Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds

Sanidhya Kashyap, Changwoo Min, Taesoo Kim

School of Computer Science, Georgia Institute of Technology

ABSTRACT

With increasing demand for big-data processing and faster in-memory databases, cloud providers are moving towards large virtualized instances besides focusing on the horizontal scalability.

However, our experiments reveal that such instances in popular cloud services (e.g., 32 vCPUs with 208 GB supported by Google Compute Engine) do not achieve the desired scalability with increasing core count even with a simple, embarrassingly parallel job (e.g., Linux kernel compile). On a serious note, the internal synchronization scheme (e.g., paravirtualized ticket spinlock) of the virtualized instance on a machine with higher core count (e.g., 80-core) dramatically degrades its overall performance. Our finding is different from the previously well-known scalability problem (i.e., lock contention problem) and occurs because of the sophisticated optimization techniques implemented in the hypervisor—what we call *sleepy spinlock anomaly*. To solve this problem, we design and implement OTICKET, a variant of paravirtualized ticket spinlock that effectively scales the virtualized instances in both undersubscribed and oversubscribed environments.

1. INTRODUCTION

The cloud is often considered the abyss of horizontal scalability. However, the advent of commodity and cost-effective multicore machines allows cloud providers to aim at achieving not only horizontal but also vertical scalability. For example, popular cloud providers now enable provisioning of large virtualized instances with higher vCPU count (up to 40 vCPUs) and larger memory space (up to 488 GB).[1] Not surprisingly, this trend will continue as the number of cores becomes readily available on commodity CPUs (e.g., up to 1,000 cores in SPARC M7 [27]). Following this trend, recently Amazon announced the introduction of an X1 instance [9] that comprises more than 100 vCPUs and 2 TB of memory, to cater to the increasing demands of large in-memory databases (Microsoft SQL server [25] and processing engines [5, 35]).

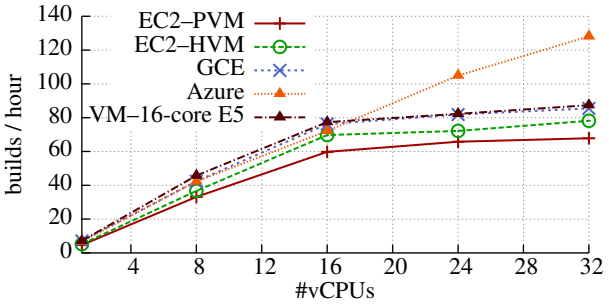


Figure 1: Performance of a Linux kernel compile on high-end VMs on Amazon EC2, Google Compute Engine, Microsoft Azure, and our in-house machine with a similar hardware configuration. According to our experiment, cloud environments (except Azure) with increasing vCPU count do not guarantee scalable performance to end users.

Given these emerging large machines, the main question we would like to answer in this paper is the following. What are the scalability characteristics of popular cloud providers? Additionally, is the underlying virtualization technology scalable enough to support VMs with hundreds of vCPUs in the future? We attempt to answer these questions by performing a Linux kernel compile, an embarrassingly parallel job that end users might expect to scale linearly by delegating the task to the cloud (e.g., elastically adjust the vCPU count on demand). We then replicate the same environment on our 80-core machine to project its scalability characteristics.

Figure 1 shows our experiment's results on the largest instances provided by three cloud services—Amazon Web Services (EC2), [2] Google Compute Engine (GCE), and Microsoft Azure (see Table 1). We can clearly observe that all VMs provided by the cloud services are scalable to 16 vCPUs. However, there is degradation after 16 vCPUs in the EC2 and GCE instances as the compilation plateaus. This happens because both cloud providers use hyperthreads for provisioning VMs with 32 vCPUs. We confirm this by replicating the same experiment in our lab with a 16-core E5-2630 v3 machine that has a similar hardware configuration as the VMs provided by the cloud providers (Table 1). On the contrary, Azure scales well beyond 16 vCPUs, showing ideal scalability characteristics for 32 vCPUs VM with respect to bare metal. Although there is no information available online, we assume that Azure allocates more physical cores (28) than logical ones (4), as the processor is a E5-2698B v3, which consists of 14 physical cores.

On our 80-core machine, we use the highly optimized paravirtual-

[1] Amazon Web Service (AWS) provides 40 vCPUs with 60 GB of memory, Google Compute Engine (GCE) provides up to 32 vCPUs with 208 GB of memory, and Microsoft Azure provides 32 vCPUs with 488 GB of memory.

[2] For an exact comparison, we use 32 vCPU VM rather than 36 vCPU one.

APSys'15 July 27-28, 2015, Tokyo, Japan

This is a minor revision of the work published in APSys'15, Proceedings of the 6th Asia-Pacific Workshop on Systems, ©ACM ISBN 978-1-4503-3554-6/15/07 http://dx.doi.org/10.1145/2797022.2797037.

Instances	# Cores (P / L)	Sockets	CPU Freq. (GHz)	Mem (GB)	L3 (MB)	Guest OS	Kernel	Virtualization type	Instance type	Cost / hour ($)
AWS	16 / 16	1	2.8	60.0	25	Ubuntu 14.04	3.13	PVM / HVM	c3.8xlarge	1.68
GCE	16 / 16	1	2.3	28.8	45	Ubuntu 15.04	3.19	PVM	n1-highcpu-32	1.28
Azure	28 / 4	2	2.0	488.0	40	Ubuntu 15.04	3.19	HVM	Standard G5	8.69
E5-2630 v3	16 / 16	2	2.4	64.0	20	Ubuntu 15.04	4.0.0	PVM / HVM	-	-

Table 1: Hardware and VM configurations of the Amazon EC2 (AWS), Google Compute Engine (GCE), Microsoft Azure (Azure), and our in-house machine used in Figure 1 for comparison. The following experiments were performed on May 2, 2015.

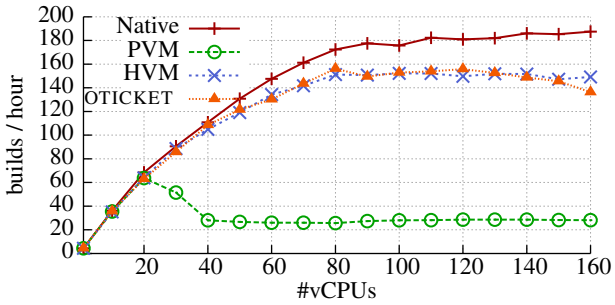


Figure 2: Performance of a Linux kernel compile on an 80-core machine. We enabled hyperthreads per core, similar to the cloud environments, and measured performance (builds/hour) on the host (marked Native), PVM, HVM, and our own implementation, OTICKET. Unlike our speculation—hyperthreads being the only performance bottleneck as we observed from Figure 1—we found serious performance degradation in the PVM-based hypervisor at higher core counts.

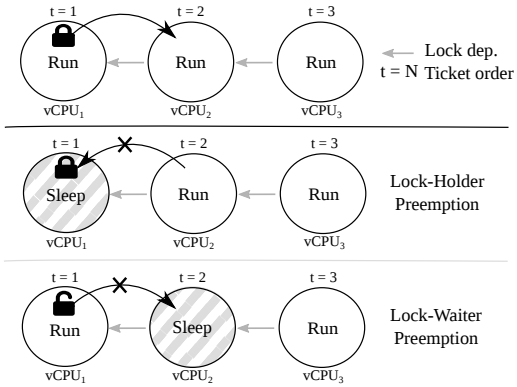


Figure 3: The issue of the lock-holder preemption (LHP) and lock-waiter preemption (LWP) problem. Each circle represents a vCPU scheduled on a CPU. The *Sleep* state is the preempted vCPU, whereas the *Run* state is the running one. `t` is the ticket order in which the vCPUs are waiting for the holder to release so that the waiters can acquire in FIFO order.

ized VM (PVM). Theoretically, the performance of PVM should be the same or better than the hardware-assisted VM (HVM)[3], as it can provide better performance (e.g., I/O or network throughput) due to the virtualization-aware interfaces. This trend should continue even for a large number of cores. Unfortunately, this trend tends to break, as Figure 2 pinpoints the scalability collapse for the increasing vCPU count from 20 to 30. This result is counterintuitive to what has been the case of paravirtualized instances and is only visible when the number of vCPUs is greater than 20 physical cores. We classify this problem as the *sleepy spinlock anomaly*, which is visible only in VMs using paravirtual spinlocks [1]. This problem does not stem from the cacheline contention, which has been observed in commodity OSes [10] and which previous studies have tried to reduce [10, 13, 14]. Instead, it arises from the introduction of ticket-based spinlock implementation that guarantees fairness. We address this problem by introducing two optimizations to the existing ticket spinlock. We improve the performance of PVMs for both undersubscribed and oversubscribed virtualized workloads by modifying 21 lines of code (LoC) without breaking the fairness guarantee.

In this paper, we make the following three contributions:

- We first reveal the scalability characteristics of three popular cloud services and develop an open source benchmark framework to evaluate various workloads by extending the benchmark tool, Mosbench [10], for the virtualized environment.
- We identify a scalability bottleneck called the *sleepy spinlock anomaly* in the paravirtual spinlocks for VMs with high vCPU count, which degrades the performance of both undersubscribed and oversubscribed environments.
- We propose OTICKET, a variant of the paravirtualized ticket spinlock that scales in both undersubscribed and oversubscribed virtualized environments.

[3]HVM model only relies on the hardware for the VM execution.

In the rest of the paper, we provide a high-level overview of the paravirtual spinlock implementation (§2) and describe our two optimizations in §3. §4 discusses the implementation of the optimization for the ticket spinlock in the Linux kernel, and §5 evaluates our optimization performance. §6 discusses the limitation and potential issues. Finally, §7 compares our approach with previous research and §8 provides the conclusion.

2. BACKGROUND

Spinlock is a basic building block for synchronization primitives inside the Linux kernel. As core count increases, the scalability of the spinlock becomes important. A recent study shows that non-scalable spinlock can cause performance collapse in an entire system [10, 11]. Therefore, to ensure scalability and fairness [3, 11, 23], the key design choices are to minimize shared cacheline contention as well as guarantee fairness, which is typically achieved by preserving FIFO ordering, to prevent starvation with increasing core count. The Linux kernel[4] uses ticket spinlock for the fairness guarantee and recently adopted queue-based spinlock for better scalability [21]. A ticket spinlock is represented as a tuple—[`head`, `tail`]. The current ticket holder holds the `head`, while a lock waiter increments the `tail` and spins until the `head` becomes equal to the `tail`. During the unlock phase, `head` is incremented for the next lock waiter to acquire the lock.

In virtualized environments, the introduction of vCPUs complicates the scalability of the busy-waiting synchronization primitives—

[4]The current Linux kernel version (v4.3), as of this writing, uses `qspinlock` [21]. Like ticket spinlock, `qspinlock` guarantees FIFO ordering for fairness and has additional optimization to reduce shared cacheline contention. Since both ticket spinlock and `qspinlock` guarantee FIFO ordering for fairness, they are not free from the *sleepy spinlock anomaly*. In §5.2, we discuss the *sleepy spinlock anomaly* in `qspinlock`.

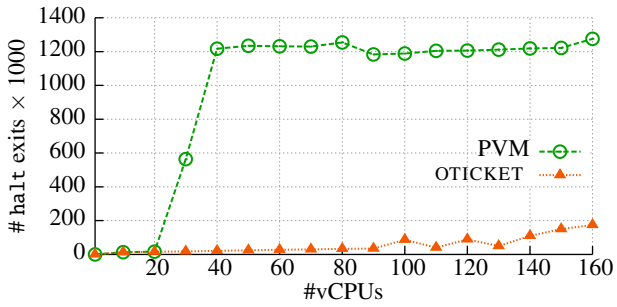


Figure 4: The number of `halt` exits that occur during a Linux kernel compile for two variants of spinlocks: in-stock ticket spinlock (PVM) and our OTICKET implementation.

spinlocks. Figure 3 illustrates two anomalies, namely, lock-holder preemption (LHP) and lock-waiter preemption (LWP). LHP occurs when the vCPU that is holding the lock gets preempted and none of the lock waiters make progress. On the contrary, LWP happens due to the FIFO-ordered spinlock algorithms (like ticket spinlock), in which none of the waiters make progress unless the exact next waiter is scheduled.

To address the aforementioned issues, Intel added the *Pause Loop Exiting* (PLE) feature to its processors [29]. In PLE, a processor detects whether a task is contending on a spinlock. If a core keeps executing `pause` instructions, which is called in every spin loop, a guest OS gets trapped by the hypervisor. The hypervisor then optimistically tries to yield the vCPU's remaining time-slice to the other preempted vCPU. Unfortunately, the hypervisor cannot determine who is the current lock holder (when a lock is acquired) or the next lock waiter (when a lock is released) by solely relying on the PLE event. Therefore, HVM, which only relies on PLE, is suboptimal in spinlock scalability.

From the software perspective, the *paravirtual spinlock* [16, 28] tries to address these problems without any architectural support. Unlike typical spinlock implementations, the paravirtual spinlock consists of two paths, namely, fast-path and slow-path. In the fast-path routine, a lock waiter first spins for a lock for fixed iterations. It falls back to the slow-path if it fails to acquire the lock. In the slow-path routine, the waiter issues a `halt` instruction, which gets trapped by the hypervisor. The hypervisor schedules out the waiter and voluntarily yields to the other vCPUs. During the unlock phase, the holding vCPU wakes up the next waiter via a `kick` hypercall, which notifies the hypervisor to re-schedule the waiting vCPU. Thus, voluntary yielding alleviates the LHP problem and precisely waking the next vCPU alleviates the LWP problem.

Surprisingly, our evaluation results in Figure 2 show sudden performance collapse at a higher number of cores. This occurs due to the increase in `halt` exits after 30 vCPUs (Figure 4). This is different from both LHP and LWP. The performance collapse is triggered due to high contention among the vCPUs for shared resources (e.g., critical section or memory bus). Therefore, the duration between lock acquisition to release tends to be longer, and at a certain point (30 vCPUs in this case), most vCPUs fail to acquire a lock during optimistic spinning (i.e., fast-path) and trap to the hypervisor at the same time (as evident after 20 vCPUs in Figure 4). From this point, both of the switching overhead between the guest OS and the hypervisor, and the communication cost to wake other vCPUs starts dominating, which results in such a drastic performance collapse.

It is challenging to design and implement a spinlock that performs well in virtualized environments, especially at higher core count. In the rest of this paper, we present our *opportunistic ticket spinlock* (OTICKET), a variant of paravirtualized ticket spinlock, that opportunistically tries to keep the waiters in the fast-path routine.

3. DESIGN

We propose *opportunistic ticket spinlock* (OTICKET), a new spinlock algorithm that is specifically designed for virtualized environments. OTICKET's design not only resolves the performance anomaly as shown in Figure 2, but also addresses the LHP and LWP problems, which are critical to achieving scalability in virtualized environments.

Like the stock paravirtual ticket spinlock in the Linux kernel, OTICKET is composed of a fast-path and a slow-path. Each vCPU spins first and then voluntarily yields to other vCPUs if it is unable to acquire the lock. In addition, to resolve LHP and LWP, we introduce two schemes, *opportunistic spinning* and *opportunistic wakeup*. To make the optimal decision on spinning and waking-up, we exploit the *distance* between the lock holder and the lock waiter. Since ticket spinlock guarantees strict FIFO ordering of waiters, we assume that the time to acquire a lock is roughly proportional to the waiter's distance. With the help of both schemes, OTICKET mitigates the problem of *sleepy spinlock anomaly* by keeping the waiters in the fast-path, thereby decreasing the number of halt-exits (Figure 4).

Opportunistic spinning. Determining the spinning duration of the fast-path is challenging since it is dependent on the workload and hardware combination. Longer spins unnecessarily hog the CPU cycles, but shorter durations result in the performance collapse shown in Figure 2.

In OTICKET, the spin duration is dynamically determined by the distance between the lock waiter and its holder. Closer waiters (i.e., waiters with smaller ticket distance from the holder) opportunistically spin for a longer duration, hoping to acquire the lock sooner. If a lock is acquired while spinning, the vCPU can avoid the problems of costly switching between the guest OS and hypervisor. Conversely, farther waiters spin shorter and yield early to give more opportunities for a lock holder to make progress. In OTICKET, as the distance of a lock waiter increases, the spinning iteration exponentially decreases (Lines 27–30 in Figure 5). Consequently, this results in LHP problem mitigation, as only effective vCPUs get scheduled by the hypervisor.

Opportunistic wakeup. Waking up a `halt`-ed vCPU takes significant time [15], since the process involves notifying the hypervisor which schedules the target vCPU. The unlocked vCPU uses hypercall for the notification to wakeup the target vCPU. To hide this wakeup latency, OTICKET allows the unlocking vCPU to wake the next `N` lock waiters in advance (Lines 65–68 in Figure 5). This allows the waiters to fall back to the fast-path and subsequently keep on spinning for their turn, thereby eagerly waiting for the lock-holder to release the lock soon. Therefore, in conjunction with *opportunistic spinning*, it partially mitigates the LWP problem.

4. IMPLEMENTATION

We implemented our opportunistic ticket spinlock in Linux kernel v4.0 by replacing the paravirtual ticket spinlock in the KVM hypervisor with OTICKET. Our paravirtual spinlock is practical because of its minimal modification (lines starting with + in Figure 5) without any changes to the size of its lock structure. The design of OTICKET can be easily used by other open-source hypervisors such as Xen [8]. In our locking function, `arch_spin_lock()`, `__ticket_distance()` calculates the distance between a lock holder and a waiter (Lines 6–11) before spinning, and `__ticket_lock_spin-ning()` makes its running vCPU yield to other vCPUs by executing a `halt` instruction (Line 43). In our unlock function, `arch_spin_unlo-ck()`, OTICKET

```
1  #define SPIN_THRESHOLD     (1 << 15)
2  #define SPIN_MAX_THRESHOLD (1UL << 34)
3  #define TICKET_QUEUE_WAIT  (18)
4  #define OPPORTUNISTIC_WAKEUP_NCPU  (4)
5
6  + static __always_inline
7  + unsigned int __ticket_distance(__ticket_t head, __ticket_t tail)
8  + {
9  +   return (tail - (head & ~TICKET_SLOWPATH_FLAG)) \
10 +     / TICKET_LOCK_INC;
11 + }
12
13   static __always_inline
14   void arch_spin_lock(arch_spinlock_t *lock)
15   {
16     register struct __raw_tickets inc = {.tail = TICKET_LOCK_INC};
17 +   unsigned int dist;
18
19     /* default threshold set in Linux */
20     u64 threshold = SPIN_THRESHOLD
21
22     /* try locking */
23     inc = xadd(&lock->tickets, inc);
24     if (likely(inc.head == inc.tail))
25       goto out;
26
27 +   /* opportunistically determines spinning threshold */
28 +   dist = __ticket_distance(inc.head, inc.tail);
29 +   if (dist < TICKET_QUEUE_WAIT)
30 +     threshold = SPIN_MAX_THRESHOLD >> (dist - 1);
31
32     for (;;) {
33       /* spinning (fast path) */
34 +     u64 count = threshold;
35       do {
36         inc.head = READ_ONCE(lock->tickets.head);
37         if (__tickets_equal(inc.head, inc.tail))
38           goto clear_slowpath;
39         cpu_relax();
40       } while (--count);
41
42       /* yield (slow path) */
43       __ticket_lock_spinning(lock, inc.tail);
44     }
45
46   clear_slowpath:
47     __ticket_check_and_clear_slowpath(lock, inc.head);
48   out:
49     barrier();
50   }
51
52   static __always_inline
53   void arch_spin_unlock(arch_spinlock_t *lock)
54   {
55     if (TICKET_SLOWPATH_FLAG &&
56         static_key_false(&paravirt_ticketlocks_enabled)) {
57       __ticket_t head;
58
59       head = xadd(&lock->tickets.head, TICKET_LOCK_INC);
60
61       if (unlikely(head & TICKET_SLOWPATH_FLAG)) {
62 +       u8 count;
63         head &= ~TICKET_SLOWPATH_FLAG;
64
65 +       /* opportunistic wakeup */
66 +       for (count = 1; count <= OPPORTUNISTIC_WAKEUP_NCPU;
67 +            ++count)
68 +         __ticket_unlock_kick(lock,
69 +                     (head + count * TICKET_LOCK_INC));
70       }
71     } else
72       __add(&lock->tickets.head,
73             TICKET_LOCK_INC, UNLOCK_LOCK_PREFIX);
74   }
```

Figure 5: Our opportunistic ticket spinlock code implemented in the Linux kernel 4.0 [1]. It opportunistically increases the spinning threshold from the static threshold in the stock Linux, and opportunistically wakes up more vCPUs near their ticketing turn.

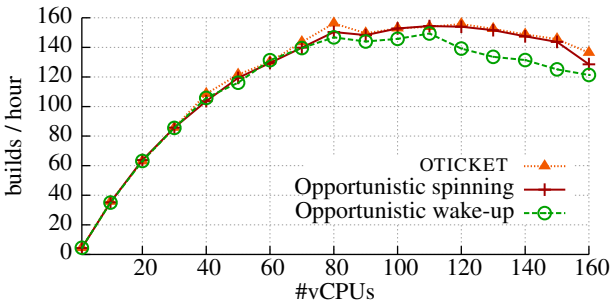


Figure 6: Performance impact of each optimization for the Linux kernel compile using modified paravirtualized spinlock interface. OTICKET is the implementation shown in Figure 5. Opportunistic wakeup and spinning are the individual ticket spinlock implementations that constitute OTICKET.

opportunistically wakes up vCPUs (Lines 65–69). The waking-up function, `__ticket_unlock_kick()`, is implemented using a hypercall. Besides this, we do not modify any other kernel functions for the OTICKET implementation.

5. EVALUATION

We evaluate OTICKET by answering the following four questions:

- Does OTICKET improve the scalability of PVM? (§5.1)
- Does any other spinlock implementation solve the scalability issue? (§5.2)
- Does OTICKET help to improve the performance of VMs in an oversubscribed environment? (§5.3)
- Is the design of OTICKET generic across other multicore machines? (§5.4)

Experimental setup. We created VBENCH[5], a fork of Mosbench [10], to evaluate the scalability of the host and hypervisor while running multiple virtual machines. For our evaluation, we chose three benchmarks from VBENCH: Linux kernel compilation (LKC), EXIM [4], and METIS [22].

Linux kernel compilation (LKC) is an embarrassingly parallel job in which each process independently compiles source files. To hide potential IO latency, we set the number of parallel jobs of kernel compile to twice the number of cores, both for VM and host.

EXIM is the most deployed mail-server [18]. For each SMTP connection, EXIM forks two processes for message processing and delivery. Since the processing and delivery of each message is independent, it is also an embarrassingly parallel job. We use 160 clients to deliver 100 messages continuously in a single connection for 15 seconds.

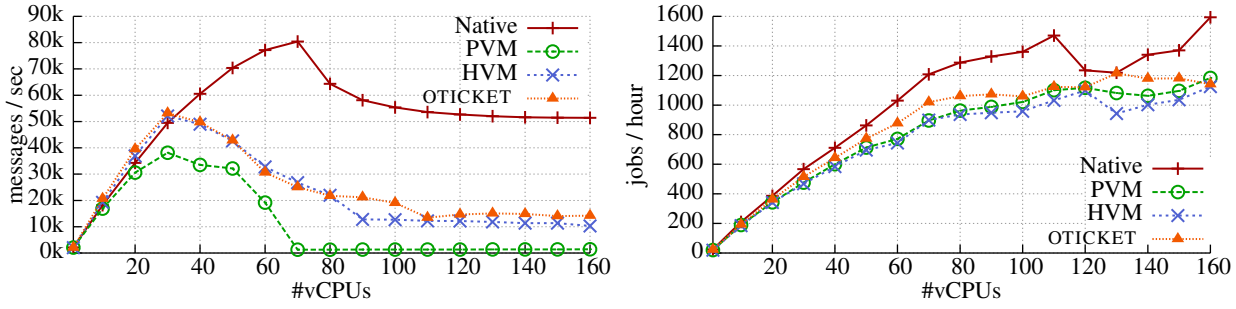
METIS is a map-reduce library for a single multicore server. In our experiments, METIS is used with an application for generating inverted indices, which mostly stresses the kernel's memory allocator and soft page-fault handling component, but it does not suffer from spinlock contention. We chose METIS to show that OTICKET does not have an overhead in the case of low spinlock contention.

All of these workloads scale almost linearly with increasing core count without any virtualization. To isolate the effect of I/O, we run these benchmarks on top of the memory-backed file system, `tmpfs`, while pre-loading all of the input source files before measuring the performance, if required. Also, to isolate the effect of process migration at host, we pin each vCPU to each physical core.

5.1 Performance Analysis

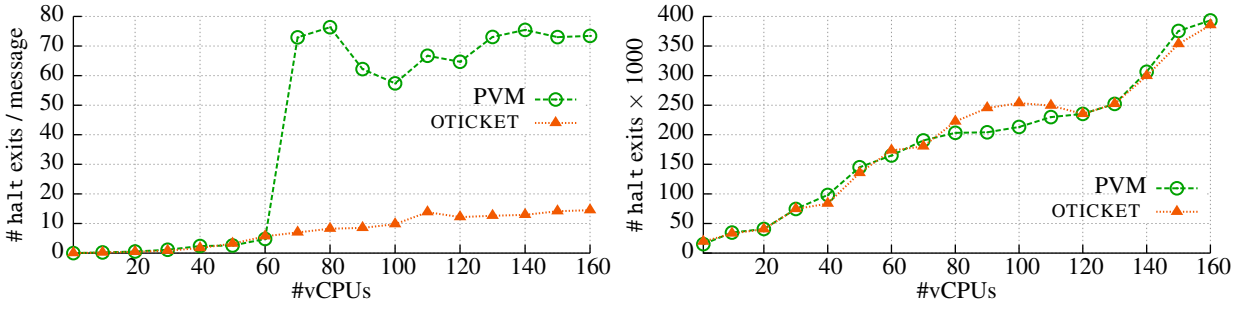
Figure 2 shows that the stock ticket spinlock (PVM) starts suffering after 30 vCPUs and its performance completely collapses at

[5] https://github.com/sslab-gatech/vbench



(a) EXIM **(b)** METIS

Figure 7: Performance impact of OTICKET compared to PVM and HVM while running EXIM and METIS benchmarks on our 80-core machine.



(a) EXIM **(b)** METIS

Figure 8: The number of halt-exits for both EXIM and METIS. For EXIM, halt-exits are per message whereas for METIS, they are per the whole run.

40 vCPUs. In contrast, OTICKET achieves consistent performance improvement until 80 vCPUs (all physical CPUs assigned) and does not show equivalent performance collapse in PVM until 160 vCPUs (all hardware threads assigned). Figure 4 illustrates the difference between OTICKET and PVM as the number of halt-exits of the PVM starts soaring after 30 vCPUs, but remains almost constant for OTICKET. It reveals that the voluntary sleeping optimization for the virtualized environments can result in performance collapse (*sleepy spinlock anomaly*) for the kernel-intensive workloads, which OTICKET tries to avoid.

Figure 6 shows the effectiveness of each scheme. OTICKET provides the best of both worlds—opportunistic spinning and opportunistic wakeup, by performing slightly better than both and the best at 80 vCPUs (consisting of only physical cores). The opportunistic spinning approach prohibits the nearest waiters from going to sleep, thereby immediately acquiring the lock. Both approaches start degrading after 110 cores because of the large number of waiters that are going to sleep from both the use of logical cores and increasing the contention among vCPUs.

Figure 7 shows the performance of both EXIM and METIS on the host as well as on the different configuration of the guest. OTICKET outperforms PVM since from 30 cores onward PVM starts to suffer from the *sleepy spinlock anomaly*. This happens because EXIM spends around 70% of its time in the kernel and around 20% of the time in spinlock contention for the new inode allocation. As shown in Figure 8, this contention severely increases the amount of halt-exits for the PVM, which increases from 0.5 per message at 20-core to 72 at 70-core and remains consistent. Compared to PVM, OTICKET only has 7 halt-exits per message at 70-core. OTICKET reduces the number of halt-exits by 10X with both of its techniques. The increase in halt-exits that are observed with increasing core count are due to the contention on the blocking synchronization

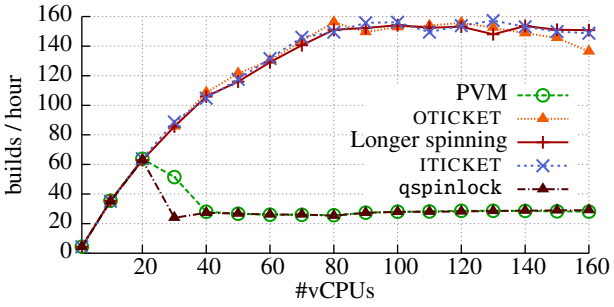


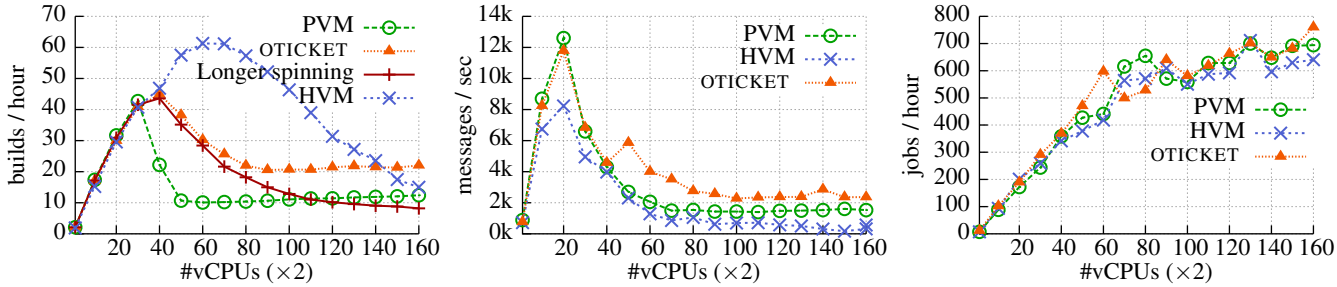
Figure 9: Linux kernel build performance on 80-core for various paravirtual spinlock implementations: Opportunistic ticket spinlock (OTICKET), ticket spinlock with longer spinning (Longer spinning), fine-grained control of spinning period (ITICKET), and a MCS variant queue-based spinlock (`qspinlock`).

primitives (e.g., mutex and read-write semaphore) which hampers all of the configuration, even including the native.

In the case of METIS, each of the configurations performs the same, thereby illustrating that OTICKET has the same performance impact as that of PVM, as it does not harm the scalability of the application in the non-contending scenario. Figure 8b further confirms OTICKET's negligible effect on METIS performance, as the amount of halt-exits remains almost the same for both PVM and OTICKET.

5.2 Comparison with Design Alternatives

We explore three design alternatives—(1) longer spinning, (2) fine-grained control of the spinning period (ITICKET), and (3) queue-based spinlock (`qspinlock`) for further reducing cacheline contention—and compare their performance to OTICKET and the stock ticket spinlock (PVM). We use LKC for comparing other design



(a) LKC **(b)** EXIM **(c)** METIS

Figure 10: Performance impact of OTICKET compared to PVM and HVM while running LKC, EXIM, and METIS benchmarks in an over-committed environment on an 80-core machine (2 vCPUs per core). Each benchmark is co-scheduled with the same instance on another exact VM instance. For the LKC workload, we also have included *longer spinning* performance to show its adverse effect in an oversubscribed scenario.

decisions.

To spin longer, we modify the default spinning threshold of the stock ticket spinlock to the maximum (`SPIN_MAX_THRESHOLD` in Figure 5) expecting longer spinning to reduce halt-exits significantly.

We further modify the existing ticket spinlock structure by introducing a new variable for holding the threshold value for each lock structure. The default spinning threshold value (`SPIN_THRES-HOLD` in Figure 5) is increased twice whenever a contended thread of the same lock instance ends up in the slow-path. This allows more fine-grained control of the spinning duration per spinlock. We call this ticket spinlock implementation ITICKET.

Practitioners are leaning towards using variants of MCS lock [3] for better scalability in large NUMA machines by further reducing cacheline contention. In practice, Linux kernel recently adopted a space-optimized variant of MCS lock, called queue-based spinlock [21] or `qspinlock`, which maintains the same size of the spinlock instance as one of the ticket spinlock (4 bytes).

Figure 9 illustrates that the longer spin approach performs slightly better than OTICKET as the number of cores increases from 130 cores. This happens because all waiters are spending more time spinning than going to the slow-path. But, as expected, this pays the price in an oversubscribed environment (Figure 10a).

The performance of ITICKET spinlock implementation is similar to OTICKET but adds a significant overhead of 2 bytes at every place it is being used. In practice, increasing the size of a spinlock data structure has serious repercussions to tightly packed container data structures such as page structures.

Figure 9 shows the performance of three alternatives against the existing spinlock implementation (PVM). We can observe that `qspinlock` also suffers from the *sleepy spinlock anomaly*, as there is no improvement in the virtualized environment. This proves that the anomaly can occur for any spinlock algorithm that guarantees ordering and also relies on the slow-path in the virtualized environment.

5.3 Performance in an Over-Committed Host

Another interesting aspect of the spinlock design for virtualized environments is the performance behavior in an over-committed setting (i.e., running more vCPUs than physical cores). Although the highly over-committed case will be avoided using virtual machine migration, it is desirable to have good performance in the over-committed case to cope with the overlapping peaks among VMs.

Figure 10 shows the performance of each workload running inside a VM that has been co-scheduled with another VM, simultaneously running the same workload.

OTICKET outperforms the existing ticket spinlock in the case the benchmark suffers from the *sleepy spinlock anomaly* (LKC and EXIM), and its performance is equivalent to PVM for the non-contending case (METIS).

We further use the alternative longer spinning approach with LKC (refer Figure 10a). This proves that, although longer spinning is advantageous for under-committed environments, it drastically degrades the performance in the over-committed setting. Longer spinning is helpful until 40 cores, but starts degrading due to the wasting of CPU cycles.

PVM suffers from the sleepy spinlock anomaly as well as contention with other vCPUs since another VM is performing the same job. This inherently comes from the strict FIFO ordering and contention on blocking synchronization primitives (for EXIM) which severely limits its performance. On the other hand, OTICKET suffers from the same problem, but performs better than these two techniques. Our opportunistic wakeup scheme partially hides the latency of other waiters by waking them up beforehand. This also allows the vCPU to schedule other tasks, thus allowing the VM to progress further.

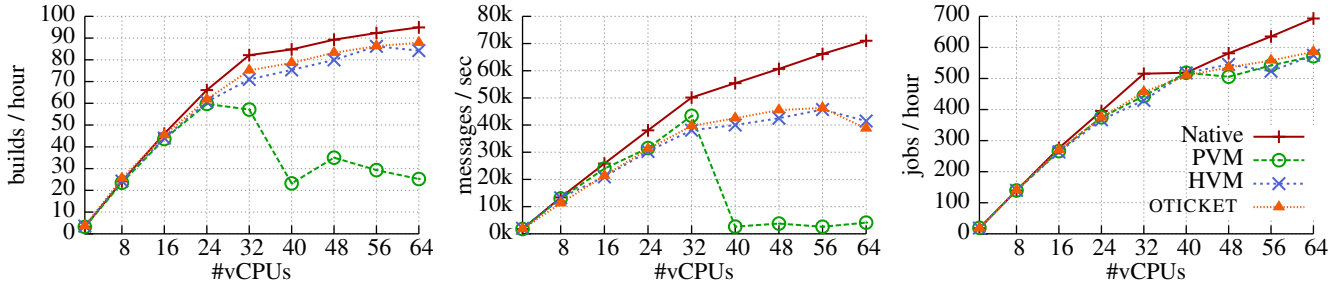
The HVM configuration performs better than OTICKET and PVM for LKC because it does not rely on the paravirtual spinlock interface, thereby not suffering from the *sleepy spinlock anomaly*. However, its performance starts suffering from 60-core onward because of the I/O thread contention (54%), which is not observed in the case of others, since the *sleepy spinlock anomaly* hides this contention for both PVM and OTICKET.

For EXIM, HVM's performance degrades more than PVM and OTICKET after 60-core. This occurs because of the contention at the hypervisor level during vCPU scheduling.

We observed that all three configurations perform the same in the case of METIS, which neither suffers from *sleepy spinlock anomaly* nor performs any heavy kernel-based operations, unlike other benchmarks.

5.4 Machine Independence

We use our 32-core machine (64-core with HT enabled) to further confirm the *sleepy spinlock anomaly* as well as the impact of OTICKET. Figure 11 shows the performance evaluation of PVM, HVM, and OTICKET for all three workloads. All three workloads illustrate the same trend as that seen in case of 80-core (Figure 2 and Figure 7), and both LKC and EXIM suffer from the *sleepy spinlock anomaly*. Here, as well, OTICKET solves this anomaly with its two effective schemes that try to keep the waiters in the fast path. Similarly, METIS performance in both host and guest remains the same as that observed for 80-core.



(a) LKC **(b)** EXIM **(c)** METIS

Figure 11: Performance impact of OTICKET compared to PVM and HVM for EXIM, LKC, and METIS on our 32-core machine with HT enabled.

6. DISCUSSION AND LIMITATIONS

The paravirtualized interfaces often improve the VM's performance [1, 17, 28], as they provide more control to the guest execution by enabling coordination between the guest OS and hypervisor. However, we observe an anomaly in VMs with high core count leveraging these paravirtualized interfaces, which we try to fix via OTICKET. However, OTICKET does not solve the issue found in either LKC's oversubscribed case nor in EXIM. We believe that more coordination is necessary between the guest and hypervisor to have the scalability trend comparable to the host. Two possible solutions are to tightly integrate PLE with OTICKET's logic or perform co-scheduling vCPUs [19] at the time of yielding. Both solutions may further improve the performance and decrease the performance gap between the bare metal and virtualized executions. These approaches may not only solve the problem of performance drop for over-subscribed machines with high core count, but also should improve the performance of EXIM representative workloads.

7. RELATED WORK

VM scheduling and synchronization have a serious impact on the performance of a VM. There have been prior significant efforts to improve the performance of VM scheduling.

Spinlocks for virtualized environment. Uhlig et al. [33] defined and addressed the lock synchronization issue (LHP) in the virtualized environment via scheduling hints. Later, paravirtual hooks were used in the spinlock [16] for notifying the hypervisor to block the vCPU after it has exhausted its busy wait threshold. This approach, however, prevents LHP for smaller core counts. Besides LHP, two different problems have been identified: lock-waiter preemption (LWP) [26] and blocked-waiter wakeup problem (BWW) [15, 30]. BWW occurs when the workload uses blocking synchronization in an over-subscribed environment.

From the hardware perspective, processor manufacturers added an execution control to the VMCS structure—Pause Loop Exiting (PLE) [29]—that notifies the hypervisor of the waiter via VM exit. PLE partially solves the LHP problem but can also result in false positives. Ahn et al. [7] proposed a solution on the basis of a smaller time slice to resolve both interrupt handling and LHP-LWP problems. They proposed an LLC-based architectural solution to resolve the large overhead. This approach will result in a huge overhead for VMs with a high core count, and degradation might remain consistent.

There have been other alternatives of spinlock implementations such as MCS locks [3, 12, 21] that are considered a better alternative to ticket spinlock implementation. Unfortunately, the issue of *sleepy spinlock anomaly* stems in spinlock implementations following strict FIFO ordering. Therefore, this problem will continue to be seen in the queue-based spinlock for virtualized environments.

Virtualization overhead and scheduling. There have been several studies on the virtualization overhead due to software-hardware redirection [6, 30] and co-scheduling issues [15, 16, 24, 26]. In the vCPU scheduling space, hypervisors, such as VMware, adopted the co-scheduling of multiple vCPUs [2] to deal with guest and VMM synchronization. This was further improved by using an adaptive scheme for scheduling the vCPUs [7, 19, 33, 34]. Later, Orathai et al. [32] came up with the approach of dedicating the vCPU with a physical CPU rather than co-scheduling. Furthermore, Song et al. [31] used the approach of vCPU ballooning on top of physical CPUs, which avoided the problem of double scheduling.

8. CONCLUSION

In this paper, we analyze the scalability performance of a VM on an 80-core machine for the Linux kernel compilation benchmark. Our study suggests that in addition to cacheline contention, the use of spinlock, which guarantees strict FIFO ordering, is another culprit for the performance degradation. We identify this issue for VMs with large vCPU count and provide a variant of the ticket spinlock implementation to address this problem. The VBENCH source code is publicly available at `https://github.com/sslab-gatech/vbench` and is easily extensible to identify more issues with respect to the virtualization for large multicore machines. Our initial idea of the spinlock design has been already acknowledged and adopted by the `qspinlock` developer [20], which will be in the mainline in the next kernel version.

In the future, we would like to devise a generic OTICKET design that is not susceptible to increasing core count and can perform equivalently or better than HVM in an over-subscribed environment for workloads relying on busy-waiting synchronization. We will further extend our insight to other synchronization primitives with respect to virtualization.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers at APSYS'15, and our shepherd, Ketan Maheshwari, for their helpful feedback, as well as the operations staff for their proofreading efforts. This research was supported by the NSF award CNS-1017265, CNS-0831300, CNS-1149051 and DGE-1500084, ONR under grant No. N000140911042 and N000141512162, DHS under contract No. N66001-12-C-0133, the United States Air Force under contract No. FA8650-10-C-7025, DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006, ETRI MSIP/IITP[B0101-15-0644], and NRF BSRP/MOE[2015R1A6A3A03019983].

References

[1] Paravirtualized Spinlocks, 2008. http://lwn.net/Articles/289039/.

[2] The CPU Scheduler in VMware ESX 4.1.

[3] MCS locks and qspinlocks, 2014. https://lwn.net/Articles/590243/.

[4] Exim Internet Mailer, 2015. http://www.exim.org/.

[5] SAP HANA, 2015. http://hana.sap.com/abouthana.html.

[6] ADAMS, K., AND AGESEN, O. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), ASPLOS.

[7] AHN, J., PARK, C. H., AND HUH, J. Micro-Sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), MICRO.

[8] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (Bolton Landing, NY, Oct. 2003).

[9] BARR, J. EC2 Instance Update – X1 (SAP HANA) & T2.Nano (Websites), 2015. https://aws.amazon.com/blogs/aws/ec2-instance-update-x1-sap-hana-t2-nano-websites/.

[10] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI.

[11] BOYD-WICKIZER, S., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. Non-scalable locks are dangerous. In *Ottawa Linux Symposium* (2012), OLS.

[12] BUESO, D. Scalability Techniques for Practical Synchronization Primitives, 2014. https://queue.acm.org/detail.cfm?id=2698990.

[13] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the ACM EuroSys Conference* (Prague, Czech Republic, Apr. 2013), pp. 211–224.

[14] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farmington, PA, Nov. 2013), pp. 1–17.

[15] DING, X., GIBBONS, P. B., KOZUCH, M. A., AND SHAN, J. Gleaner: Mitigating the Blocked-waiter Wakeup Problem for Virtualized Multicore Applications. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (2014), USENIX ATC.

[16] FRIEBEL, T. How to Deal with Lock-Holder Preemption. In *Xen Summit North America* (2008), XenSummit.

[17] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. Eli: Bare-metal performance for i/o virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (London, UK, Mar. 2012), ASPLOS XVII, pp. 411–422.

[18] INC., E.-S. Mail (MX) Server Survey, 2014. http://www.securityspace.com/s_survey/data/man.201404/mxsurvey.html.

[19] KIM, H., KIM, S., JEONG, J., LEE, J., AND MAENG, S. Demand-based Coordinated Scheduling for SMP VMs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS.

[20] LONG, W. locking/qspinlock: Enhance pvqspinlock & introduce queued unfair lock, 2015. https://lwn.net/Articles/650776/.

[21] LONG, W. qspinlock: a 4-byte queue spinlock with PV support, 2015. https://lkml.org/lkml/2015/4/24/631.

[22] MAO, Y., MORRIS, R., AND KAASHOEK, F. M. Optimizing MapReduce for Multicore Architectures. In *MIT CSAIL, Technical Report* (2010).

[23] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-Copy Update. In *Ottawa Linux Symposium* (2002), OLS.

[24] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (2005), VEE.

[25] MICROSOFT. SQL Server 2014, 2014. http://www.microsoft.com/en-us/server-cloud/products/sql-server/features.aspx.

[26] OUYANG, J., AND LANGE, J. R. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2013), VEE.

[27] PHILLIPS, S. M7: Next Generation SPARC.

[28] RAGHAVENDRA, K., VADDAGIRI, S., DADHANIA, N., AND FITZHARDINGE, J. Paravirtualization for Scalable Kernel-Based Virtual Machine (KVM). In *Cloud Computing in Emerging Markets (CCEM)* (2012).

[29] RIGHINI, M. Enabling Intel Virtualization Technology Features and Benefits, 2010.

[30] SONG, X., CHEN, H., AND ZANG, B. Characterizing the Performance and Scalability of Many-core Applications on Virtualized Platforms. In *Fudan University - Technical Report* (2010).

[31] SONG, X., SHI, J., CHEN, H., AND ZANG, B. Schedule Processes, Not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems* (2013), APSys.

[32] SUKWONG, O., AND KIM, H. S. Is Co-scheduling Too Expensive for SMP VMs? In *Proceedings of the ACM EuroSys Conference* (Salzburg, Austria, Apr. 2011).

[33] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3* (2004), VM.

[34] WENG, C., LIU, Q., YU, L., AND LI, M. Dynamic Adaptive Scheduling for Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing* (2011), HPDC.

[35] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (2010), HotCloud'10.