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, Barrelish, 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)
Throughput vs Cores for Exim on stock Linux: collapse

Throughput (messages/second)

Cores
Exim on stock Linux: collapse

Throughput vs. Cores Graph

Throughput (messages/second) vs. Cores

Throughput peaks at 40 cores and then sharply declines, indicated by the red ellipse.
Exim on stock Linux: collapse

Throughput (messages/second) vs. Kernel time (milliseconds/message) for different numbers of cores.
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>
<tr>
<td>13515</td>
<td>34.8657</td>
<td>vmlinux</td>
<td>lookup_mnt</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>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

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

Critical section is short. Why does it cause a scalability bottleneck?
Bottleneck: reading mount table

- `sys_open` eventually 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;
}
```

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

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

Allocate a ticket
Linux spin lock implementation

```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;
}
```

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 lock implementation

```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;
}
```

Spin until it's my turn
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;
}
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]

```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;
}
```
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;
}
```

500 – 4000 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.
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) {
    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;
}
```
Bottleneck: reading mount table

• `sys_open` eventually 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;
}
```

• 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

```c
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

```c
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

```c
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

```c
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

Throughput with per-core lookup

Throughput of stock Linux

Cores

Throughput (messages/second)
Per-core lookup: scalability is better
# No obvious bottlenecks

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3319</td>
<td>5.4462</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>3119</td>
<td>5.2462</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>1966</td>
<td>3.3069</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1950</td>
<td>3.2800</td>
<td>vmlinux</td>
<td>page_fault</td>
</tr>
<tr>
<td>1627</td>
<td>2.7367</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
<tr>
<td>1626</td>
<td>2.7350</td>
<td>vmlinux</td>
<td>clear_page_c</td>
</tr>
<tr>
<td>1578</td>
<td>2.6542</td>
<td>vmlinux</td>
<td>kmem_cache_free</td>
</tr>
</tbody>
</table>

32 cores:

- 10041 msg/sec

48 cores:

- 11705 msg/sec

- Functions execute more slowly on 48 cores
No obvious bottlenecks

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3319</td>
<td>5.4462</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>3119</td>
<td>5.2462</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>1966</td>
<td>3.3069</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1950</td>
<td>3.2800</td>
<td>vmlinux</td>
<td>page_fault</td>
</tr>
<tr>
<td>1627</td>
<td>2.7367</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
<tr>
<td>1626</td>
<td>2.7350</td>
<td>vmlinux</td>
<td>clear_page_c</td>
</tr>
<tr>
<td>1578</td>
<td>2.6542</td>
<td>vmlinux</td>
<td>kmem_cache_free</td>
</tr>
</tbody>
</table>

32 cores:
10041 msg/sec

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4207</td>
<td>5.3145</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>4191</td>
<td>5.2943</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>2632</td>
<td>3.3249</td>
<td>vmlinux</td>
<td>page_fault</td>
</tr>
<tr>
<td>2525</td>
<td>3.1897</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>2210</td>
<td>2.7918</td>
<td>vmlinux</td>
<td>clear_page_c</td>
</tr>
<tr>
<td>2131</td>
<td>2.6920</td>
<td>vmlinux</td>
<td>kmem_cache_free</td>
</tr>
<tr>
<td>2000</td>
<td>2.5265</td>
<td>vmlinux</td>
<td>dput</td>
</tr>
</tbody>
</table>

48 cores:
11705 msg/sec

- Functions execute more slowly on 48 cores
No obvious bottlenecks

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3319</td>
<td>5.4462</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>3119</td>
<td>5.2462</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>1966</td>
<td>3.3069</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1950</td>
<td>3.2800</td>
<td>vmlinux</td>
<td>page_fault</td>
</tr>
<tr>
<td>1627</td>
<td>2.7367</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
<tr>
<td>1626</td>
<td>2.7350</td>
<td>vmlinux</td>
<td>clear_page_c</td>
</tr>
<tr>
<td>1578</td>
<td>2.6542</td>
<td>vmlinux</td>
<td>kmem_cache_free</td>
</tr>
</tbody>
</table>

32 cores: 10041 msg/sec

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4207</td>
<td>5.3145</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>4191</td>
<td>5.2943</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>2632</td>
<td>3.3249</td>
<td>vmlinux</td>
<td>page_fault</td>
</tr>
<tr>
<td>2525</td>
<td>3.1897</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>2210</td>
<td>2.7918</td>
<td>vmlinux</td>
<td>clear_page_c</td>
</tr>
<tr>
<td>2131</td>
<td>2.6920</td>
<td>vmlinux</td>
<td>kmem_cache_free</td>
</tr>
<tr>
<td>2000</td>
<td>2.5265</td>
<td>vmlinux</td>
<td>dput</td>
</tr>
</tbody>
</table>

48 cores: 11705 msg/sec

- Functions execute more slowly on 48 cores
## No obvious bottlenecks

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3319</td>
<td>5.4462</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>3119</td>
<td>5.2462</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>1966</td>
<td>3.3069</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1950</td>
<td>3.2800</td>
<td>vmlinux</td>
<td>page_fault</td>
</tr>
<tr>
<td>1627</td>
<td>2.7367</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
<tr>
<td>1626</td>
<td>2.7350</td>
<td>vmlinux</td>
<td>clear_page_c</td>
</tr>
<tr>
<td>1578</td>
<td>2.6542</td>
<td>vmlinux</td>
<td>kmem_cache_free</td>
</tr>
</tbody>
</table>

32 cores: 10041 msg/sec

48 cores: 11705 msg/sec

- Functions execute more slowly on 48 cores
Bottleneck: reference counting

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

```c
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, ...

```c
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, ...

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

- Reading the reference count is slow
- 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;
    ...
};
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
    {
        return;
        dentry_free(dentry);
    }
}

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

struct dentry {
    ... int ref;
    ... }

120 – 4000 cycles depending on congestion
Reading the reference count delays memory operations from other cores

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

```c
tstruct dentry {
    ...;
    int ref;
    ...;
};
```
Reading the reference count delays memory operations from other cores

```c
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

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

```c
struct dentry {
    ...
    int ref; // READONLY
    ...
};
```
Reading the reference count delays memory operations from other cores

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

```c
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

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

```c
struct dentry {
    ...
    int ref;
    ...
};
```

- Contention on a reference count congests the interconnect
Reading the reference count delays memory operations from other cores

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

- Contention on a reference count congests the interconnect
Reading the reference count delays memory operations from other cores

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

- 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

![Diagram of Core 0 and Core 1 with shared dentry sloppy counter]

0

shared
dentry sloppy counter
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

![Diagram showing a dentry sloppy counter with per-core, shared, and per-core sections.](image-url)
Solution: sloppy counters

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

![Diagram showing per-core, shared, and per-core counters for dentry sloppy counter]
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

![Diagram showing per-core and shared references for Core 0 and Core 1, with a sloppy counter value of 2.]}
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

![Diagram showing_core_0_and_core_1_with_sloppy_counter](image-url)
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

![Diagram showing per-core and shared references](image-url)
Solution: sloppy counters

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

```
rm /tmp/foo
```

```
Core 0

Core 1
```

```
per-core 1 shared per-core

dentry sloppy counter
```
Solution: sloppy counters

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

```
rm /tmp/foo
```
Solution: sloppy counters

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

```
rm /tmp/foo
```

![Diagram showing_core_0_and_core_1_with_dentry_sloppy_counter](image)
Properties of sloppy counters

• Simple to start using:
  • Change data structure
  • $\text{atomic\_inc} \rightarrow \text{sloppy\_inc}$
• 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

Throughput with sloppy counters
Throughput with per-core lookup
Throughput of stock Linux

Throughput (messages/second)

Cores
Sloppy counters: more scalability

Throughput with sloppy counters
Throughput with per-core lookup
Throughput of stock Linux

Throughput (messages/second)

Cores

1 4 8 12 16 20 24 28 32 36 40 44 48
### Summary of changes

<table>
<thead>
<tr>
<th></th>
<th>memcached</th>
<th>Apache</th>
<th>Exim</th>
<th>PostgreSQL</th>
<th>gmake</th>
<th>Psearchy</th>
<th>Metis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mount tables</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open file table</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sloppy counters</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inode allocation</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock-free dentry lookup</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Super pages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DMA buffer allocation</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network stack false sharing</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel accept</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application modifications</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Application</th>
<th>Bottleneck</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcached</td>
<td>HW: transmit queues on NIC</td>
</tr>
<tr>
<td>Apache</td>
<td>HW: receive queues on NIC</td>
</tr>
<tr>
<td>Exim</td>
<td>App: contention on spool directories</td>
</tr>
<tr>
<td>gmake</td>
<td>App: serial stages and stragglers</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>App: spin lock</td>
</tr>
<tr>
<td>Psearchy</td>
<td>HW: cache capacity</td>
</tr>
<tr>
<td>Metis</td>
<td>HW: DRAM throughput</td>
</tr>
</tbody>
</table>

- 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