Need a more systematic analysis

Compares Simple Spinlocks

Test and Set

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

Test and Test and Set

```c
void lock (volatile lock_t *l) {
    while (*l == BUSY || test_and_set(l)) ;
}
```
test_and_test_and_set LOCK

Avoids bus traffic contention by delaying test_and_set until it might succeed

Normal read (‘test’) spins on local cache line

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 observing 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 early ethernet
    » Degrades to static case in worst case.
Examining Inserting Delays

<table>
<thead>
<tr>
<th>TABLE III</th>
<th>DELAY AFTER SPINNER NOTICES RELEASED LOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>while (lock = BUSY or TestAndSet (Lock) = BUSY) begin</td>
</tr>
<tr>
<td></td>
<td>while (lock = BUSY) ;</td>
</tr>
<tr>
<td></td>
<td>Delay ();</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE IV</th>
<th>DELAY BETWEEN EACH REFERENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>while (lock = BUSY or TestAndSet (lock = BUSY)</td>
</tr>
<tr>
<td></td>
<td>Delay ();</td>
</tr>
</tbody>
</table>
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

![Graph showing overhead (sec.) vs. number of processors for different synchronization methods: spin on read, static release, backoff rel., static ref., backoff ref., and queue. The graph illustrates the overhead in seconds for varying processor counts.]
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)
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
end type

// parameter I, below, points to a qnode 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->next->locked := false
Sample MCS code for ARM MPCore

```c
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?
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
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

![Graph showing throughput as a function of cores]

- Throughput (messages/second) vs. Cores
- The graph illustrates the throughput for different numbers of cores.
- The throughput increases as the number of cores increases, reaching a peak at around 36 cores before dropping sharply at 40 cores.
Exim on stock Linux: collapse

Throughput vs Cores graph showing a drop in throughput at 44 cores.
Exim on stock Linux: collapse
Oprofile shows an obvious problem

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2616</td>
<td>7.3522</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>2329</td>
<td>6.5456</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>2197</td>
<td>6.1746</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1488</td>
<td>4.1820</td>
<td>vmlinux</td>
<td>__do_fault</td>
</tr>
<tr>
<td>1348</td>
<td>3.7885</td>
<td>vmlinux</td>
<td>copy_page_c</td>
</tr>
<tr>
<td>1182</td>
<td>3.3220</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
<tr>
<td>966</td>
<td>2.7149</td>
<td>vmlinux</td>
<td>page_fault</td>
</tr>
</tbody>
</table>

40 cores:
10000 msg/sec

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>13515</td>
<td>34.8657</td>
<td>vmlinux</td>
<td>lookup_mnt</td>
</tr>
<tr>
<td>2002</td>
<td>5.1647</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>1661</td>
<td>4.2850</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1497</td>
<td>3.8619</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>1026</td>
<td>2.6469</td>
<td>vmlinux</td>
<td>__do_fault</td>
</tr>
<tr>
<td>914</td>
<td>2.3579</td>
<td>vmlinux</td>
<td>atomic_dec</td>
</tr>
<tr>
<td>896</td>
<td>2.3115</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
</tbody>
</table>

48 cores:
4000 msg/sec
Oprofile shows an obvious problem

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2616</td>
<td>7.3522</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>2329</td>
<td>6.5456</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>2197</td>
<td>6.1746</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1488</td>
<td>4.1820</td>
<td>vmlinux</td>
<td>__do_fault</td>
</tr>
<tr>
<td>1348</td>
<td>3.7885</td>
<td>vmlinux</td>
<td>copy_page_c</td>
</tr>
<tr>
<td>1182</td>
<td>3.3220</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
<tr>
<td>966</td>
<td>2.7149</td>
<td>vmlinux</td>
<td>page_fault</td>
</tr>
</tbody>
</table>

40 cores: 10000 msg/sec

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>13515</td>
<td>34.8657</td>
<td>vmlinux</td>
<td>lookup_mnt</td>
</tr>
<tr>
<td>2002</td>
<td>5.1647</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>1661</td>
<td>4.2850</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1497</td>
<td>3.8619</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>1026</td>
<td>2.6469</td>
<td>vmlinux</td>
<td>__do_fault</td>
</tr>
<tr>
<td>914</td>
<td>2.3579</td>
<td>vmlinux</td>
<td>atomic_dec</td>
</tr>
<tr>
<td>896</td>
<td>2.3115</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
</tbody>
</table>

48 cores: 4000 msg/sec
Oprofile shows an obvious problem

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2616</td>
<td>7.3522</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>2329</td>
<td>6.5456</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>2197</td>
<td>6.1746</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1488</td>
<td>4.1820</td>
<td>vmlinux</td>
<td>__do_fault</td>
</tr>
<tr>
<td>1348</td>
<td>3.7885</td>
<td>vmlinux</td>
<td>copy_page_c</td>
</tr>
<tr>
<td>1182</td>
<td>3.3220</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
<tr>
<td>966</td>
<td>2.7149</td>
<td>vmlinux</td>
<td>page_fault</td>
</tr>
</tbody>
</table>

40 cores: 10000 msg/sec

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>13515</td>
<td>34.8657</td>
<td>vmlinux</td>
<td>lookup_mnt</td>
</tr>
<tr>
<td>2002</td>
<td>5.1647</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>1661</td>
<td>4.2850</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1497</td>
<td>3.8619</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>1026</td>
<td>2.6469</td>
<td>vmlinux</td>
<td>__do_fault</td>
</tr>
<tr>
<td>914</td>
<td>2.3579</td>
<td>vmlinux</td>
<td>atomic_dec</td>
</tr>
<tr>
<td>896</td>
<td>2.3115</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
</tbody>
</table>

48 cores: 4000 msg/sec
Bottleneck: reading mount table

- Delivering an email calls `sys_open`
- `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;
}
```
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;
}
```
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;
}
```

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]

```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]

```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]

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

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

```c
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]

```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]

```c
void spin_lock(spinlock_t *lock) {
    int 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;
}
```

![Diagram illustrating scalability collapse due to non-scalable locks](attachment:image.png)

500 cycles
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]

```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]

```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;
}
```

Previous lock holder notifies next lock holder after sending out 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
Transactional memory to manage concurrency
The problem – concurrency

CPU 1
a = a + 1

CPU 2
a = a - 1

CPU 3
a = a + 2

Time
The solution: mutual exclusion

CPU 1: $a = a + 1$
CPU 2: $a = a - 1$
CPU 3: $a = a + 2$
Synchronisation granularity

Fine-grained / lock-free

Coarse-grained

Good scalability

Complexity

Verification

tractability

Legacy proof/ code base

Legacy proof/ code base
Course-grained mutual exclusion

CPU 1
a=a+1

CPU 2
b=b+1

CPU 3
c=c+1

Critical sections serialised unnecessarily
Optimistic concurrency

Concurrent execution correct if no conflicting accesses
Transactional Memory

• A transaction is a sequence of machine instructions satisfying the following properties:
  • Serializability:
    • Transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another.
    • Committed transactions are never observed by different processors in different orders.
  • Atomicity:
    • Each transaction makes a sequence of tentative changes to shared memory.
    • A transaction can commit, making its changes visible to other processors
    • Or a transaction aborts, causing its changes to be discarded.
Transactions

- Updates only visible locally
- Commit publishes update if conflict free
Transactions

CPU 1

a = a + 1

CPU 2

b = b + 1

Time
Conflict detection

Hardware maintains:

- *Read set:* The set of all memory addresses loaded from
- *Write set:* The set of all memory addresses stored to
  - The write set is not visible to other CPUs until a successful commit

A transaction is conflict free if:

- No other processor reads a location that is part of the transactional region’s write-set
- And, no other processor writes a location that is a part of the read- or write-set of the transactional region.
Implementation Intuition

- Cache coherence protocol already coordinates reads and writes to cache lines
- Write-back caches could isolate updates until successfully committed

→ Implement transactions by augmenting cache hardware
Some Papers

Herlihy, Maurice / Moss, J. Eliot B.
   Transactional Memory: Architectural Support for Lock-Free Data Structures
   1993
   Proceedings of the 20th annual international symposium on Computer architecture - ISCA '93

Yoo, Richard M. / Hughes, Christopher J. / Lai, Konrad / Rajwar, Ravi
   Performance evaluation of Intel transactional synchronization extensions for high-performance computing
   2013
   Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC 13
Some Hardware Limitations

Aborts

- Caches are a finite size, transactions will abort if they exceed cache capacity to manage read and write set
- High contention on transaction region can trigger repeated aborts
Sample Elided Lock

Elided lock:

/* Start transactional region. On abort we come back here. */
if (_xbegin() == _XBEGIN_STARTED) {
    /* Put lock into read-set and abort if lock is busy */
    if (lock variable is not free)
        _xabort(_XABORT_LOCK_BUSY);
} else {
    /* Fallback path */
    /* Come here when abort or lock not free */
    lock lock;
}
/* Execute critical region either transaction or with lock */

Elided unlock:

/* Critical region ends */
/* Was this lock elided? */
if (lock is free)
    _xend();
else
    unlock lock
Microkernel vs Linux Execution

Linux

App

Kernel

10s of ms

10s of ms

10s of ms

Microkernel

App

Server

Kernel

10s of ms

10s of ms

10s of ms

0.3µs
Experiments with seL4 and Intel TSX

Basic idea: put the kernel in a transaction
• Coarse-grained transaction
• Fallback on BKL

Microkernel small enough to fit in a transaction

Repeated non-conflicting parallel IPC benchmark
None: No concurrency control
Fine-grained scales well
• Expected
RTM also scales well
• Extremely low abort rates