Mitigating Excessive vCPU Spinning in VM-Agnostic KVM

Kenta Ishiguro
Keio University
Japan
kentaishiguro@sslab.ics.keio.ac.jp

Naoki Yasuno
Keio University
Japan
naokiyasuno@sslab.ics.keio.ac.jp

Pierre-Louis Aublin
Keio University
Japan
pl@sslab.ics.keio.ac.jp

Kenji Kono
Keio University
Japan
kono@sslab.ics.keio.ac.jp

Abstract

In virtualized environments, oversubscribing virtual CPUs (vCPUs) on physical CPUs (pCPUs) is common to utilize CPU resources efficiently. Unfortunately, excessive vCPU spinning, which occurs when a vCPU is waiting in a spin loop for an event from a descheduled vCPU, causes serious performance degradation. Usually, the VM-agnostic hypervisor tries to prevent excessive vCPU spinning by rescheduling vCPUs when an excessive spin is detected by hardware support for virtualization.

This paper investigates the effectiveness of KVM vCPU scheduler and shows it fails to avoid excessive vCPU spinning in many opportunities. Our in-depth analysis reveals simple modifications to KVM (41 LOC) improve the mitigation of excessive vCPU spinning. We have identified three problems: 1) scheduler mismatch, 2) lost opportunity, and 3) overboost. The first problem comes from the mismatch between the KVM vCPU scheduler and the Linux scheduler. The second and third problems come from an inefficient algorithm for choosing the next candidate vCPU to be scheduled. Our simple modifications gracefully resolves the problems and the performance improves by up to 80 %. Our results imply the VM-agnostic hypervisor can resolve excessive vCPU spinning more gracefully than previously believed.

CCS Concepts: • **Software and its engineering**; • **Virtual machines**;

Keywords: Virtualization, Hypervisor

VEE '21, April 16, 2021, Virtual, USA

ACM ISBN 978-1-4503-8394-3/21/04…$15.00
https://doi.org/10.1145/3453933.3454020

ACM Reference Format:
Kenta Ishiguro, Naoki Yasuno, Pierre-Louis Aublin, and Kenji Kono. 2021. Mitigating Excessive vCPU Spinning in VM-Agnostic KVM. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21), April 16, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453933.3454020

1 Introduction

Virtualization is widely used in cloud computing platforms. To improve hardware utilization and reduce energy consumption, the cloud providers are striving to oversubscribe hardware resources, consolidating multiple virtual machines (VMs) on a single physical machine. However, oversubscription does not come for free: it requires multiplexing of virtual CPUs (vCPUs) on physical CPUs (pCPUs).

Even more, oversubscription violates an underlying assumption of the operating system (OS) design: OSes assume all the CPUs to be active, and even if halted, they can respond to interrupts immediately. If pCPUs are oversubscribed, the execution of vCPUs are preempted by the hypervisor to schedule vCPUs, and vCPUs are not always active or cannot respond to interrupts immediately. The violation of the OS design assumption results in a well-known problem of *excessive vCPU spinning* [1, 2, 5, 7, 9, 11, 12, 23–25, 27–30, 33, 37–39, 41]. Excessive vCPU spinning occurs when a vCPU is waiting in a tight loop for an event that a descheduled vCPU will cause. An event-waiting vCPU spins in a tight loop until an event-sending vCPU is scheduled by the hypervisor. Excessive vCPU spinning comes from many variants of the scheduling problems: 1) lock-holder preemption (LHP) [33], 2) lock-waiter preemption (LWP) [23], and 3) delayed response to interrupts.

Excessive vCPU spinning is hard to solve in VM agnostic hypervisors. It stems from a semantic gap between the hypervisor and guest OSes, and the hypervisor is ignorant of the *contexts* in which a vCPU is running. To mitigate the problem of excessive vCPU spinning, recent hardware support for virtualization supports the detection of long-running

tight loops. When an excessive spinning is detected, a processor raises an event called *Pause Loop Exit* (abbreviated as PLE) [26], and the control is transferred to the hypervisor. The hypervisor reschedules vCPUs to solve the root cause of excessive spinning.

Recent hypervisors such as KVM [13] have incorporated a mechanism to mitigate excessive vCPU spinning. It is expected that excessive vCPU spinning is not an issue anymore in current hypervisors. When a PLE event occurs in the guest, KVM *boosts* vCPUs that can be the root cause of the excessive spinning, and attempts to schedule event-sending vCPUs as soon as possible. In spite of the mitigation against the excessive vCPU spinning, KVM still suffers from non-negligible overheads due to the excessive spinning. Performance of some benchmarks is degraded up to 45 % in our evaluation. PLE events are raised continuously more than 600 times at the same code location in the guest, although KVM attempts to resolve the root cause at every PLE event.

Our in-depth analysis reveals the KVM vCPU scheduler fails to solve the excessive spinning for three reasons: 1) *scheduler mismatch*, 2) *lost opportunity*, and 3) *overboost*. *Scheduler mismatch* is peculiar to integrated hypervisors where the host OS scheduler schedules other threads, based on its own policy, together with vCPUs without any distinction. To mitigate excessive vCPU spinning, the vCPU scheduler gives a hint on vCPU scheduling to the host OS scheduler. Scheduler mismatch occurs if the scheduling hints are eventually ignored by the host OS scheduler, probably because the hints contradict the host scheduling policy. This problem is not restricted to KVM. The semantic gap between the vCPU scheduler and the host OS scheduler can lead to a similar problem in VirtualBox [22] and VMware workstation [36]. Although both hypervisors delegate vCPU scheduling to the host, they do not adjust vCPU priorities to mitigate the excessive spinning.

Lost opportunity and *overboost* are caused by the semantic gap between the guest operating system and the hypervisor. Since the vCPU scheduler cannot know exactly which vCPU to be boosted to mitigate the excessive spinning, it sometimes gives false hints to the host scheduler; it sometimes misses the opportunities to boost vCPUs, which is called *lost opportunity*, or falsely boosts vCPUs that should not be boosted, which is called *overboost*. It is not straightforward to select the candidate vCPUs for boosting because of the semantic gap; the hypervisor must carefully collect thin shreds of information to infer root causes of excessive spinning.

To compensate for the negative impact of scheduler mismatch, lost opportunity, and overboost, we have modified the KVM vCPU scheduler to incorporate 1) vCPU debooster and 2) strict vCPU boosting. The vCPU debooster mitigates the scheduler mismatch problem by lowering the priority of the vCPU preempted by PLE. Since lowering the priority does not interfere with other high-prioritized threads in the host, the host scheduler is less likely to ignore the hint of lowering the priority. The strict vCPU boosting mitigates the lost opportunity and overboost. It collects the information on IPI senders and receivers, and strictly boosts IPI receivers only when necessary.

Our evaluation results show that 1) for PLE-intensive benchmarks, our modified KVM gracefully improves the performance by up to 80 %, 2) for less PLE-intensive benchmarks, the performance is not degraded compared with the baseline KVM, and 3) the fairness of scheduling is not compromised by introducing the debooster and the strict boost.

The paper is organized as follows. Section 2 demonstrates KVM suffers from non-negligible overheads due to continuous PLE occurrences. Section 3 analyzes the root causes of continuous PLE occurrences. Section 4 presents our mitigation against scheduler mismatch, lost opportunity, and overboost. Section 5 shows the evaluation results. Section 6 relates our work to others, and Section 7 concludes the paper.

2 Background and Motivation

Excessive vCPU spinning is a widely known problem that has been thoroughly addressed by research and development communities [1, 2, 5, 7, 9, 11, 12, 16, 23–25, 27–30, 33, 37–39, 41]. Modern hypervisors such as KVM have incorporated solutions to excessive vCPU spinning, and it is expected that excessive vCPU spinning is not an issue anymore. Unfortunately, as we show in this section, KVM fails to solve excessive vCPU spinning comprehensively; it still suffers from non-trivial overheads due to excessive vCPU spinning.

2.1 Excessive vCPU Spinning

In consolidated environments, virtual CPUs (vCPUs) share a limited number of physical CPUs (pCPUs) to utilize the resources efficiently. Guest operating systems (OSes) are given an illusion that their (virtual) CPUs are running continuously but in reality their executions are interrupted by the hypervisor to schedule vCPUs. This interrupted execution of vCPUs violates the assumption of operating-system design: the OSes assume all the CPUs to be active, and even if halted, they can respond to interrupts quickly.

The violation of this underlying assumption results in a well-known problem of *excessive vCPU spinning* in VM agnostic hypervisors; a vCPU spins for a long time without making any progress. Excessive vCPU spinning typically happens in two scenarios. First, if a vCPU holding a spin lock is scheduled out by the hypervisor, another vCPU waiting for the spin lock cannot make any progress until the lock holding vCPU is re-scheduled. This problem is called the LHP (lock holder preemption) problem [33]. Older versions of Linux (until version 4.1) supported ticket spin locks in which a lock is acquired in requesting order. Recent Linux has dropped the support because the ticket spin lock amplifies the problem of vCPU spinning [32]. A vCPU waiting for

a ticket spin lock cannot make any progress until all the vCPUs preceding it in ticket requesting order are scheduled.

Second, inter-processor interrupts (IPIs) cause excessive vCPU spinning. If a vCPU is not scheduled when an IPI is sent to it, it cannot receive it immediately and the interrupt handling is delayed. For example, TLB shootdown is implemented with IPIs. When a processor updates a page table, it sends IPIs to other processors and waits for acknowledgements. This wait is implemented with a spinning loop because the acknowledgement is sent back immediately on a bare-metal machine. If a recipient vCPU is not scheduled, the acknowledgement is delayed until it is re-scheduled. This delay makes the IPI sender to spin excessively.

To mitigate vCPU spinning, the hardware-level support for virtualization provides the function to detect excessive vCPU spinning and enable the hypervisor to re-schedule vCPUs. Modern processors are equipped with a special instruction (PAUSE instruction in Intel x86) that gives a hint to the processor that the code is in a spinning loop. The use of PAUSE instructions in a spinning-loop is strongly recommend by Intel to avoid the memory order violation and unnecessary pipeline flushes.

The Pause Loop Exiting (PLE) [26] hardware assist feature of Intel x86 processors checks the interval between consecutive PAUSE instructions performed in kernel mode. If the interval is shorter than *PLE_gap*, a pre-defined parameter, the vCPU is considered to be spinning. If the spinning continues beyond another pre-defined parameter, *PLE_window*, a VM-Exit is triggered to transfer control to the hypervisor, which de-schedules the spinning vCPU and schedules another vCPU. AMD supports PF (Pause Filter) [3] which is essentially the same as PLE. This paper focuses on PLE but can be applied to PF as well.

2.2 KVM Solution

To leverage existing kernel functionalities, some hypervisors are integrated with a host OS kernel. The integrated hypervisor schedules vCPUs in collaboration with the host OS scheduler [21, 35]. For example, KVM is integrated with the Linux kernel and thus KVM vCPU scheduler cooperates with the Linux scheduler. Instead of scheduling vCPUs directly, KVM gives some hints to the Linux scheduler so that it can prioritize or deprioritize vCPUs appropriately. A vCPU given higher priority by KVM is called a *boosted* vCPU.

By making use of this mechanism, KVM mitigates excessive vCPU spinning. KVM de-schedules a vCPU that caused PLE and chooses another vCPU to boost, expecting that the boosted vCPU resolves the root cause of the PLE. KVM chooses a boosted vCPU based on the candidate vCPU selection [25]. Originally the vCPU candidate selection was designed only for the LHP problem (Linux version $\leq$ 5.2) [19]. Recently it has been extended to reduce IPI latency (Linux version $\geq$ 5.3) [17, 18].

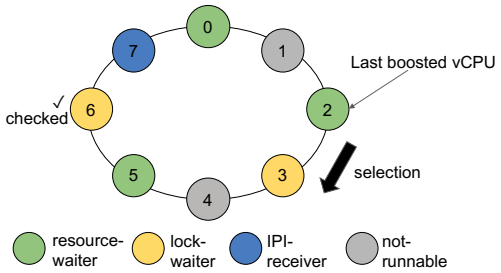

Figure 1. Candidate vCPU selection in KVM. Lock-waiter $vCPU_3$ (turned into "checked") and non-runnable $vCPU_4$ are skipped, and resource-waiter $vCPU_5$ is boosted. Next, lock-waiter $vCPU_6$ is boosted because it is already checked.

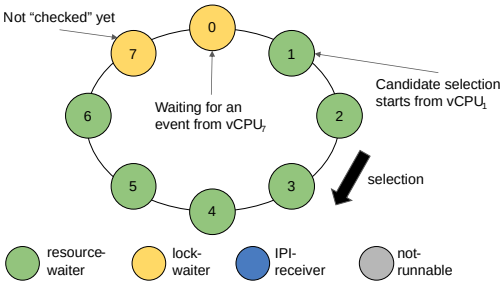

Figure 2. Worst-case scenario. $vCPU_0$ is scheduled repeatedly by the Linux scheduler until $vCPU_7$ is boosted.

Under the candidate vCPU selection, vCPUs are classified into four categories: 1) resource-waiter, 2) lock-waiter, 3) IPI-receiver, and 4) not-runnable. A resource-waiter is a vCPU that has used up its time slice and waiting for another time slice is assigned. A lock-waiter is a vCPU that has been preempted because of PLE; a lock-waiter is waiting in a tight loop for some event to happen. This name comes from the fact that the original vCPU candidate selection was designed solely for LHP. An IPI-receiver is a vCPU that has been halted and an IPI has been sent to. A not-runnable is a vCPU that has been halted but no IPI has been sent to.

KVM selects a vCPU that should be boosted next in the round-robin fashion. The candidate vCPU selection consists of two rounds. In the first round, KVM skips lock-waiter vCPUs to avoid boosting a lock-waiter that is less likely to make progress. In other words, KVM boosts the resource-waiter and IPI-receiver vCPUs in turn in the first round. By doing so, a resource-waiter vCPU is expected to cause an event a lock-waiter vCPU is waiting for, or an IPI-receiver is expected to acknowledge the pending IPI. Note that after the boosted vCPU is preempted, the Linux scheduler chooses another vCPU to run based on its own scheduling algorithm.

Figure 1 illustrates the candidate vCPU selection. All the vCPUs in the same VM form a circle. When a vCPU exits with PLE, KVM searches a candidate vCPU, following this circle. If a vCPU next in this circle is a resource-waiter or

Table 1. Benchmarks

Benchmark name	Workload
mosbench.gmake	Parallel build system
mosbench.psearchy	In-memory parallel search & indexer
parsec.dedup	Compression with deduplication
parsec.ferret	Content-based similarity search
parsec.swaptions	Pricing of a portfolio of swaptions
parsec.streamcluster	Online clustering of input stream
parsec.vips	Image processing
parsec.raytrace	Real-time raytracing
pbzip2	Data compressor
dbench	Filesystem I/O
ebizzy	Common web application servers
hackbench	Unix-socket or pipe stress

Table 2. Experimental Environment

Machine	Dell PowerEdge R440
CPU	2.1GHz Intel Xeon Silver 4110
Core	8 cores (no hyperthreading)
Memory	64 GB
Host kernel	Linux kernel 5.6.0
Guest OS	Ubuntu 18.04 LTS
Guest kernel	Linux kernel 4.15
Guest #vCPUs	8 vCPUs
Guest Memory	16 GB

IPI-receiver, it is selected to boost unconditionally. If it is a lock-waiter, it is labeled as "checked" and then skipped to avoid boosting a vCPU less likely to make progress. If a lock-waiter is already labeled as "checked", KVM selects it to be boosted because all the vCPUs boosted after it caused PLE and the root cause of PLE has been resolved. The "checked" label is removed after the vCPU is boosted.

This design is expected to resolve the root cause of PLE with, in the worst case, roughly $2 \times$ *#vCPUs* attempts to boost vCPUs. Figure 2 shows the worst case where $vCPU_0$ causes a PLE event waiting for an event from $vCPU_7$. Suppose $vCPU_7$ is a lock-waiter. It will be boosted in the second round, and thus there will be roughly $2 \times$ *#vCPUs* attempts to boost vCPUs until $vCPU_7$ is boosted and the root cause of the PLE event on $vCPU_0$ is solved. Every time after each vCPU is boosted, $vCPU_0$ can be selected to run by the Linux scheduler even though the root cause has not been resolved yet. In the worst scenario, PLE events occur *continuously* on the same vCPU ($vCPU_0$ in this case) for $2 \times$ *#vCPUs* times.

2.3 Ineffectiveness of KVM Solution

To confirm whether the KVM solution is effective or not, we measure the number of PLE events on several benchmarks shown in Table 1 on the experimental environment shown in Table 2. We run two VMs on the same host: the first one executes a benchmark shown in Table 1, and the other one executes the CPU-intensive `swaptions` benchmark which rarely causes PLE events.

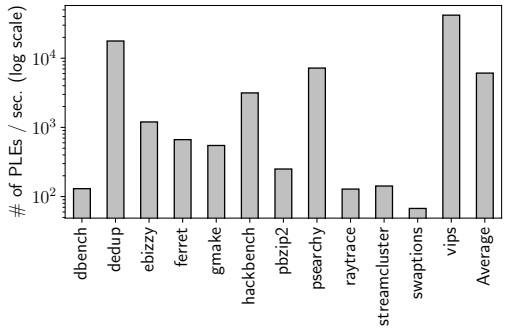

Figure 3. Number of PLE events per second.

Figure 3 shows the number of PLE events per second in each benchmark in log scale. Some benchmarks show an extremely large number of PLE events: `vips`, `dedup`, and `psearchy` show respectively 42,000, 18,000, and 7,200 PLE events per second. In the case of `vips`, a PLE event happens every 50,000 cycles, corresponding to 23.8μs on our machine.

To determine whether this high frequency of PLE events is reasonable or not, we investigate how often PLE events occur *continuously* in the benchmarks, that this to say if they occur at the same code location on the same vCPU without being interleaved with PLEs from other code locations or VM exits other than PLE. If the candidate vCPU selection in KVM works as expected, a sequence of continuous PLEs is not longer than $2 \times$ *#vCPUs* because the root cause should be resolved after the two-round boosting of vCPUs.

Figure 4 shows the CDF of the length of continuous PLE events in each benchmark. If the candidate vCPU selection resolves excessive vCPU spinning perfectly, the length should not exceed 16 (8 vCPUs in our setting). However, we observe that, except for `ebizzy` and `raytrace`, more than 50 % of the PLE events come from continuous PLE events whose length exceeds 16. Surprisingly, more than half of the PLE events come from 100+ PLE events in a row in `dbench`, `ferret`, and `swaptions`. A large proportion of all the PLE events occur continuously once a PLE event occurs. This implies that KVM candidate vCPU selection is not working properly.

3 Analysis of KVM Behaviors

In this section, we analyze KVM behaviors when it encounters PLE events and investigate the reason KVM vCPU scheduling causes continuous PLE occurrences.

We first identify the root causes of PLE events, and then show a detailed analysis of KVM behaviors that fail to solve PLE events. We have identified three issues: 1) scheduler mismatch, 2) lost opportunity, and 3) overboost. Section 3.2 shows *scheduler mismatch* in which a vCPU boosted by the

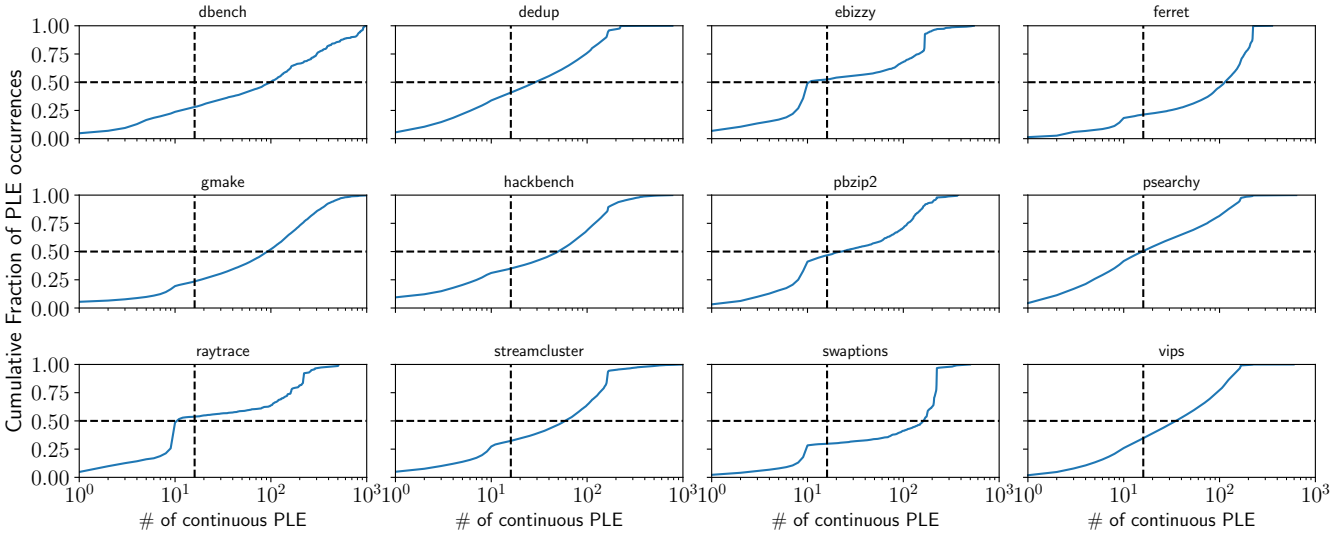

Figure 4. Proportion of the length of continuous PLE events.

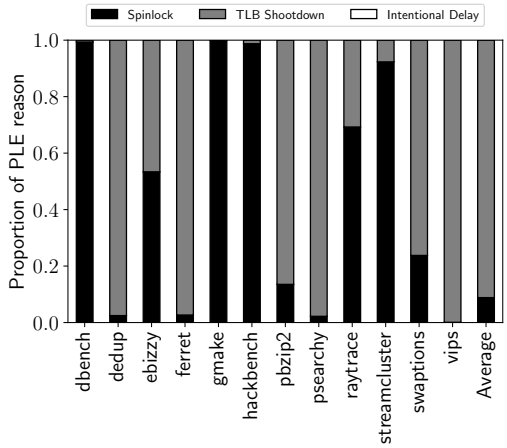

Figure 5. PLE reasons

KVM vCPU scheduler is not scheduled by the Linux CPU scheduler. Section 3.3 shows *lost opportunity* and *overboost* in vCPU re-scheduling. Figure 6 shows how often these three cases occur in the benchmarks. Scheduler mismatch occurs from 2.6 % to 64.7 % (17.7 % on average); lost opportunity occurs from 0.0 % to 36.4 % (9.0 % on average); and overboost occurs from 0.1 % to 7.3 % (3.3 % on average). PLE events are handled successfully 70.0 % on average. Note that "success" means the KVM vCPU scheduler succeeds to schedule a vCPU other than the PLE-causing vCPU.

3.1 PLE Reasons

To solve excessive vCPU spinning quickly, the hypervisor should take the PLE reason into account when it decides which vCPU should be boosted. Spinning loops with PAUSE instruction are ubiquitous in the kernel, and used for various purposes. To decide which vCPU to be boosted, the vCPU scheduler should identify the PLE reason and change the criteria for vCPU selection for each reason. Because of the semantic gap between the guest OS and hypervisor, PLE events can not tell anything about the PLE reason. They simply detect excessive vCPU spinning and give the hypervisor a chance to re-schedule vCPUs.

To investigate which kernel functions causes PLE, we trace the guest VMs vCPUs' instruction pointers. The instruction pointer where a vCPU caused a PLE event is a good clue to revealing why the vCPU is waiting for another one. For instance, if a vCPU exits due to the `native_queued_spin_lock_slowpath` function (a spinlock implementation in Linux), the vCPU might be waiting for completion of a lock-holder vCPU's execution in a critical section.

Table 3 summarizes functions which cause PLE in the Linux kernel and what those functions use PAUSE instruction for. The results show that reasons of PLE are three functionalities in Linux: *spinlock*, *TLB shootdown*, and *intentional delay*. Figure 5 shows the proportion of PLE events in each benchmark whose root cause is spinlock, TLB shootdown, or intentional delay. Although the major PLE reason differs from benchmark to benchmark, spinlock and TLB shootdown are two major causes of PLE events, respectively causing 8.8 % and 91.2 % of the PLE events on average. The intentional delay is negligible (less than 0.1 %). As shown in Figure 5, a major cause for PLE events in most of the benchmarks is vCPU executing spinlock functions, in more than 53 % in 6 out of 12 benchmarks.

Another major cause of PLE events is TLB shootdown. TLB shootdown is a kernel-level operation for TLB synchronization. PLE events related to TLB shootdown are caused in `smp_call_function_many`, which sends an inter-processor

Table 3. PLE reasons and functions. "spinlock" includes low-lever primitives used in higher-level synchronization primitives such as semaphore

Function	Functionalitiy
d_alloc_parallel	spinlock
d_lookup	spinlock
do_get_write_access	spinlock
hrtimer_active	spinlock
jbd2_journal_dirty_metadata	spinlock
jbd2_journal_file_buffer	spinlock
jbd2_journal_write_metadata_buffer	spinlock
ktime_get	spinlock
ktime_get_snapshot	spinlock
ktime_get_ts64	spinlock
ktime_get_update_offsets_now	spinlock
ktime_get_with_offset	spinlock
mutex_spin_on_owner	spinlock
native_queued_spin_lock_slowpath	spinlock
osq_lock	spinlock
path_init	spinlock
queued_write_lock_slowpath	spinlock
queued_read_lock_slowpath	spinlock
rwsem_spin_on_owner	spinlock
rwsem_down_write_failed	spinlock
rwsem_down_write_failed_killable	spinlock
tick_nohz_next_event	spinlock
try_to_wake_up	spinlock
smp_call_function_many	TLB shootdown
smp_call_function_single	TLB shootdown
delay_tsc	Intentional delay

interrupt (IPI) to multiple cores. This function is not solely for TLB shootdown but all the calls to it that caused PLE events are for TLB shootdown. A PLE event can occur in this function because a vCPU needs to wait for all other vCPUs to flush their own TLBs if it flushes a remote TLB. This wait is implemented with a spinning loop and can cause PLEs if the vCPU receiving an IPI has been preempted by the vCPU scheduler. PLE events caused by TLB shootdown occur in more than 65 % in 6 out of 12 benchmarks.

Intentional delay refers to the case where the `PAUSE` instruction is used to insert a short delay by waiting specific cycles based on the timestamp counter. For example, `delay_tsc` uses a spinning loop to insert a delay. As can be seen from Figure 5, intentional delay is negligible and thus we do not consider reducing its associated PLE events.

These results demonstrate that KVM still suffers from excessive vCPU spinning caused by the lock-holder preemption (LHP) problem and TLB shootdown synchronization despite attempts to mitigate their negative effects [17–19].

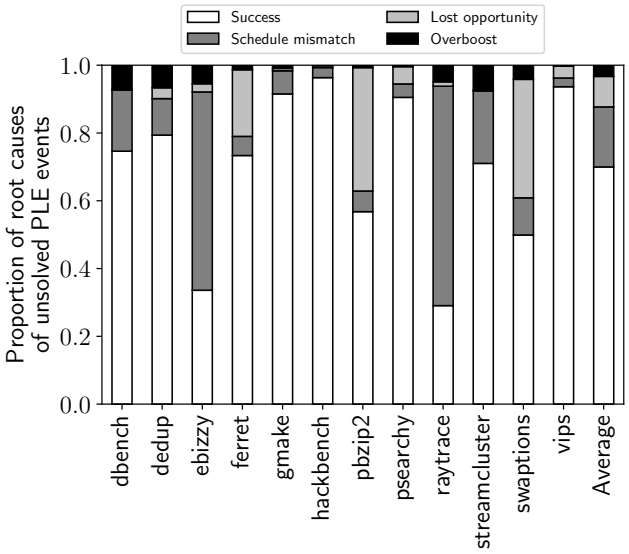

Figure 6. Proportion of unsolved PLE events root causes.

3.2 Scheduler Mismatch

Scheduler mismatch is a problem that integrated hypervisors can suffer from. In the integrated hypervisor, the vCPU scheduler does not schedule their vCPUs directly. Instead, it gives a scheduling hint to the host OS scheduler, which schedules vCPUs based on its own scheduling policy, together with other threads in the host. Scheduler mismatch occurs if the scheduling hint is eventually ignored by the host OS scheduler. Since the host OS scheduler decides which vCPU to be scheduled according to its own policy, it sometimes considers other vCPUs to be prioritized than the one boosted by the vCPU scheduler.

According to Figure 6, scheduler mismatch with the Linux scheduler is one of the root causes of continuous PLE events in KVM. When a vCPU is boosted by the KVM vCPU scheduler, it is labeled as "boosted" in the Linux scheduler. The default Linux scheduler, CFS (Completely Fair Scheduler), does not always schedule it immediately in order to keep the fairness among vCPUs (and other threads). It computes "virtual runtime" for each vCPU (or thread) and schedules the one with the lowest virtual runtime. The virtual runtime is calculated by *execution time*/*weight*, where *execution time* is the actual execution time in the previous time slice; CFS prefers vCPUs running for a short time. If the boosted vCPU virtual runtime is much larger than that of the vCPU with the lowest virtual runtime, CFS chooses *not* to run it.

The virtual runtime of a vCPU causing a PLE event tends to be small because its execution has been preempted by the PLE. On the other hand, the virtual runtime of a boosted vCPU can be large especially when a resource-waiter has been boosted. Since the resource-waiter has used up its time slice, its virtual runtime is large (the execution time is equal

to the time slice). In the worst case, if the PLE-causing vCPU happens to be in the same runqueue as the boosted vCPU, it is chosen again to run and exits due to the PLE triggered immediately after the rescheduling. Since its virtual runtime remains small, it is rescheduled repeatedly, resulting in continuous PLE events.

At first glance, there is no need to keep fairness between vCPUs in the same VM. However, the host OS scheduler cannot be aware of the vCPU semantics because it treats vCPUs completely in the same way as other normal threads. The host OS scheduler adheres to its scheduling algorithm even if KVM makes a request for boosting a vCPU or if vCPUs are in the same resource management group (i.e. `cgroup`).

3.3 Lost Opportunity and Overboost

To remove the cause of a PLE event quickly, which vCPU being scheduled after the PLE event occurs is important. The candidate vCPU selection is a non-trivial task in VM-agnostic hypervisors because of the semantic gap between hypervisors and guest VMs.

Lost opportunity. As explained in Section 2.2, an IPI receiver that has halted is always a candidate for a boost. Thanks to this rule, hypervisors can suppress excessive spinning due to TLB shootdown because IPI recipients can be boosted even if they are halted. However, TLB shootdown is still the major cause of PLE events in a couple of PLE-intensive workloads as shown in Figure 3. This comes from the fact that KVM vCPU scheduler was originally designed to deal with the LHP (lock-holder preemption) problem. Intel x86 raises PLE events only when the guest is in kernel mode, and thus only the spinlock waiter in the kernel can be detected on Intel x86. Therefore, the KVM vCPU scheduler wakes up vCPUs in *kernel mode* only (while the spinlock holder is running in kernel mode). This assumption does not hold for IPI recipients; they can be in either kernel or user mode. If IPI recipients are in user mode, KVM vCPU scheduler does not wake them up even if they should respond to IPIs. We refer to this problem as *lost opportunity*. Figure 6 shows the KVM vCPU scheduler does not boost any vCPU due to lost opportunity on average in 9.0 % of all benchmarks. The lost opportunity problem can lead to a large number of continuous PLE events because the IPI sender causes PLE events repeatedly until the recipient is scheduled while the recipient is not candidate for a boost.

Overboost. Another possible problem arises if an IPI receiver that has halted is always a candidate for a boost. Suppose that a vCPU exits due to PLE when it is waiting for the lock-holder. The KVM vCPU scheduler should schedule the lock-holder vCPU instead of the PLE-causing vCPU while the lock-holder vCPU is never halted at the time because it is executing in a critical section. However, if another vCPU has sent asynchronous IPIs to a halted vCPU, the halted vCPU is also candidate for the boost. This phenomenon introduces unnecessary boosting and PLE events. We refer to this problem as *overboost*. Figure 6 shows the KVM vCPU scheduler boosts the IPI recipient vCPU which has halted even if the PLE-causing vCPU has not sent an IPI to the IPI recipient on average in 3.3 % of all benchmarks.

3.4 Intentional Delay

As described in Section 3.1, the intentional delay is another reason of continuous PLEs. Since the intentional delay is not dominant in the benchmarks, we do not deeply discuss solving this problem. One possible approach is to adjust PLE_window to tolerate the delay.

4 Design and Implementation

This section presents the design and implementation of our proposed mitigations against excessive vCPU spinning in guest VMs running on top of KVM[1]. The goal of our mitigations is to suppress PLE events due to scheduler mismatch, lost opportunity and overboost, which will result in a reduction of the total number of PLE events, without scarifying the VM agnostic feature of KVM.

4.1 vCPU Debooster

We propose a mechanism called *debooster* to alleviate the negative impact of the scheduler mismatch problem. This mechanism *deboosts*, or lowers the priority of, vCPUs that are preempted because of PLE. The debooster helps to boost a target vCPU which is not boosted because of the host OS scheduler policy even if the KVM vCPU scheduler makes a request to boost it.

As explained in Section 3.2, the host OS scheduler does not always schedule the boosted vCPU immediately to keep the fairness between two vCPUs. This induces a large number of continuous PLE events by scheduling the PLE-causing vCPU repeatedly before the root cause is resolved. When the KVM vCPU scheduler makes a request for boost, the debooster compares the virtual runtime of the boosted vCPU with the PLE-causing vCPU if both vCPUs are in the same runqueue. If the virtual runtime difference is larger than a threshold, the debooster increases the virtual runtime of the PLE-causing vCPU up to the virtual runtime of the boosted vCPU minus the threshold to convince the host OS scheduler to pick up the boosted vCPU as a next task. By lowering the PLE-causing vCPU priority, the host OS scheduler always schedules a boosted vCPU instead of the PLE-causing vCPU because the debooster makes the virtual time difference between two vCPUs not too large to boost.

This design does not violate the fairness of vCPU scheduling. The host OS scheduler can keep the fairness as usual among the VMs and other threads on the host because the debooster does not raise the priority of the boosted vCPU.

[1]Source code is available at `https://github.com/sslab-keio/ple-kvm`.

Furthermore, the debooster results in more efficient utilization of CPU time. Without the debooster, the CPU time for the PLE-causing vCPU will be wasted to execute pause-loop until the host OS scheduler consider the boost is fair. However, the CPU time can be used for another task by deferring scheduling of PLE-causing vCPUs. Since the PLE-causing vCPU cannot make any progress until the root cause is resolved, it is reasonable to defer its execution.

Note that vCPU pinning is effective against the scheduler mismatch problem by ensuring that each per-core runqueue does not contain two or more vCPUs in the same VM. The boosted vCPU is not disturbed by PLE-causing vCPUs. However, it requires careful configuration of vCPU affinity and reduces the degree of oversubscription. As a consequence, vCPU pinning is usually used only for high-performance settings [42]. The debooster allows the hypervisors to delegate management of load-balancing to the host OS scheduler while mitigating the scheduler mismatch problem.

4.2 Strict Boost for IPI Recipients

As explained in Section 3.3, the candidate vCPU selection for a boost is important whereas KVM still suffers from two problems: *lost opportunity* and *overboost*. In the current KVM vCPU scheduler, the halted IPI recipients or the resource waiters are candidates for a boost in the first iteration. The lock waiters are also candidates in the second iteration. Therefore, IPI recipients in user mode are excluded from the candidates for a boost. If an IPI is sent to a vCPU in user mode, the IPI-sending vCPU needs to wait in a tight loop for its acknowledgement. Besides, IPI recipients should be boosted to resolve PLE events due to IPI synchronization, and resource waiters should be high priority for a boost if the PLE-causing vCPU exits because of a lock-holder preemption.

We introduce *strict boost* to mitigate the lost opportunity and overboost problems. Strict boost prepares two rules of candidate vCPU selection for either spinlock or TLB shootdown (see Section 2.2). For TLB shootdown, strict boost adds to the candidates for boost the IPI recipients of not only halted vCPUs but also vCPUs in user mode. For spinlock, strict boost prioritizes the resource waiters by limiting the candidate vCPUs in the first iteration.

Strict boost is stricter than the KVM vCPU scheduler in the following three points: 1) IPI recipient in user mode is boosted, 2) IPI recipient is boosted only when an IPI-sending vCPU causes a PLE event, and 3) IPI recipient is not boosted if it is likely to have responded to the IPI. By selecting a candidate vCPU in a strict way, the strict boost can add wide range of vCPUs to the candidates for boost without amplifying the negative side effects when a PLE event occurs because of a spinlock.

4.3 Implementation

We have implemented our mitigations on Linux/KVM with Linux kernel version 5.6.0. Our implementation modifies less than 50 lines of code in KVM and does not require code modifications at the guest.

vCPU Debooster is implemented without modifying the KVM vCPU scheduler or the Linux scheduler. The Linux scheduler, designed to work with the KVM vCPU scheduler, provides the `yield_to_task` interface through which the vCPU scheduler can give scheduling hints. This interface is implemented for each Linux scheduler and "translates" the hints into the terminology that each scheduler can understand[2]. Debooster is implemented inside the interface.

Strict boost is implemented in the KVM vCPU scheduler and the KVM virtual IPI handler. Once a vCPU issues an IPI to another vCPU, the virtual IPI handler in KVM is invoked. The virtual IPI handler monitors all IPIs to record IPI senders and recipients. When a vCPU causes a PLE event, the strict boost checks this record to know which PLE-causing vCPU has issued IPIs. If there are vCPUs that have not been scheduled after an IPI has been sent to them, strict boost adds them as candidates for boosting. This implementation avoids adding to the candidates vCPUs that have already been scheduled after an IPI has been sent to them because they likely have already responded to the IPI. Since the candidates are not restricted to those that have been halted, the strict boost can boost vCPUs in user mode as well.

5 Evaluation

We evaluate debooster and strict boost in the conditions presented in Section 5.1. Our evaluation demonstrates that debooster and strict boost:

- are effective at reducing the number of PLE occurrences by up to 87.7 % (Section 5.2);
- improve the performance in our benchmarks by up to 80 % and reduce the number of PLEs as the number of VMs increases(Section 5.3);
- are effective respectively for spinlock-intensive and mmap-intensive applications (Section 5.4);
- maintain the fairness of the system (Section 5.5);
- reduce the latency of spinlock and TLB shootdown inside the guest VMs by 56.7 % on average (Section 5.6).

5.1 Experimental Settings

We evaluate debooster and strict boost in the environment shown in Table 2 with the benchmarks presented in Table 1: the `mosbench` and `parsec` benchmark suites, `pbzip2`, `dbench`, `ebizzy` and `hackbench`. All the VMs run on the same machine and execute Ubuntu 18.04 LTS with the Linux kernel 4.15. One VM executes the CPU-intensive `swaptions` benchmark and rarely causes PLE events, while the other VMs execute one of the benchmarks of Table 1. Each VM has 8 vCPUs. The total number of vCPUs is 8 times × the number of VMs; those vCPUs are multiplexed on 8 pCPUs. All our experiments are executed ten times.

[2]To the best of the authors' knowledge, only CFS implements this interface.

We use the default configuration of the PLE parameters PLE_gap (set to 128) and PLE_window (dynamically adjusted by KVM). APPLES [27] proposes auto-tuning of these parameters. If incorporated to our approaches, we believe we can achieve better performance.

We configure the kernels to use `qspinlock` instead of ticket spin locks to avoid the lock-waiter preemption problem. The PCID-based optimization [20] is enabled to reduce TLB misses. Finally, hyperthreading is turned off.

5.2 Reduction of PLE Occurrences

Figure 7 and 8 show the proportion of the number of PLE occurrences on each benchmark when respectively four and two VMs are running. In these figure, `combined` is when both debooster and strict boost are enabled.

In the four VMs case our approach reduces the number of PLE occurrences from the baseline in all benchmarks, by up to 87.6 % in `ebizzy`. Our approach also reduces the number of PLE occurrences for benchmarks other than `hackbench` in the two VMs case. The reduction of the number of PLE occurrences in the four VMs case is more significant than the two VMs case. This means our approach is more and more effective as more and more VMs are consolidated on a single physical machine. This is because, as the number of VMs running simultaneously increases, PLE events occurs more frequently and the scheduler mismatch problem and lost opportunity get more serious.

While Figure 6 shows that the cause of more than 70 % of the PLE events is "success", our mitigations can reduce PLE events by more than 50 % in most of the benchmarks in Figure 8. This is because, in the context of Figure 6, "success" simply means that a vCPU that does not generate PLE events has been boosted and scheduled. However this does not mean that the root cause of the PLE events has been solved. Moreover, this could be amplified because the ring-based candidate selection algorithm could choose the same vCPU as the target in the next iteration. Removing the root cause of a PLE by introducing our mitigations can eliminate PLE events which seem to be handled well but for which the target of the boost is actually not the root cause.

The number of PLE occurrences increases in `hackbench` in the two VMs experiment. This is due to a lot of rescheduling IPIs in `hackbench`. Unlike IPIs for TLB shootdown, the rescheduling IPIs are not synchronous; the IPI sender does not wait for acknowledgements from IPI recipients. In hackbench, 99.97 % of IPIs are sent to request rescheduling, which is the highest rate in all benchmarks. For example, in the TLB shootdown intensive workload like `vips`, the rate of reschedule IPI is only 13.85 %. As a consequence, the effectiveness of strict boost is limited by the following reason. Since vCPUs that receive rescheduling IPIs are also candidates to boost by strict boost when a PLE occurs, the possibility of boosting a vCPU which is not the cause of the PLE is increased. However, scheduler mismatch and lost opportunity are more

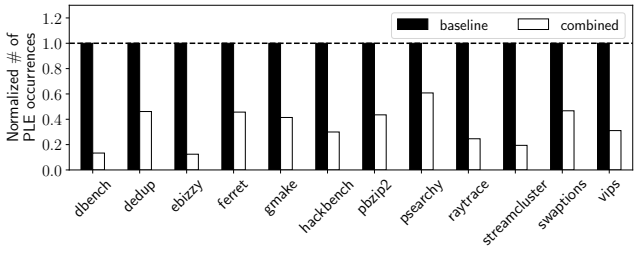

Figure 7. Number of PLE occurrences with the baseline KVM and both our debooster and strict boost mitigations active in parallel when four VMs run simultaneously.

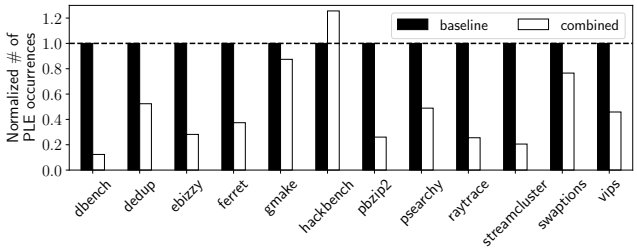

Figure 8. Normalized number of PLE occurrences with the baseline KVM and both our debooster and strict boost mitigations activated in parallel when two VMs run simultaneously.

frequent in four VMs experiment than two VMs experiment. Our approach reduces the number of PLE events by 70 % in `hackbench` in four VMs experiment.

Since the strict boost implementation is VM-agnostic, strict boost does not distinguish between the reschedule IPI and the TLB shootdown IPI. Although strict boost could eliminate unnecessary boosts by analyzing IPI information, this is not used for general guests and it is vulnerable to changes in the semantics of IPIs. Consequently, we do not implement this optimization and leave it as future work.

5.3 Performance Impact of Our Mitigations

In this section we show the impact of our mitigations on the performance of the considered benchmarks. The first column of Figure 9 shows the performance improvement with our mitigations and compares it to six kernel configurations: 1) "Baseline", i.e., Linux 5.6.0, 2) "Debooster" means debooster is solely turned on, 3) "Strict boost" means strict boost is solely turned on, 4) "Usermode" means the kernel in this configuration treats all preempted vCPUs in user-mode as candidate, 5) "Overboost" means the kernel in this configuration avoids boosting halted vCPUs if the yielding vCPU does not send an IPI to those vCPUs, 6) "Combined" where both debooster and strict boost are turned on. The configurations of both "Usermode" and "Overboost" are variant of the strict boost. All the results are normalized to the baseline performance in two VMs experiment so that higher is better.

Overall, the performance improvements introduced by our mitigations have the same characteristics in any number of VMs running simultaneously. Thus, our approach improve the performance by eliminating unnecessary excessive vCPU spinning in the guests regardless of the number of simultaneously running VMs.

Figure 9 shows the performance of dedup, vips, and ebizzy improves respectively by 42.5 %, 61.2 %, and 80.7 %, in the two VMs experiment. As seen from Figure 3 in Section 2, dedup and vips are PLE-intensive. They generated 10^4 or more PLE events per second. Therefore, our approaches can reduce PLE events more than in other benchmarks, resulting in better performance improvements.

Ebizzy is less PLE-intensive (10^3 PLE events per second) than dedup and vips, according to Figure 3. The impressive performance improvement of ebizzy comes from the fact that PLE events in ebizzy are not handled well in the baseline KVM. According to Figure 6, more than 60 % of the PLE events are not handled in the baseline KVM. Our approaches reduce PLE events effectively in ebizzy and result in a 80 % performance improvement in the two VMs experiment.

Figure 3 shows that psearchy is one of the most PLE-intensive applications. However, the performance improvement in psearchy is limited by up to 6.1 % (see Figure 9). This is because the baseline KVM handles PLE events in psearchy better than PLE events in other PLE-intensive workloads: 90 % of PLE events are handled successfully (see Figure 6).

Figure 10 shows the performance on benchmarks which are not PLE-intensive workloads. At best, the performance improves by up to 4.2 % for hackbench (four VMs experiment) and 2.8 % in pbzip2 (two VMs experiment). This is because, except for hackbench as explained in Section 5.2, the number of PLE events per second in these benchmarks is considerably lower than for the ones displayed in Figure 9. Therefore, the performance improvement is lower than for the previously mentioned benchmarks that generates a high number of PLE events per second.

5.4 Effectiveness of Debooster and Strict Boost

To confirm the effectiveness of debooster and strict boost, we report the breakdown of the performance improvement with four benchmarks: 1) dedup, 2) vips, 3) ebizzy, and 4) psearchy in the first column of Figure 9. The second column of this figure shows the breakdown of the PLE events reduction. The performance of the co-located swaptions is also shown in the third column.

In these four benchmarks, the best performance improvement can be seen when both debooster and strict boost are active in parallel. The scheduler mismatch problem hinders a vCPU from boosting even if the strict boost helps the KVM vCPU scheduler to select a good candidate. Also, the lost opportunity problem makes eliminating the root cause of a PLE event difficult even if debooster allows CFS to schedule the boosted vCPU evenly. As a consequence, debooster and the strict boost work complementary to improve the performance by reducing the number of PLE events.

In dedup and ebizzy, debooster reduces the number of PLE events significantly (> 25 %) while strict boost is less effective (< 20 %). However, in vips and psearchy, strict boost significantly reduces the number of PLE events (> 45 %) while debooster alone infrequently removes the PLE events (< 20 %). This shows that debooster addresses the scheduler mismatch problem while strict boost address both the lost opportunity and overboost problems.

According to Figure 6, the second cause of unresolved PLE events after scheduler mismatch is overboost in dedup and ebizzy, but lost opportunity in vips and psearchy. This difference comes from the benchmarks characteristics. In dedup and ebizzy, the vCPUs request to halt more frequently than in vips and psearchy. To halt the vCPUs, HLT instructions are invoked about 4,500 times per second in dedup. This is 20 times higher than the rate in vips. This high HLT rate causes frequent task migrations by CFS and thus, the scheduler mismatch problem surfaces. Also, the frequency of overboost problems is related to the HLT rate because the overboost problem occurs if the KVM vCPU scheduler tries to boost halting vCPUs when a spinlock-waiter yields. Contrarily, the lost opportunity problem comes from frequent TLB shootdown requests as well as the time spent by the benchmark in user-mode. This is because preempted vCPUs are not selected as a target to boost if they are executing the code in user-mode. The time spent in user-mode is longer with vips or psearchy compared to dedup or ebizzy. Therefore, strict boost alone is effective in vips and psearchy.

To see a more fine-grained breakdown for strict boost, we introduce two configurations "Usermode" and "Overboost" as variants of strict boost. "Usermode" kernel treats all preempted vCPUs in user-mode as a candidate. It reduces the number of PLE events significantly in TLB shootdown intensive benchmarks such as vips and psearchy (> 50 %). However, this configuration increases the number of PLE events in more frequent spinlock benchmark like ebizzy because the preempted vCPUs in user-mode are not lock-holder. The "Overboost" configuration reduces the number of PLE events by only 2 %, which is not enough to suppress the TLB shootdown latency while this configuration avoids unnecessary boosting when a PLE event occurs due to spinlock. Thus, partial mitigation for lost opportunity does not work well; strict boost needs to be used in conjunction.

5.5 System Fairness

To confirm our approaches do not have negative side-effects on the co-located VM, we measure the performance of swaptions, running inside the other VM. The third column of Figures 9 and 11 show the normalized execution time. The performance of swaptions is almost the same as the baseline KVM. Since both debooster and strict boost do not raise the priorities under CFS control, CFS can maintain the fairness it

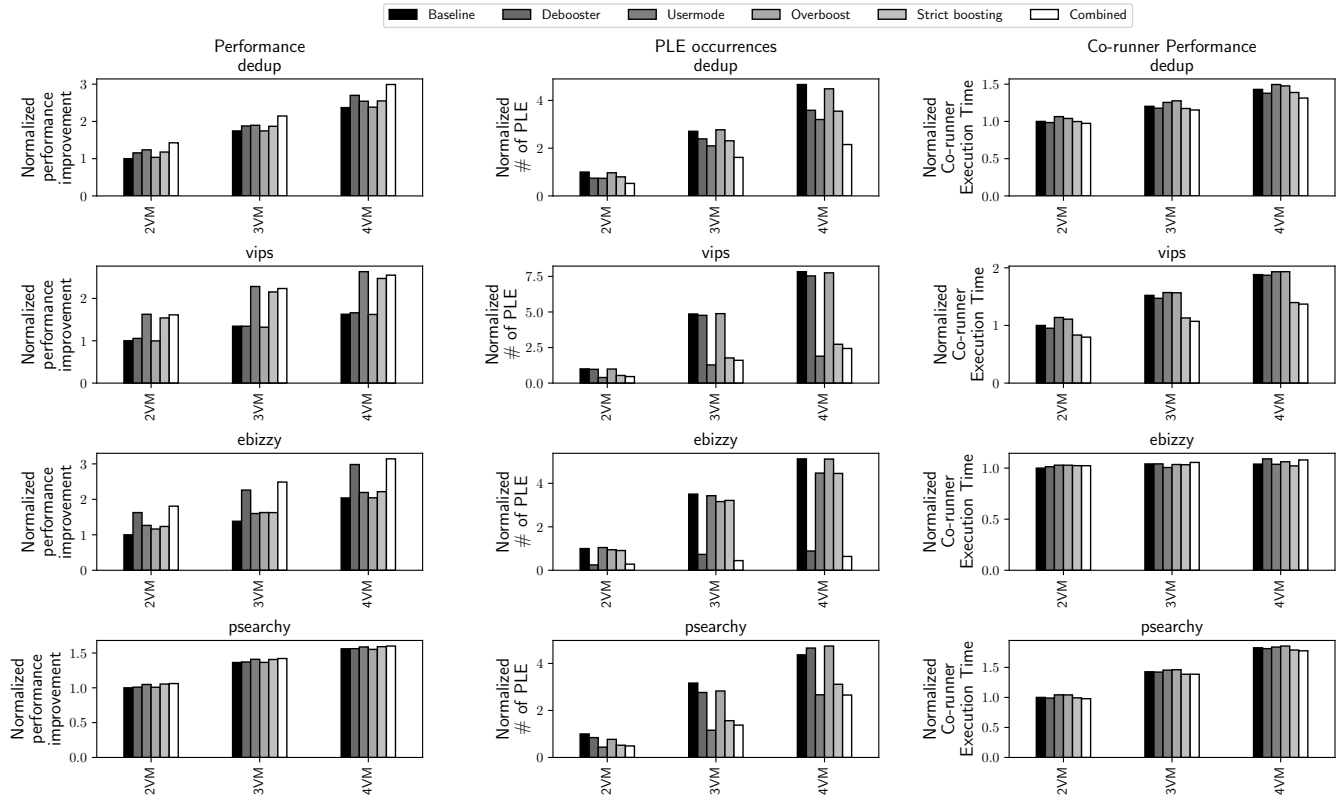

Figure 9. Performance breakdown: `dedup`, `vips`, `ebizzy`, and `psearchy`. For four kernel configurations, the first column shows the performance, the second column shows the normalized number of PLEs, and the last column shows the execution time of the co-runner application.

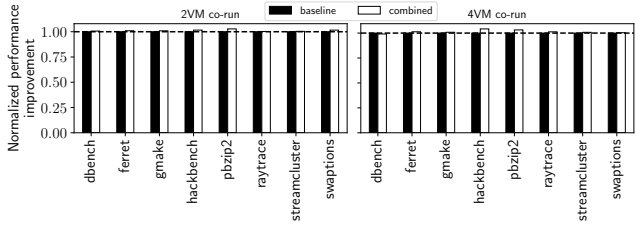

Figure 10. Performance results with two kernel configurations when two or four VMs run simultaneously

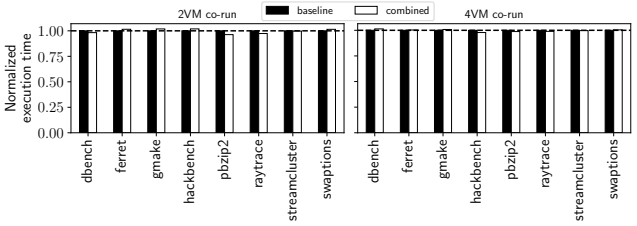

Figure 11. Co-runner performance results with two kernel configurations when two or four VMs run simultaneously

guarantees. More importantly, the performance gain in some benchmarks is not at the expense of the performance in the co-located VM. Interestingly, the performance of `swaptions` co-located with `vips` improves by 25 %. This is because our approach reduces unnecessary PLE events, by 69 %.

5.6 Time Spent in Spinlock and TLB Shootdown

We evaluate the execution time in spinlock and TLB shootdown inside the guests to confirm that reducing the number of PLE events has a positive impact of them. We monitors two functions, `native_queued_spin_lock_slowpath` and `smp_call_function_many`, because these two functions are the major producers of PLE events.

Figure 12 shows the execution time of spinlock and TLB shootdown inside the guests in four benchmarks. The normalized total execution time is shown in the first column, the average is shown in the second column, and the 95th percentile is shown in the third column.

When our two mitigations are turned on, the total, average and 95th percentile of execution time are reduced from the baseline for both spinlock and TLB shootdown in all benchmarks by 55.8 % on average. Since our approach suppresses a

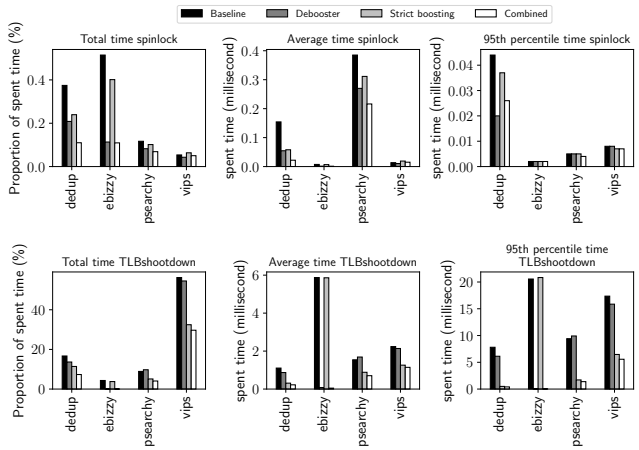

Figure 12. Execution time of spinlock and TLB shootdown inside the guests.

large number of continuous PLE events, the 95th percentile of execution time is reduced, which in turns makes the total execution time lower than the baseline.

In general, debooster improves the performance of spinlock and TLB shootdown because the scheduler mismatch problem can happen when a PLE occurs due to both of spinlock and TLB shootdown. In contrast, strict boost improves the performance of TLB shootdown only because the lost opportunity and the overboost problem surface when a PLE event happens due synchronization for TLB shootdown. However, we can see exceptions in two cases: the TLB shootdown execution time in `ebizzy` and `psearchy`. In `ebizzy`, frequent scheduler mismatch prevents the vCPU scheduler from boosting even strict boost provides precise vCPU candidates for a boost. Consequently, strict boost alone does not reduce the TLB shootdown execution time in `ebizzy`. In `psearchy`, debooster alone does not reduce the TLB shootdown execution time. This is because scheduler mismatch does not occur frequently in `psearchy`, thus we should provide precise vCPU candidates for a boost to reduce the TLB shootdown latency in `psearchy`. Therefore, solely introduced these mitigations are less effective in these two cases than in other general cases.

6 Related Work

Excessive vCPU spinning is caused by waiting for preempted vCPUs. Traditionally, detecting unusually long wait in guest VMs is required for an efficient vCPU scheduler against the lock-holder preemption (LHP) problem [7, 33]. Hardware-based excessive vCPU spinning detection [3, 26, 37] alone unfortunately does not solve this problem. Several vCPU schedulers that leverage these hardware features have been proposed [25, 27]. The latency of IPI synchronization in guest VMs is also a major cause of excessive vCPU spinning. To mitigate the problem, some researches prioritize IPI involved vCPUs for fast responses to IPI by monitoring IPI signals [5, 6, 11]. Configuring the vCPU time slice dynamically is another way to complete a critical section in guest VMs quickly [1, 2, 12, 31, 39, 40]. However, as we show, these approaches either suffer from the scheduler mismatch problem when they rely on the existing host OS scheduler to manage their resources, or require heavy modifications to their host or guest scheduler. In contrast, our work is VM agnostic yet mitigates the scheduler mismatch problem.

For hypervisor scheduler-based approaches, co-scheduling is another way to mitigate the negative effects due to the spinlock synchronization issues. It simultaneously schedules all the sibling vCPUs in the same VM [34, 38, 41]. However, co-scheduling approaches suffer from CPU fragmentation and priority inversion. Although balance scheduling [29] can alleviate such drawbacks, it prevents migrating vCPUs to keep a fair load balancing. In contrast, because our mitigations do not require modifications of the host OS scheduler core, a fair load balancing is kept.

Para-virtualization can bridge the semantic gap between the hypervisor and guest VMs to mitigate the latency that comes from both the spinlock and IPI synchronization. The para-virtualized TLB shootdown schema can improve VM performance by completing the TLB shootdown without waiting for remote vCPUs to be scheduled [4, 10, 15, 24]. To avoid the lock-waiter preemption problem, it uses a queue-based spinlock instead of the ticket-based spinlock [32] for a virtualization environment. To further improve the performance, new para-virtual lock primitives have been proposed [8, 23]. Sharing the scheduling information of host OS and guest VMs is another approach [5, 28, 30]. Contrarily to our approach, these approaches require guest OS modification to bridge the semantic gap.

7 Conclusion

Excessive vCPU spinning is a widely known problem caused by a semantic gap between a hypervisor and guest operating systems. Unfortunately, this problem is not solved yet.

We presented an in-depth analysis of excessive vCPU spinning in the VM-agnostic KVM hypervisor and analyzed the root causes of this problem. We then showed that slight modifications (41 LOC) on the KVM vCPU scheduler can gracefully solves these issues and improve the performance by up to 80 % without sacrificing scheduler fairness.

Recent research has shown promising steps towards proving some properties of schedulers [14]. We believe it is necessary to extend these proofs to verify the correctness of schedulers co-operating in a virtualized environment.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. This work was supported in part by Japan Science and Technology Agency (JST CREST JPMJCR19F3).

References

[1] Jeongseob Ahn, Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. 2018. Accelerating Critical OS Services in Virtualized Systems with Flexible Micro-sliced Cores. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. ACM, New York, NY, USA, Article 29, 14 pages. https://doi.org/10.1145/3190508.3190521

[2] Jeongseob Ahn, Chang Hyun Park, and Jaehyuk Huh. 2014. 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* (Cambridge, United Kingdom) *(MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 394–405. https://doi.org/10.1109/MICRO.2014.49

[3] AMD. 2006. *AMD64 architecture programmer's manual volume 2:System programming*.

[4] Nadav Amit and Michael Wei. 2018. The Design and Implementation of Hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 97–112. https://www.usenix.org/conference/atc18/presentation/amit

[5] Luwei Cheng, Jia Rao, and Francis C. M. Lau. 2016. vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 2, 14 pages. https://doi.org/10.1145/2901318.2901321

[6] Xiaoning Ding, Phillip B. Gibbons, Michael A. Kozuch, and Jianchen Shan. 2014. Gleaner: Mitigating the Blocked-waiter Wakeup Problem for Virtualized Multicore Applications. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 73–84. http://dl.acm.org/citation.cfm?id=2643634.2643643

[7] Thomas Friebel and Sebastian Biemueller. 2008. How to Deal with Lock Holder Preemption *(Xen Summit)*.

[8] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2015. Scalability in the Clouds! A Myth or Reality?. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (Tokyo, Japan) *(APSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2797022.2797037

[9] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2018. Scaling Guest OS Critical Sections with eCS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 159–172. https://www.usenix.org/conference/atc18/presentation/kashyap

[10] O. Kilic, S. Doddamani, A. Bhat, H. Bagdi, and K. Gopalan. 2018. Overcoming Virtualization Overheads for Large-vCPU Virtual Machines. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 369–380.

[11] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. 2013. Demand-based Coordinated Scheduling for SMP VMs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 369–380. https://doi.org/10.1145/2451116.2451156

[12] T. Kim, C. H. Park, J. Huh, and J.Ahn. 2020. Reconciling Time Slice Conflicts of Virtual Machines With Dual Time Slice for Clouds. *IEEE Transactions on Parallel and Distributed Systems* 31, 10 (2020), 2453–2465.

[13] KVM. 2016. *KVM*. http://www.linux-kvm.org/page/Main_Page

[14] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. 2020. Provable multicore schedulers with Ipanema: application to work conservation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[15] Wanpeng Li. 2017. *KVM: X86: Add Paravirt TLB Shootdown*. https://lwn.net/Articles/740363/

[16] Wanpeng Li. 2018. *Towards a more Scalable KVM Hypervisor*. https://events19.linuxfoundation.org/wp-content/uploads/2017/12/Torwards-a-More-Scalable-KVM-Hypervisor-Wanpeng-Li-Tencent-Cloud.pdf

[17] Wanpeng Li. 2019. *KVM: Boost vCPUs that are delivering interrupts*. https://patchwork.kernel.org/patch/11048999/

[18] Wanpeng Li. 2020. *KVM Latency Performance Tuning*. https://static.sched.com/hosted_files/kvmforum2020/6e/KVM%20Latency%20and%20Scalability%20Performance%20Tuning.pdf

[19] Mike Longpeng. 2017. *KVM: add spinlock-exiting optimize framework*. https://lore.kernel.org/patchwork/patch/818160/

[20] Andy Lutomirski. 2017. *x86/mm: Implement PCID based optimization: try to preserve old TLB entries using PCID*. https://lore.kernel.org/patchwork/patch/813646/

[21] Microsoft. 2018. *Managing Hyper-V hypervisor scheduler types*. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-scheduler-types

[22] Oracle 2021. *Oracle VM VirtualBox*. https://www.virtualbox.org/

[23] Jiannan Ouyang and John R. Lange. 2013. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Houston, Texas, USA) *(VEE '13)*. ACM, New York, NY, USA, 191–200. https://doi.org/10.1145/2451512.2451549

[24] Jiannan Ouyang, John R. Lange, and Haoqiang Zheng. 2016. Shoot4U: Using VMM Assists to Optimize TLB Operations on Preempted VCPUs. In *Proceedings of The12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Atlanta, Georgia, USA) *(VEE '16)*. Association for Computing Machinery, New York, NY, USA, 17–23. https://doi.org/10.1145/2892242.2892245

[25] K. Raghavendra. 2013. Virtual CPU scheduling techniques for kernel based virtual machine (KVM). In *2013 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. 1–6. https://doi.org/10.1109/CCEM.2013.6684443

[26] M. Righini. 2010. *Enabling Intel virtualization technology features and benefits*.

[27] J. Shan, X. Ding, and N. Gehani. 2017. APPLES: Efficiently Handling Spin-lock Synchronization on Virtualized Platforms. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (July 2017), 1811–1824. https://doi.org/10.1109/TPDS.2016.2625249

[28] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. 2013. Schedule Processes, Not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems* (Singapore, Singapore) *(APSys '13)*. ACM, New York, NY, USA, Article 1, 7 pages. https://doi.org/10.1145/2500727.2500736

[29] Orathai Sukwong and Hyong S. Kim. 2011. Is Co-scheduling Too Expensive for SMP VMs?. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. ACM, New York, NY, USA, 257–272. https://doi.org/10.1145/1966445.1966469

[30] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. 2017. The Lock Holder and the Lock Waiter Pre-emption Problems: Nip Them in the Bud Using Informed Spinlocks (I-Spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 286–297. https://doi.org/10.1145/3064176.3064180

[31] Boris Teabe, Alain Tchana, and Daniel Hagimont. 2016. Application-specific Quantum for Multi-core Platform Scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. ACM, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/2901318.2901340

[32] Waiman TLong. 2014. *qspinlock: a 4-byte queue spinlock with PV support*. https://lwn.net/Articles/597672/

[33] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. 2004. Towards Scalable Multiprocessor Virtual Machines. In

Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3 (San Jose, California) *(VM'04)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1267242.1267246

[34] VMware. 2010. *VMware vSphere 4: The CPU Scheduler in VMware ESX 4.1*. Technical Report.

[35] VMware. 2019. *Configuring Process Priorities on Windows Hosts*. https://docs.vmware.com/en/VMware-Workstation-Pro/15.0/com.vmware.ws.using.doc/GUID-01BFCBEA-A646-4372-B5B0-6A0C6290672B.html

[36] VMware 2021. *Workstation Pro - VMWare Products*. https://www.vmware.com/products/workstation-pro.html

[37] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. 2006. Hardware Support for Spin Management in Overcommitted Virtual Machines. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques* (Seattle, Washington, USA) *(PACT '06)*. ACM, New York, NY, USA, 124–133. https://doi.org/10.1145/1152154.1152176

[38] Chuliang Weng, Qian Liu, Lei Yu, and Minglu Li. 2011. Dynamic Adaptive Scheduling for Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing* (San Jose, California, USA) *(HPDC '11)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/1996130.1996163

[39] S. Wu, Z. Xie, H. Chen, S. Di, X. Zhao, and H. Jin. 2016. Dynamic Acceleration of Parallel Applications in Cloud Platforms by Adaptive Time-Slice Control. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 343–352. https://doi.org/10.1109/IPDPS.2016.77

[40] Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. 2012. vSlicer: Latency-Aware Virtual Machine Scheduling via Differentiated-Frequency CPU Slicing. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing* (Delft, The Netherlands) *(HPDC '12)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2287076.2287080

[41] L. Zhang, Y. Chen, Y. Dong, and C. Liu. 2012. Lock-Visor: An Efficient Transitory Co-scheduling for MP Guest. In *2012 41st International Conference on Parallel Processing*. 88–97. https://doi.org/10.1109/ICPP.2012.36

[42] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-Density Multi-Tenant Bare-Metal Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 483–495. https://doi.org/10.1145/3373376.3378507