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 vs cores](image)

- Throughput (messages/second)
- Cores

Throughput peaks at approximately 36 cores before dropping sharply.
Exim on stock Linux: collapse

![Graph showing throughput vs. cores](image-url)
Exim on stock Linux: collapse

![Graph showing throughput and kernel time against cores]

- Throughput (messages/second)
- Kernel time (milliseconds/message)

- Throughput and kernel time change as the number of cores increases.

Cores: 1, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48

Throughput:
- 0 messages/second at 1 core
- Increases with an increase in cores

Kernel time:
- 0 milliseconds/message at 1 core
- Shows an exponential increase after reaching a peak at 40 cores

Graph highlights the point of collapse where performance deteriorates significantly.

38
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>

<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>
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 */
}
```

```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]

```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]

<table>
<thead>
<tr>
<th>void spin_lock(spinlock_t *lock)</th>
<th>void spin_unlock(spinlock_t *lock)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>t = atomic_inc(lock-&gt;next_ticket);</td>
<td>lock-&gt;current_ticket++;</td>
</tr>
<tr>
<td>while (t != lock-&gt;current_ticket)</td>
<td></td>
</tr>
<tr>
<td>; /* Spin */</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>struct spinlock_t {</td>
</tr>
<tr>
<td></td>
<td>int current_ticket;</td>
</tr>
<tr>
<td></td>
<td>int next_ticket;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
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;
}
```

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++;
}
```

```c
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 */
}

t = atomic_inc(lock->next_ticket);
while (t != lock->current_ticket) ; /* Spin */
```

```c
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

Legacy proof/ code base

good scalability
Course-grained mutual exclusion

CPU 1

a = a + 1

CPU 2

b = b + 1

CPU 3

c = c + 1

Critical sections serialised unnecessarily

Time
Optimistic concurrency

CPU 1: \(a = a + 1\)

CPU 2: \(b = b + 1\)

CPU 3: \(c = c + 1\)

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 transactions 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

CPU 1: \( a = a + 1 \)
CPU 2: \( a = a - 1 \)

Abort

Time
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 10s of ms

Kernel 10s of ms

10s of ms

Microkernel

App 10s of ms

Server 10s of ms

Kernel 0.3µs

10s of ms
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