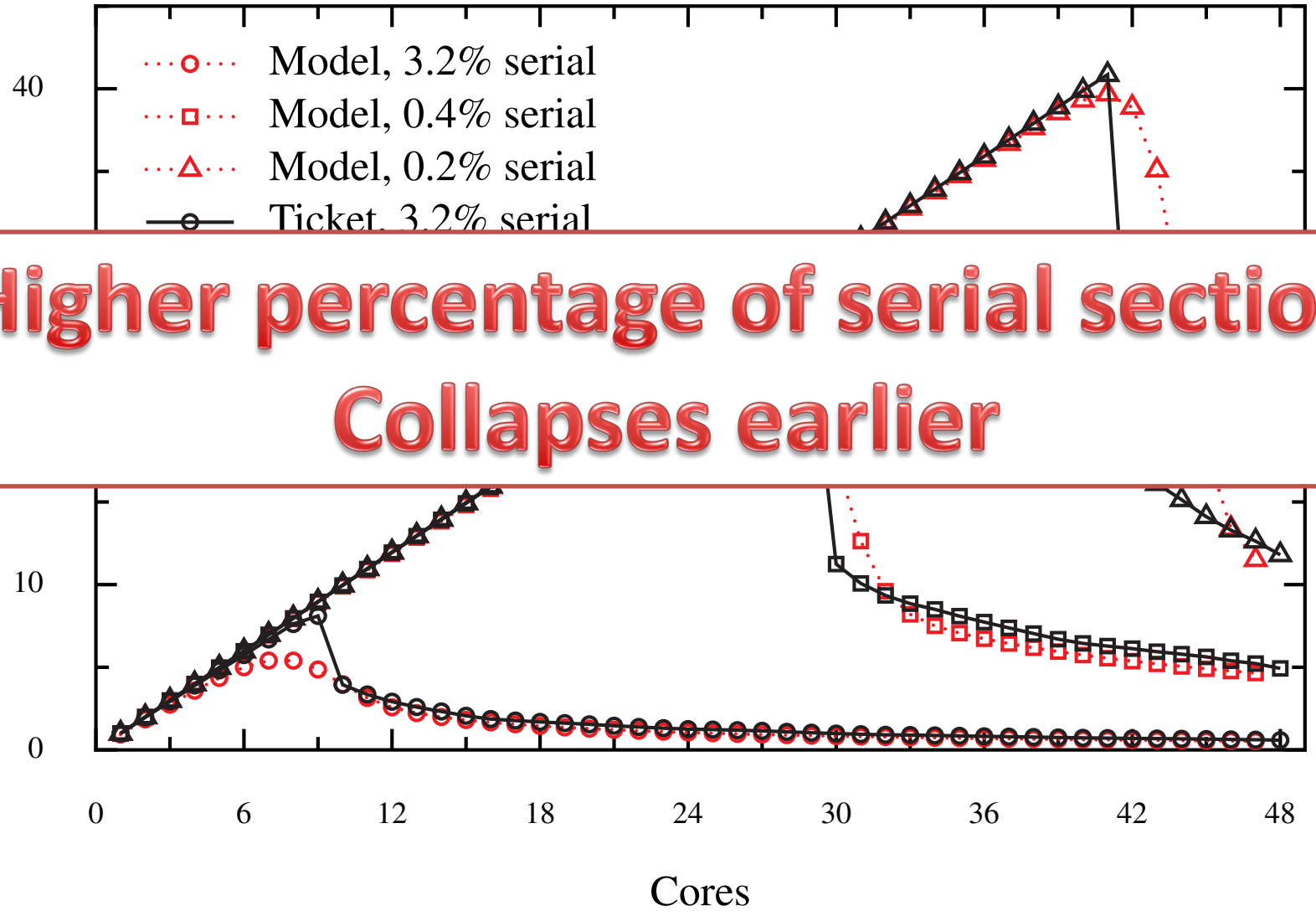
Compare the predication with actual speed up

- control critical section length with micro benchmarks

400-cycle serial section

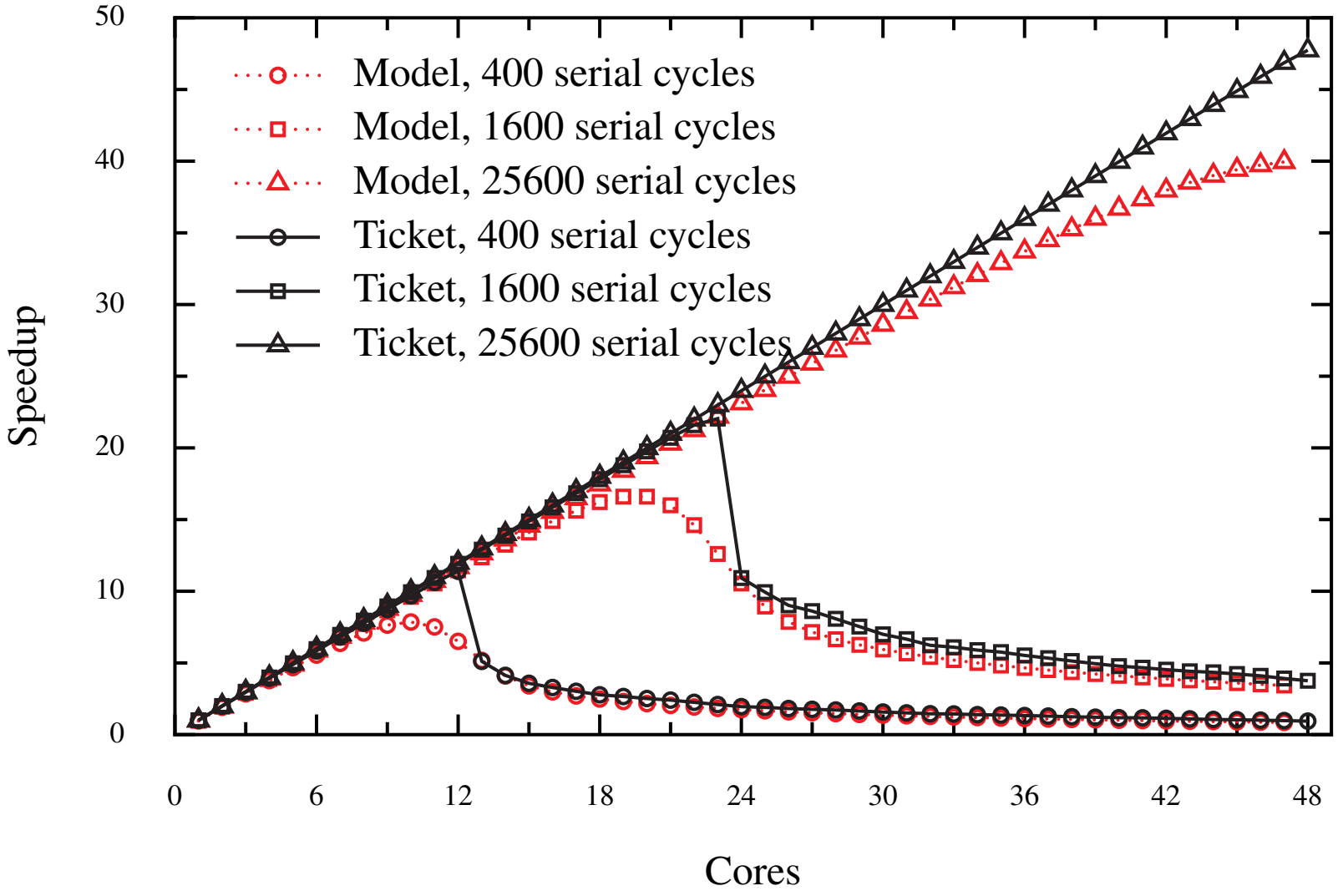


400-cycle serial section

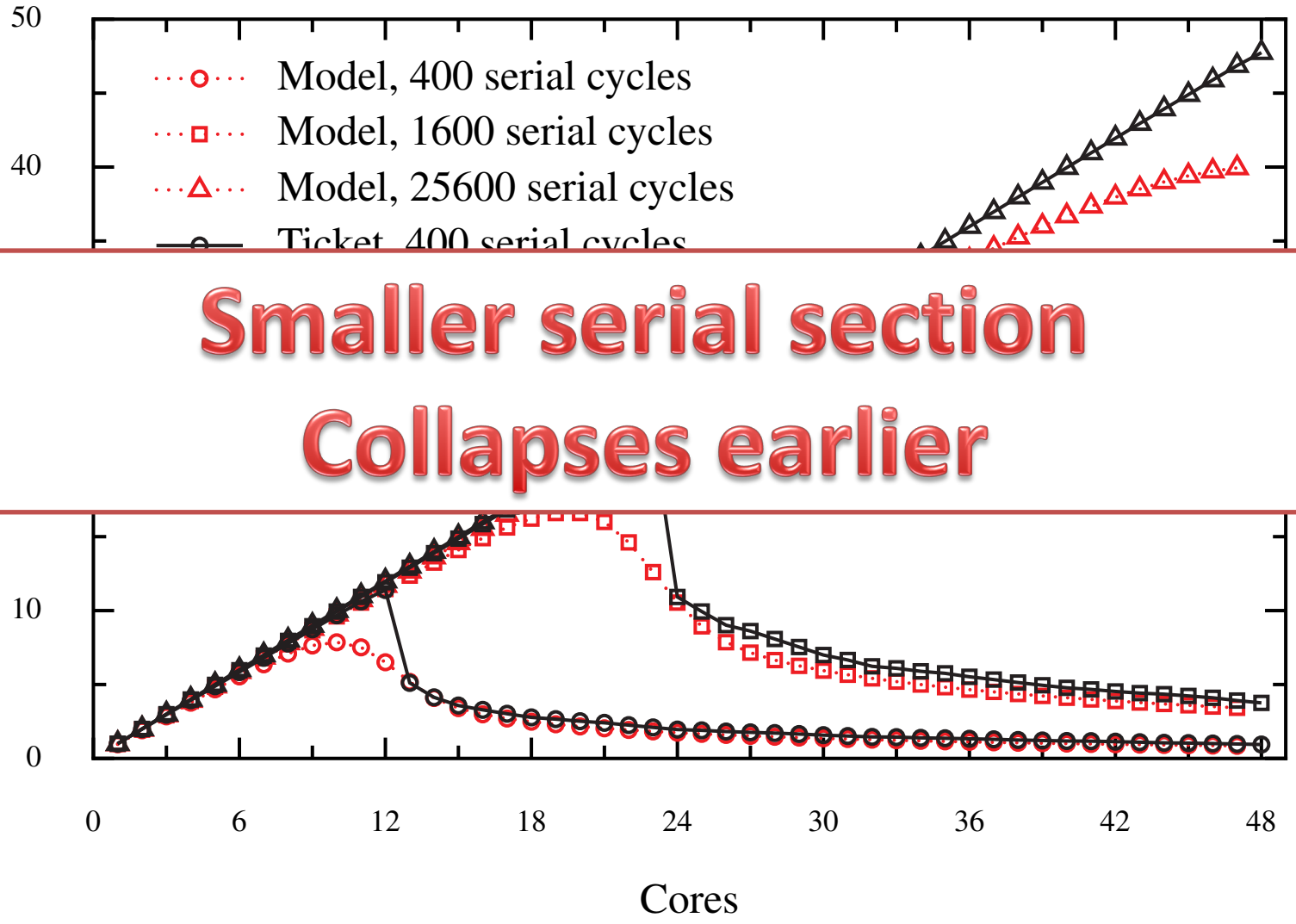


Higher percentage of serial section
Collapses earlier

Fixed serial section percentage (2%)



Fixed serial section percentage (2%)



Smaller serial section
Collapses earlier

Implications

The collapse of ticket lock is a property of their design

More cores contending

- Effectively increase the length of the critical section
- Thus increase the probability that another core start contending for the lock

Implications

Collapse only occurs for short critical sections

Service rate $s_k = 1/(s+ck/2)$

- When s is small, strongly influenced by *k*

The collapse of the ticket lock prevents the application from reaching the maximum performance predicted by Amdahl's law

Making ticket spinlock scalable

Common way: use proportional back-off

```
void spin_lock(spinlock_t *l) {
    int t = atomic_xadd(&l->next_ticket);
    while (t != lock->current_ticket)
        // wait more time with each failure
        ;
}
```

Why this would work? (answer this later in questions)

Problem with back-off

Hard to choose the back off time is important and difficult

May penalize non contended case

May confuse smart hardware which detects busy loops to enter power saving mode

Linus' Response

http://linux-kernel.2935.n7.nabble.com/PATCH-v5-0-5-x86-smp-make-ticket-spinlock-proportional-backoff-w-auto-tuning-td596698i20.html

So I claim:

- it's *really* hard to trigger in real loads on common hardware.

- if it does trigger in any half-way reasonably common setup (hardware/software), we most likely should work really hard at fixing the underlying problem, not the symptoms.

- we absolutely should *not* pessimize the common case for this

Using scalable locks

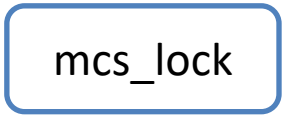
Many existing scalable locks

Main idea is to avoid contending on a single cache line

Example

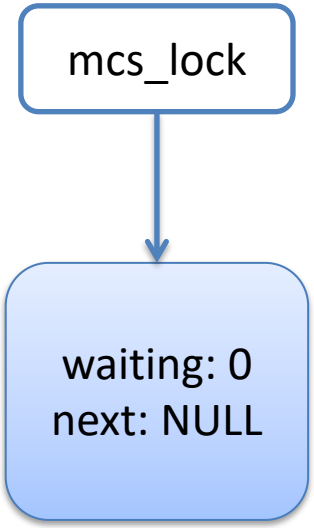
- MCS (John M. Mellor-Crummey and Michael L. Scott)
- K42

General idea of MCS lock



NULL

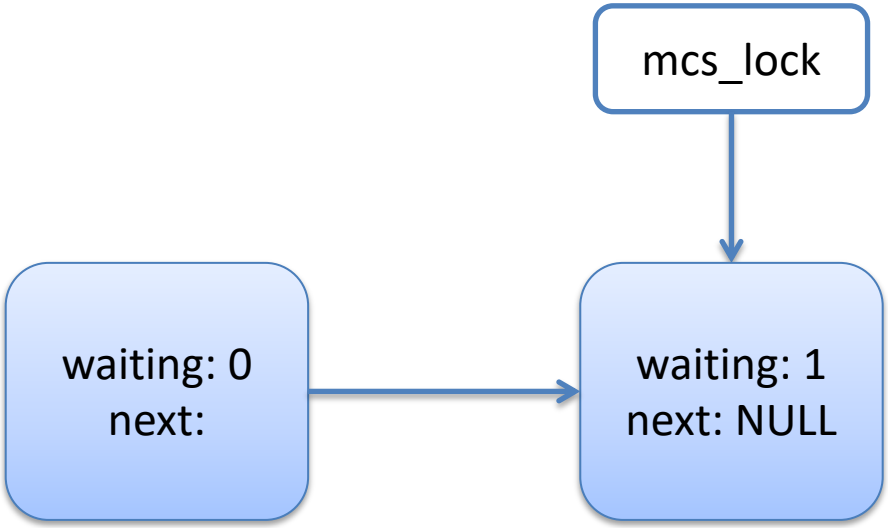
General idea of MCS lock



Use compare and swap to change mcs_lock point to self node

Check previous node
NULL in this case, no need to wait

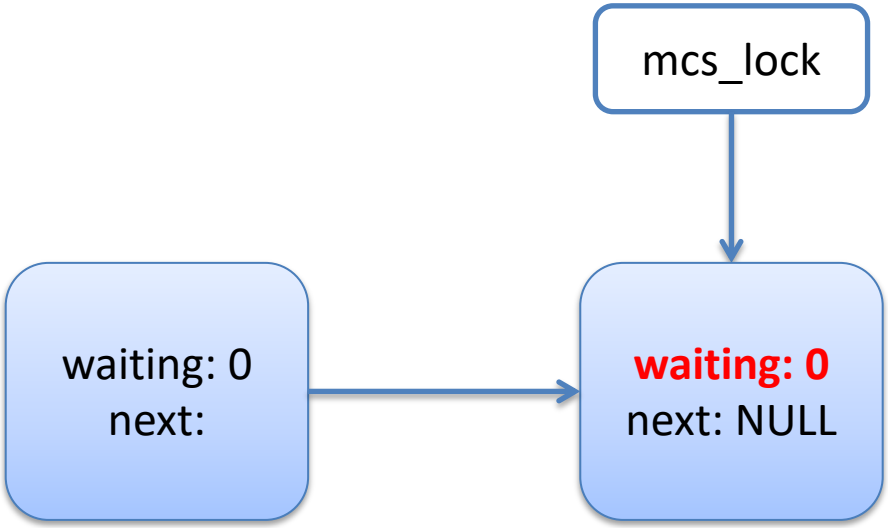
General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

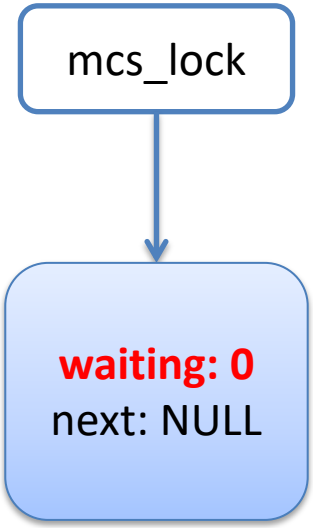
General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

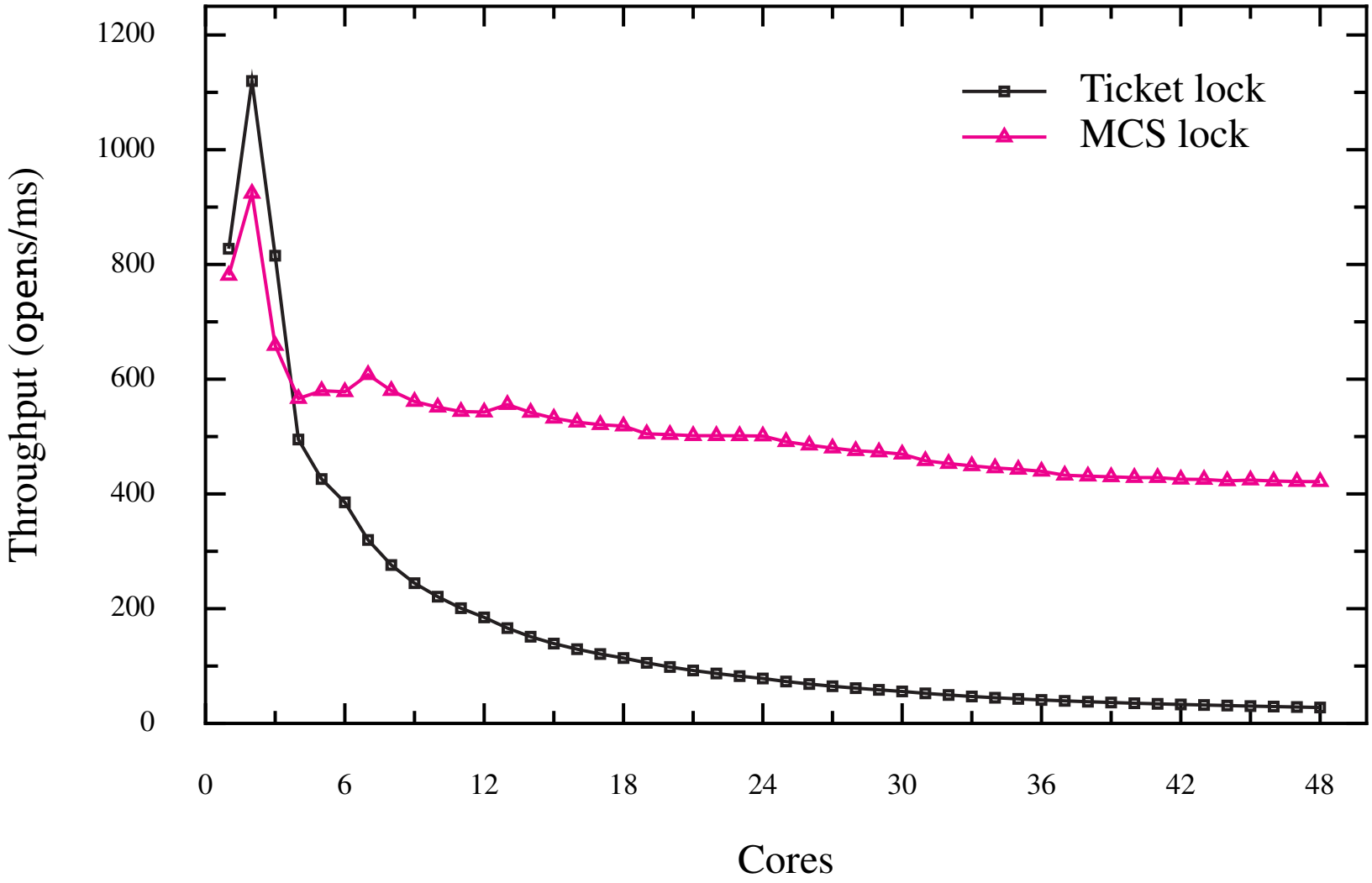
General idea of MCS lock



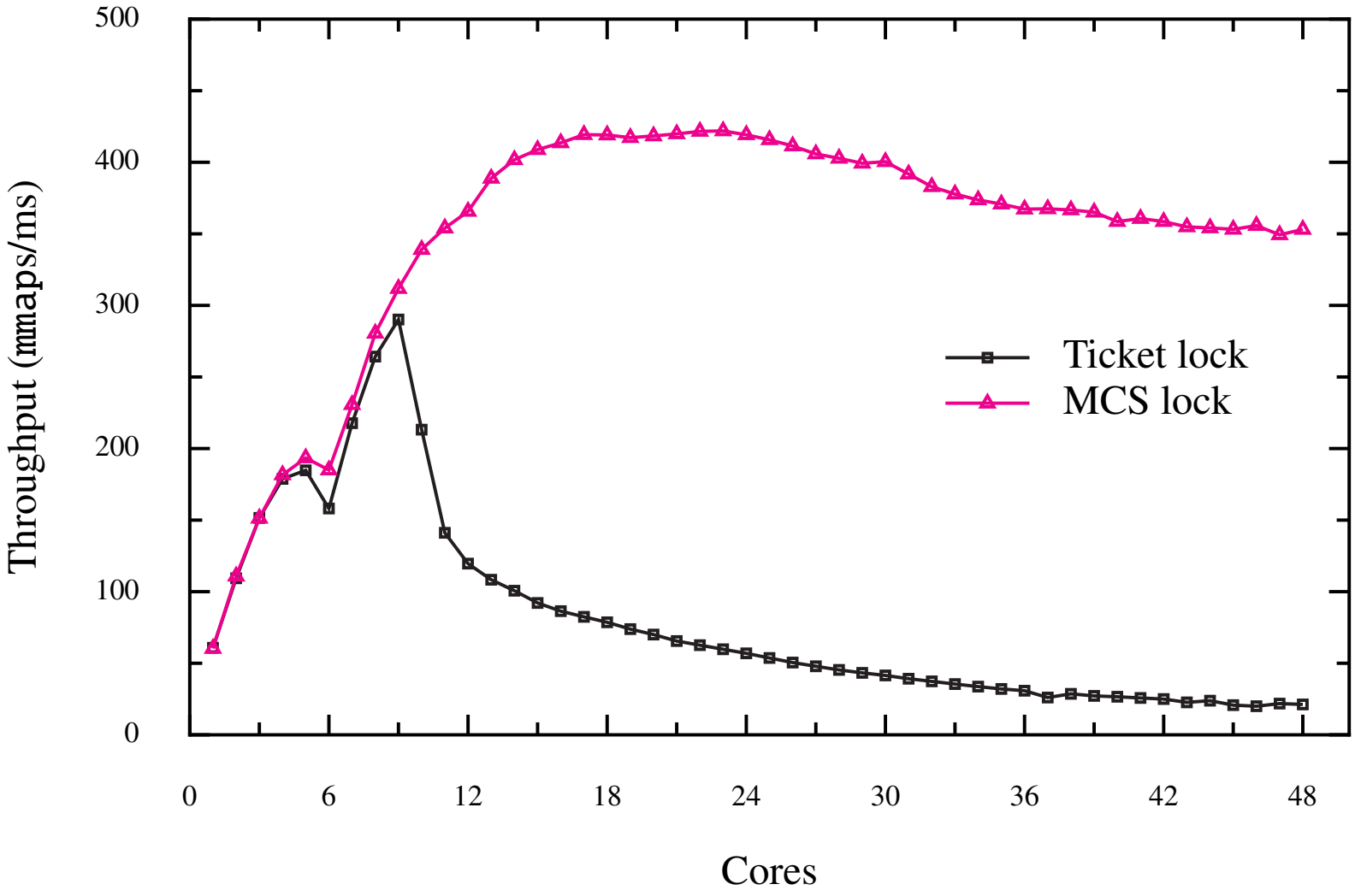
Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

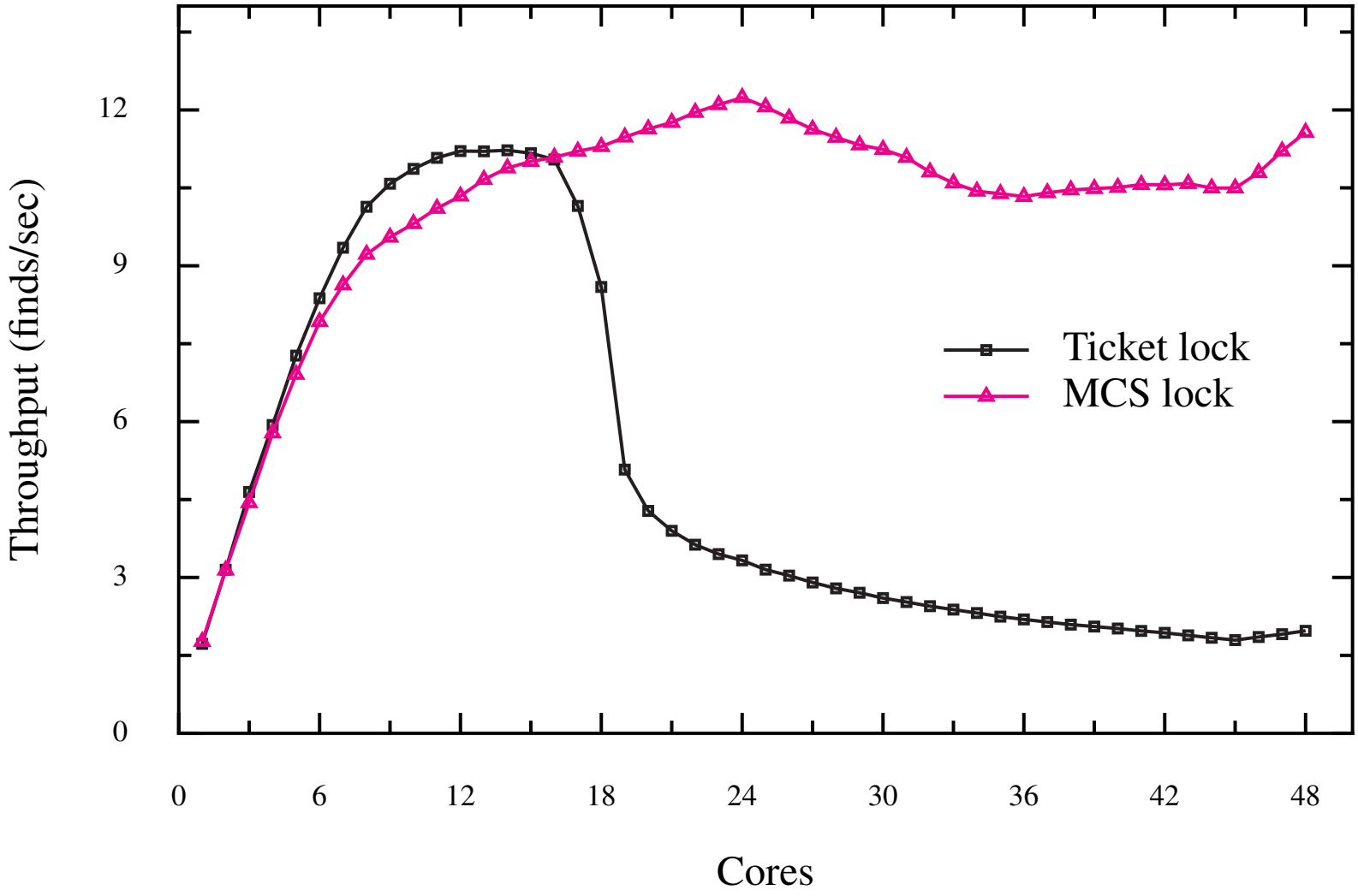
Using scalable locks: FOPS



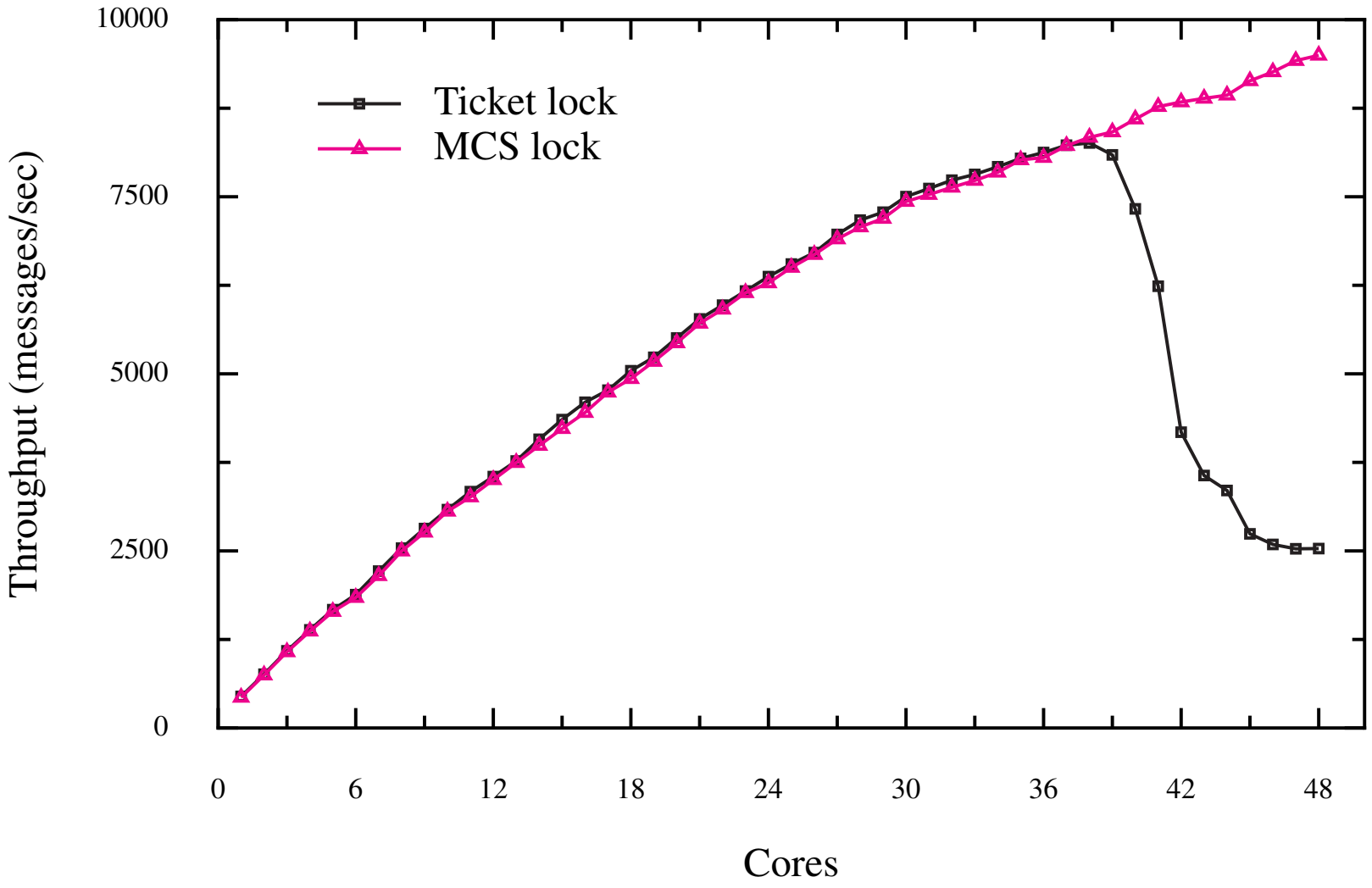
Using scalable locks: MEMPOP



Using scalable locks: PFIND



Using scalable locks: EXIM



Questions

Problems with the benchmark

- Only one real world application
- Collapses near #40 cores

Why the Linux kernel does not use the scalable locks?

- They perform worse with small number of cores and no contention
- The ultimate solution is to avoid contention instead of using scalable locks

Conclusion

Non-scalable locks are dangerous

- Short critical section may lead to performance collapse
- Caused by contention on lock cache line

Scalable locks is a way to relax the time-criticality of applying more fundamental scaling improvements to the kernel