Scalability in the Clouds!
A Myth or Reality?

Sanidhya Kashyap, Changwoo Min, Taesoo Kim
Programmer's Paradise?

- A programmer day-to-day task: *program compilation*, like Linux kernel compilation.
- Relies on Buildbot to complete the job ASAP!
- Expects the job to complete sooner with increasing core count.
  - With respect to vertical scalability, a parallel job with no sequential bottleneck should scale with increasing core count.
Programmer's Paradise?

• A programmer day-to-day task: *program compilation*, like Linux kernel compilation.

How about using Cloud providers for our fun and their profit?

increasing core count.

- With respect to vertical scalability, a parallel job with no sequential bottleneck should scale with increasing core count.
Clouds Trend

- Trend is changing → Larger instances (40 vCPUs) are available.
- Will Buildbot really scale?
Scalability Behavior in the Clouds

![Graph showing the number of builds per hour against the number of vCPUs for EC2. The graph demonstrates an upward trend as the number of vCPUs increases.]
Scalability Behavior in the Clouds

The graph shows the scalability behavior of EC2 and GCE as the number of virtual CPUs (vCPUs) increases. The y-axis represents the number of builds per hour, while the x-axis represents the number of vCPUs. The lines indicate that both EC2 and GCE can handle increasing loads as the number of vCPUs increases, with GCE showing slightly higher scalability at higher vCPU counts.
Scalability Behavior in the Clouds

![Graph showing builds per hour vs. number of vCPUs for EC2, GCE, and Azure]

- **EC2** (green line)
- **GCE** (blue line)
- **Azure** (red line)
Scalability Behavior in the Clouds

- EC2
- GCE
- Azure

16-core E5

#vCPUs
Scalability Behavior in the Clouds

![Graph showing scalability behavior in the clouds for EC2, GCE, Azure, and 16-core E5. The graph plots builds per hour against the number of vCPUs.]
Scalability Behavior in the Clouds

![Graph showing scalability behavior across different cloud providers.](image-url)
Scalability Behavior in the Clouds

![Graph showing scalability behavior in the Clouds, with curves for EC2, GCE, Azure, and 16-core E5.](image)
Scalability Behavior in the Clouds

![Scalability Behavior in the Clouds Graph](image-url)
Scalability Behavior in VMs with Higher-core count

(builds / hour)

#vCPUs

Host
Scalability Behavior in VMs with Higher-core count

![Graph showing builds/hour vs #vCPUs for Host and Guest.]
Scalability Behavior in VMs with Higher-core count

![Graph showing scalability behavior for Host and Guest with varying vCPUs]

- **Host**: Green line showing scalability behavior.
- **Guest**: Blue line showing scalability behavior.

**X-axis (vCPUs)**: 0 to 160

**Y-axis (builds/hour)**: 0 to 250

- There is a significant difference in scalability between Host and Guest systems.
- At 20 vCPUs, Guest shows a drop in builds/hour compared to Host.
- Host maintains a steady increase in builds/hour up to 160 vCPUs.

**Key Observation**: Guest shows a 6.7x decrease in maintains a steady increase in builds/hour up to 160 vCPUs.

**Conclusion**: VMs with higher-core count exhibit different scalability patterns between Host and Guest systems, with Host showing a more consistent increase.
Performance degradation occurs due to drastic increase in VMEXITS (halt exits).

Why?

- Performance degradation occurs due to drastic increase in VMEXITS (halt exits).

![Graph showing performance degradation](image-url)
Performance degradation occurs due to drastic increase in VMEXITS (halt exits).

Why?

- Spinlock is sleeping!
Spinlock Evolution in the Linux Kernel
Spinlock Evolution in the Linux Kernel

Test-and-Set
Test-And-Set
spinlock
Spinlock Evolution in the Linux Kernel

Test-and-Set spinlock

Fairness

2.6.25 (April 2008)
Ticket spinlock
Spinlock Evolution in the Linux Kernel

- Test-and-Test-And-Set spinlock
  - Fairness
- 2.6.25 (April 2008) Ticket spinlock
  - Shared cacheline contention
- 3.15 (July 2014) qspinlock, variant of MCS lock (yet to be merged)
Spinlock Evolution in the Linux Kernel

- **Test-and-Test-And-Set spinlock**
- **2.6.25 (April 2008)**
  - Ticket spinlock
  - **Fairness**
- **3.11 (2013)**
  - Paravirtual Ticket spinlock
  - **Shared cacheline contention**
- **3.15 (July 2014)**
  - qspinlock, variant of MCS lock (yet to be merged)
Spinlock Evolution in the Linux Kernel

- Test-and-Test-And-Set spinlock
- 2.6.25 (April 2008) Ticket spinlock
- 3.11 (2013) Paravirtual Ticket spinlock
- 4.0 (May, 2015) OTicket
- 3.15 (July 2014) qspinlock, variant of MCS lock (yet to be merged)

- Fairness
- Shared cacheline contention
Ticket Spinlock

- Guaranteed FIFO ordering.
- Mitigates starvation with increasing core count.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;
void lock() {
    my_ticket = F&I(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
    }
out: ;
}
void unlock() {
    head++;
}
```
Ticket Spinlock

- Guaranteed FIFO ordering.
- Mitigates starvation with increasing core count.
Ticket Spinlock

- Guaranteed FIFO ordering.
- Mitigates starvation with increasing core count.
Ticket Spinlock

- Guaranteed FIFO ordering.
- Mitigates starvation with increasing core count.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;

void lock() {
    my_ticket = F&I(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
    }
out: ;
}
void unlock() {
    head++;
}
```
Ticket Spinlock

- Guaranteed FIFO ordering.
- Mitigates starvation with increasing core count.
Ticket Spinlock

- Guaranteed FIFO ordering.
- Mitigates starvation with increasing core count.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;

void lock() {
    my_ticket = F&I(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
                goto out;
        } while(--count);

        out: ;
    }
}

void unlock() {
    head++;
}
```
Ticket Spinlock

- Guaranteed FIFO ordering.
- Mitigates starvation with increasing core count.
Ticket Spinlock

- Guaranteed FIFO ordering.
- Mitigates starvation with increasing core count.
Complexity of Ticket Spinlock in Virtualized Environment

- vCPUs are scheduled by host scheduler.
- Semantic gap between the hypervisor and guest OS.
Complexity of Ticket Spinlock in Virtualized Environment

- *Lock Holder Preemption*: vCPU holding the lock gets preempted.
Complexity of Ticket Spinlock in Virtualized Environment

- **Lock Holder Preemption**: vCPU holding the lock gets preempted.
Complexity of Ticket Spinlock in Virtualized Environment

- **Lock Holder Preemption**: vCPU holding the lock gets preempted.
Complexity of Ticket Spinlock in Virtualized Environment

- *Lock Holder Preemption*: vCPU holding the lock gets preempted.

```
head = 0
tail = 2
```

![Diagram showing lock holder preemption]
Complexity of Ticket Spinlock in Virtualized Environment

- Lock Holder Preemption: vCPU holding the lock gets preempted.
Complexity of Ticket Spinlock in Virtualized Environment

- **Lock Holder Preemption**: vCPU holding the lock gets preempted.

```
head = 1
```

![Diagram showing lock holder preemption]
Complexity of Ticket Spinlock in Virtualized Environment

- *Lock Holder Preemption*: vCPU holding the lock gets preempted.
Complexity of Ticket Spinlock in Virtualized Environment

- *Lock Holder Preemption*: vCPU holding the lock gets preempted.

```
head = 1
```

![Diagram showing vCPU preemption](image)
Complexity of Ticket Spinlock in Virtualized Environment

- *Lock Waiter preemption*: The next waiter is preempted before acquiring the lock.
Complexity of Ticket Spinlock in Virtualized Environment

- **Lock Waiter preemption**: The next waiter is preempted before acquiring the lock.

```
head = 1
```

```
tail = 4
```

```
1 ← 2 ← 3
```

Scheduled  Preempted
Complexity of Ticket Spinlock in Virtualized Environment

- *Lock Waiter preemption*: The next waiter is preempted before acquiring the lock.

```
head = 1
```

```
tail = 4
```

![Diagram](image)
Complexity of Ticket Spinlock in Virtualized Environment

- **Lock Waiter preemption**: The next waiter is preempted before acquiring the lock.
Complexity of Ticket Spinlock in Virtualized Environment

- *Lock Waiter preemption*: The next waiter is preempted before acquiring the lock.

![Lock Waiter Preemption Diagram]

```
2       3
```

```
tail = 4
```

- Scheduled
- Preempted
Current Solution to LHP and LWP

• Handling lock requests depending on the lock state.
  – **Lock**: yield if long wait.
  – **Unlock**: wake up the preempted waiter.

• A paravirtual interface to track state change.
Paravirtual Ticket Spinlock

• **Lock:**
  - **Fast path:** spin till a certain threshold value.
  - **Slow path:** notify the hypervisor to de-schedule the thread.

• **Unlock:**
  - Wake-up procedure to re-schedule the next waiting thread.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;
void lock() {
    my_ticket = F&l(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
                goto out;
        } while(--count);
        slowpath_spin(tail);
    }
}
void unlock() {
    wakeup_cpu(head + 1);
    head++;
}
```

<table>
<thead>
<tr>
<th>Lock</th>
<th>Unlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Fast path: spin till a certain threshold value.</td>
<td>- Wake-up procedure to re-schedule the next waiting thread.</td>
</tr>
<tr>
<td>- Slow path: notify the hypervisor to de-schedule the thread.</td>
<td></td>
</tr>
</tbody>
</table>
Paravirtual Ticket Spinlock

• **Lock:**
  - **Fast path:** spin till a certain threshold value.
  - **Slow path:** notify the hypervisor to de-schedule the thread.

• **Unlock:**
  - Wake-up procedure to re-schedule the next waiting thread.
Paravirtual Ticket Spinlock

● **Lock:**
  - **Fast path:** spin till a certain threshold value.
  - **Slow path:** notify the hypervisor to de-schedule the thread.

● **Unlock:**
  - Wake-up procedure to re-schedule the next waiting thread.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;
void lock() {
    my_ticket = F&I(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
        slowpath_spin(tail);
    }
}
void unlock() {
    wakeup_cpu(head + 1);
    head++;
}
```
Paravirtual Ticket Spinlock

• Lock:
  - **Fast path**: spin till a certain threshold value.
  - **Slow path**: notify the hypervisor to de-schedule the thread.

• Unlock:
  - Wake-up procedure to re-schedule the next waiting thread.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;

void lock() {
    my_ticket = F&I(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
        + slowpath_spin(tail);
    }
    out: ;
}

void unlock() {
    + wakeup_cpu(head + 1);
    head++;
}
```
Paravirtual Ticket Spinlock

- **Lock:**
  - **Fast path**: spin till a certain threshold value.
  - **Slow path**: notify the hypervisor to de-schedule the thread.

- **Unlock:**
  - Wake-up procedure to re-schedule the next waiting thread.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;
void lock() {
    int my_ticket = F&l(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
        slowpath_spin(tail);
    }
    out: ;
}
void unlock() {
    wakeup_cpu(head + 1);
    head++;
}
```
Paravirtual Ticket Spinlock

• Lock:
  - Fast path: spin till a certain threshold value.
  - Slow path: notify the hypervisor to de-schedule the thread.

• Unlock:
  - Wake-up procedure to re-schedule the next waiting thread.
# Paravirtual Ticket Spinlock

**Lock:**
- **Fast path:** spin till a certain threshold value.
- **Slow path:** notify the hypervisor to de-schedule the thread.

**Unlock:**
- Wake-up procedure to re-schedule the next waiting thread.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;
void lock() {
    my_ticket = F&I(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if (my_ticket == head);
            goto out;
        } while(--count);
        + slowpath_spin(tail);
    }
    out: ;
}
void unlock() {
    + wakeup_cpu(head + 1);
    head++;
}
```
Paravirtual Ticket Spinlock

• Lock:
  - **Fast path**: spin till a certain threshold value.
  - **Slow path**: notify the hypervisor to de-schedule the thread.

• Unlock:
  - Wake-up procedure to re-schedule the next waiting thread.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;

void lock() {
    my_ticket = F&I(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
        + slowpath_spin(tail);
    }
    out: ;
}

void unlock() {
    + wakeup_cpu(head + 1);
    head++;
}
```
Paravirtual Ticket Spinlock

• **Lock:**
  - **Fast path:** spin till a certain threshold value.
  - **Slow path:** notify the hypervisor to de-schedule the thread.

• **Unlock:**
  - Wake-up procedure to re-schedule the next waiting thread.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;

void lock() {
    my_ticket = F&I(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
        slowpath_spin(tail);
    }
}

void unlock() {
    wakeup_cpu(head + 1);
    head++;
}
```

- **Lock:**
  - Fast path: spin till a certain threshold value.
  - Slow path: notify the hypervisor to de-schedule the thread.

- **Unlock:**
  - Wake-up procedure to re-schedule the next waiting thread.
Paravirtual Ticket Spinlock

• Lock:
  - **Fast path**: spin till a certain threshold value.
  - **Slow path**: notify the hypervisor to de-schedule the thread.

• Unlock:
  - Wake-up procedure to re-schedule the next waiting thread.
Paravirtual Ticket Spinlock

• Lock:
  - **Fast path**: spin till a certain threshold value.
  - **Slow path**: notify the hypervisor to de-schedule the thread.

• Unlock:
  - Wake-up procedure to re-schedule the next waiting thread.

```c
#define SPIN_THRESHOLD (1 << 15)
int head = 0;
int tail = 0;
int threshold = SPIN_THRESHOLD;
void lock() {
    my_ticket = F&I(tail);
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
        slowpath_spin(tail);
    } while(--count);
    out: ;
}
void unlock() {
    wakeup_cpu(head + 1);
    head++;
}
```
Problem: The Mechanism to Annotate the Slow Behavior

The `slowpath_spin` issues the `hlt` instruction

The hypervisor traps the instruction

Then it de-schedules the vCPU.

- Probable cause of degradation:
  - Most vCPUs trap to the hypervisor
  - Switching overhead between guest and host + communication cost to wake-up other vCPUs increases
Key idea: Ordering

- OTicket tries to exploit the ordering.

- **Lock:**
  - Lower ticket distance $\rightarrow$ longer spin.
  - Allows more spinning to nearby waiters.

- **Unlock:**
  - Wake-up multiple waiters.
  - Reduces latency for the upcoming waiters.
OTicket: Opportunistic Spinning

```c
#define EAGER_WAITERS 4
#define TICKET_QUEUE 18
#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD 15

int head = 0;
int tail = 0;

u64 threshold = SPIN_THRESHOLD;

void lock() {
    my_ticket = F&I(tail);
    if(my_ticket - head < TICKET_QUEUE) {
        threshold = SPIN_MAX_THRESHOLD
                    >> (dist – 1);
    }
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
        slowpath_spin(tail);
    }
    out: ;
}
```
OTicket: Opportunistic Spinning

```c
#define EAGER_WAITERS 4
#define TICKET_QUEUE 18
#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD 15

int head = 0;
int tail = 0;

u64 threshold = SPIN_THRESHOLD;

void lock() {
    my_ticket = F&I(tail);
    if (my_ticket - head < TICKET_QUEUE) {
        threshold = SPIN_MAX_THRESHOLD >> (dist – 1);
    }
    for( ; ; ) {
        int count = threshold;
        do {
            if (my_ticket == head);
            goto out;
        } while(--count);
        slowpath_spin(tail);
    }
    out: ;
}
```
OTicket: Opportunistic Spinning

```c
#define EAGER_WAITERS 4
#define TICKET_QUEUE 18
#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD 15
int head = 0;
int tail = 0;
u64 threshold = SPIN_THRESHOLD;
void lock() {
    my_ticket = F&l(tail);
    if (my_ticket - head < TICKET_QUEUE) {
        threshold = SPIN_MAX_THRESHOLD >> (dist – 1);
    }
    for( ; ; ) {
        int count = threshold;
        do {
            if (my_ticket == head);
            goto out;
        } while (--count);
        slowpath_spin(tail);
    }
    out: ;
}
```
OTicket: Opportunistic Spinning

```c
#include <asm-generic/bitops.h>

#define EAGER_WAITERS 4
#define TICKET_QUEUE 18
#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD 15

int head = 0;
int tail = 0;

+u64 threshold = SPIN_THRESHOLD;

void lock() {
  my_ticket = F&l(tail);

  +if(my_ticket - head < TICKET_QUEUE) {
    threshold = SPIN_MAX_THRESHOLD
    >> (dist – 1);
  }

  for( ; ; ) {
    int count = threshold;
    do {
      if(my_ticket == head);
      goto out;
    } while(--count);
    slowpath_spin(tail);
  }
  out: ;
}
```
OTicket: Opportunistic Spinning

```c
#define EAGER_WAITERS 4
#define TICKET_QUEUE 18
#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD 15

int head = 0;
int tail = 0;

u64 threshold = SPIN_THRESHOLD;

void lock() {
    my_ticket = F&I(tail);
    if(my_ticket - head < TICKET_QUEUE) {
        threshold = SPIN_MAX_THRESHOLD
        >> (dist – 1);
    }
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
        slowpath_spin(tail);
    }
    out: ;
}
```
#define EAGER_WAITERS 4
#define TICKET_QUEUE 18
#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD 15
int head = 0;
int tail = 0;
+ u64 threshold = SPIN_THRESHOLD;
void lock() {
    my_ticket = F&I(tail);
    + if(my_ticket - head < TICKET_QUEUE) {
    +     threshold = SPIN_MAX_THRESHOLD
    +     >> (dist – 1);
    + }
    for( ; ; ) {
        int count = threshold;
        do {
            if(my_ticket == head);
            goto out;
        } while(--count);
        } while(--count);
        slowpath_spin(tail);
    }
out: ;
}
OTicket: Opportunistic Spinning

```c
#define EAGER_WAITERS 4
#define TICKET_QUEUE 18
#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD 15

int head = 0;
int tail = 0;

u64 threshold = SPIN_THRESHOLD;

void lock() {
    my_ticket = F&I(tail);

    if (my_ticket - head < TICKET_QUEUE) {
        threshold = SPIN_MAX_THRESHOLD >> (dist - 1);
    }

    for( ; ; ) {
        int count = threshold;
        do {
            if (my_ticket == head);
        } while (--count);

        slowpath_spin(tail);
    }
}
```

1000
100
10

1000
100
10
OTicket: Opportunistic Spinning

```c
#define EAGER_WAITERS 4
#define TICKET_QUEUE 18
#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD 15

int head = 0;
int tail = 0;

u64 threshold = SPIN_THRESHOLD;

void lock() {
    my_ticket = F&I(tail);
    if (my_ticket - head < TICKET_QUEUE) {
        threshold = SPIN_MAX_THRESHOLD
                      >> (dist – 1);
    }
    for( ; ; ) {
        int count = threshold;
        do {
            if (my_ticket == head);
        } while(--count);
        slowpath_spin(tail);
    }
}
```

1000 → 100 → 10 → lock → slowpath_spin(tail)
OTicket: Opportunistic Wake-up

```c
void unlock() {
    for (count = 1; count <= EAGER_WAITERS; ++count) {
        wakeup_cpu(head + count);
    }
    head++;
}
```

[t]  [t+1]  [t+2]  [t+3]  Wake-up sleeping waiters
OTicket: Opportunistic Wake-up

```c
void unlock() {
    for(count = 1; count <= EAGER_WAITERS; ++count) {
        wakeup_cpu(head + count);
    }
    head++;
}
```

Wake-up sleeping waiters
OTicket: Opportunistic Wake-up

```c
void unlock() {
    for (count = 1; count <= EAGER_WAITERS; ++count) {
        wakeup_cpu(head + count);
    }
    head++;
}
```

Wake-up sleeping waiters
OTicket: Opportunistic Wake-up

```c
void unlock() {
    for (count = 1; count <= EAGER_WAITERS; ++count) {
        wakeup_cpu(head + count);
    }
    head++;
}
```

Wake-up sleeping waiters
OTicket: Opportunistic Wake-up

```c
void unlock() {
    for (count = 1; count <= EAGER_WAITERS; ++count) {
        wakeup_cpu(head + count);
    }
    head++;
}
```

Wake-up sleeping waiters

1000

100
OTicket: Opportunistic Wake-up

```c
void unlock() {
    for (count = 1; count <= EAGER_WAITERS; ++count) {
        wakeup_cpu(head + count);
    }
    head++;
}
```

Wake-up sleeping waiters
OTicket: Opportunistic Wake-up

```c
void unlock() {
    for (count = 1; count <= EAGER_WAITERS; ++count) {
        wakeup_cpu(head + count);
    }
    head++;
}
```

Wake-up sleeping waiters
OTicket: Opportunistic Wake-up

```c
void unlock() {
    for (count = 1; count <= EAGER_WAITERS; ++count) {
        wakeup_cpu(head + count);
    }
    head++;
}
```

Wake-up sleeping waiters
Outline

• Scalability issue in the Clouds
• Scalability issue in VMs with higher core count
• OTicket design
• Evaluation
• Conclusion
OTicket: Guest vs Host

- Improves guest performance by almost 5x.
- Reduces halt exits by 6x.
OTicket: Guest vs Host

- Improves guest performance by almost $5x$.
- Reduces halt exits by $6x$. 
OTicket Performance Breakdown

- Opportunistic spinning prohibits sleeping.
OTicket Performance Breakdown

- Opportunistic spinning prohibits sleeping.
OTicket Performance Breakdown

- Opportunistic spinning prohibits sleeping.
Importance of Wake-ups

- Oversubscribed tenants.
- OTicket performs better due to opportunistic wake-up.
Importance of Wake-ups

- Oversubscribed tenants.
- OTicket performs better due to opportunistic wake-up.

![Graph showing builds/hour vs. #vCPUs]

- Guest
- Longer spinning
Importance of Wake-ups

- Oversubscribed tenants.
- OTicket performs better due to opportunistic wake-up.
Other Spinlock Alternatives

- Two spinlock implementations:
  - Current ticket spinlock
  - Fast-queue spinlock
Other Spinlock Alternatives

- Two spinlock implementations:
  - Current ticket spinlock
  - Fast-queue spinlock
Other Spinlock Alternatives

- Two spinlock implementations:
  - Current ticket spinlock
  - Fast-queue spinlock

Qspinlock has the same issue.
Our design has been already acknowledged!
Conclusion

• Identified a new class of problem.
  – not cacheline contention.
  – sleepy spinlock anomaly.

• Carefully utilized the ordering property can scale the spinlock:
  – Opportunistic spinning.
  – Opportunistic wake-up.
locking/qspinlock: Enhance pvqspinlock & introduce queued unfair lock

From: Waiman Long <Waiman.Long@hp.com>
To: Peter Zijlstra <peterz@infradead.org>, Ingo Molnar <mingo@redhat.com>, Thomas Gleixner <tglx@linutronix.de>, "H. Peter Anvin" <hpa@zytor.com>
Subject: [PATCH 0/7] locking/qspinlock: Enhance pvqspinlock & introduce queued unfair lock
Date: Sat, 11 Jul 2015 16:55:51 -0400
Message-ID: <1436647018-49734-1-git-send-email-Waiman.Long@hp.com>
Cc: x86@kernel.org, linux-kernel@vger.kernel.org, Scott J Norton <scott.norton@hp.com>, Douglas Hatch <doug.hatch@hp.com>, Waiman Long <Waiman.Long@hp.com>

Archive-link: Article, Thread

This patchset consists of two parts:

1) Patches 1-5 enhance the performance of PV qspinlock especially for overcommitted guest. The first patch moves all the CPU kicking to the unlock code. The 2nd and 3rd patches implement a kick-ahead and wait-early mechanism that was shown to improve performance for overcommitted guest. They are inspired by the "Do Virtual Machines Really Scale?" blog from Sanidhya Kashyap. The 4th patch adds code to collect PV qspinlock statistics. The last patch adds the pending bit support to PV qspinlock to improve performance at light load. This is important as the PV queuing code has even higher overhead than the native queuing code.
Future Work

- Scalability of other synchronization primitives in virtualized environment?
Thank you!

Sanidhya Kashyap
sanidhya@gatech.edu
Changwoo Min, Taesoo Kim

Questions?
https://github.com/sslab-gatech/vbench