Common Multiprocessor Spin Lock

```
void mp_spinlock (volatile lock t *1) {
    cli(); // prevent preemption
    while (test and set(l)) ; // lock
}
void mp unlock (volatile lock t *1) {
    *1 = 0;
    sti();
}
```

Only good for short critical sections

Does not scale for large number of processors

Relies on bus-arbitrator for fairness

```
Not appropriate for user-level
```

Used in practice in small SMP systems



Need a more systematic analysis

Thomas Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol 1, No. 1, 1990



Compares Simple Spinlocks

```
Test and Set
```

```
void lock (volatile lock_t *1) {
   while (test_and_set(l)) ;
}
```

Test and Test and Set

```
void lock (volatile lock_t *1) {
   while (*1 == BUSY || test_and_set(1)) ;
}
```



test_and_test_and_set LOCK

Avoid bus traffic contention caused by test_and_set until it is likely to succeed

Normal read spins in cache

Can starve in pathological cases



Benchmark

```
for i = 1 .. 1,000,000 {
    lock(l)
    crit_section()
    unlock()
    compute()
}
```

Compute chosen from uniform random distribution of mean 5 times critical section

Measure elapsed time on Sequent Symmetry (20 CPU 30386, coherent write-back invalidate caches)







Results

Test and set performs poorly once there is enough CPUs to cause contention for lock

• Expected

Test and Test and Set performs better

- Performance less than expected
- Still significant contention on lock when CPUs notice release and all attempt acquisition

Critical section performance degenerates

- Critical section requires bus traffic to modify shared structure
- Lock holder competes with CPU that missed as they test and set
 - lock holder is slower
- Slower lock holder results in more contention



Idea

Can inserting delays reduce bus traffic and improve performance

Explore 2 dimensions

- Location of delay
 - Insert a delay after release prior to attempting acquire
 - Insert a delay after each memory reference
- Delay is static or dynamic
 - Static assign delay "slots" to processors
 - » Issue: delay tuned for expected contention level
 - Dynamic use a back-off scheme to estimate contention
 - » Similar to ethernet
 - » Degrades to static case in worst case.



Examining Inserting Delays

TABLE III DELAY AFTER SPINNER NOTICES RELEASED LOCK

Lock	<pre>while (lock = BUSY or TestAndSet (Lock) = BUSY) begin while (lock = BUSY); Delay (); end;</pre>
	TABLE IV
	DELAY BETWEEN EACH REFERENCE
Lock	while (lock = BUSY or TestAndSet (lock) = BUSY) Delay ();



Queue Based Locking

Each processor inserts itself into a waiting queue

- It waits for the lock to free by spinning on its own separate cache line
- Lock holder frees the lock by "freeing" the next processors cache line.



Results



Results

Static backoff has higher overhead when backoff is inappropriate

- Dynamic backoff has higher overheads when static delay is appropriate
- as collisions are still required to tune the backoff time
- Queue is better when contention occurs, but has higher overhead when it does not.
- Issue: Preemption of queued CPU blocks rest of queue (worse than simple spin locks)



John Mellor-Crummey and Michael Scott, "Algorithms for Scalable Synchronisation on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems,* Vol. 9, No. 1, 1991



MCS Locks

Each CPU enqueues its own private lock variable into a queue and spins on it

No contention

On lock release, the releaser unlocks the next lock in the queue

- Only have bus contention on actual unlock
- No livelock (order of lock acquisitions defined by the list)



MCS Lock

Requires

- compare_and_swap()
- exchange()
 - Also called fetch_and_store()



```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode
// parameter I, below, points to a quode record allocated
// (in an enclosing scope) in shared memory locally-accessible
// to the invoking processor
procedure acquire_lock (L : `lock, I : `qnode)
    I->next := nil
    predecessor : `qnode := fetch_and_store (L, I)
    if predecessor != nil // queue was non-empty
        I->locked := true
       predecessor->next := I
       repeat while I->locked
                                          // spin
procedure release_lock (L : `lock, I: `qnode)
    if I->next = nil
                           // no known successor
        if compare_and_swap (L, I, nil)
            return
           // compare_and_swap returns true iff it swapped
       repeat while I->next = nil
                                           // spin
   I->nert->locked := false
```





Sample MCS code for ARM MPCore

```
void mcs_acquire(mcs_lock *L, mcs_qnode_ptr I)
```

{

}

```
I->next = NULL;
MEM_BARRIER;
mcs qnode ptr pred = (mcs qnode*) SWAP PTR( L, (void *)I);
if (pred == NULL)
             /* lock was free */
{
    MEM_BARRIER;
     return;
}
I->waiting = 1; // word on which to spin
MEM BARRIER;
pred->next = I; // make pred point to me
```



Selected Benchmark

Compared

- test and test and set
- Anderson's array based queue
- test and set with exponential back-off
- MCS





Fig. 17. Performance of spin locks on the Symmetry (empty critical section).



Confirmed Trade-off

Queue locks scale well but have higher overhead Spin Locks have low overhead but don't scale well What do we use?



Beng-Hong Lim and Anant Agarwal, "Reactive Synchronization Algorithms for Multiprocessors", *ASPLOS VI*, 1994











Idea

Can we dynamically switch locking methods to suit the current contention level???



Issues

How do we determine which protocol to use?

• Must not add significant cost

How do we correctly and efficiently switch protocols? How do we determine when to switch protocols?



Protocol Selection

Keep a "hint"

Ensure both TTS and MCS lock a never free at the same time

- Only correct selection will get the lock
- Choosing the wrong lock with result in retry which can get it right next time
- Assumption: Lock mode changes infrequently
 - hint cached read-only
 - infrequent protocol mismatch retries



Changing Protocol

Only lock holder can switch to avoid race conditions

• It chooses which lock to free, TTS or MCS.



When to change protocol

Use threshold scheme

- Repeated acquisition failures will switch mode to queue
- Repeated immediate acquisition will switch mode to TTS



Results





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.

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. *In the Proceedings of the Linux Symposium, Ottawa, Canada, July 2012.*



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: <u>Short critical sections can lead to</u> 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

	samples	%	app name	symbol name
	2616	7.3522	vmlinux	radix_tree_lookup_slot
	2329	6.5456	vmlinux	unmap_vmas
40 cores:	2197	6.1746	vmlinux	filemap_fault
10000 msg/sec	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
	samples	%	app name	symbol name
	samples 13515	% 34.8657	app name vmlinux	symbol name lookup_mnt
49 00100:	samples 13515 2002	% 34.8657 5.1647	app name vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot
48 cores: 4000 msa/sec	samples 13515 2002 1661	% 34.8657 5.1647 4.2850	app name vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault
48 cores: 4000 msg/sec	samples 13515 2002 1661 1497	% 34.8657 5.1647 4.2850 3.8619	app name vmlinux vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault unmap_vmas
48 cores: 4000 msg/sec	samples 13515 2002 1661 1497 1026	% 34.8657 5.1647 4.2850 3.8619 2.6469	app name vmlinux vmlinux vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault unmap_vmas do_fault
48 cores: 4000 msg/sec	samples 13515 2002 1661 1497 1026 914	% 34.8657 5.1647 4.2850 3.8619 2.6469 2.3579	app name vmlinux vmlinux vmlinux vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault unmap_vmas do_fault atomic_dec

Oprofile shows an obvious problem

	samples	%	app name	symbol name
	2616	7.3522	vmlinux	radix_tree_lookup_slot
	2329	6.5456	vmlinux	unmap_vmas
40 cores: 10000 msg/sec	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
			2	
			177	
	samples	%	app name	symbol name
	samples 13515	% 34.8657	app name vmlinux	symbol name lookup_mnt
19 coros:	samples 13515 2002	% 34.8657 5.1647	app name vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot
48 cores: 4000 msg/sec	samples 13515 2002 1661	% 34.8657 5.1647 4.2850	app name vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault
48 cores: 4000 msg/sec	samples 13515 2002 1661 1497	% 34.8657 5.1647 4.2850 3.8619	app name vmlinux vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault unmap_vmas
48 cores: 4000 msg/sec	samples 13515 2002 1661 1497 1026	% 34.8657 5.1647 4.2850 3.8619 2.6469	app name vmlinux vmlinux vmlinux vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault unmap_vmas do_fault
48 cores: 4000 msg/sec	samples 13515 2002 1661 1497 1026 914	% 34.8657 5.1647 4.2850 3.8619 2.6469 2.3579	app name vmlinux vmlinux vmlinux vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault unmap_vmas do_fault atomic_dec
48 cores: 4000 msg/sec	samples 13515 2002 1661 1497 1026 914 896	% 34.8657 5.1647 4.2850 3.8619 2.6469 2.3579 2.3115	app name vmlinux vmlinux vmlinux vmlinux vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault unmap_vmas do_fault atomic_dec unlock_page

Oprofile shows an obvious problem

	samples	%	app name	symbol name
	2616	7.3522	vmlinux	radix_tree_lookup_slot
	2329	6.5456	vmlinux	unmap_vmas
40 cores:	2197	6.1746	vmlinux	filemap_fault
10000 msg/sec	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
	samples	%	app name	symbol name
	samples 13515	% 34.8657	app name vmlinux	symbol name lookup_mnt
18 coros:	samples 13515 2002	% 34.8657 5.1647	app name vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot
48 cores: 4000 msg/sec	samples 13515 2002 1661	% 34.8657 5.1647 4.2850	app name vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault
48 cores: 4000 msg/sec	samples 13515 2002 1661 1497	% 34.8657 5.1647 4.2850 3.8619	app name vmlinux vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault unmap_vmas
48 cores: 4000 msg/sec	samples 13515 2002 1661 1497 1026	% 34.8657 5.1647 4.2850 3.8619 2.6469	app name vmlinux vmlinux vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault unmap_vmas do_fault
48 cores: 4000 msg/sec	samples 13515 2002 1661 1497 1026 914	% 34.8657 5.1647 4.2850 3.8619 2.6469 2.3579	app name vmlinux vmlinux vmlinux vmlinux vmlinux vmlinux	symbol name lookup_mnt radix_tree_lookup_slot filemap_fault unmap_vmas do_fault atomic_dec

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)



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?

































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