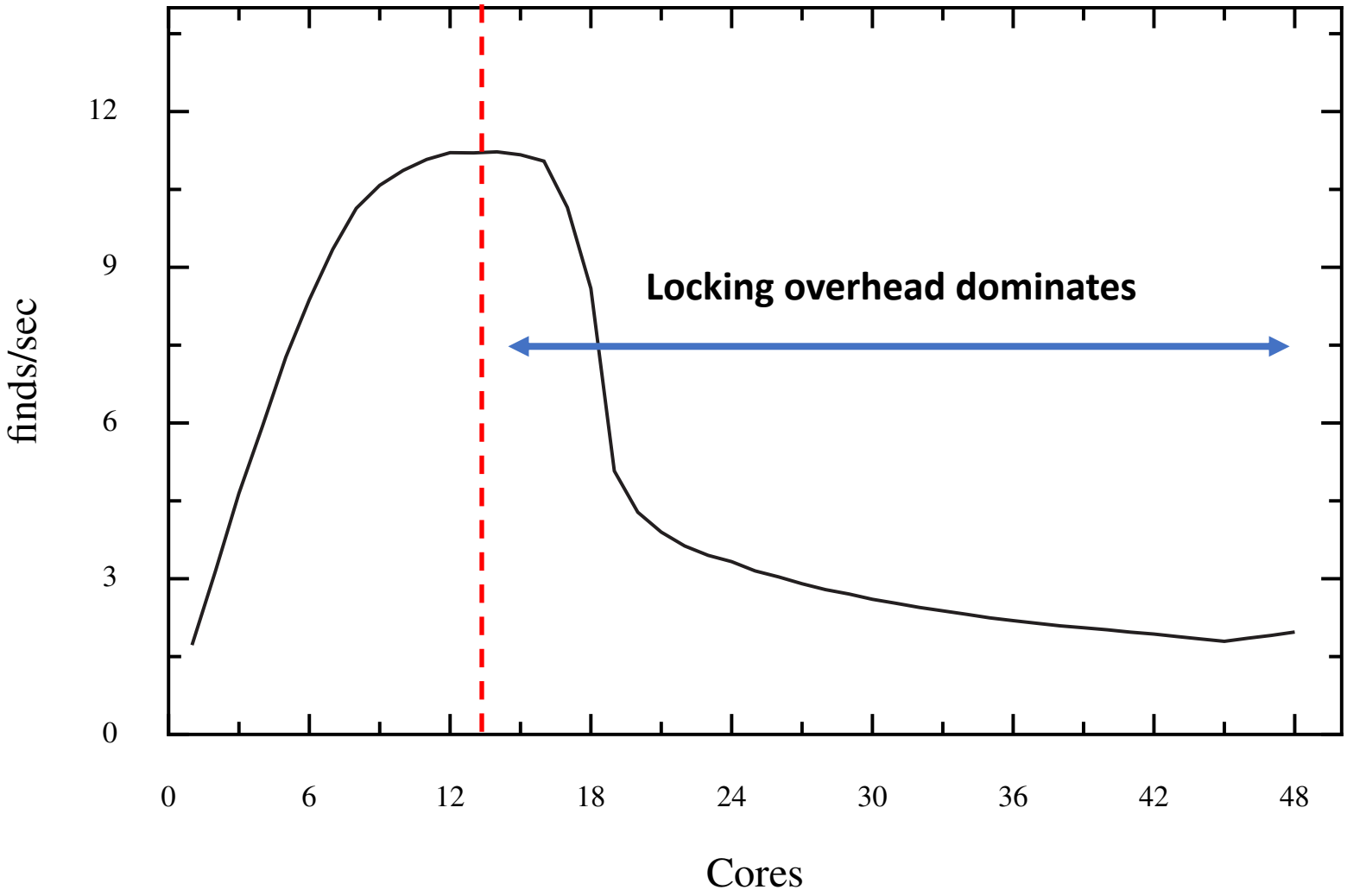
Scalable Locking

Adam Belay <abelay@mit.edu>

Problem: Locks can ruin performance

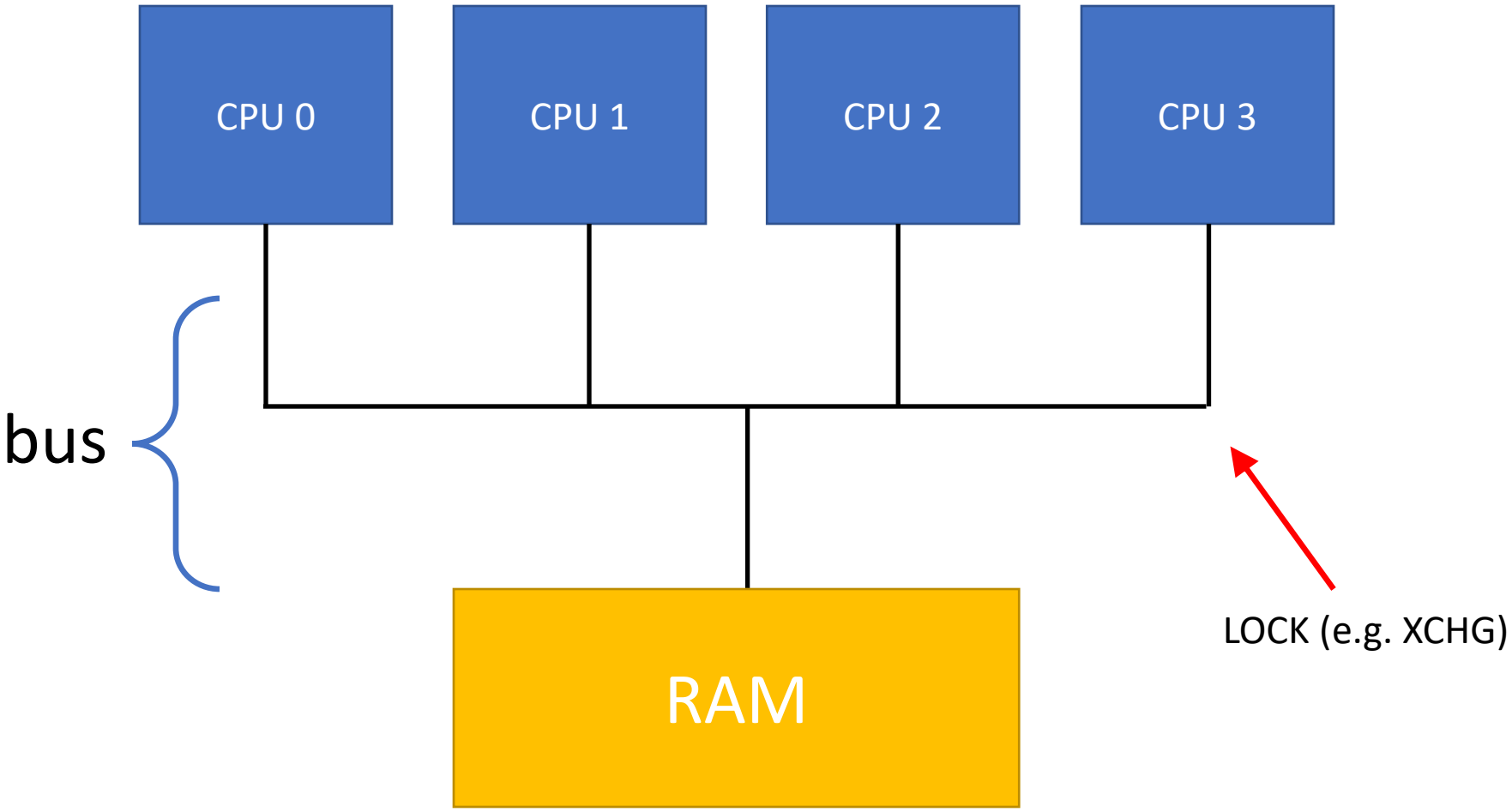


Problem: Locks can ruin performance

- Locks themselves can prevent us from harnessing multi-core to improve performance
- Why it happens is interesting and worth understanding
- The solutions are clever exercises in parallel programming

Locking bottleneck is caused by interaction with multicore caching

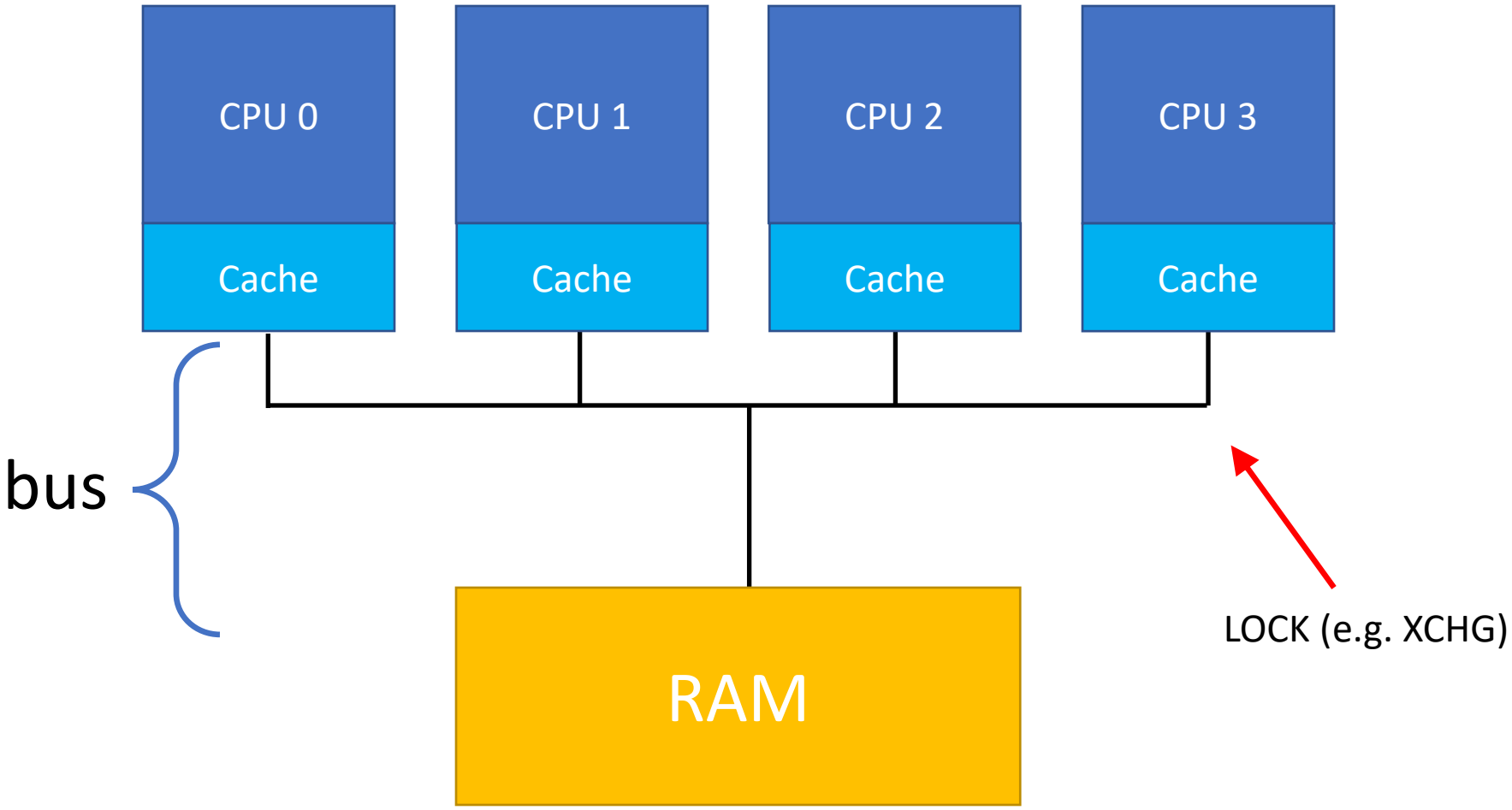
Recall picture from last time



Not how multicores actually work

- RAM is much slower than processor, need to cache regions of RAM to compensate
- **Cache Consistency:** *order* of reads and writes between memory locations
- **Cache Coherence:** data *movement* caused by reads and writes for a single memory location

Imagine this picture instead…



How does cache coherence work?

- Many different schemes are possible
- Here's a simple but relevant one:
  - Divide cache into fixed-sized chunks called "**cache-lines**"
  - Each cache-line is 64 bytes and in one of three states: (**M**)odified, (**S**)hared, or (**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

Invalid:

- On CPU read -> find, **Shared**
- On CPU write -> find, invalidate, **Modified**
- On message find, invalidate -> do nothing

MSI state transitions

Shared:

- On CPU read, find -> do nothing
- On CPU write -> invalidate, **Modified**
- On message invalidate -> **Invalid**
- On message find -> do nothing

MSI state transitions

Modified:

- On CPU read and write -> do nothing
- On message find -> **Shared**
- On message invalidate -> **Invalid**

Compatibility of states between cores

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

Invariants:

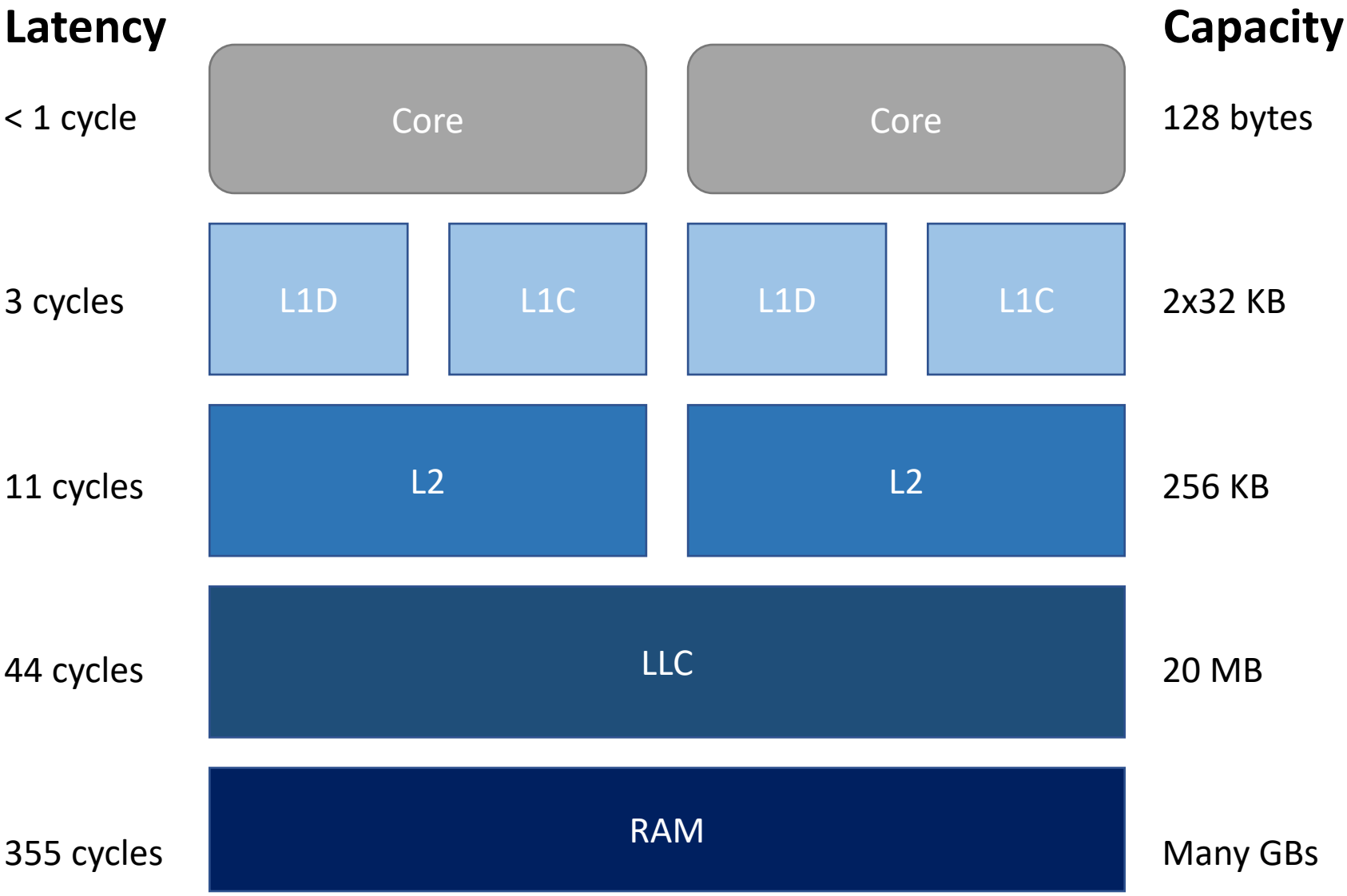
- At most one core can be in **M** state
- Either one **M** or many **S**, never both

What access patterns work well?

What access patterns work well?

- Read only data: **S** allows every core to keep a copy
- Data written multiple times by a core: **M** gives exclusive access, reads and writes are free basically after first state transition

Real caches are hierarchical



Still a simplification

- Real CPUs use complexer state machines (e.g. MOESI)
  - Why? Fewer bus messages, no broadcasting, etc.
- Real CPUs have complex interconnects
  - Buses are broadcast domains, can't scale
  - On-chip network for communication within die
    - Data sent in special packets called "flits"
  - Off-chip network for communication between dies
    - E.g. Intel QPI (Quick-Path Interconnect)
- Real CPUs have "Cache Directories"
  - Central structure that tracks which CPUs have copies of data
- Consider taking 6.823!

Why locks if we have cache coherence?

Why locks if we have cache coherence?

- **cache coherence** ensures that cores read fresh data
- locks avoid lost updates in read-modify-write cycles and prevent anyone from seeing partially updated data structures
- Locks are a tool to control **cache consistency** --- the order of reads and writes

Locks are built from atomic instructions

- So far we use XCHG in xv6 (via AMOSWAP)
- Many other atomic ops, including add, test-and-set, CAS, etc.
- How does hardware implement locks?
  - Get the line in **M** state
  - Defer coherence messages
  - Do all the steps (read and write)
  - Resume handling messages

Locking performance criteria

- Assume N cores are waiting for a lock
- How long does it take to hand off from previous to next holder?
- Bottleneck is usually the interconnect
  - So measure cost in terms of # of messages

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

Test & set (TAS) spinlocks

```
struct lock { int locked; };

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

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

Test & set (TAS) spinlocks

- Spinning cores repeatedly execute atomic exchange
- Is this a problem?
  - Yes!
    - It's okay if waiting cores waste their own time
    - But bad if waiting cores slow lock holder!
  - Time for critical section and release:
    - Holder must wait in line for access to bus
    - So holder's handoff takes O(N) time
- O(N) handoff means all N cores take $O(N^2)$!

Ticket locks (Linux)

- Goal: read-only spinning rather than repeated atomic instructions
- Goal: fairness -> waiter order preserved
- Key idea: assign numbers, wake up one waiter at a time

Ticket locks (Linux)

```
struct lock {
  uint16 current_ticket; uint16 next_ticket;
}

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

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

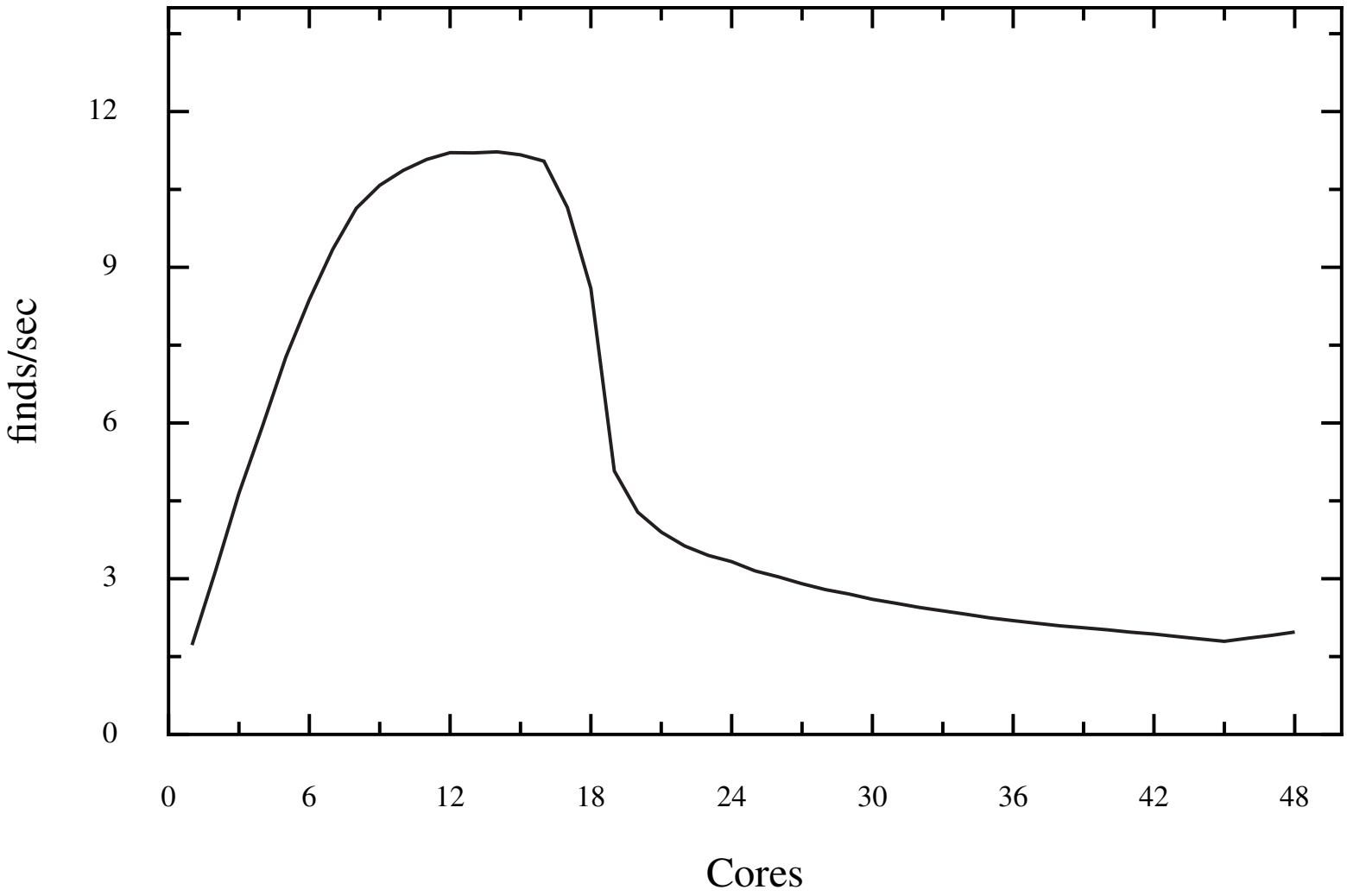
Ticket lock time analysis

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

TAS and Ticket are “non-scalable” locks

Cost of handoff scales with number of waiters

Let’s reconsider figure 2 (PFIND)



Reasons for collapse

- Critical section takes just 7% of request time
  - So with 14 cores, you'd expect just one core wasted by serial execution
- So it's odd that the collapse happens so soon
- However, once cores waiting for unlock is substantial, critical section + handoff takes longer
- Slower handoff time makes N grow even further

Perspective

Consider:

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

- uncontended: ~40 cycles
- if a different core used the lock last: ~100 cycles
- With dozens of cores: thousands of cycles

So 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

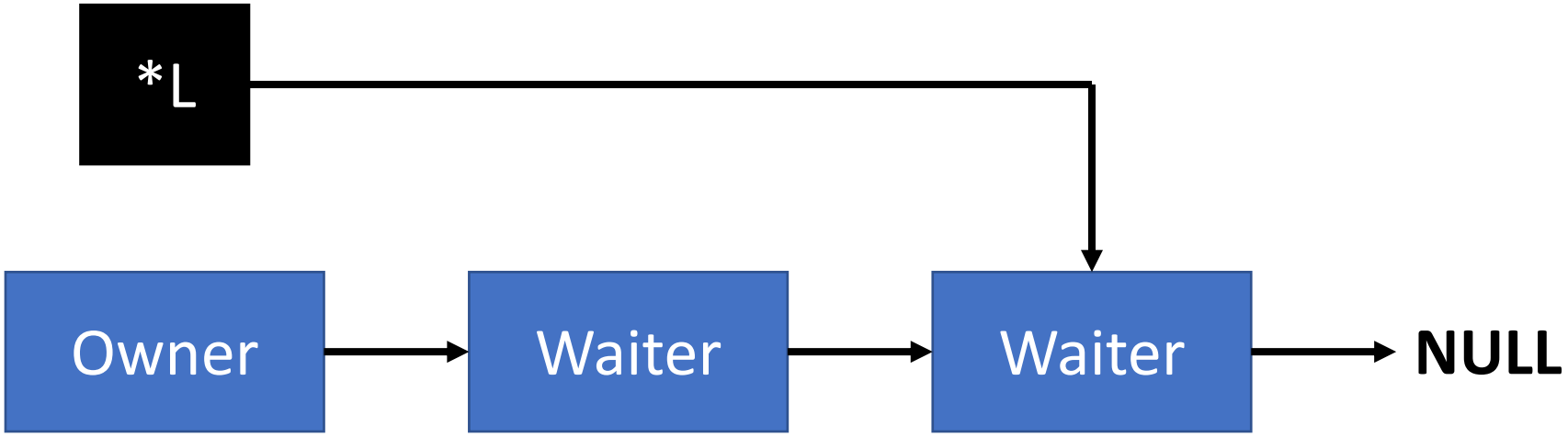
MCS Locks

- Each CPU has a qnode structure in its local memory

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

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

MCS locks



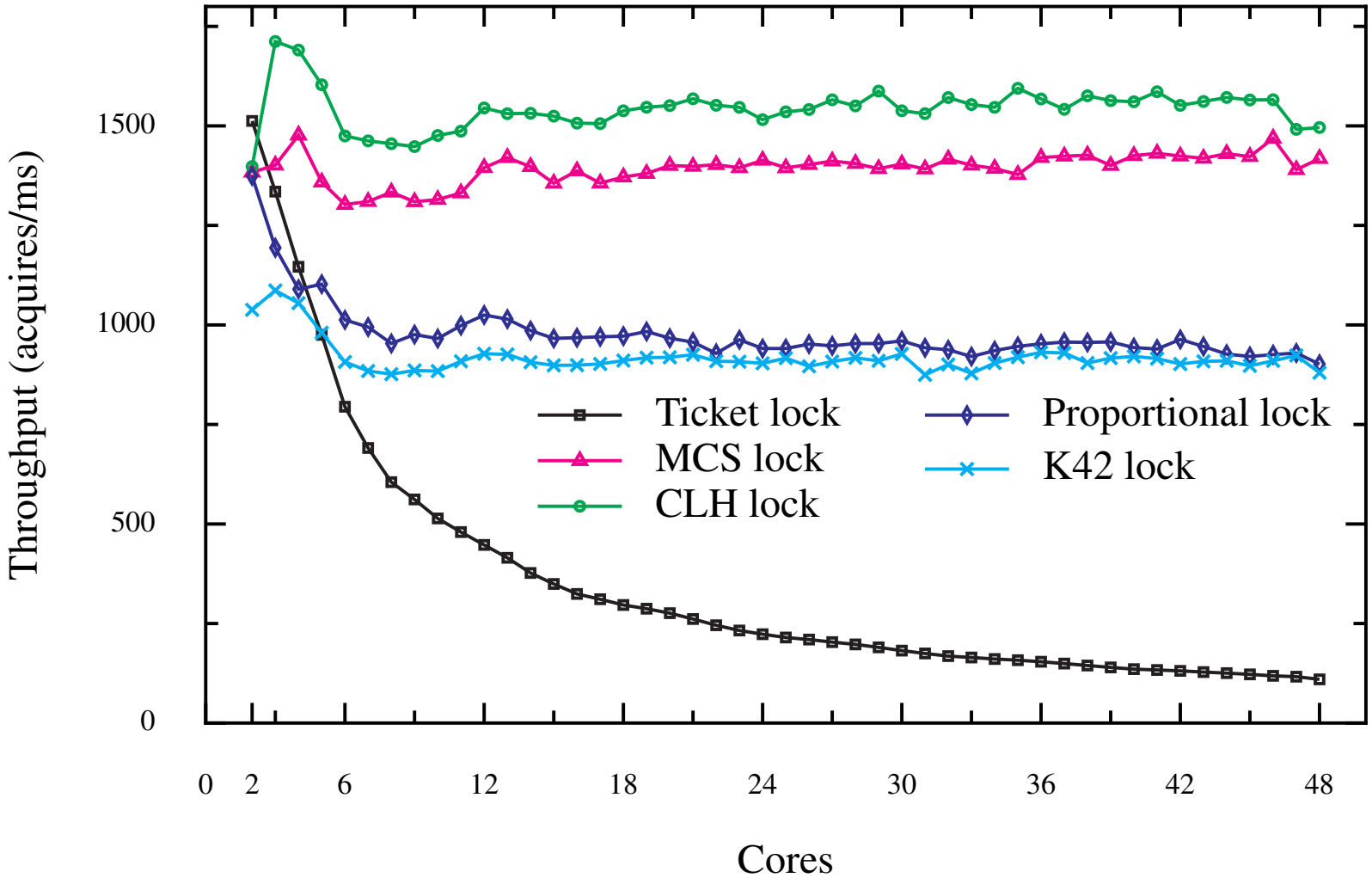
Acquiring MCS locks

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

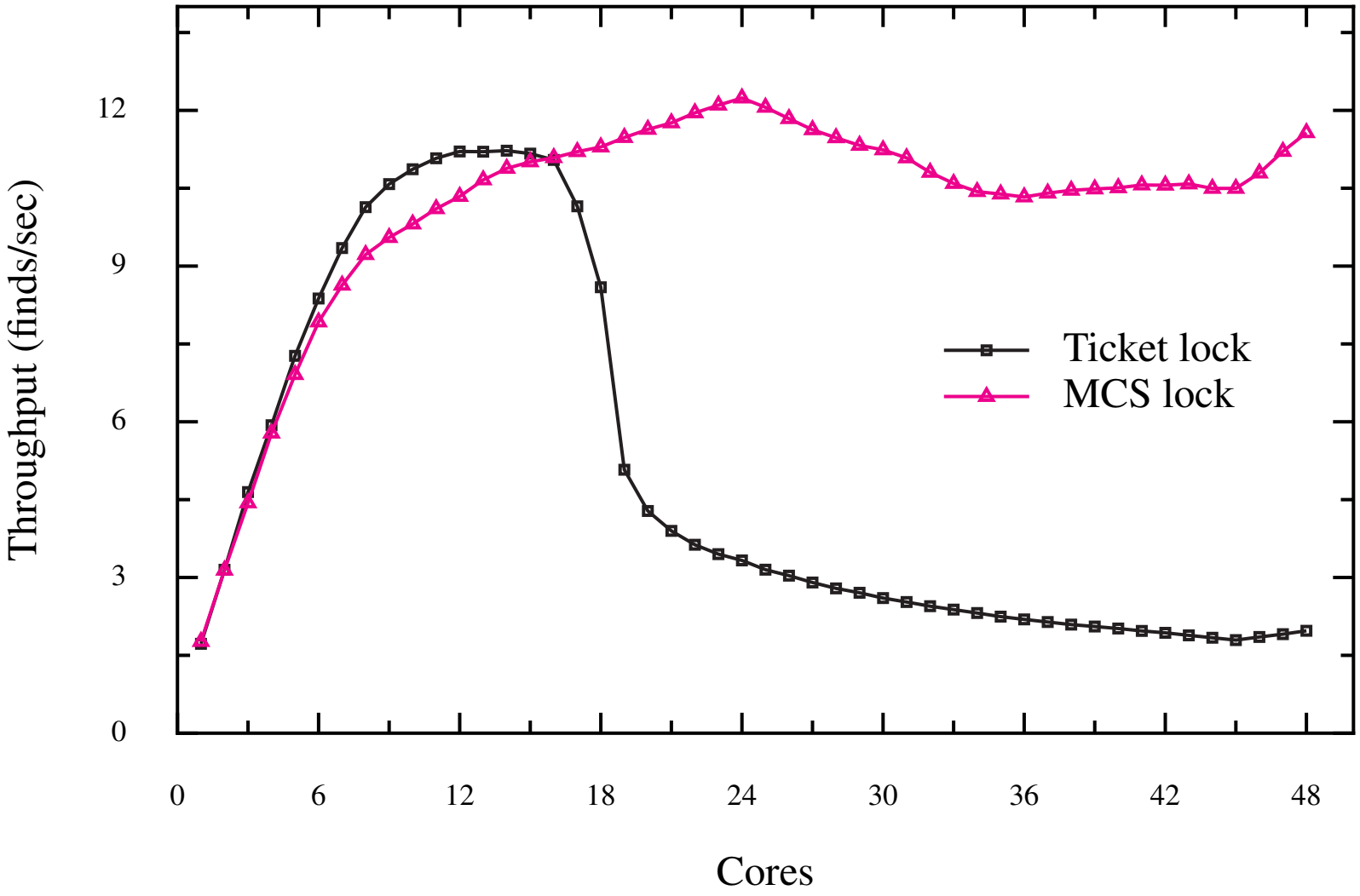
Releasing MCS locks

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

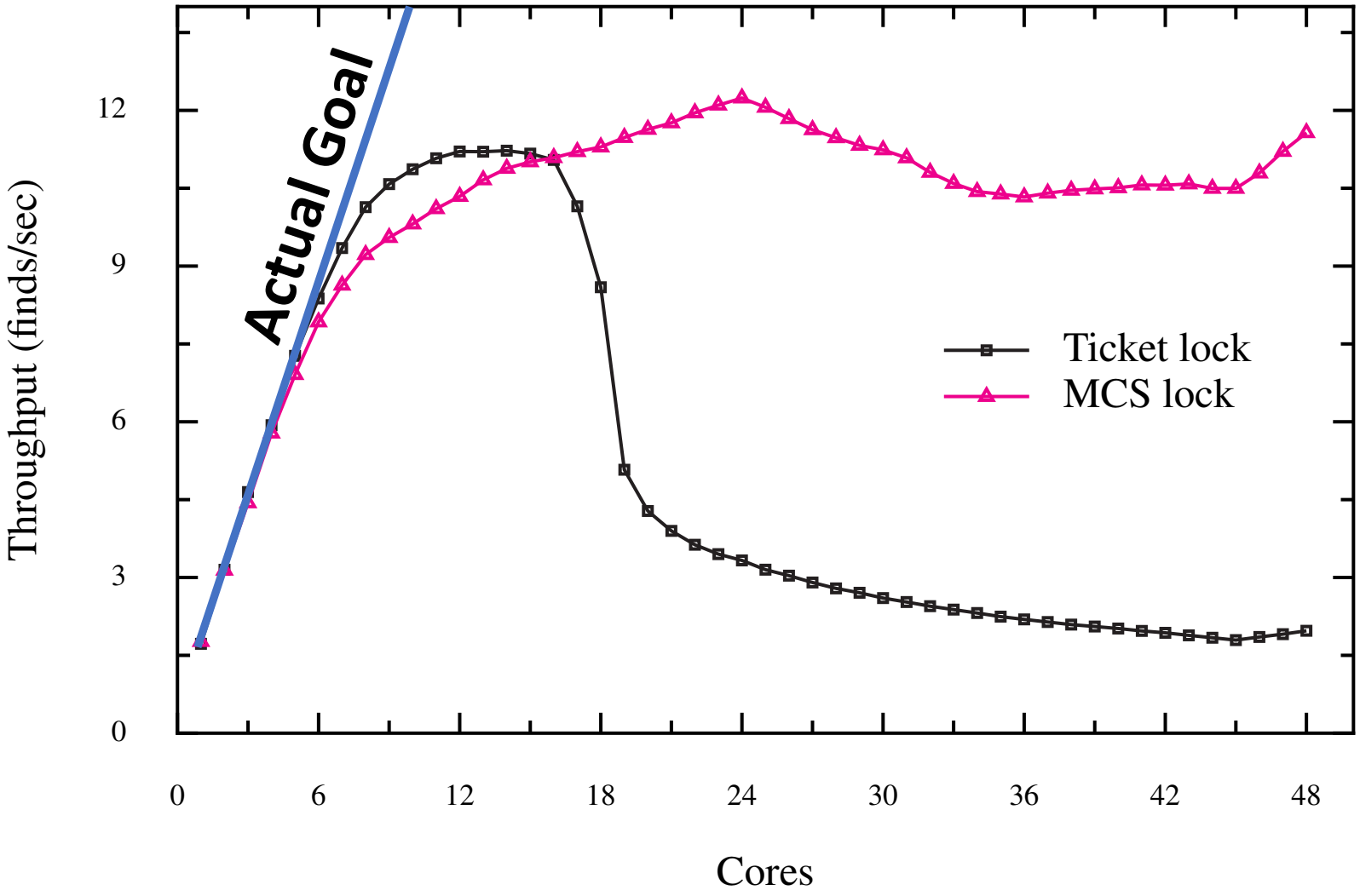
Locking strategy comparison



But not a panacea



But not a panacea



Conclusion

- Scalability is limited by length of critical section
- Scalable locks can only avoid collapse
- Better plan: Use algorithms that avoid contention all together
- Example in next lecture!