# 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*

# This talk

- No “vision”, but tries to answer a natural question
- How to scale an operating system to many cores?
  - What is the right kernel interface?
  - How do you implement a kernel for 100s cores?
- Insights from implementing 3 different kernels
  - Corey, Linux, and xv6
- Research style: focus on hard problems
  - More on this in tomorrow's panel

# Insights about scalability

1. Locks: Short critical sections can lead to sharp scalability collapse

2. Avoiding read locks using Bonsai tree

3. Avoiding write locks: Skip list versus radix tree

4. Achieving perfect scalability (contention-free): scalable commutativity rule for interfaces

➔ Think in terms of cache-line movement

➔ Optimistic about POSIX API scaling

- Insights reflect that research is in progress

# Many cores on a single chip

- Number of cores is 2-10 now
- Moore's law continues
  - doubling of number of cores/18 months

➔ Goal: develop software whose performance scales with number of cores

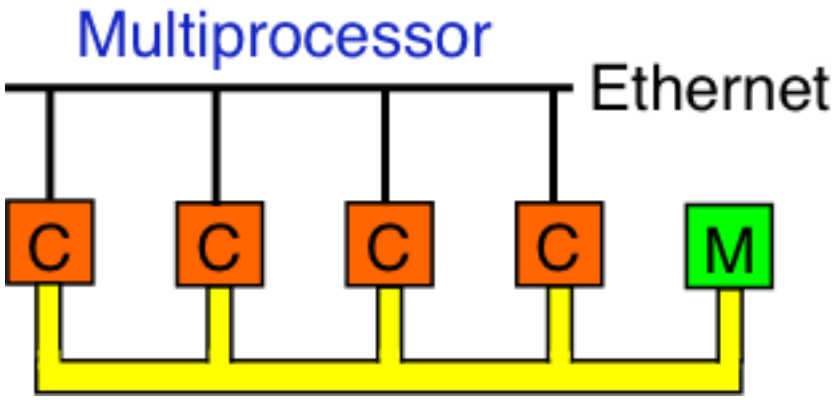
  - Focus on future systems with many cores (e.g., hundreds of cores)
  - Somewhat speculative, but it is research

# Scalability Goal

- Application does N times as much work on N cores as it could on 1 core
- Scalability may be limited by serial sections
  - Locks on shared data structures, ...
  - Shared hardware (DRAM, NIC, …)
- Runtime = P / N + S
  - As N → ∞, runtime = S

# Parallel computing: an old story?

- VU (1987)
- Shared-memory cluster
- 16 68030s (16 Mhz)




* Source: Andy Tanenbaum

# What is new?

- 1987: parallel computing didn't address a burning problem
  - Easier solutions available to obtain performance
  - Clock speed doubled every year
  - 16 x 16 Mhz = 256 Mhz
- Now:  more performance requires parallelism
  - You cannot buy a faster single-core processor
  - Re-birth of parallel computing

# Why look at the OS kernel?

- Many applications spend time in the kernel
  - E.g. On a uniprocessor, the Exim mail server spends 70% in kernel
- These applications should scale with more cores
- If OS kernel doesn't scale, apps won't scale

- An example of an interesting parallel program

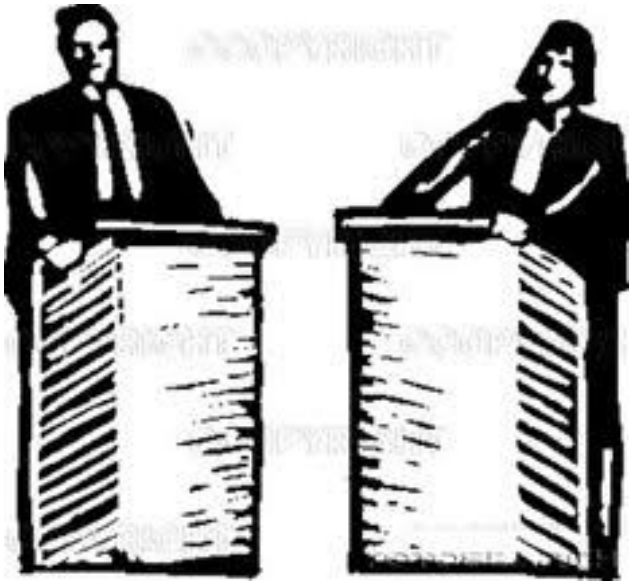
# Evolution of kernel scalability so far

- 20 years ago: uniprocessor design
  - No parallelism, no need for locking etc.
- 20 years of continuous redesign (e.g., Linux):
  - Fine-grained locking
  - Partition & replication of shared data structures
    - E.g., per-core free lists and locks
  - Lock-free reads of certain data structures
    - E.g., directory-entry cache
- Uses many recent parallel computing ideas

# Speculation about future scalability

- POSIX is just the wrong interface
  - Shared data fundamentally limits scalability
  - Kernels are too big/complex to fix
  - Designs cannot keep up with core count
- New OS kernel designs will be needed:
  - Corey, Barrelfish, fos, Tessellation, …

# Revolution or evolution?



- In 5 years time do we need an new OS interface?
  - Better start planning now ...
- What do we want in an OS interface?

# Goal of an OS interface

- Make application developer's job easy
  - Allow sharing
    - File system, buffer cache, load balancing, etc.
  - But, perfectly scalable
  - Efficient implementation

# Candidate Interfaces

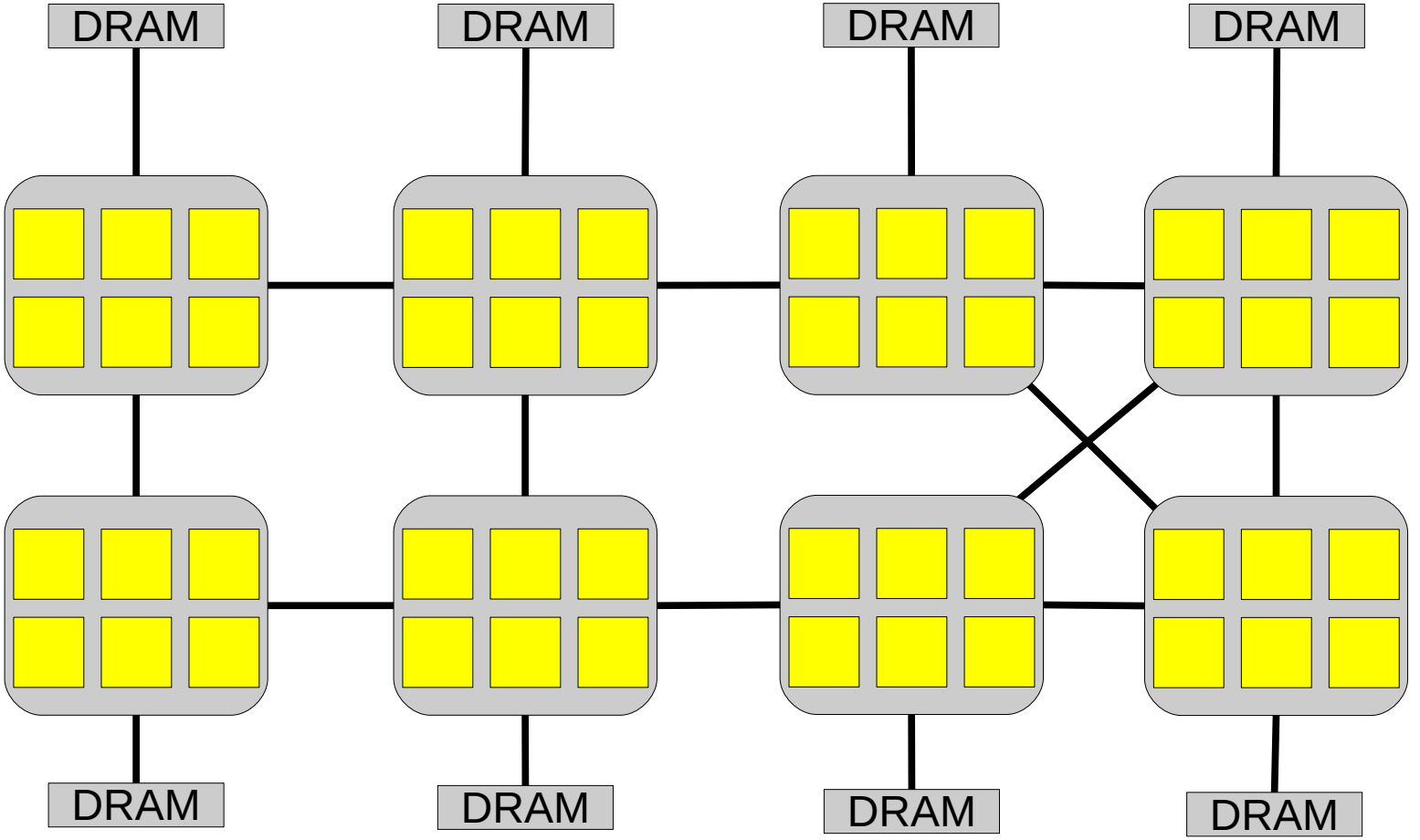
- POSIX/Linux
  - Allows sharing
  - Some calls scale perfectly, but others don't
  - App developer must understand OS implementation
- Corey [OSDI 2008]
  - Expose kernel shared structures and allow applications to control their use
  - Pushes the problem to the app developer
- Barrelfish [SOSP 2009]
  - No shared structure; all sharing through messages
  - Turns the problem into scalable distributed sharing

# 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

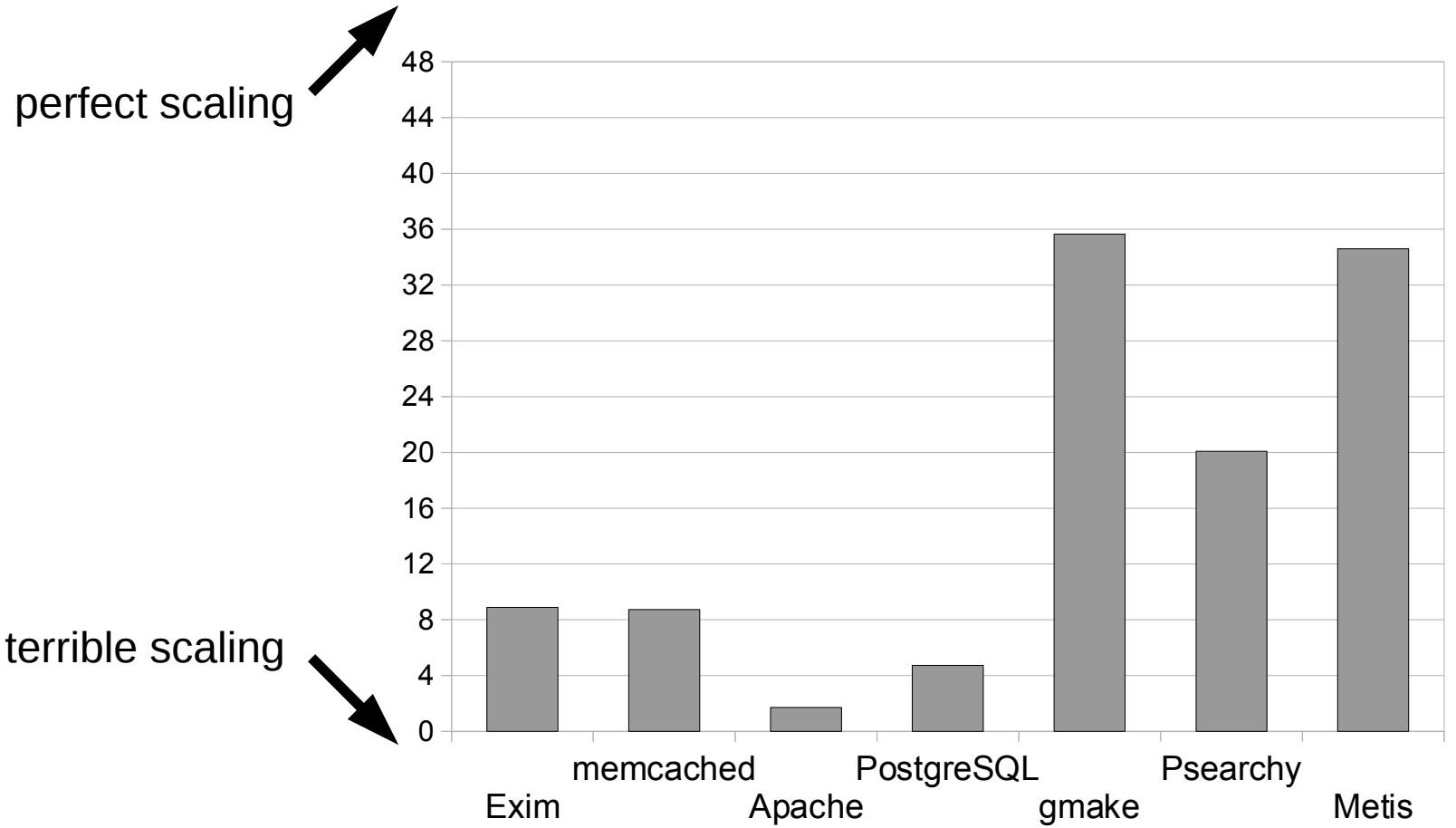
- Insight 1: Short critical sections can lead to sharp performance collapse

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



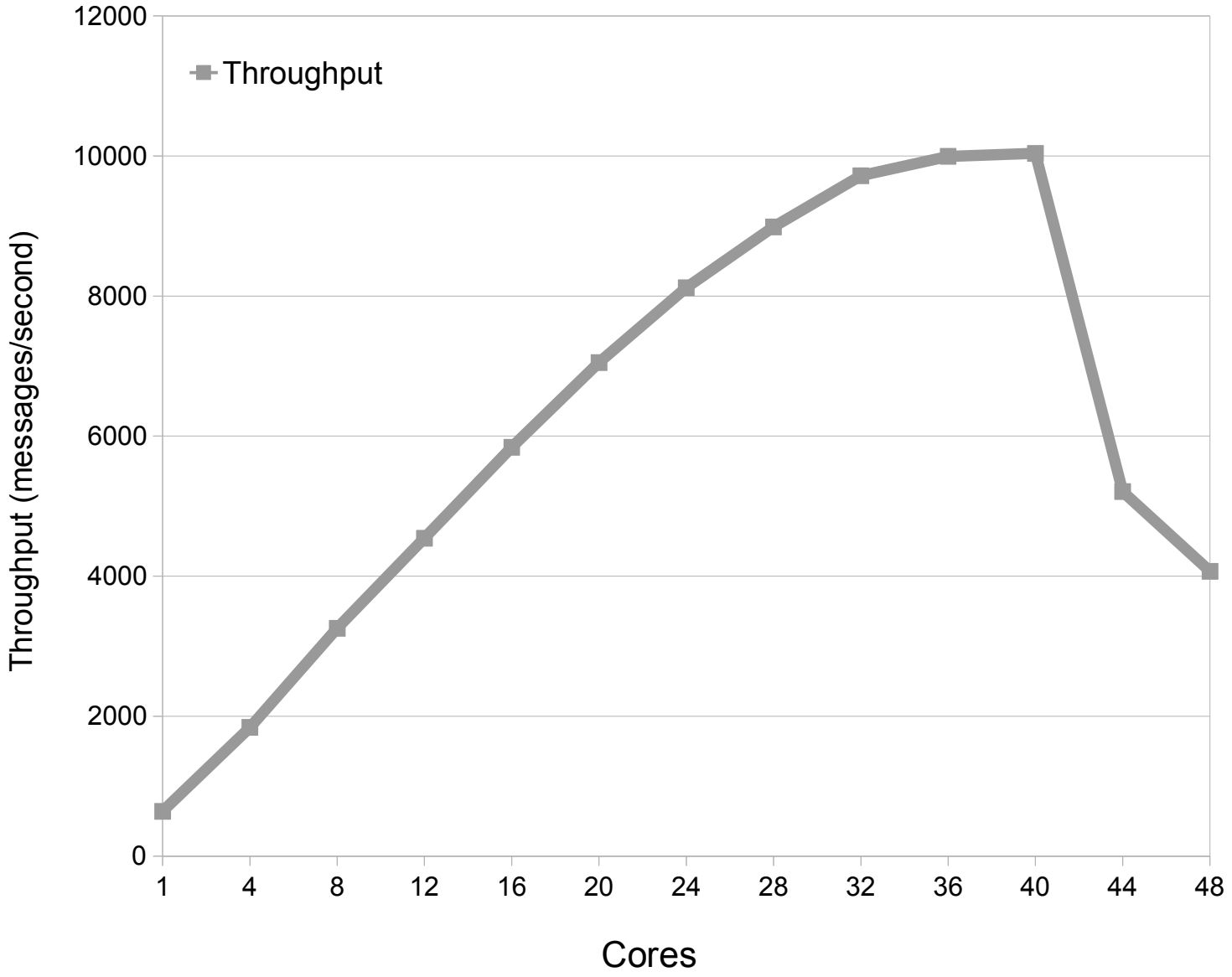

- Cache-coherent and non-uniform access
- An approximation of a future 48-core chip

# Poor scaling on stock Linux kernel

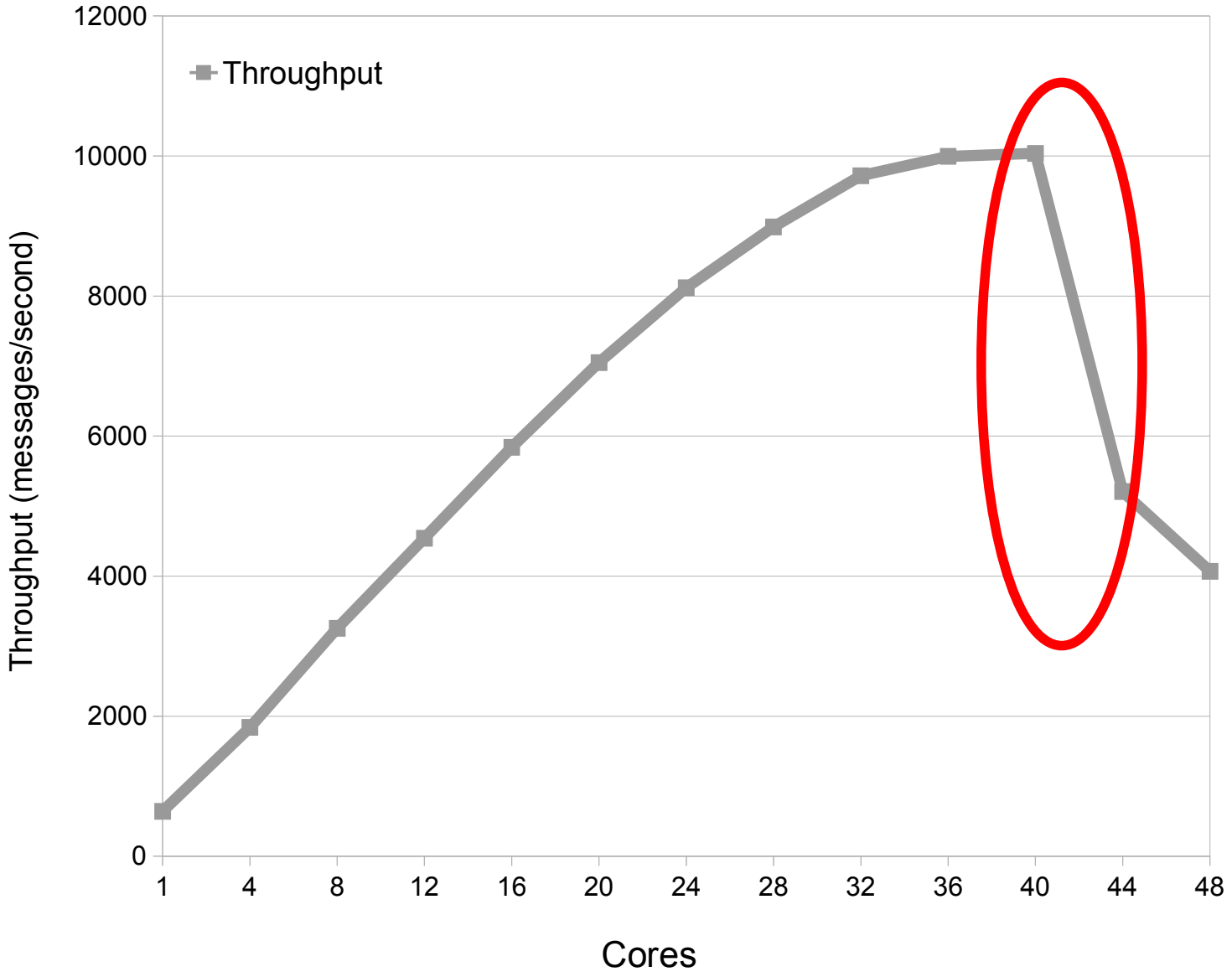



Y-axis: (throughput with 48 cores) / (throughput with one core)

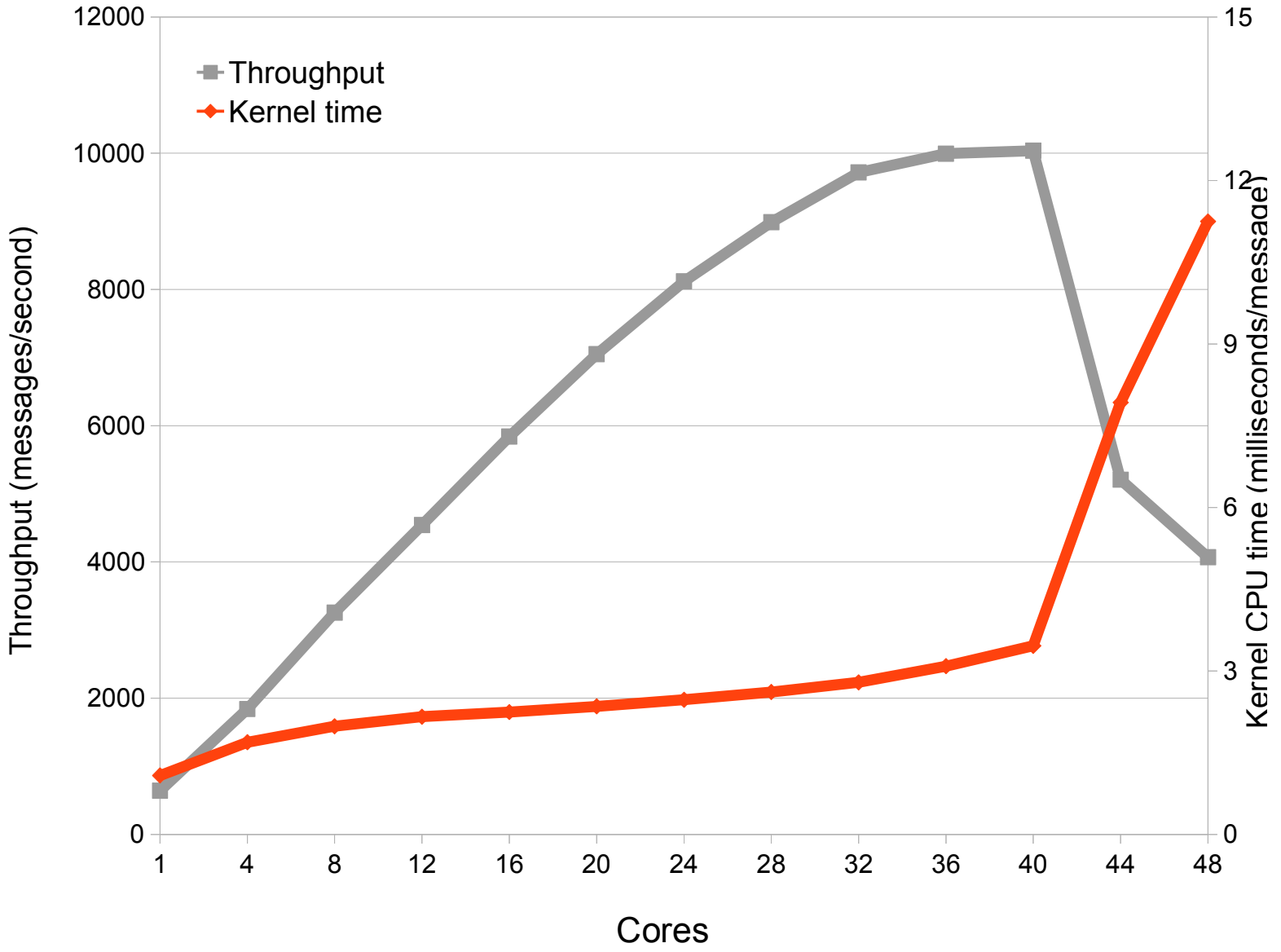
# Exim on stock Linux: collapse



# 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

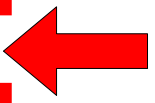
- `sys_open` calls:

```
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?

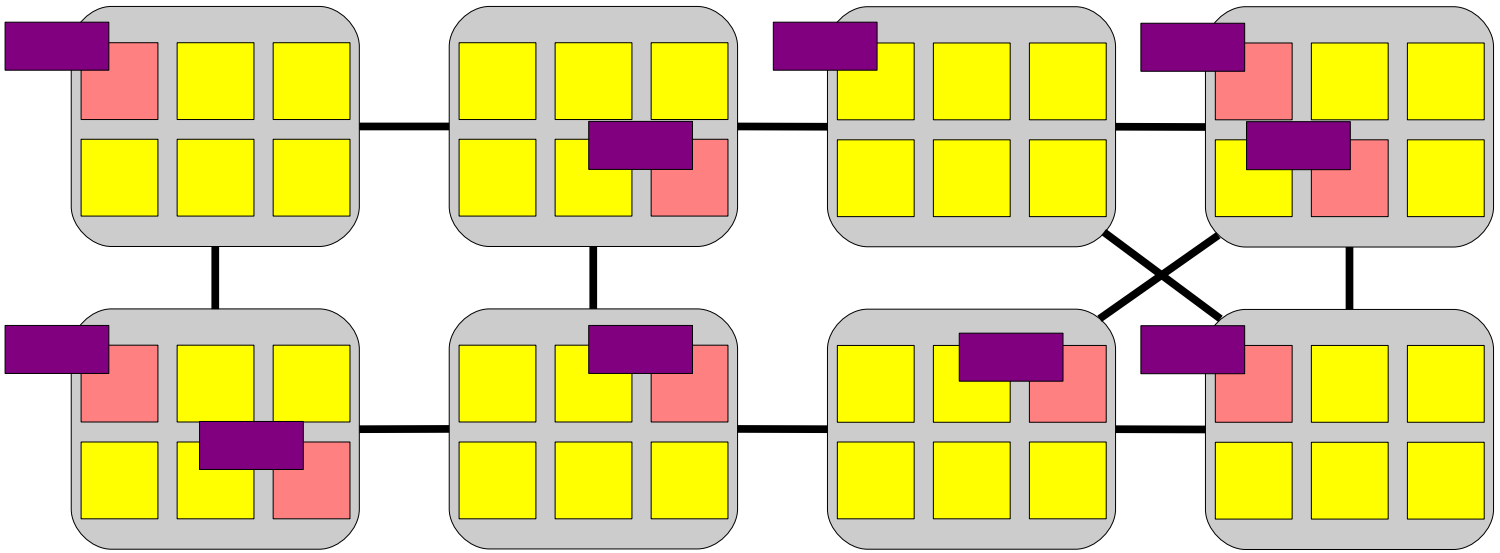
- Linux uses ticket spin locks, which are non-scalable
  - So we should expect collapse [Anderson 90]
- But why so sudden, and so sharp, for a short section?
  - Is spin lock/unlock implemented incorrectly?
  - Is hardware cache-coherence protocol at fault?

# 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;
}
```

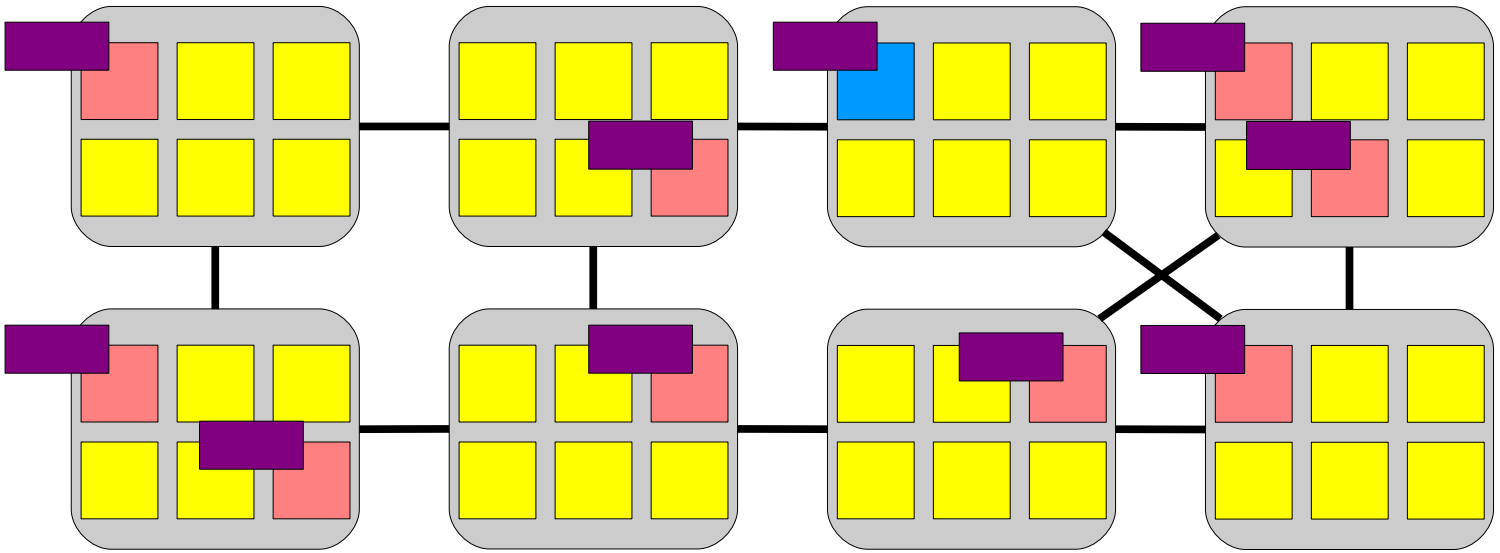


# Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

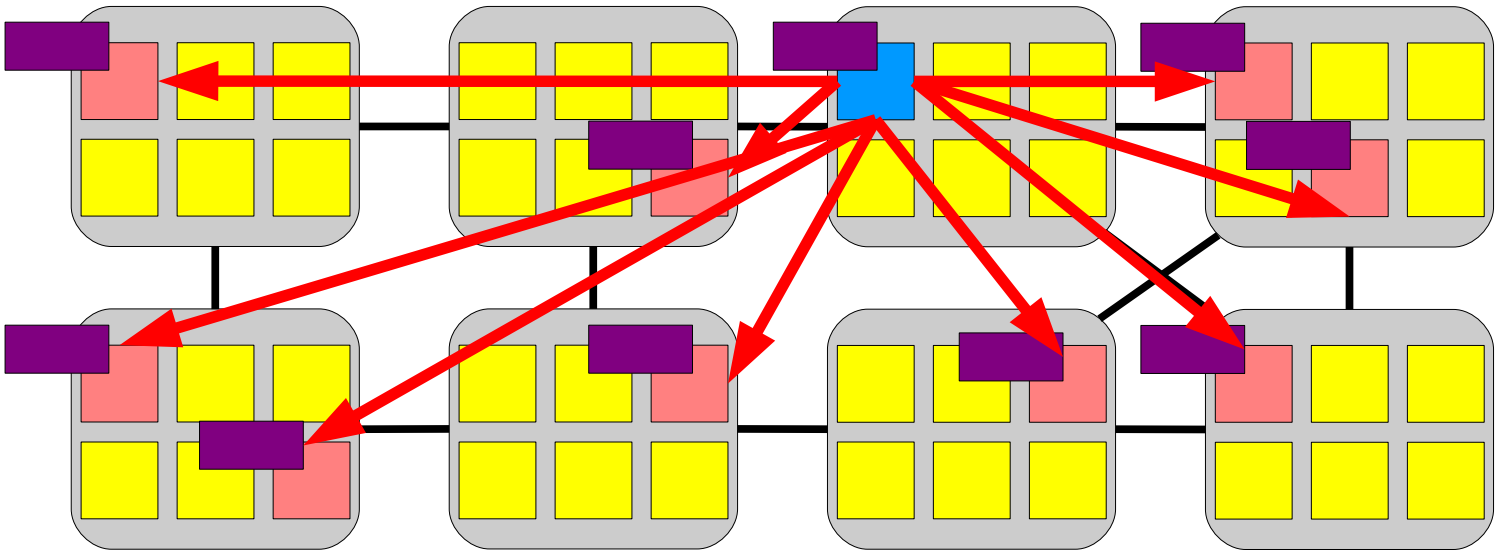


# Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

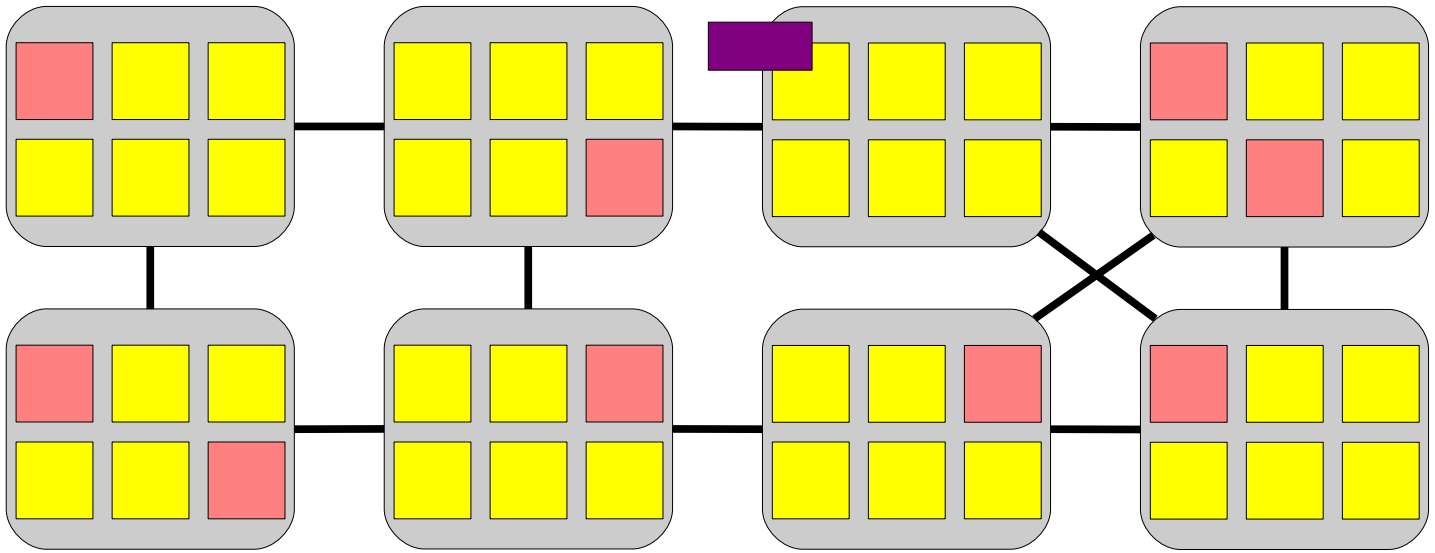


# Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

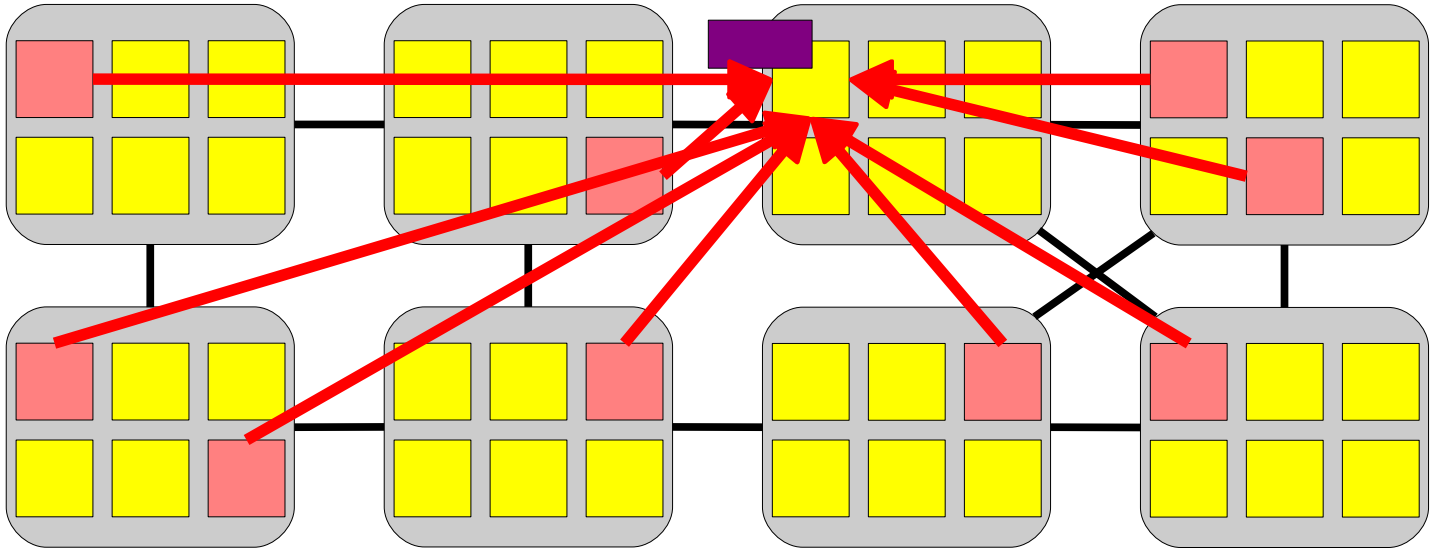


# Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

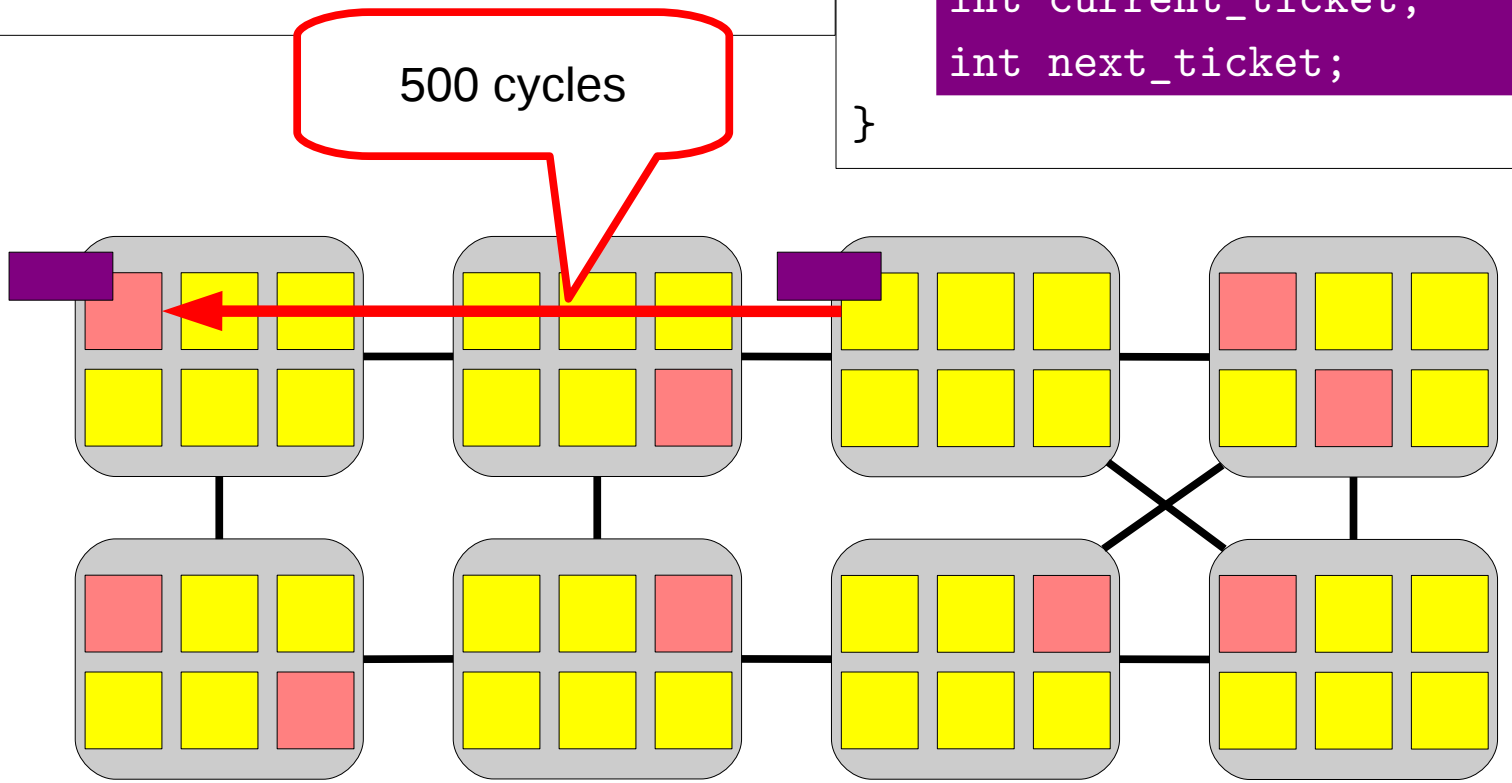


# Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

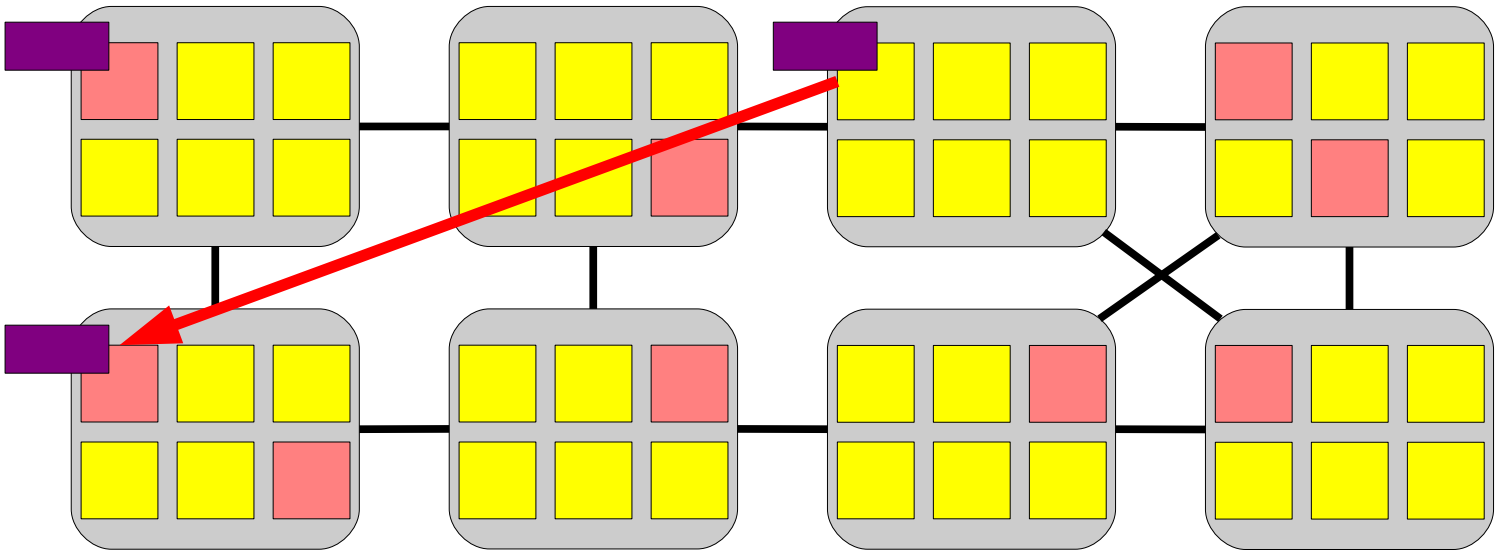


# Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

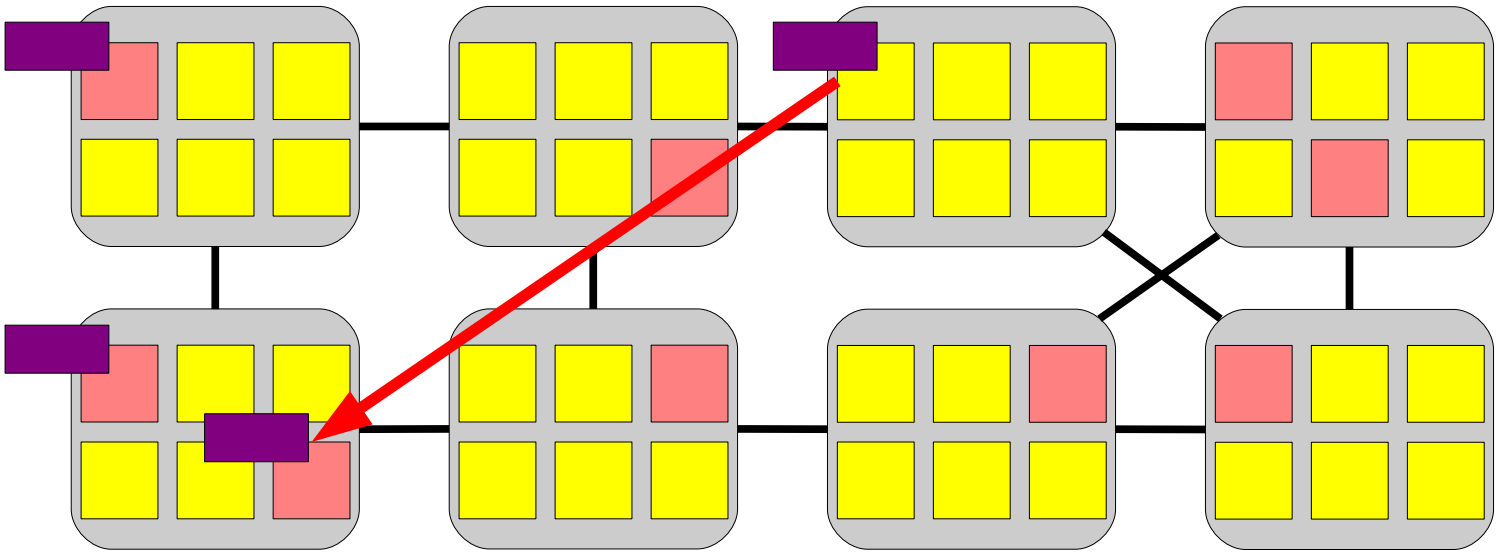


# Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

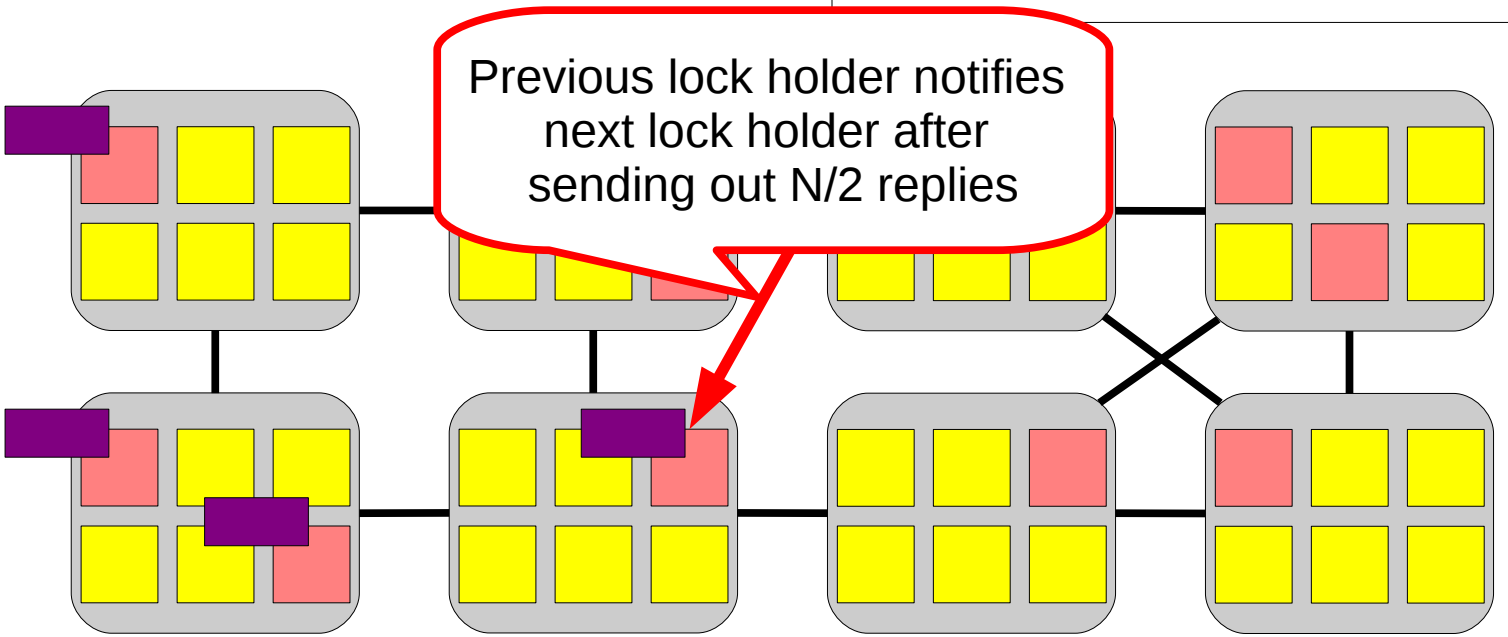


# Scalability collapse caused by non-scalable locks [Anderson 90]

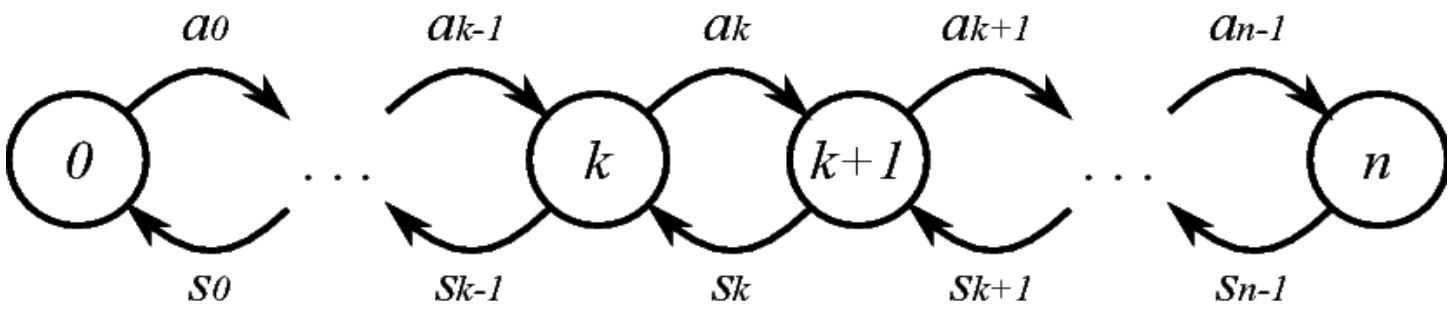
```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

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

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

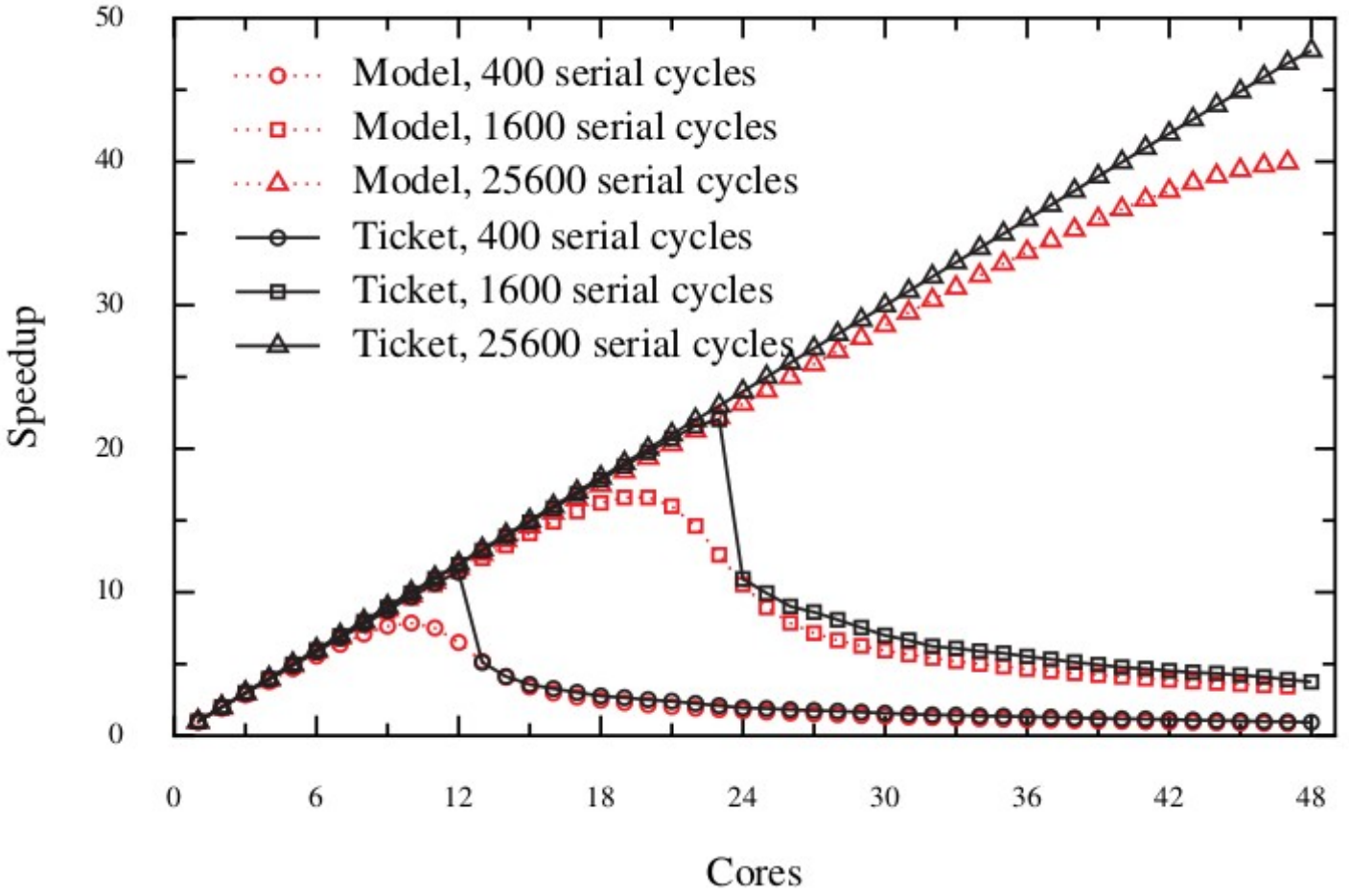


# Why collapse with short sections?




- 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

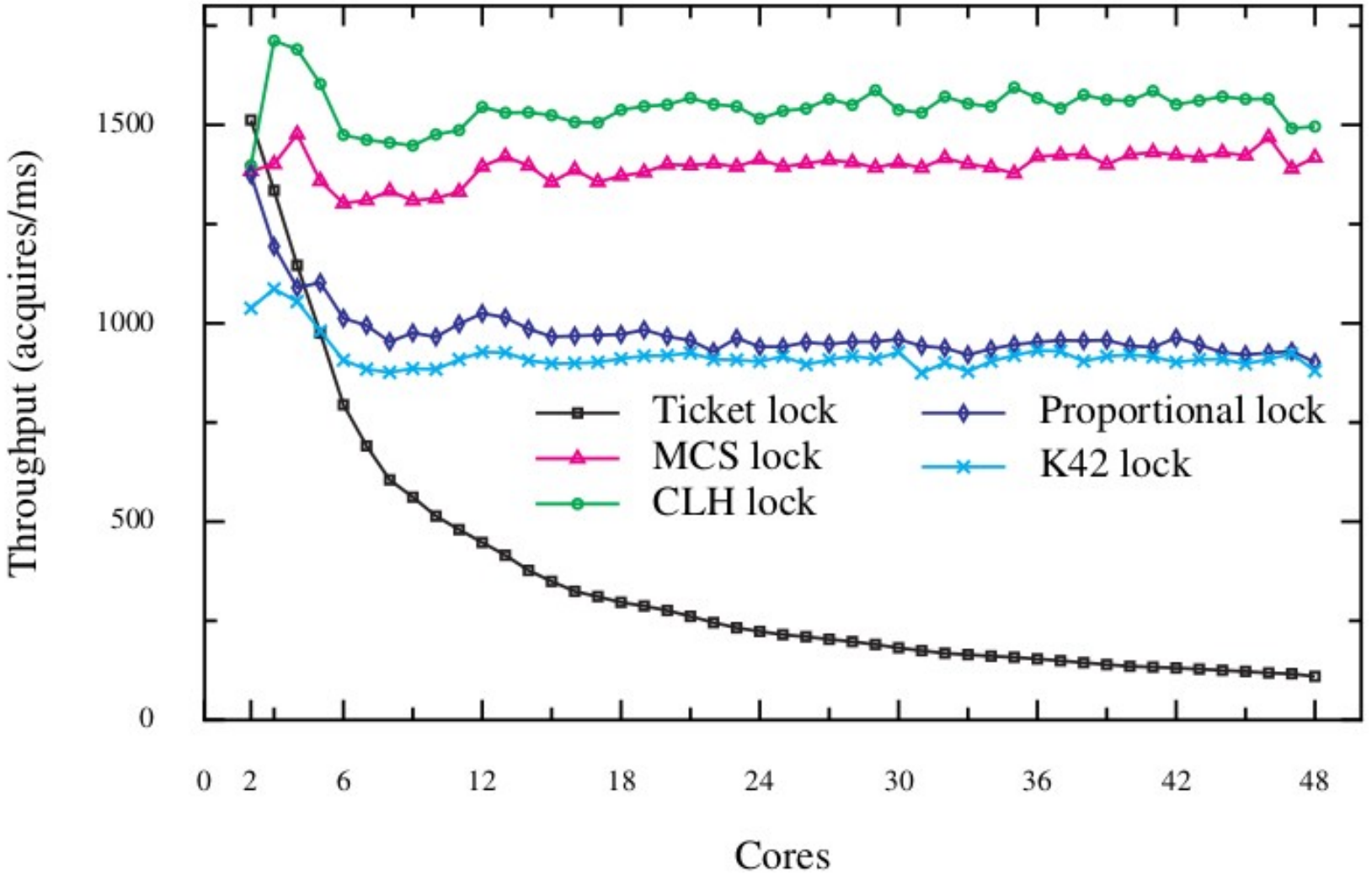



- Experiment: 2% of time spent in critical section
- Critical sections become “longer” with more cores
- Lesson: non-scalable locks fine for long sections

# 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

- How to avoid locking altogether?

# Better: avoid locks in common case

- Observation: mount table is rarely modified

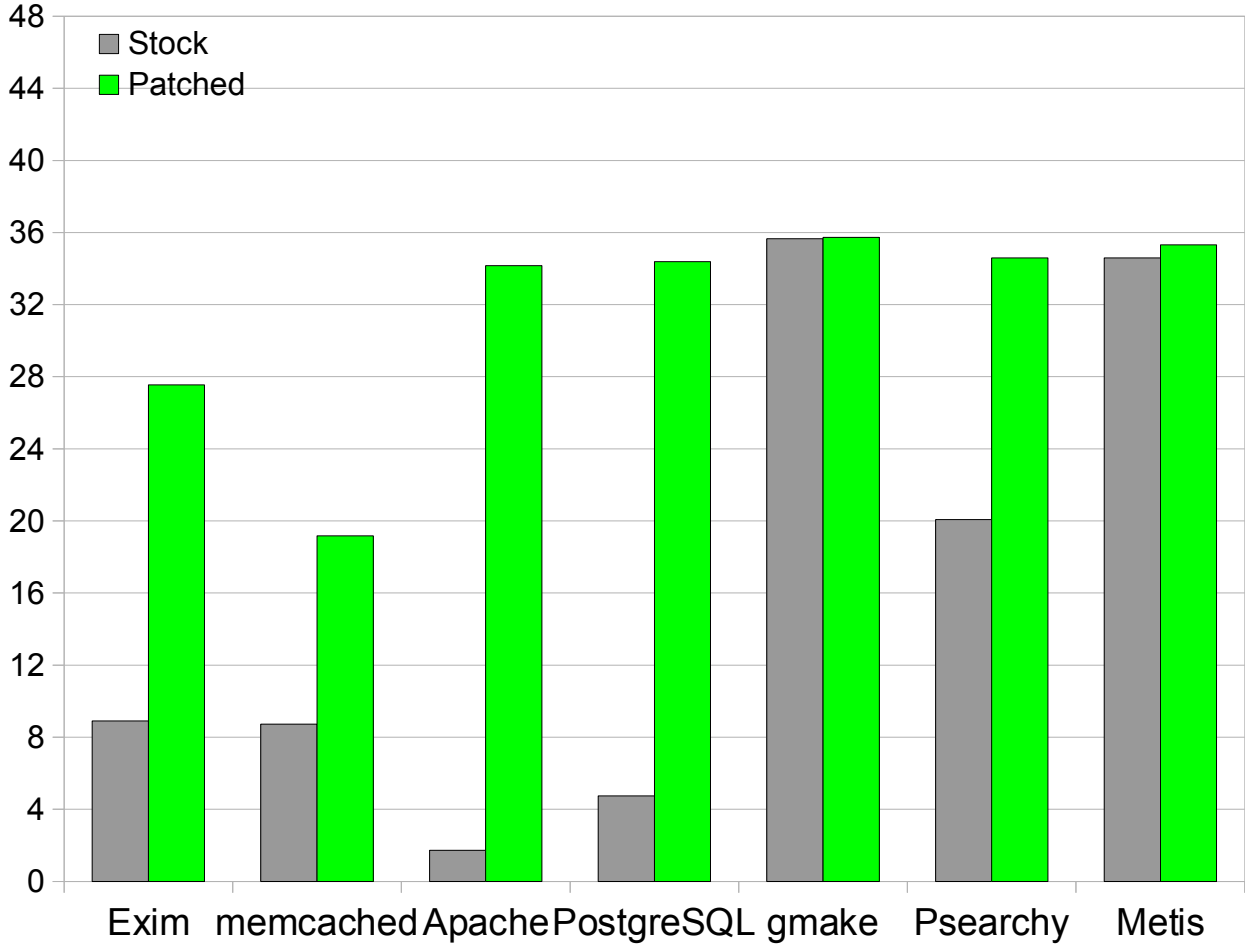
```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        if ((mnt = hash_get(percore_mnts[cpu()], path)))
                return mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        hash_put(percore_mnts[cpu()], path, mnt);
        return mnt;
}
```

# Summary of changes

| | memcached | Apache | Exim | PostgreSQL | gmake | Psearchy | Metis |
|---|---|---|---|---|---|---|---|
| Mount tables | | X | X | | | | |
| Open file table | | X | X | | | | |
| Sloppy counters | X | X | X | | | | |
| inode allocation | X | X | | | | | |
| Lock-free dentry lookup | | X | X | | | | |
| Super pages | | | | | | | X |
| DMA buffer allocation | X | X | | | | | |
| Network stack false sharing | X | X | | X | | | |
| Parallel accept | | X | | | | | |
| Application modifications | | | | X | | X | X |

- 3002 lines of changes to the kernel
- 60 lines of changes to the applications

# Better scaling with our modifications




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

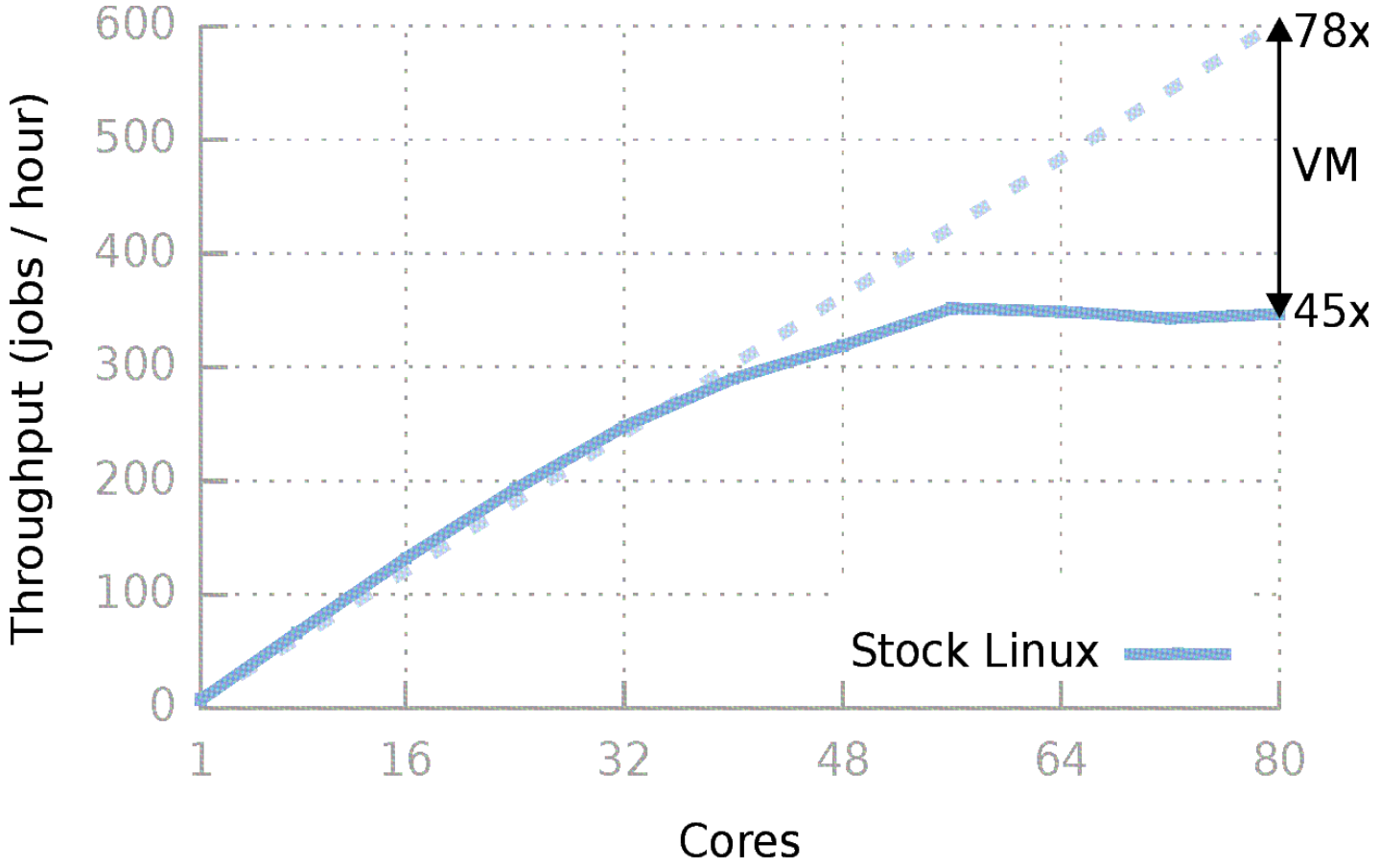
- Most of the scalability is due to the Linux community's efforts

# POSIX scaling is promising but ...

- Some application changes are undesirable
  - Avoiding pthreads in Metis, Psearchy, …
  - Pthread is a common way to write parallel apps

- Insight 2: Avoiding read locks using Bonsai tree

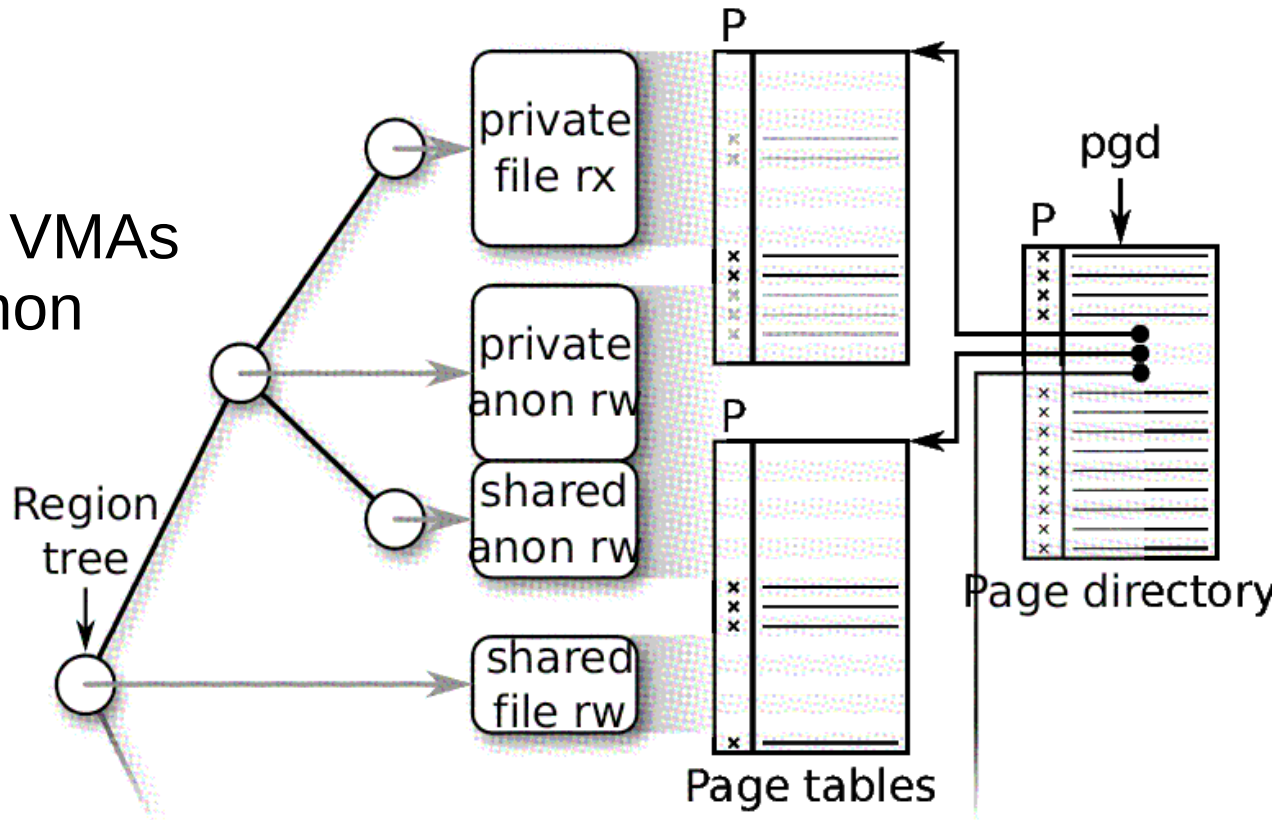
# Pthread example: dedup [parsec]




- Memory allocator library maps/unmaps memory
- Bottleneck by VM operations on address space

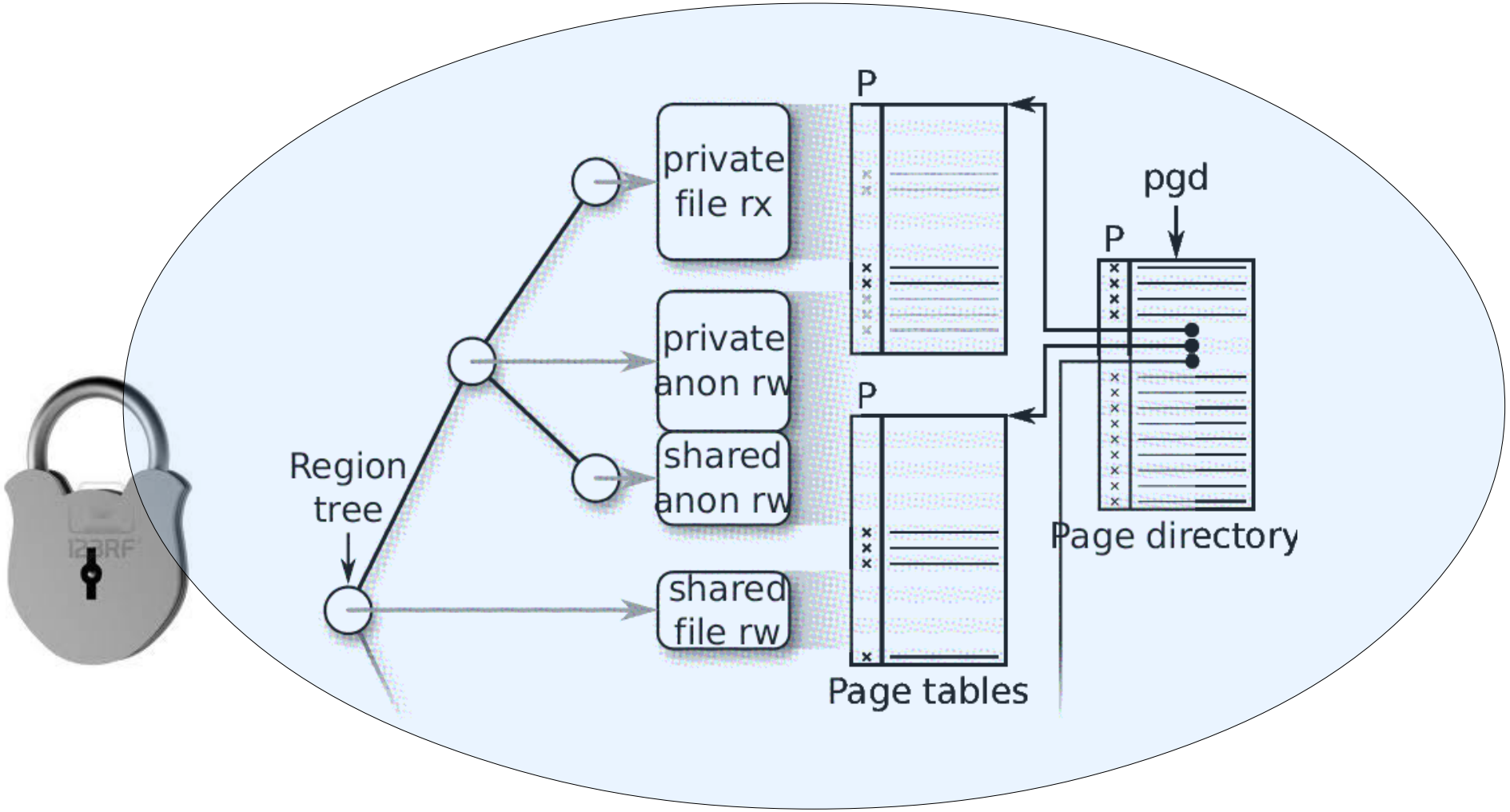
# Address space is complex

1,000s of VMAs are common



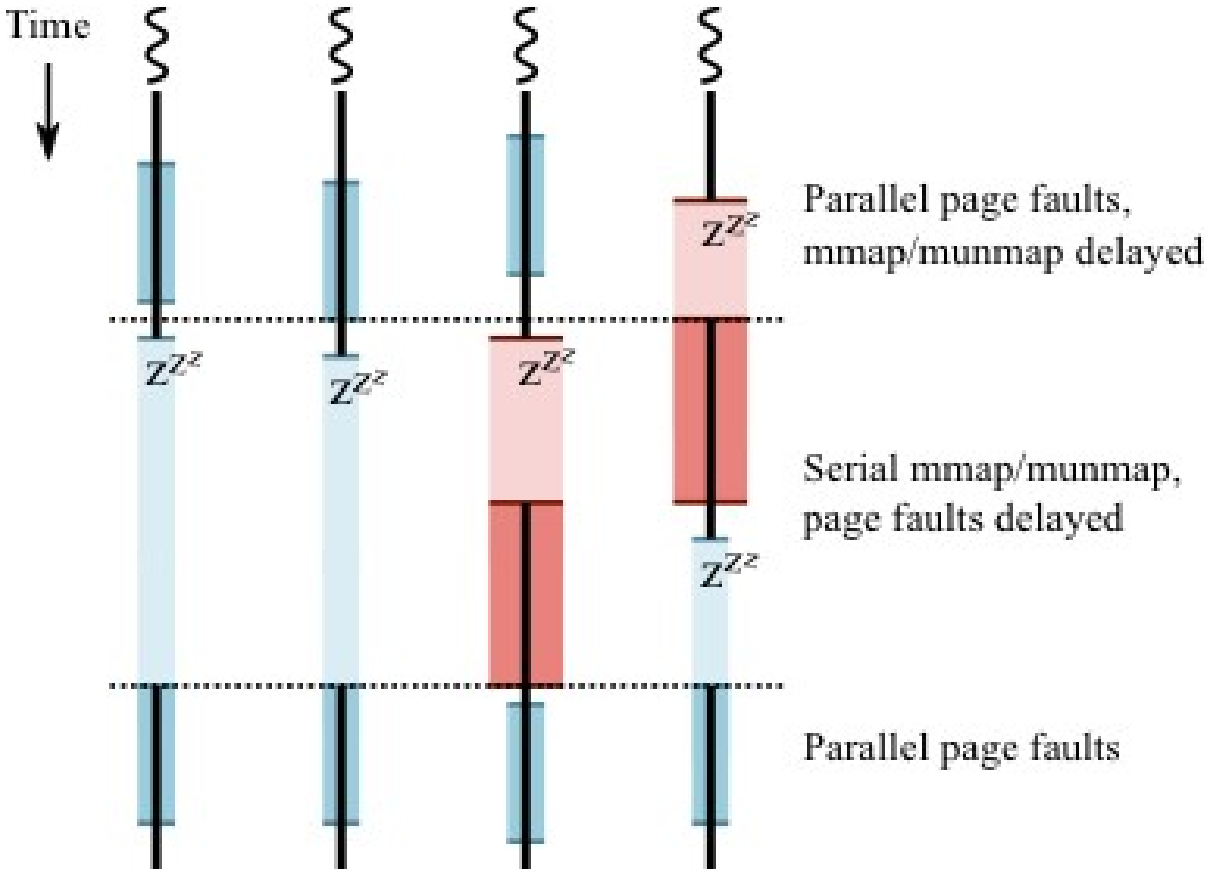

- mmap: create VMA and insert into VMA tree
- pagefault: look up VMA, allocate page, insert in PT
- munmap: remove VMA, clear PT, free pages
  - (Just considering anonymous VMAs for now)

# Address space is complex



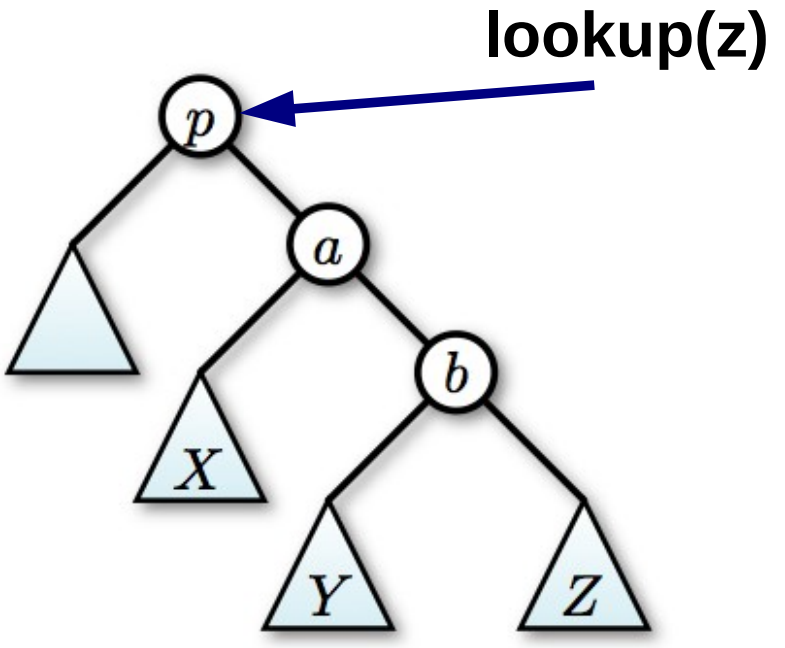

- Lock protects address-space invariants
  - Tree is consistent, PT has no stale entries, ...

# Problem: soft page faults are delayed

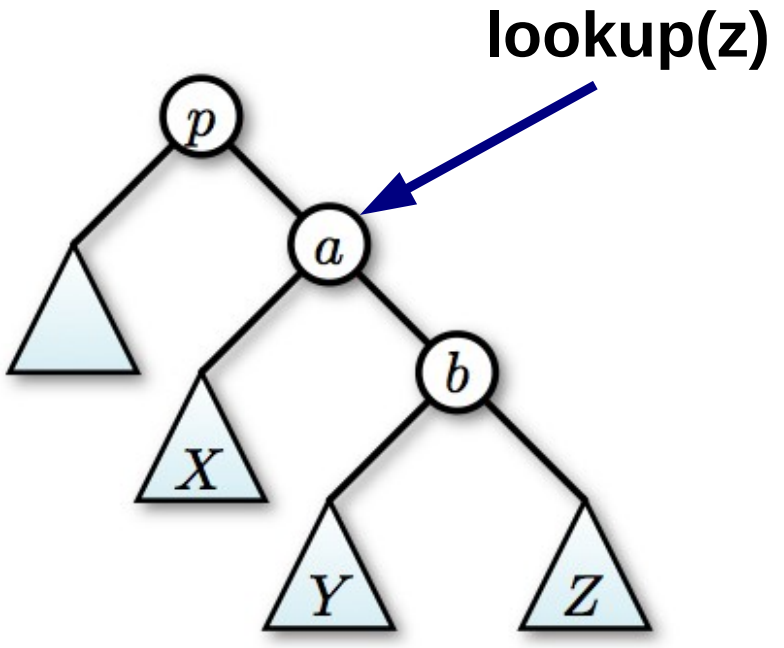



1) No pagefaults can run while mmap/munmap holds lock
2) Pagefaults modify cache line storing lock, which does not scale

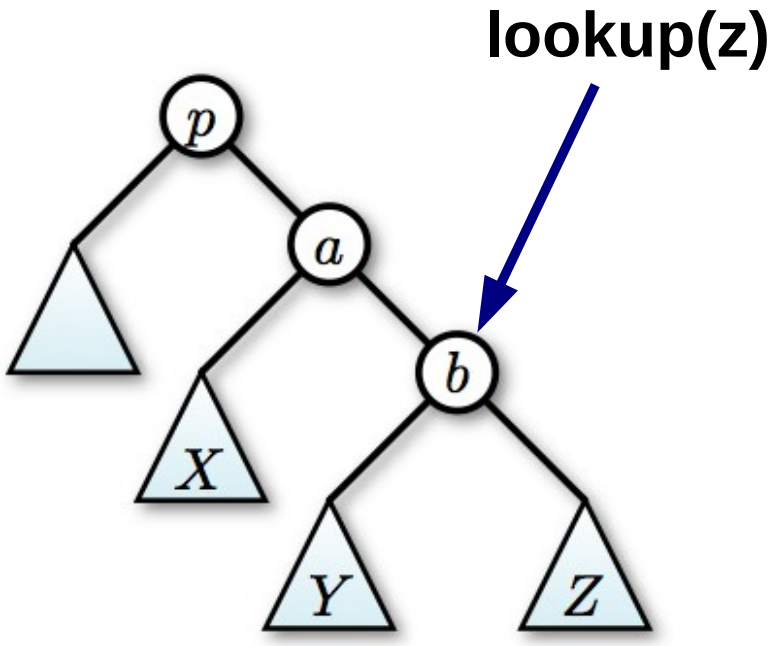
# Why do page faults need to acquire lock?



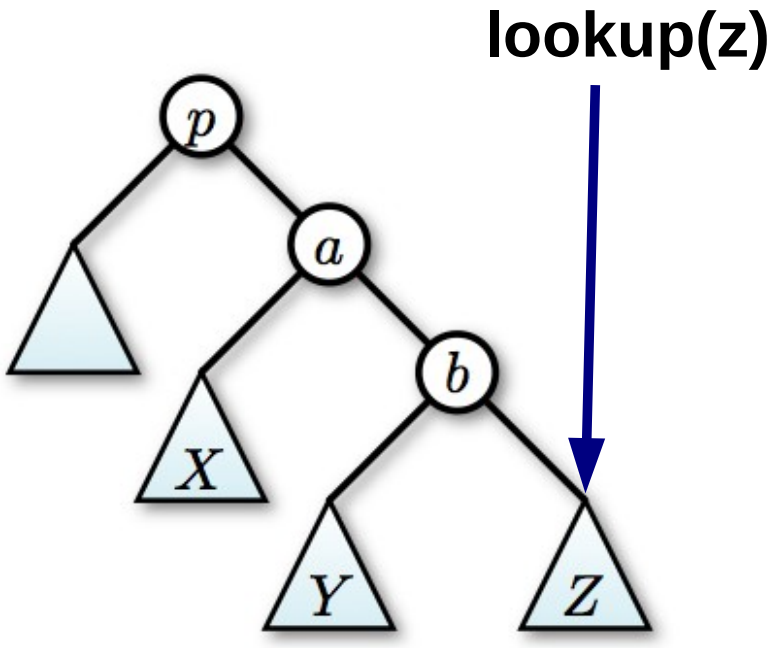
# Why do page faults need to acquire lock?



# Why do page faults need to acquire lock?

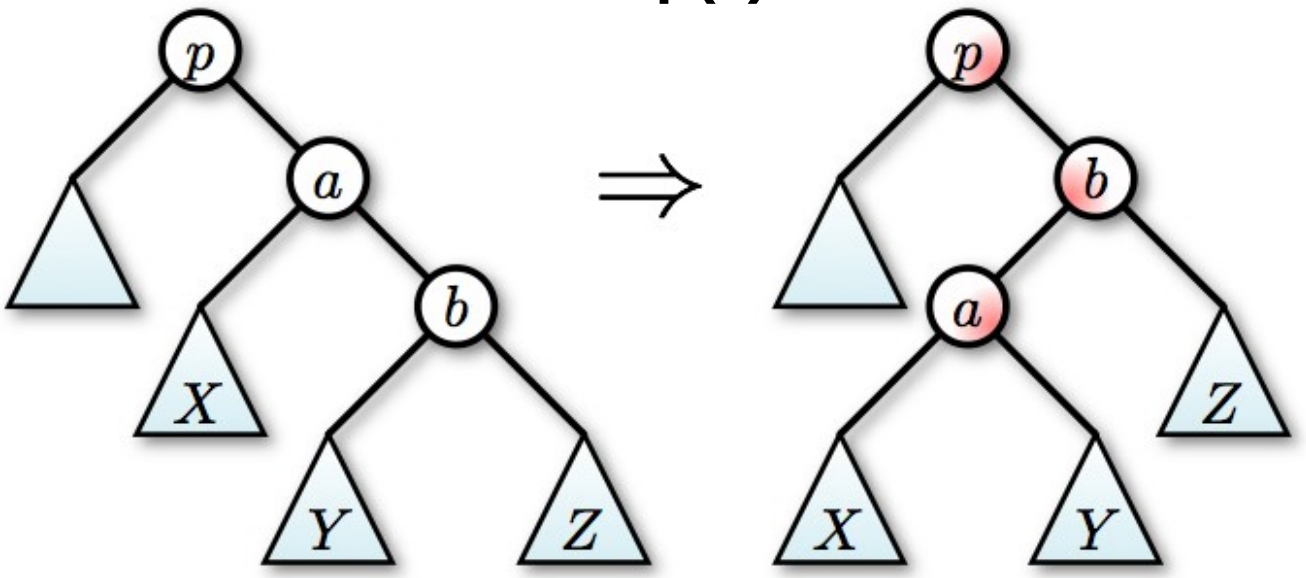


# Why do page faults need to acquire lock?

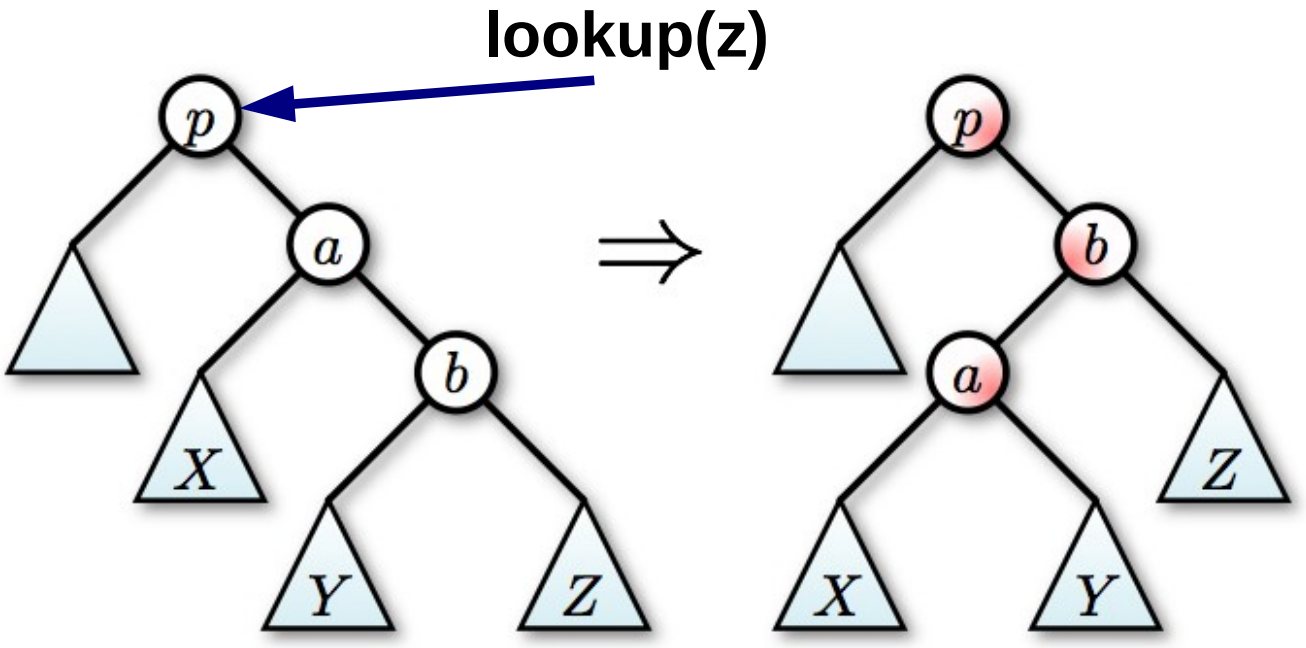


# Problem: concurrent rotates
## (to keep search tree balanced)

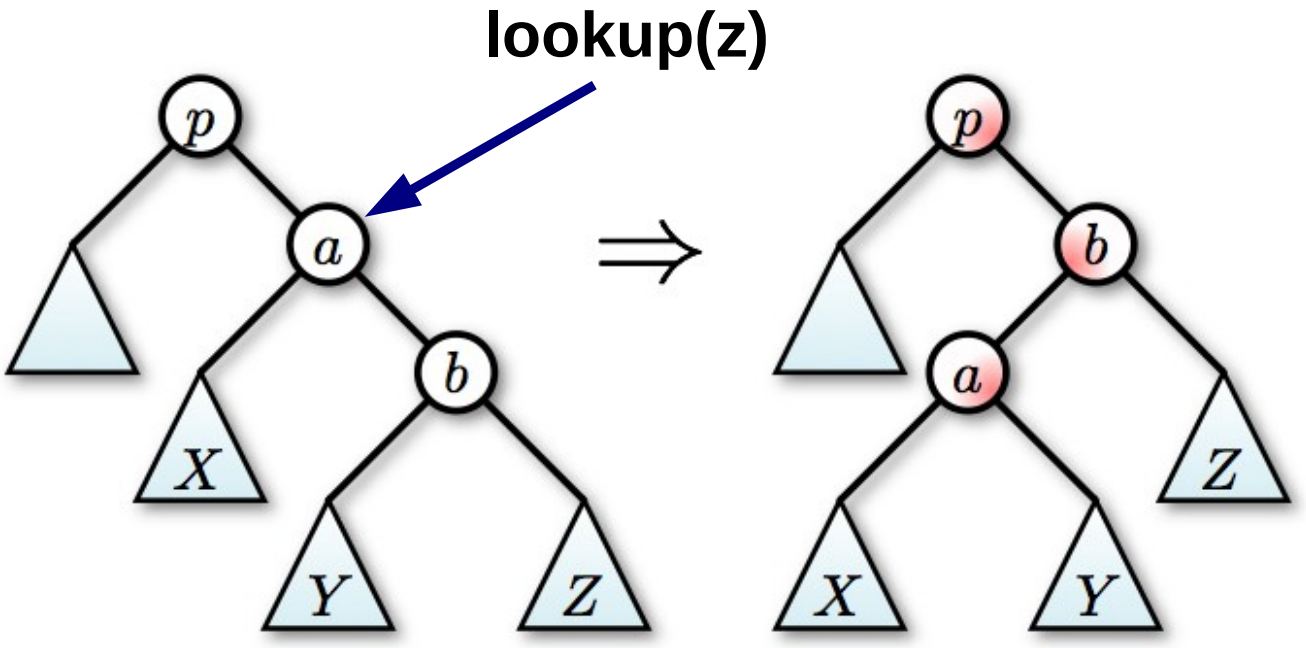
**lookup(z)**



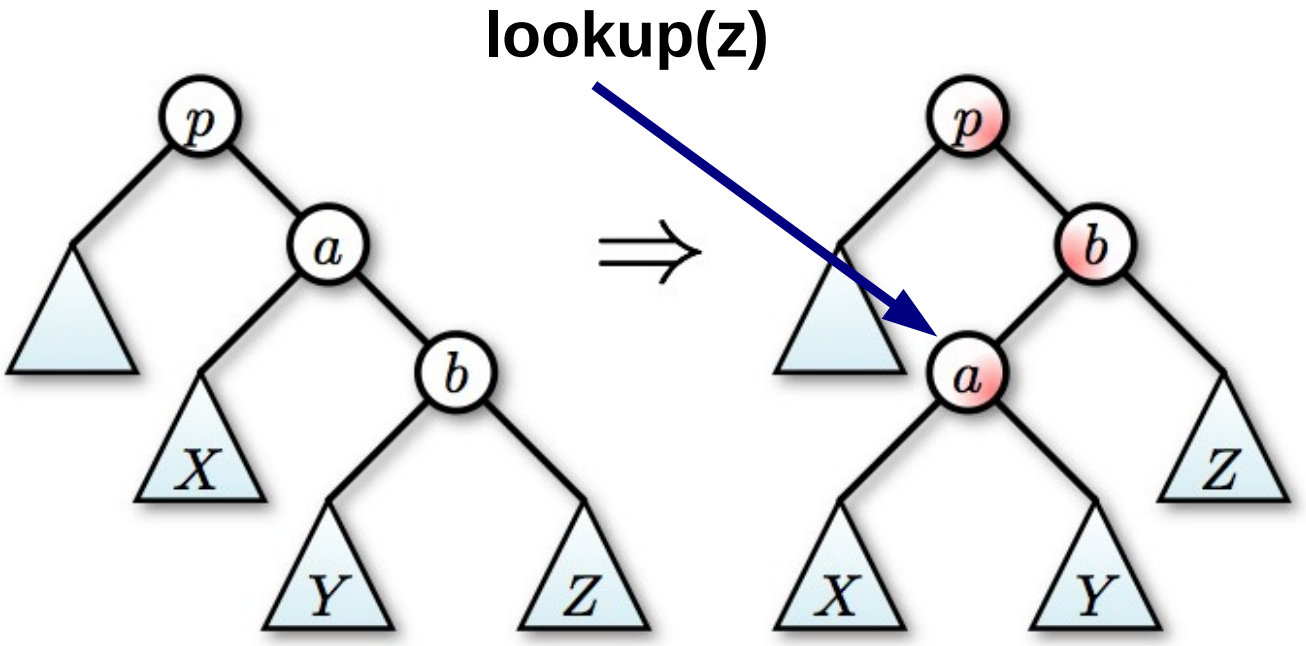
# Without lock:
# race between lookup and rotate



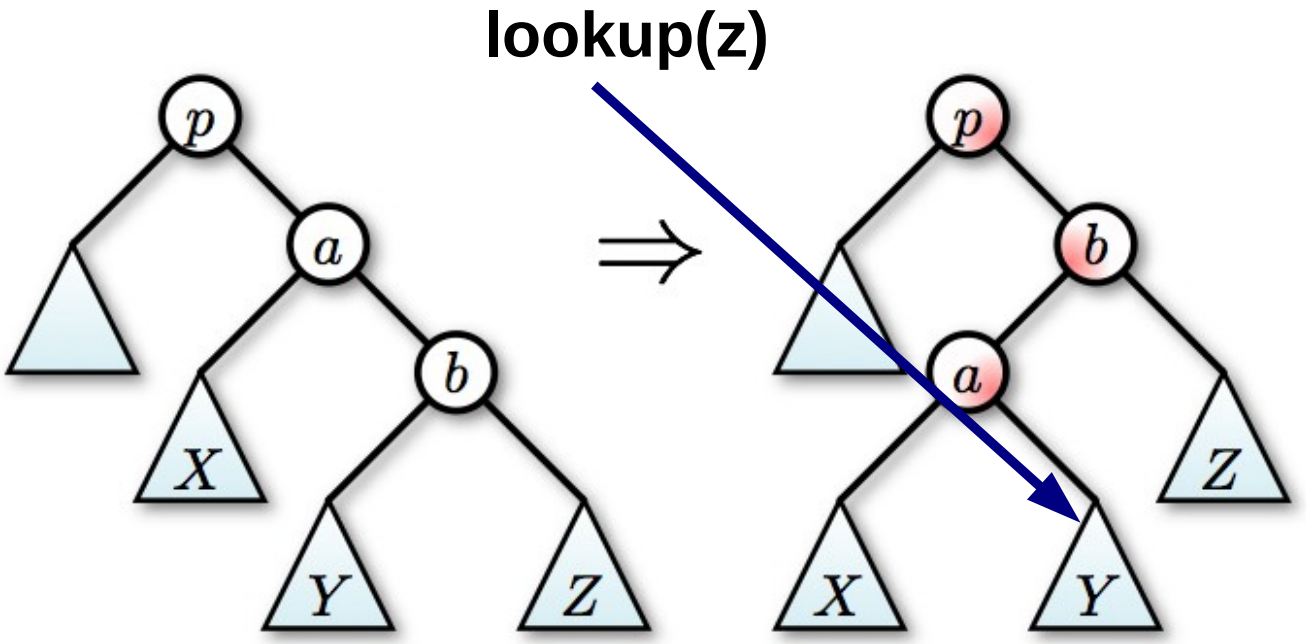
# Without lock:
# race between lookup and rotate



# Without lock:
# race between lookup and rotate



# Without lock: race between lookup and rotate




- Lookup returns the wrong result
- Lock avoids this race, but limits scalability

# Approach: use RCU [McKenney]

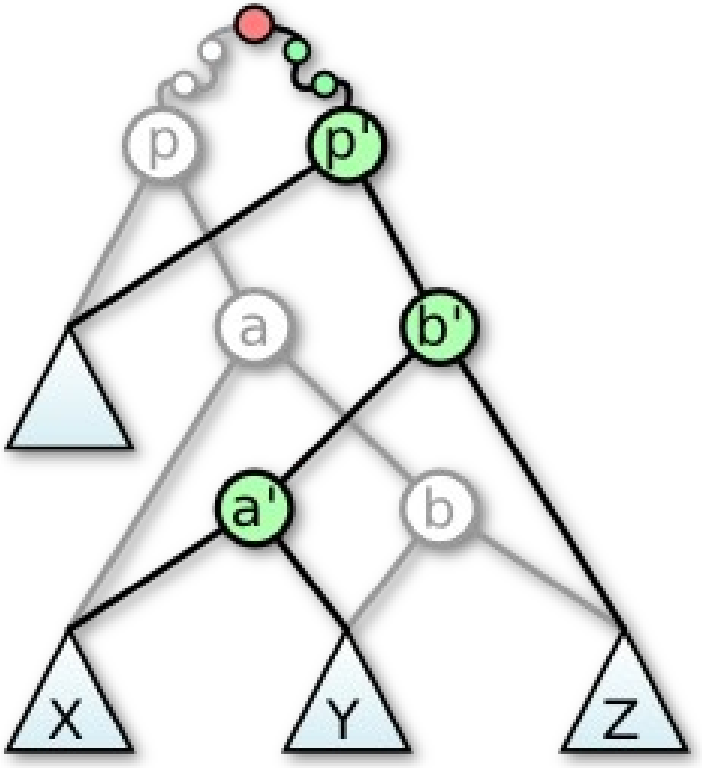
- RCU lets read-only operations run without locks
  - Writers still acquire locks (but not readers)
  - Writes must update a single pointer
  - Delay garbage-collection of memory

# Approach: use RCU [McKenney]

- RCU lets read-only operations run without locks
  - Writers still acquire locks (but not readers)
  - Writes must update a single pointer
  - Delay garbage-collection of memory

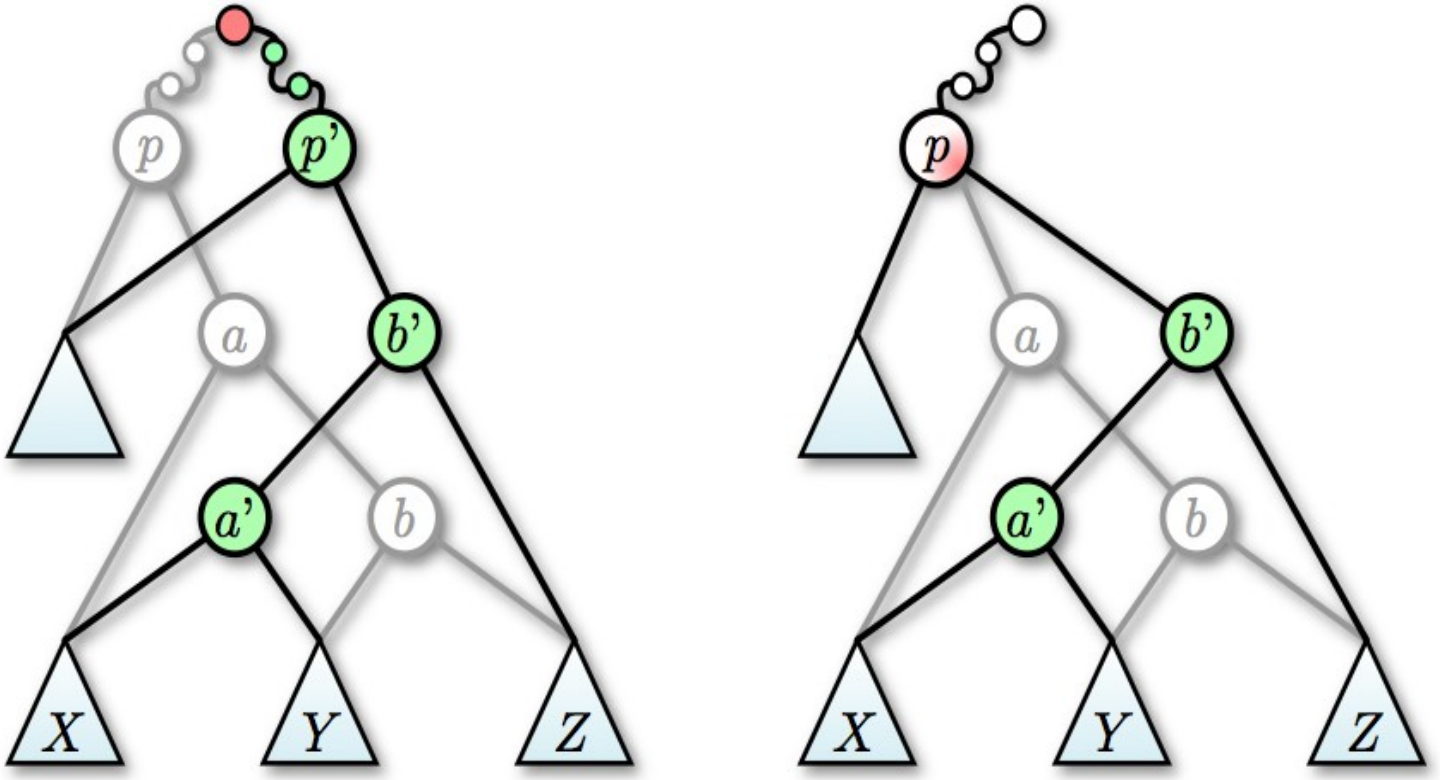
- Applying RCU to the VMA tree is difficult
  - Tree balancing involves multiple pointer updates
  - Page faults are not quite read-only
    [ see ASPLOS paper for more details about this ]

# New concurrent data structure: the Bonsai tree



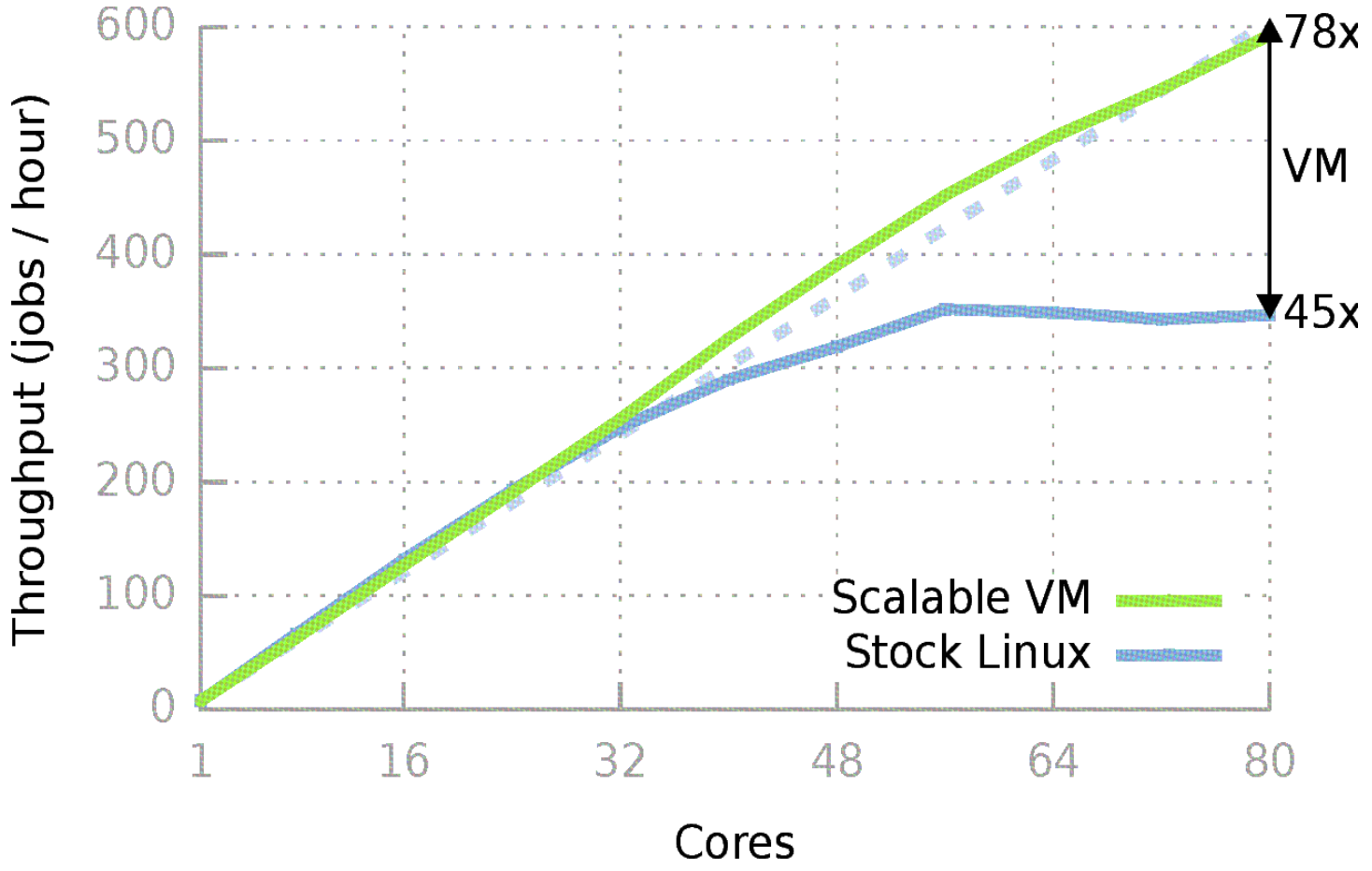

- Functional data structure: don't update pointers
- Page faults can execute without lock
- Rotate creates log(n) new nodes [tree depth]

# Optimized Bonsai tree




- Optimization reduces cost to O(1) nodes
- Other solutions exist [Fraser 2003, Howard 2010, Zijlstra 2010 – speculative pagefaults]: complex!

# dedup scales better with Bonsai




- Including other optimizations in Bonsai style
- Lock-free anon VMA pagefaults scale perfectly

# Avoiding write locks

- With Bonsai, lock acquired by mmap/munmap

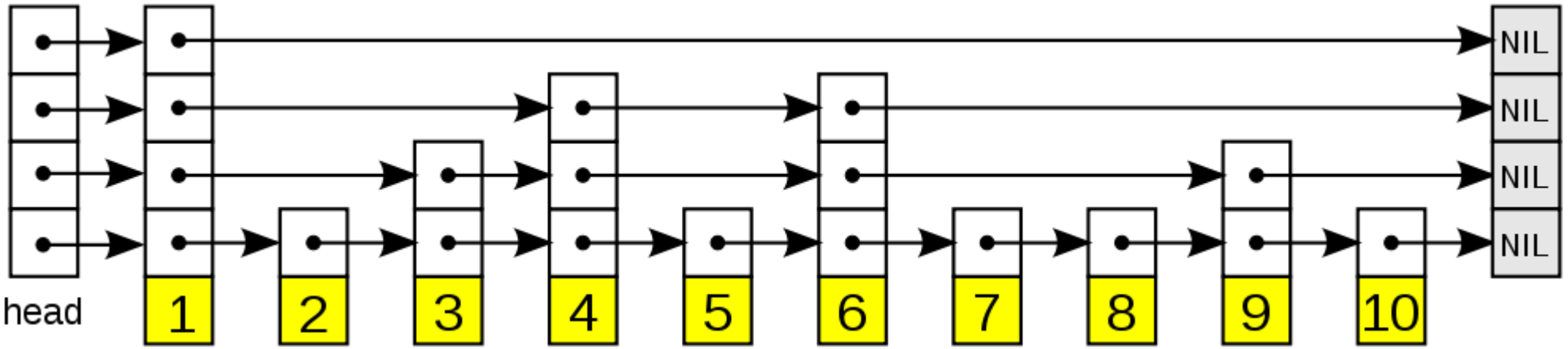
- mmap/munmap for overlapping ranges: probably not
- mmap/munmap for disjoint ranges: in principle, yes

- Can we replace the Bonsai tree?
  - For now,  focus on the VMA data structure

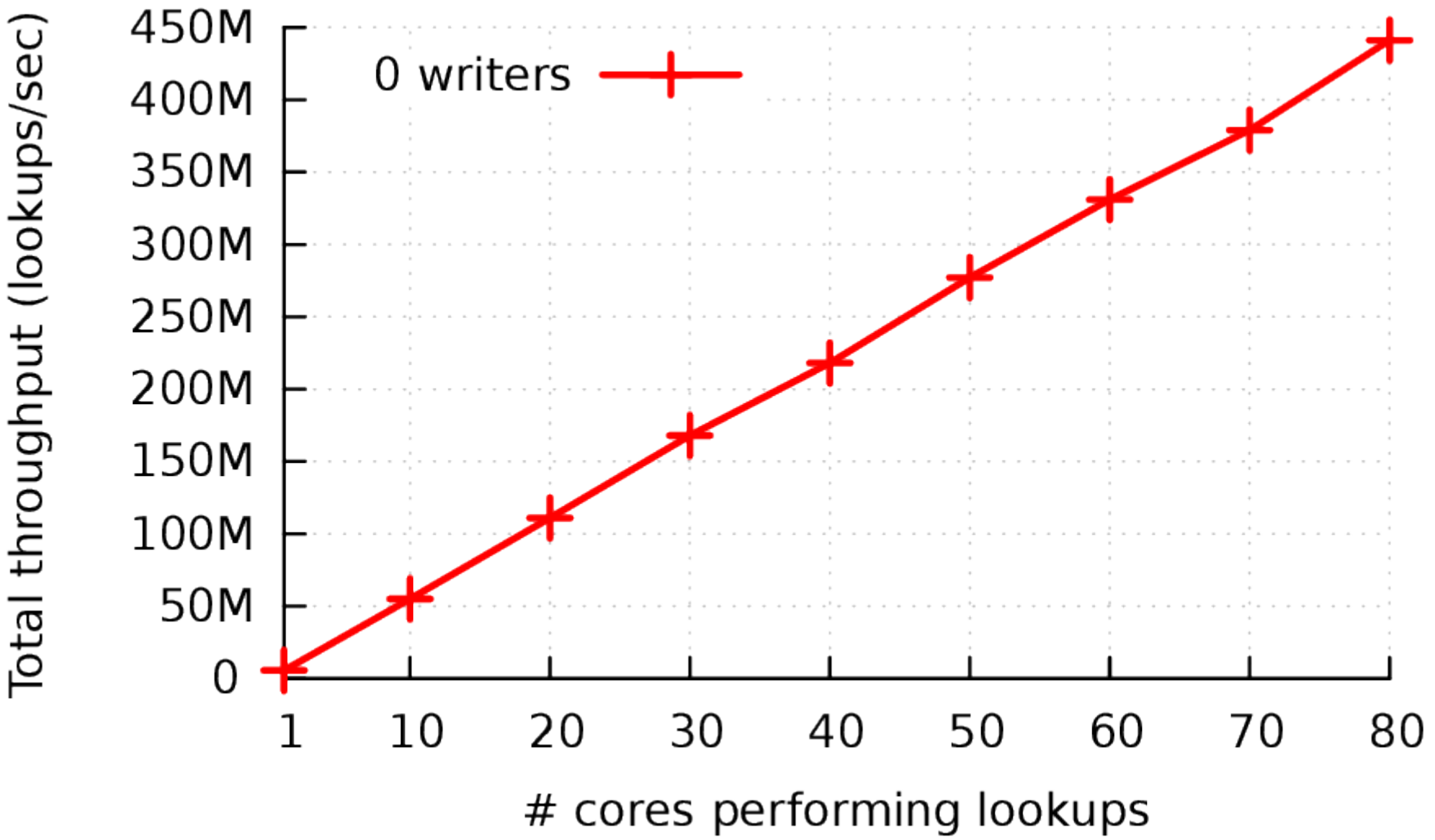
- Insight 3: Avoiding write locks: skip list versus radix tree

# Strawman approach: concurrent skip list

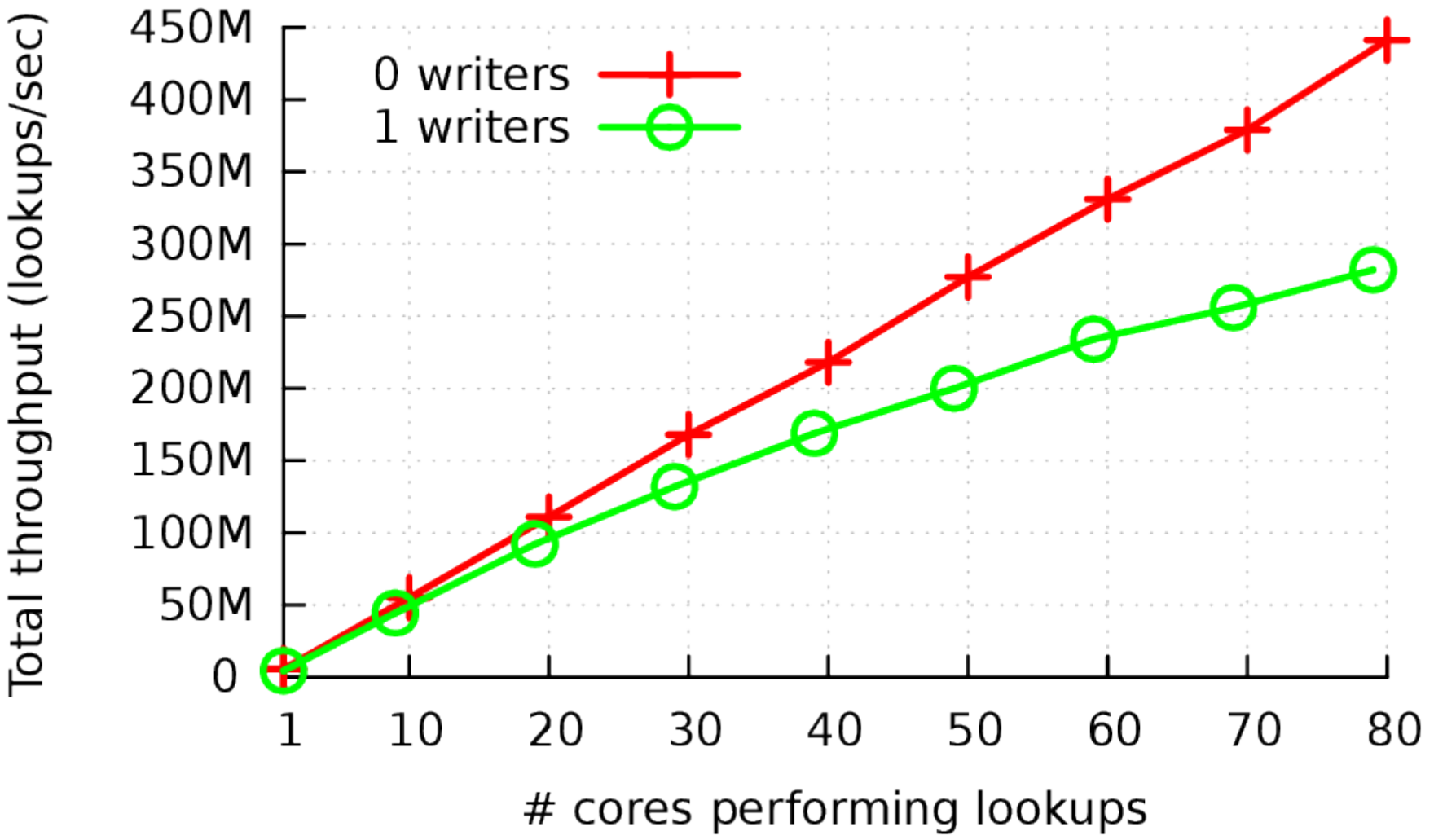
- Suggested by books, researchers, …
- Allows concurrent lookup, insert, remove
- Lock-free data structure: no lock to wait for!



# Concurrent skip list performance

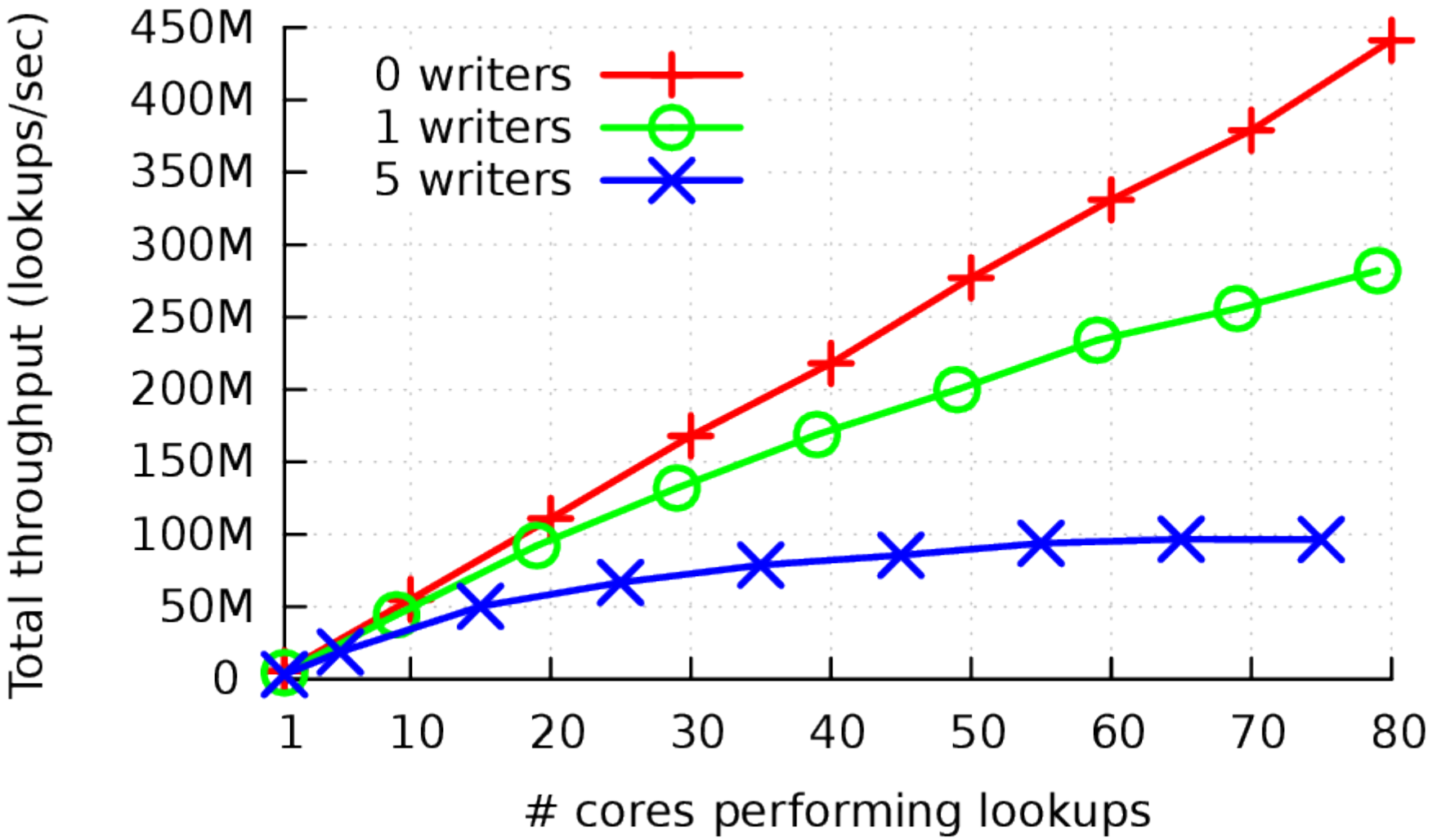


# Concurrent skip list performance




- Writers and lookups access different keys in skip list

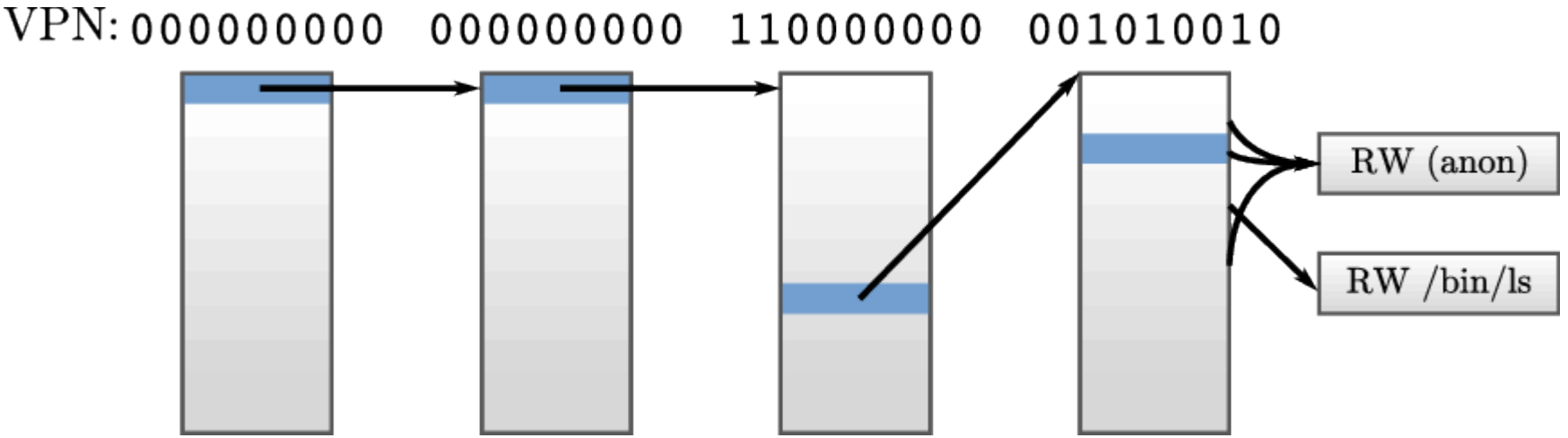
# Concurrent skip list performance




- Same cache lines accessed by independent ops
- Can't scale, even if cores' accesses independent

# Better data structure: “radix tree”

- Effectively same structure as HW page table
  - Fixed number of levels (4 levels for 36-bit virt. page #s)
  - Each level stores 512 pointers to next level, or pointers to VMA (in the last level)



# Radix tree has good scalability

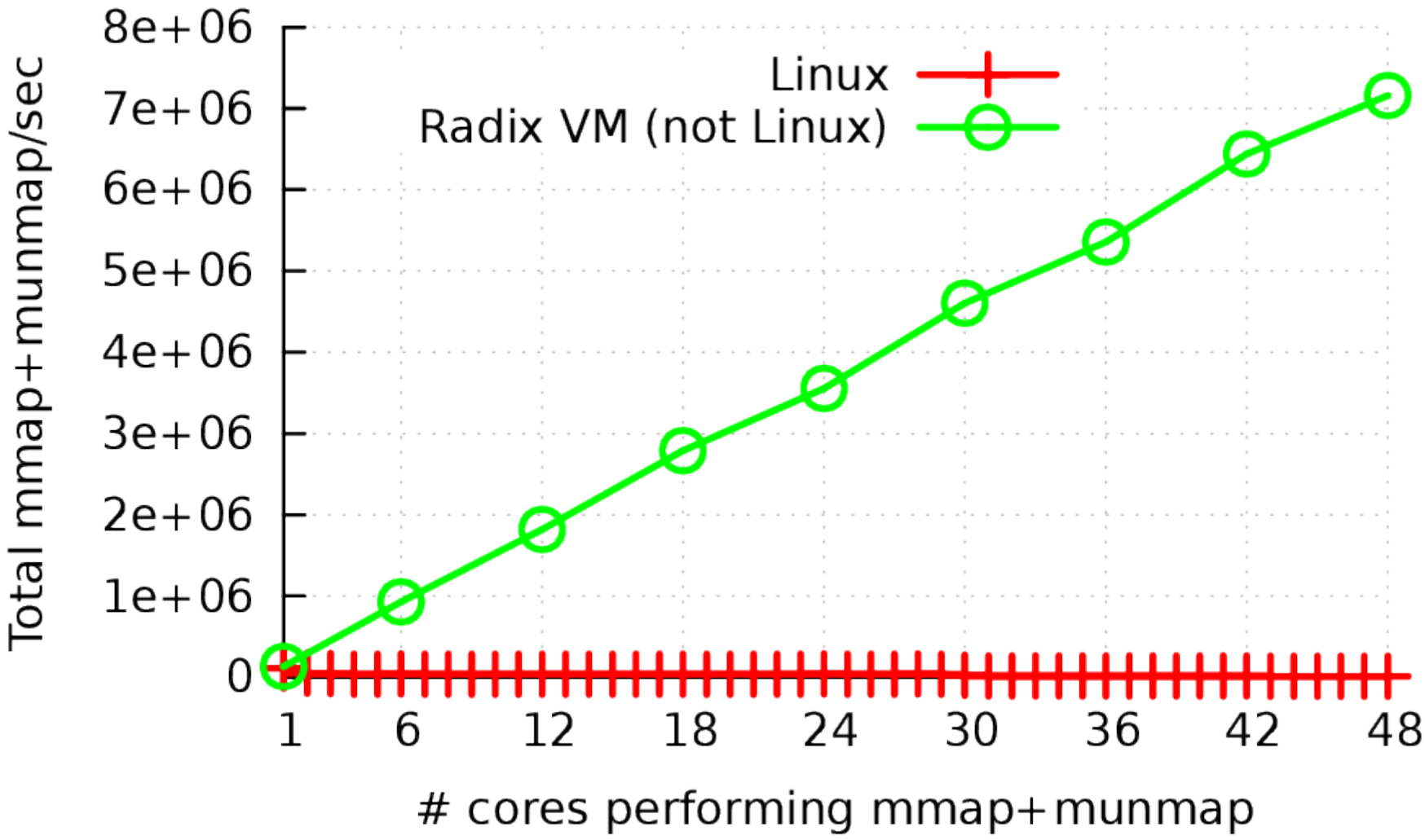
- If two cores perform radix tree operations that access distinct keys

  →

  no physical cache lines need to be transferred between cores

- Radix tree can scale as long as operations are independent

# Radix tree VM performance




- VM system in our OS nowhere near as mature or complex as Linux, but approach seems promising

# Observations POSIX API

- Unix/POSIX API allows for much concurrency
  - State-of-the-art design: many short critical sections
  - Contention-free designs is next big step
- Can all system call invocations scale?
  - No!  E.g., Two mmaps for overlapping addresses ranges
- Can we state a contract between app and OS that states which invocations can scale and which not?
- Insight 4: Scalable commutative rule for interface design

# Scalable Commutativity Rule

- If two invocations commute, then there is an implementation that is perfectly scalable
  - Mmaps for same address range not commutative
  - Mmaps for different address range are commutative
- Implementation maybe difficult to find
  - Skip list isn't good enough, but radix tree is

# Why is SC rule nice?

- Application writers can design their applications to be scalable by using only commutative invocations
- Don't have to run experiments with 2,4, 8, 16, etc. cores to find out if an invocation scales
- Don't have to understand OS internals

- SC rule extends beyond operating systems

# Many research opportunities

- Programming language
- Theory
- Architecture
- .....

# Related work

- Linux and Solaris scalability studies [Yan 09,10] [Veal 07] [Tseng 07] [Jia 08] ...
- Scalable multiprocessor Unix variants (IBM, SGI, Sun, …)
- NUMA research operating systems (e.g., Flash, Disco, K42)
- Lock-free data structures, RCU
- Linux scalability improvements
- Laws of order, disjoint-access parallelism

# Summary

- Insights about scalability:
  - Spin locks cause collapse in short critical sections
  - Avoiding read locks with Bonsai tree
  - Avoiding write locks: skip list versus radix tree
  - Achieving perfect scalability (contention free): scalable commutativity rule for interfaces
- Think in terms of cache line movement
- Optimistic about evolution for scaling OSes