Scalability in the Clouds! A Myth or Reality?

Sanidhya Kashyap Changwoo Min Taesoo Kim

School of Computer Science
Georgia Institute of Technology

Abstract

With increasing demand of big-data processing and faster in-memory databases, cloud providers are gearing towards large virtualized instances rather than 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., 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 a previously well-known scalability problem (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 virtual instances with higher vCPU count (up to 36 vCPUs)

APSys'15, July 27-28, 2015, Tokyo, Japan.
Copyright © 2015 ACM ISBN 978-1-4503-3554-6/15/07…$15.00.
http://dx.doi.org/10.1145/2797022.2797037

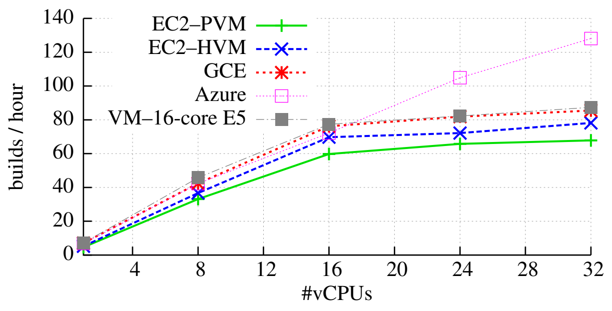

Figure 1: Performance of a Linux kernel compile on high-end VMs on Amazon EC2, Google Compute Engine, Microsoft Azure, and our in-house machine with similar hardware configuration. According to our experiment, cloud environments (except Azure) with increasing vCPU count do not guarantee scalable performance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

and larger memory space (up to 488 GB) [1]. Not surprisingly, this trend will continue as increasing number of cores become readily available on commodity CPUs (e.g., up to 1K cores in SPARC M7 [3]) since there is a huge demand on catering large in-memory databases and processing engines (e.g., 240-core machine [5] for SAP HANA [6]).

Given these upcoming 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 question by performing a Linux kernel compile, an embarrassingly parallel job that end users might expect to scale vertically 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), Google Compute Engine (GCE), and Mi-

[1] Amazon Web Service (AWS) provides 36 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.

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.

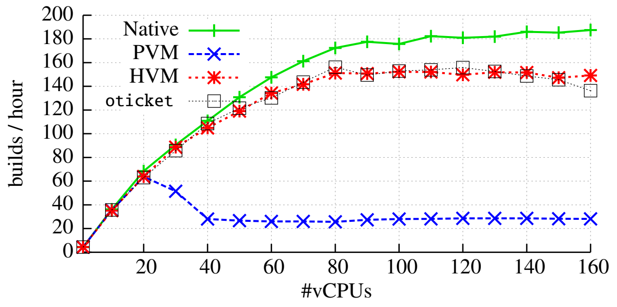

Figure 2: Performance of a Linux kernel compile on an 80-core machine. We enabled hyperthreads per core, similar to the cloud environments, and measured performance (builds/hour) on host (marked Native), PVM, HVM, and our own implementation, `oticket`. Unlike our speculation—hyperthreads being the only performance bottleneck as we observed from Figure 10– we found serious performance degradation in the PVM-based hypervisor at higher core counts.

crosoft 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 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 believe 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 paravirtualized VM (PVM). Theoretically, the performance of PVM should be the same or better than HVM even for large number of cores. Unfortunately, this trend tends to break as Figure 2 pinpoints the scalability bottleneck for the increasing vCPU count from 20 to 30. This result is counter-intuitive 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 only visible in VMs using paravirtual spinlocks [1]. This problem does not stem from the cacheline contention that has been observed in commodity OSes [9]; instead it arises from the introduction of ticket based spinlock implementations that try to guarantee 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 15 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 [9]) for the virtualized environment.
- We identify a 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 environment.
- We propose `oticket`, a variant of the paravirtualized ticket spinlock that scales in both undersubscribed and oversubscribed virtualized environment.

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. Lastly, §7 compares our approach with previous research and §8 concludes.

2. Background

Spinlock is the basic building block for synchronization primitives inside the Linux kernel. As core count increases, scalability of spinlock becomes important; a recent study shows that non-scalable spinlock can cause performance collapse in an entire system [9, 10]. Therefore, to ensure scalability and fairness [4, 10, 16], the key design choices are to minimize shared cacheline contention as well as guarantee the FIFO ordering to prevent starvation with increasing core count. The Linux kernel uses ticket spinlock for fairness guarantee and recently adopted queue-based spinlock for better scalability [15]. 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`. At lock release, `head` is incremented for the next lock waiter to acquire the lock.

In virtualized environments, the introduction of vCPUs complicates the scalability of spinlocks. Figure 3 illustrates

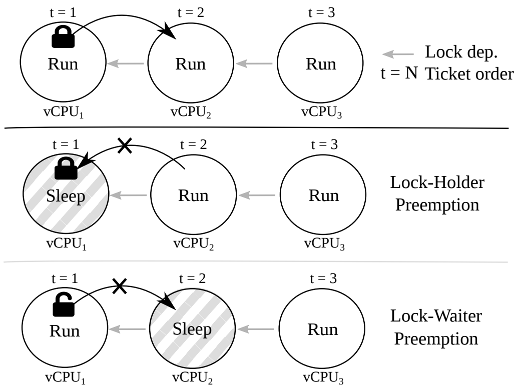

Figure 3: The issue of lock-holder preemption and lock-waiter preemption problem. Each of the circles represent a vCPU scheduled on a CPU. The *Sleep* state is the preempted vCPU whereas the *Run* state is the running one. *t* is the ticket order in which the vCPUs are waiting for the holder to release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

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 problems, Intel recently added the *Pause Loop Exiting* (PLE) feature to its processors [20]. In PLE, a processor detects whether a task is running a spin loop, executes `pause` instructions, and traps a guest OS to the hypervisor. However, methods to efficiently choose a lock holder (when a lock is acquired) or the next lock waiter (when a lock is released) remains an open problem. The recently proposed *paravirtual spinlock* [13, 19] addresses these problems without architectural support: a lock waiter first spins for a lock for a fixed iteration (fast path), and if it fails to acquire a lock, it voluntarily yields to other vCPUs by issuing a `halt` instruction (slow path). When a lock is released, the next waiter is woken up (kicking the next waiting vCPU). Voluntary yielding reduces the LHP problem and precisely waking the next vCPU reduces the LWP problem.

Surprisingly, our evaluation results in Figure 2 show sudden performance collapse at higher number of cores. This occurs due to the increase in `halt` exits after the 30th core (Figure 4). This is different from both LHP and LWP, and results from 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 certain point (30 vCPUs in this case), most vCPUs fail to acquire a lock during optimistic spinning and trap to the hypervisor at the same time (as evident in Figure 4). From this point, switching overhead between guest OS and hypervisor, and communication cost to wake other vCPUs start dominating resulting in 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`) that opportunistically changes the threshold of spin and opportunistically wakes up.

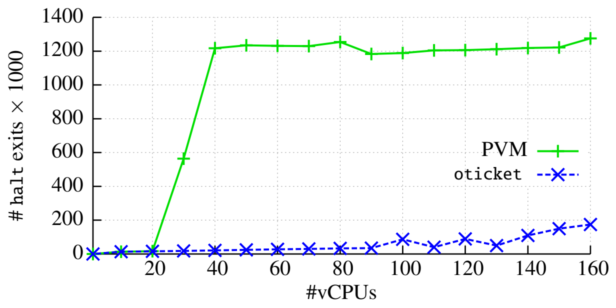

Figure 4: Number of `halt` exits that occur during a Linux kernel compile for two variants of spinlocks: in-stock ticket spinlock (PVM) and our `oticket` implementation.

3. Design

We propose a new spinlock algorithm for virtualized environments, named *opportunistic ticket spinlock* (or `oticket` for short). Our opportunistic ticket spinlock not only resolves the performance anomaly as shown in Figure 2, but also addresses the LHP and LWP problems, which are critical in 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 wake-up*. 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 performance collapse as shown in Figure 2.

In `oticket`, the spin duration is dynamically determined by the distance between the lock waiter and its holder. Closer waiters 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 chance for a lock holder to make progress. Consequently, this results in LHP problem mitigation. In `oticket`, as the distance of a lock waiter increases, the spinning iteration exponentially decreases (Lines 27–30 in Figure 5).

```
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 CPUs near their ticketing turn.

Opportunistic wake-up. Waking up a `halt`-ed vCPU takes a long time because after an unlocked vCPU is trapped to the hypervisor, the hypervisor schedules the target vCPU. To hide this wake-up latency, `oticket` implementation allows the unlocking vCPU to wake the next `N` lock waiters in advance (Lines 65–68 in Figure 5). Naturally, in conjunction with the opportunistic spinning scheme, this mitigates the LWP problem.

4. Implementation

We implemented our opportunistic ticket spinlock in Linux kernel 4.0 by replacing the paravirtual ticket spinlock in KVM with `oticket`. Our paravirtual spinlock is practical because of its minimal modification (lines starting with + in Figure 5) and without any changes to the size of its lock structure. In our locking function, `arch_spin_lock()`, `__ticket_distance()` calculates distance between a lock holder and waiter (Lines 6–11) before spinning, and `__ticket_lock _spinning()` makes its running vCPU yield to other vCPUs by executing a `halt` instruction (Line 43). In our unlock function, `arch_spin_unlock()`, `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 three questions:

- What is the impact of the two proposed techniques for the scalability of PVM? (§5.1)
- Does any other spinlock implementation solve the scalability issue? (§5.2)
- Does the opportunistic wake-up mechanism boost the performance of VMs in an oversubscribed environment? (§5.3)

Experimental setup. We created `VBench`[2], a fork of Mosbench [9], to evaluate the scalability of the host and hypervisor while running multiple virtual machines. Among the workloads in `VBench`, we chose the Linux kernel compile for evaluation since it is embarrassingly parallel and easily scales in high degree of parallelism without virtualization. To isolate the effect of I/O, we run the benchmark on top of the memory-based file system, `tmpfs`, while pre-loading all of the input source files before measuring the performance. We also set the number of parallel jobs of kernel compile to twice the number of cores, both for VM and Host. For the optimal performance of VMs, we pin each vCPU to a 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 40 vCPUs. In contrast, `oticket` achieves consistent performance improvement until 80 vCPUs (all physical

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

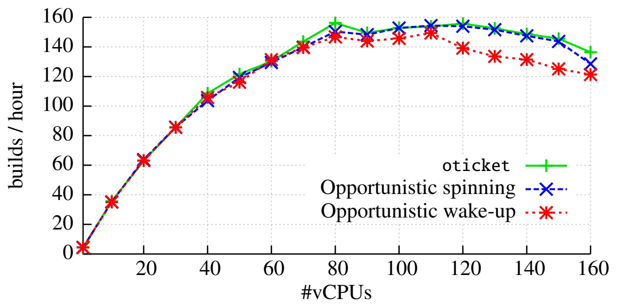

Figure 6: Performance impact of each optimization for the Linux kernel compile using modified paravirtualized spinlock interface. `oticket` is the implementation shown in Figure 5. Opportunistic wake-up and spinning are the individual ticket spinlock implementations that constitute `oticket`.

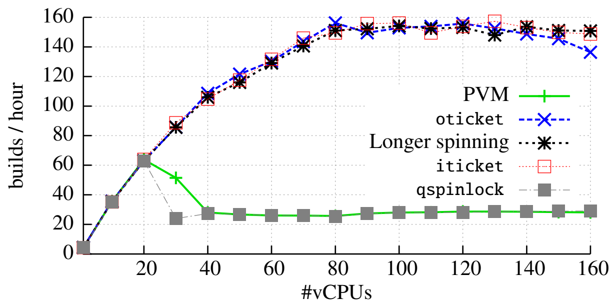

Figure 7: Linux kernel build performance for various FIFO based paravirtual spinlock implementations: Opportunistic ticket spinlock (`oticket`), ticket spinlock with longer spinning (Longer spinning), fine grained control of spinning period (`iticket`), and a MCS variant queue spinlock (`qspinlock`).

CPUs assigned) and does not show equivalent performance collapse in PVM until 160 vCPUs. 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 voluntary sleeping optimization for virtualized environments can result in performance collapse (*sleepy spinlock anomaly*) and `oticket` effectively avoids this.

Figure 6 shows the effectiveness of each scheme. `oticket` provides the best of both worlds—opportunistic spinning and opportunistic wake-up, 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 the use of both logical cores and increasing contention among vCPUs.

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 for further reducing cacheline contention (`qspinlock`) – and compare their performance to `oticket` and the stock ticket spinlock (PVM).

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

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_THRESHOLD` 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 inclining towards using variants of MCS lock [4] or its variant - queue spinlock [15] as they perform better on large NUMA machines due to less cacheline contention. There is an implementation of fast queue spinlock [15] (called `qspinlock`) with structure size the same as that of the ticket spinlock structure (4 bytes).

Figure 7 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 condition. But, as expected, this pays the price in an oversubscribed environment (Figure 8).

The `iticket` spinlock implementation performs 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 and all.

Figure 7 shows the performance of three alternatives against the existing spinlock implementation (PVM). We can observe that `qspinlock` also suffers from *sleepy spinlock anomaly* as there is no improvement in the virtualized environment. This proves that the anomaly can occur for any spinlock algorithm relying on 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 8 shows the compilation time of the Linux kernel inside a VM that has been co-scheduled with another VM. Both VMs are compiling the Linux kernel simultaneously.

We observed that `oticket` outperforms the existing ticket spinlock and its longer spinning alternative. 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 increase in cache contention and number of waiters. PVM suffers from

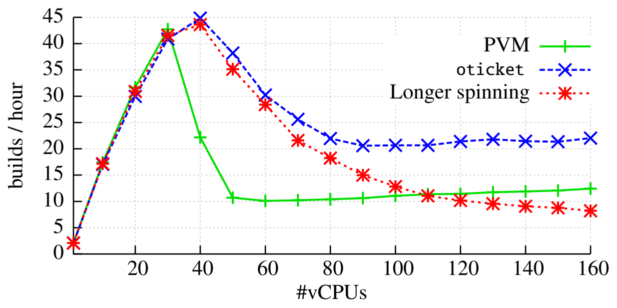

Figure 8: Performance impact of spinlock variants on Linux kernel compile for a single VM when co-scheduled with another exact VM instance running the same compilation.

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 inconsistencies in the vCPU scheduling. On the other hand, `oticket` suffers from the same problem, but performs better than these two techniques. Our opportunistic wake-up 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.

6. Discussion and Limitations

Paravirtualized interfaces are the second most efficient interfaces (after hardware assistance) for providing better performance in virtualized environments. They provide more control to the guest execution by enabling the coordination between the guest OS and hypervisor. However, we observe an anomaly existing for VMs with high core count. We believe that more coordination is necessary between the guest and hypervisor to improve scalability. Two possible solutions are tight integration of PLE with `oticket` and co-scheduling vCPUs at the time of yielding. This can further improve the performance and decrease the performance gap between bare metal and virtualized instance. This approach might solve the problem of performance drop for over-subscribed machines with high core count.

7. Related work

VM scheduling and synchronization has 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. [24] defined and addressed the lock synchronization issue (LHP) in the virtualized environment via scheduling hints. Later, paravirtual hooks were used in the spinlock [13] 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) [18] and Blocked Waiter Problem (BWW) [12, 21]. BWW occurs when the workload is using 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) [20]– 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. [8] 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 [4, 11, 15] 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 [7, 21] and co-scheduling issues [12, 13, 17, 18]. 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 [8, 14, 24, 25]. Later, Orathai et al. [23] came up with the approach of dedicating the vCPU with a physical CPU rather than co-scheduling. Furthermore, Song et al. [22] 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 preliminary study suggests that in addition to cache-contention bottleneck, the usage of spinlock, which guarantees strict FIFO ordering, is another culprit for 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.

In future, we would like to devise a formally verified generic `oticket` which is not susceptible to increasing core count and can perform equivalent to HVM in an over-subscribed environment. We will further extend our insight to other synchronization primitives with respect to virtualization.

9. Acknowledgment

We thank the anonymous reviewers, and our shepherd, Ketan Maheshwari, for their feedback. This work was supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korean government (MSIP) under contract ETRI MSIP/IITP[B0101-15-0644].

References

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

[2] The CPU Scheduler in VMware ESX 4.1. 2010.

[3] M7: Next Generation SPARC. 2014.

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

[5] HP to Transform Server Market with Single Platform for Mission-critical Computing, 2015. http://www8.hp.com/us/en/hp-news/press-release.html?id=1147777.

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

[7] K. Adams and O. Agesen. 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*, ASPLOS, 2006.

[8] J. Ahn, C. H. Park, and J. Huh. 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*, MICRO, 2014.

[9] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2010.

[10] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Ottawa Linux Symposium*, OLS, 2012.

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

[12] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the Blocked-waiter Wakeup Problem for Virtualized Multicore Applications. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC, 2014.

[13] T. Friebel. How to Deal with Lock-Holder Preemption. 2008.

[14] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-based Coordinated Scheduling for SMP VMs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2013.

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

[16] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-Copy Update. In *Ottawa Linux Symposium*, OLS, 2002.

[17] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE, 2005.

[18] J. Ouyang and J. R. Lange. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE, 2013.

[19] K. Raghavendra, S. Vaddagiri, N. Dadhania, and J. Fitzhardinge. Paravirtualization for Scalable Kernel-Based Virtual Machine (KVM). In *Cloud Computing in Emerging Markets (CCEM)*, 2012.

[20] M. Righini. Enabling Intel Virtualization Technology Features and Benefits, 2010.

[21] X. Song, H. Chen, and B. Zang. Characterizing the Performance and Scalability of Many-core Applications on Virtualized Platforms. In *Fudan University - Technical Report*, 2010.

[22] X. Song, J. Shi, H. Chen, and B. Zang. Schedule Processes, Not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys, 2013.

[23] O. Sukwong and H. S. Kim. Is Co-scheduling Too Expensive for SMP VMs? In *Proceedings of the ACM EuroSys Conference*, Salzburg, Austria, Apr. 2011.

[24] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM, 2004.

[25] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic Adaptive Scheduling for Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC, 2011.