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.

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

Poor scaling on stock Linux kernel

Exim on stock Linux: collapse

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

- Cache-coherent and non-uniform access
- An approximation of a future 48-core chip
Exim on stock Linux: collapse

Throughput

Cores

Exim on stock Linux: collapse

Throughput

Cores

Oprofile shows an obvious problem

samples | % | app name | symbol name

2616 | 7.3522 | vmlinux | radix_tree_lookup_slot
2230 | 6.5452 | vmlinux | unmap_vma
2197 | 6.1746 | vmlinux | filemap_fault
1480 | 4.1830 | vmlinux | __do_fault
1348 | 3.7883 | vmlinux | copy_page_c
1182 | 3.0232 | vmlinux | unlock_page
986 | 2.7149 | vmlinux | page_fault

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:

```c
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;
}
```

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?

Ticket Lock

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

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++;
}
```

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

```c
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;
}

Previous lock holder notifies next lock holder after sending out \( \frac{N}{2} \) replies

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