The multicore evolution and operating systems

Frans Kaashoek

Joint work with: Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, Robert Morris, and Nickolai Zeldovich

MIT

Non-scalable locks are dangerous.

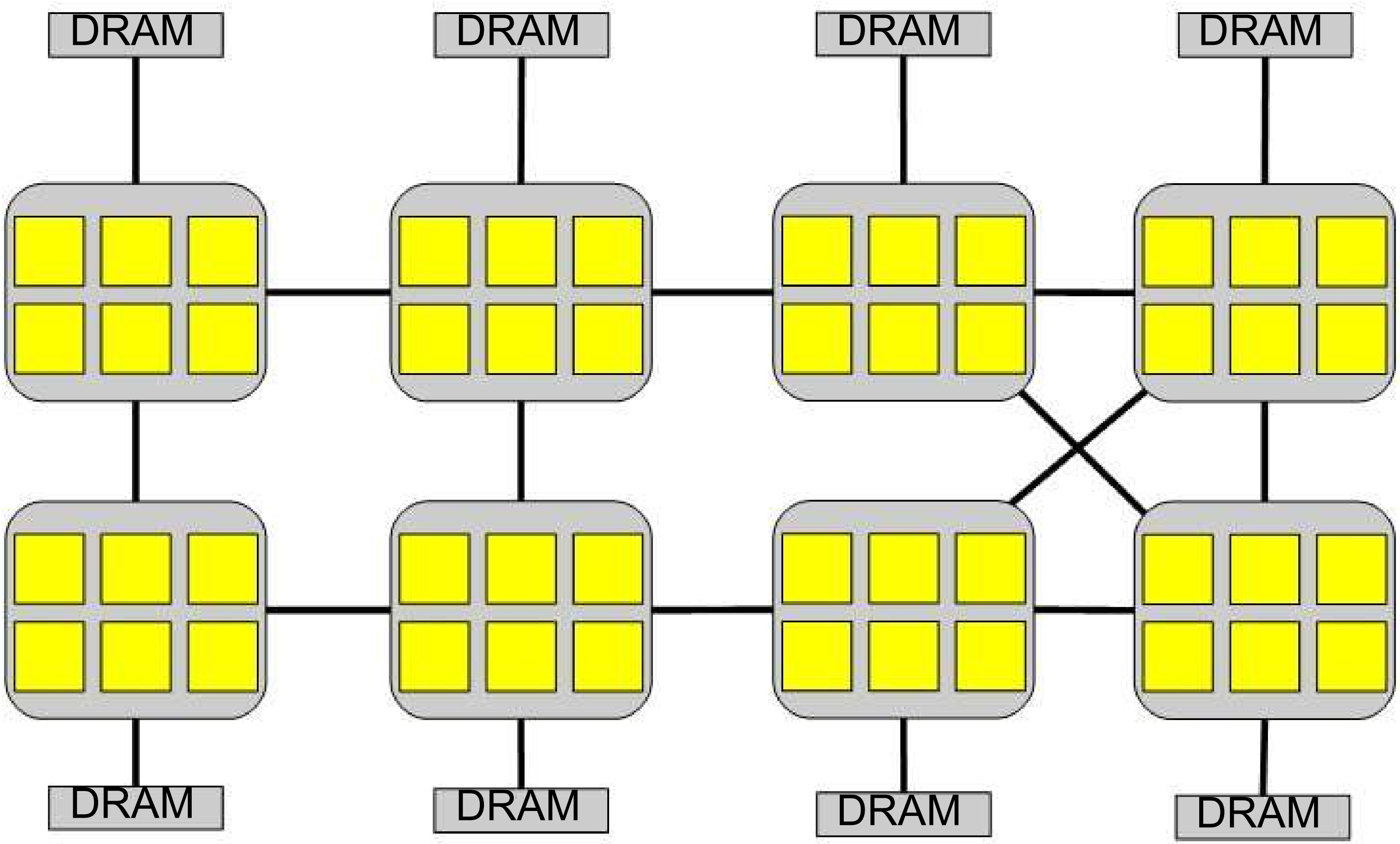
Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. *In the Proceedings of the Linux Symposium, Ottawa, Canada, July 2012.*

How well does Linux scale?

- Experiment:
  - Linux 2.6.35-rc5 (relatively old, but problems are representative of issues in recent kernels too)
  - Select a few inherent parallel system applications
  - Measure throughput on different # of cores
  - Use tmpfs to avoid disk bottlenecks

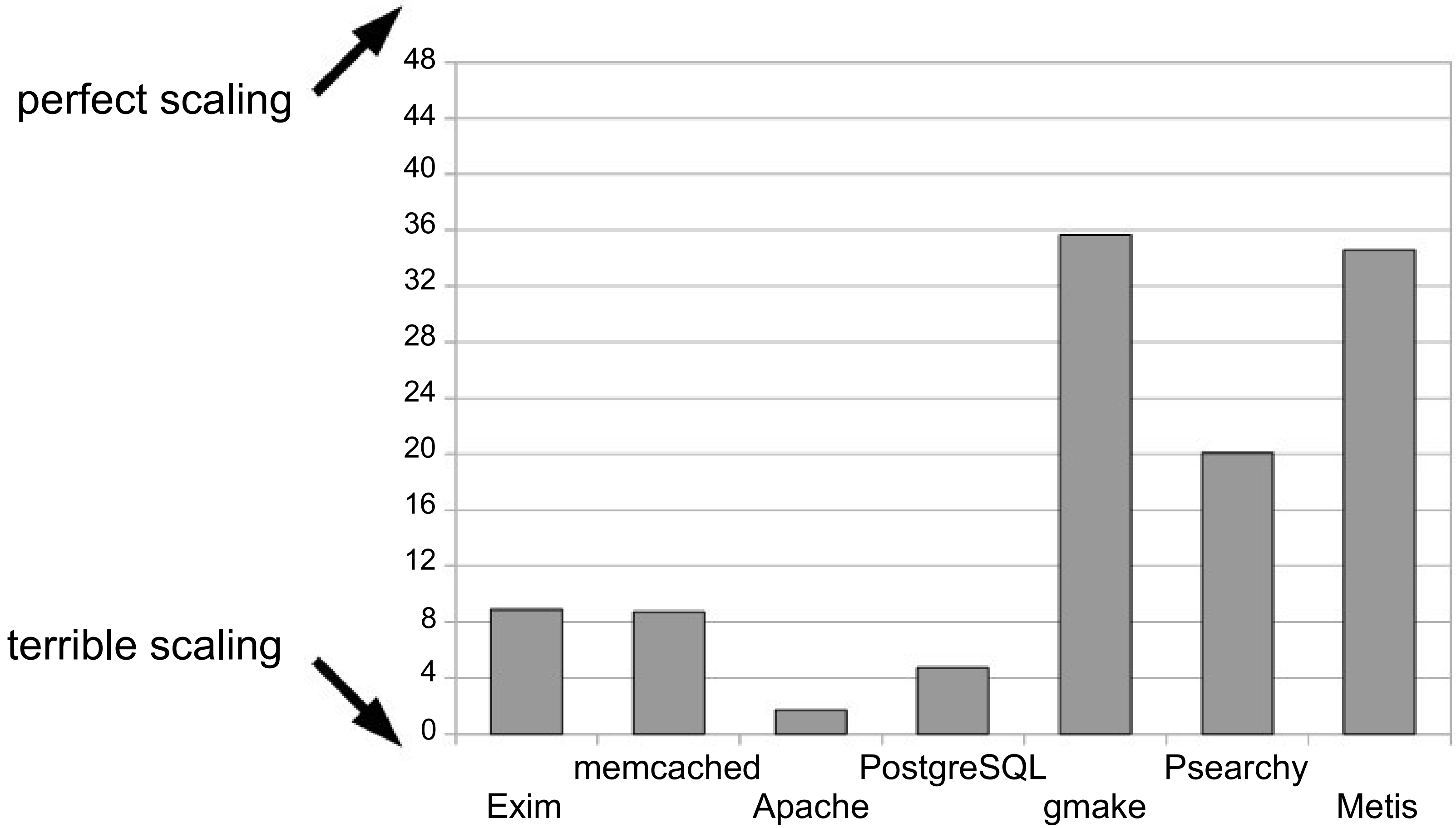
- Insight 1: Short critical sections can lead to sharp performance collapse

Off-the-shelf 48-core server (AMD)



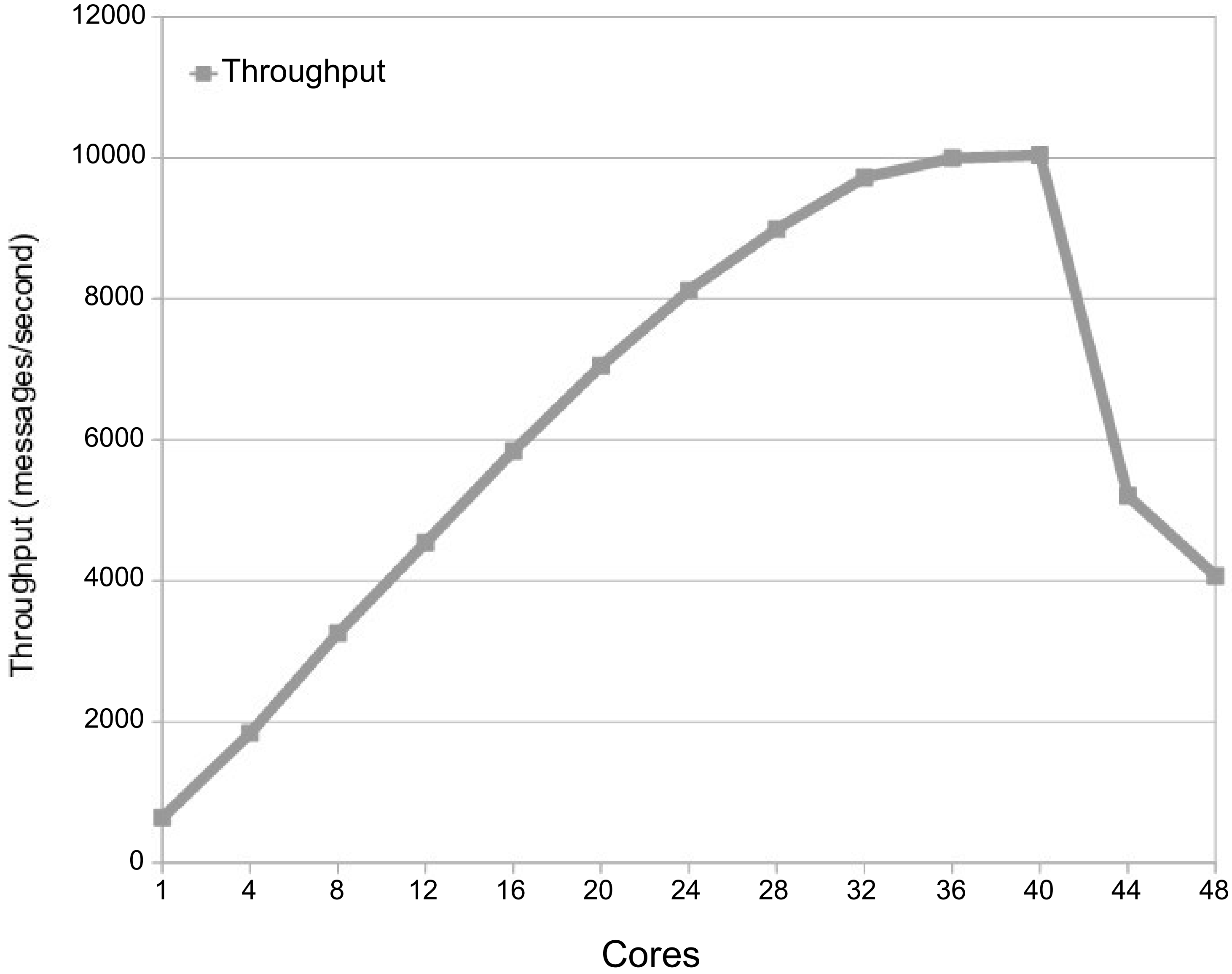
- Cache-coherent and non-uniform access
- An approximation of a future 48-core chip

Poor scaling on stock Linux kernel

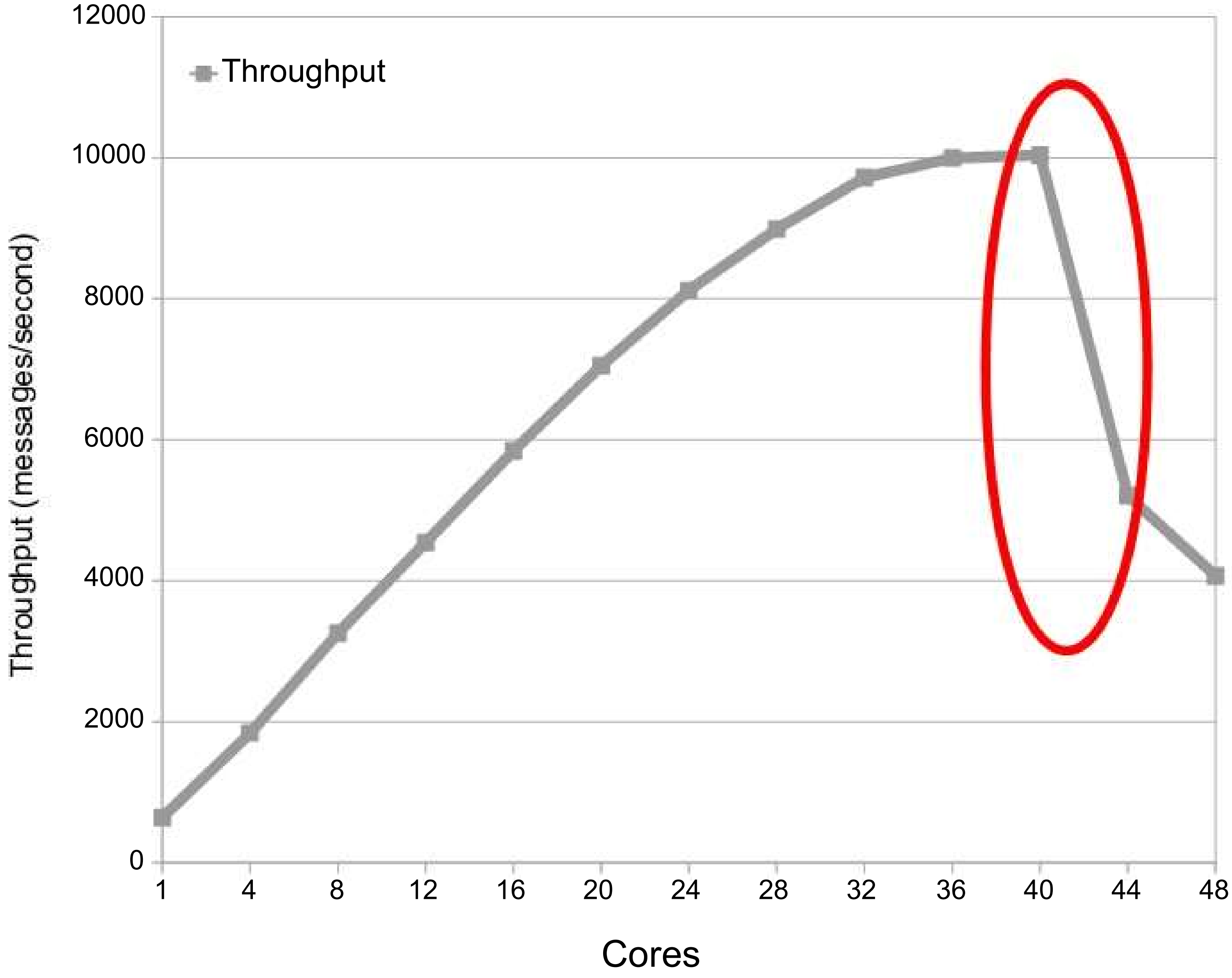


Y-axis: (throughput with 48 cores) / (throughput with one core)

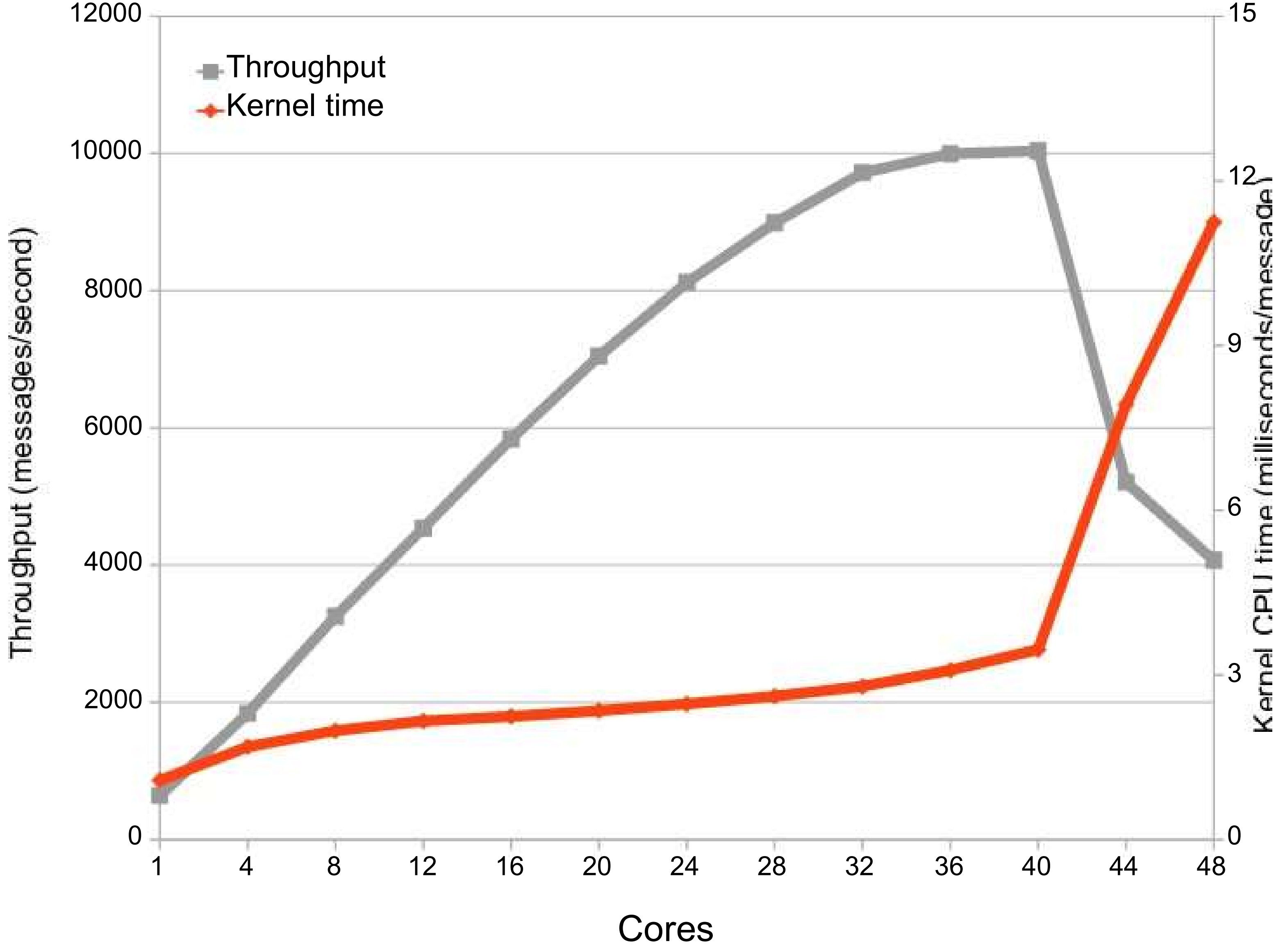
Exim on stock Linux: collapse



Exim on stock Linux: collapse



Exim on stock Linux: collapse



Oprofile shows an obvious problem

40 cores:
10000 msg/sec

samples	%	app name	symbol name
2616	7.3522	vmlinux	radix_tree_lookup_slot
2329	6.5456	vmlinux	unmap_vmas
2197	6.1746	vmlinux	filemap_fault
1488	4.1820	vmlinux	__do_fault
1348	3.7885	vmlinux	copy_page_c
1182	3.3220	vmlinux	unlock_page
966	2.7149	vmlinux	page_fault

48 cores:
4000 msg/sec

samples	%	app name	symbol name
13515	34.8657	vmlinux	lookup_mnt
2002	5.1647	vmlinux	radix_tree_lookup_slot
1661	4.2850	vmlinux	filemap_fault
1497	3.8619	vmlinux	unmap_vmas
1026	2.6469	vmlinux	__do_fault
914	2.3579	vmlinux	atomic_dec
896	2.3115	vmlinux	unlock_page

Oprofile shows an obvious problem

40 cores:
10000 msg/sec

samples	%	app name	symbol name
2616	7.3522	vmlinux	radix_tree_lookup_slot
2329	6.5456	vmlinux	unmap_vmas
2197	6.1746	vmlinux	filemap_fault
1488	4.1820	vmlinux	__do_fault
1348	3.7885	vmlinux	copy_page_c
1182	3.3220	vmlinux	unlock_page
966	2.7149	vmlinux	page_fault

48 cores:
4000 msg/sec

samples	%	app name	symbol name
13515	34.8657	vmlinux	lookup_mnt
2002	5.1647	vmlinux	radix_tree_lookup_slot
1661	4.2850	vmlinux	filemap_fault
1497	3.8619	vmlinux	unmap_vmas
1026	2.6469	vmlinux	__do_fault
914	2.3579	vmlinux	atomic_dec
896	2.3115	vmlinux	unlock_page

Oprofile shows an obvious problem

40 cores:
10000 msg/sec

samples	%	app name	symbol name
2616	7.3522	vmlinux	radix_tree_lookup_slot
2329	6.5456	vmlinux	unmap_vmas
2197	6.1746	vmlinux	filemap_fault
1488	4.1820	vmlinux	__do_fault
1348	3.7885	vmlinux	copy_page_c
1182	3.3220	vmlinux	unlock_page
966	2.7149	vmlinux	page_fault

48 cores:
4000 msg/sec

samples	%	app name	symbol name
13515	34.8657	vmlinux	lookup_mnt
2002	5.1647	vmlinux	radix_tree_lookup_slot
1661	4.2850	vmlinux	filemap_fault
1497	3.8619	vmlinux	unmap_vmas
1026	2.6469	vmlinux	__do_fault
914	2.3579	vmlinux	atomic_dec
896	2.3115	vmlinux	unlock_page

Bottleneck: reading mount table

- Delivering an email calls sys_open
- sys_open calls

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        return mnt;
}
```

Bottleneck: reading mount table

- sys_open calls:

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        return mnt;
}
```

Bottleneck: reading mount table

- sys_open calls:

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        return mnt;
}
```

Serial section is short. Why does it cause a scalability bottleneck?

What causes the sharp performance collapse?

- Linux uses ticket spin locks, which are non-scalable
  - So we should expect collapse [Anderson 90]

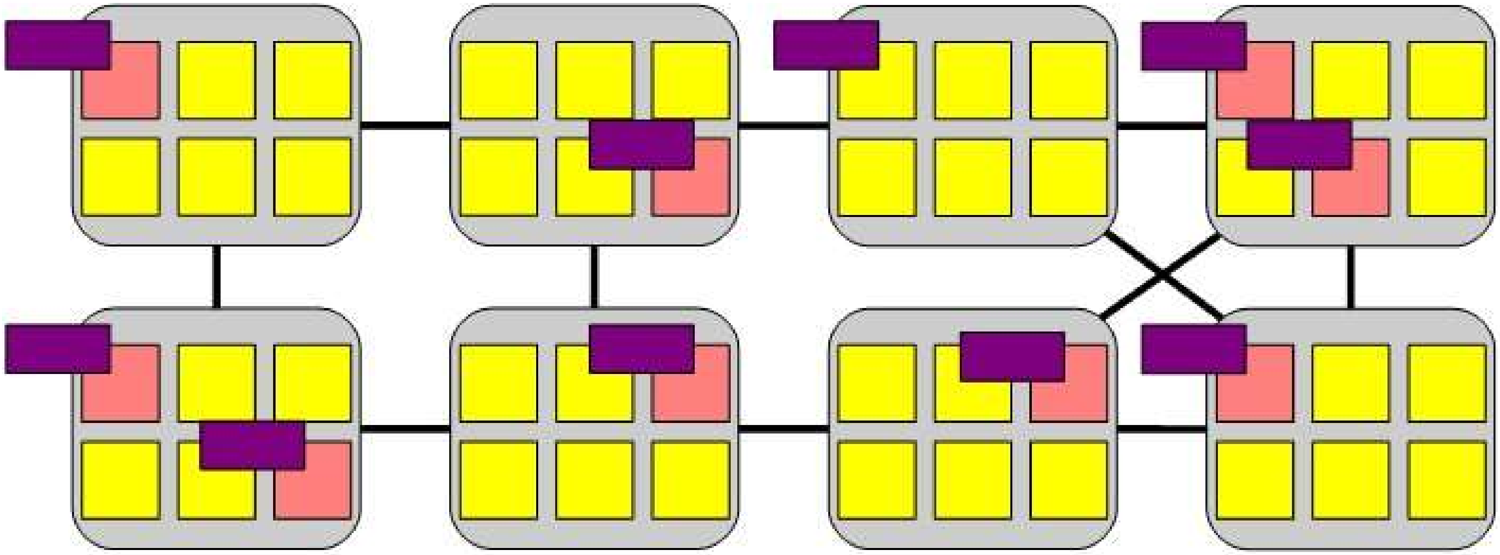
- But why so sudden, and so sharp, for a short section?
  - Is spin lock/unlock implemented incorrectly?
  - Is hardware cache-coherence protocol at fault?

Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

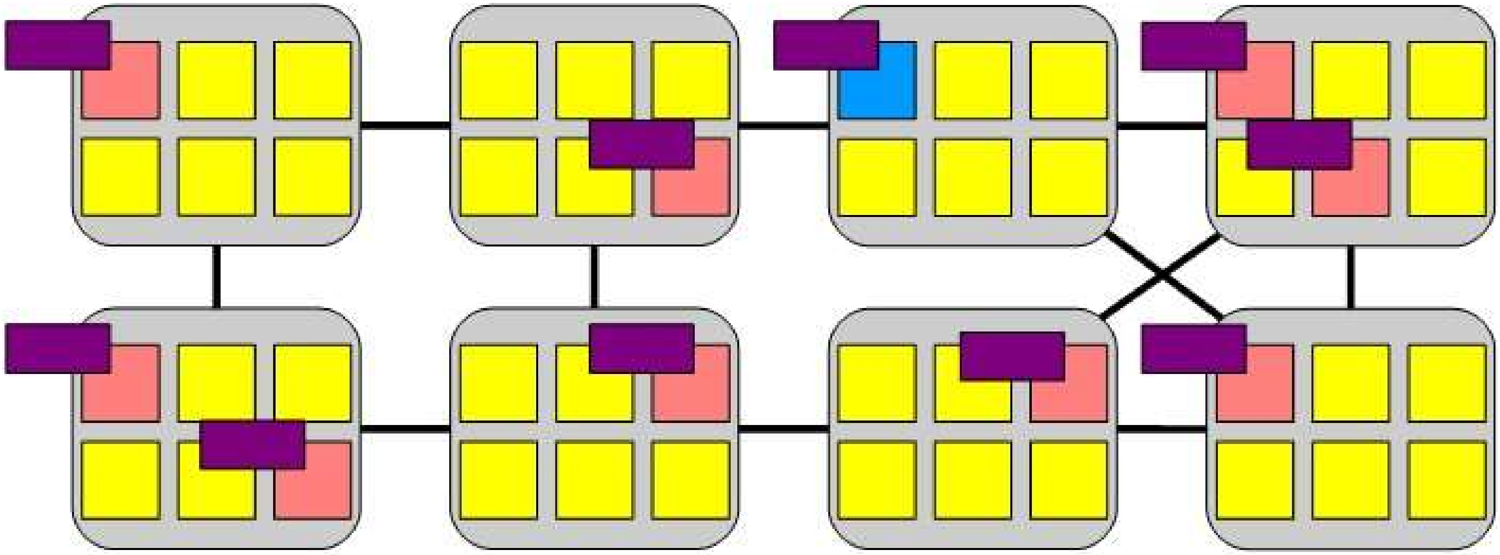


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

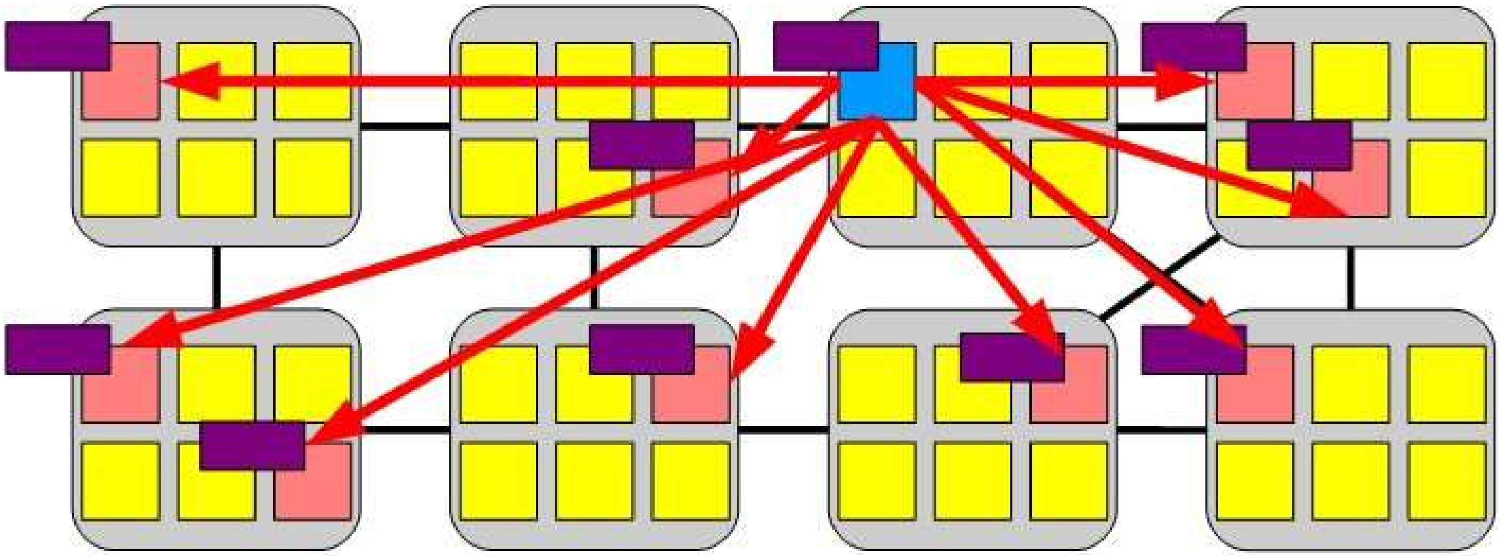


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

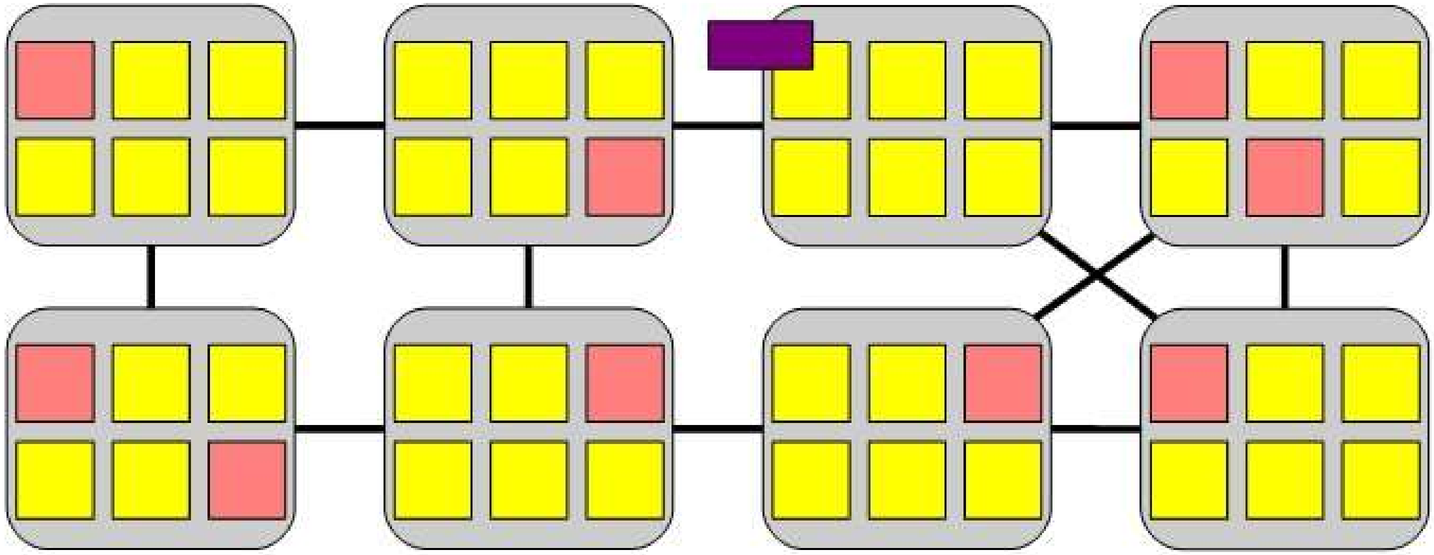


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

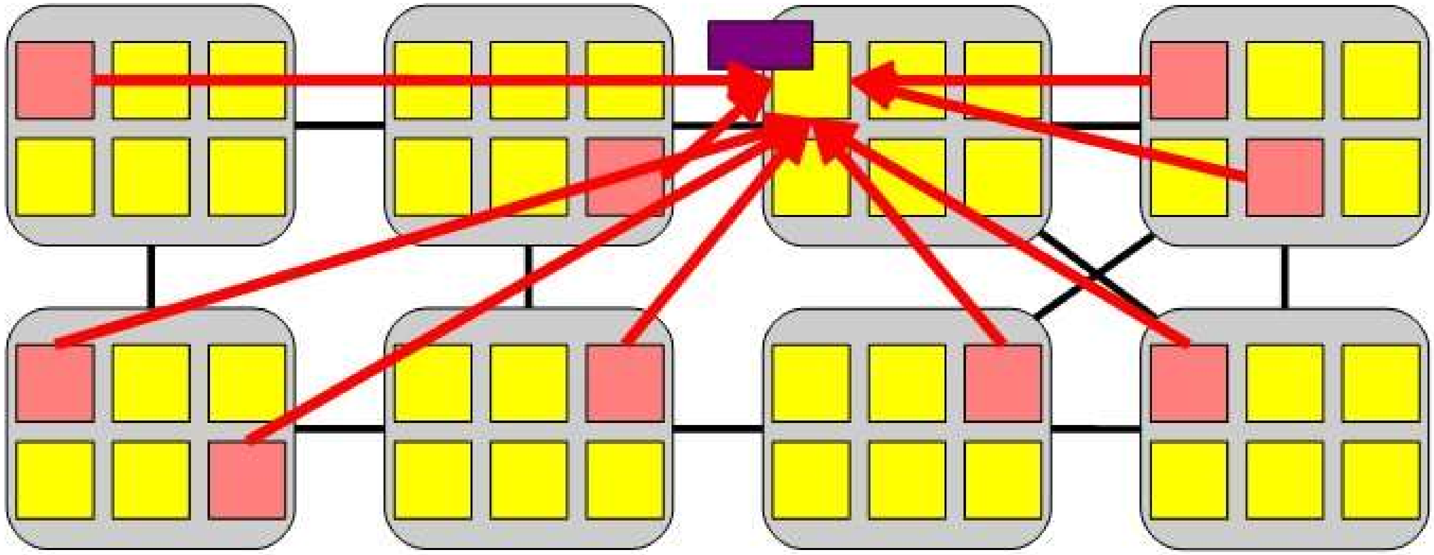


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

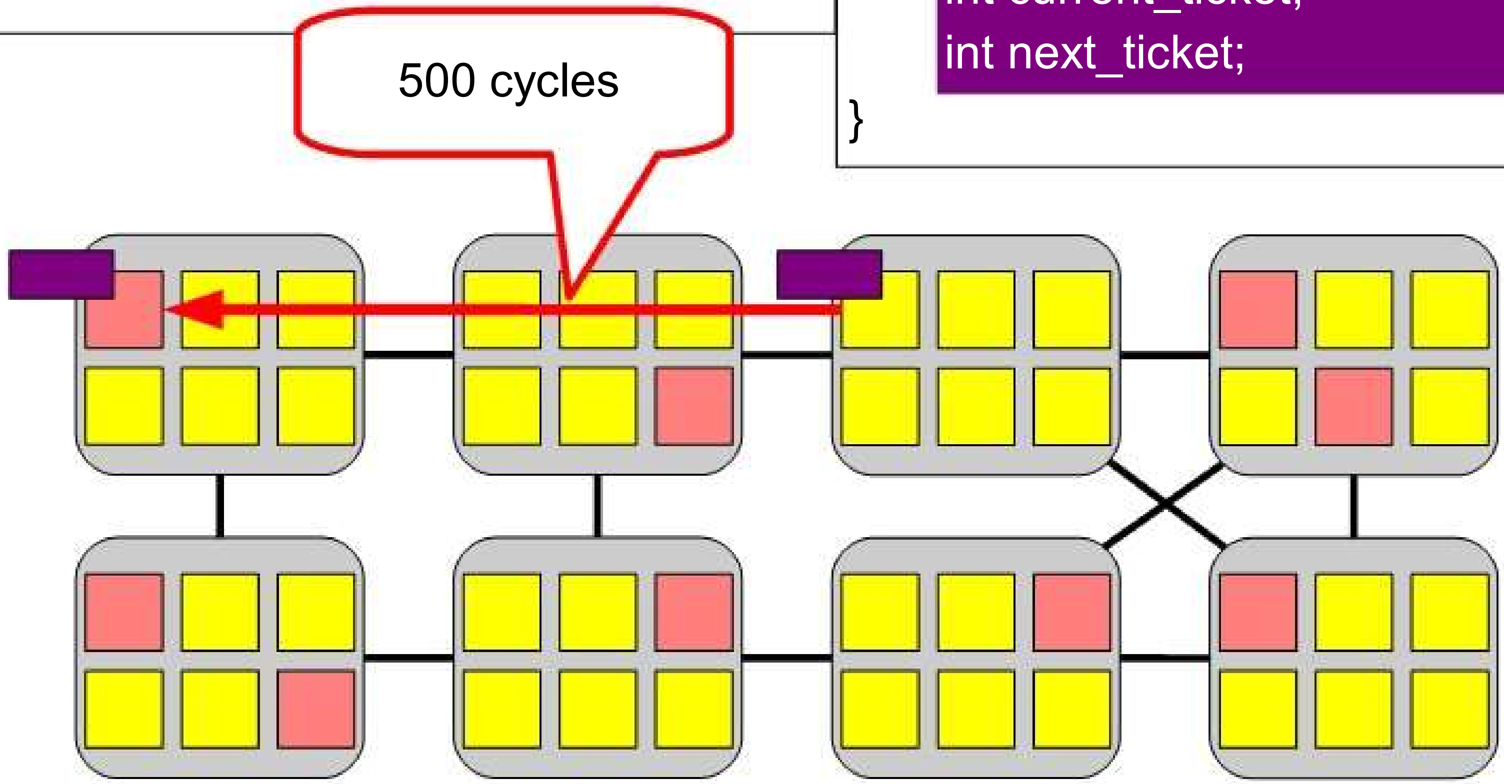


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

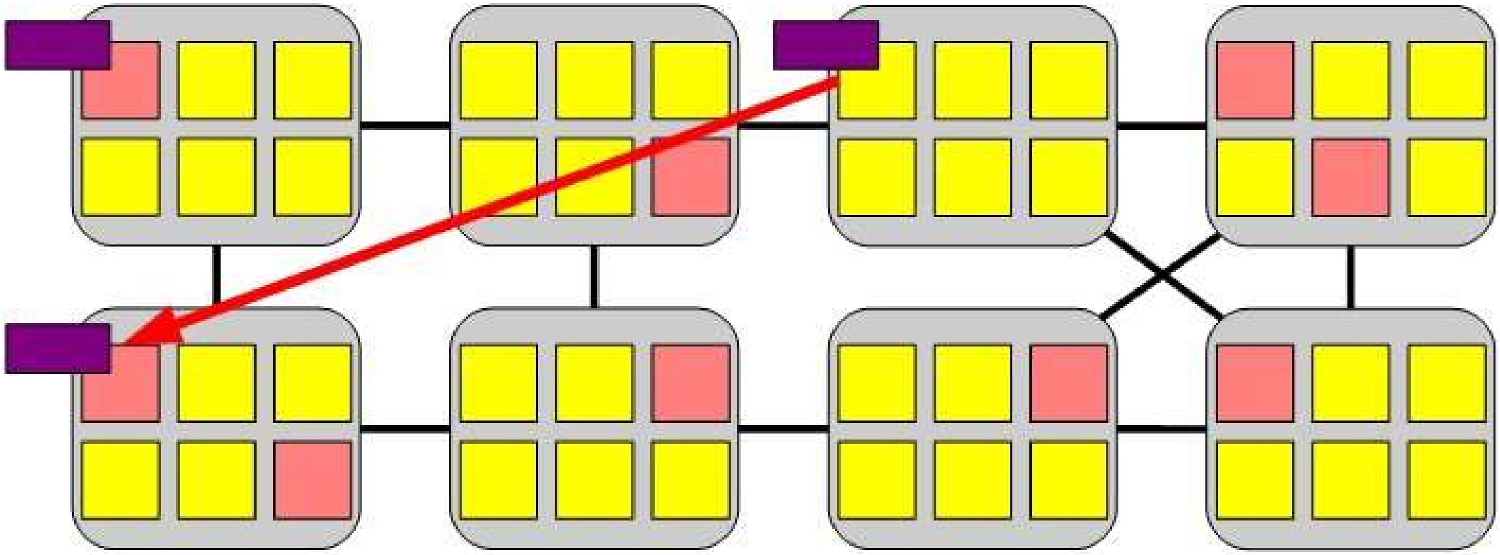


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

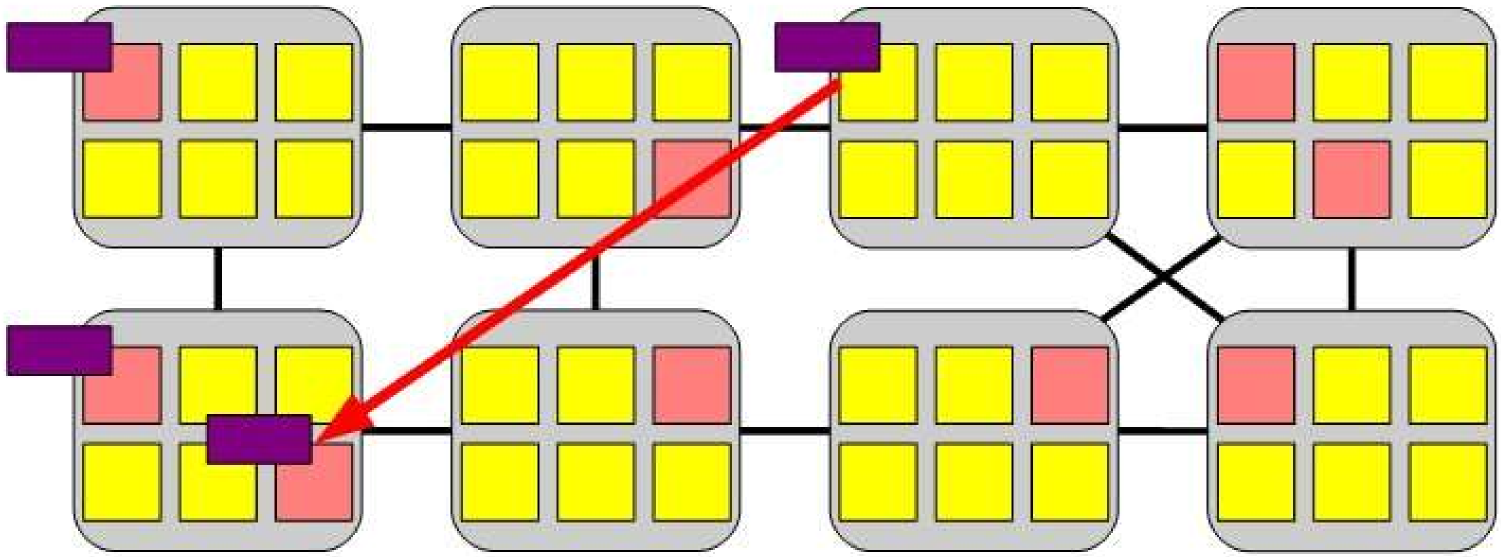


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

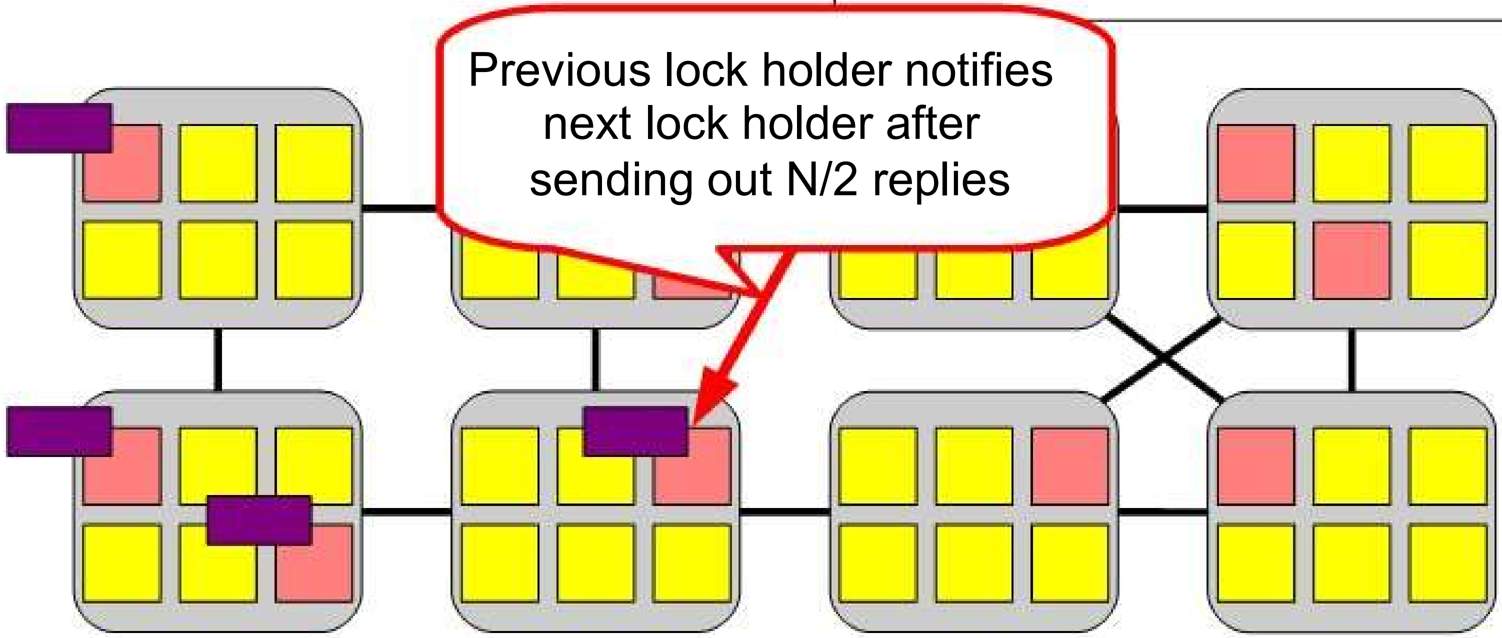


Scalability collapse caused by non-scalable locks [Anderson 90]

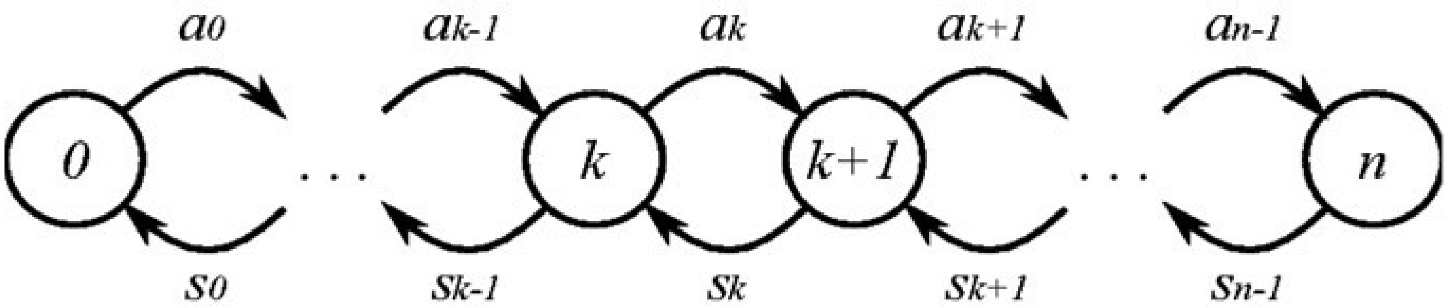
```
void spin_lock(spinlock_t *lock)
{
        t = atomic_inc(lock->next_ticket);
        while (t != lock->current_ticket)
                ;    /* Spin */
}
```

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

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

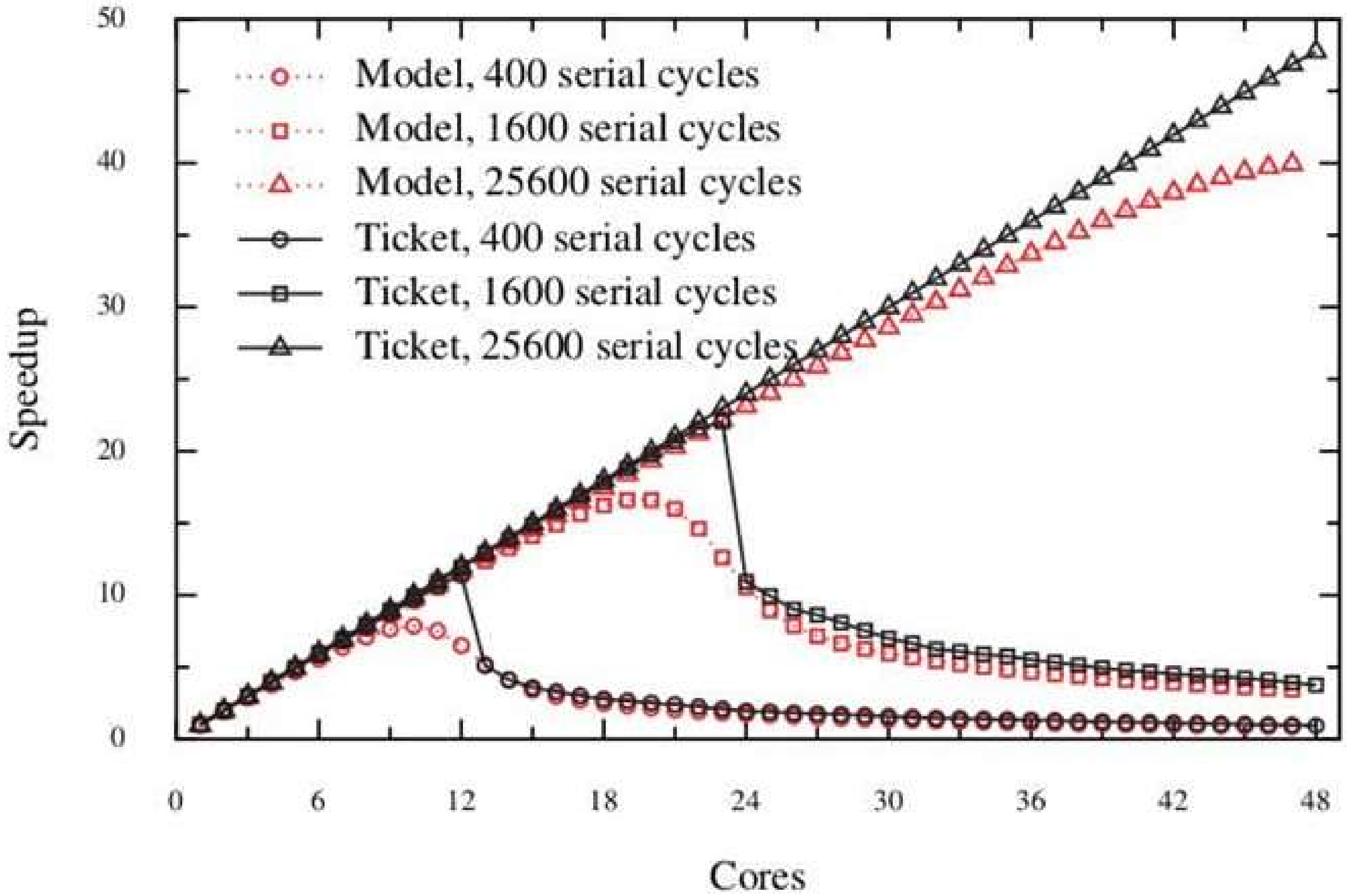


Why collapse with short sections?



- Arrival rate is proportional to # non-waiting cores
- Service time is proportional to # cores waiting (*k*)
  - As *k* increases, waiting time goes up
  - As waiting time goes up, *k* increases
- System gets stuck in states with many waiting cores

Short sections result in collapse

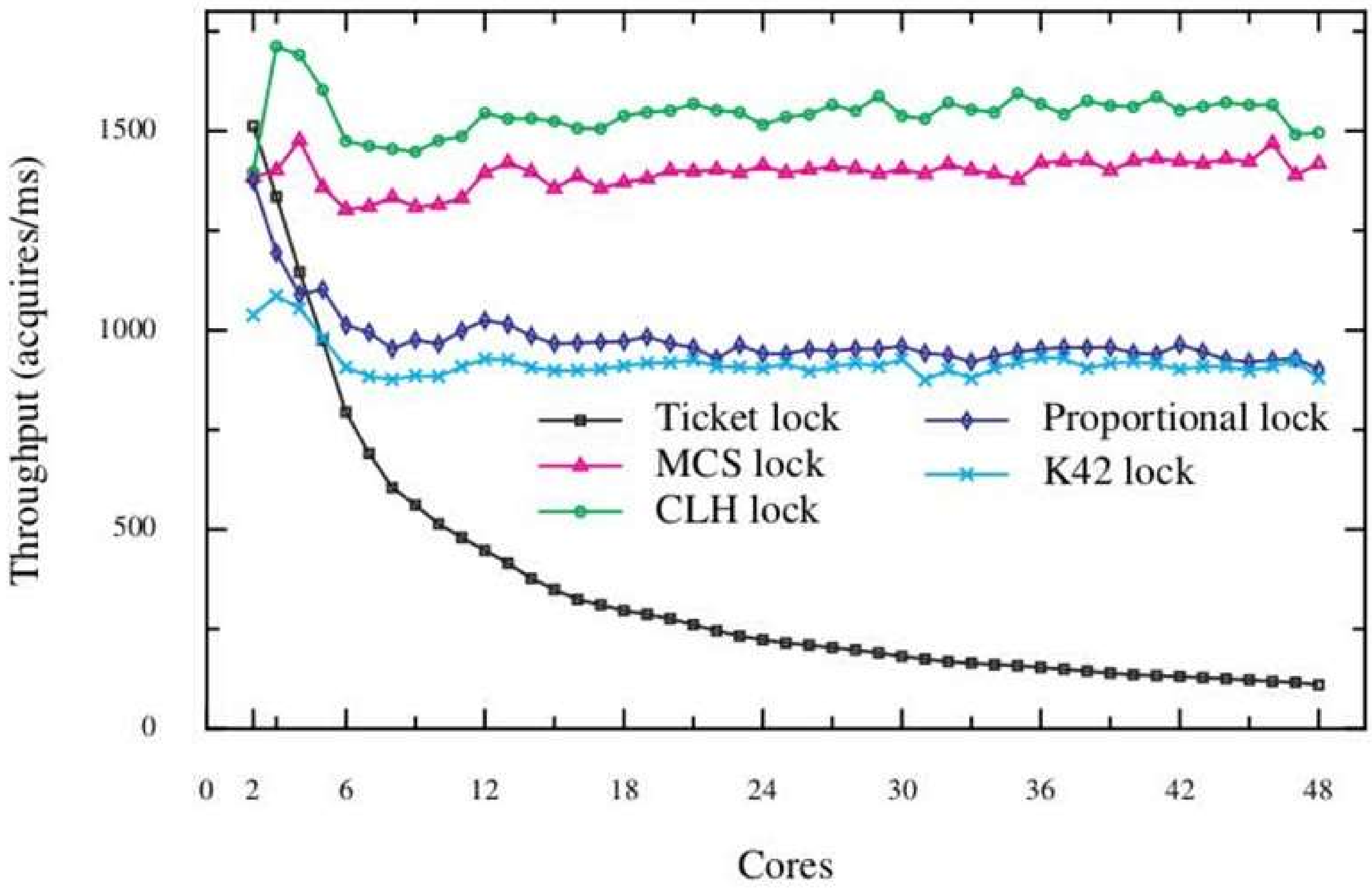


- Experiment: 2% of time spent in critical section
- Critical sections become “longer” with more cores
- Lesson: non-scalable locks fine for long sections

Avoiding lock collapse

- Unscalable locks are fine for long sections
- Unscalable locks collapse for short sections
  - Sudden sharp collapse due to “snowball” effect
- Scalable locks avoid collapse altogether
  - But requires interface change

Scalable lock scalability



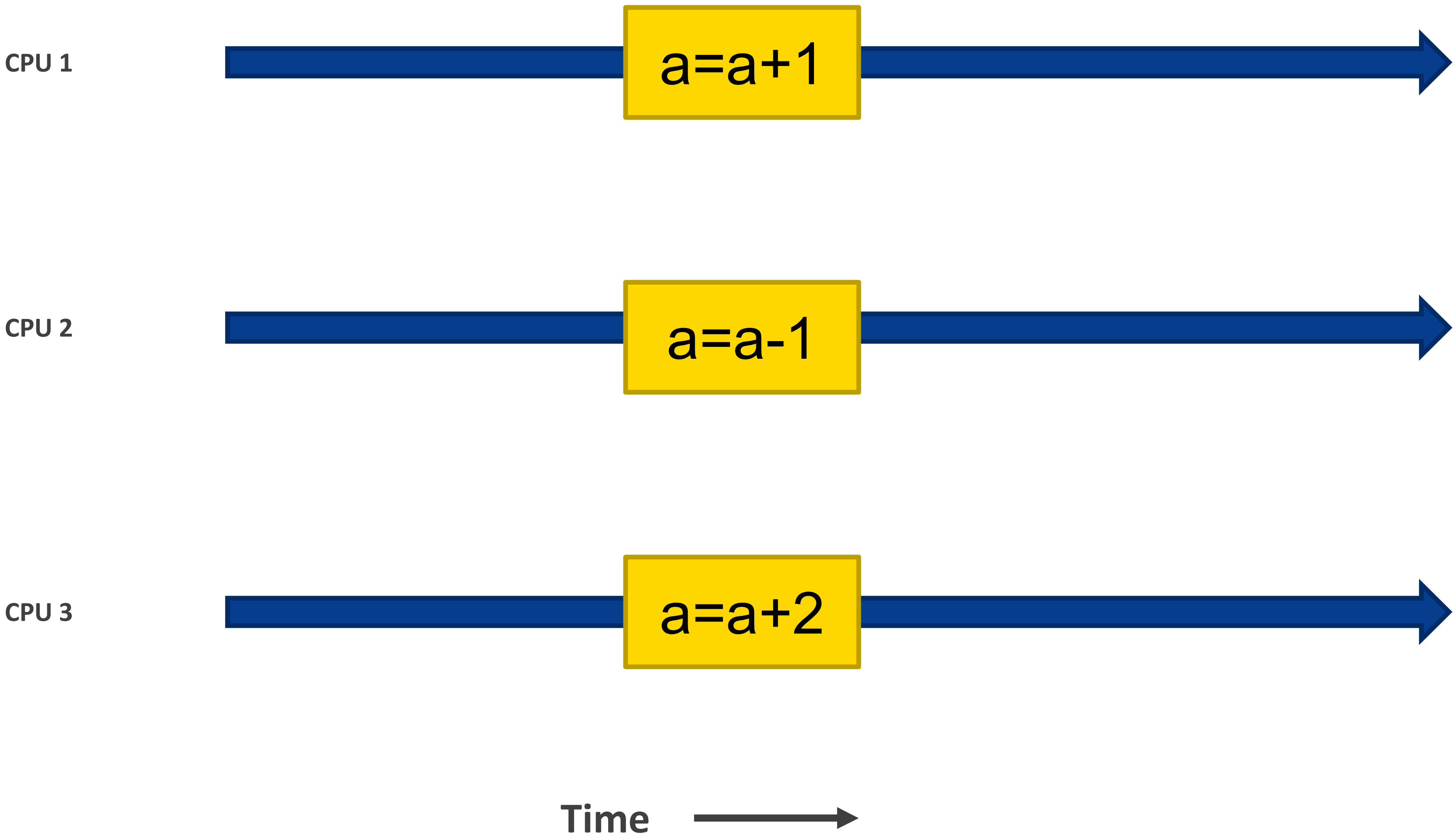
- It doesn't matter much which one
- But all slower in terms of latency

Avoiding lock collapse is not enough to scale

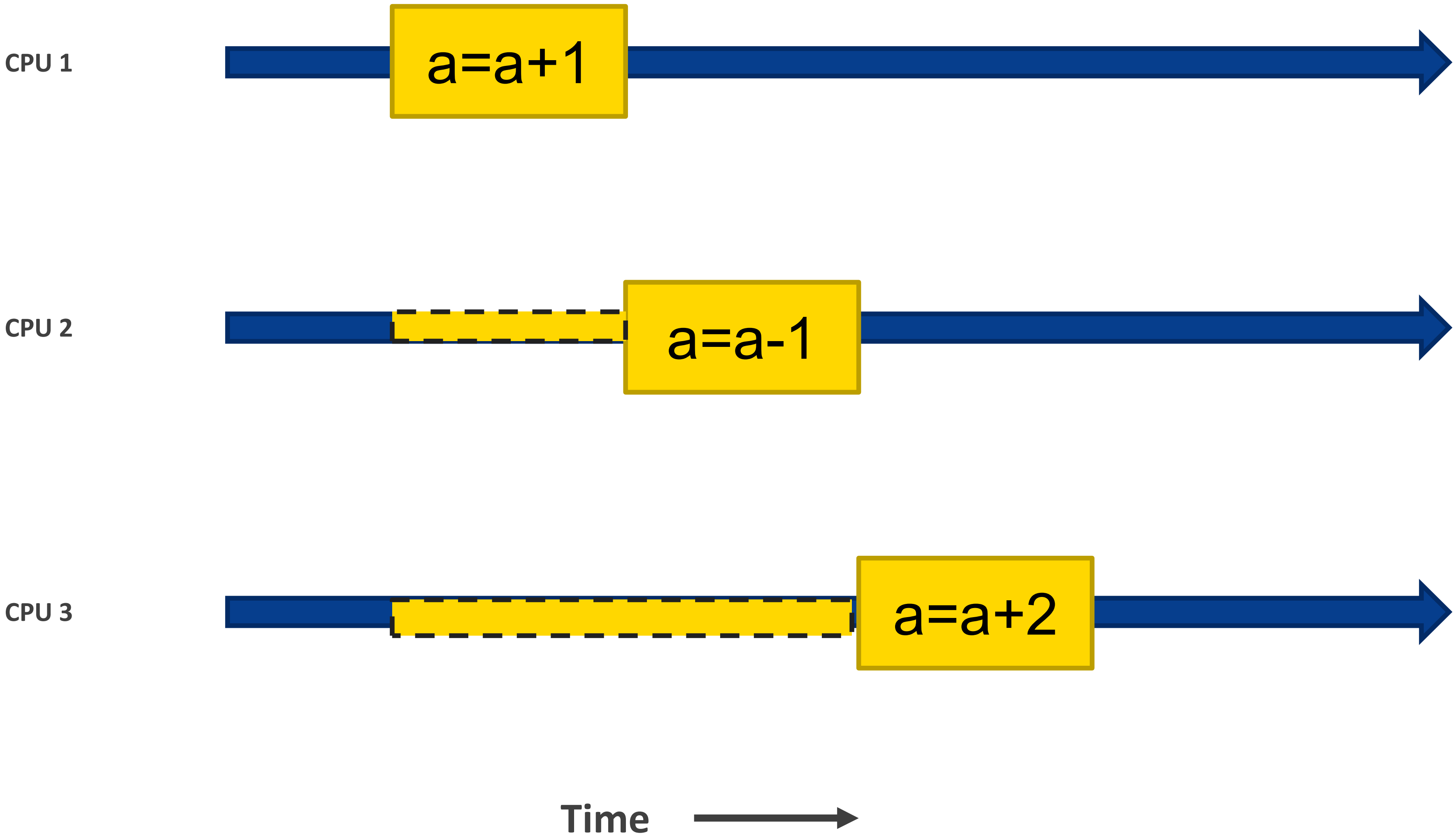
- “Scalable” locks don't make the kernel scalable
  - Main benefit is avoiding collapse: total throughput will not be lower with more cores
  - But, usually want throughput to keep increasing with more cores

Transactional memory to manage concurrency

The problem – concurrency



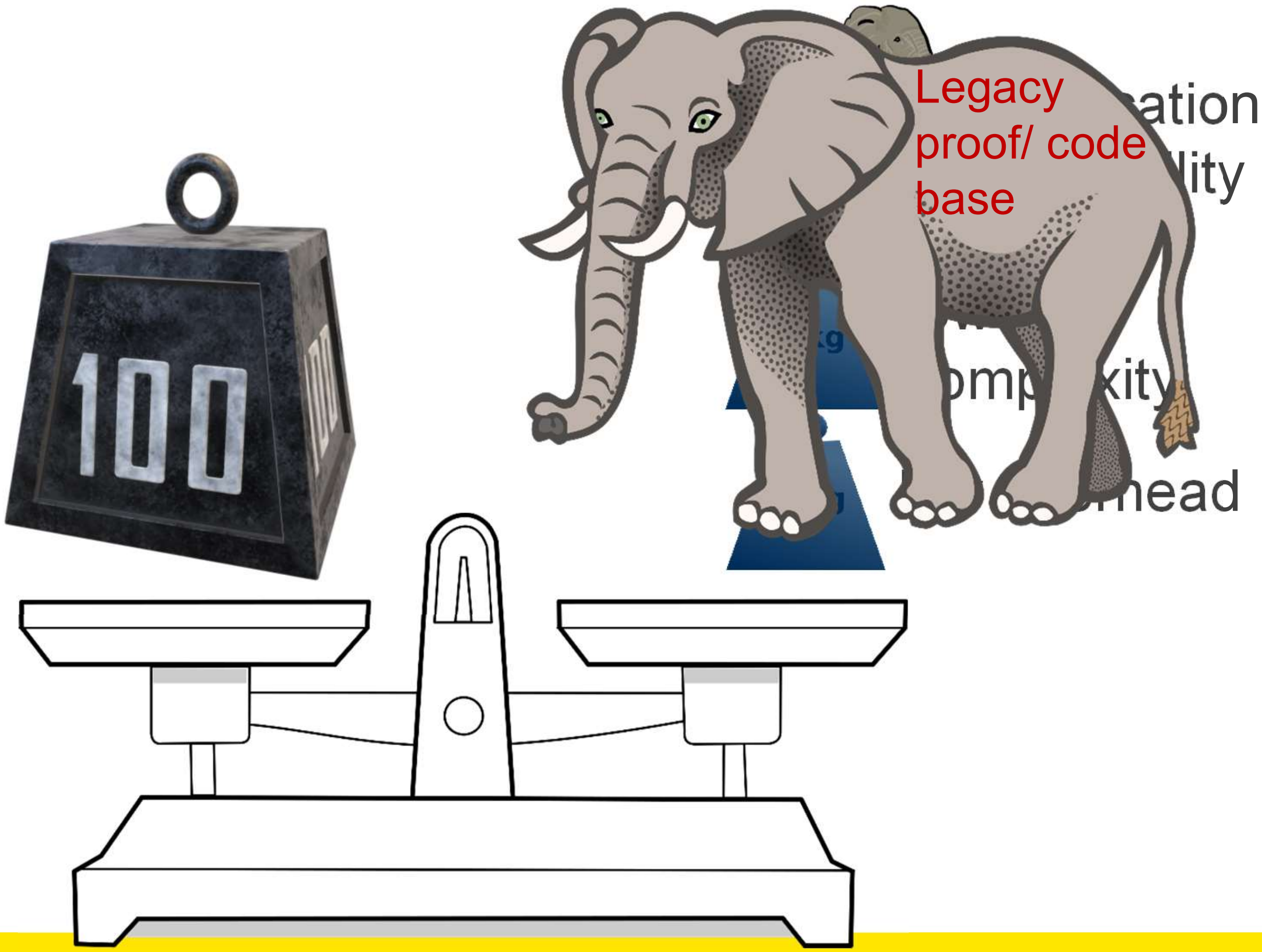
The solution: mutual exclusion



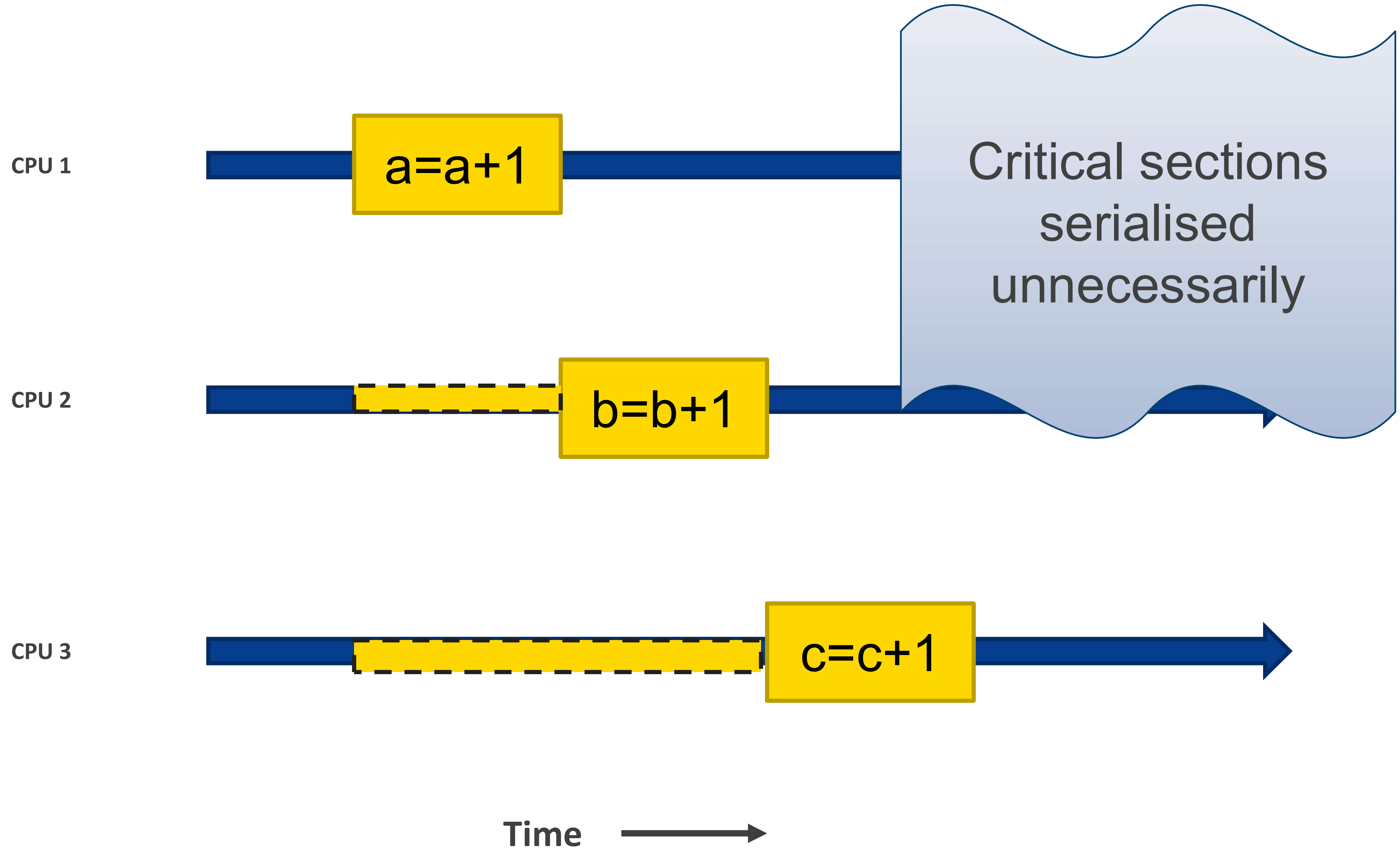
Synchronisation granularity

Fine-grained / lock-free — Coarse-grained

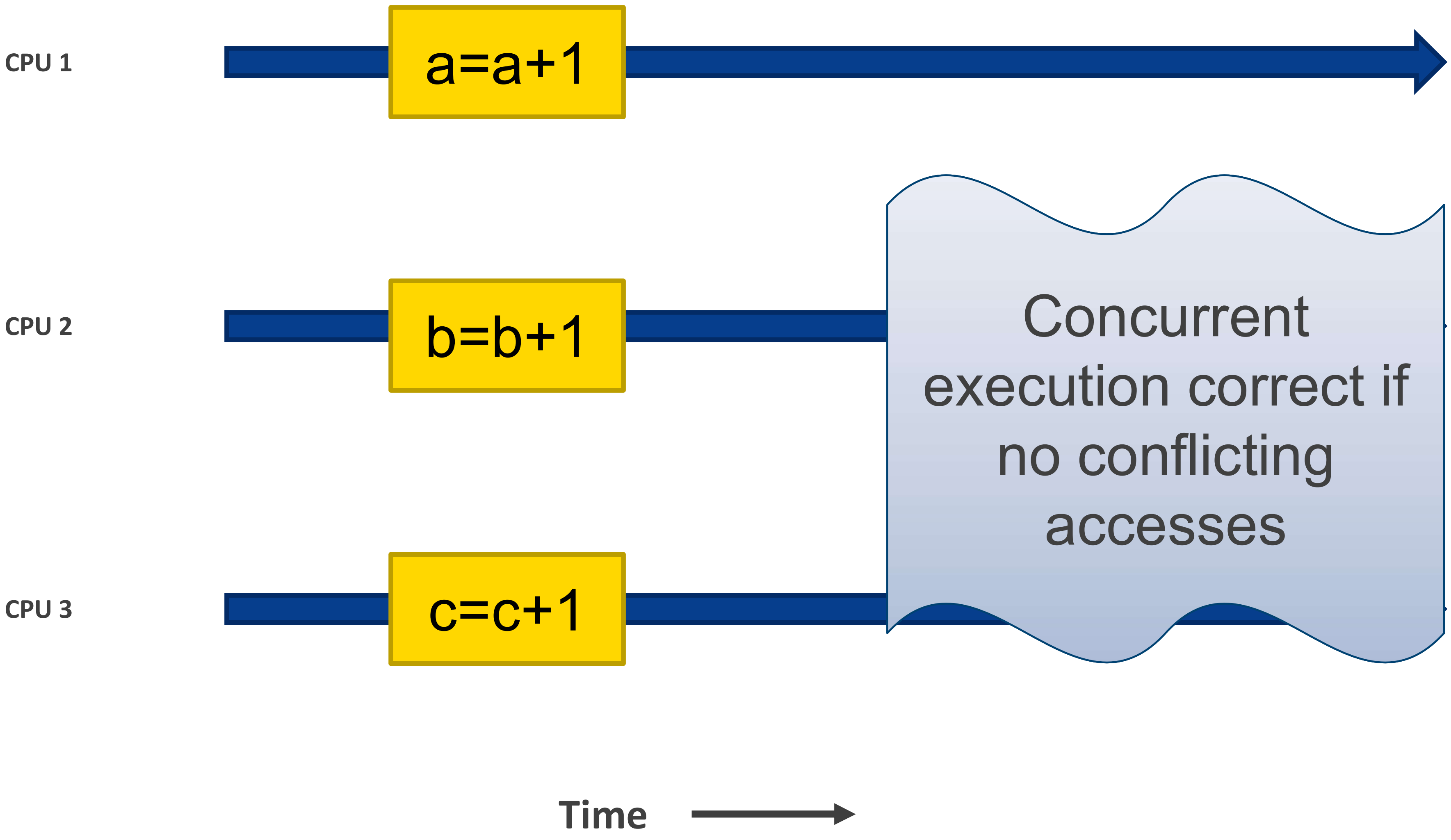
good scalability



Course-grained mutual exclusion



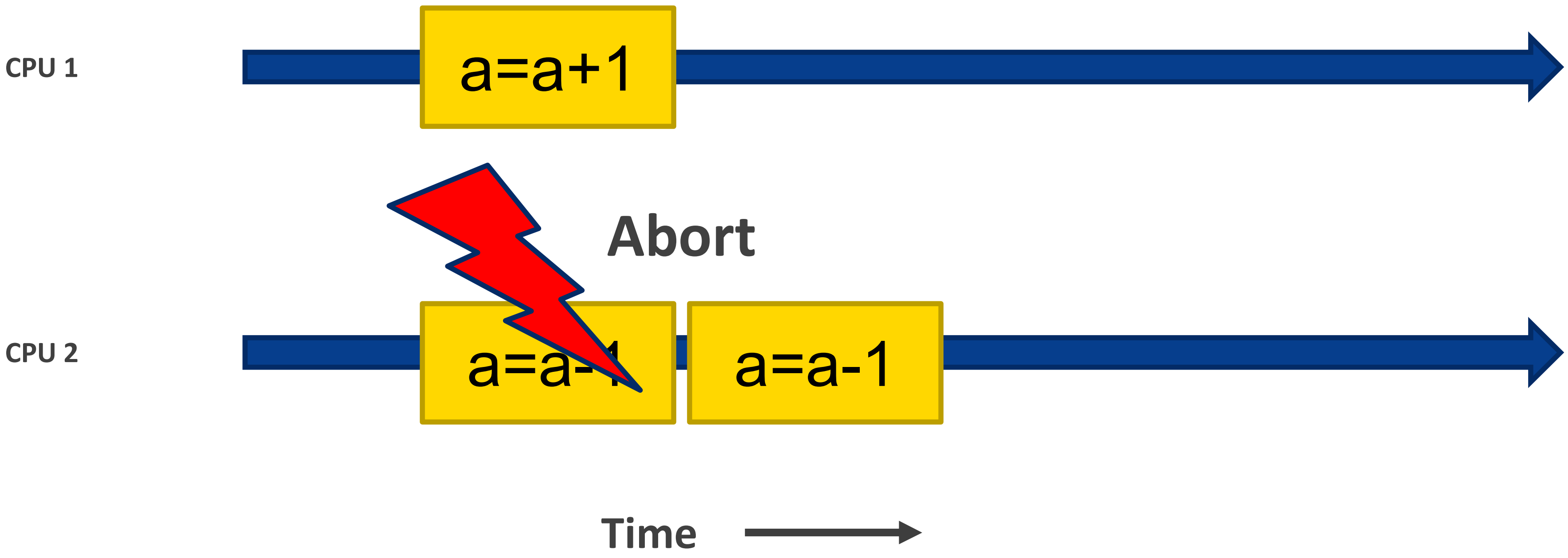
Optimistic concurrency



Transactional Memory

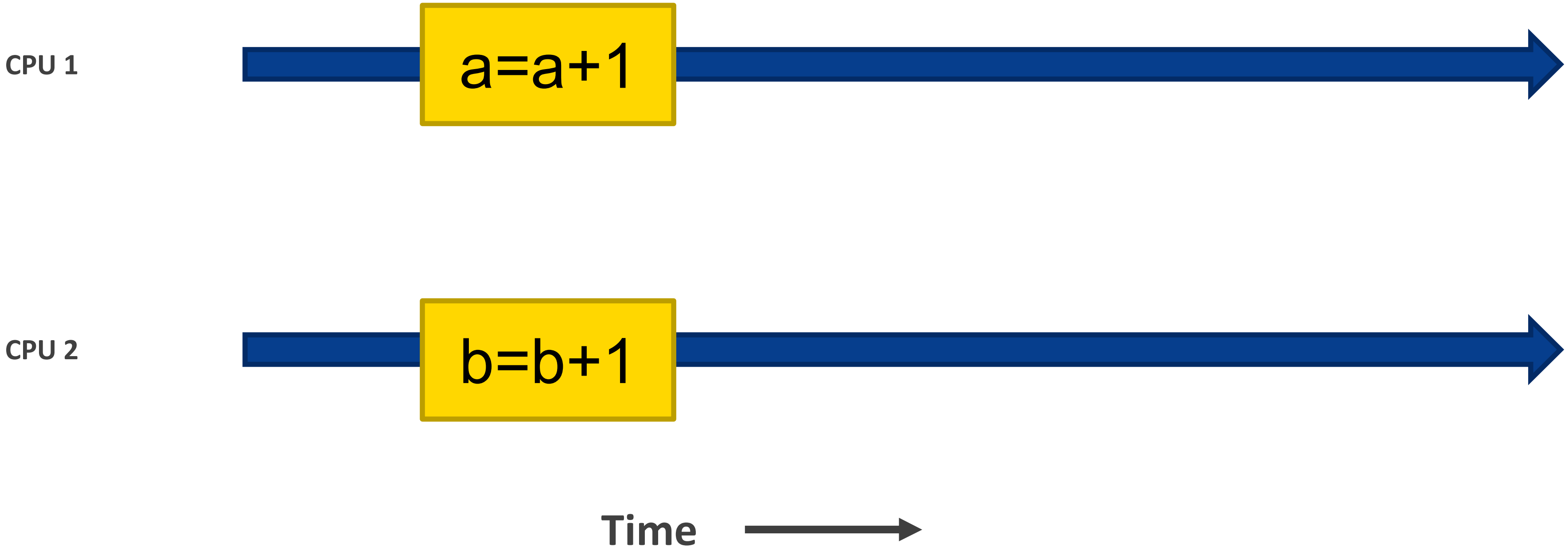
- A transaction is a sequence of machine instructions satisfying the following properties:
  - Serializability:
    - Transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another.
    - Committed transactions are never observed by different processors in different orders.
  - Atomicity:
    - Each transaction makes a sequence of tentative changes to shared memory.
    - A transactions can *commits*, making its changes visible to other processors
    - Or a transaction aborts, causing its changes to be discarded.

Transactions



- Updates only visible locally
- Commit publishes update if conflict free

Transactions



Conflict detection

Hardware maintains:

- *Read set:* The set of all memory addresses loaded from
- *Write set:* The set of all memory addresses stored to
  - The write set is not visible to other CPUs until a successful commit

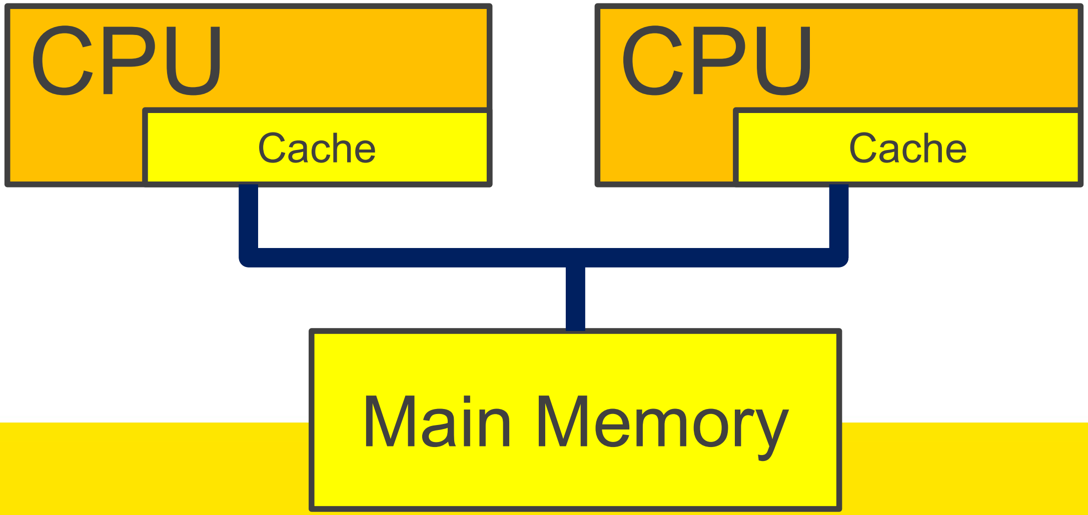
A transaction is conflict free if:

- No other processor reads a location that is part of the transactional region's write-set
- And, no other processor writes a location that is a part of the read- or write-set of the transactional region.

Implementation Intuition

- Cache coherence protocol already coordinates reads and writes to cache lines
- Write-back caches could isolate updates until successfully committed

→ Implement transactions by augmenting cache hardware



Some Papers

Herlihy, Maurice / Moss, J. Eliot B.

Transactional Memory: Architectural Support for Lock-Free Data Structures
1993
Proceedings of the 20th annual international symposium on Computer architecture - ISCA ‘93

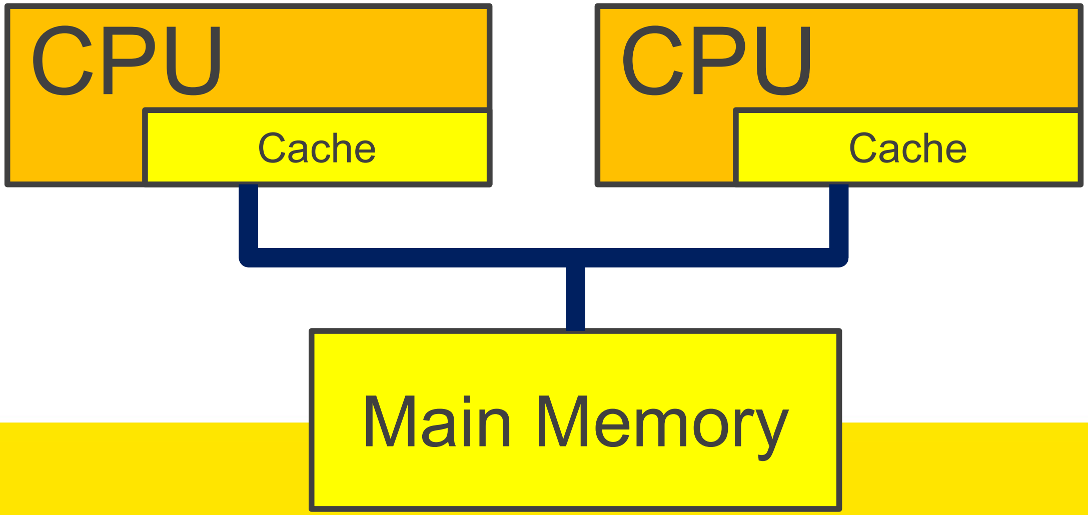
Yoo, Richard M. / Hughes, Christopher J. / Lai, Konrad / Rajwar, Ravi

Performance evaluation of Intel transactional synchronization extensions for high-performance computing
2013
Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC 13

Some Hardware Limitations

Aborts

- Caches are a finite size, transactions will abort if they exceed cache capacity to manage read and write set
- High contention on transaction region can trigger repeated aborts



Sample Elided Lock

Elided lock:

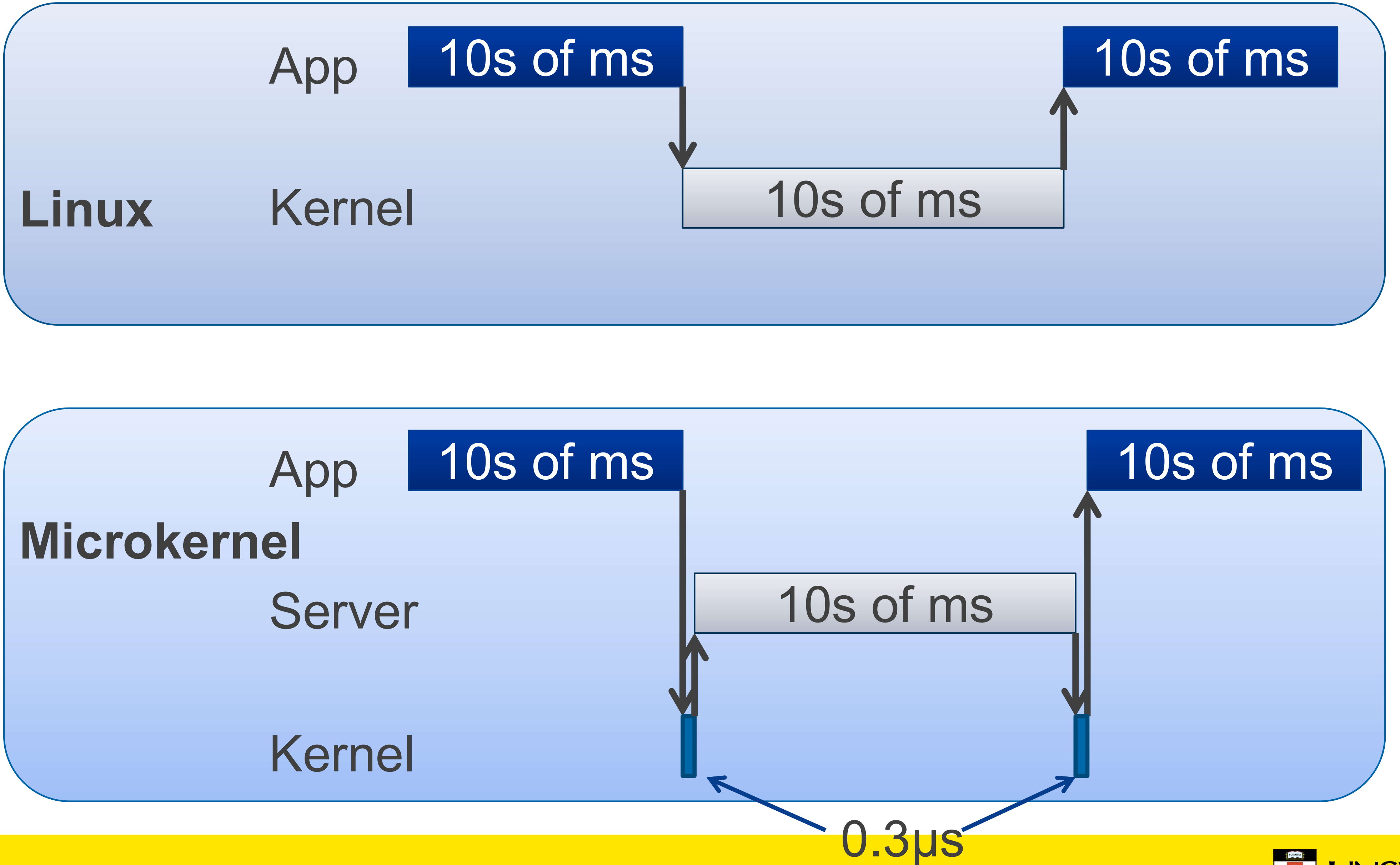
```
/* Start transactional region. On abort we come back here. */
if (_xbegin() == _XBEGIN_STARTED) {
        /* Put lock into read-set and abort if lock is busy */
        if (lock variable is not free)
                _xabort(_XABORT_LOCK_BUSY);
} else {
        /* Fallback path */
        /* Come here when abort or lock not free */
        lock lock;
}
/* Execute critical region either transaction or with lock */
```

Elided unlock:

```
/* Critical region ends */
/* Was this lock elided? */
if (lock is free)
        _xend();
else
        unlock lock
```

UNSW SYDNEY

Microkernel vs Linux Execution



UNSW SYDNEY

Experiments with seL4 and Intel TSX

Basic idea: put the kernel in a transaction

- Coarse-grained transaction
- Fallback on BKL

Microkernel small enough to fit in a transaction

Repeated non-conflicting parallel IPC benchmark

None: No concurrency control

Fine-grained scales well

- Expected

RTM also scales well

- Extremely low abort rates