Scalability in the Clouds! A Myth or Reality?

Sanidhya Kashyap, Changwoo Min, Taesoo Kim

Georgia Institute of Technology

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.

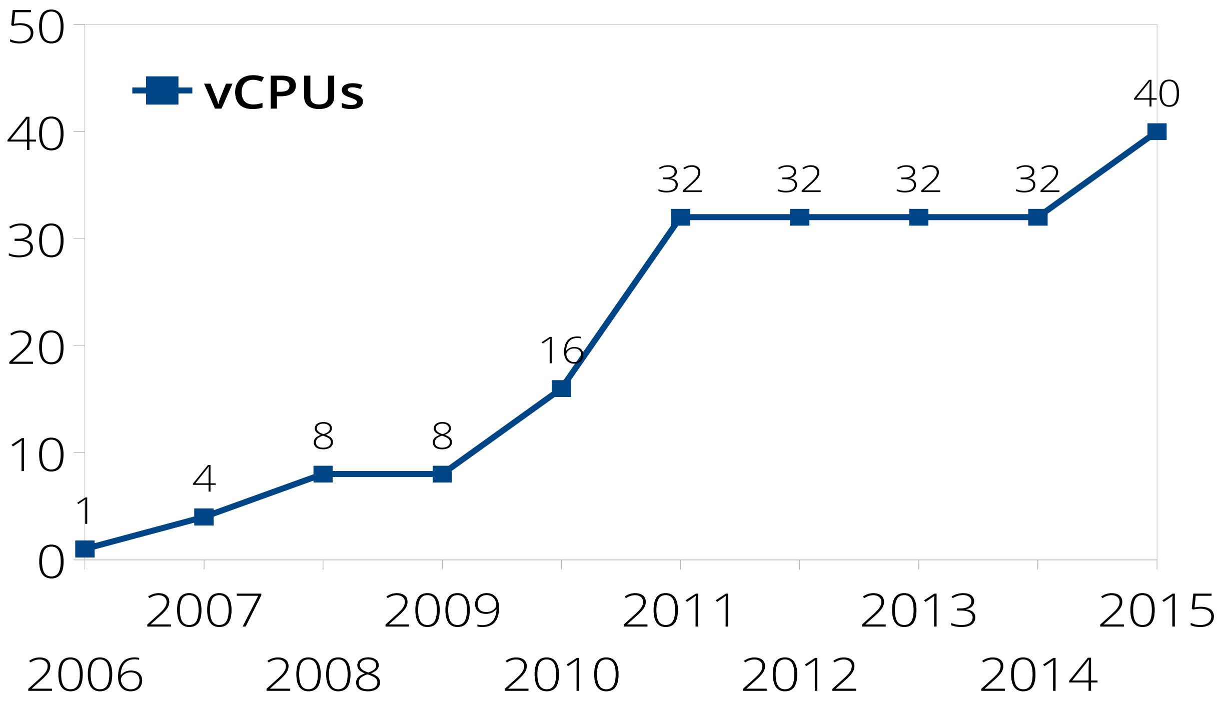
How about using Cloud providers for our fun and their profit?

increasing core count.

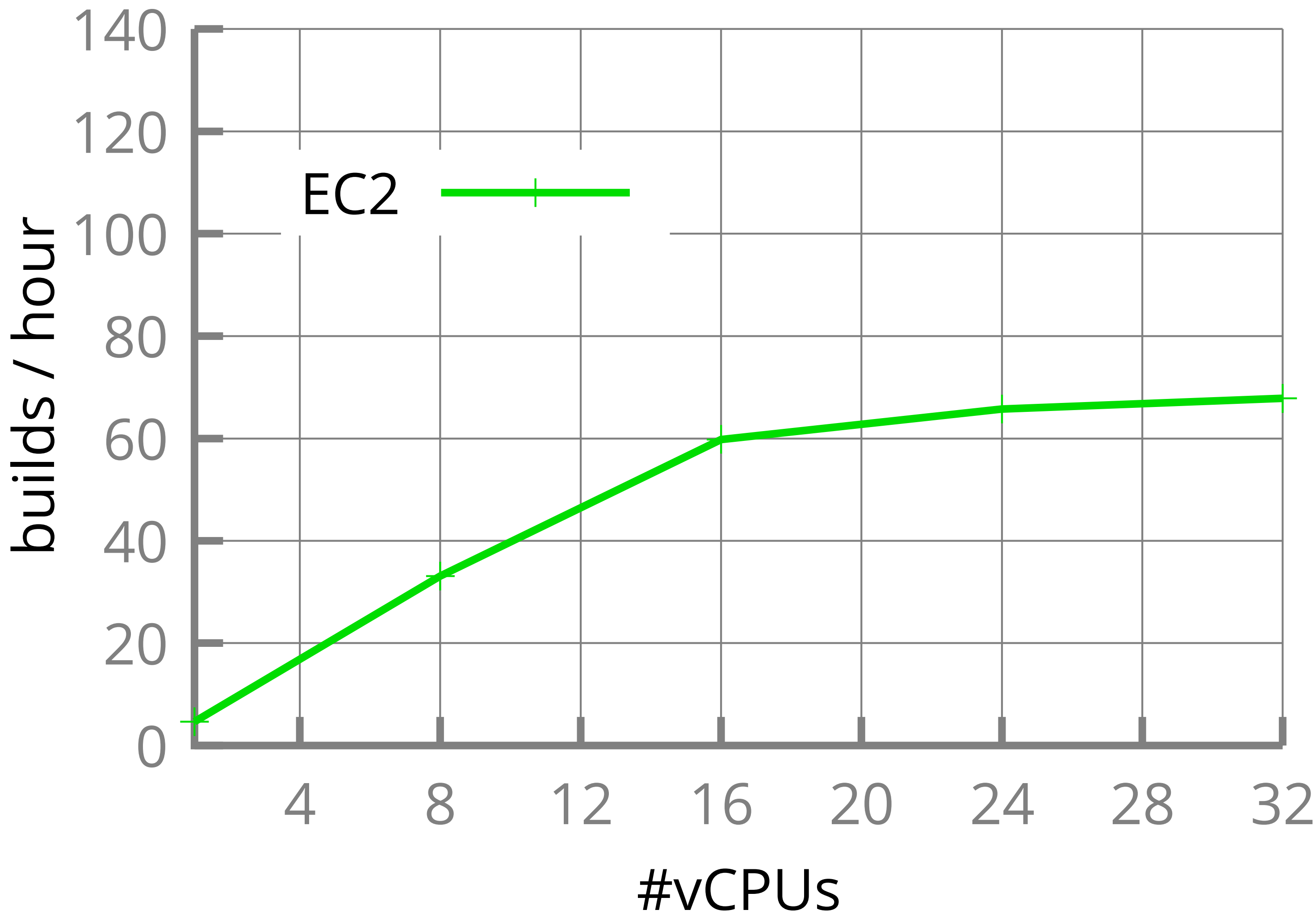
  - With respect to vertical scalability, a parallel job with no sequential bottleneck should scale with increasing core count.

Clouds Trend

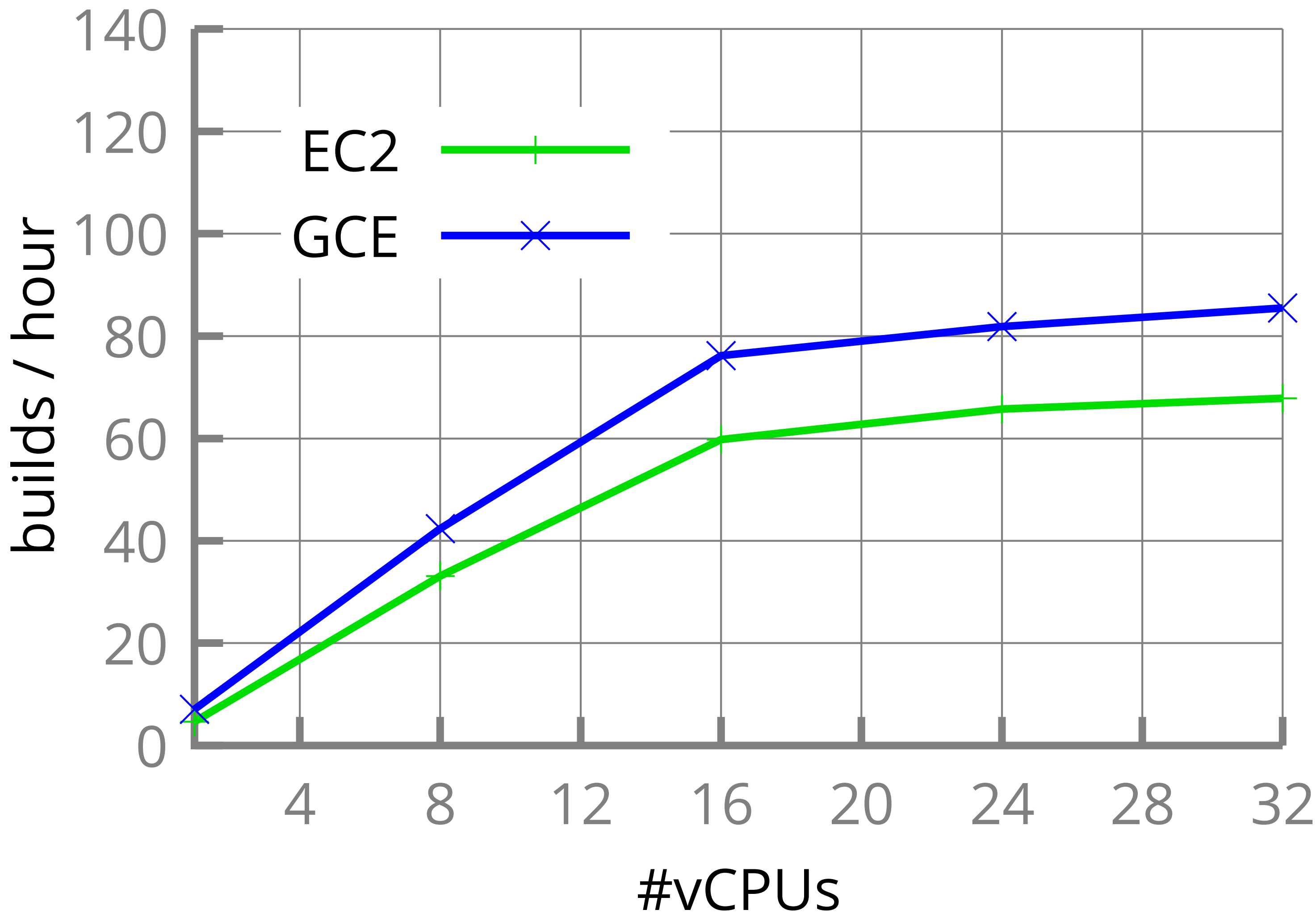
- Trend is changing → Larger instances (40 vCPUs) are available.
- Will Buildbot really scale?



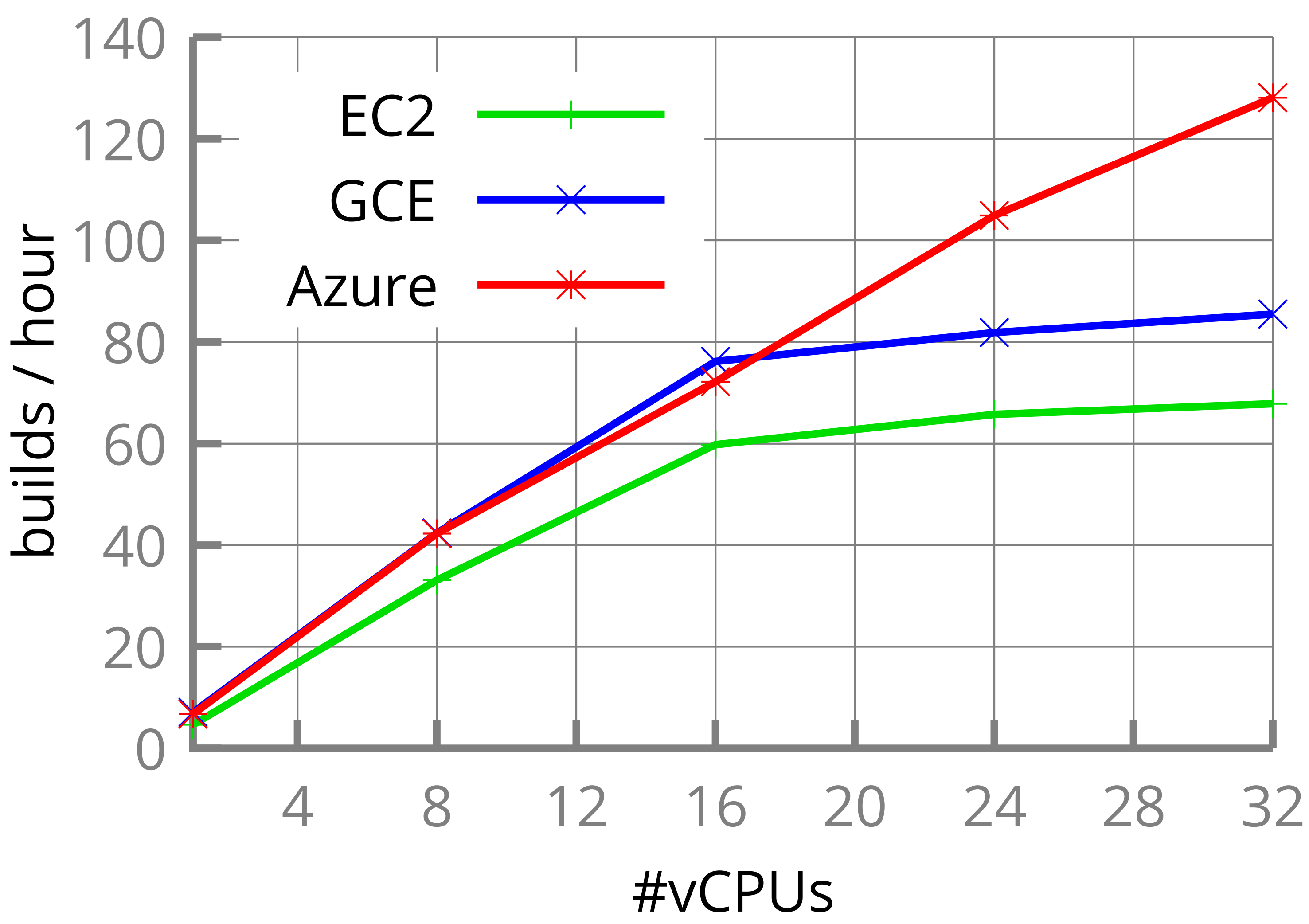
Scalability Behavior in the Clouds



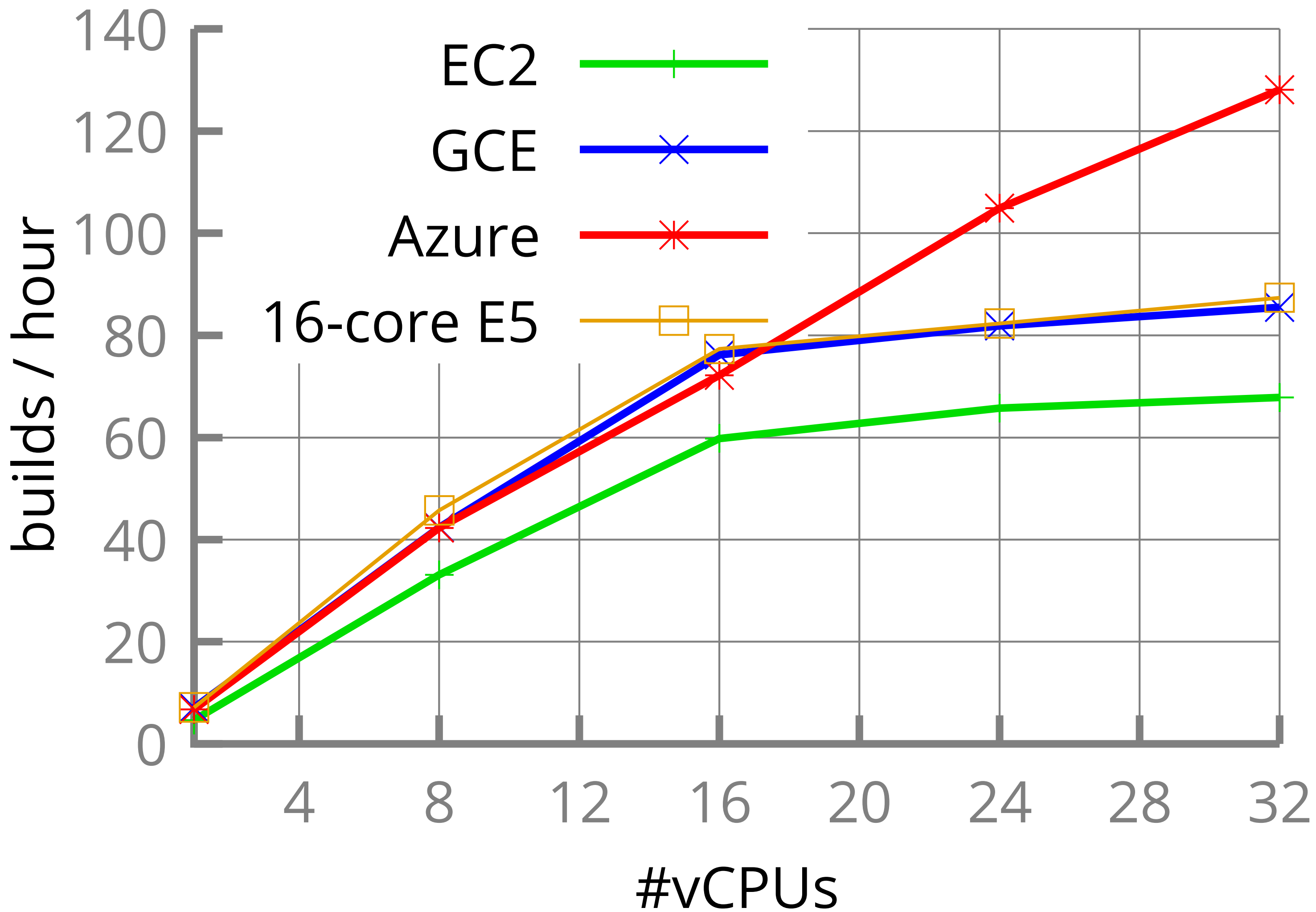
Scalability Behavior in the Clouds



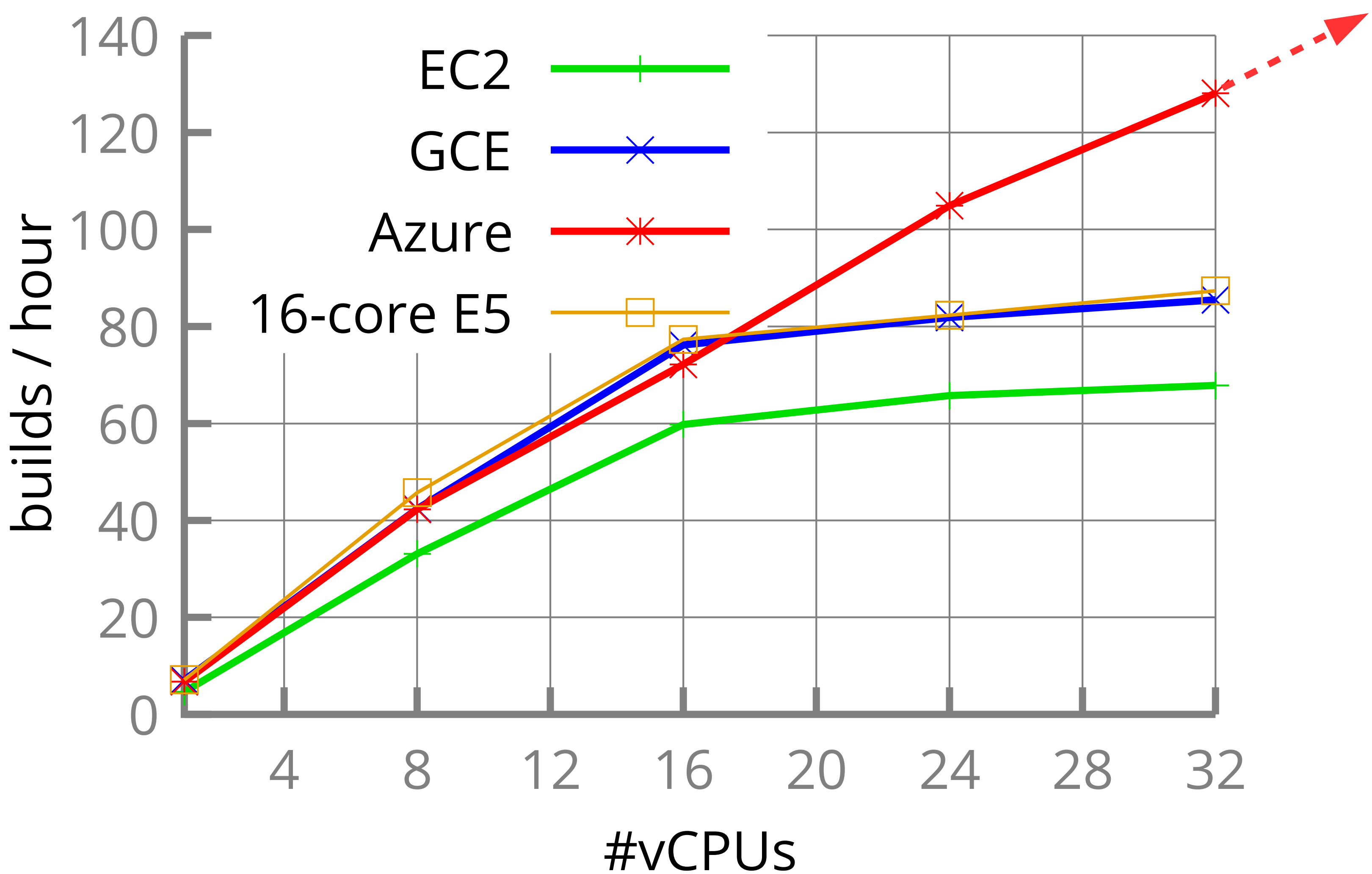
Scalability Behavior in the Clouds



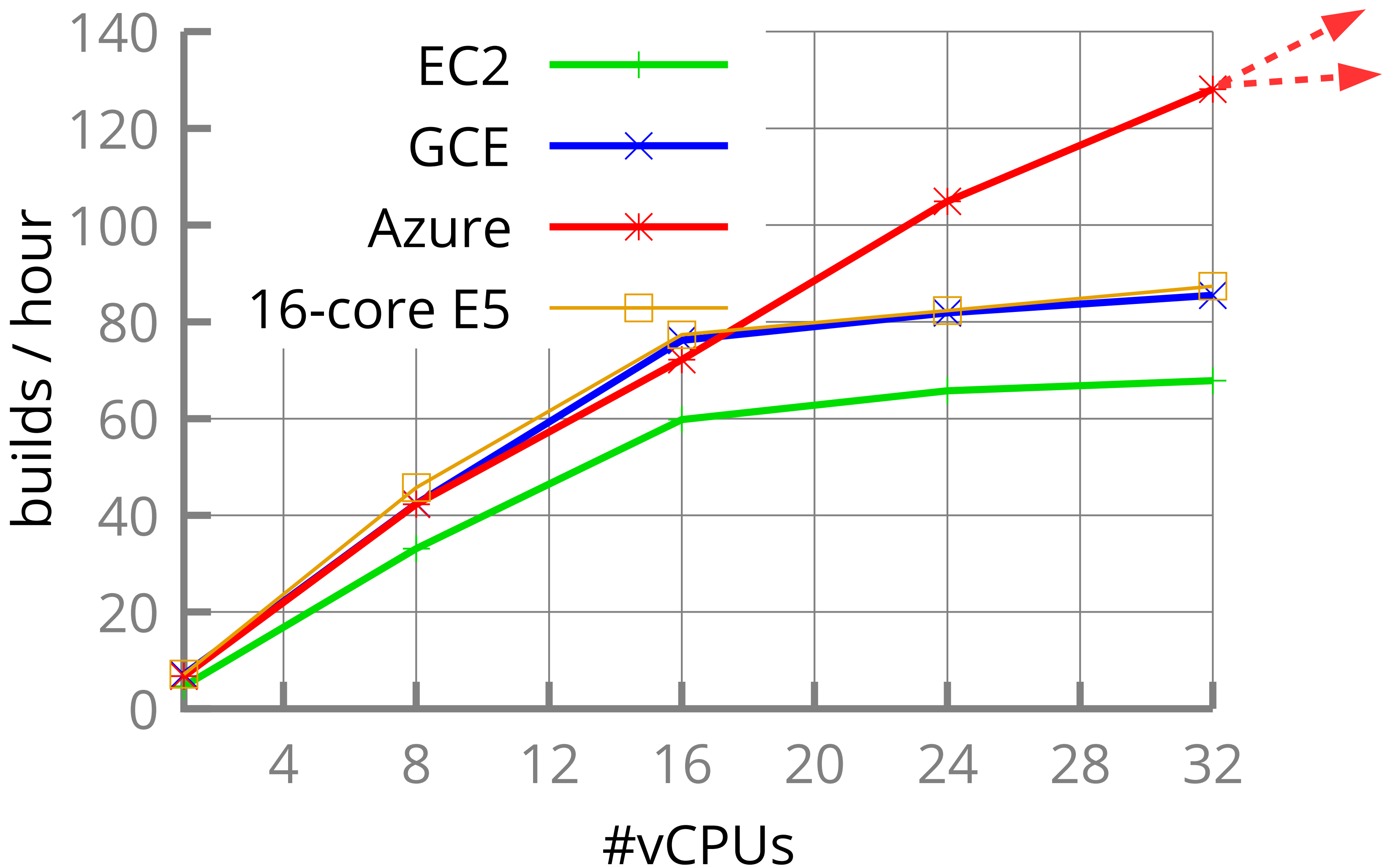
Scalability Behavior in the Clouds



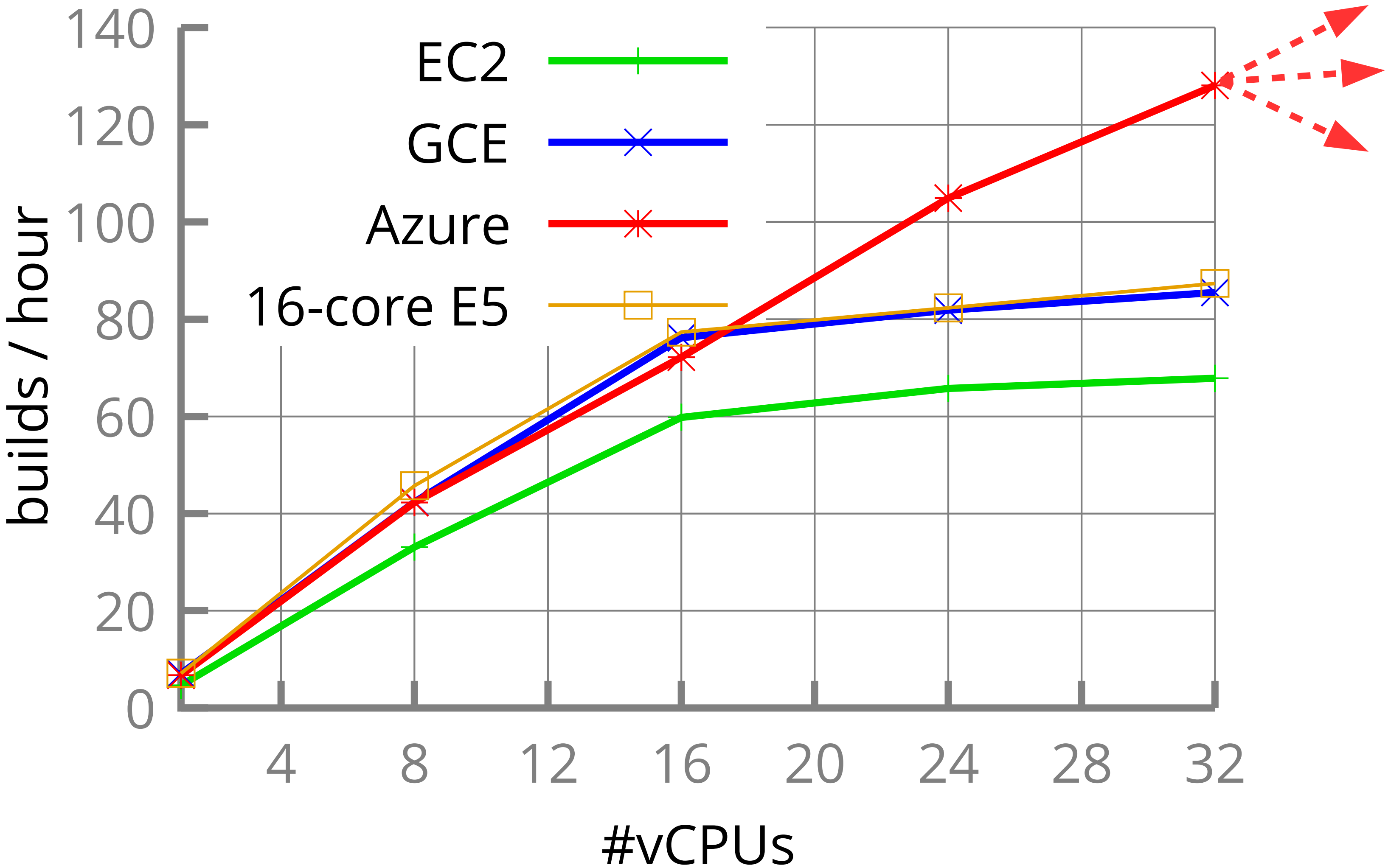
Scalability Behavior in the Clouds



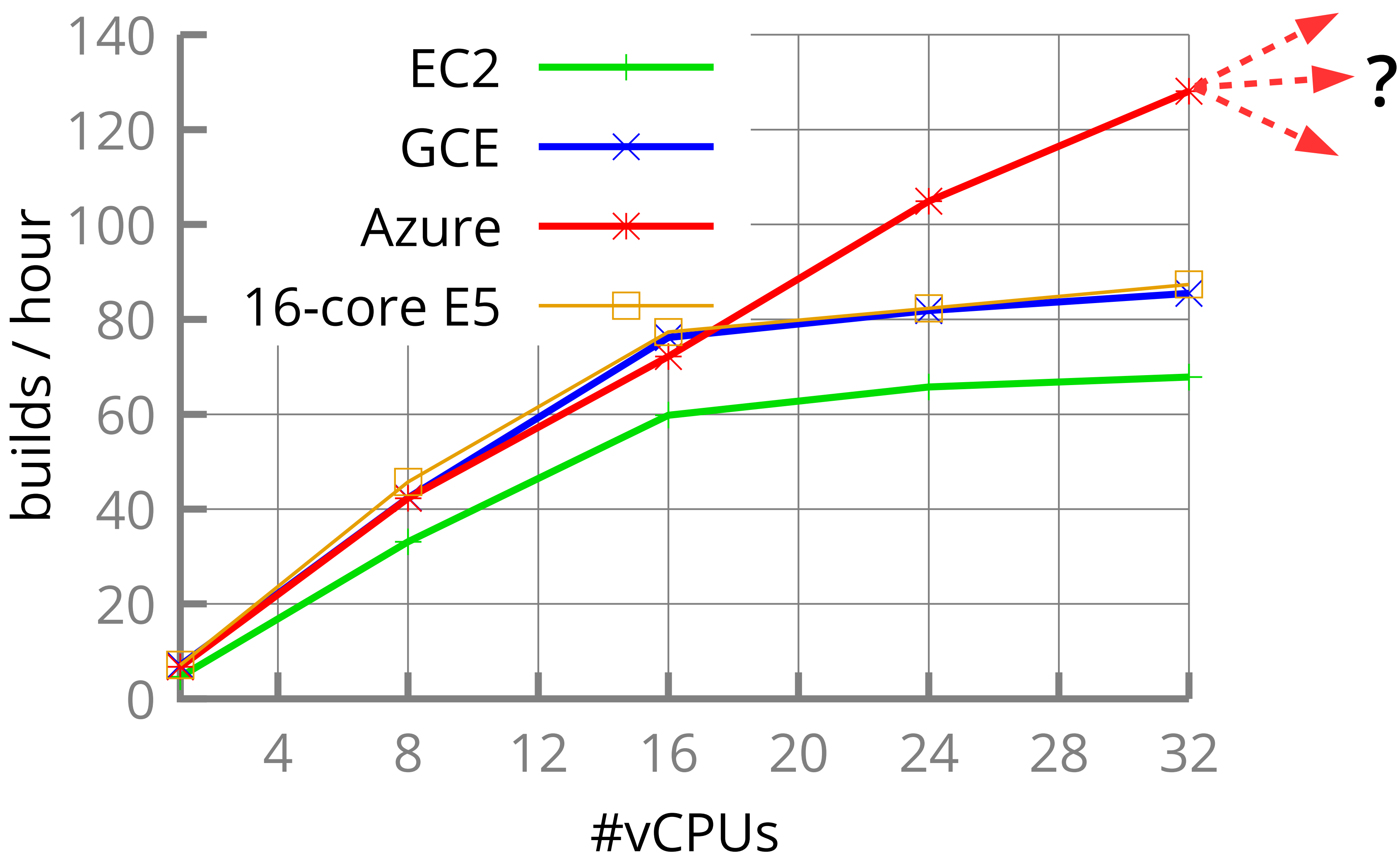
Scalability Behavior in the Clouds



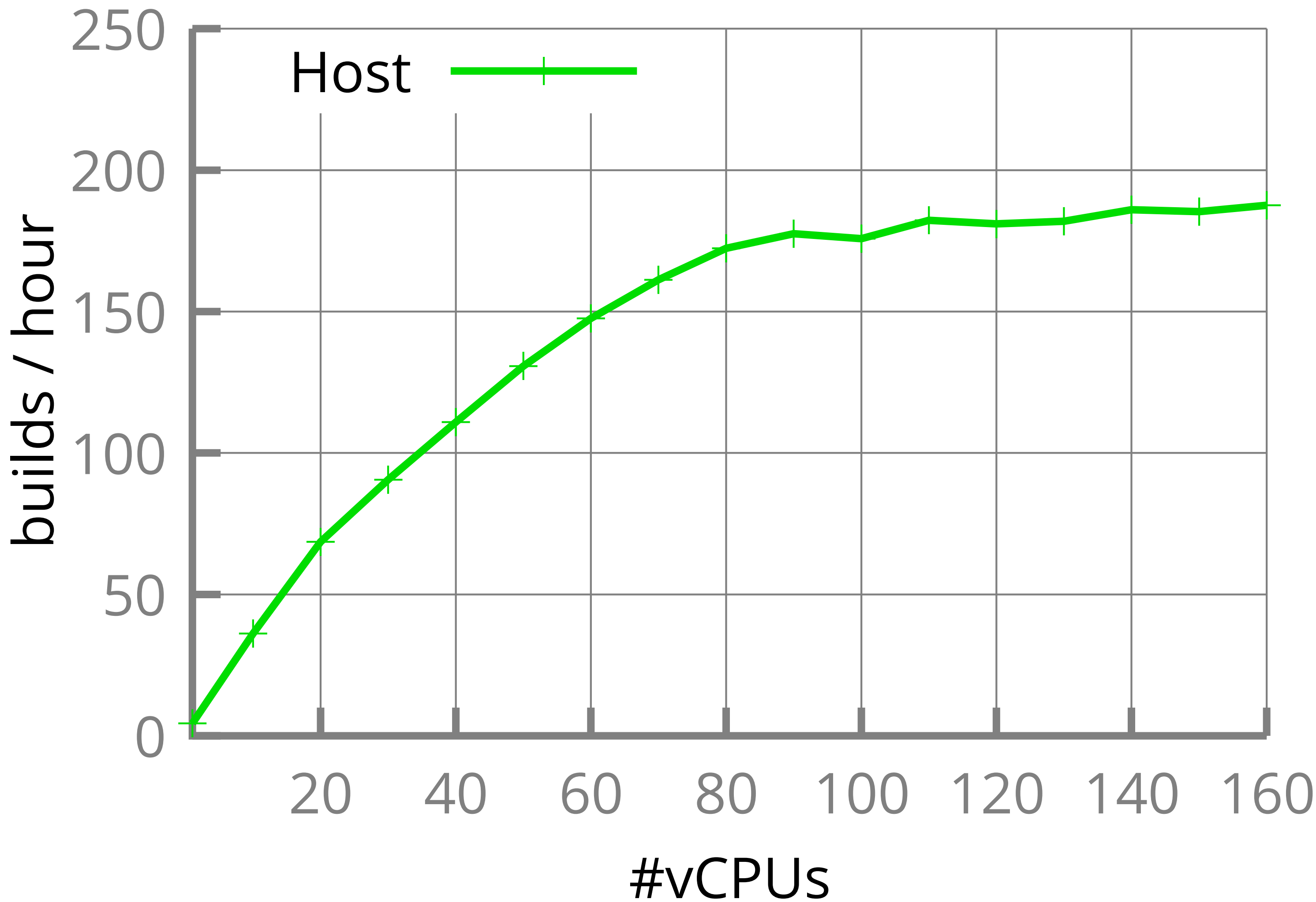
Scalability Behavior in the Clouds



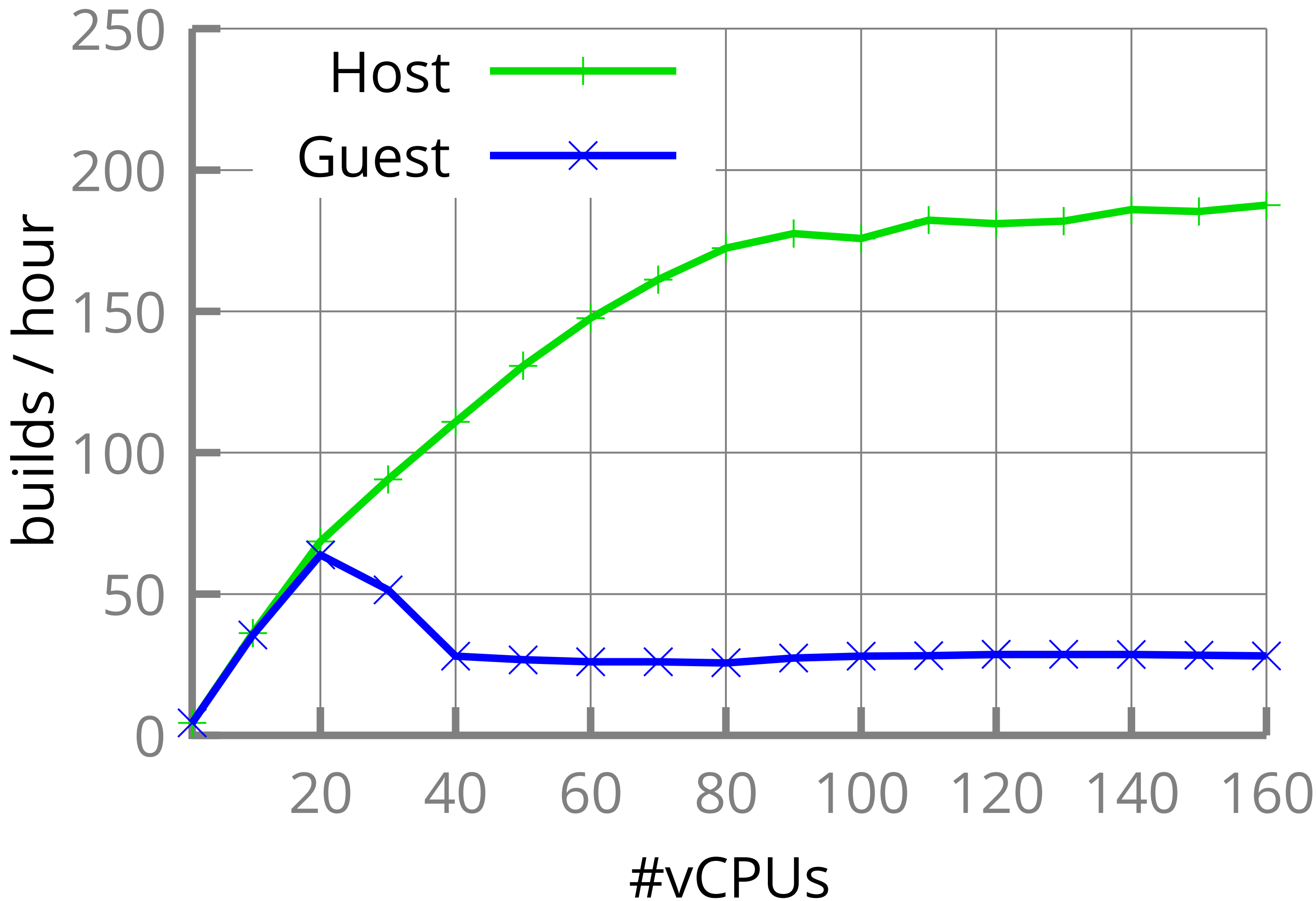
Scalability Behavior in the Clouds



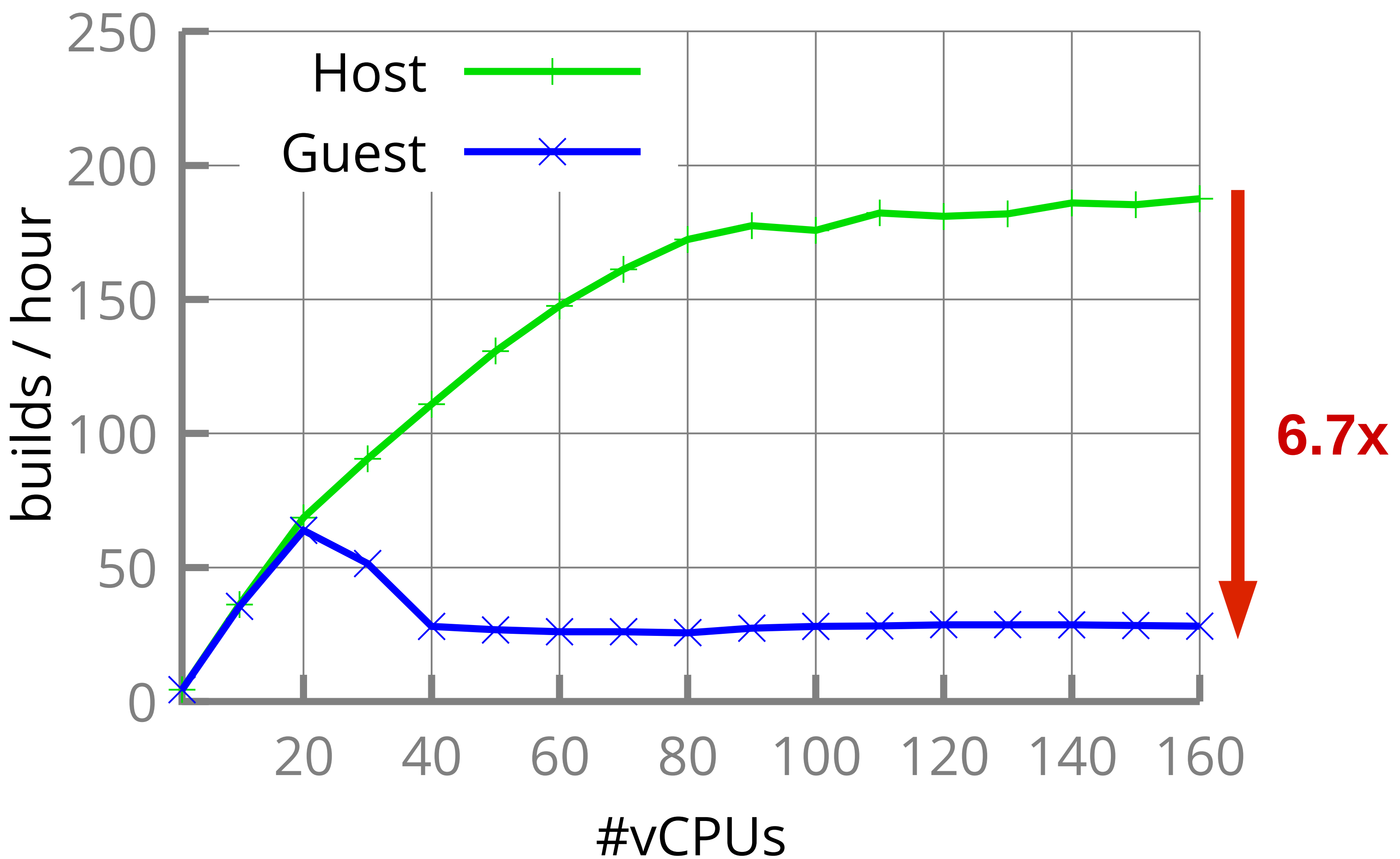
Scalability Behavior in VMs with Higher-core count



Scalability Behavior in VMs with Higher-core count

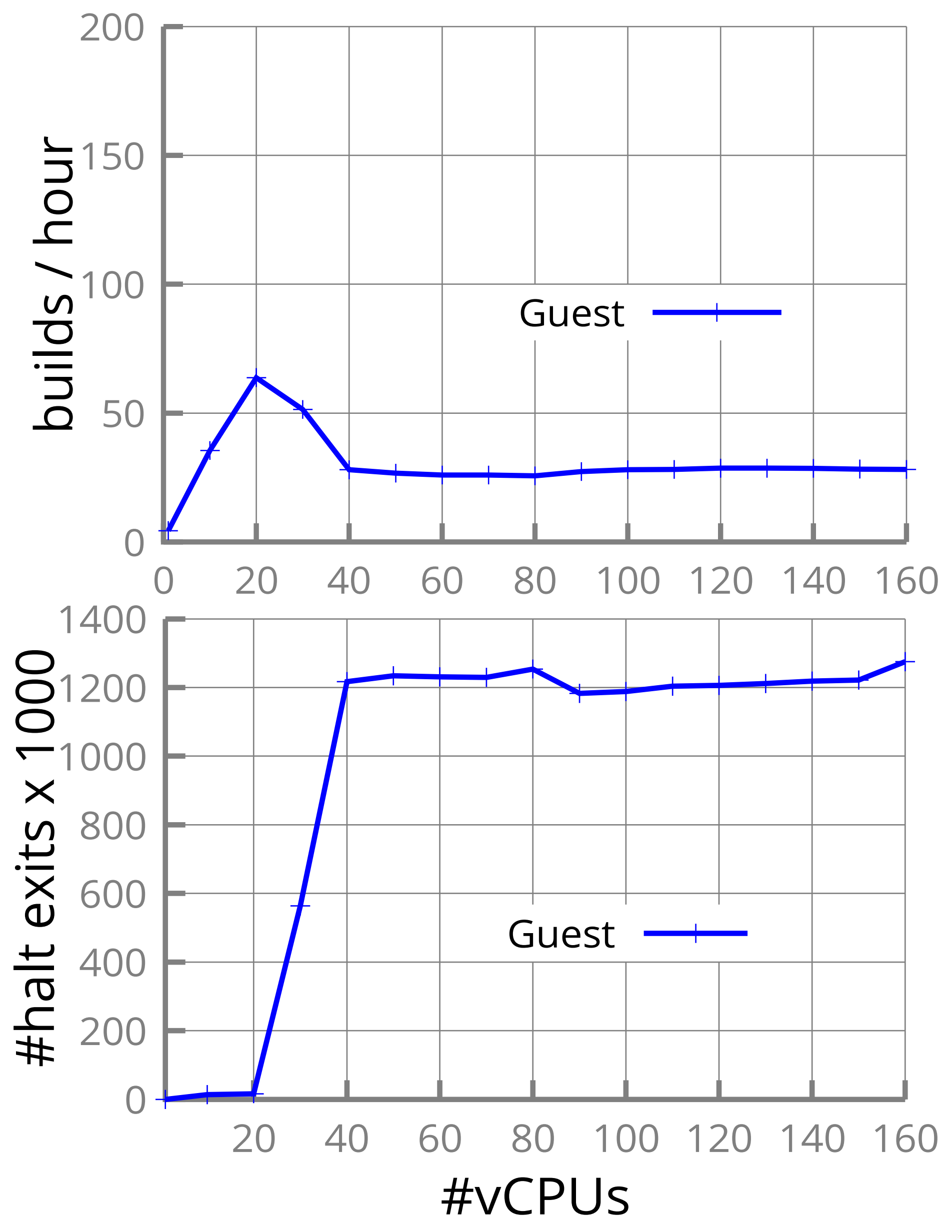


Scalability Behavior in VMs with Higher-core count



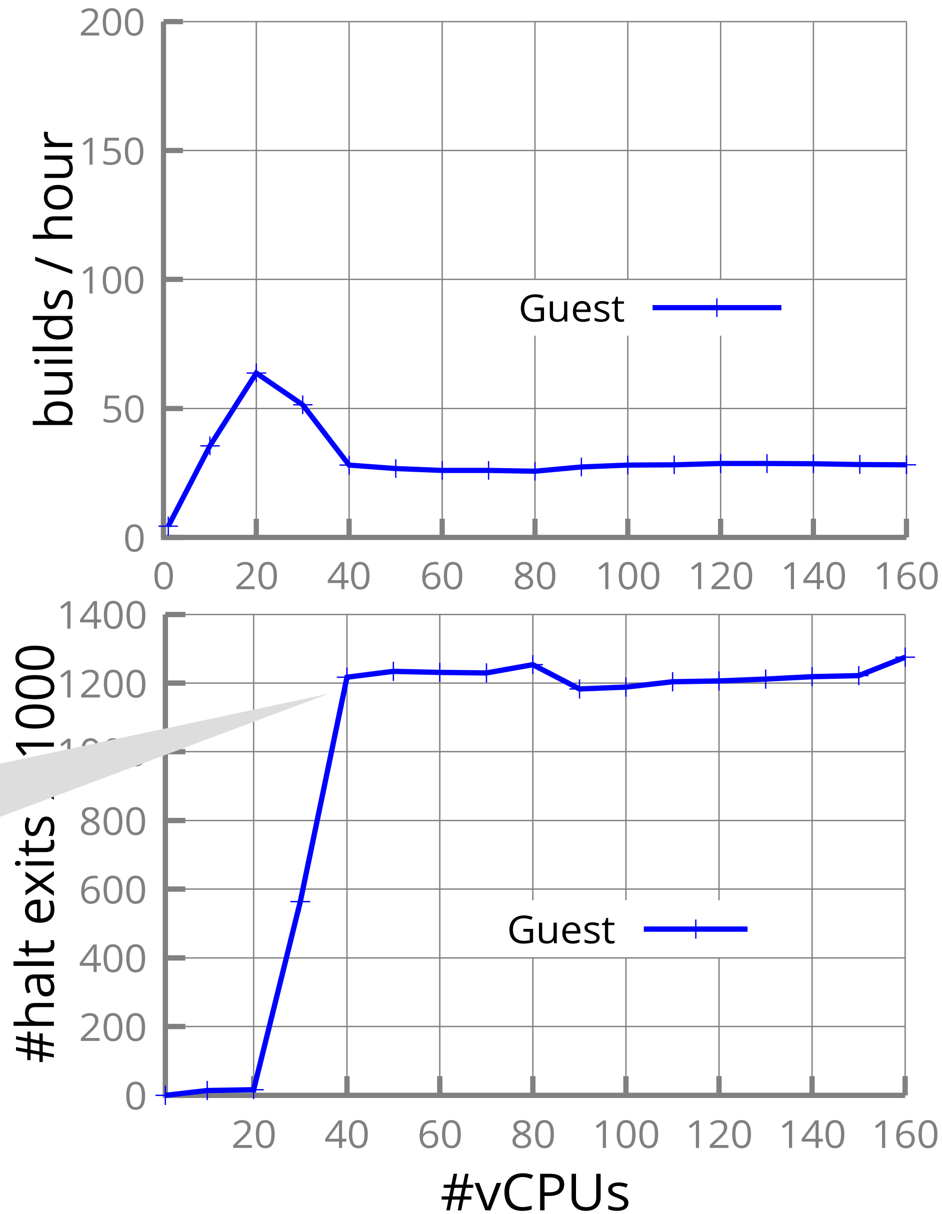
Why?

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



Why?

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



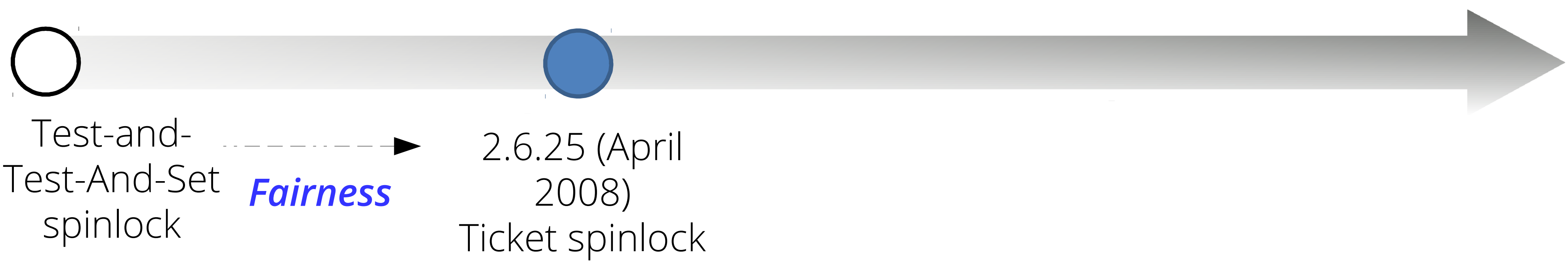
Spinlock is sleeping!

Spinlock Evolution in the Linux Kernel

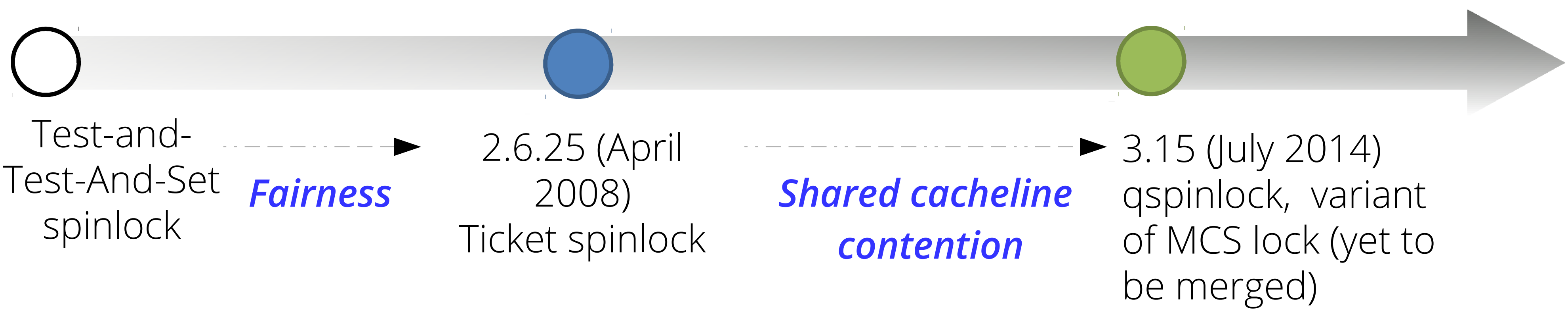
Spinlock Evolution in the Linux Kernel

Test-and-
Test-And-Set
spinlock

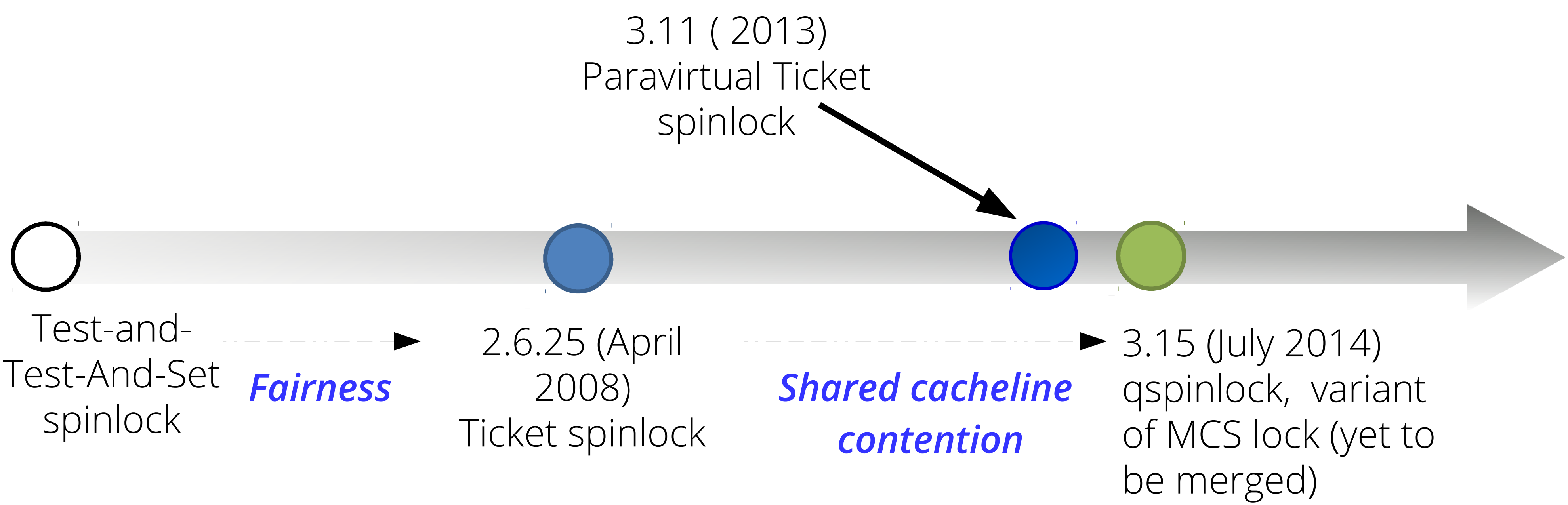
Spinlock Evolution in the Linux Kernel



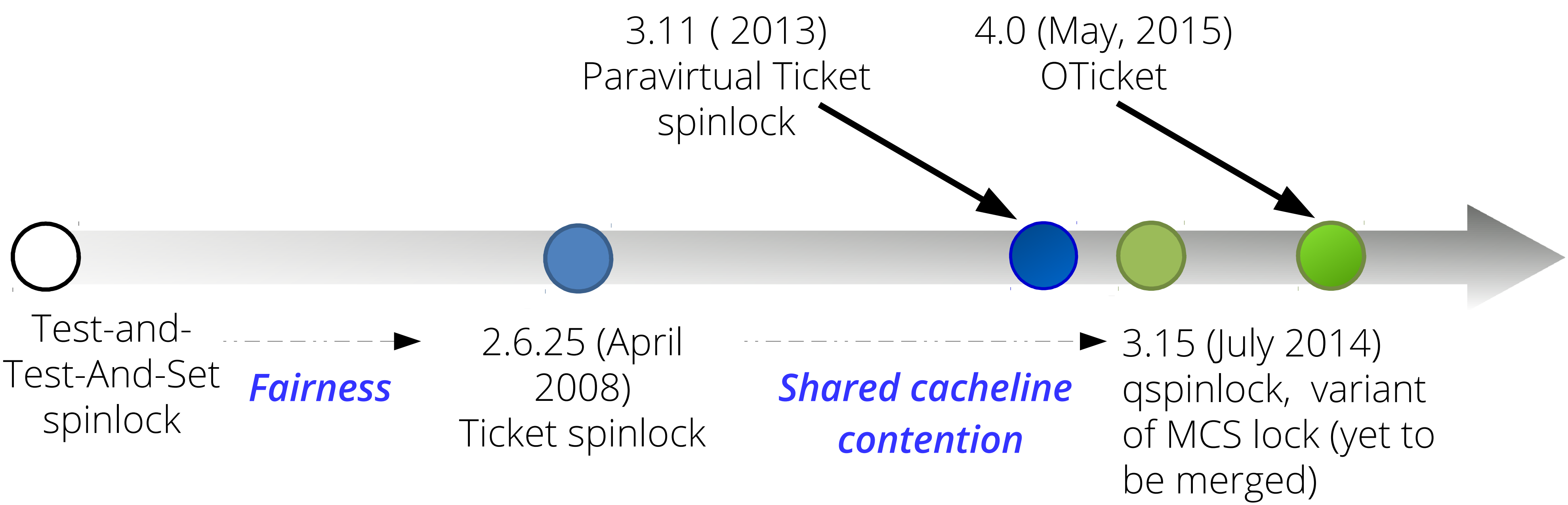
Spinlock Evolution in the Linux Kernel



Spinlock Evolution in the Linux Kernel



Spinlock Evolution in the Linux Kernel



Ticket Spinlock

```
#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++;
 }
```

```
F&I(*addr) {
        old = *addr;
        *addr++;
        return old;
}
```

head

tail

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

Ticket Spinlock

```
#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++;
}
```

```
F&I(*addr) {
        old = *addr;
        *addr++;
        return old;
}
```

head

tail

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

Ticket Spinlock

```
#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++;
}
```

```
F&I(*addr) {
        old = *addr;
        *addr++;
        return old;
}
```

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

Ticket Spinlock

```
#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++;
}
```

```
F&I(*addr) {
        old = *addr;
        *addr++;
        return old;
}
```

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

Ticket Spinlock

```
#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++;
}
```

```
F&I(*addr) {
        old = *addr;
        *addr++;
        return old;
}
```

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

Ticket Spinlock

```
#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++;
}
```

```
F&I(*addr) {
    old = *addr;
    *addr++;
    return old;
}
```

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

Ticket Spinlock

```
#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++;
}
```

```
F&I(*addr) {
        old = *addr;
        *addr++;
        return old;
}
```

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

Ticket Spinlock

```
#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++;
}
```

```
F&I(*addr) {
        old = *addr;
        *addr++;
        return old;
}
```

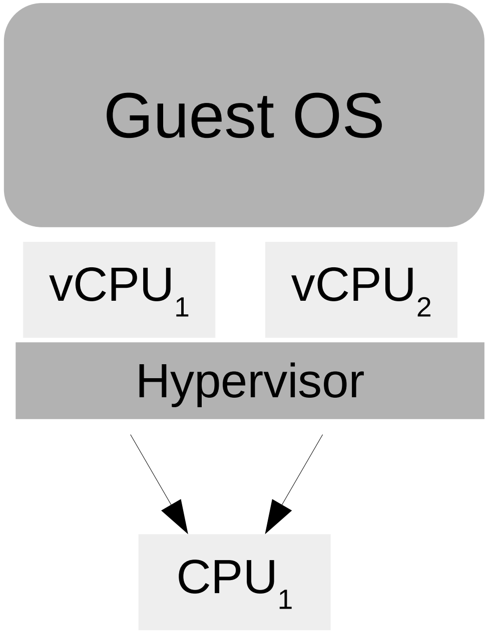
head

tail

- 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.

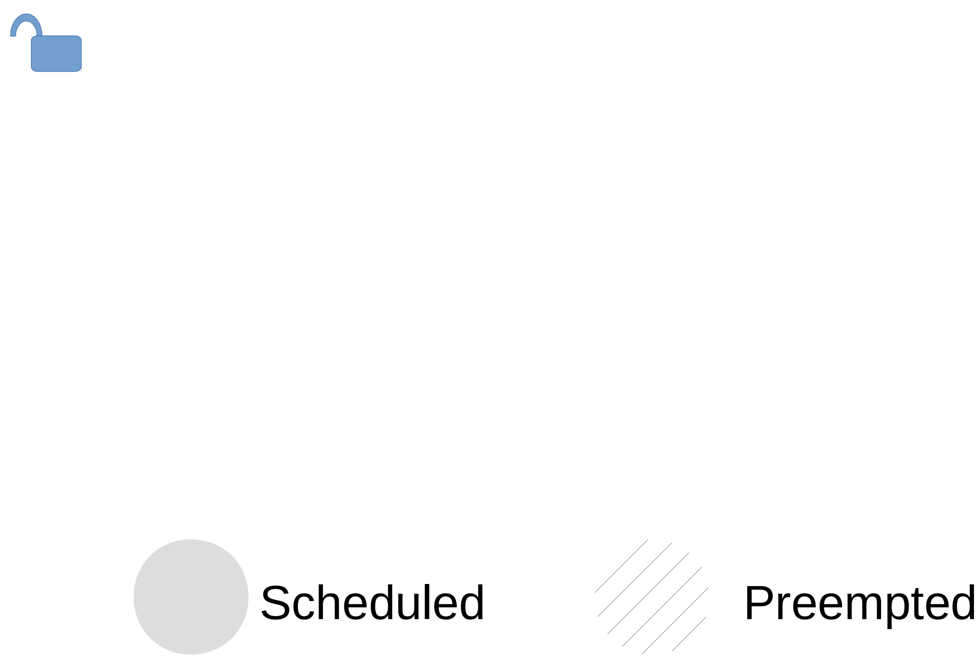


Complexity of Ticket Spinlock in Virtualized Environment

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

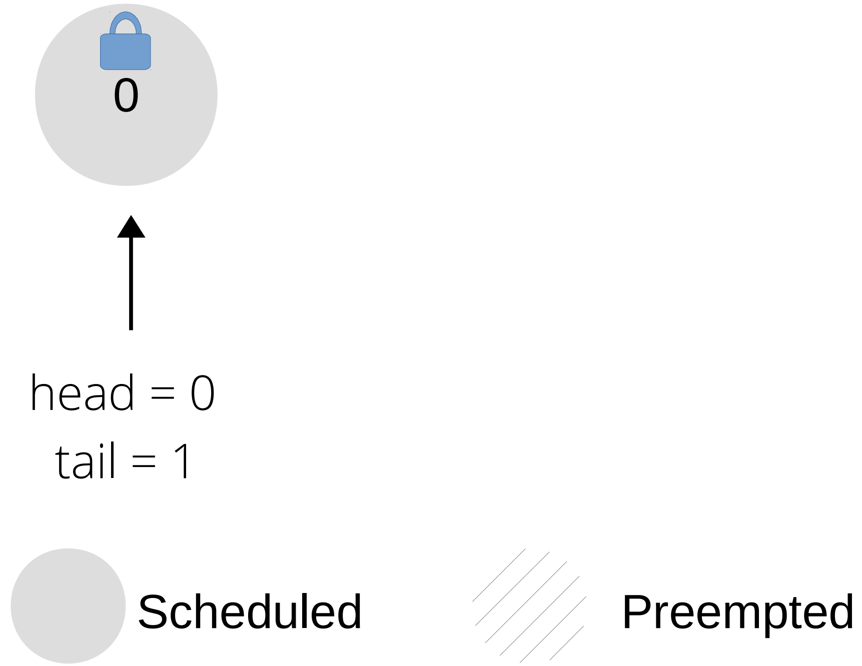
Complexity of Ticket Spinlock in Virtualized Environment

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



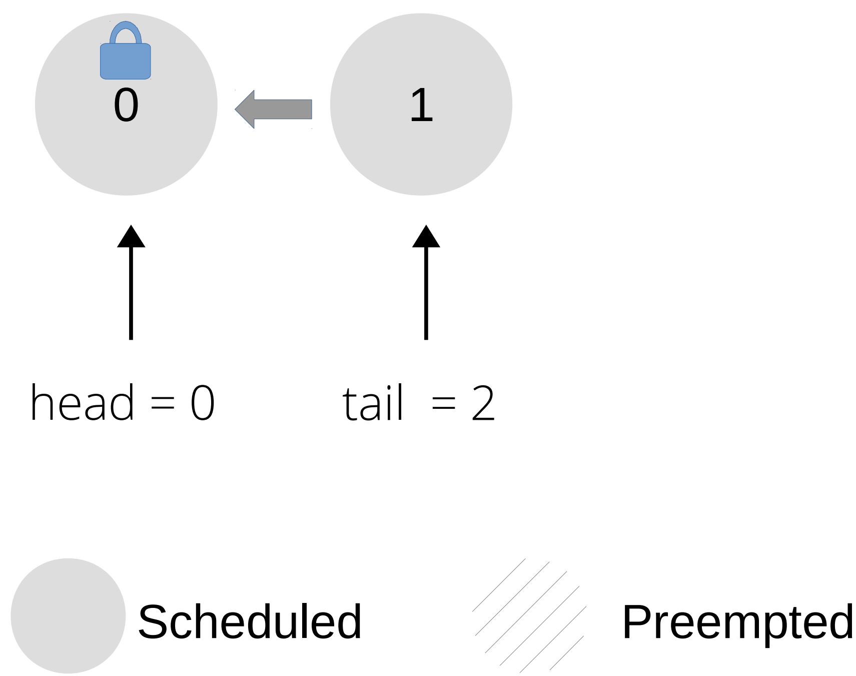
Complexity of Ticket Spinlock in Virtualized Environment

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



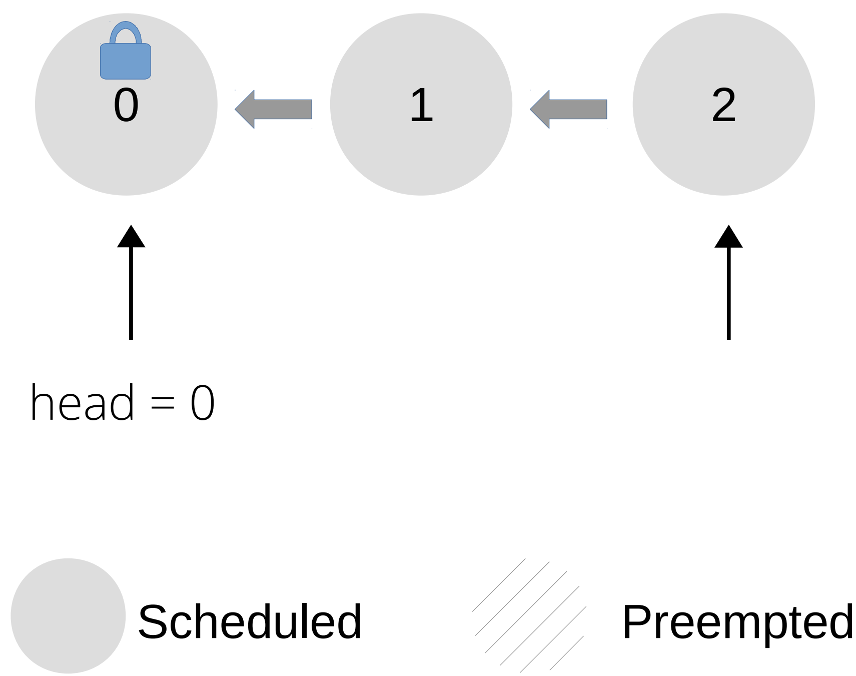
Complexity of Ticket Spinlock in Virtualized Environment

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



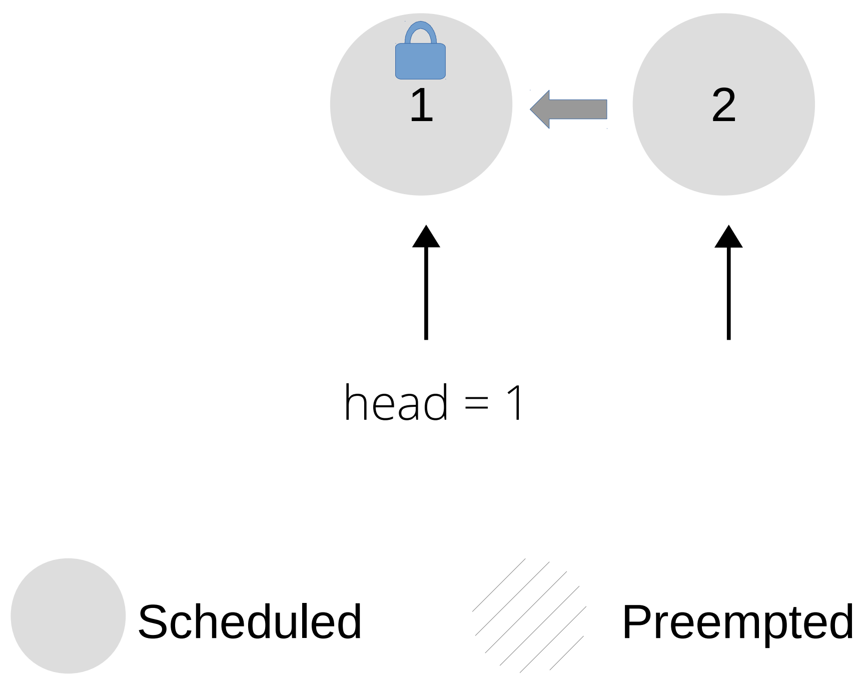
Complexity of Ticket Spinlock in Virtualized Environment

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



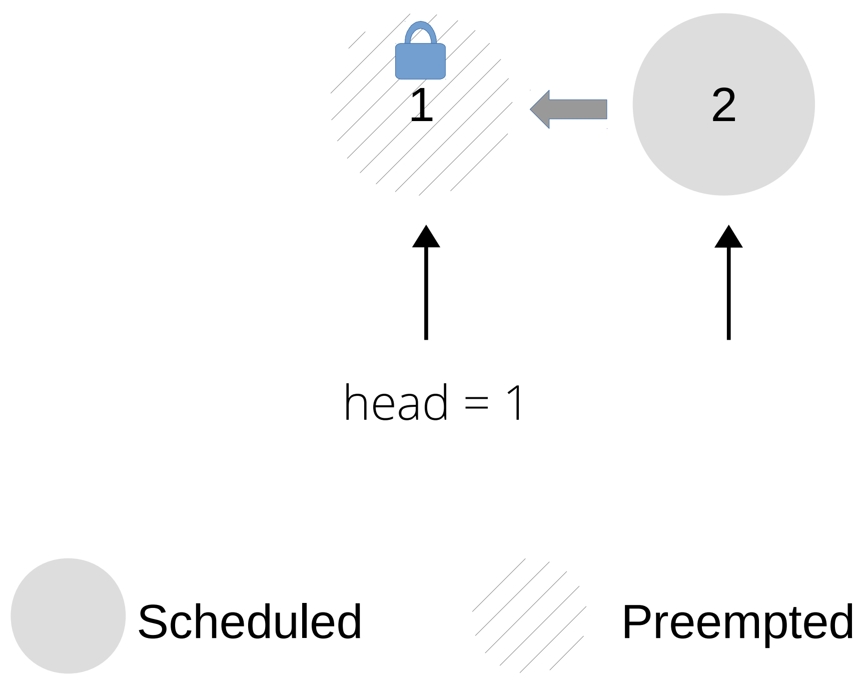
Complexity of Ticket Spinlock in Virtualized Environment

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



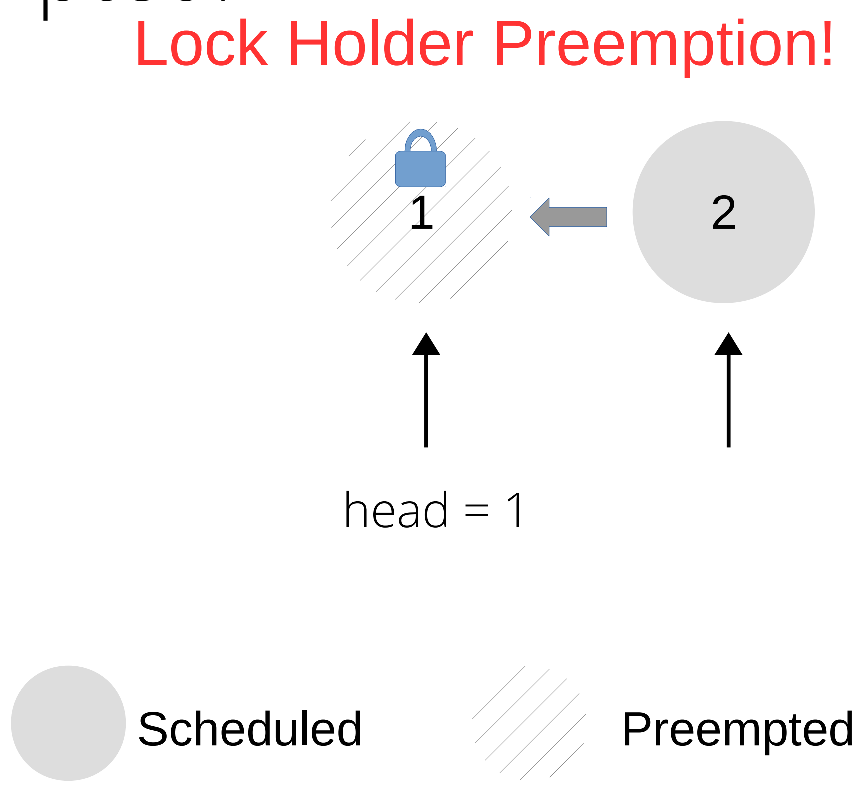
Complexity of Ticket Spinlock in Virtualized Environment

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



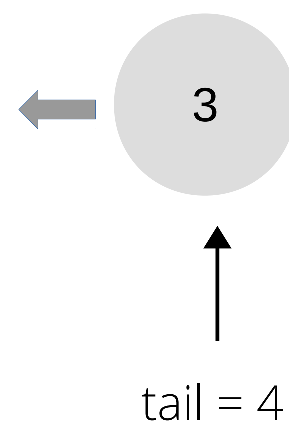
Complexity of Ticket Spinlock in Virtualized Environment

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



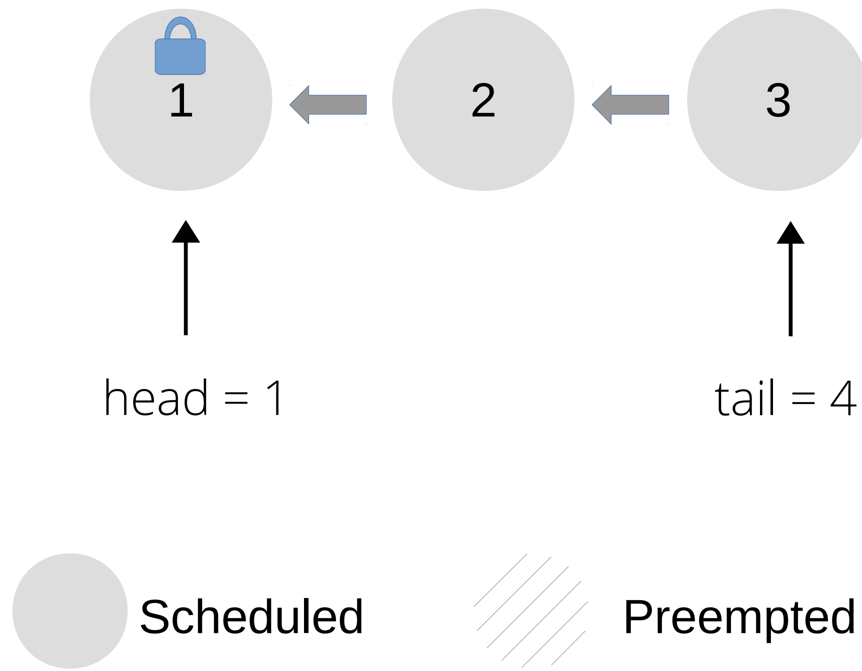
Complexity of Ticket Spinlock in Virtualized Environment

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



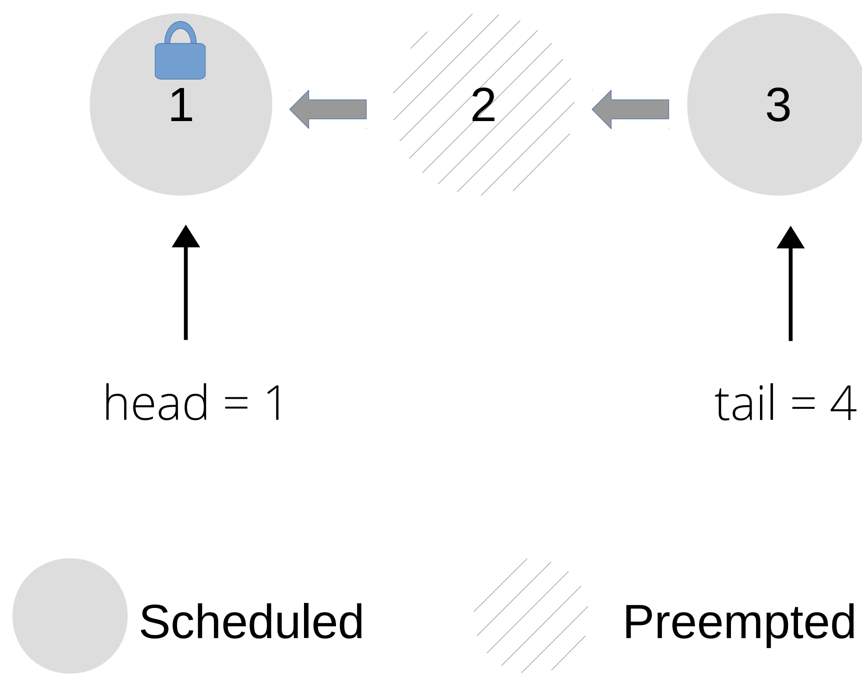
Complexity of Ticket Spinlock in Virtualized Environment

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



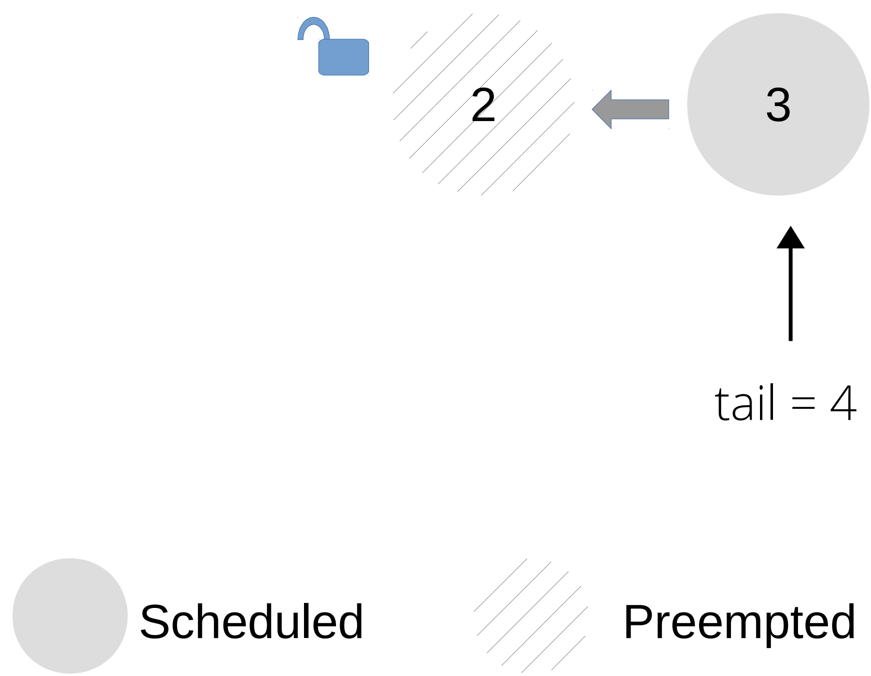
Complexity of Ticket Spinlock in Virtualized Environment

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



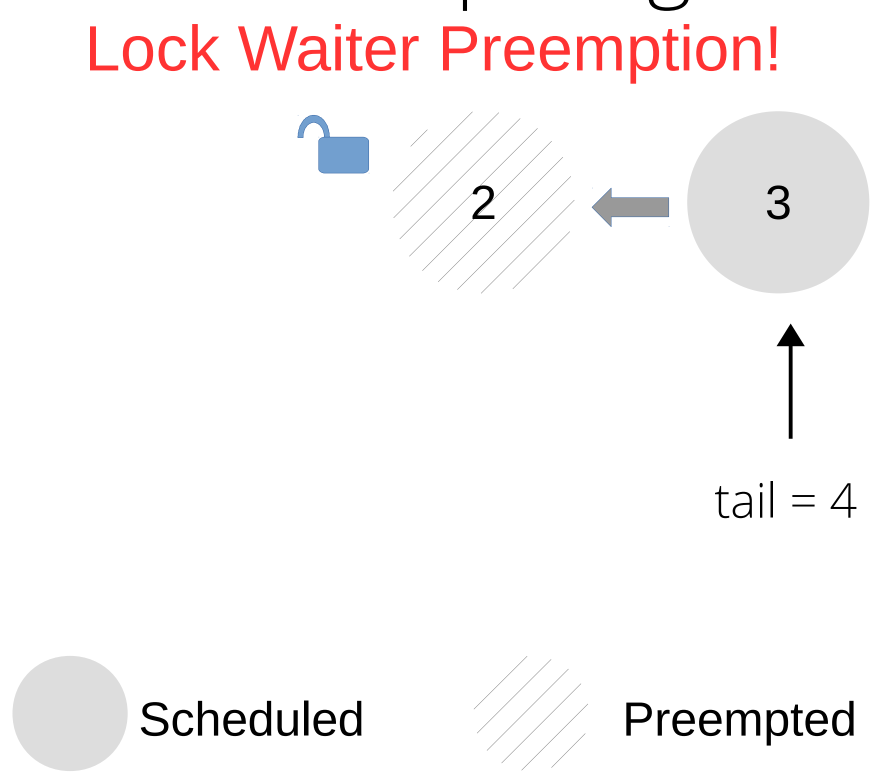
Complexity of Ticket Spinlock in Virtualized Environment

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



Complexity of Ticket Spinlock in Virtualized Environment

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



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.

```
#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.

```
#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.

```
#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.

```
#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.

```
#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.

```
#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.

```
#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.

```
#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.

```
#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.

```
#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.

```
#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++;
 }
```

Problem: The Mechanism to Annotate the Slow Behavior

The **slowpath_spin** issues the hlt instruction

↓

The hypervisor traps the instruction

↓

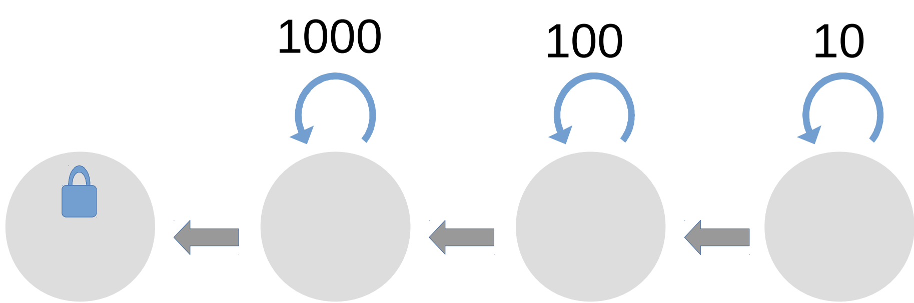
Then it de-schedules the vCPU.

- 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

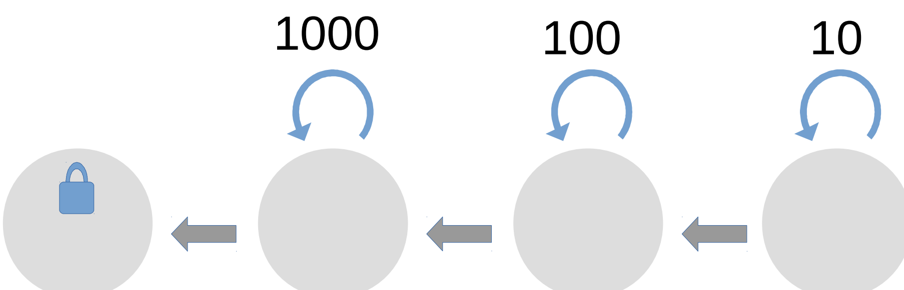
- OTicket tries to exploit the ordering.
- **Lock**:
  - Lower ticket distance → longer spin.
  - Allows more spinning to nearby waiters.
- **Unlock**:
  - Wake-up multiple waiters.
  - Reduces latency for the upcoming waiters.

OTicket: Opportunistic Spinning



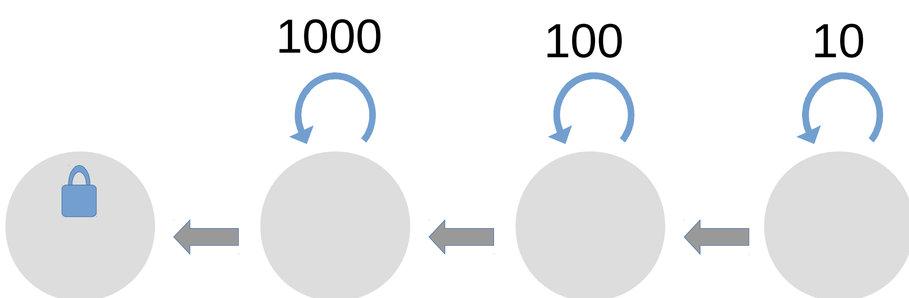

```
+#define EAGER_WAITERS   4
+#define TICKET_QUEUE    18
+#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD  15
 int head = 0;
 int tail = 0;
+ u64 threshold = SPIN_THRESHOLD;
 void lock() {
        my_ticket = F&I(tail);
+       if(my_ticket - head < TICKET_QUEUE) {
+               threshold = 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



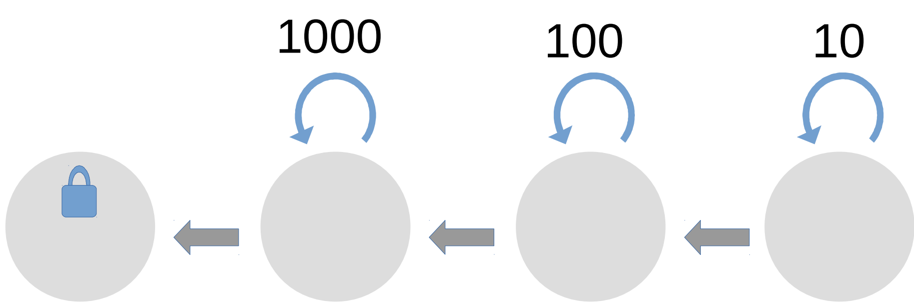

```
+#define EAGER_WAITERS   4
+#define TICKET_QUEUE    18
+#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD  15
 int head = 0;
 int tail = 0;
+ u64 threshold = SPIN_THRESHOLD;
 void lock() {
        my_ticket = F&I(tail);
+       if(my_ticket - head < TICKET_QUEUE) {
+               threshold = 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



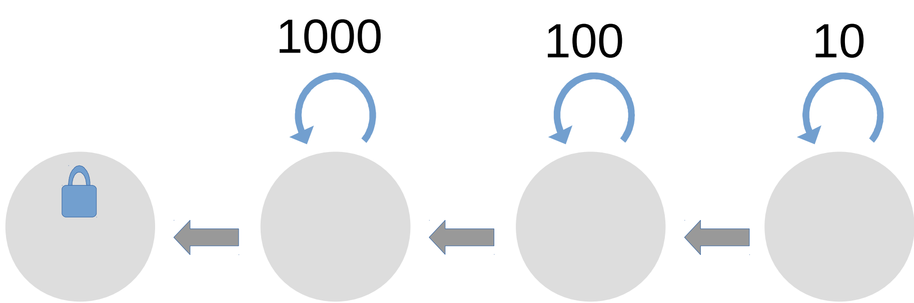

```
+#define EAGER_WAITERS   4
+#define TICKET_QUEUE     18
+#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD   15
 int head = 0;
 int tail = 0;
+ u64 threshold = SPIN_THRESHOLD;
 void lock() {
        my_ticket = F&I(tail);
+       if(my_ticket - head < TICKET_QUEUE) {
+               threshold = 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



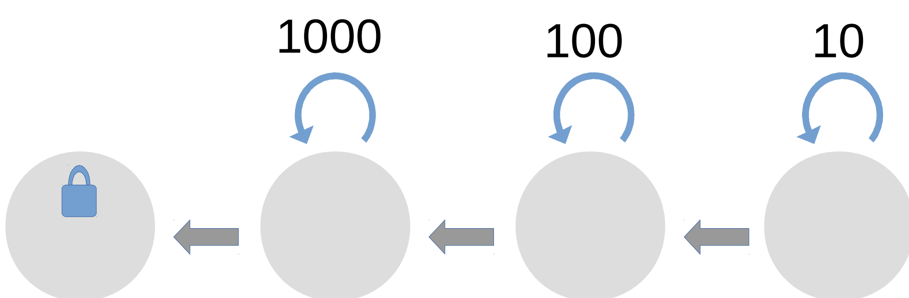

```
+#define EAGER_WAITERS   4
+#define TICKET_QUEUE     18
+#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD   15
 int head = 0;
 int tail = 0;
+ u64 threshold = SPIN_THRESHOLD;
 void lock() {
        my_ticket = F&I(tail);
+       if(my_ticket - head < TICKET_QUEUE) {
+               threshold = 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



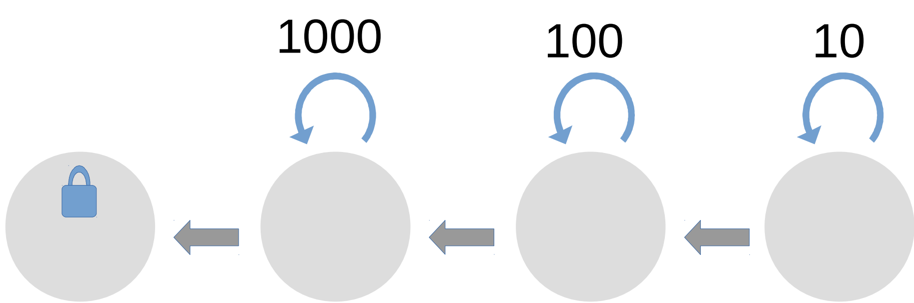

```
+#define EAGER_WAITERS   4
+#define TICKET_QUEUE    18
+#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD   15
 int head = 0;
 int tail = 0;
+ u64 threshold = SPIN_THRESHOLD;
 void lock() {
        my_ticket = F&I(tail);
+       if(my_ticket - head < TICKET_QUEUE) {
+               threshold = 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



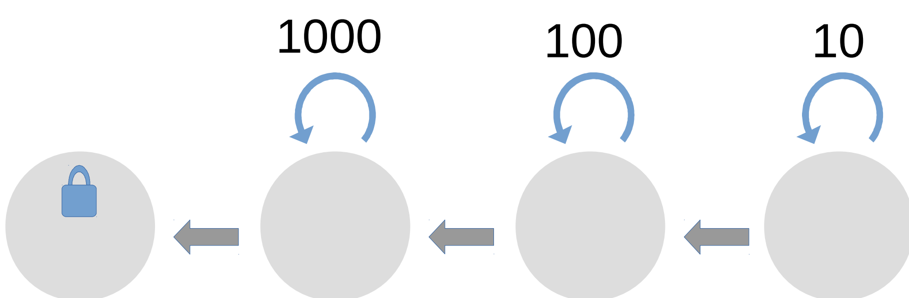

```
+#define EAGER_WAITERS   4
+#define TICKET_QUEUE    18
+#define SPIN_MAX_THRESHOLD 34
#define SPIN_THRESHOLD  15
 int head = 0;
 int tail = 0;
+ u64 threshold = SPIN_THRESHOLD;
 void lock() {
        my_ticket = F&I(tail);
+       if(my_ticket - head < TICKET_QUEUE) {
+               threshold = 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




```
+#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: ;
 }
```

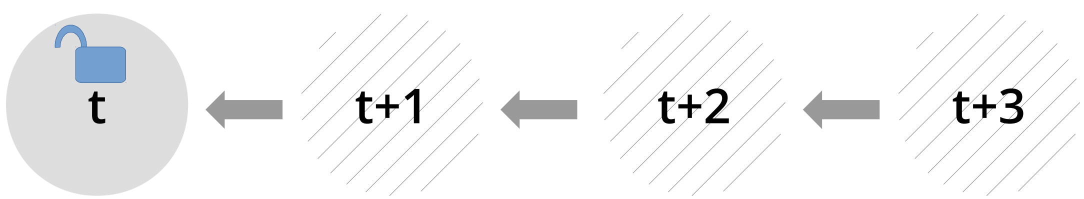
OTicket: Opportunistic Spinning




```
+#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 Wake-up

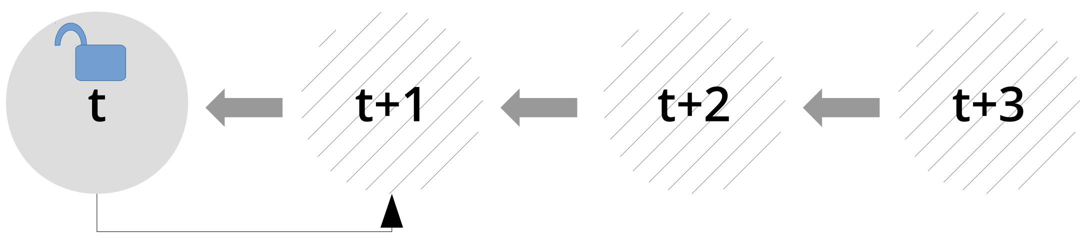
```
 void unlock() {
+      for(count = 1; count <= EAGER_WAITERS;
+                                          ++count) {
+            wakeup_cpu(head + count);
       }
       head++;
 }
```



Wake-up sleeping waiters

OTicket: Opportunistic Wake-up

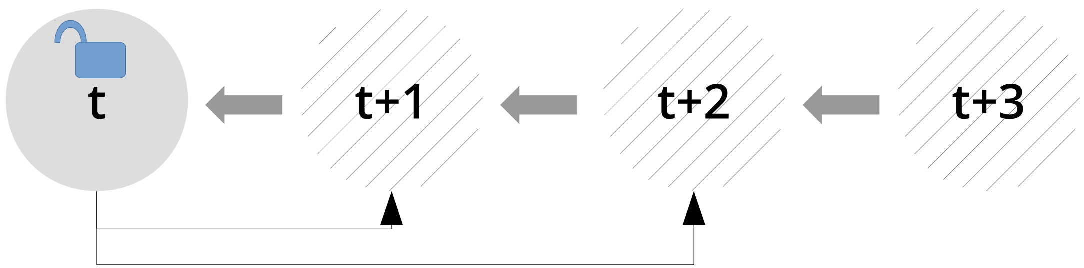
```
void unlock() {
+     for(count = 1; count <= EAGER_WAITERS;
+                                         ++count) {
+           wakeup_cpu(head + count);
      }
      head++;
}
```



Wake-up sleeping waiters

OTicket: Opportunistic Wake-up

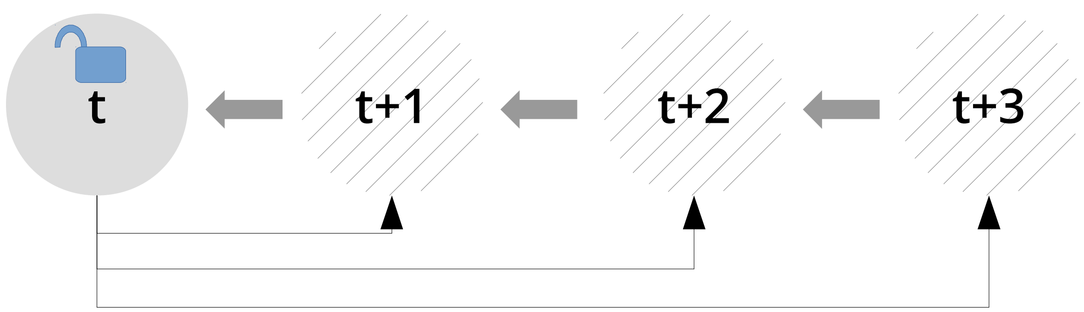
```
void unlock() {
+     for(count = 1; count <= EAGER_WAITERS;
+                                          ++count) {
+           wakeup_cpu(head + count);
      }
      head++;
}
```



Wake-up sleeping waiters

OTicket: Opportunistic Wake-up

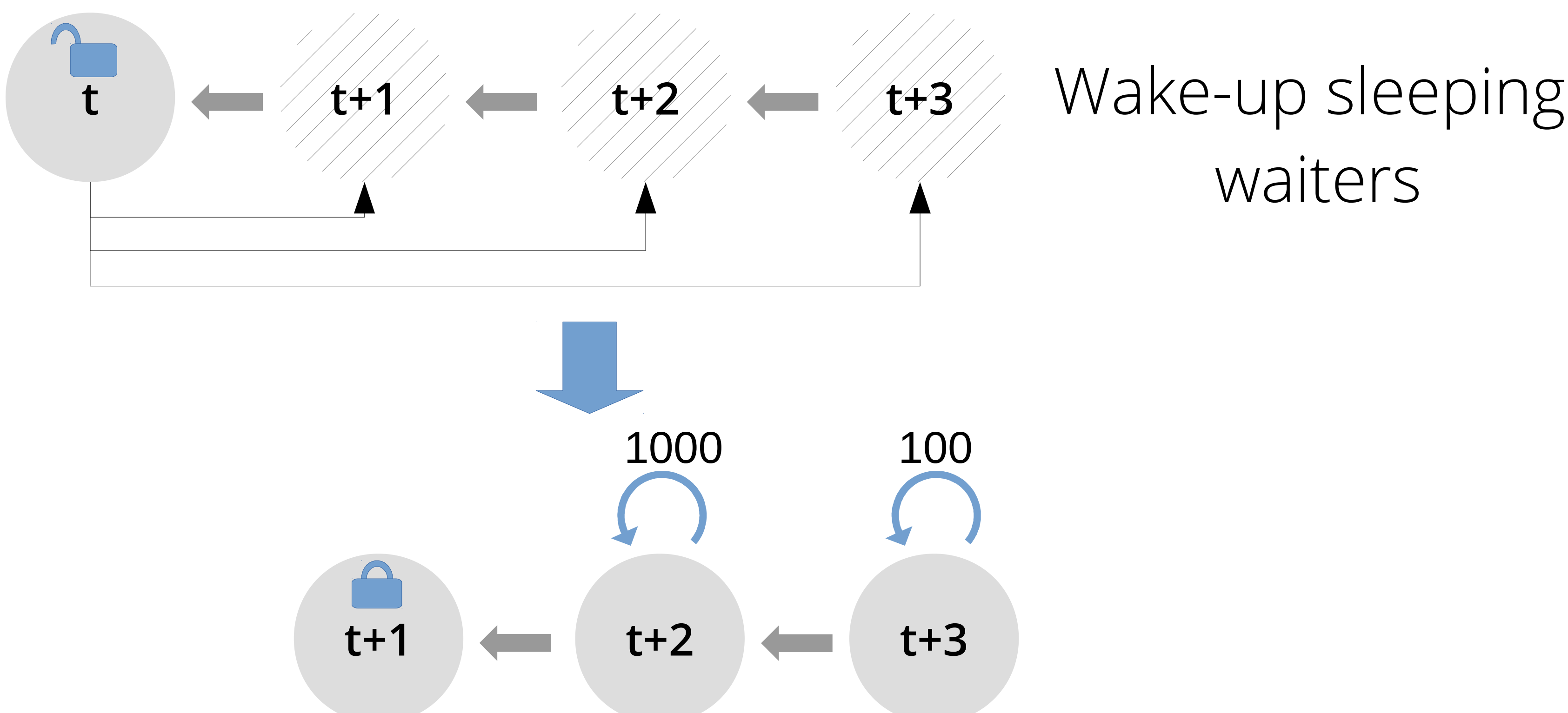
```
void unlock() {
+     for(count = 1; count <= EAGER_WAITERS;
+                                        ++count) {
+            wakeup_cpu(head + count);
      }
      head++;
}
```



Wake-up sleeping waiters

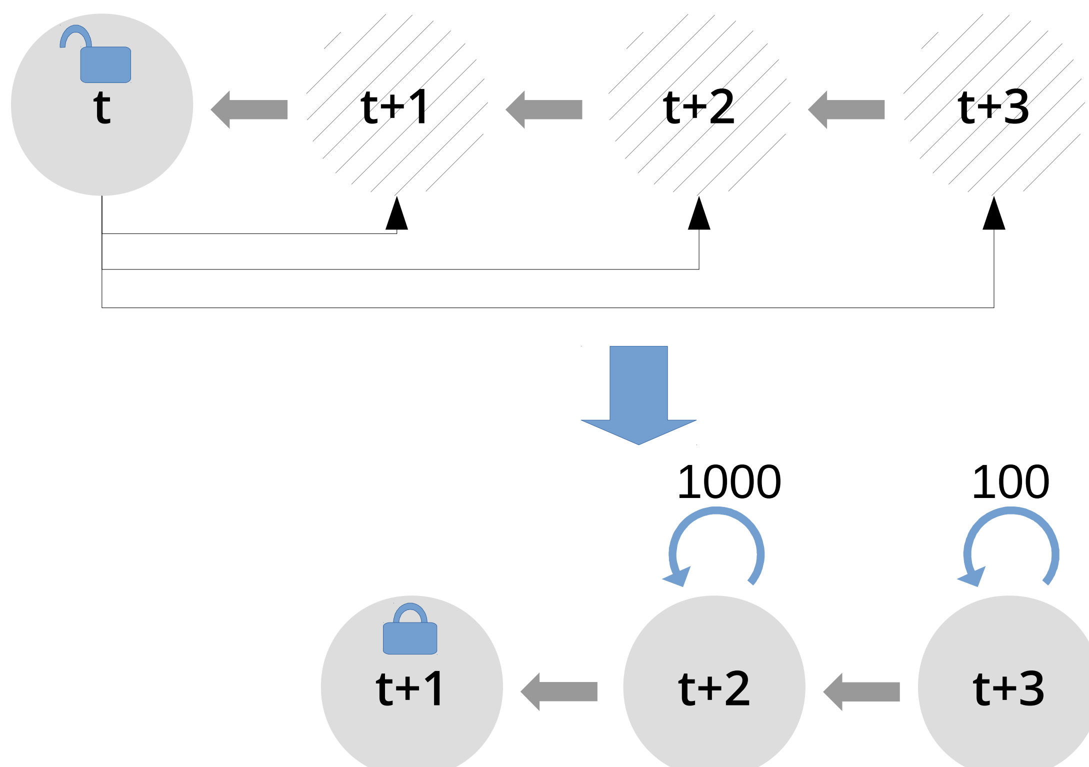
OTicket: Opportunistic Wake-up

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



OTicket: Opportunistic Wake-up

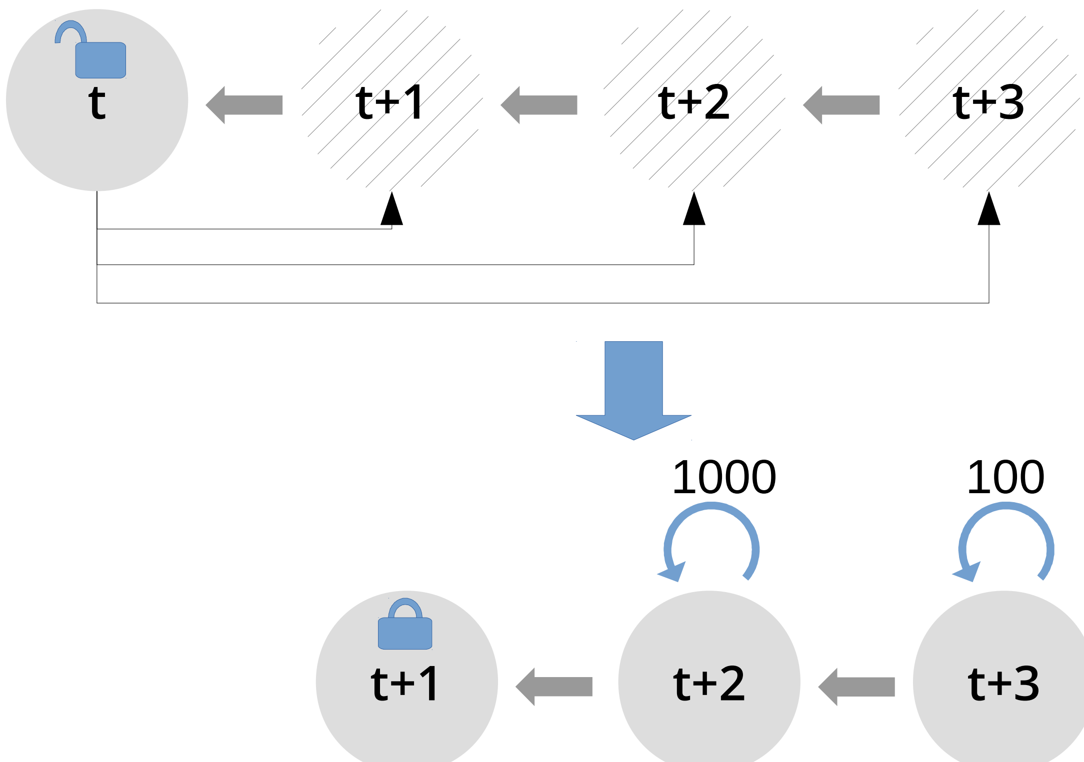
```
void unlock() {
+     for(count = 1; count <= EAGER_WAITERS;
+                                      ++count) {
+           wakeup_cpu(head + count);
      }
      head++;
}
```



Wake-up sleeping waiters

OTicket: Opportunistic Wake-up

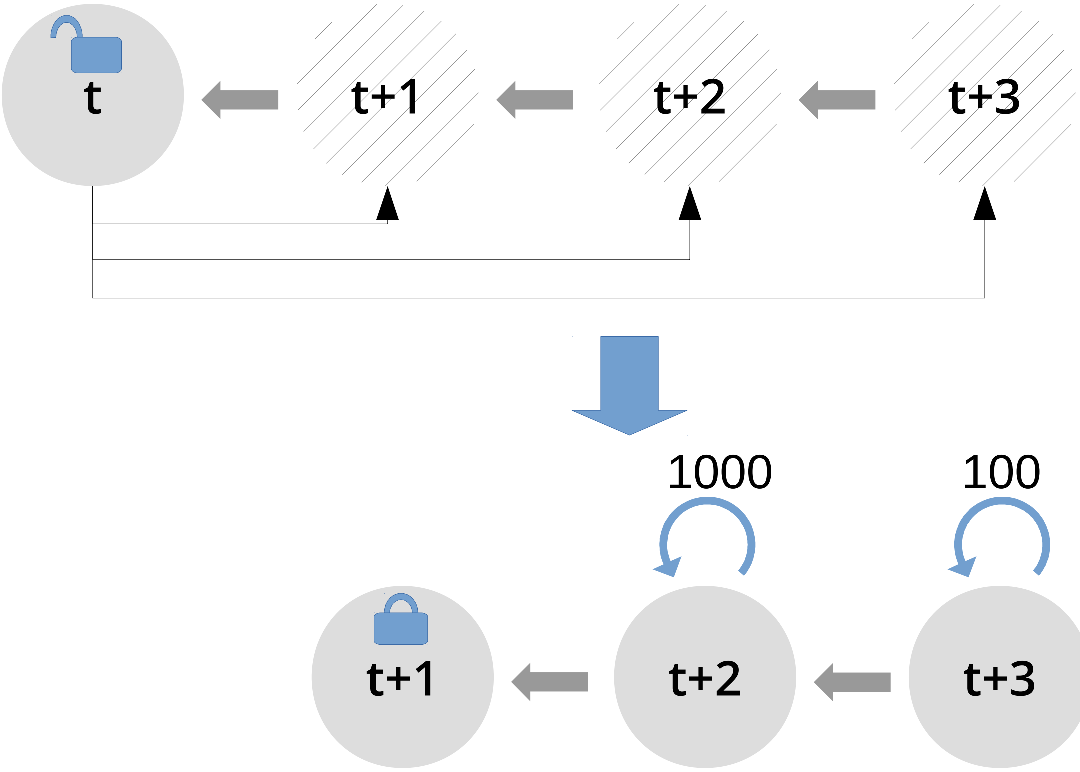
```
void unlock() {
+     for(count = 1; count <= EAGER_WAITERS;
+                                    ++count) {
+           wakeup_cpu(head + count);
      }
      head++;
}
```



Wake-up sleeping waiters

OTicket: Opportunistic Wake-up

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



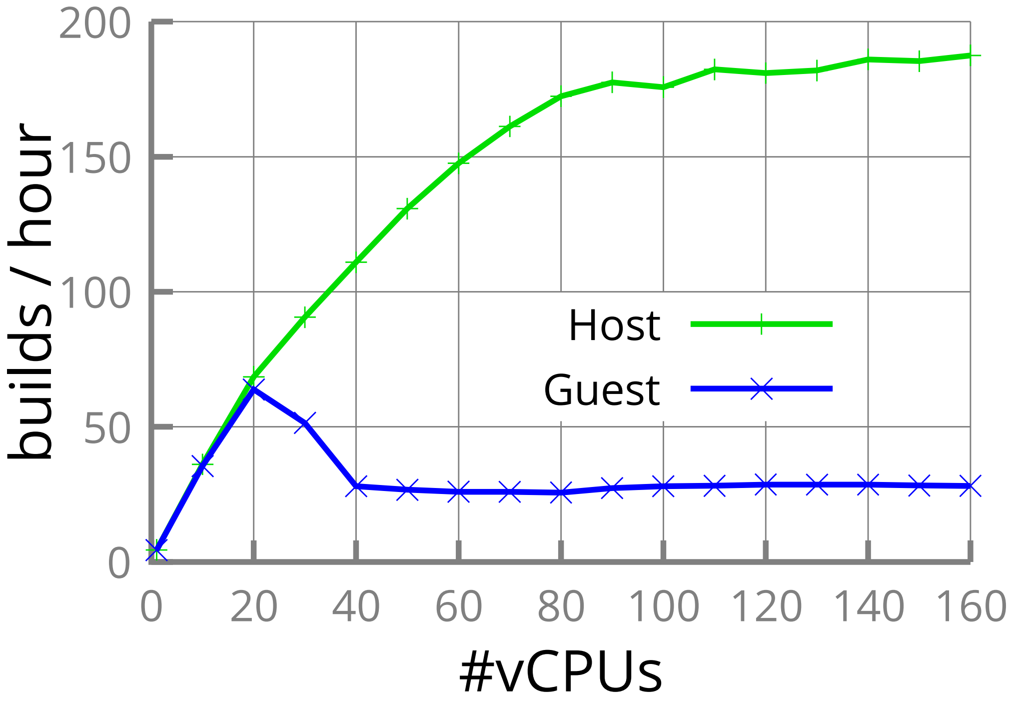
Wake-up sleeping waiters

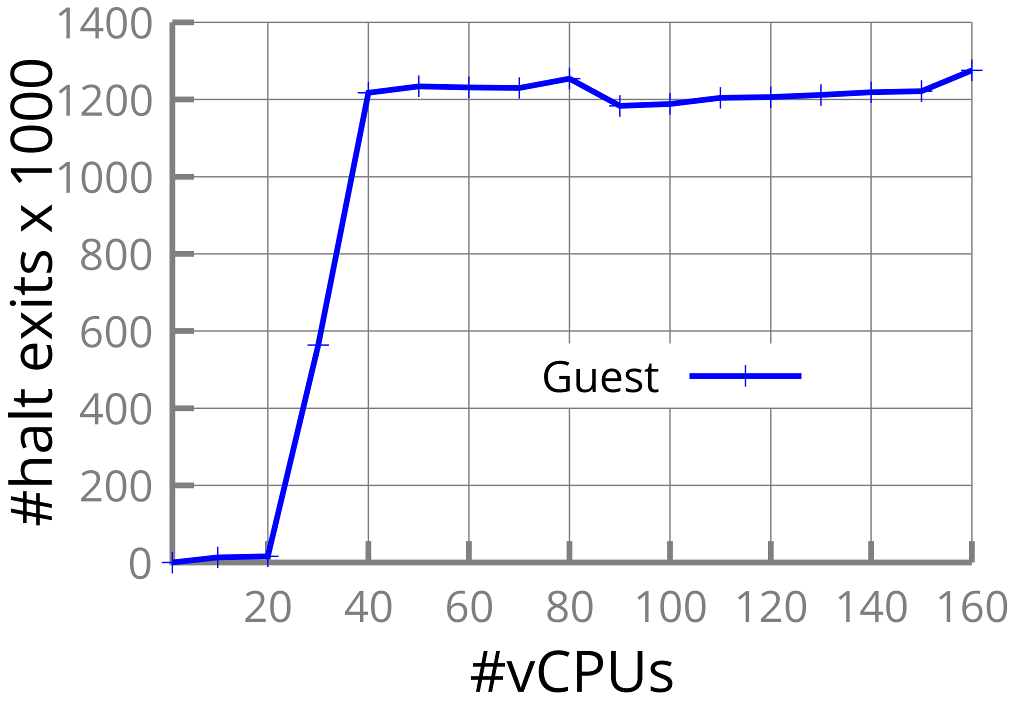
Outline

- Scalability issue in the Clouds
- Scalability issue in VMs with higher core count
- OTicket design
- **Evaluation**
- Conclusion

OTicket: Guest vs Host

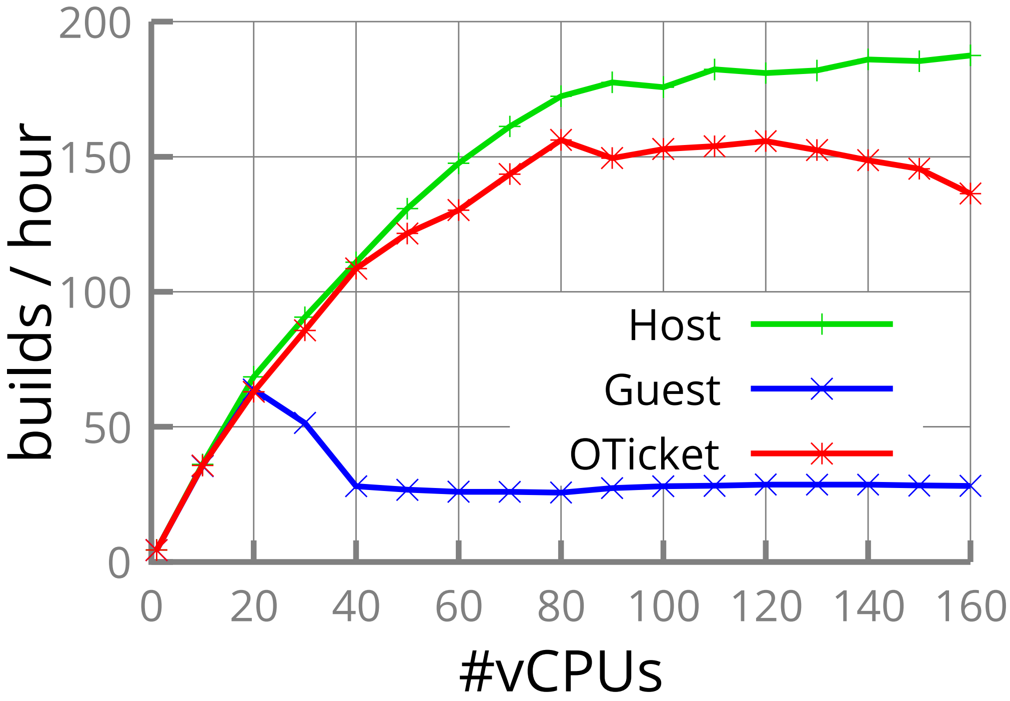
- Improves guest performance by almost **5x**.
- Reduces **halt exits** by 6x.

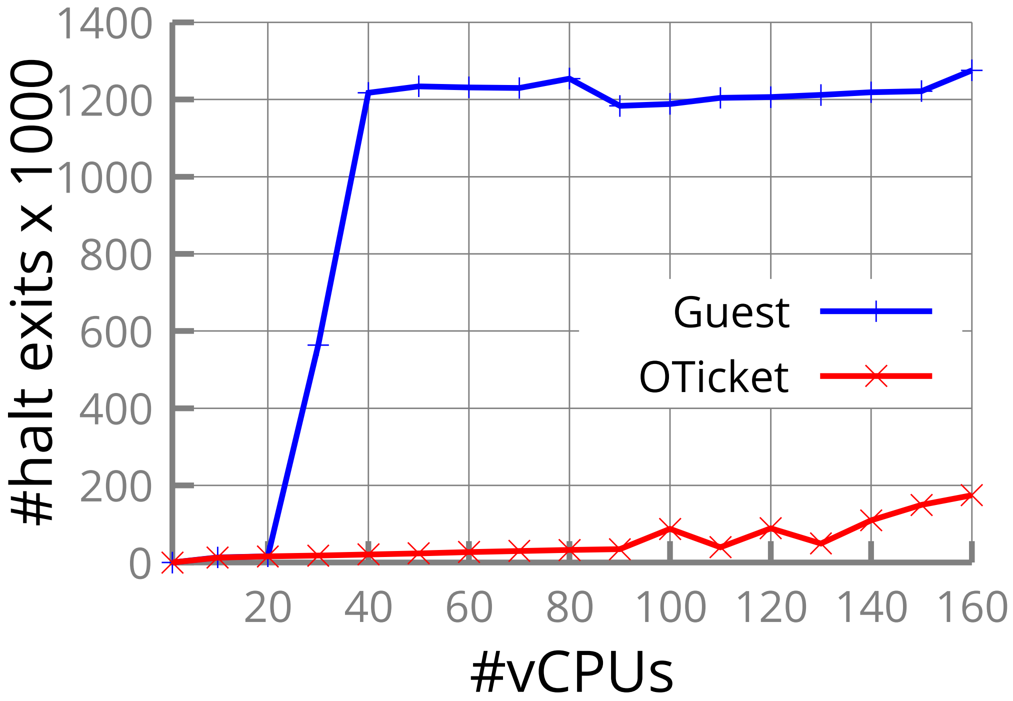




OTicket: Guest vs Host

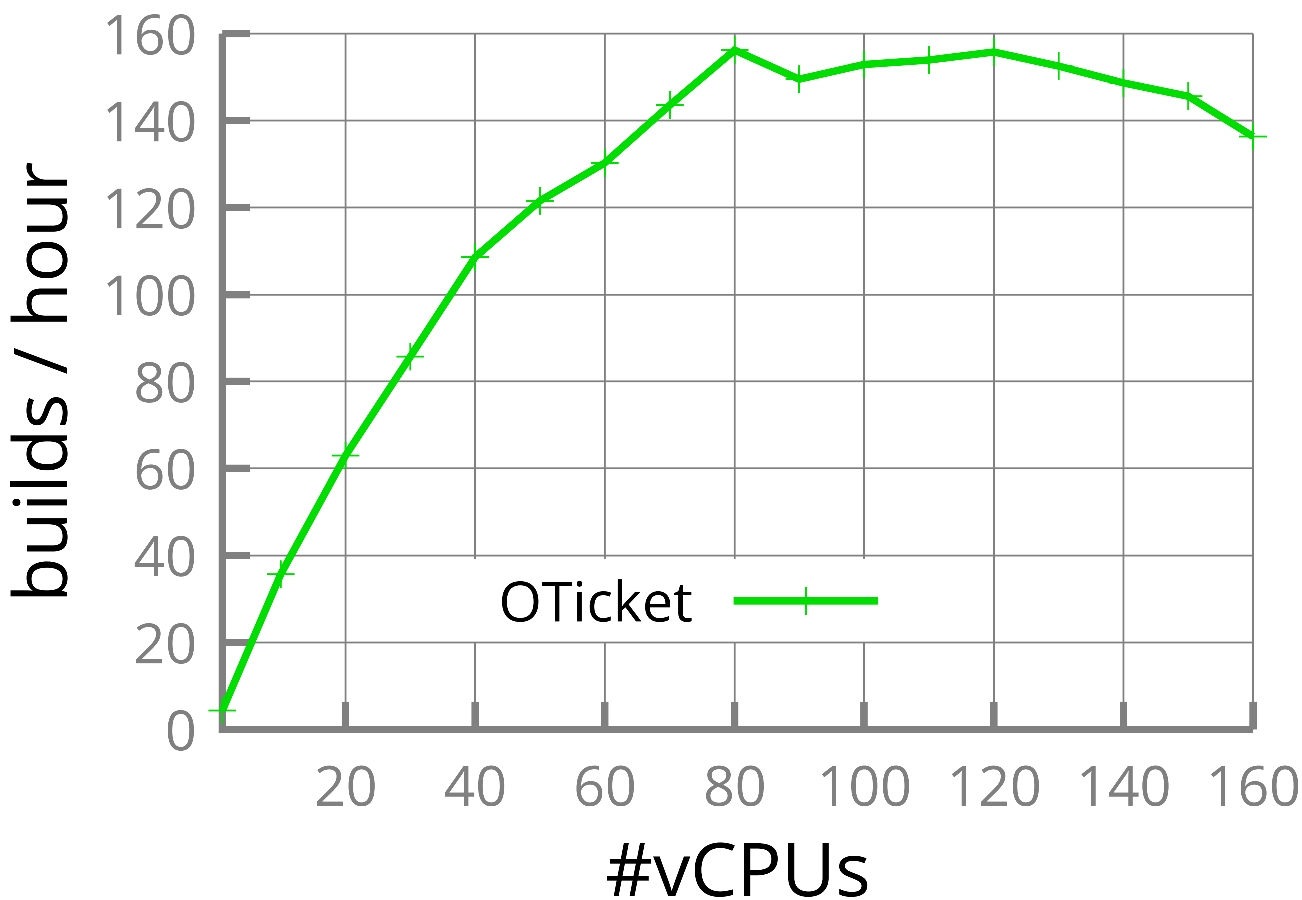
- Improves guest performance by almost **5x**.
- Reduces **halt exits** by 6x.





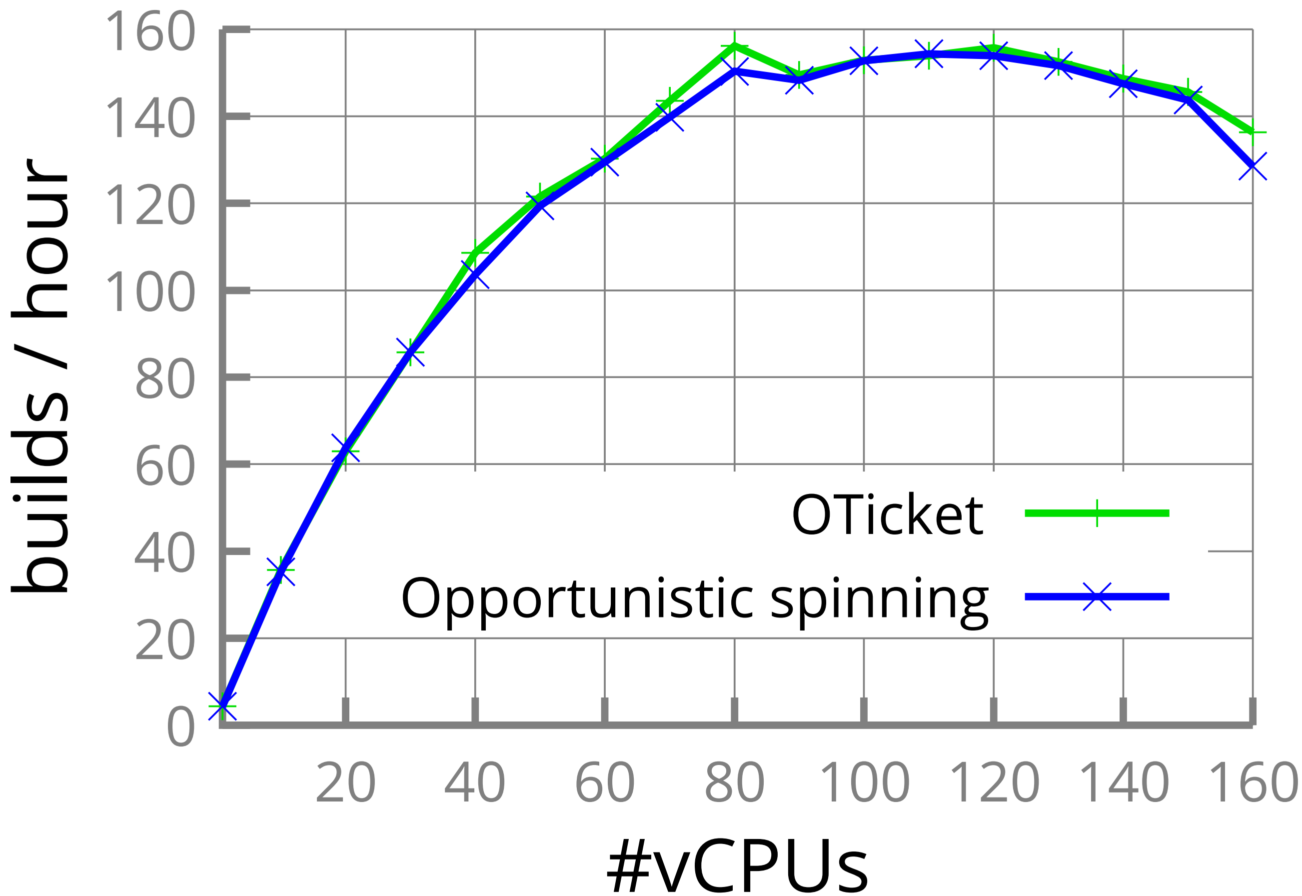
OTicket Performance Breakdown

- Opportunistic spinning prohibits sleeping.



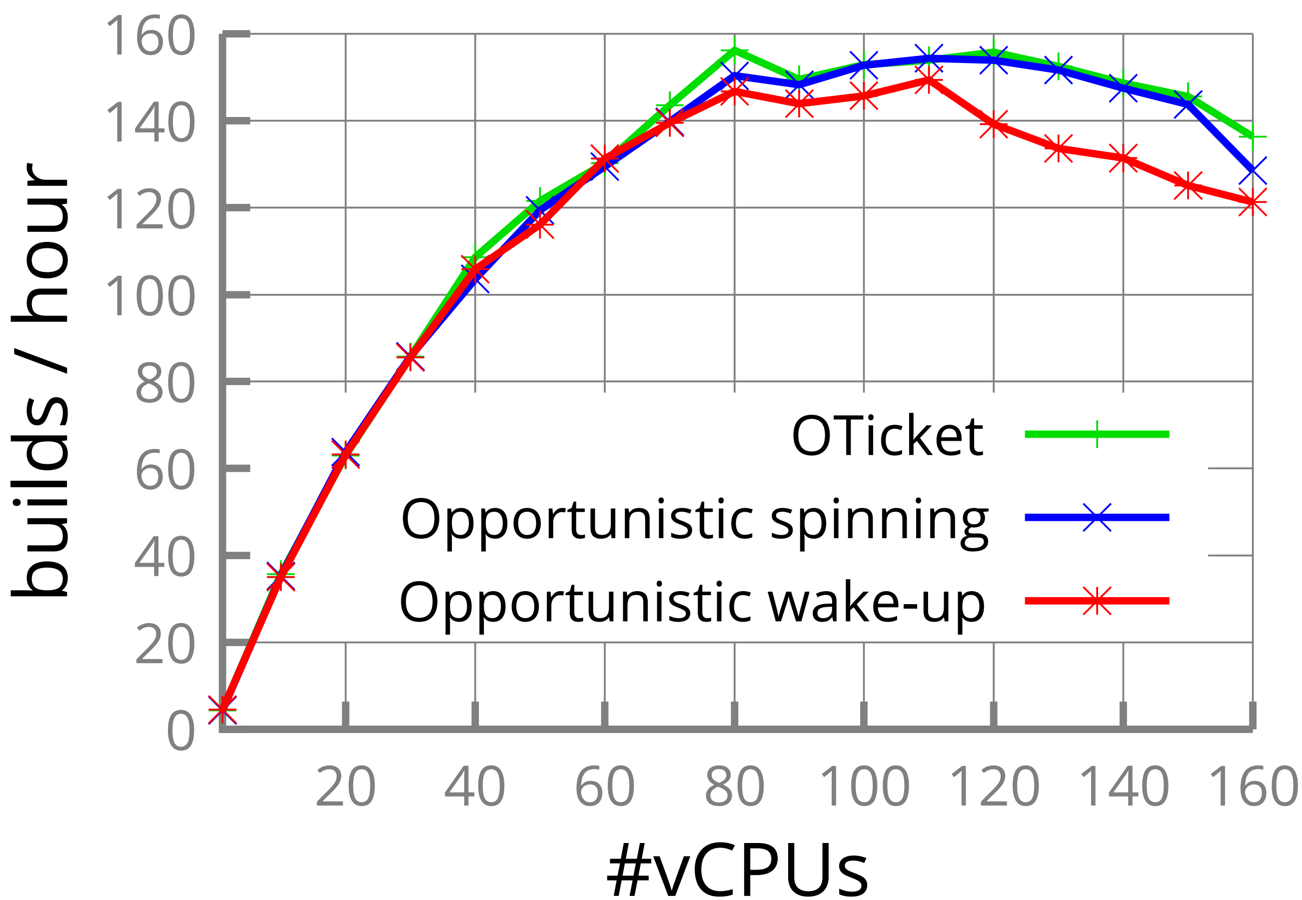
OTicket Performance Breakdown

- Opportunistic spinning prohibits sleeping.



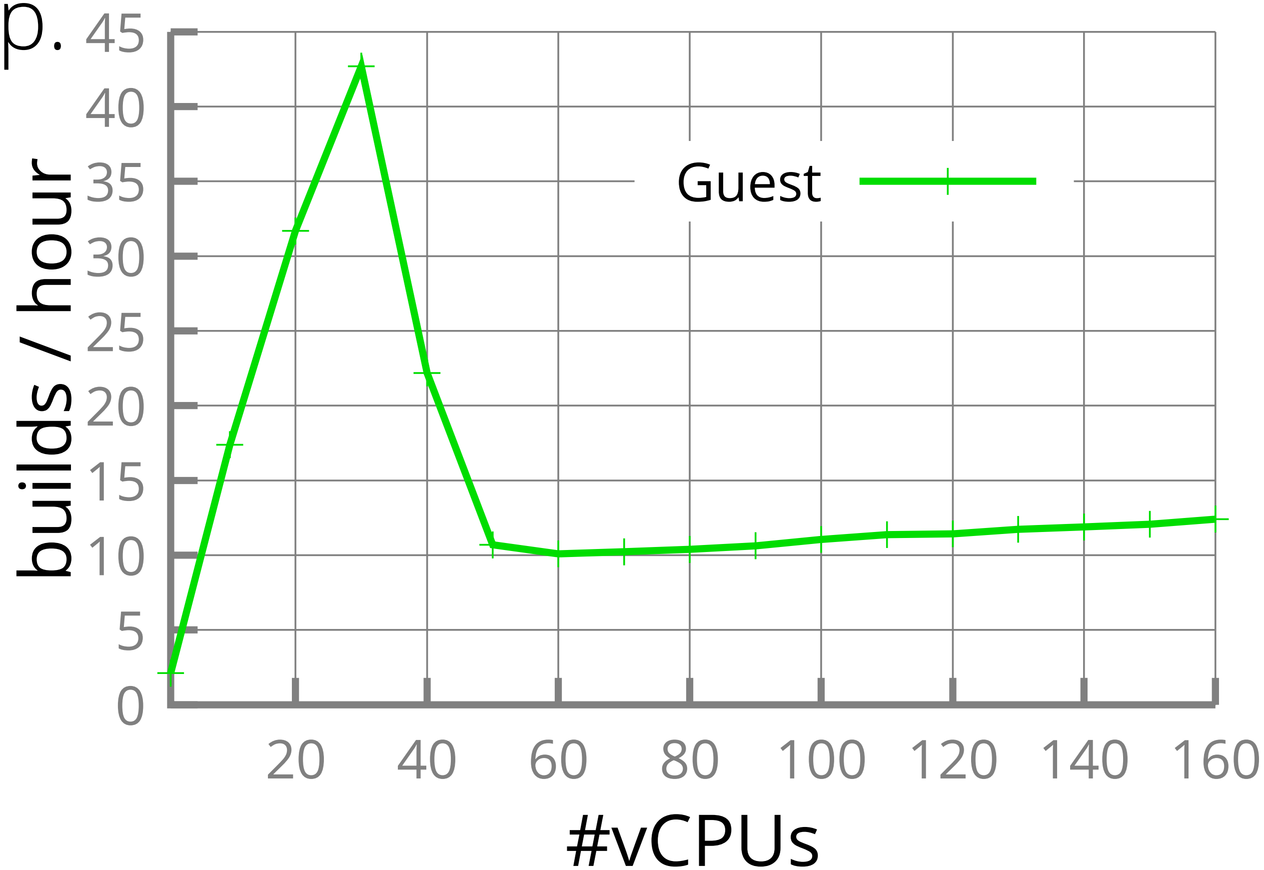
OTicket Performance Breakdown

- Opportunistic spinning prohibits sleeping.



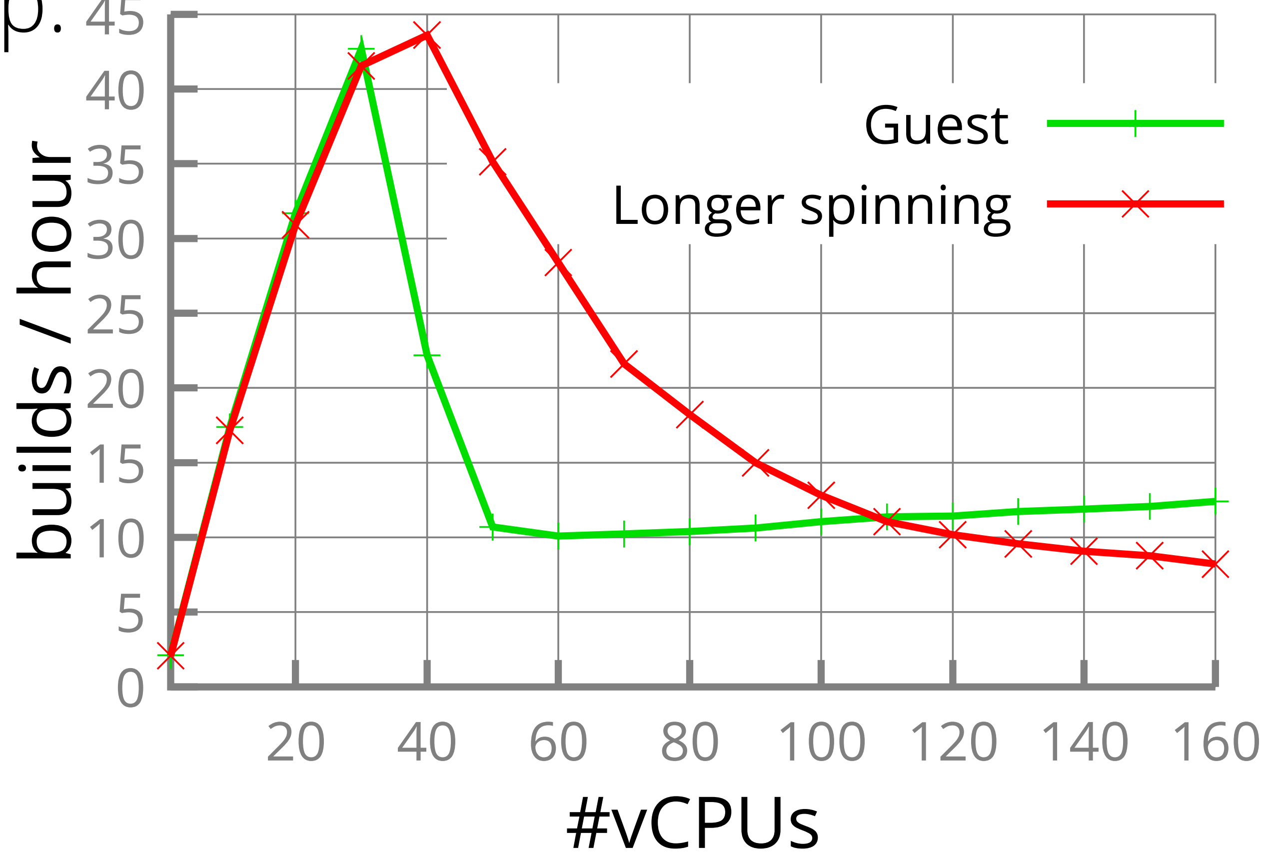
Importance of Wake-ups

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



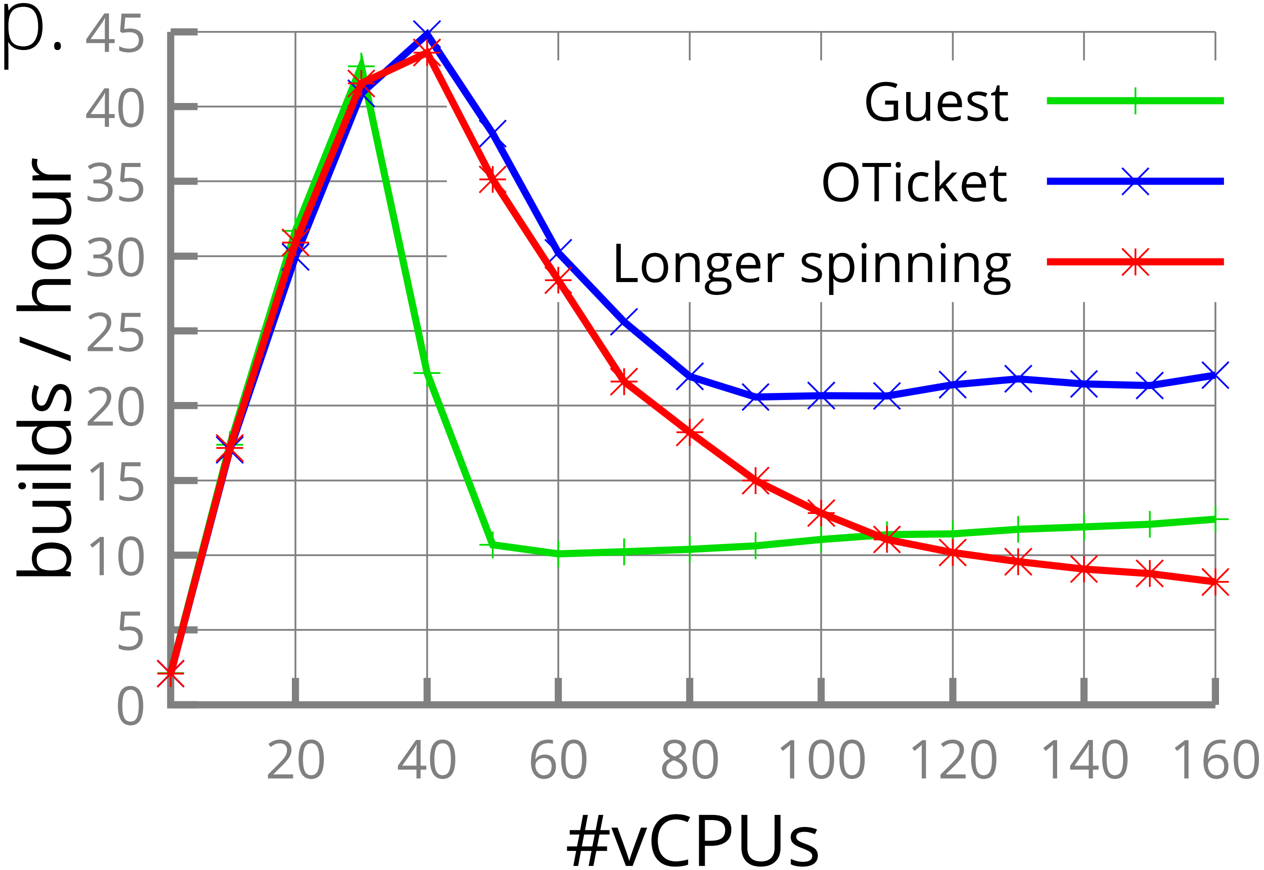
Importance of Wake-ups

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



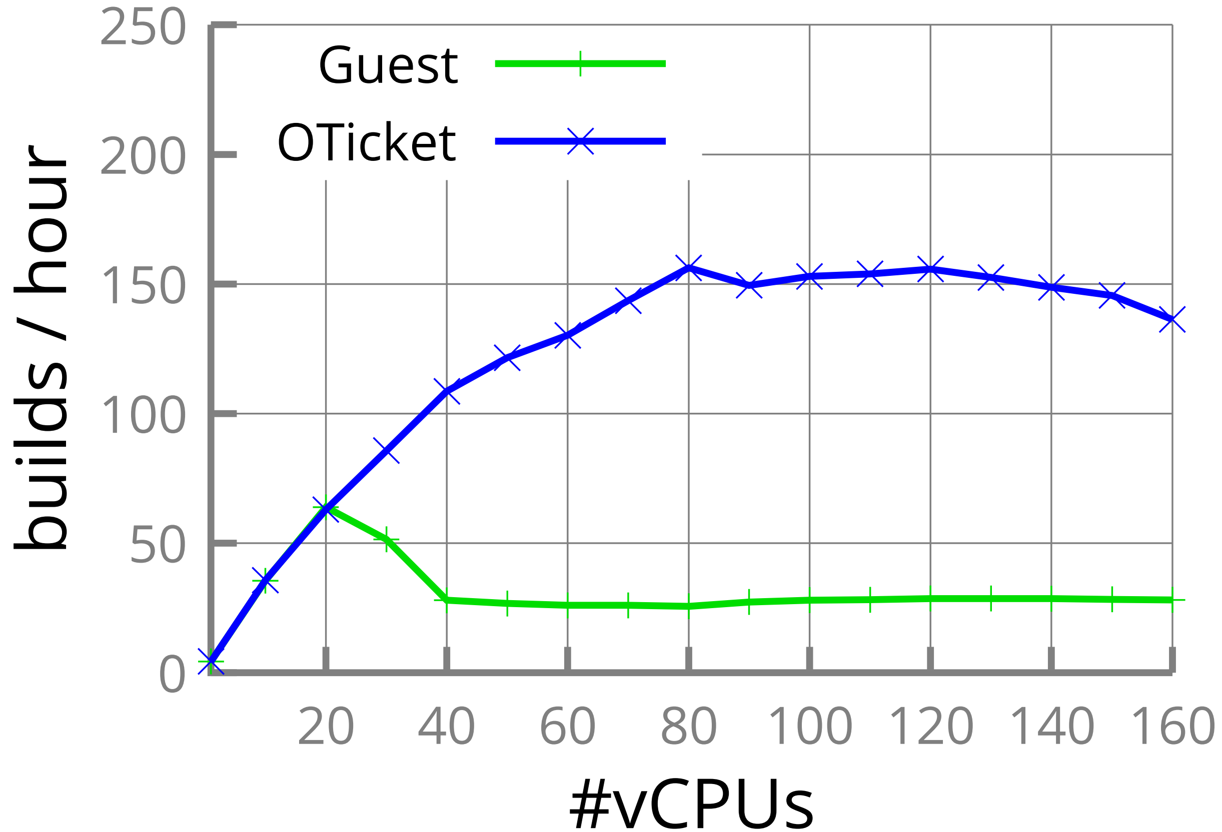
Importance of Wake-ups

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



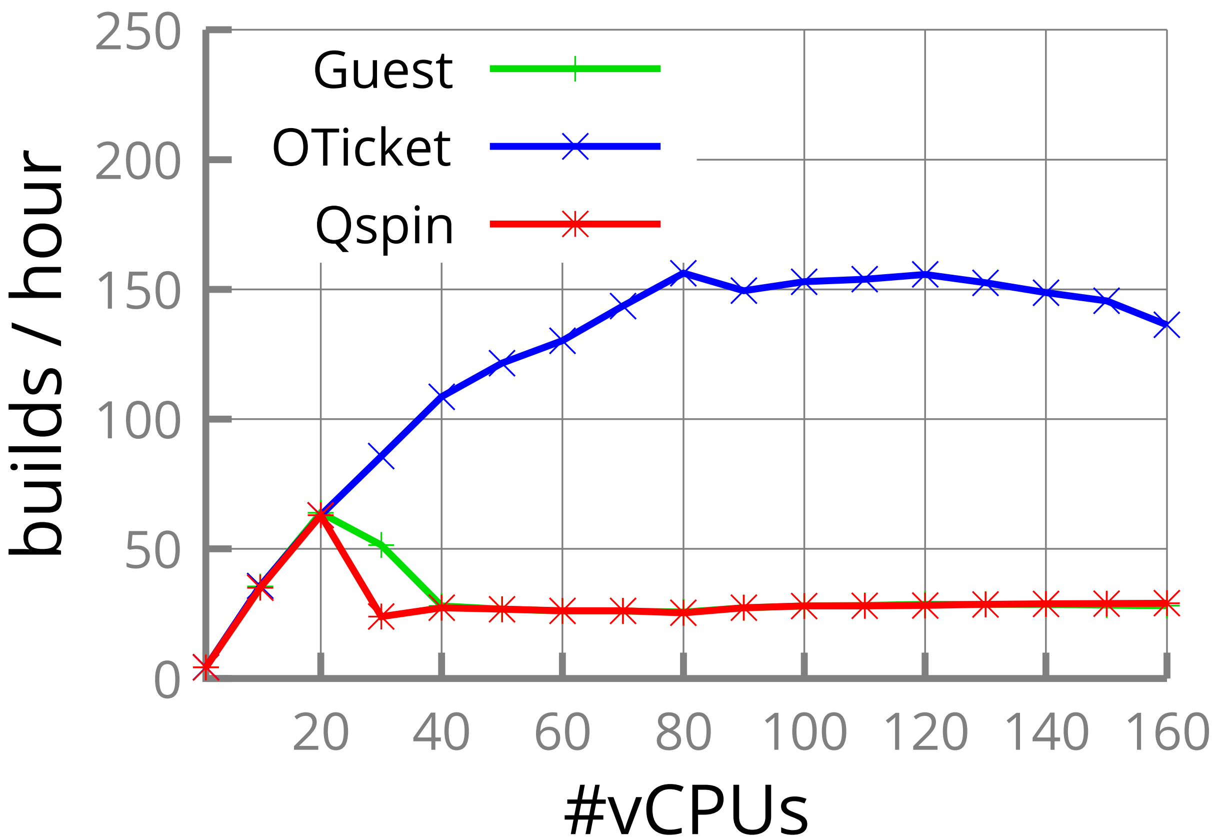
Other Spinlock Alternatives

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



Other Spinlock Alternatives

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

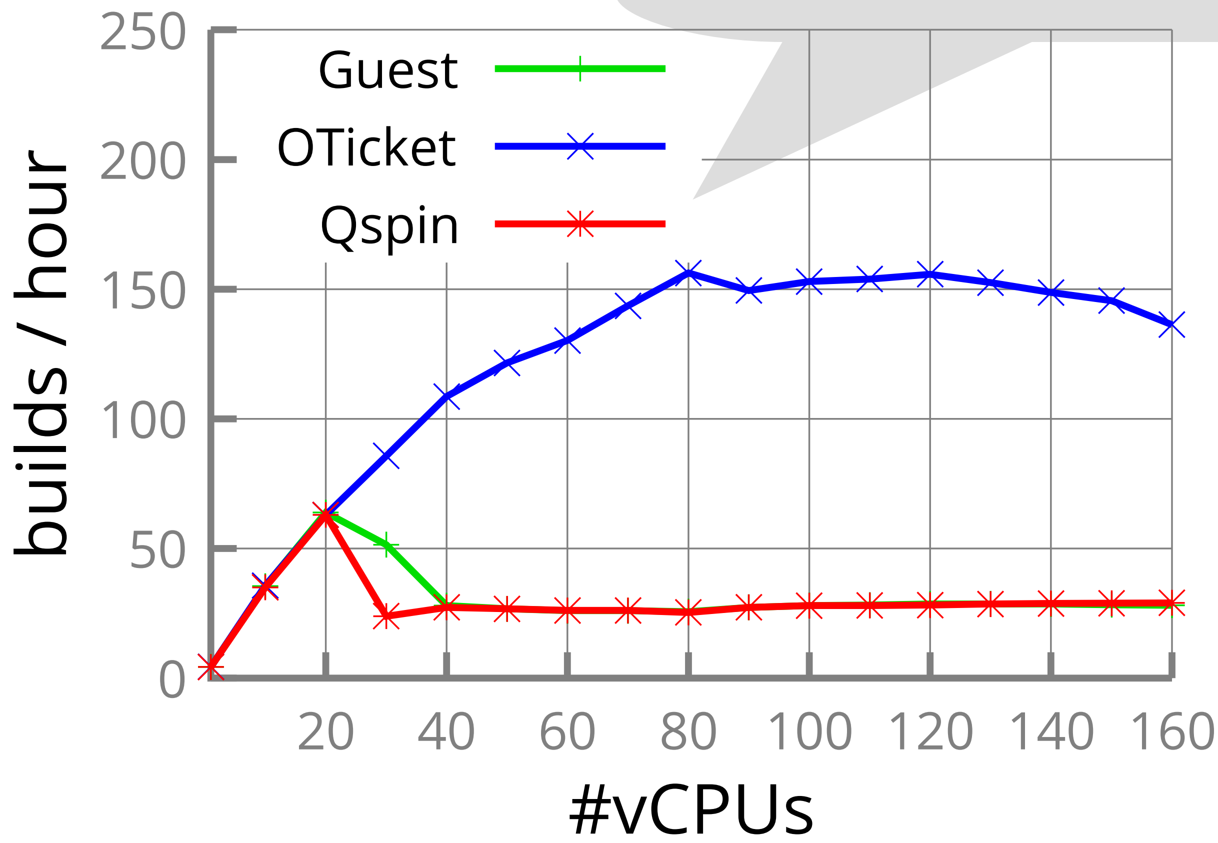


Other Spinlock Alternatives

- Two spinlock implementaions:
  - 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.

Conclusion

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 16:36:51 -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

Logged in as sanidhya
My Account
Unread comments
Log out

Weekly Edition
Return to the Kernel page

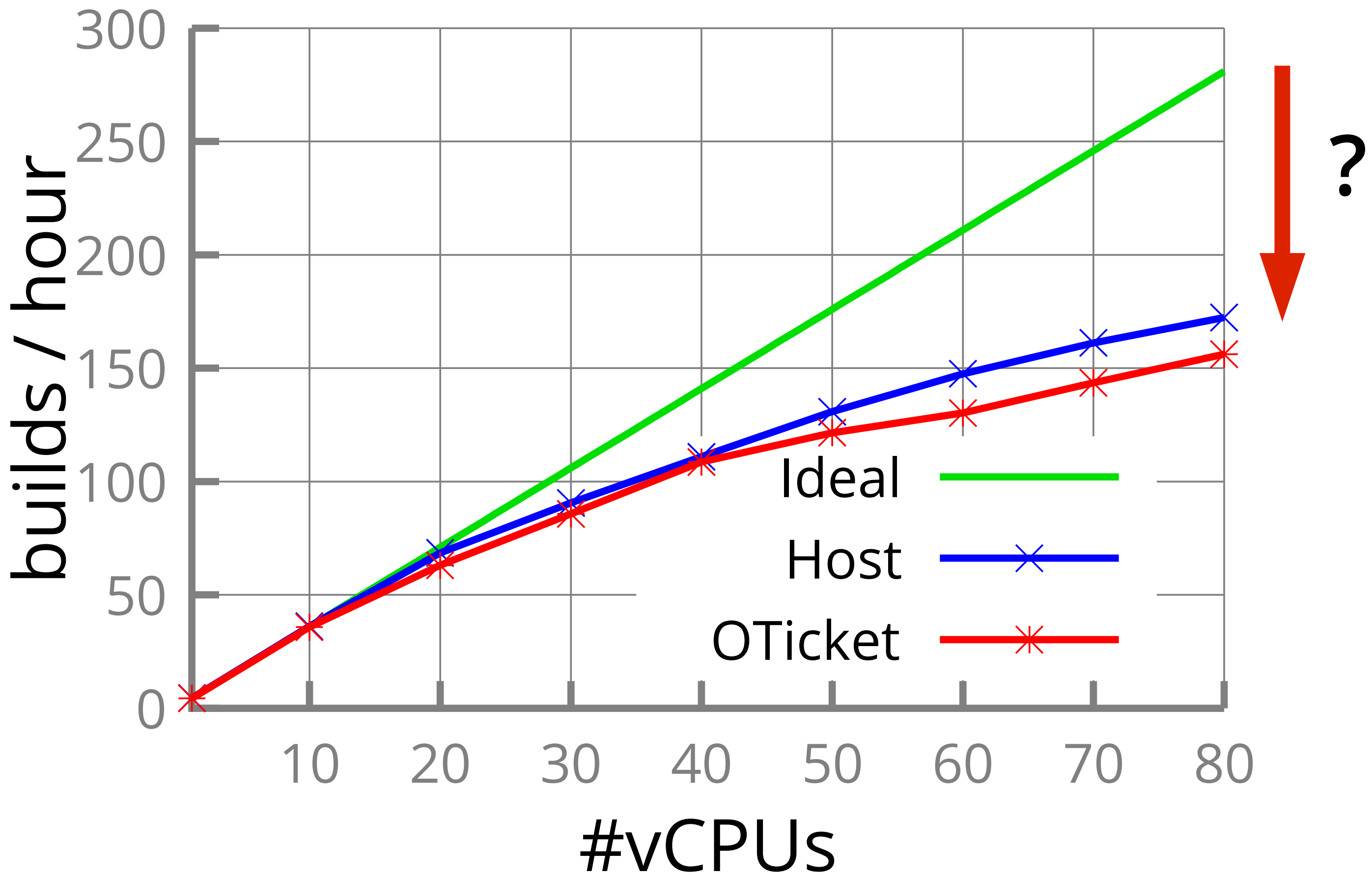
Recent Features

```
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
Georgia Institute of Technology

Questions?

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