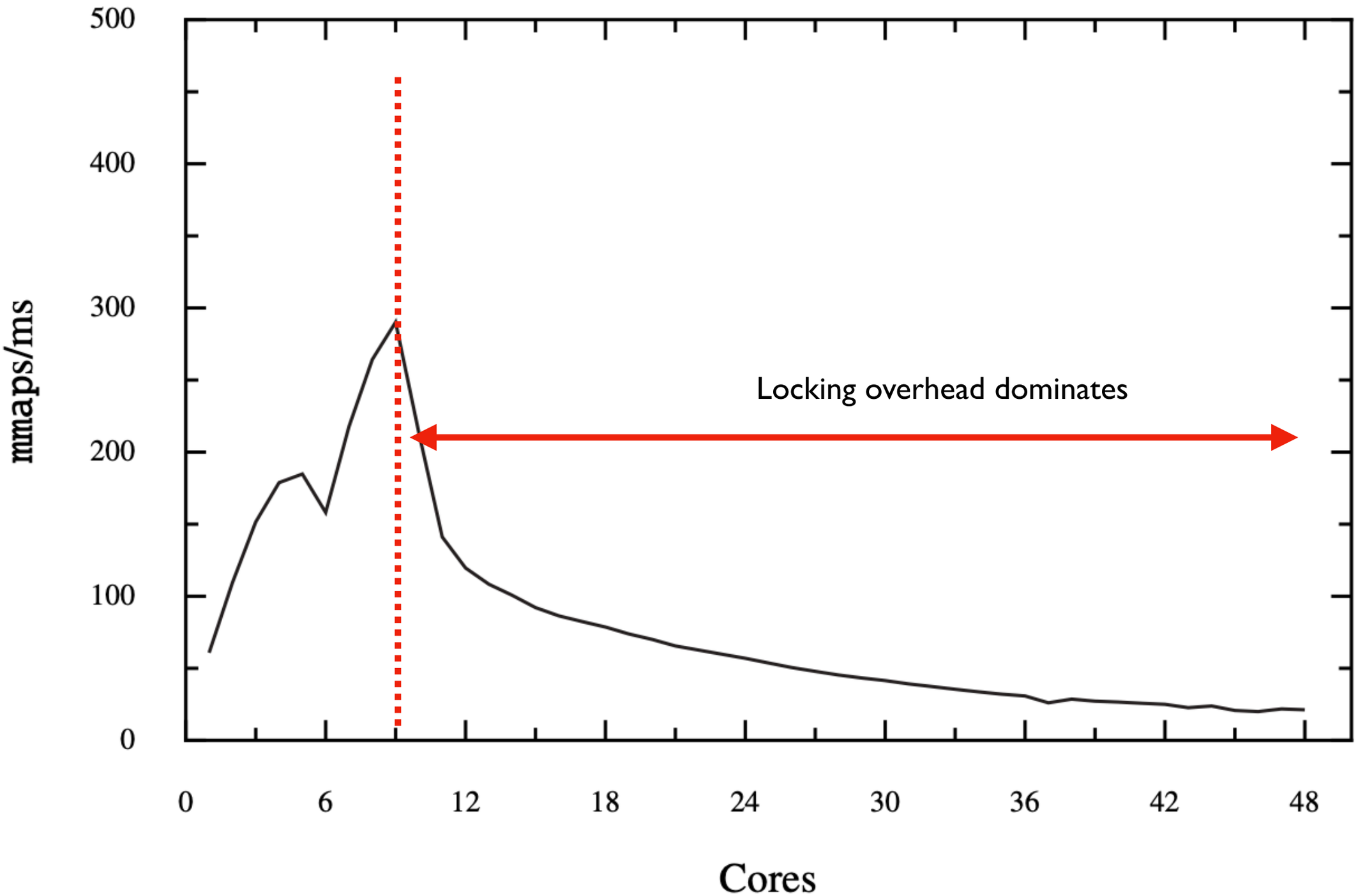
CS 134
Operating Systems

April 15, 2019

Scalable Locking

Problem: locks can ruin performance



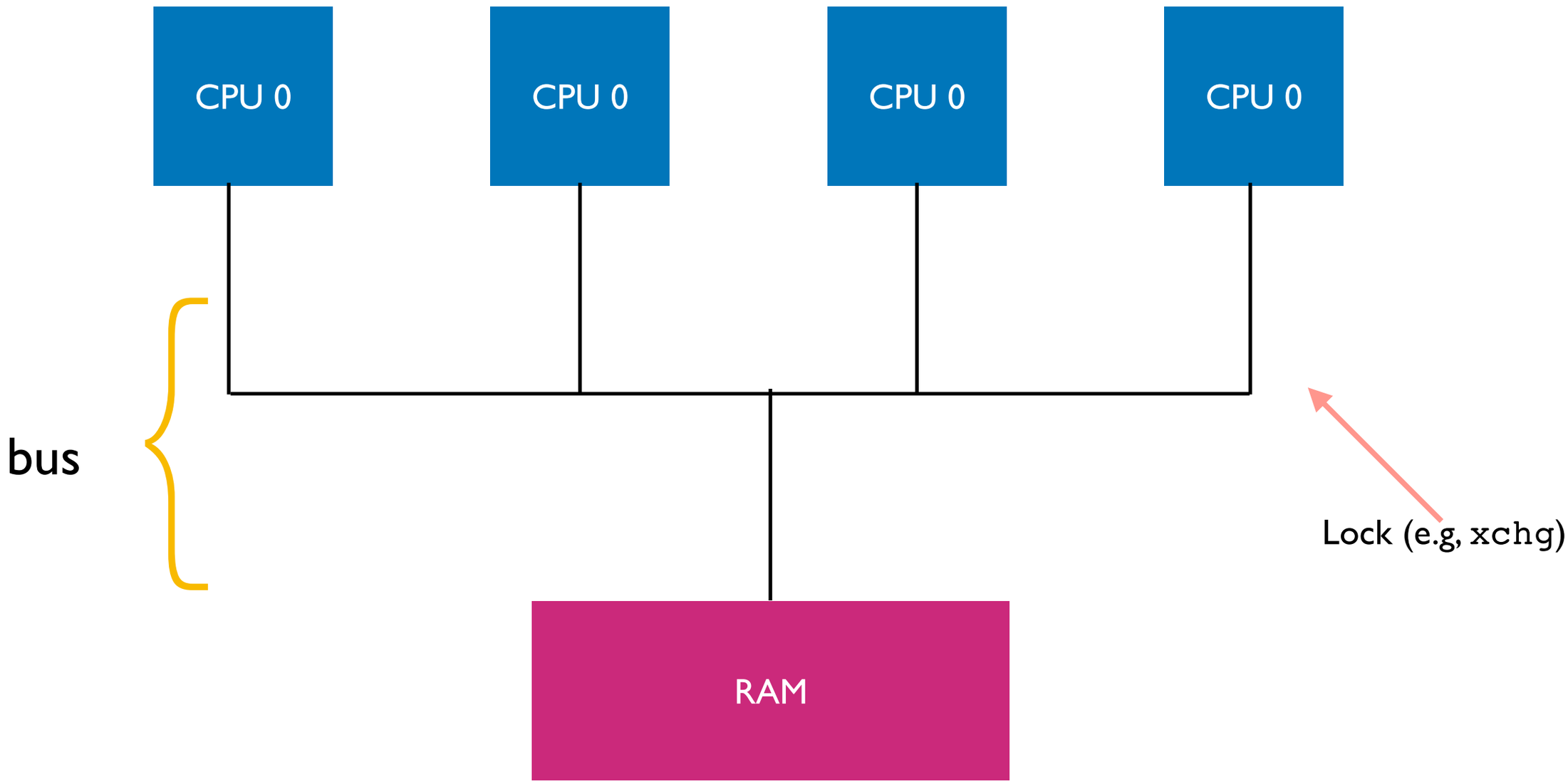
(b) Collapse for MEMPOP.

Problem: locks can ruin performance

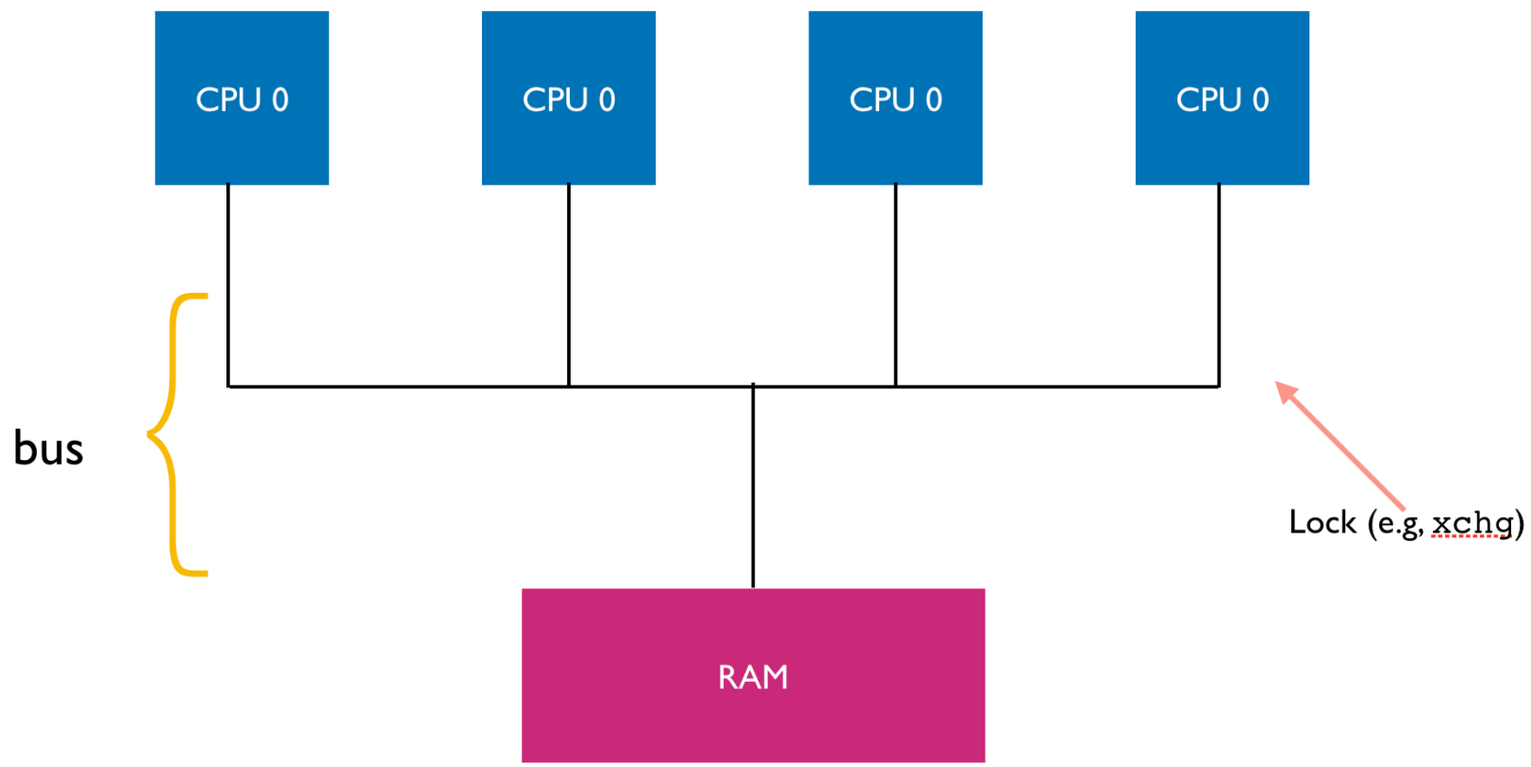
- The locks themselves prevent us from harnessing multi-core to improve performance
  - Ahmdal's law: if serial time is s%, then speedup with N processors is limited to $\frac{1}{s\,\%}$
- This "non-scalable lock" phenomena is important. Why it happens is interesting and worth understanding
- The solutions are clever exercises in parallel programming

- The locking bottleneck is caused by interaction with multicore caching

Abstract version of locking primitive

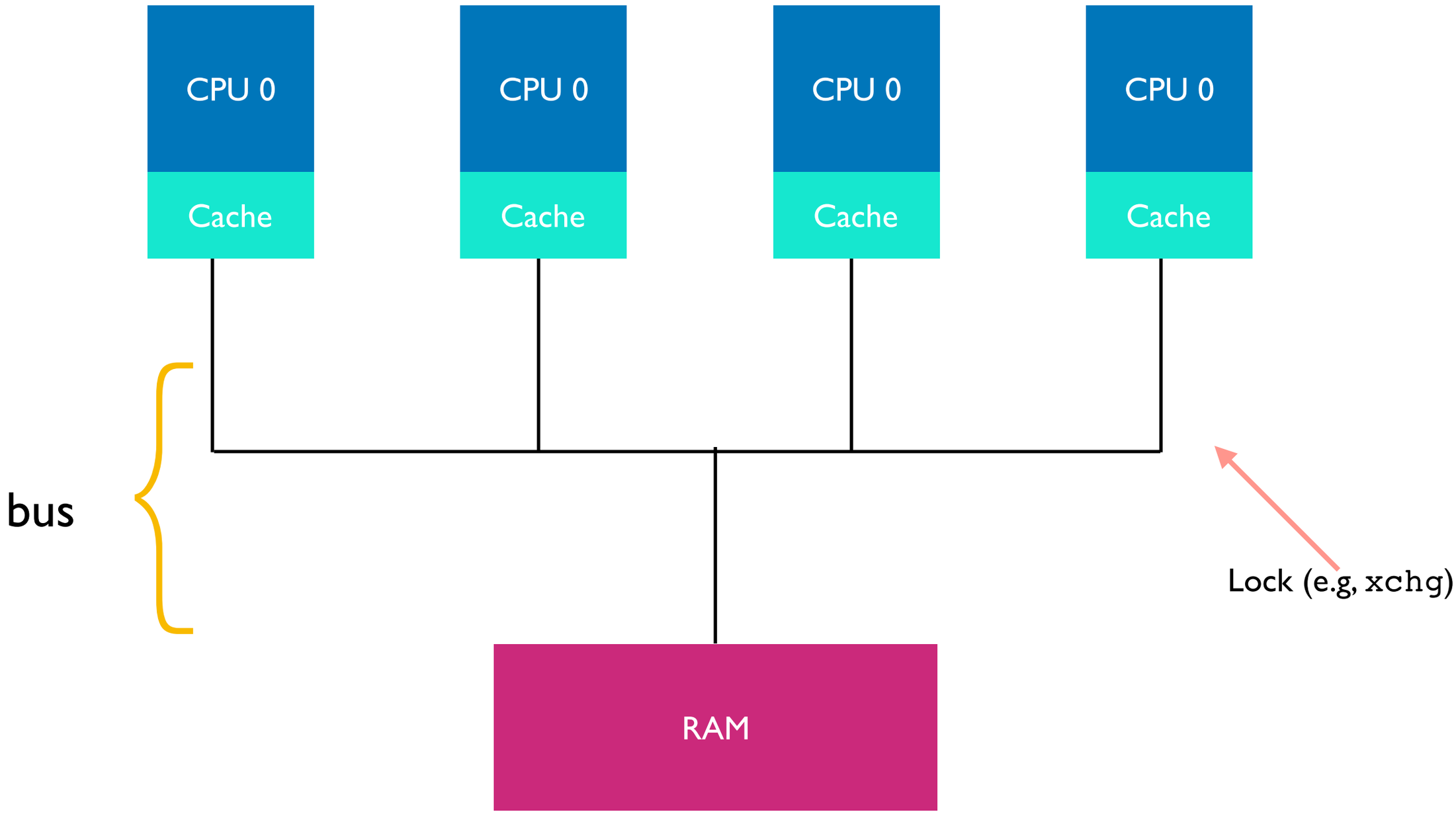


Only an abstraction



- RAM is much slower than processor; need to cache RAM
- **Cache consistency:** *order* of reads and writes between memory locations
- **Cache coherence**: data *movement* caused by reads and writes of a single memory location

Less-abstract version



How does cache coherence work?

- Many different possibilities: here's one
  - Divide cache into fixed-size chunks: *cache lines*
  - Each cache-line is 64 bytes and is in one of 3 states:
    - **M**odified
    - **S**hared
    - **I**nvalid
  - Cores exchange messages as they read and write:
    - **`invalidate(addr)`**: delete from your cache
    - **`find(addr)`**: does any core have a copy?
    - all messages are broadcast to all cores

MSI state transitions

- **I**nvalid:
  - On CPU read:
    - `find`
    - Read from main memory
    - set to **S**hared
  - On CPU write:
    - `invalidate`, then set to **M**odified
- On `find`:
  - do nothing
- On `invalidate`:
  - do nothing

MSI state transitions

- **S**hared:
  - On CPU read:
    - do nothing
  - On CPU write:
    - `invalidate`, then set to **M**odified
  - On `find`:
    - do nothing
- On `invalidate`:
  - set to **I**nvalid

MSI state transitions

- **M**odified:
  - On CPU read:
    - do nothing
  - On CPU write:
    - do nothing
  - On `find`:
    - write cached value to main memory
    - set to **S**hared
  - On `invalidate`:
    - set to **I**nvalid

Compatibility of states between cores

- Invariants for a given cache line:
  - At most one core can be in **M** state
  - Either one **M** or many **S**, never both

	M	**S**	**I**
M	N	N	Y
S	N	Y	Y
I	Y	Y	Y

What access patterns work well?

- Multiple reads from different cores
  - All in **S**hared state, cached in each core
  - Reads (after the first one) don't require any communication

- One core repeatedly writing
  - **M**odified state gives that core exclusive access
  - Reads and writes (after the first one) don't require any communication

Still a simplification

- Real CPUs use more complex state machines
  - MESI, MOESI
  - Does this for few bus messages and reduces broadcasting
- Real CPUs have complex interconnects
  - Buses are broadcast domains; don't scale
  - On-chip network for communication within die:
  - Off-chip network for communication between dies
    - E.g., Intel QPI (Quick-path interconnect)
- Real CPUs have cache directories
  - Keeps track of which CPUs have copies of data (and the state)

Why locks if we have cache coherence?

- Cache coherence ensures cores read fresh data
- Still have problem with:
  - Read-modify-write cycles
  - Partially-updated data structures
- Locks solve these

Locks are built from atomic instructions

- `XCHG` (x86) used in JOS and xv6
- Many other atomic operations:
  - Test-and-set
  - Add
  - Compare-and-swap
- How does hardware implement atomic instruction?
  - Get the cache line in **M**odified state
  - Defer coherence messages (e.g, `find`)
  - Do the read and write
  - Resume handling messages

Locking performance criteria

- Assume N cores are waiting for a lock
- How long does it take to handoff from one to another?
- Bottleneck is usually the interconnect
  - So, measure the messages

- What can we hope for?
  - If N cores are waiting, get through them all in O(N) time
  - Each handoff takes O(1) time, does not increase with N

Test & set spinlocks (JOS/xv6)

```
struct lock { int locked; };

acquire(l) {
  while(1) {
    if(!xchg(&l->locked, 1))
      break;
  }
}

Release(l) {
  l->locked = 0;
}
```

Test & set spinlocks (JOS/xv6)

```
struct lock { int locked; };

acquire(l){
  while(1){
    if(!xchg(&l->locked, 1))
      break;
  }
}

Release(l){
  l->locked = 0;
}
```

- Spinning cores repeatedly execute `xchg`
- Problem?
  - Yes
    - OK for cores to waste their own time
    - Bad if waiting cores slow down lock holder
      - Time for critical section and release
        - Holder must wait in line to access the bus
        - So, holder's handoff takes O(N) time

- O(N) time per handoff means all N cores take $O(N^2)$ time

Ticket locks (Linux, in the past)

- Goal: read-only spinning vs. repeated atomic instructions
- Goal: fairness→waiter order preserved
- Key idea: assign numbers, wakeup one waiter at a time

Ticket locks (Linux, in the past)

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

acquire(l) {
  int t = atomic_fetch_and_inc(&l->next_ticket);
  while (t != l->current_ticket){
  }
}

void release(l) {
  l->current_ticket++;
}
```

Ticket locks (Linux, in the past)

- Atomic increment
  - O(1) find message
    - Just once: not repeated
- Then, read-only spin
  - no cost until next release
- What about release?
  - Invalidate message sent to all cores
  - Then, O(N) find messages, as they re-read
- Still O(N) handoff work
- But, fairness and less bus traffic while spinning

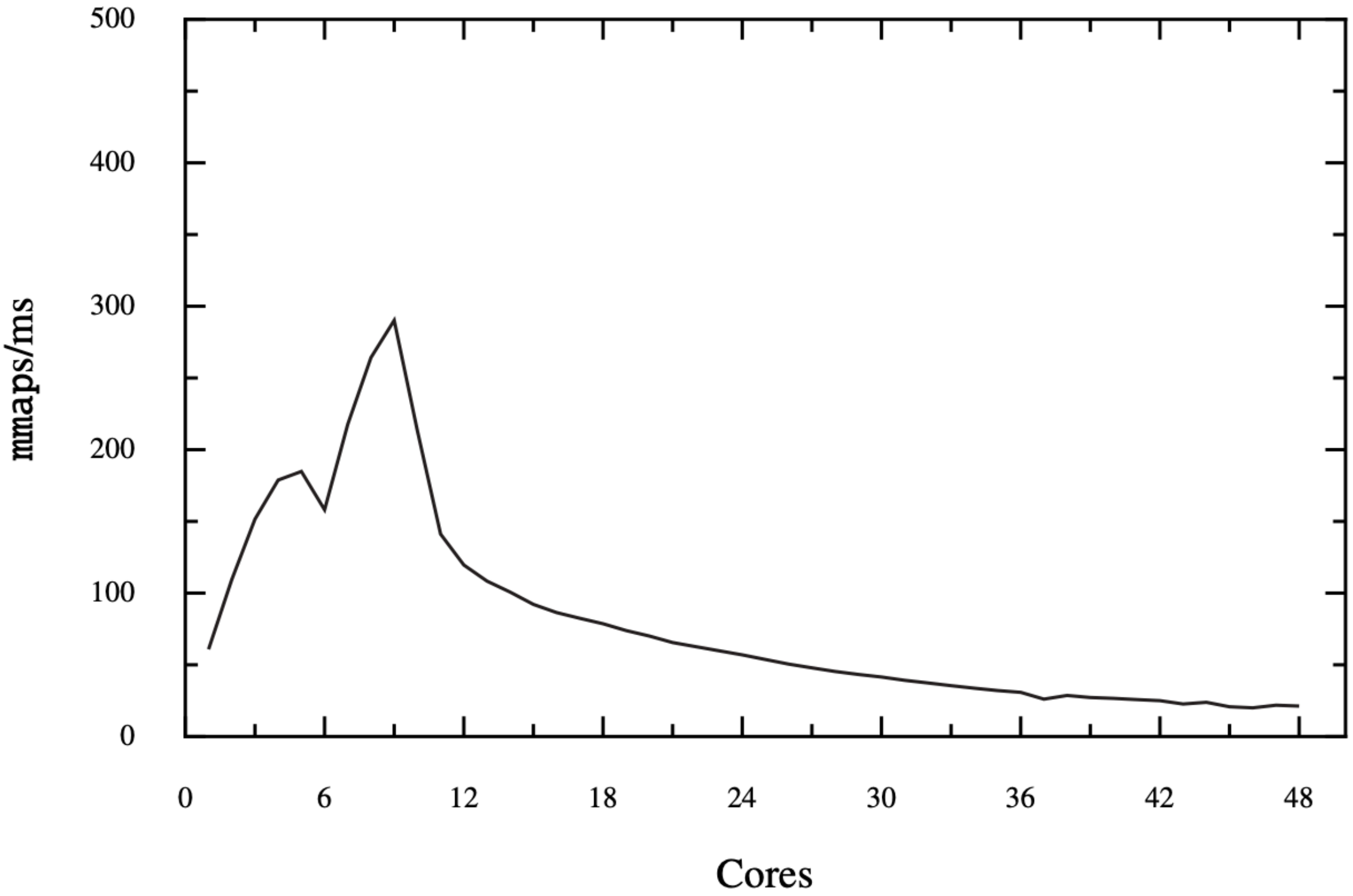
```
struct lock {
  int current_ticket;
  int next_ticket;
}

acquire(l) {
  int t = atomic_fetch_and_inc(&l->next_ticket);
  while (t != l->current_ticket){
  }
}

void release(l) {
  l->current_ticket++;
}
```

Non-scalable locks

- Non-scalable because cost of handoff scales with number of waiters

  - Test-and-set
  - Ticket

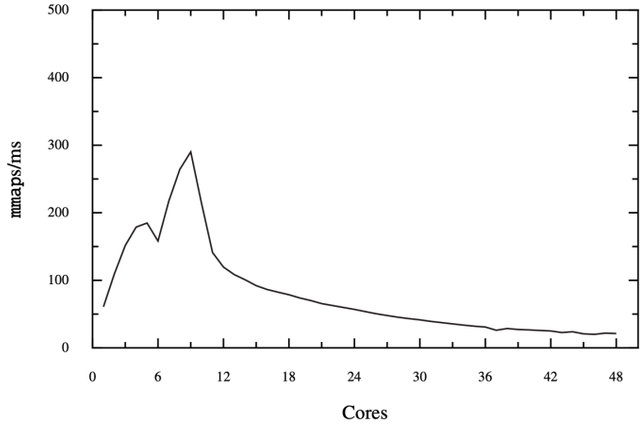
Problem: locks can ruin performance



(b) Collapse for MEMPOP.

Reasons for collapse

- Critical section takes 7% of request time
- You'd expect collapse at 14 cores
- Odd that the collapse happens so soon
- However, once cores waiting for unlock is substantial, critical section + handoff time takes longer
- Slower handoff time makes number of waiters grow



(b) Collapse for MEMPOP.

Small example

```
acquire(&l);
x++;
release(&l);
```

- Uncontended: ~40 cycles
- If a different core used the lock last: ~100 cycles
- With dozens of cores waiting: thousands of cycles

How can we make locks scale?

- Goal: O(1) message release time
- Can we wake just one core at a time?
- Idea: have each core spin on a different cache line

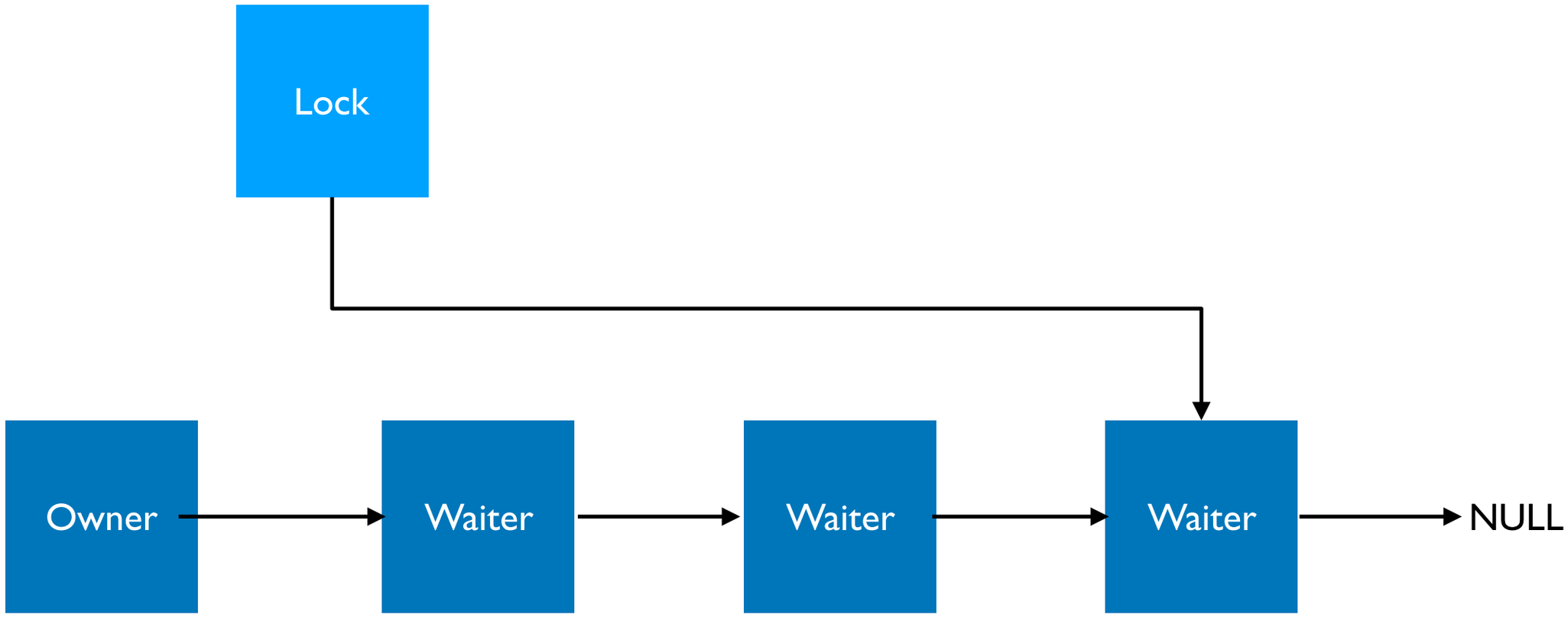
MCS (Mellor-Crummey, Scott) locks

- Each CPU has a `qnode` structure in its local memory:

```
struct qnode {
  struct qnode *next;
  bool locked;
};
```

- A lock is a `qnode` pointer to the tail of the list
- While waiting, spin on the local `locked` flag

MCS locks

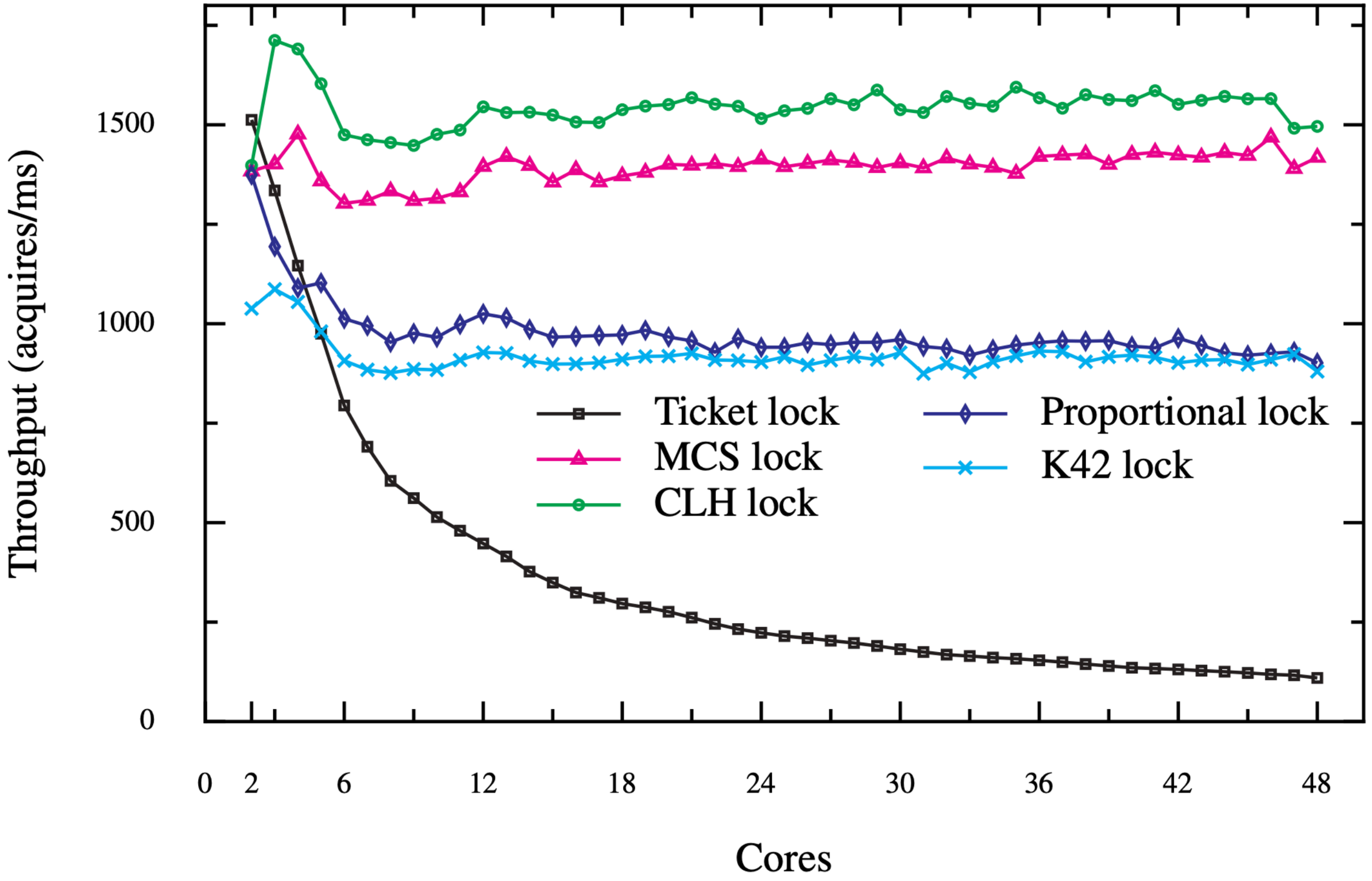


Implementation of MCS locks

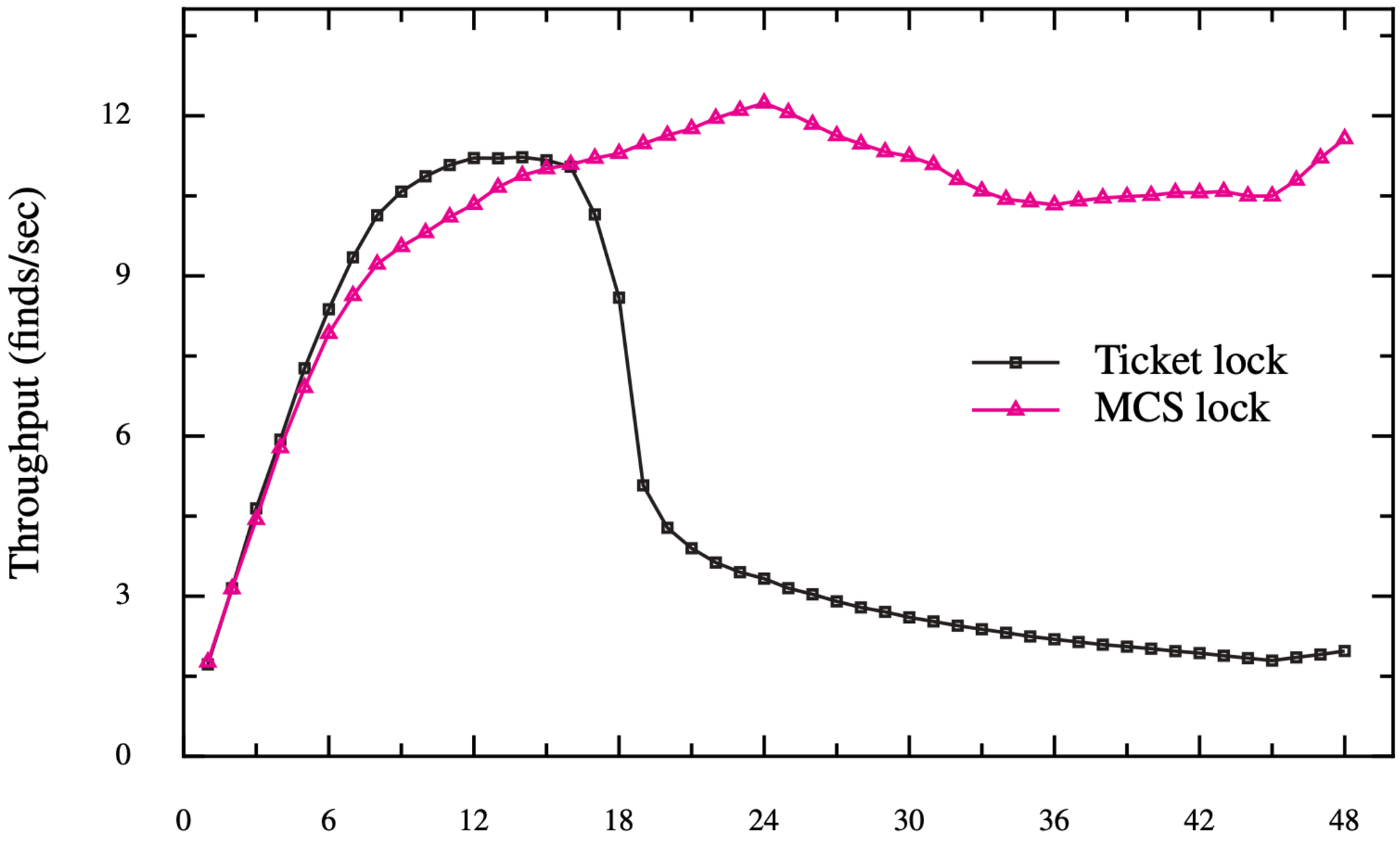
```
acquire(lock *L, qnode *I) {
  I->next = NULL;
  qnode *predecessor = I;
  XCHG(*L, predecessor);
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    while (I->locked) ;
  }
}
```

```
release(lock *L, qnode *I) {
  if (!I->next) {
    if (compare-and-swap(*L, I, NULL))
      return;
  }
  while (!L->next) {
  }
  l->next->locked = false;
}
```

Locking strategy comparison



But, not a panacea



(c) Performance for PFIND.

Conclusion

- Scalability is limited by the length of the critical section
- Scalable locks can only avoid collapse
- Preferable to use algorithms that avoid contention altogether
- Example in next lecture

Questions

- How hard it is to modify existing code to use scalable locks?
- Have kernel developers actually changed the locks they use in response to the paper?
- Does JOS/xv6 use locks only within a CPU, or do they share locks between multiple CPUs?
- How do we define the critical section which determines the time taken to transfer lock ownership?
- What defines a non-scalable lock?
- What causes the sudden dropoff in performance for ticket locks?

Questions

- How does proportional backoff work with ticket locks?
- Paper talks about scalable/non-scalable locks. Are there other types?
- How does K42 algorithm work (no API changes)?
- Are the test programs pathological or do they represent typical processes on a 48-core machine?
- How do MCS locks guarantee a constant number of cache misses each time a core tries to acquire a lock?
-