An Analysis of Linux Scalability to Many Cores

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev,
M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich

MIT CSAIL

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, ...)

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

Speculation about kernel scalability

- Several kernel scalability studies indicate existing kernels don't scale well
- Speculation that fixing them is hard
- New OS kernel designs:
  - Corey, Barrelfish, fos, Tessellation, …

- How serious are the scaling problems?
- How hard is it to fix them?
- Hard to answer in general, but we shed some light on the answer by analyzing Linux scalability

Analyzing scalability of Linux

- Use a off-the-shelf 48-core x86 machine
- Run a recent version of Linux
  - Used a lot, competitive baseline scalability
- Scale a set of applications
  - Parallel implementation
  - System intensive

Contributions

- Analysis of Linux scalability for 7 real apps.
  - Stock Linux limits scalability
  - Analysis of bottlenecks
- Fixes: 3002 lines of code, 16 patches
  - Most fixes improve scalability of multiple apps.
  - Remaining bottlenecks in HW or app
  - Result: no kernel problems up to 48 cores

Method

- Run application
  - Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- Fix bottlenecks, re-run application

- Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

Method

- Run application
  - Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- Fix bottlenecks, re-run application

- Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

Method

- Run application
  - Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- Fix bottlenecks, re-run application

- Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

Method

- Run application
  - Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- Fix bottlenecks, re-run application

- Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

Method

- Run application
  - Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- Fix bottlenecks, re-run application

- Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

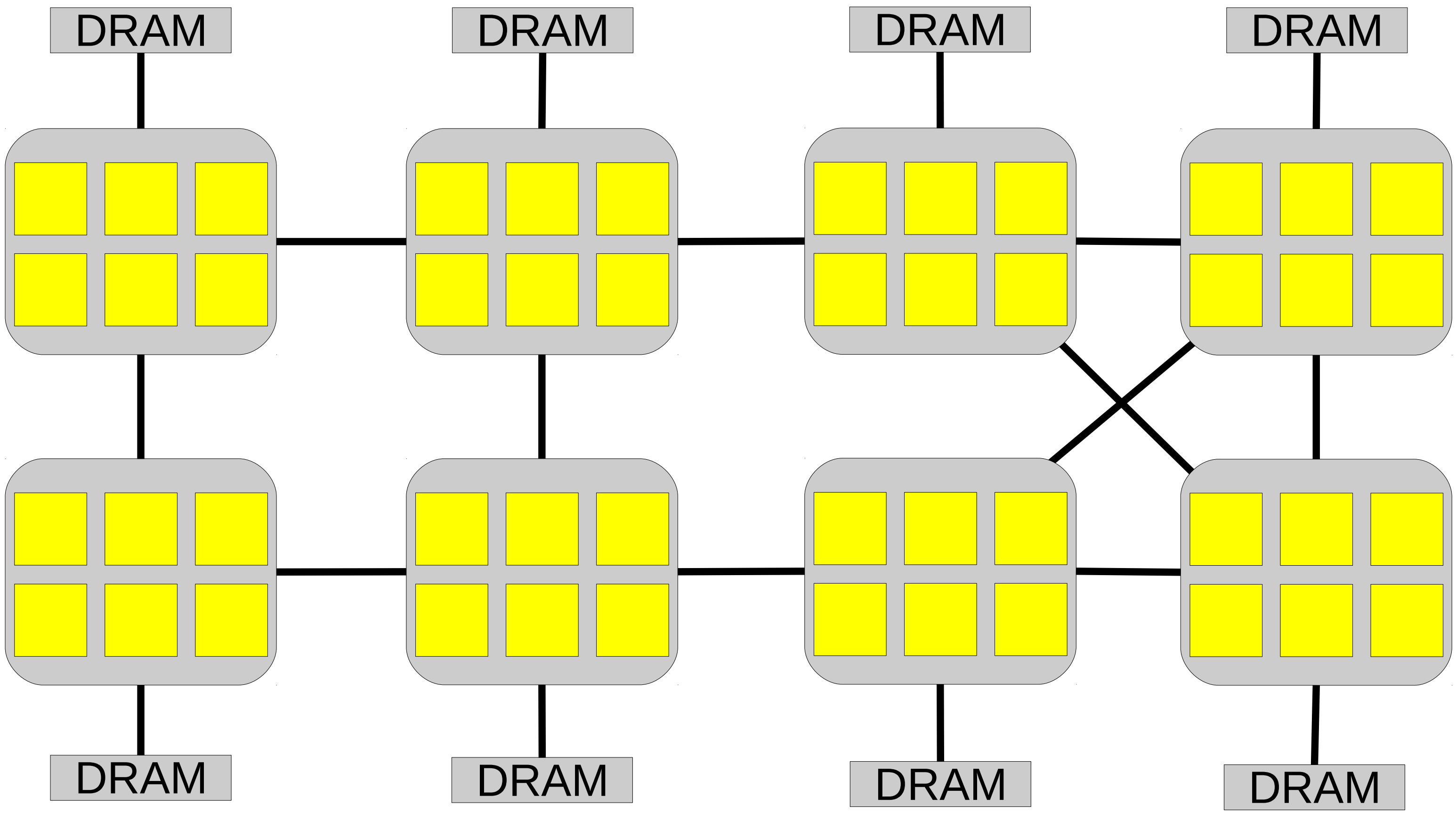
Method

- Run application
  - Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- Fix bottlenecks, re-run application

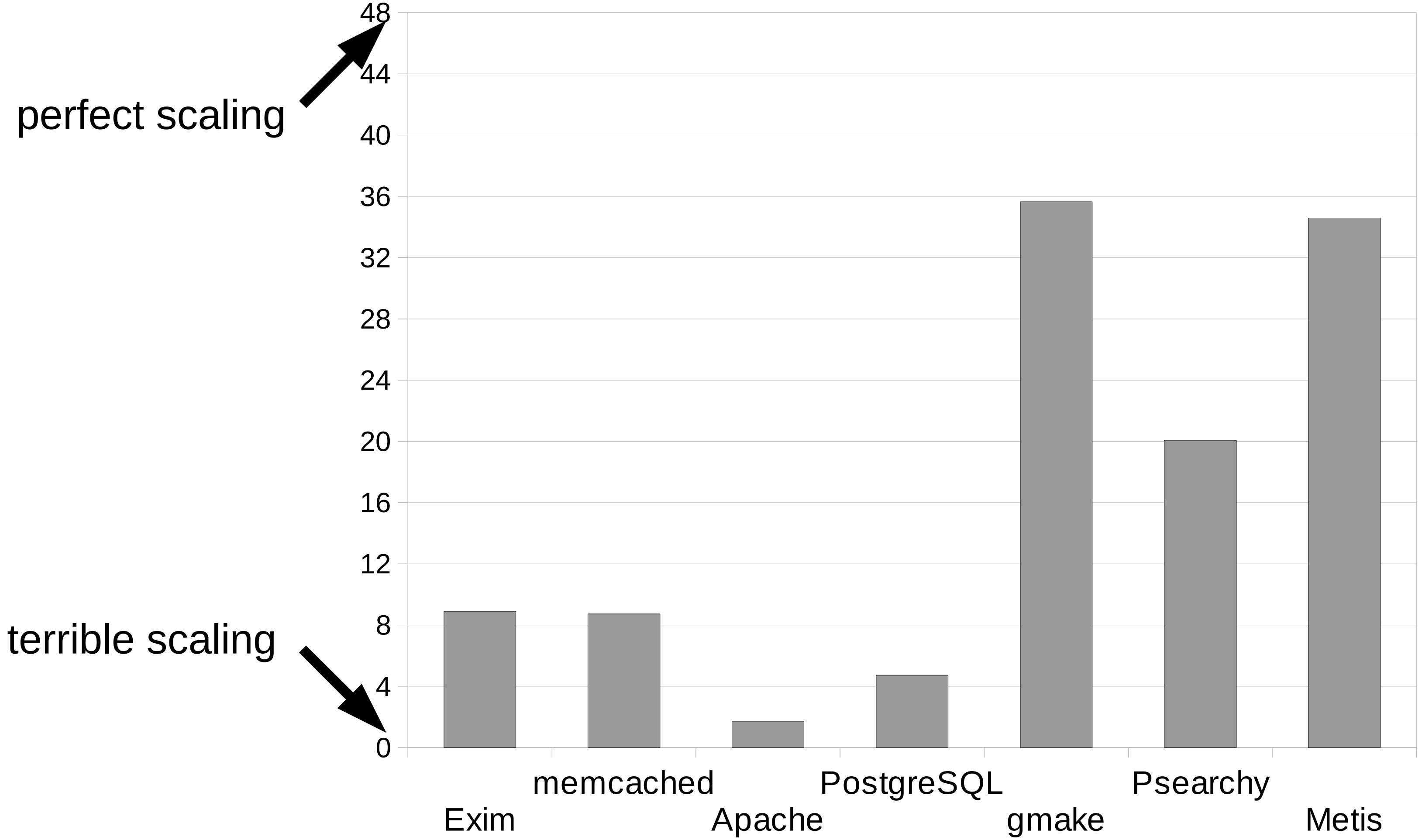
➡ Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

Off-the-shelf 48-core server

- 6 core x 8 chip AMD

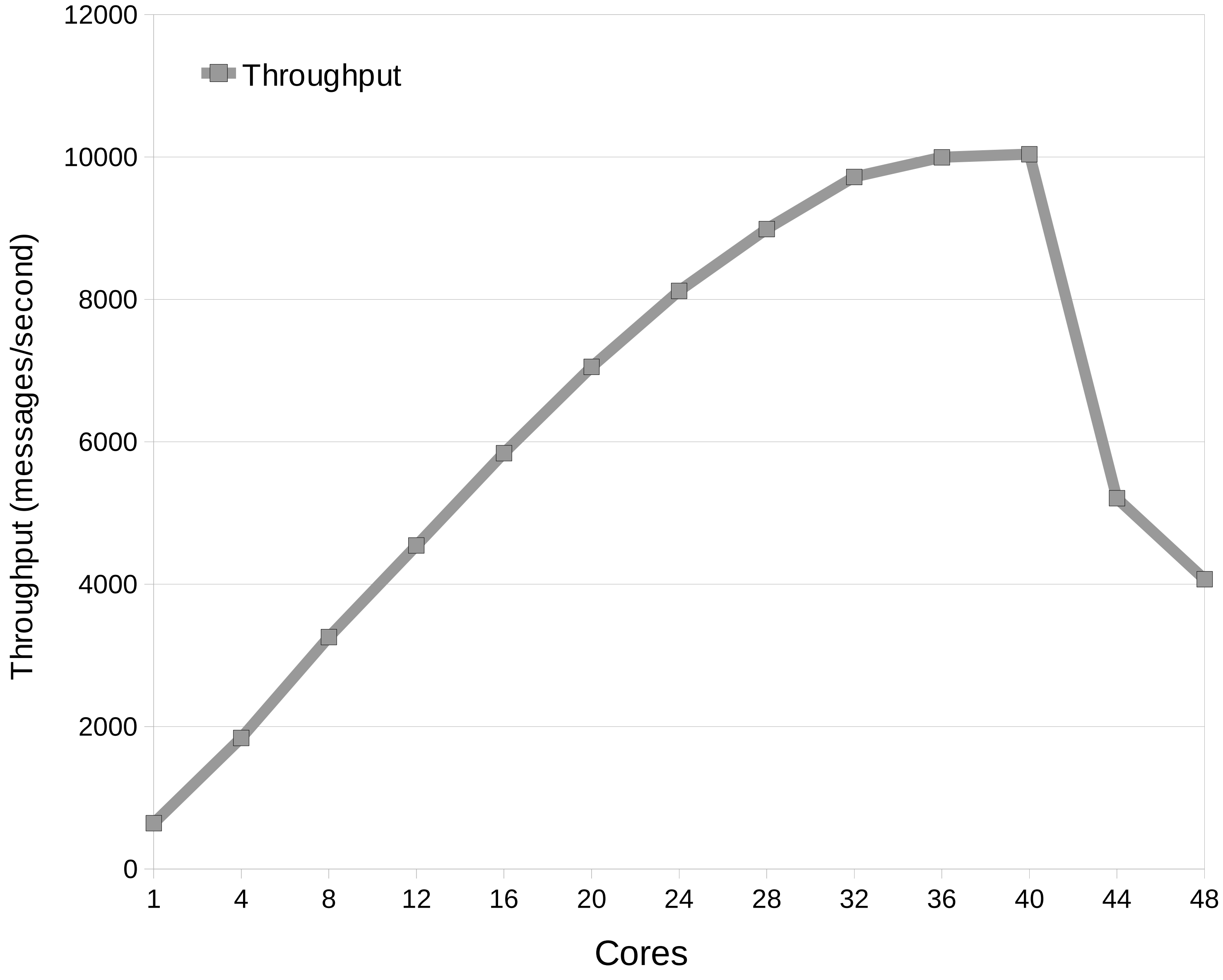


Poor scaling on stock Linux kernel

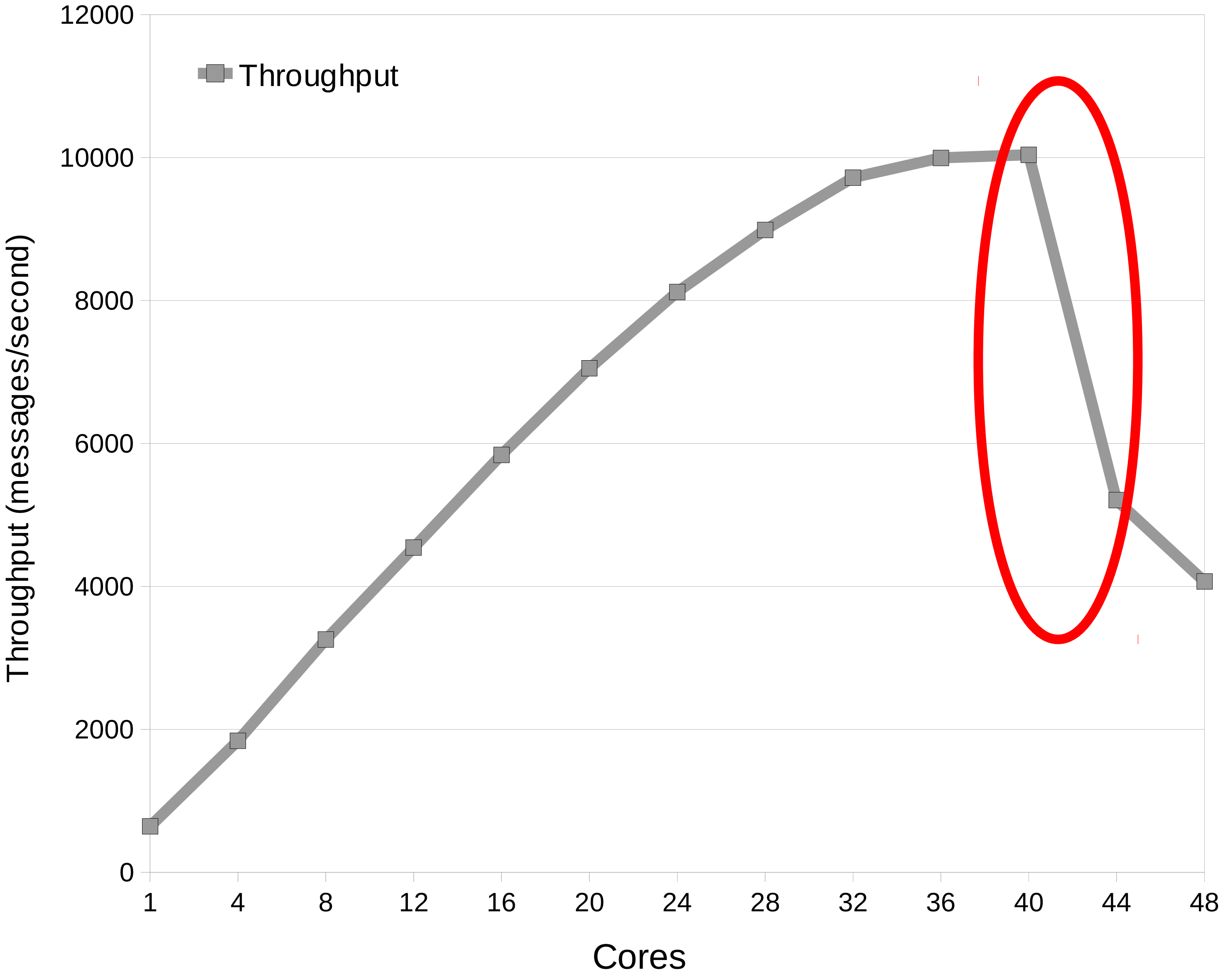


Y-axis: (throughput with 48 cores) / (throughput with one core)

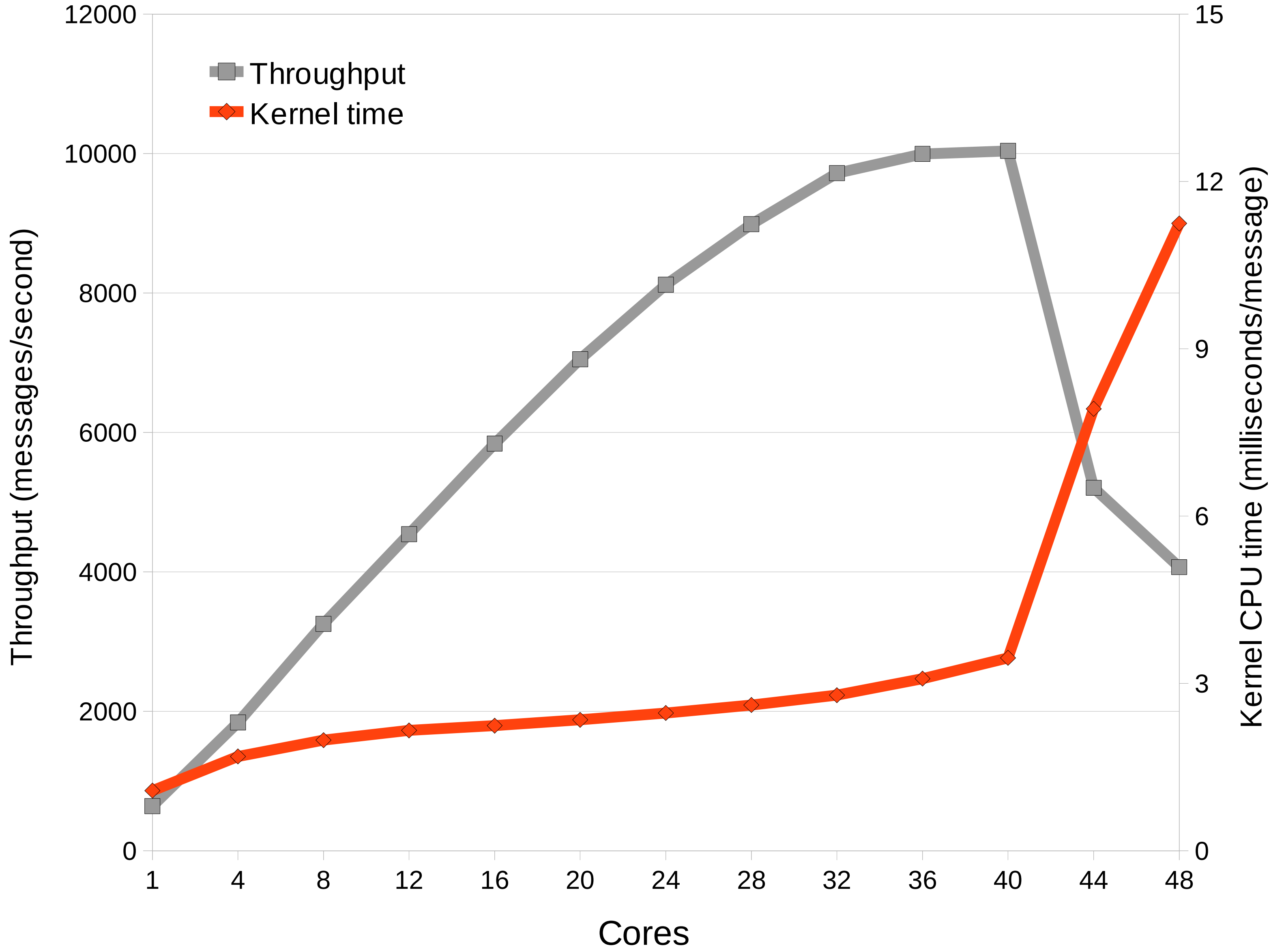
Exim on stock Linux: collapse



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

- `sys_open` eventually 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` eventually 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` eventually 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;
}
```

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

Bottleneck: reading mount table

- `sys_open` eventually 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;
}
```

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

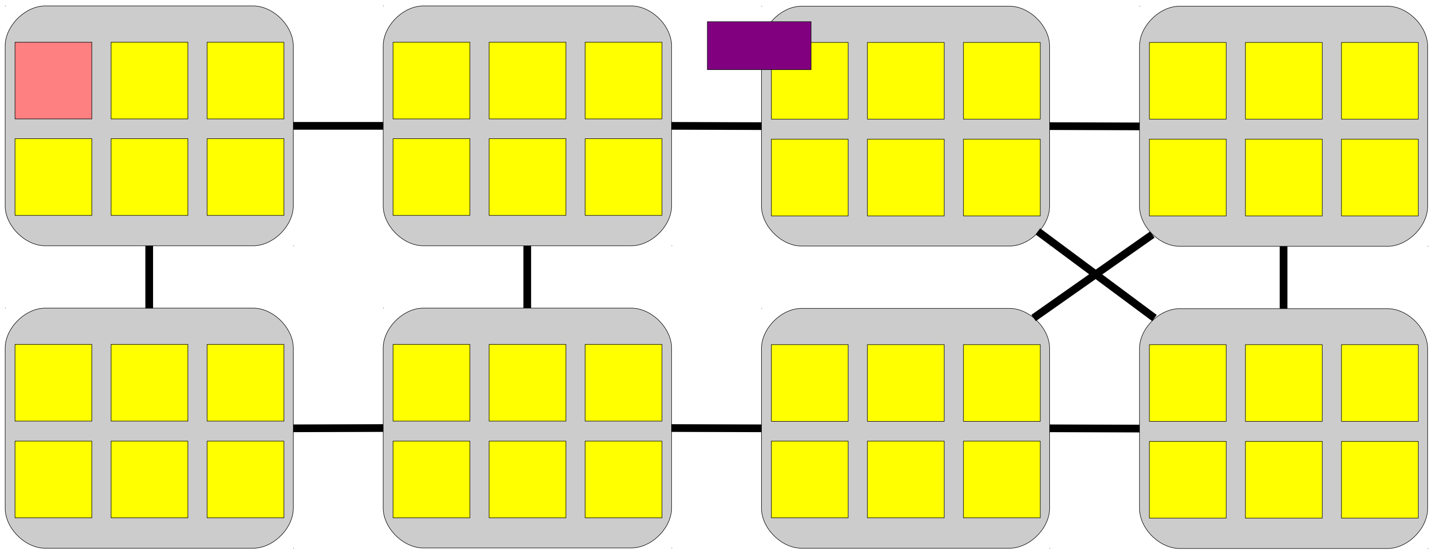
- `spin_lock` and `spin_unlock` use many more cycles than the critical section

Linux spin lock implementation

```
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;
}
```



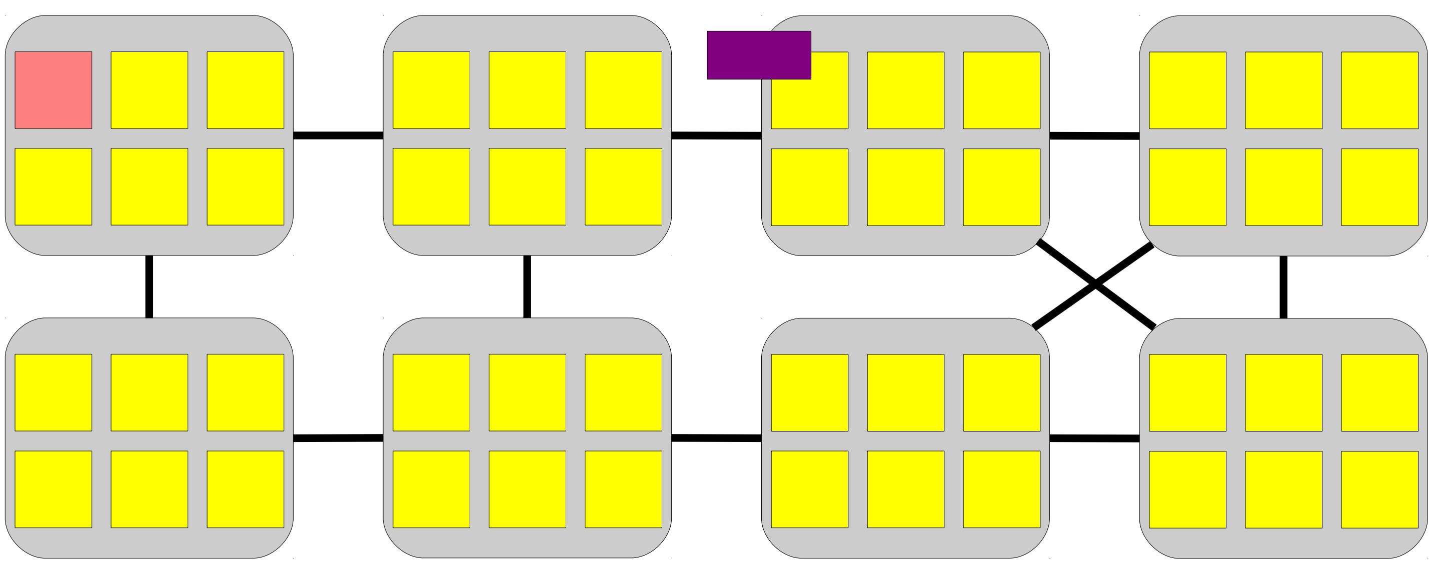
Linux spin lo… …mentation

Allocate a 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;
}
```



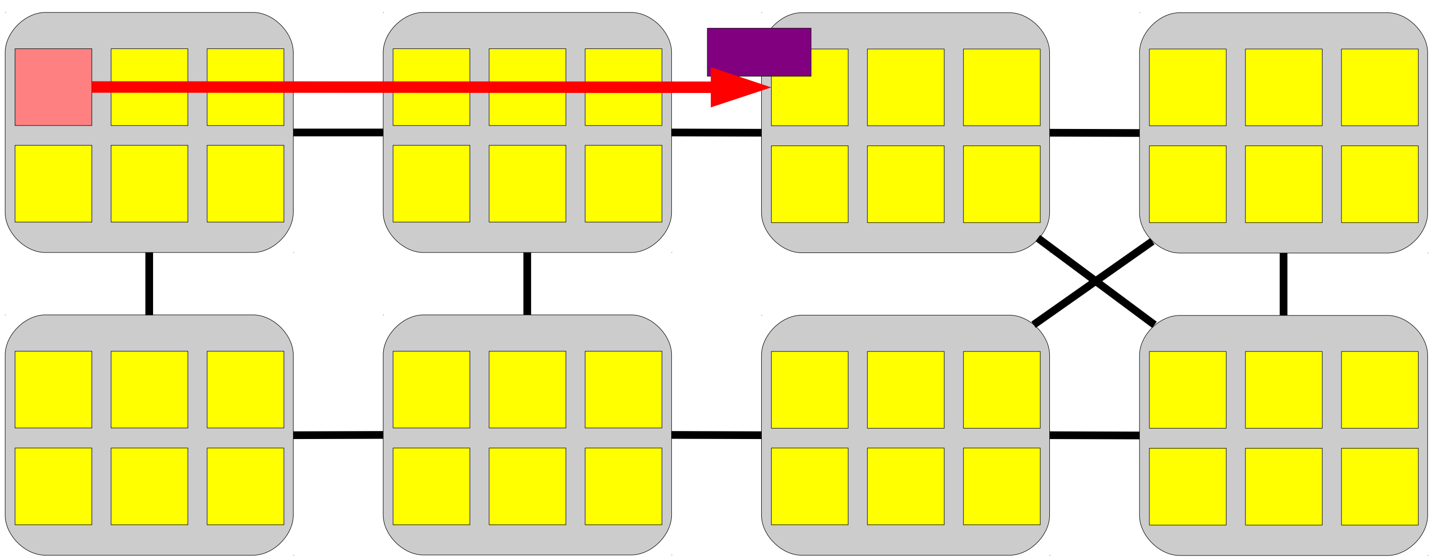
Linux spin lo… mentation

Allocate a 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;
}
```



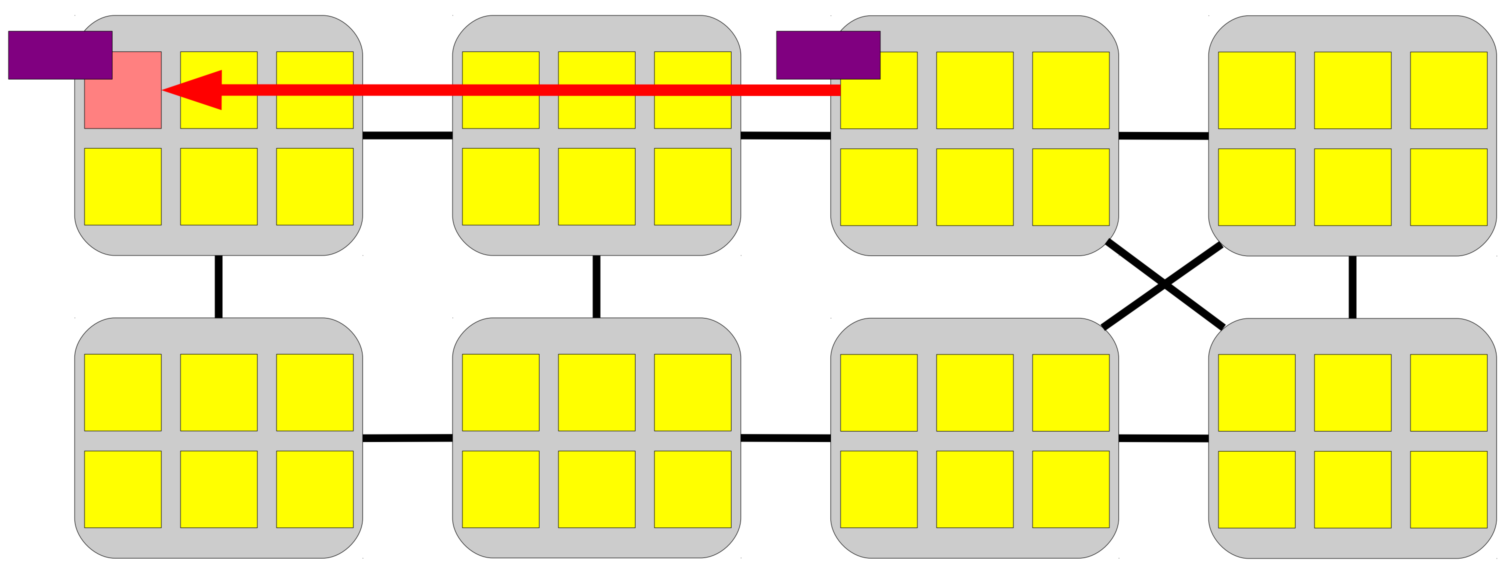
Linux spin lo… mentation

Allocate a 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;
}
```



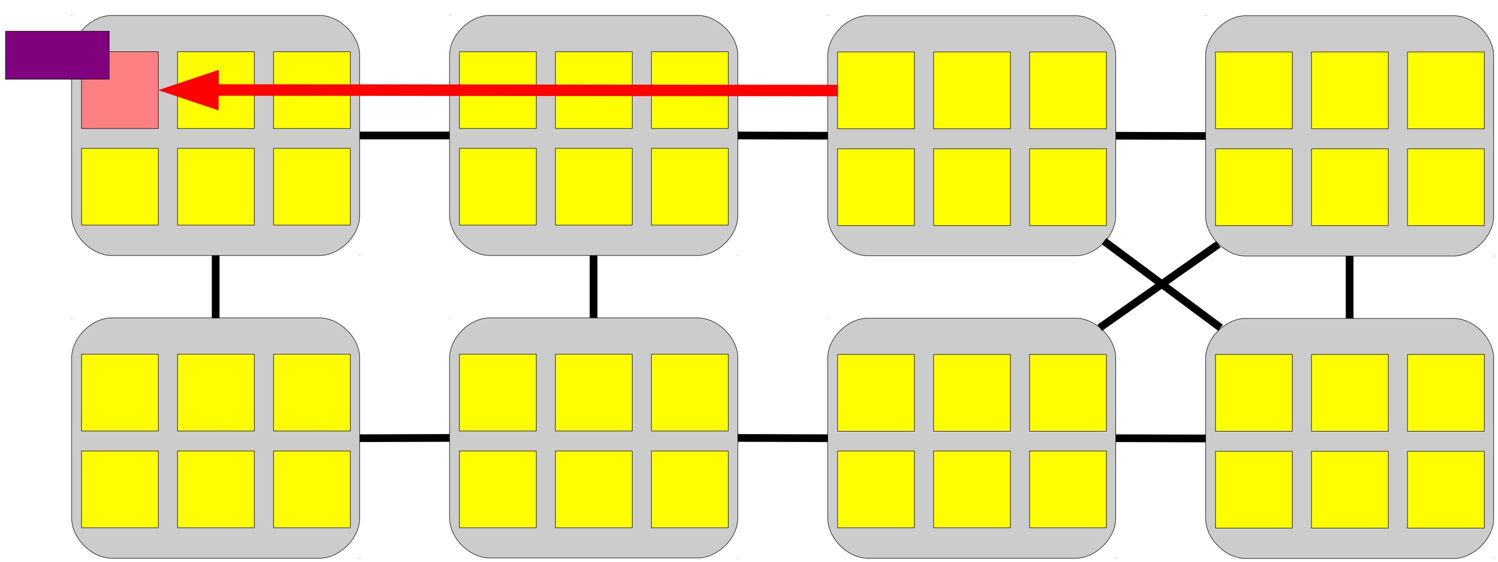
Linux spin lo … mentation

Allocate a 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;
}
```



Linux spin lo mentation

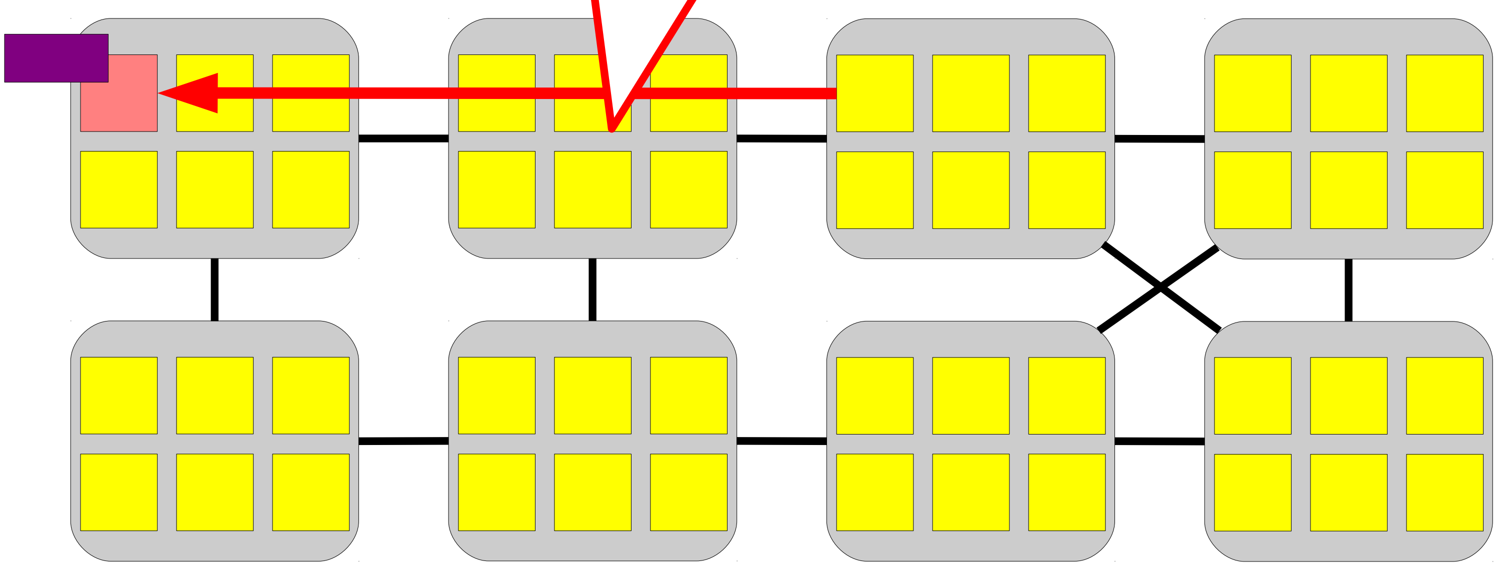
Allocate a ticket

```
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;
}
```

120 – 420 cycles



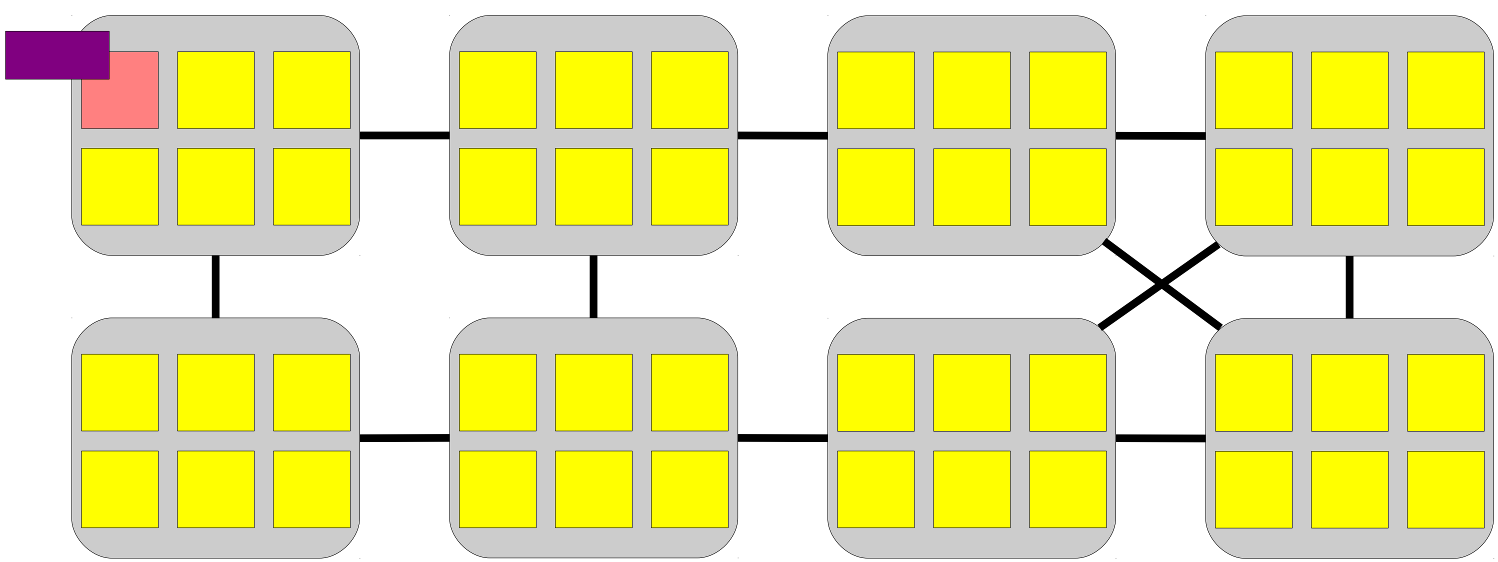
Linux spin lock implementation

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

Spin until it's my turn

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

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

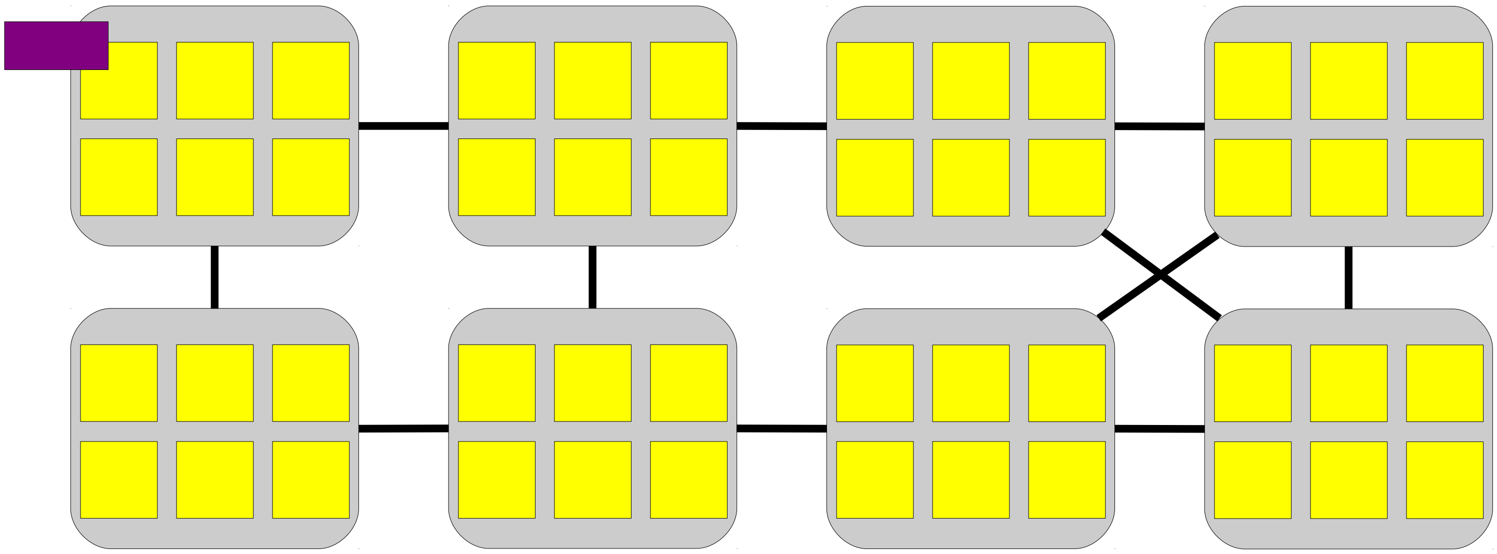


Linux spin lock implementation

```
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;
}
```



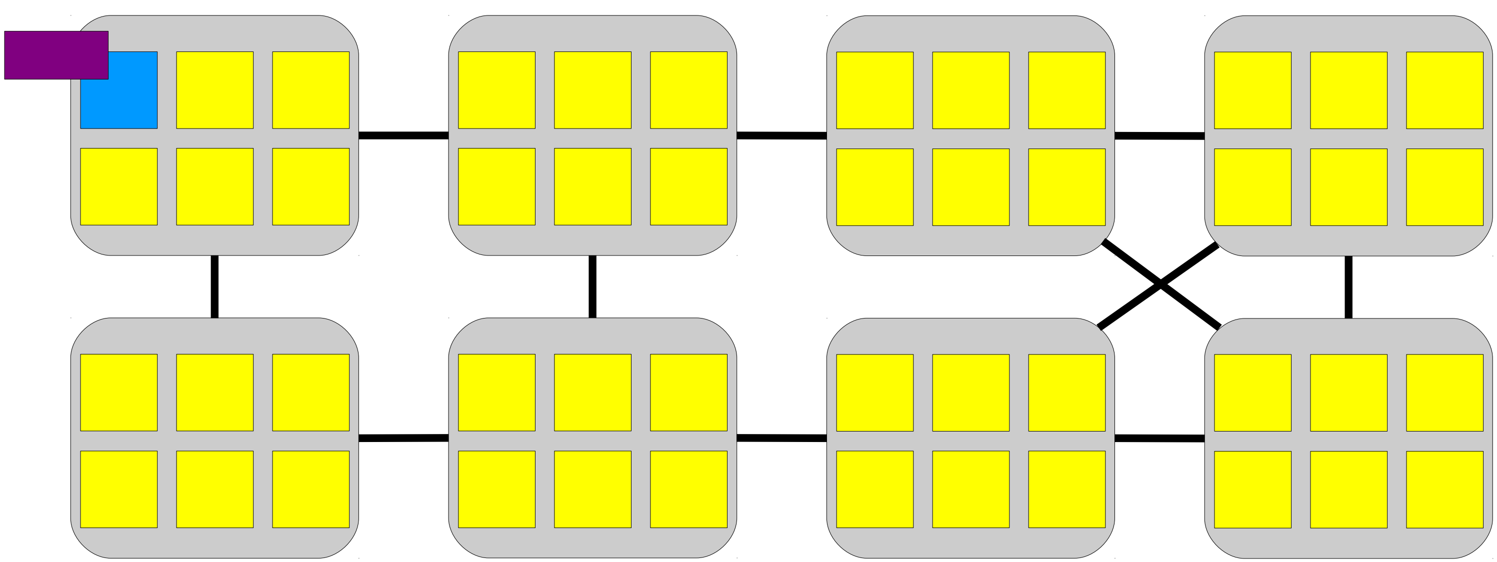
Linux spin lock implemen

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     ck)
{
    lock->current_ticket++;
}
```

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

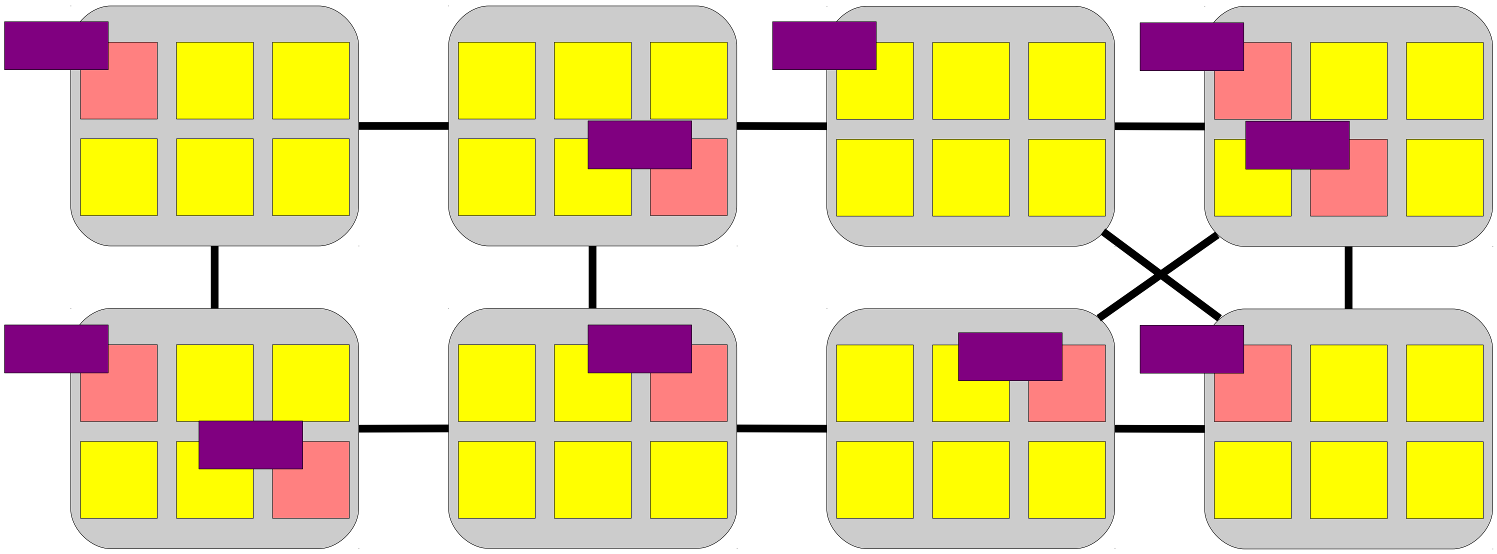


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

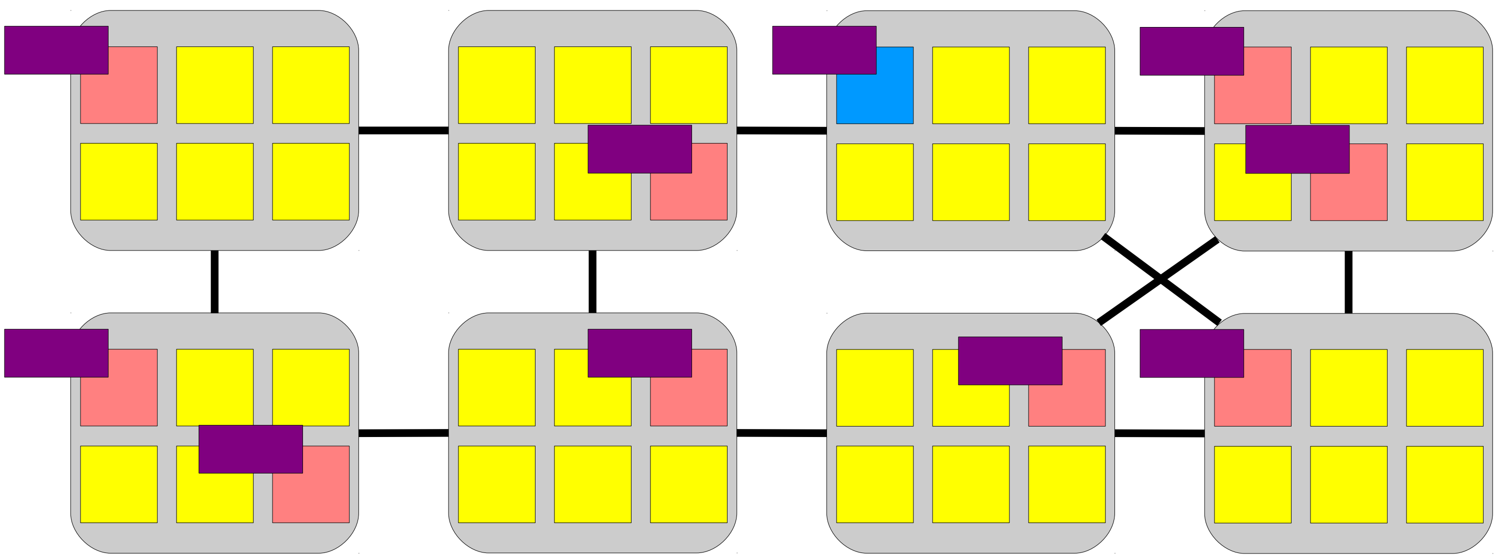


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

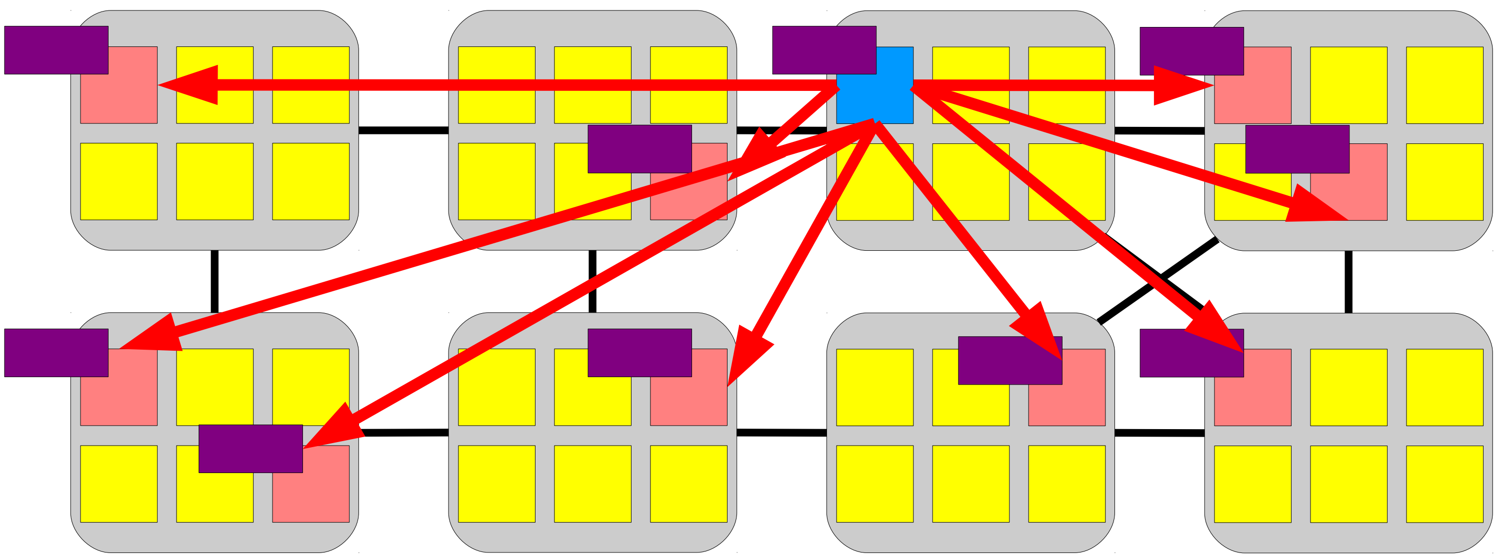


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

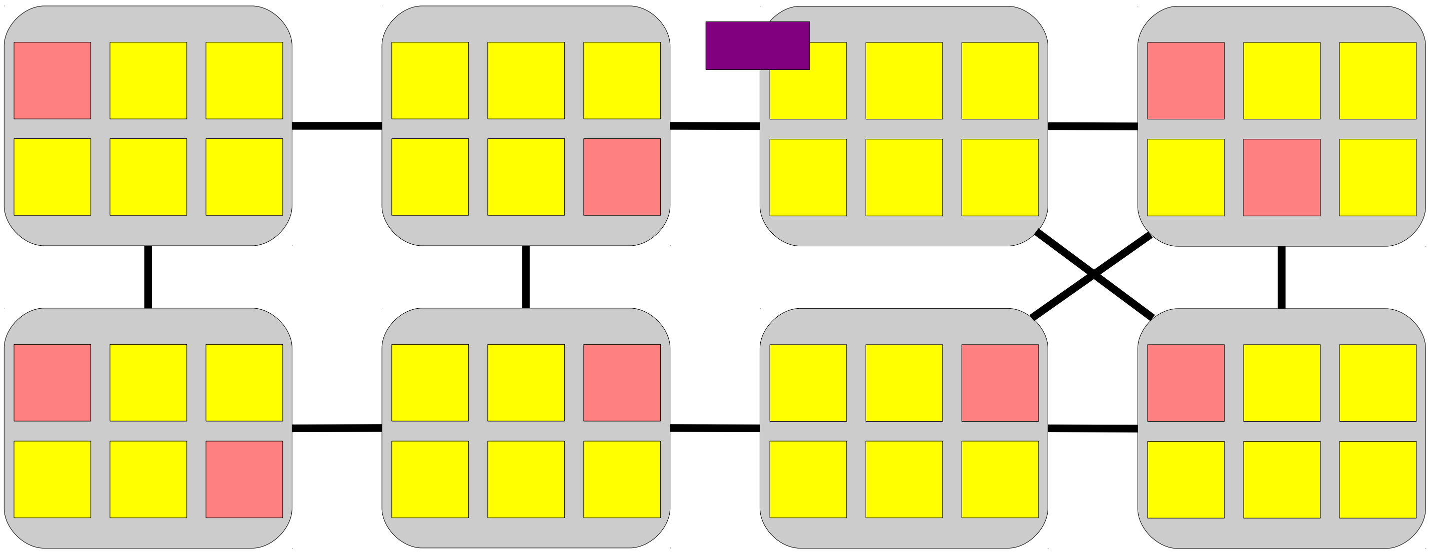


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

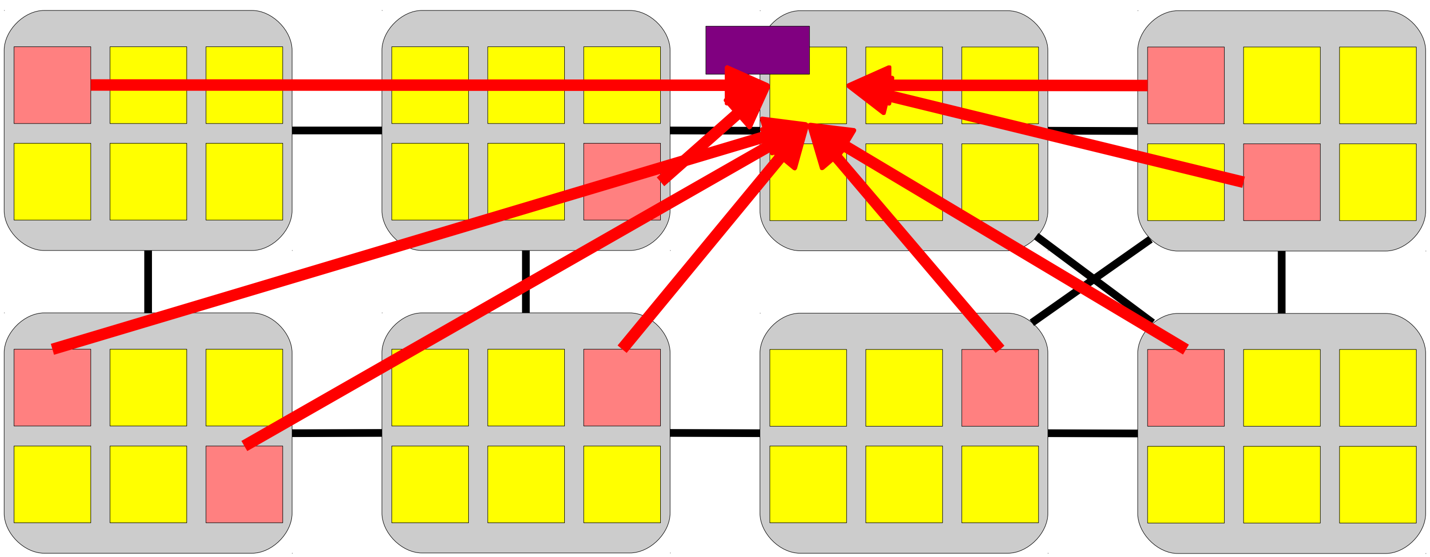


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

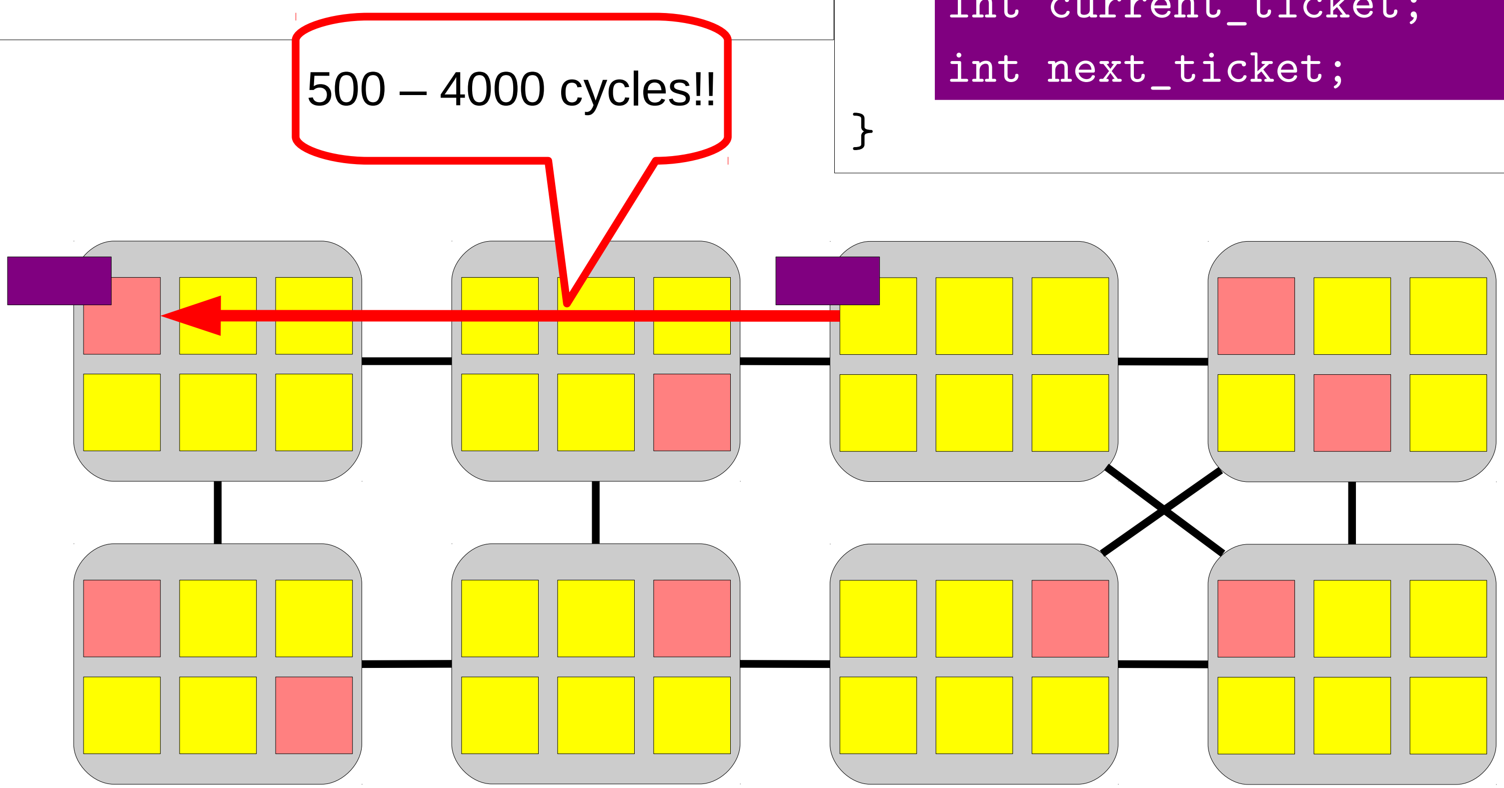


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

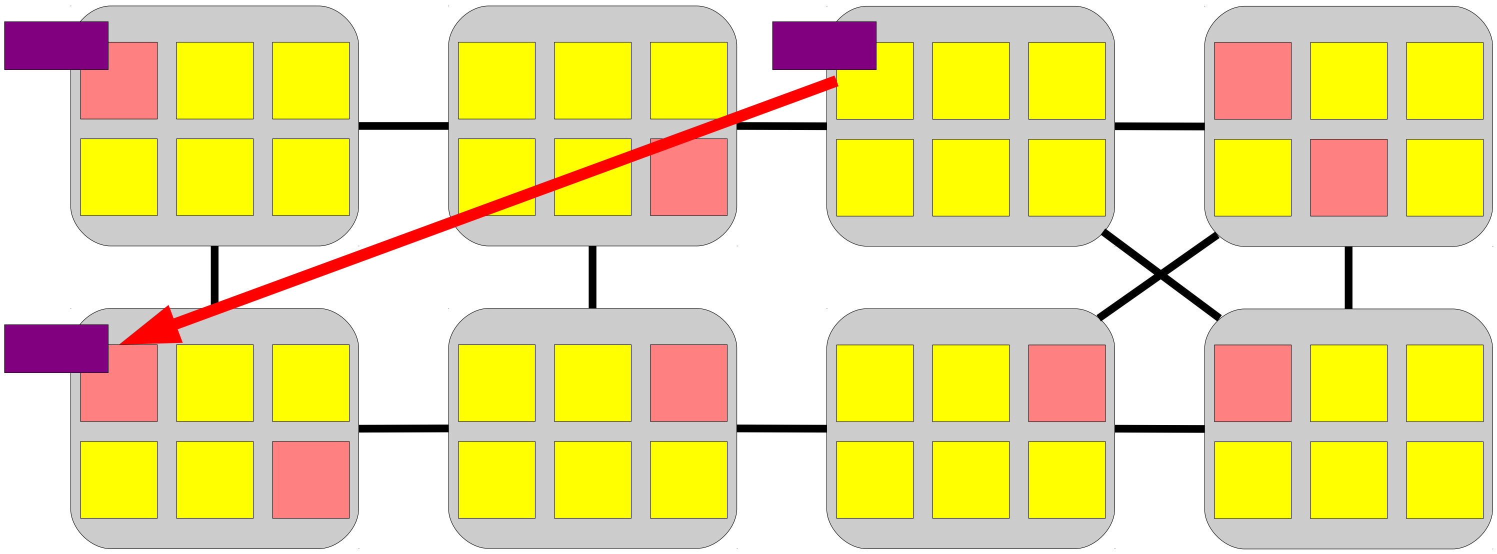


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

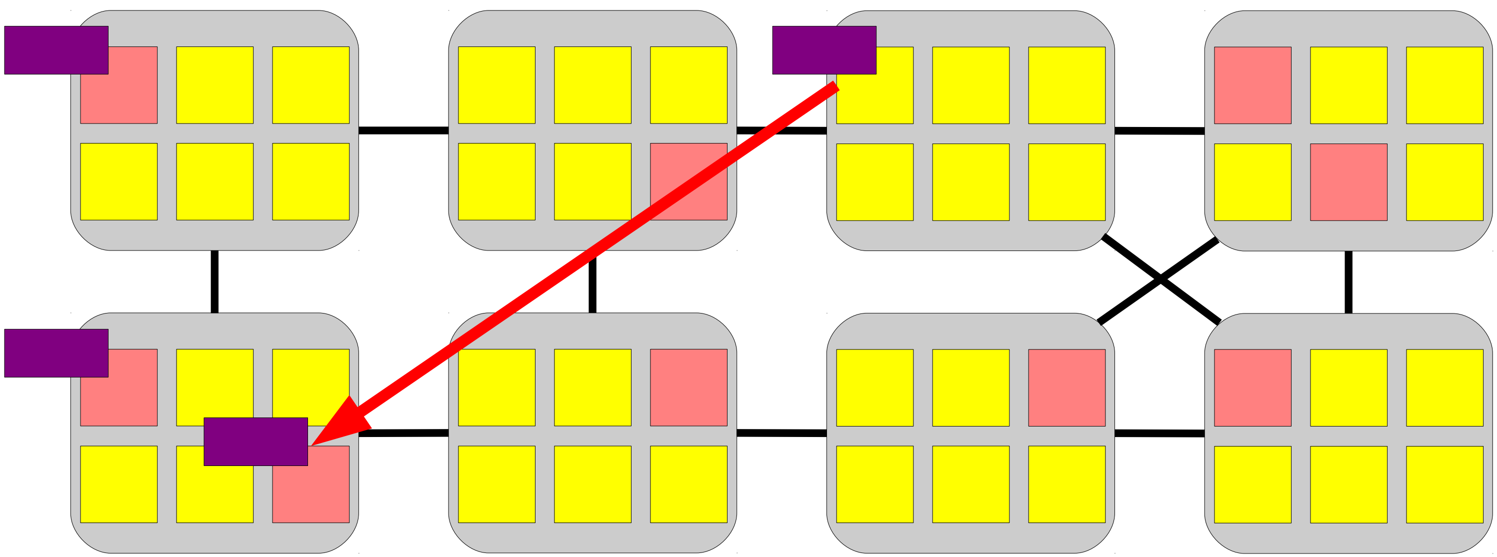


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

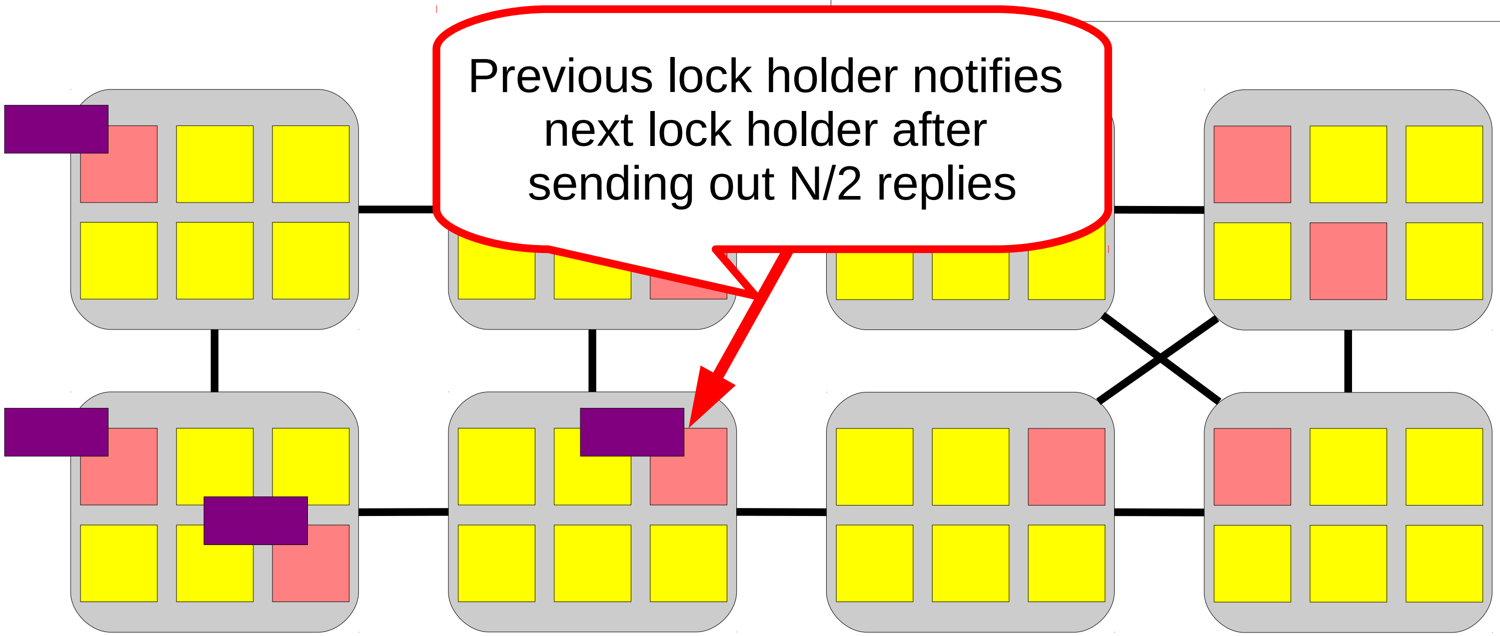


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

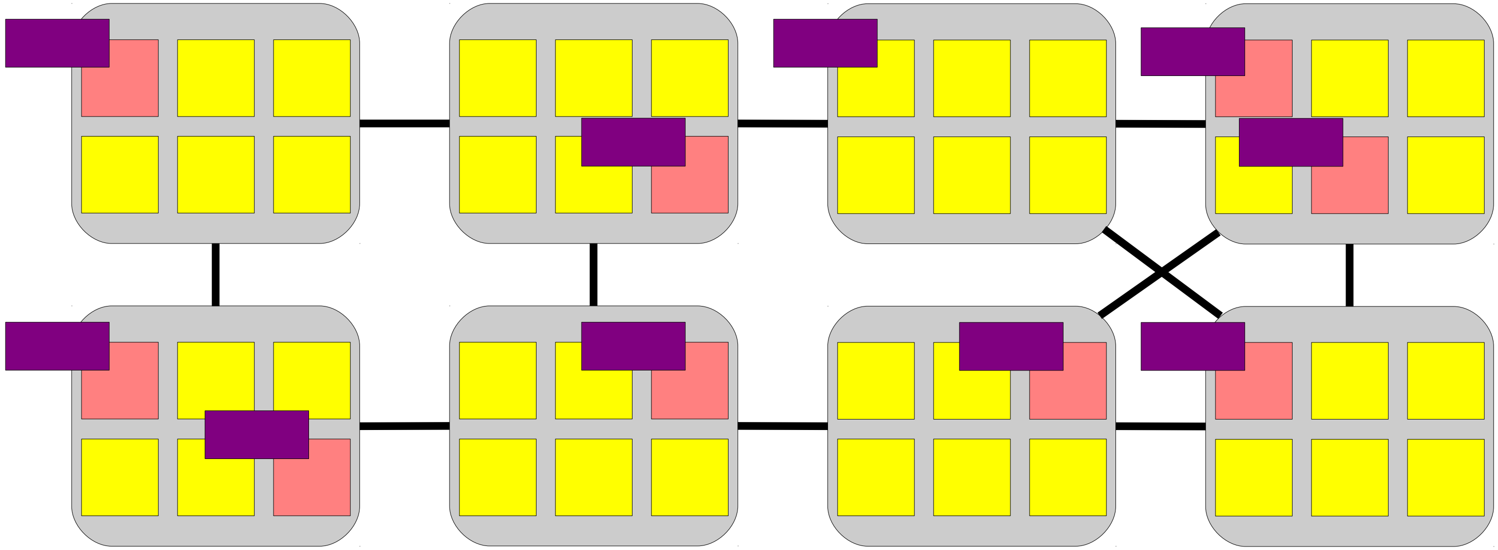


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

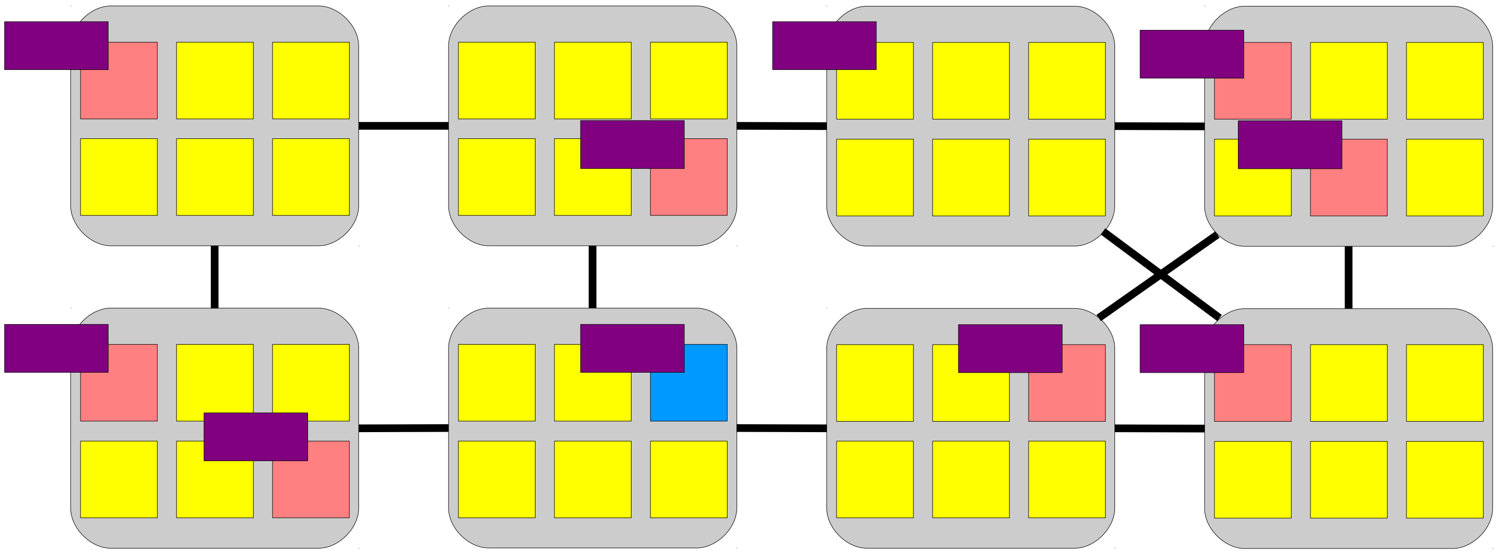


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

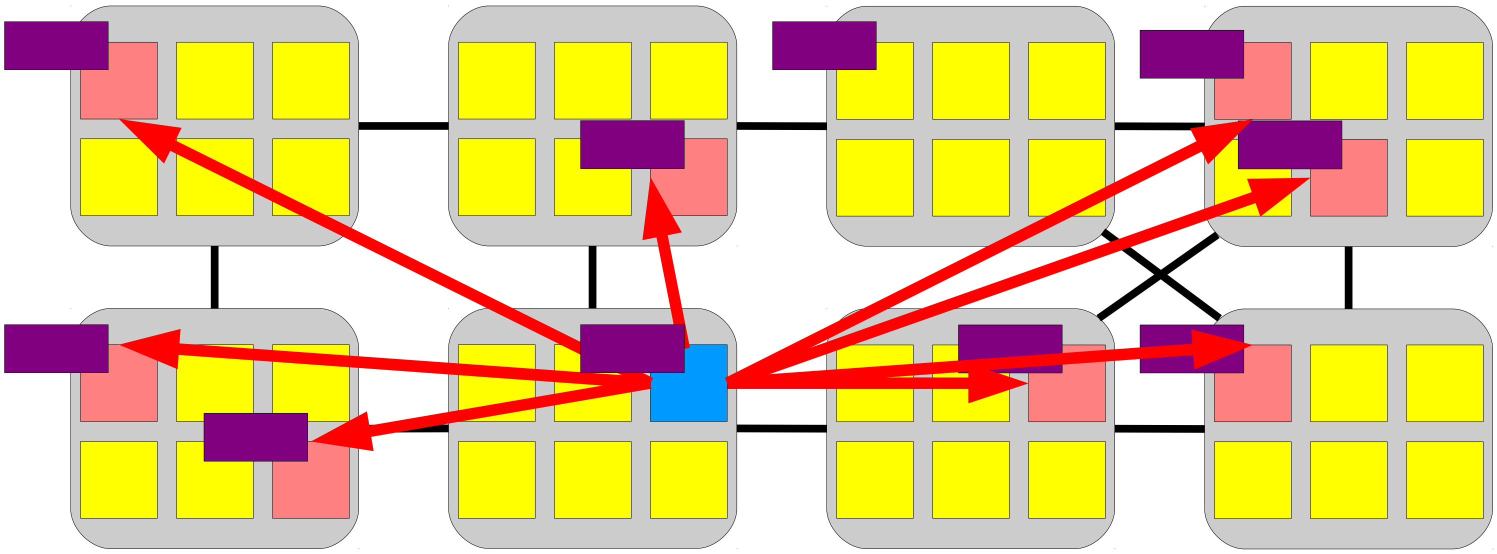


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

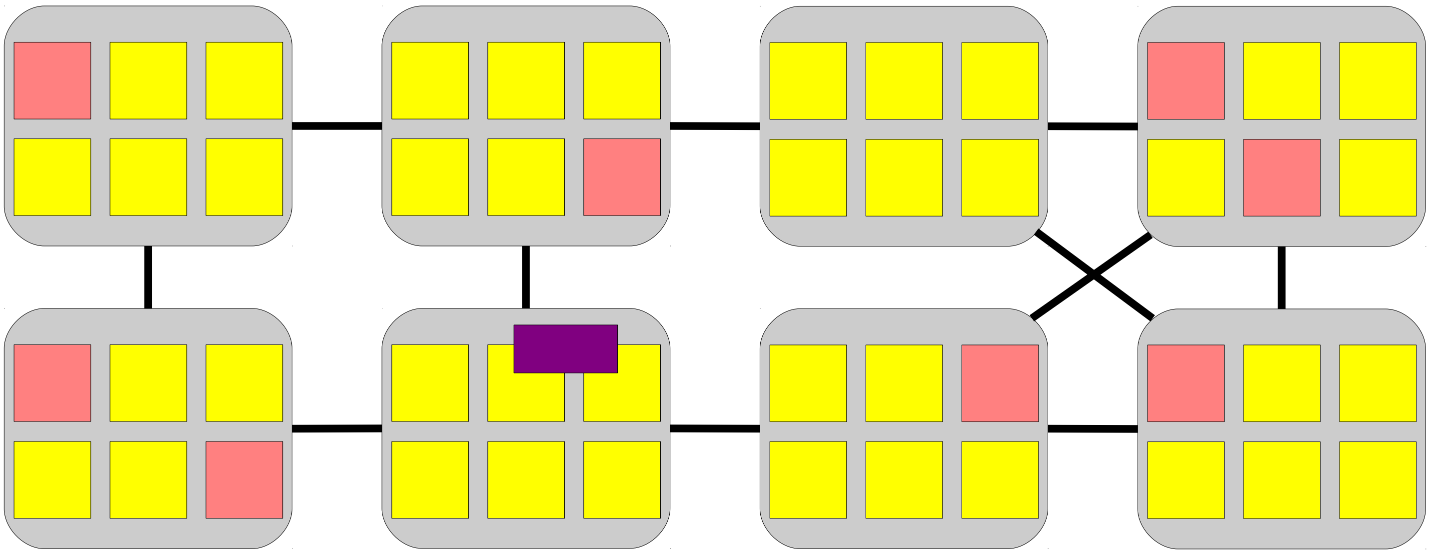


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```

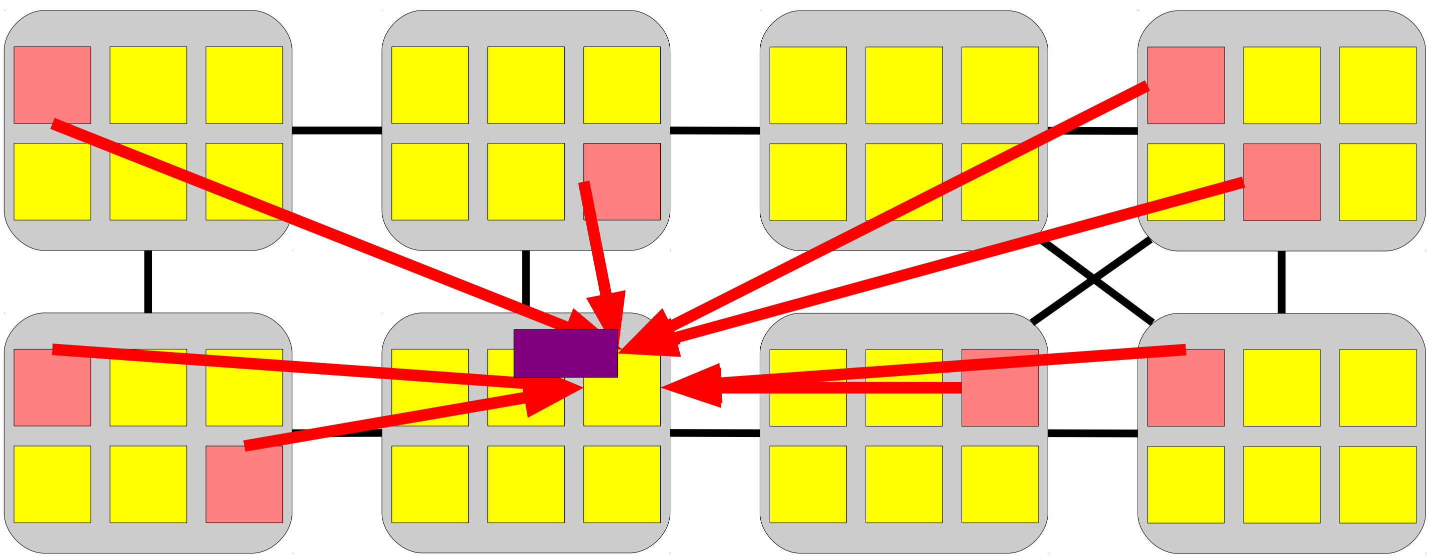


Scalability collapse caused by non-scalable locks [Anderson 90]

```
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;
}
```



Bottleneck: reading mount table

- `sys_open` eventually 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;
}
```

- Well known problem, many solutions
  - Use scalable locks [MCS 91]
  - Use message passing [Baumann 09]
  - Avoid locks in the common case

Solution: per-core mount caches

- 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;
}
```

Solution: per-core mount caches

- 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;
}
```

Solution: per-core mount caches

- 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;
}
```

- Common case: cores access per-core tables
- Modify mount table: invalidate per-core tables

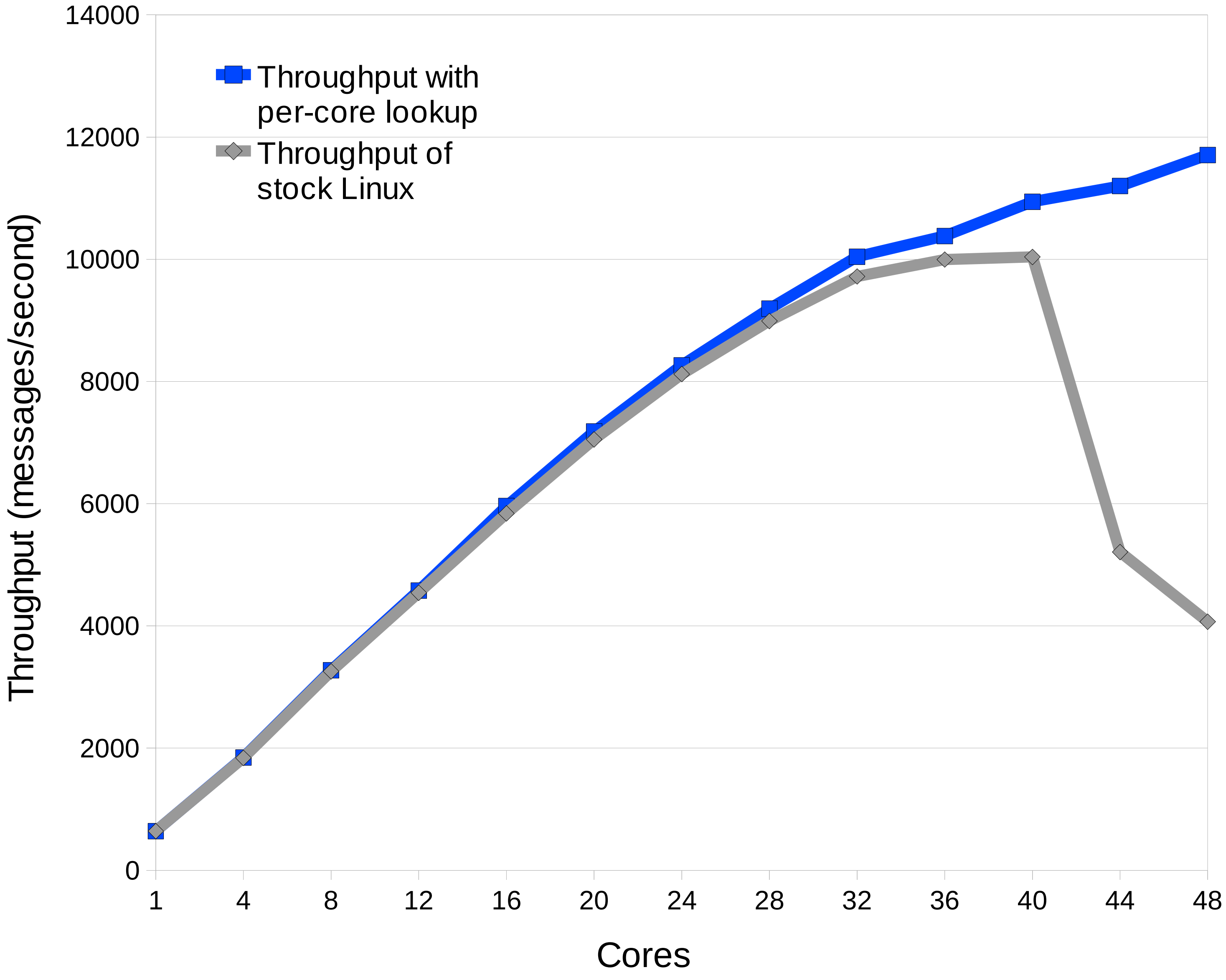
Solution: per-core mount caches

- Observation: mount table is rarely modified

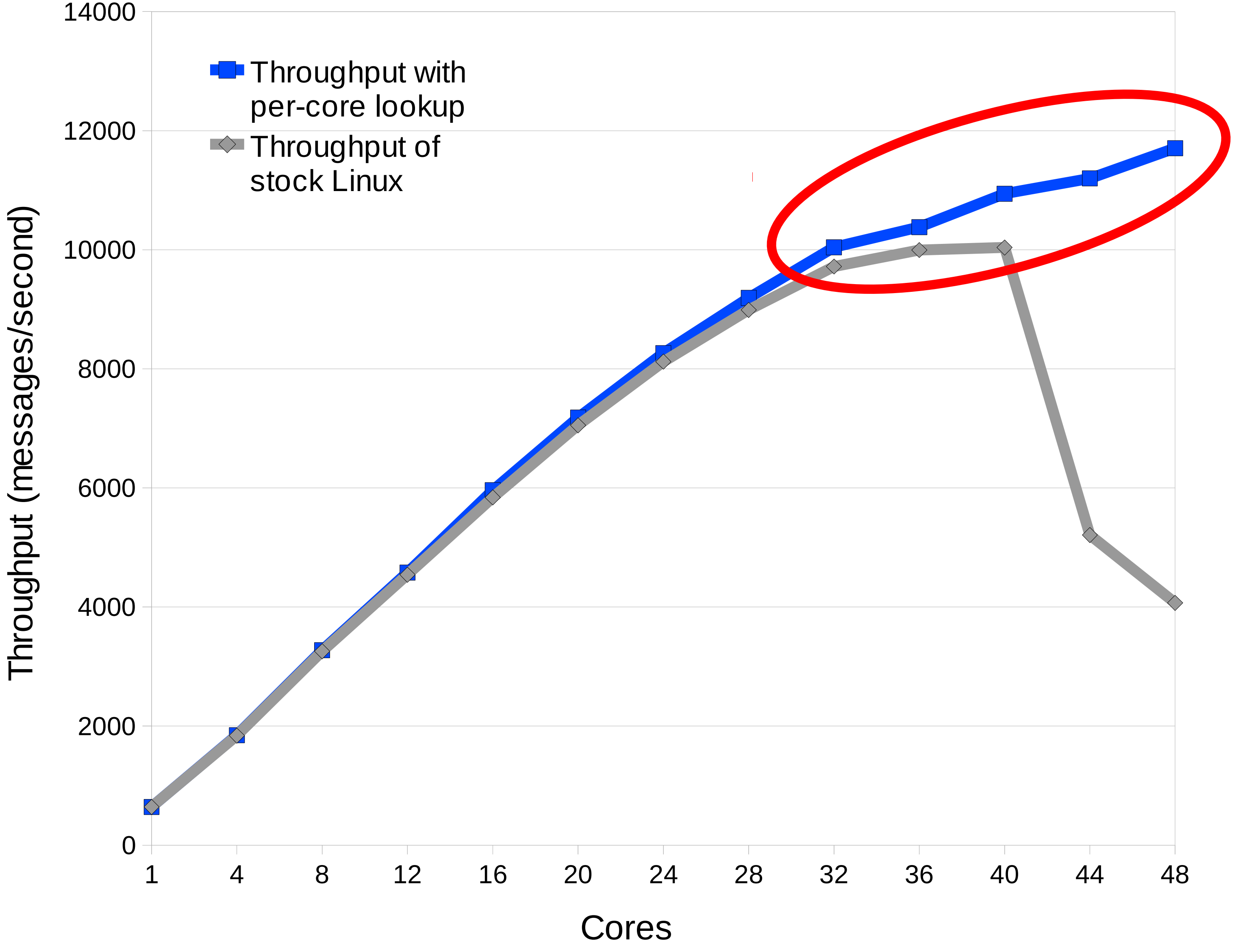
```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        if ((mnt = hash_get(percore_mnts[cpu()], path)))
                return mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        hash_put(percore_mnts[cpu()],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

- Common case: cores access per-core tables
- Modify mount table: invalidate per-core tables

Per-core lookup: scalability is better



Per-core lookup: scalability is better



No obvious bottlenecks

32 cores:
10041 msg/sec

samples	%	app name	symbol name
3319	5.4462	vmlinux	radix_tree_lookup_slot
3119	5.2462	vmlinux	unmap_vmas
1966	3.3069	vmlinux	filemap_fault
1950	3.2800	vmlinux	page_fault
1627	2.7367	vmlinux	unlock_page
1626	2.7350	vmlinux	clear_page_c
1578	2.6542	vmlinux	kmem_cache_free

48 cores:
11705 msg/sec

samples	%	app name	symbol name
4207	5.3145	vmlinux	radix_tree_lookup_slot
4191	5.2943	vmlinux	unmap_vmas
2632	3.3249	vmlinux	page_fault
2525	3.1897	vmlinux	filemap_fault
2210	2.7918	vmlinux	clear_page_c
2131	2.6920	vmlinux	kmem_cache_free
2000	2.5265	vmlinux	dput

- Functions execute more slowly on 48 cores

No obvious bottlenecks

32 cores:
10041 msg/sec

samples	%	app name	symbol name
3319	5.4462	vmlinux	radix_tree_lookup_slot
3119	5.2462	vmlinux	unmap_vmas
1966	3.3069	vmlinux	filemap_fault
1950	3.2800	vmlinux	page_fault
1627	2.7367	vmlinux	unlock_page
1626	2.7350	vmlinux	clear_page_c
1578	2.6542	vmlinux	kmem_cache_free

48 cores:
11705 msg/sec

samples	%	app name	symbol name
4207	5.3145	vmlinux	radix_tree_lookup_slot
4191	5.2943	vmlinux	unmap_vmas
2632	3.3249	vmlinux	page_fault
2525	3.1897	vmlinux	filemap_fault
2210	2.7918	vmlinux	clear_page_c
2131	2.6920	vmlinux	kmem_cache_free
2000	2.5265	vmlinux	dput

- Functions execute more slowly on 48 cores

No obvious bottlenecks

32 cores:
10041 msg/sec

samples	%	app name	symbol name
3319	5.4462	vmlinux	radix_tree_lookup_slot
3119	5.2462	vmlinux	unmap_vmas
1966	3.3069	vmlinux	filemap_fault
1950	3.2800	vmlinux	page_fault
1627	2.7367	vmlinux	unlock_page
1626	2.7350	vmlinux	clear_page_c
1578	2.6542	vmlinux	kmem_cache_free

48 cores:
11705 msg/sec

samples	%	app name	symbol name
4207	5.3145	vmlinux	radix_tree_lookup_slot
4191	5.2943	vmlinux	unmap_vmas
2632	3.3249	vmlinux	page_fault
2525	3.1897	vmlinux	filemap_fault
2210	2.7918	vmlinux	clear_page_c
2131	2.6920	vmlinux	kmem_cache_free
2000	2.5265	vmlinux	dput

- Functions execute more slowly on 48 cores

No obvious bottlenecks

32 cores:
10041 msg/sec

samples	%	app name	symbol name
3319	5.4462	vmlinux	radix_tree_lookup_slot
3119	5.2462	vmlinux	unmap_vmas
1966	3.3069	vmlinux	filemap_fault
1950	3.2800	vmlinux	page_fault
1627	2.7367	vmlinux	unlock_page
1626	2.7350	vmlinux	clear_page_c
1578	2.6542	vmlinux	kmem_cache_free

48 cores:
11705 msg/sec

samples	%	app name	symbol name
4207	5.3145	vmlinux	radix_tree_lookup_slot
4191	5.2943	vmlinux	[illegible]
2632	3.3249	vmlinux	[illegible]
2525	3.1897	vmlinux	[illegible]
2210	2.7918	vmlinux	c _page_c
2131	2.6920	vmlinux	kme _cache_free
2000	2.5265	vmlinux	dput

dput is causing other functions to slow down

- Functions execute more slowly on 48 cores

Bottleneck: reference counting

- Ref count indicates if kernel can free object
  - File name cache (`dentry`), physical pages, ...

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

Bottleneck: reference counting

- Ref count indicates if kernel can free object
  - File name cache (`dentry`), physical pages, ...

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

A single atomic instruction limits scalability?!

Bottleneck: reference counting

- Ref count indicates if kernel can free object
  - File name cache (`dentry`), physical pages, ...

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

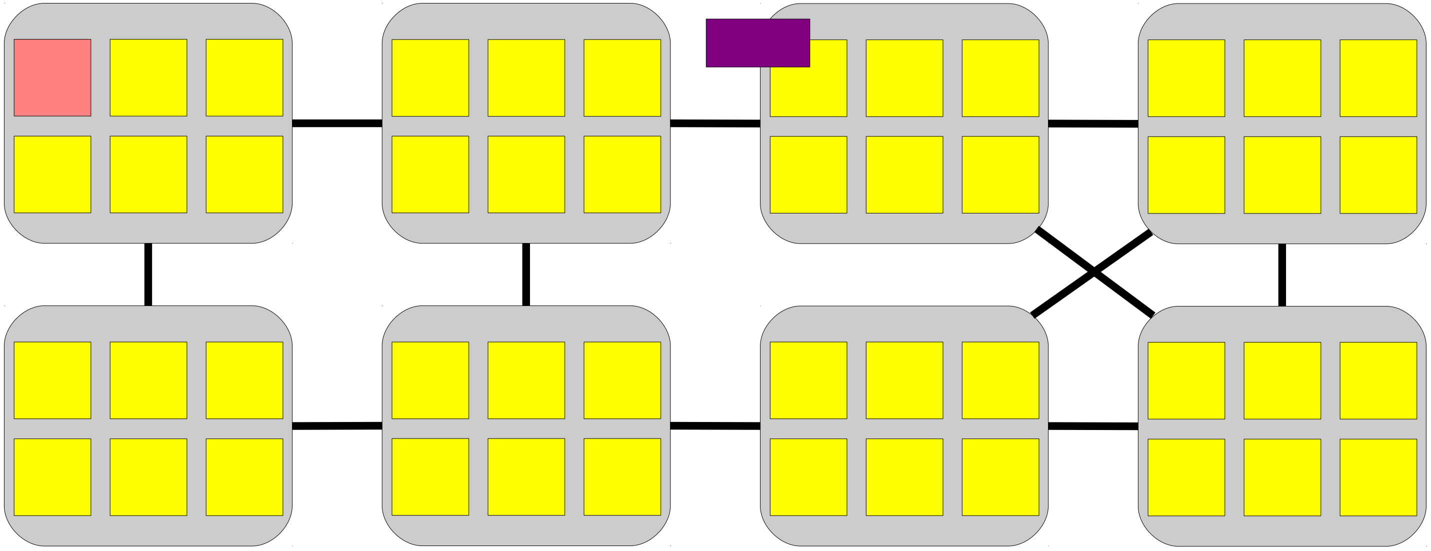
A single atomic instruction limits scalability?!

- Reading the reference count is slow
- Reading the reference count delays memory operations from other cores

Reading reference count is slow

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

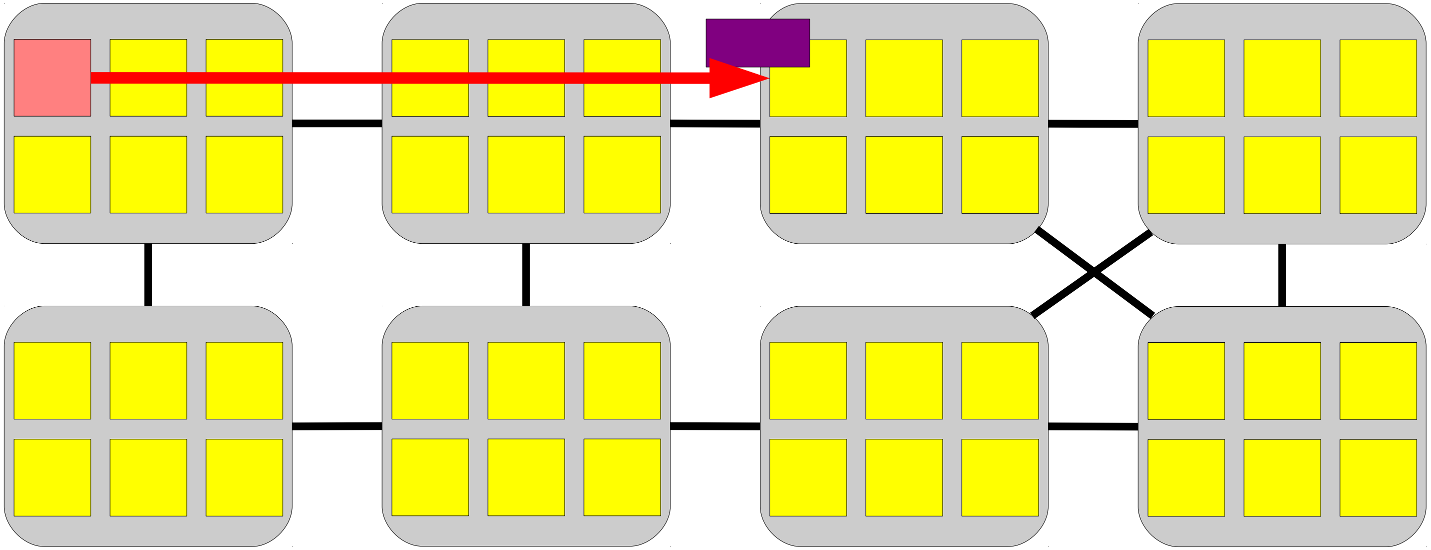
```
struct dentry {
    …
    int ref;
    …
};
```



Reading reference count is slow

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

```
struct dentry {
    …
    int ref;
    …
};
```

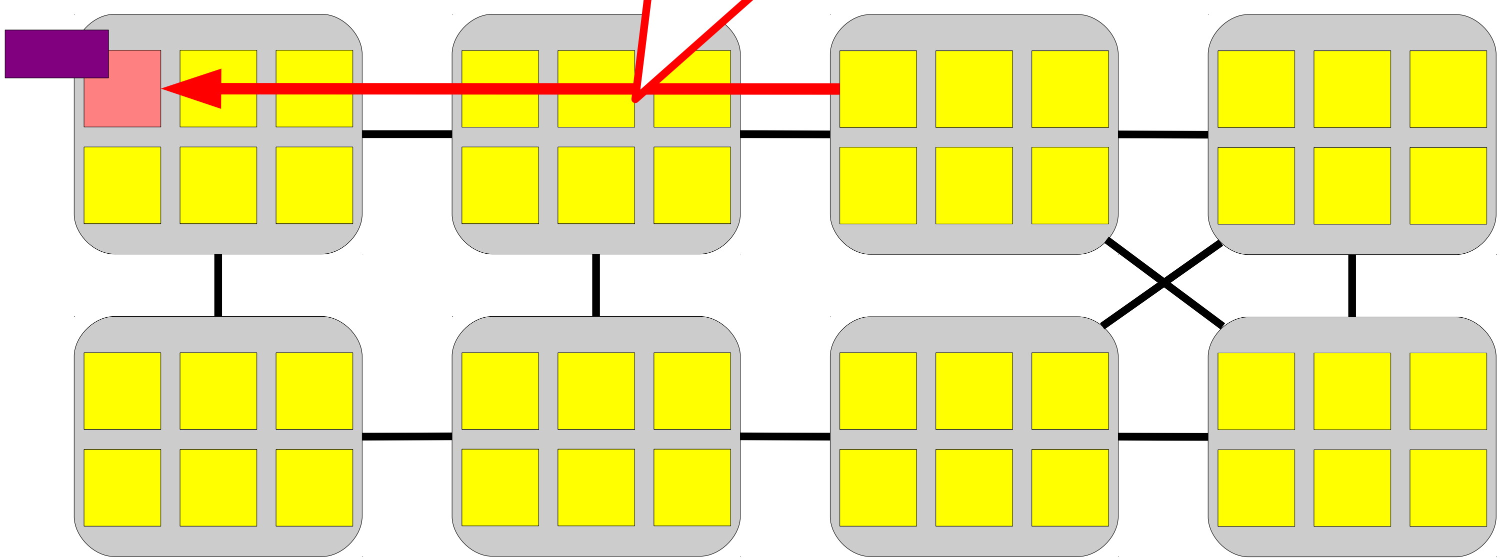


Reading reference count is slow

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(de
}
```

```
struct dentry {
    …
    int ref;
    …
};
```

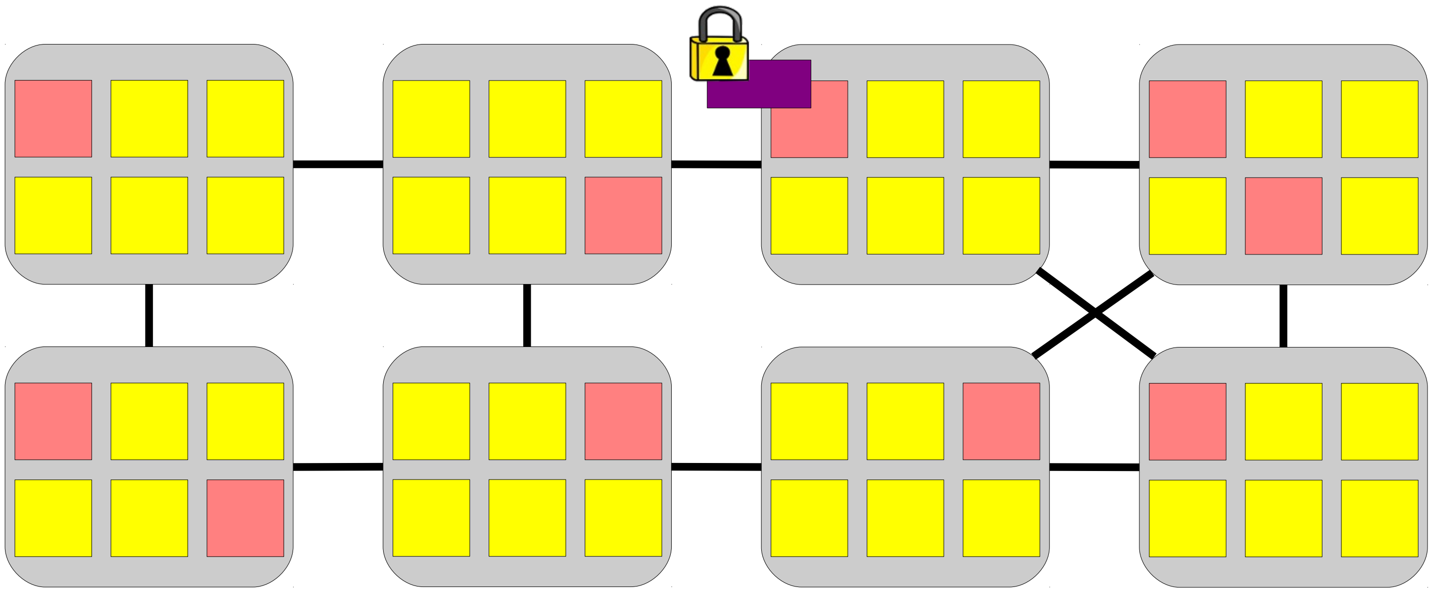
120 – 4000 cycles
depending on congestion



Reading the reference count delays memory operations from other cores

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

```
struct dentry {
    …
    int ref;
    …
};
```

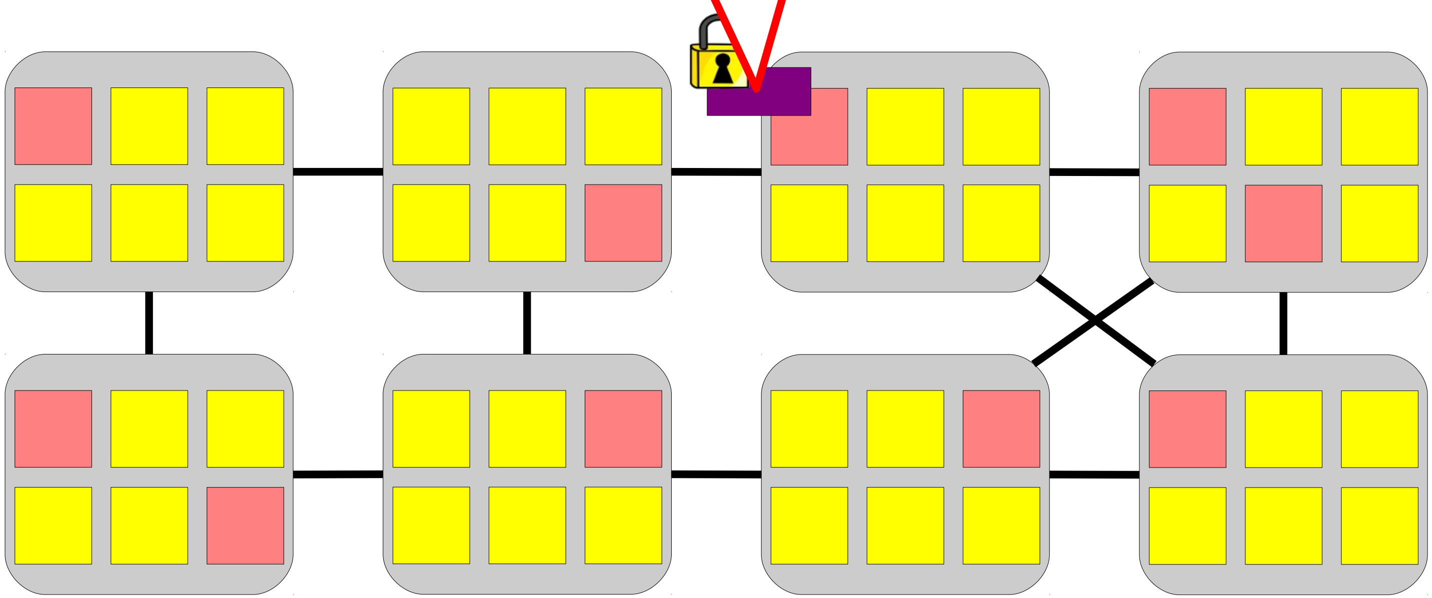


Reading the reference count delays memory operations from other cores

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

```
struct dentry {
    …
    int ref;
    …
};
```

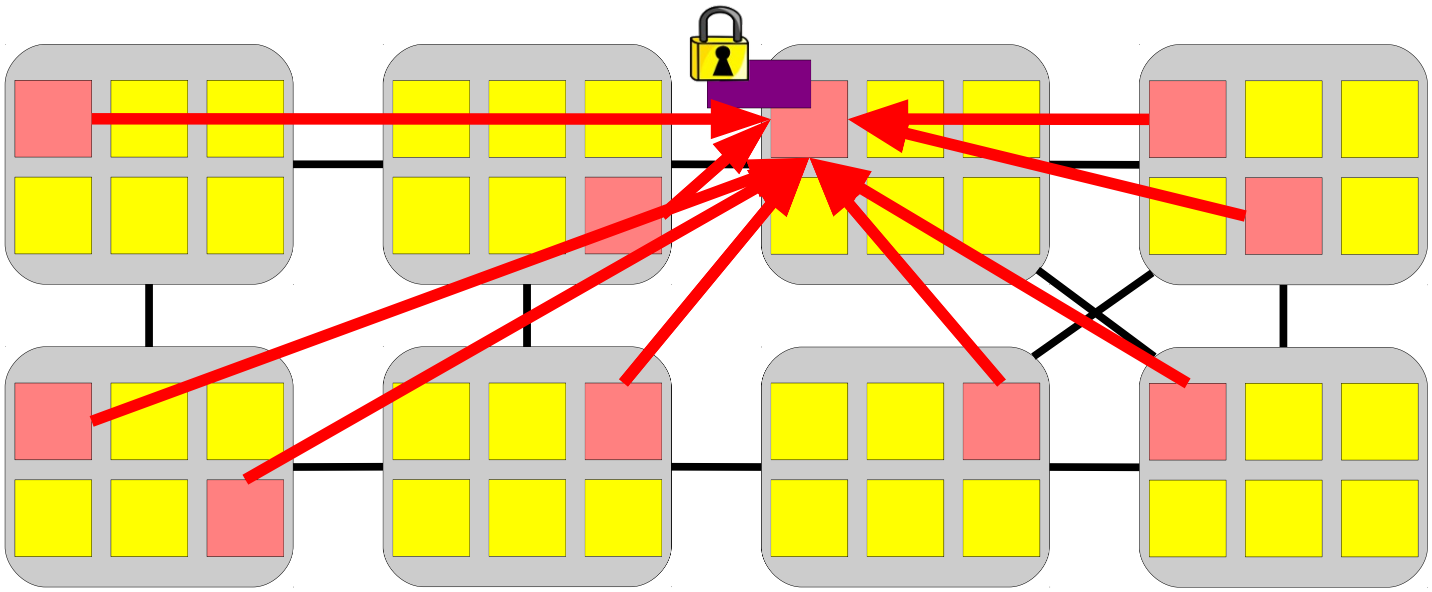
Hardware cache line lock



Reading the reference count delays memory operations from other cores

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

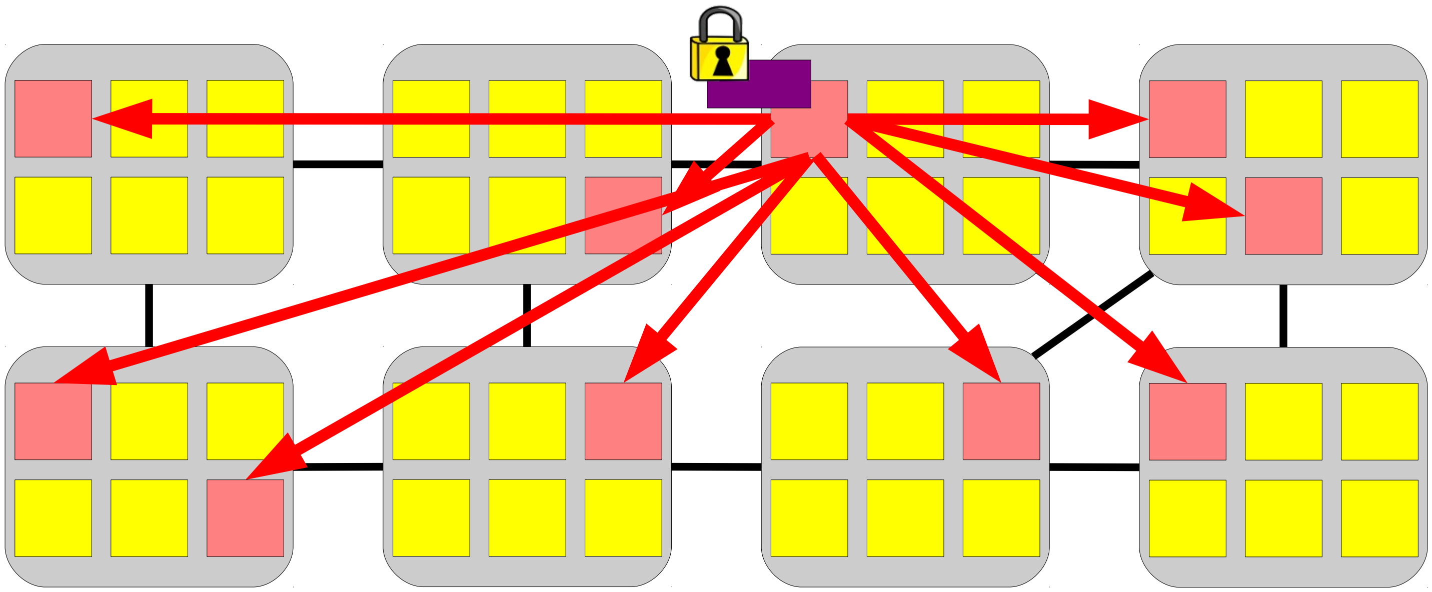
```
struct dentry {
    …
    int ref;
    …
};
```



Reading the reference count delays memory operations from other cores

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

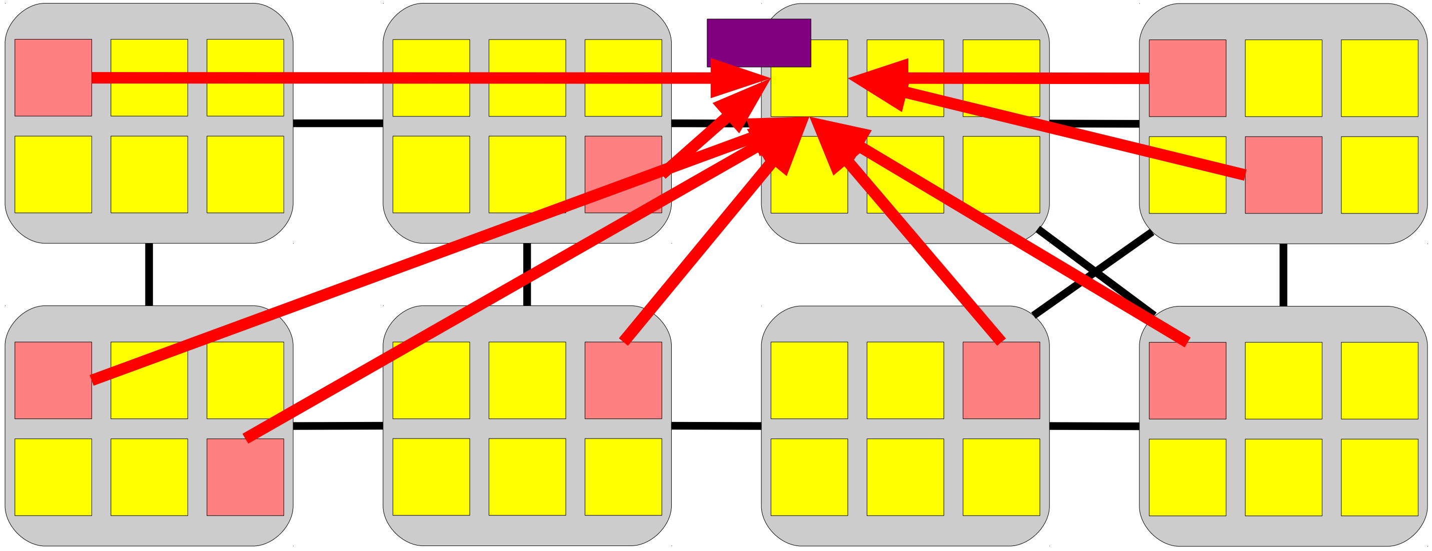
```
struct dentry {
    …
    int ref;
    …
};
```



Reading the reference count delays memory operations from other cores

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

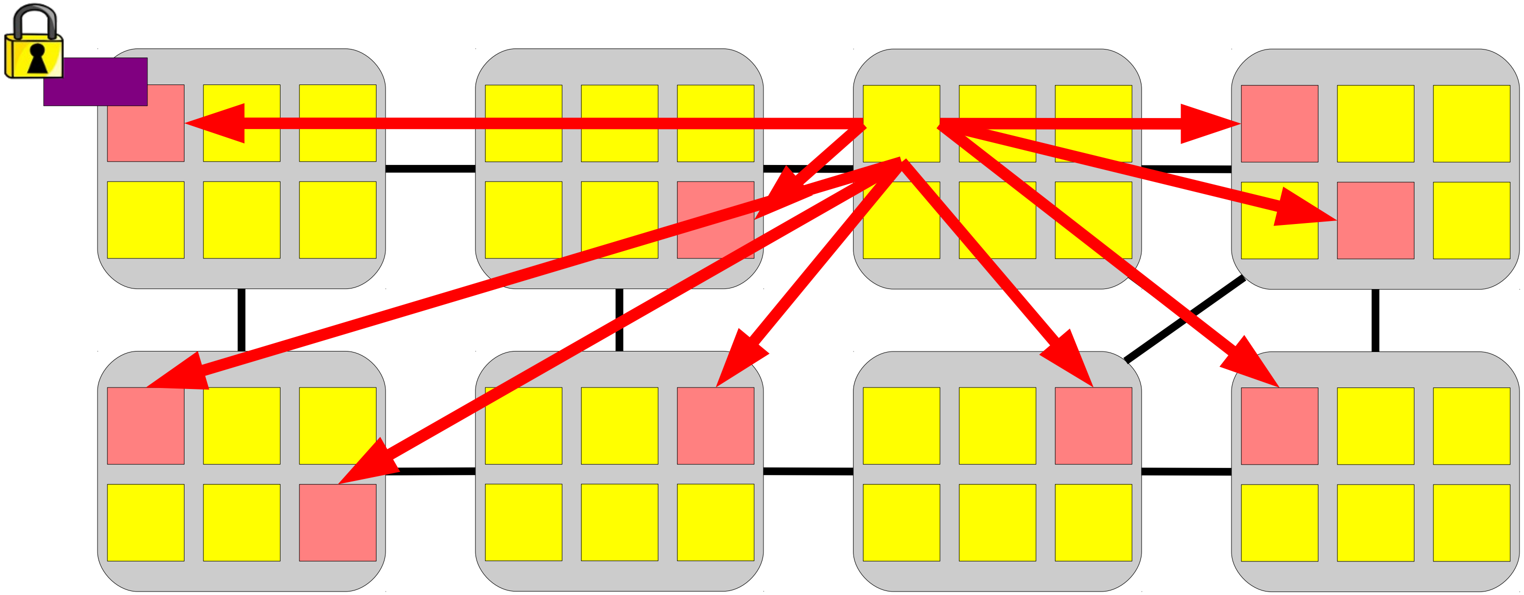
```
struct dentry {
    …
    int ref;
    …
};
```



Reading the reference count delays memory operations from other cores

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

```
struct dentry {
    …
    int ref;
    …
};
```

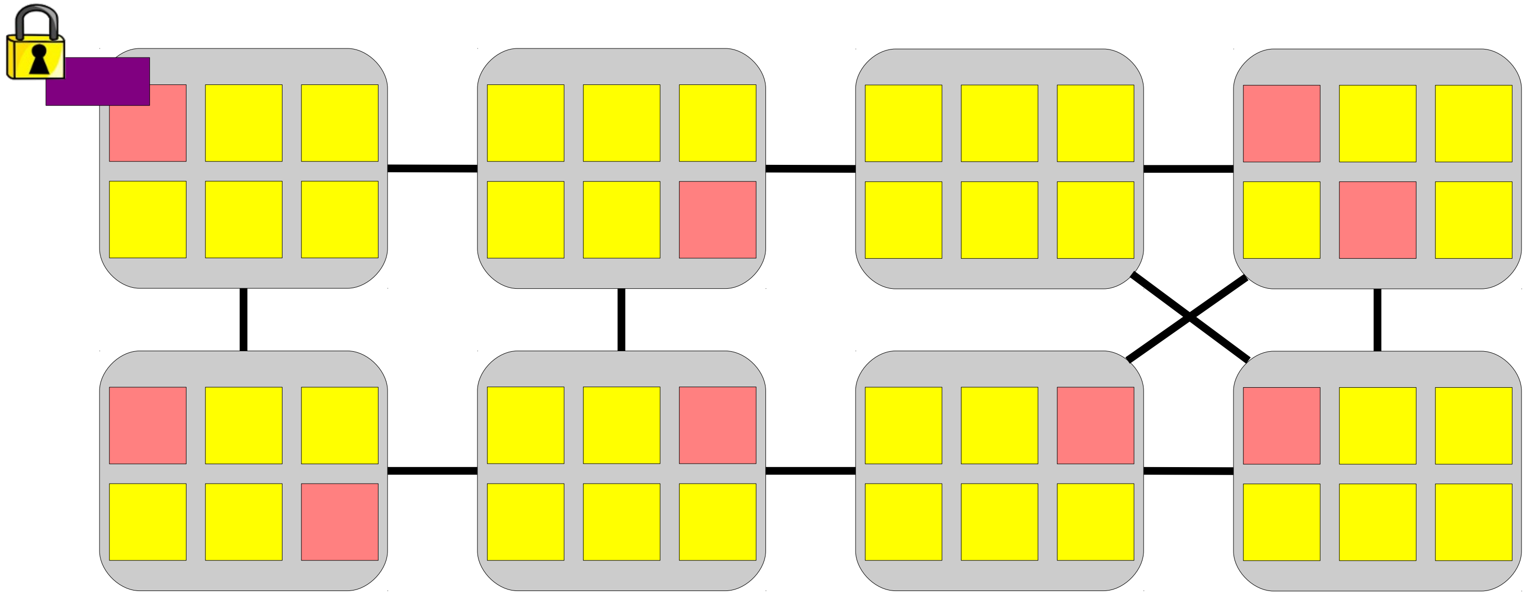


- Contention on a reference count congests the interconnect

Reading the reference count delays memory operations from other cores

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

```
struct dentry {
    …
    int ref;
    …
};
```

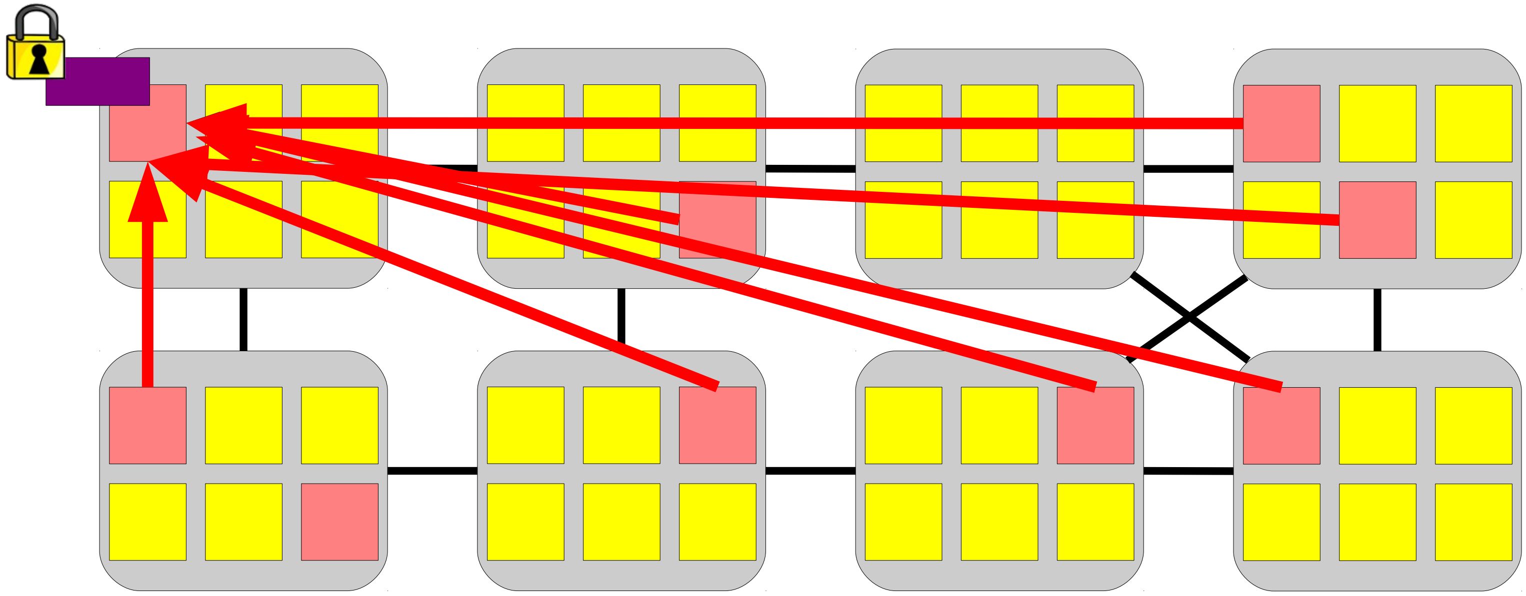


- Contention on a reference count congests the interconnect

Reading the reference count delays memory operations from other cores

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

```
struct dentry {
    …
    int ref;
    …
};
```



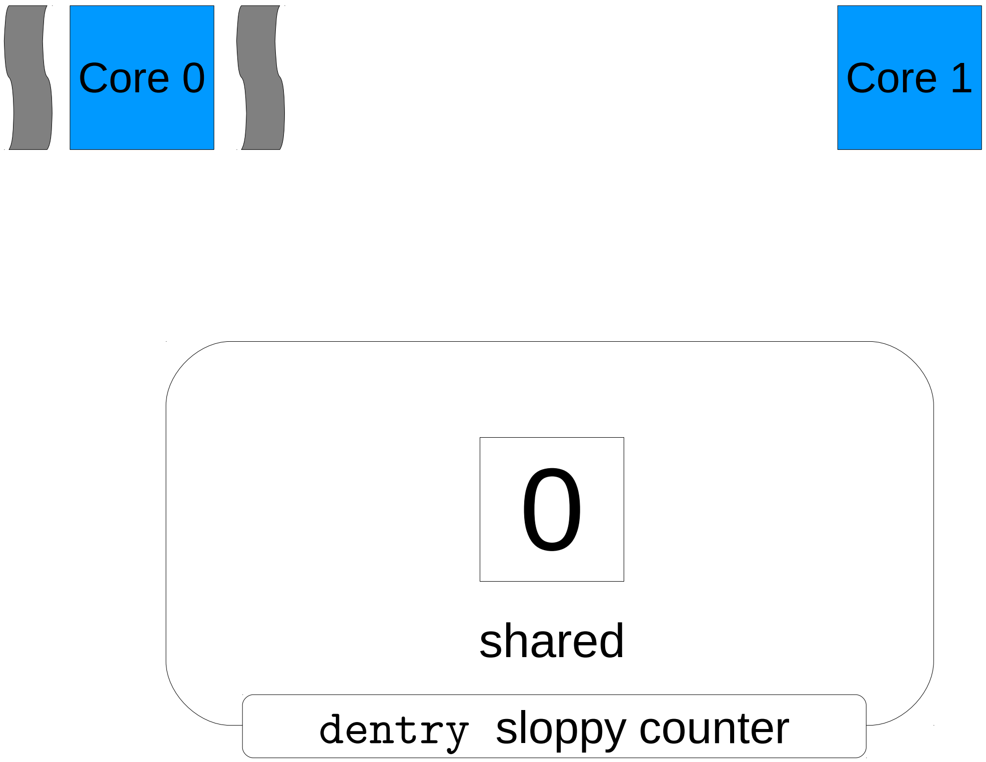
- Contention on a reference count congests the interconnect

Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references

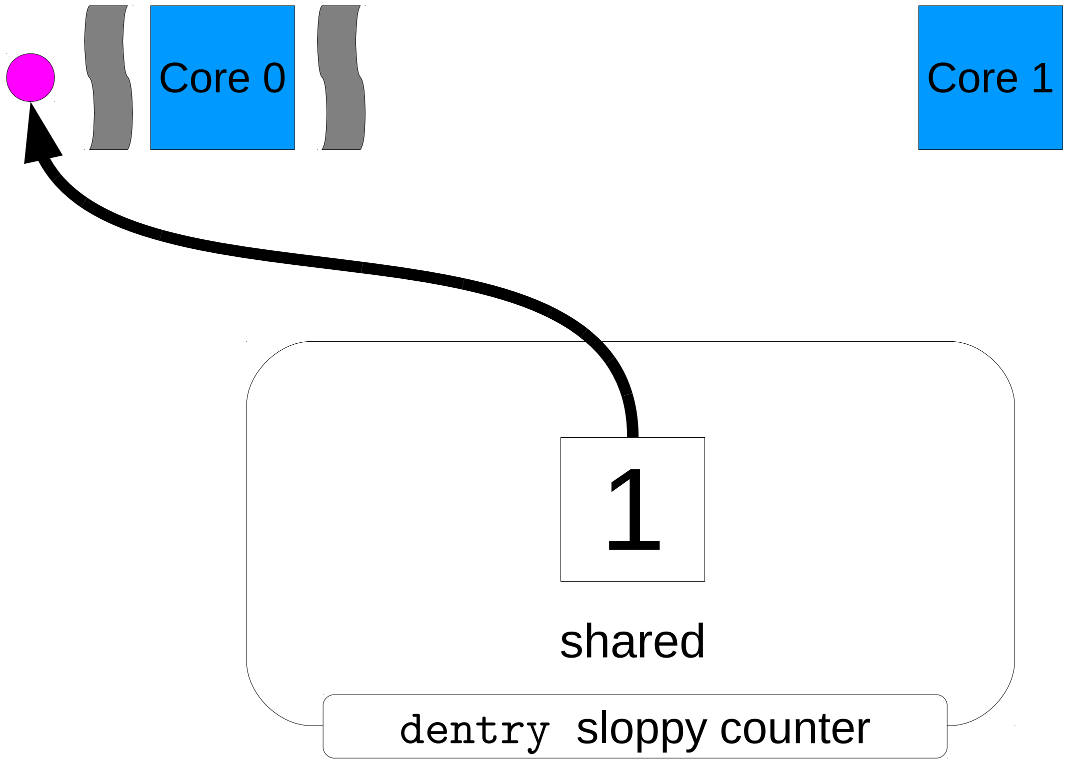
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



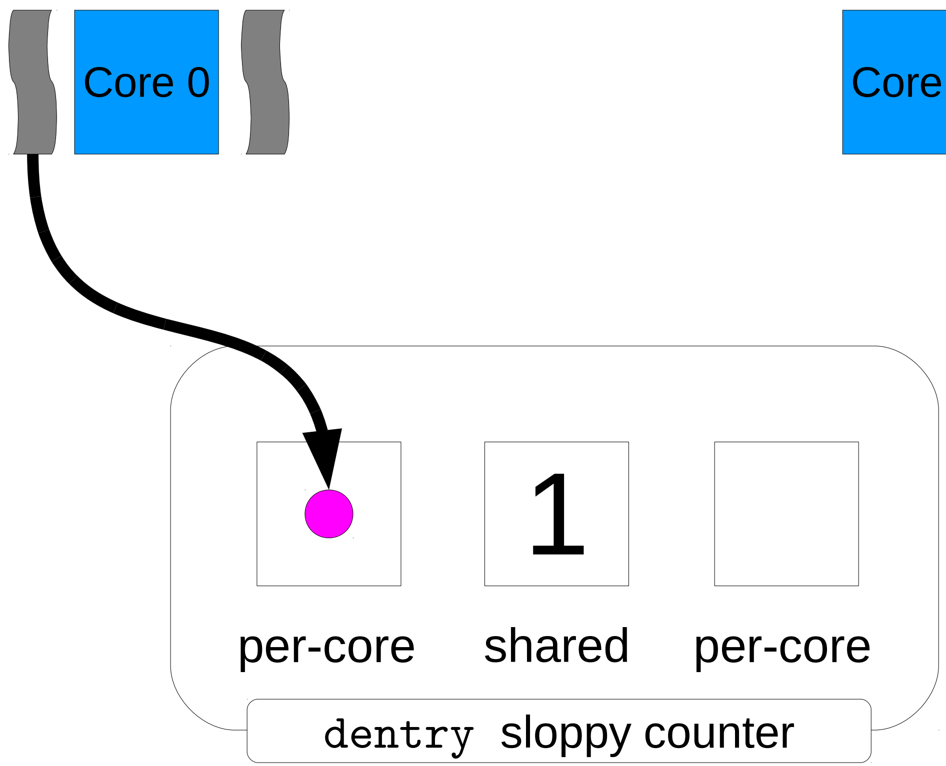
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



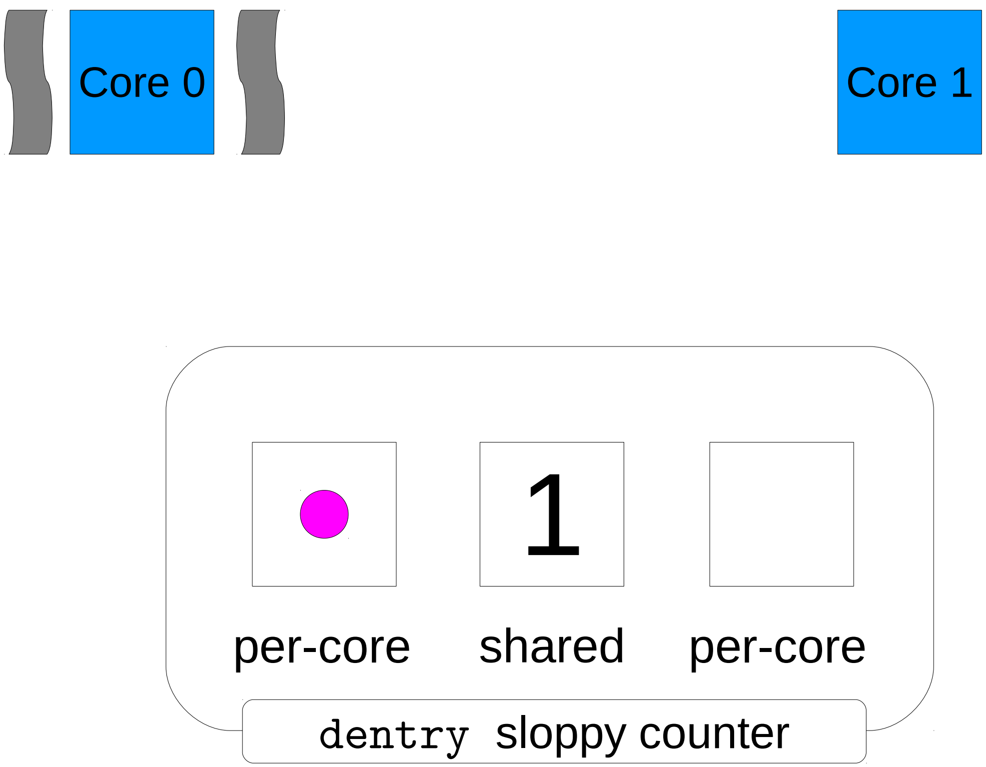
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



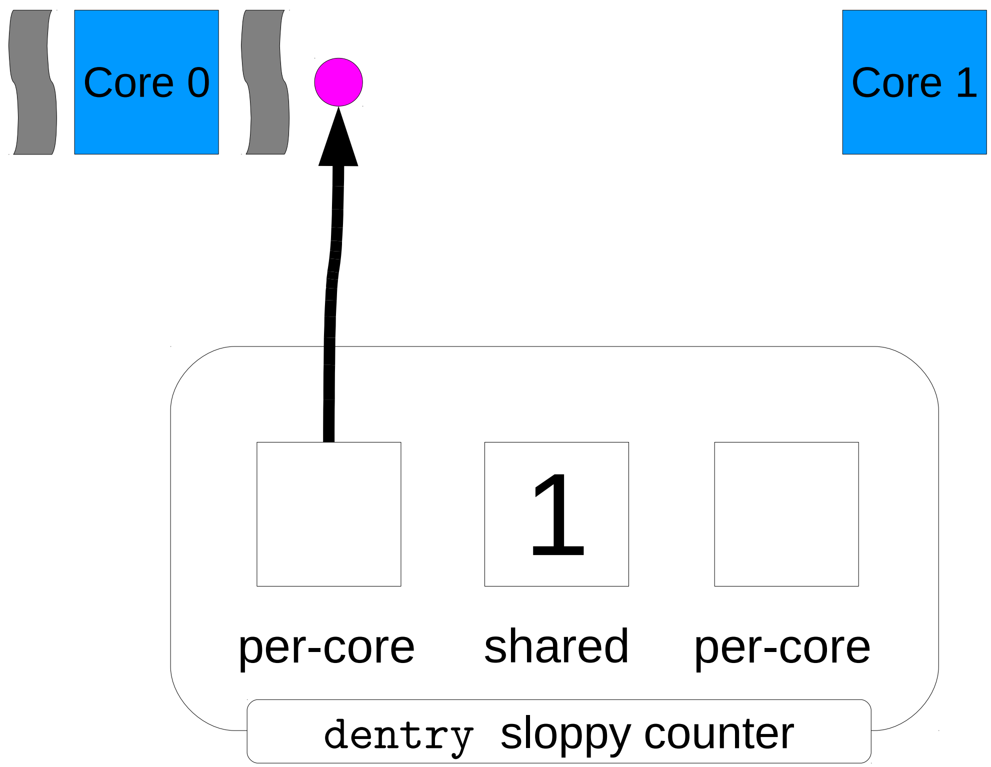
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



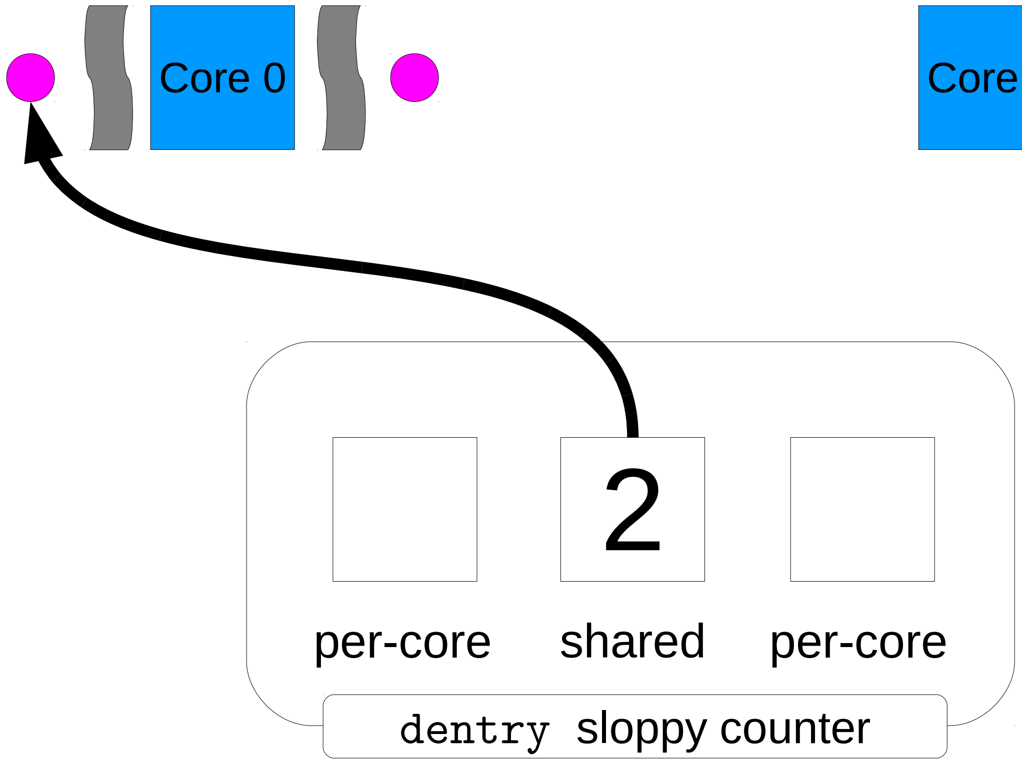
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



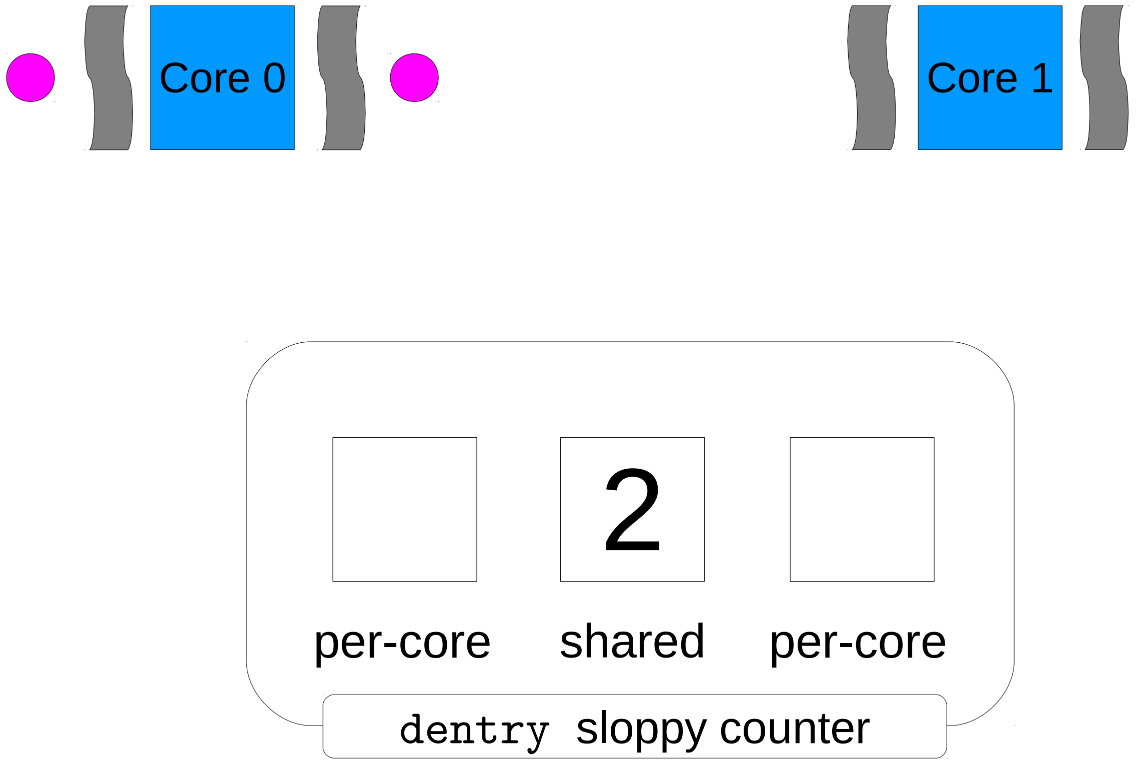
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



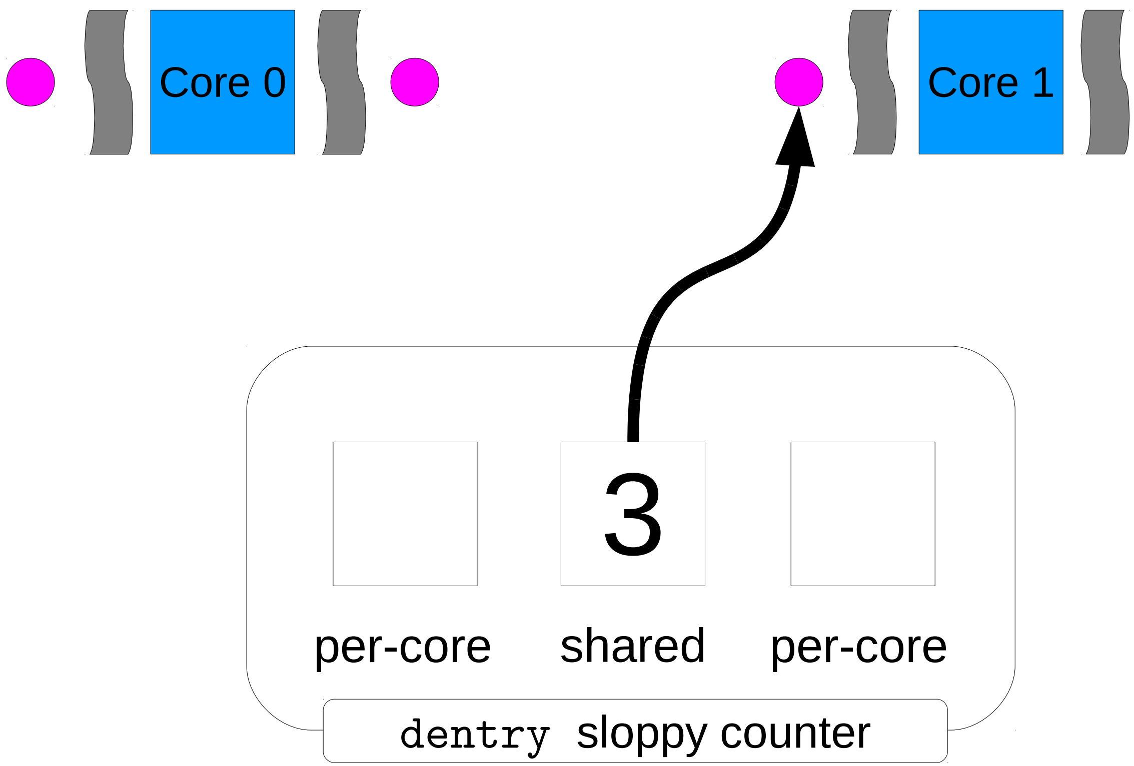
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



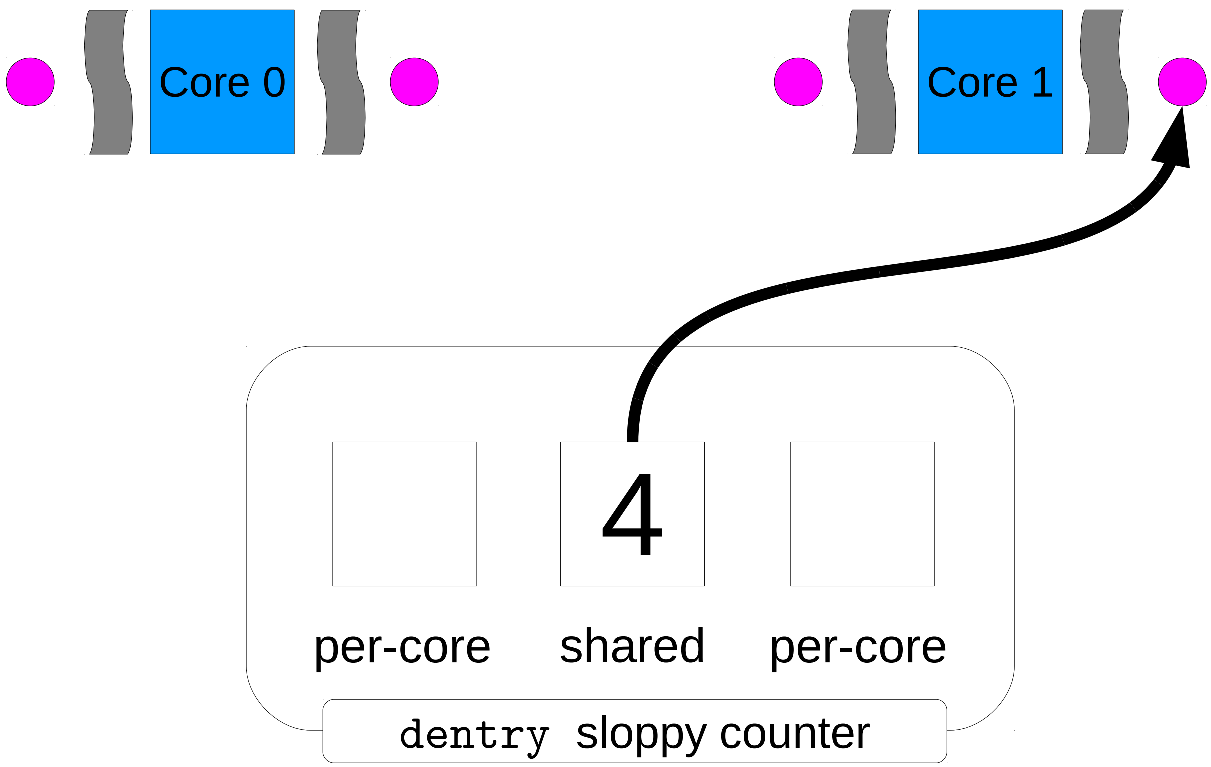
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few "spare" references



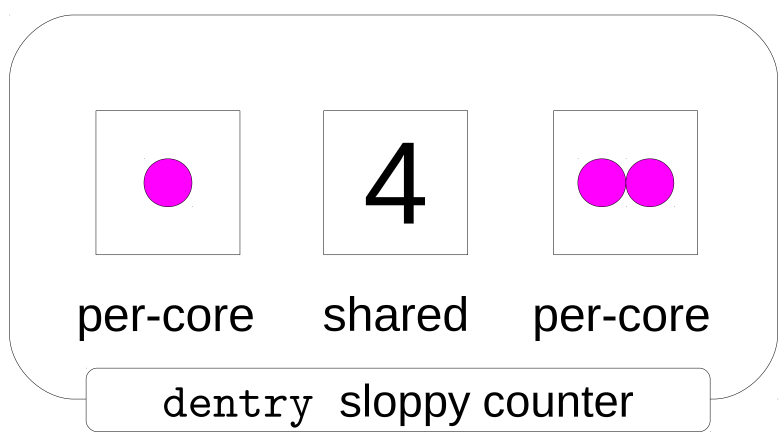
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few "spare" references



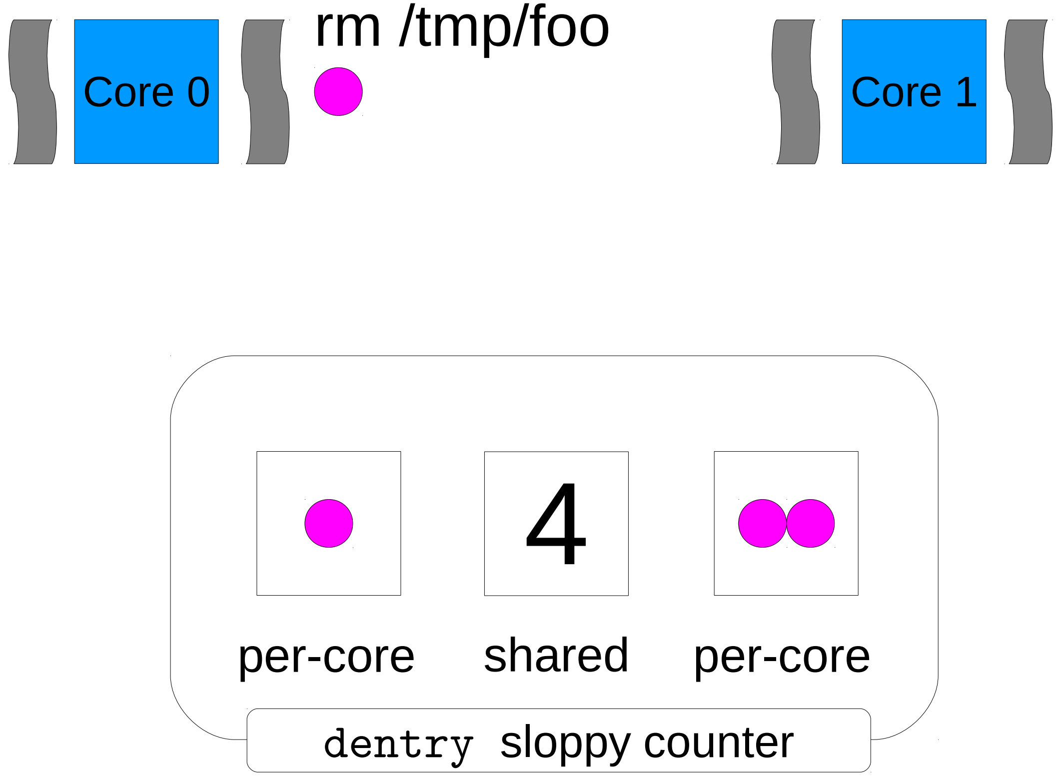
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



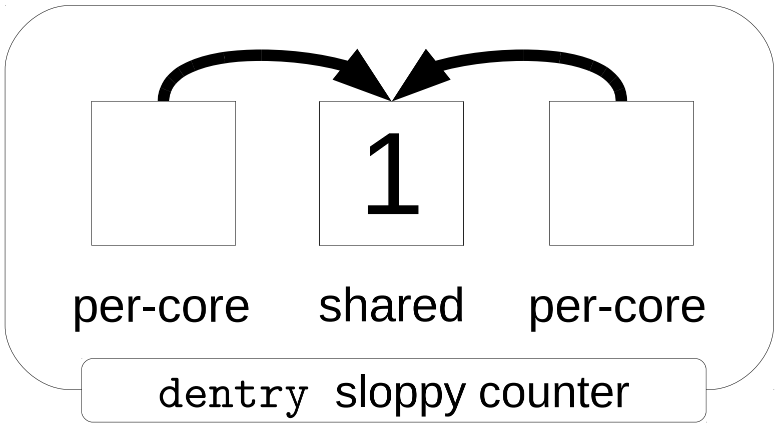
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



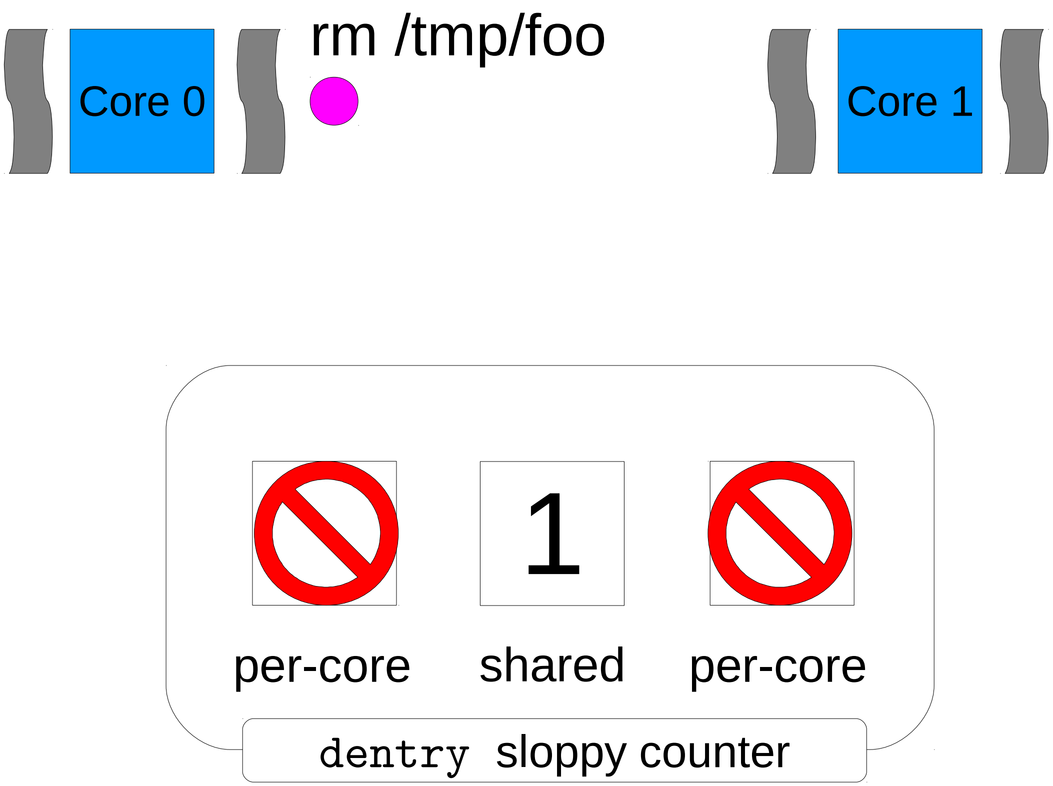
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



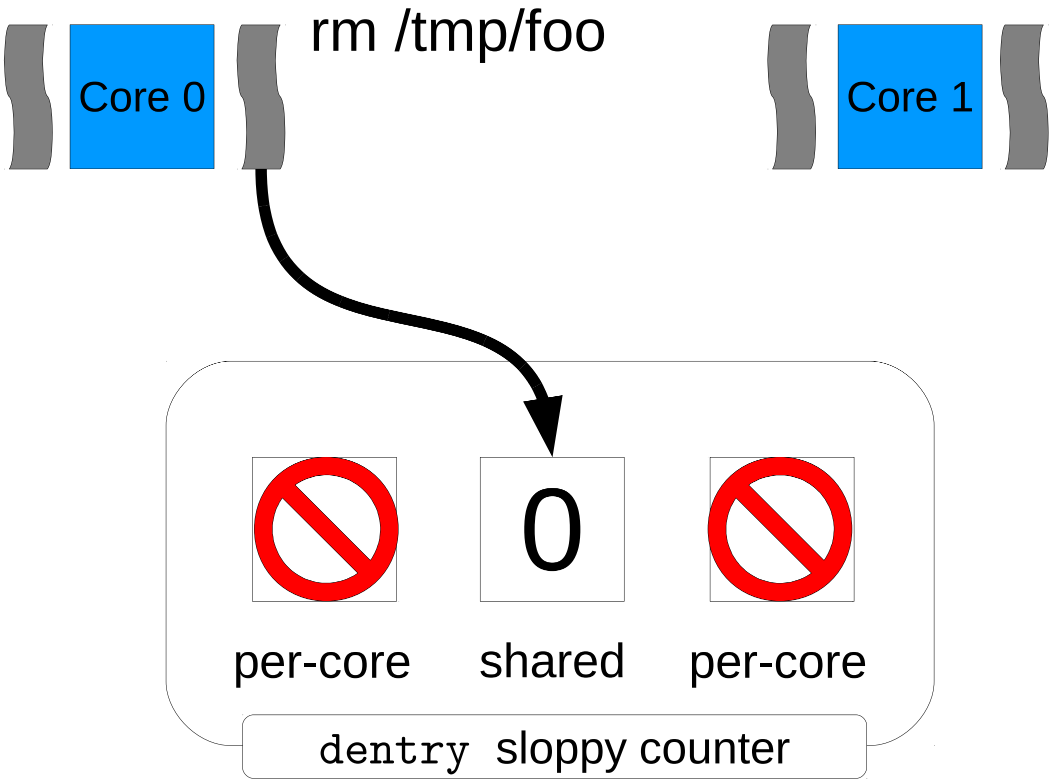
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
  - Each core holds a few "spare" references



Solution: sloppy counters

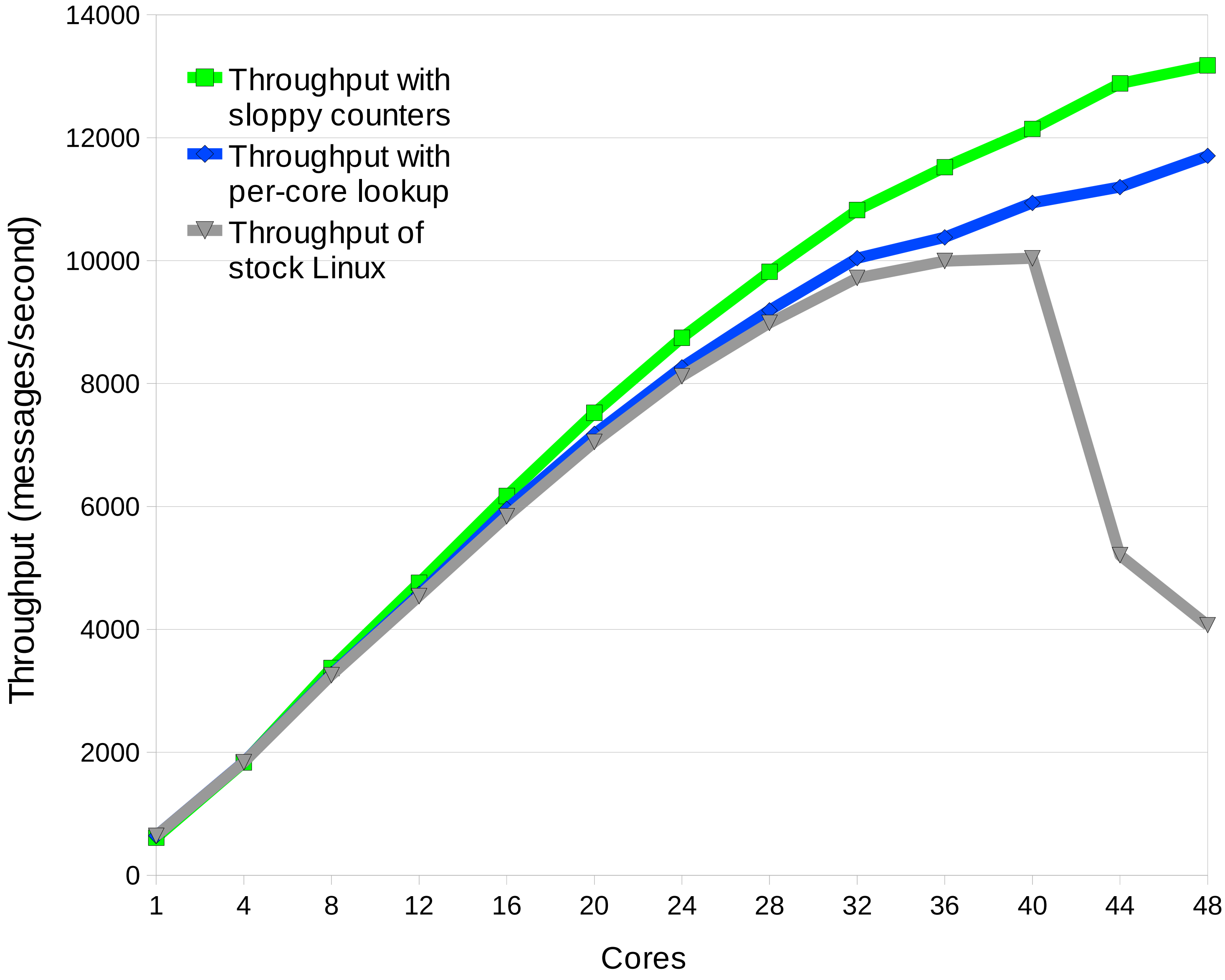
- Observation: kernel rarely needs true value of ref count
  - Each core holds a few “spare” references



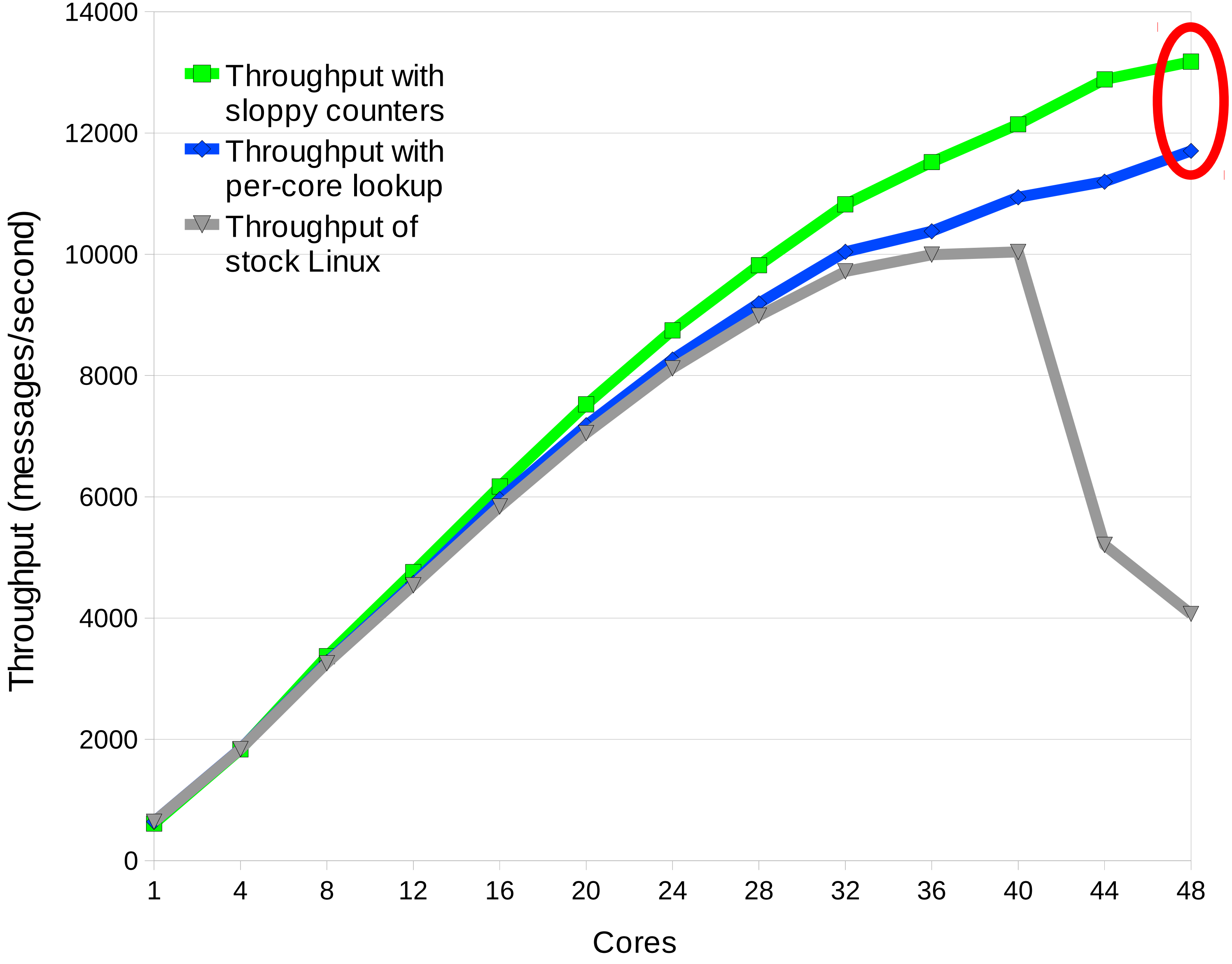
Properties of sloppy counters

- Simple to start using:
  - Change data structure
  - `atomic_inc → sloppy_inc`
- Scale well: no cache misses in common case
- Memory usage: O(N) space
- Related to: SNZI [Ellen 07] and distributed counters [Appavoo 07]

Sloppy counters: more scalability



Sloppy counters: more scalability



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

Handful of known techniques [Cantrill 08]

- Lock-free algorithms
- Per-core data structures
- Fine-grained locking
- Cache-alignment
- Sloppy counters

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

Current bottlenecks

Application	Bottleneck
memcached	HW: transmit queues on NIC
Apache	HW: receive queues on NIC
Exim	App: contention on spool directories
gmake	App: serial stages and stragglers
PostgreSQL	App: spin lock
Psearchy	HW: cache capacity
Metis	HW: DRAM throughput

- Kernel code is not the bottleneck
- Further kernel changes might help apps. or hw

Limitations

- Results limited to 48 cores and small set of applications
- Looming problems
  - fork/virtual memory book-keeping
  - Page allocator
  - File system
  - Concurrent modifications to address space
- In-memory FS instead of disk
- 48-core AMD machine ≠ single 48-core chip

Related work

- Linux and Solaris scalability studies [Yan 09,10] [Veal 07] [Tseng 07] [Jia 08] ...
- Scalable multiprocessor Unix variants
  - Flash, IBM, SGI, Sun, …
  - 100s of CPUs
- Linux scalability improvements
  - RCU, NUMA awareness, …
- Our contribution:
  - In-depth analysis of kernel intensive applications

Conclusion

- Linux has scalability problems
- They are easy to fix or avoid up to 48 cores

`http://pdos.csail.mit.edu/mosbench`