Scaling Guest OS Critical Sections with *e*CS

Sanidhya Kashyap, *Georgia Institute of Technology;* Changwoo Min, *Virginia Tech;*
Taesoo Kim, *Georgia Institute of Technology*

https://www.usenix.org/conference/atc18/presentation/kashyap

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Scaling Guest OS Critical Sections with *e*CS

Sanidhya Kashyap Changwoo Min[†] Taesoo Kim
Georgia Institute of Technology *Virginia Tech*[†]

Abstract

Multi-core virtual machines (VMs) are now a norm in data center environments. However, one of the well-known problems that VMs suffer from is the vCPU scheduling problem that causes poor scalability behaviors. More specifically, the symptoms of this problem appear as preemption problems in both under- and over-committed scenarios. Although prior research efforts attempted to alleviate these symptoms separately, they fail to address the common root cause of these problems: the missing semantic gap that occurs when a guest OS is preempted while executing its own critical section, thereby leading to degradation of application scalability.

In this work, we strive to address all preemption problems together by bridging the semantic gap between guest OSes and the hypervisor: the hypervisor now knows whether guest OSes are running in critical sections and a guest OS has hypervisor's scheduling context. We annotate all critical sections by using the *lightweight* para-virtualized APIs, so we called *enlightened critical sections (eCS)*, that provide scheduling hints to both the hypervisor and VMs. The hypervisor uses the hint to reschedule a vCPU to fundamentally overcome the *double scheduling* problem for these annotated critical sections and VMs use the hypervisor provided hints to further mitigate the blocked-waiter wake-up problem. Our evaluation results show that *e*CS guarantees the forward progress of a guest OS by 1) decreasing preemption counts by 85–100% while 2) improving the throughput of applications up to 2.5× in an over-committed scenario and 1.6× in an under-committed scenario for various real-world workloads on an 80-core machine.

1 Introduction

Virtualization is now the backbone of every cloud-based organization to run and scale applications horizontally on demand. Recently, this scalability trend is also extending towards vertical scaling [2, 11], *i.e.*, a virtual machine (VM) has up to 128 virtual CPUs (vCPUs) and 3.8 TB of memory to run large in-memory databases [24, 35] and data processing engines [47]. At the same time, cloud providers strive to oversubscribe their resources to improve hardware utilization and reduce energy consumption, without imposing any permissible overhead on the application [38, 46]. However, over subscription requires multiplexing of physical CPUs among VMs to equally distribute physical CPU cycles. Thus, the multiplexing of these VMs introduces the *double scheduling*

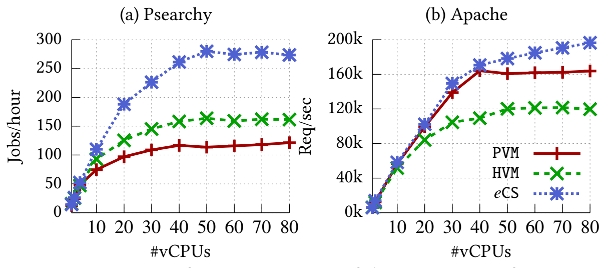

Figure 1: Impact of exposing some of the semantic information from the VM to the hypervisor and vice-versa, which leads to better scalability of Psearchy and Apache web server benchmark, in a scenario in which two VMs are running with the same benchmark. Here, PVM and HVM denote with and without para-virtualization support, while *e*CS represents our approach. Psearchy mostly suffers from LHP and BWW. Similarly, Apache suffers from LHP and ICP.

problem [40]: 1) the guest OS schedules processes on vCPUs and 2) the hypervisor schedules vCPUs on physical CPUs. Some of the prior works address this problem by adopting co-scheduling approaches [17, 41, 45], which can suffer from priority inversion, CPU fragmentation, and may mitigate the double scheduling symptoms [40]. Such symptoms, that have mostly been addressed individually, are lock-holder preemption (LHP) [8, 15, 42, 44], lock-waiter preemption (LWP) [44], and blocked-waiter wakeup (BWW) [5, 39], problems.

The root cause of this double scheduling phenomenon is a semantic gap between a hypervisor and guest OSes, in which the hypervisor is agnostic of not only the scheduling of VMs but also guest OS-specific critical code that deter the scalability of applications. Furthermore, LHP/LWP are not only limited to spinlocks [15, 19, 42], but are also possible in blocking primitives such as mutex and rwsem as well as readers of the rwsem. Moreover, because of their non work-conserving nature, these blocking primitives inherently suffer from the BWW problem (refer Psearchy in Figure 1 (a)). Besides these, none of the prior works have identified the preemption of an interrupt context that happens in interrupt-intensive applications such as Apache web-server (Figure 1 (b)). We define this problem as *interrupt context preemption* (ICP).

Our key observation is that these symptoms occur because 1) the hypervisor is scheduling out a vCPU at a time when the vCPU is executing a critical code, and 2) a vCPU, waiting to acquire a lock, is either uncooperative or sleeping [16], leading to LWP and BWW issues. Thus, we propose an alternative perspective, *i.e.*, instead of devising a solution for each symptom, we use four key ideas that allows a VM to hint the hypervisor for making an effective scheduling decision to allow its forward progress. First, we consider all of the locks and interrupt contexts as critical components. Second, we devise a set of para-virtualized APIs that annotate these critical components as *enlightened critical sections* (*e*CS). These APIs are lightweight in nature and notify a hypervisor from the VM and vice-versa with memory operations via shared memory, while avoiding the overhead of hypercall and interrupt injection. Third, the hypervisor now can figure out whether a vCPU is executing an *e*CS and can reschedule it. We empirically found that an extra schedule (one millisecond [27]) is sufficient as it decreases preemptions by 85–100%; and these critical sections are shorter (in μs [42]) than one schedule. However, by rescheduling a vCPU, we introduce unfairness in the system. We tackle this issue with the OS's fair scheduling policy [27], which compensates for that additional schedule by allowing other tasks to run for extra time, thereby maintaining the eventual fairness in the system. Lastly, we leverage our APIs to design a virtualized schedule-aware spinning strategy (*e*SchdSpin) that enables lock waiters to be work conserving as well as cooperative inside a VM. That is, a vCPU now cooperatively spins for the lock, if a physical CPU is under-committed, else it yields the vCPU.

Thus, our approach improves the scalability of real-world applications by 1.2–1.6× in an under-committed case. Moreover, our *e*CS annotation, combined with *e*SchdSpin, avoids preemption by 85–100% while improving the scalability of applications by 1.4–2.5× in an over-committed scenario on an 80-core machine.

In summary, we make the following contributions:

- We identify similarities among various subproblems that stem from the double scheduling phenomenon. Moreover, we identify three new problems: 1) LHP in blocking locks, 2) readers preemption (RP) in read-write locks and semaphores, and 3) vCPU preemption while processing an interrupt context (ICP).
- We address these subproblems with *e*CS, which we annotate with six new APIs that bridge the semantic gap between a hypervisor and a VM, and even among vCPUs inside a VM.
- Our annotation approach, along with *e*SchdSpin, improves the scalability of applications in both under- and over-committed scenarios up to 2.5× with only 0–15% preemptions, while maintaining eventual fairness with merely one extra schedule.

2 Background and Motivation

We first describe the problem of double scheduling and highlight its implications. Later, we summarize the prior attempts to solve this problem, and then motivate our approach.

2.1 Symptoms of Double Scheduling

In a virtualized environment, a hypervisor multiplexes the hardware resources for a VM, such as assigning vCPUs to physical CPUs (pCPUs). In particular, it runs a vCPU to execute by its fair share [27], which is a general policy of commodity OSes such as Linux, and preempts it because of either vCPUs of other VM or of the intermittent processes of the OS and bookkeeping tasks of the hypervisor such as I/O threads. Hence, there is a possibility that the hypervisor can preempt a vCPU while executing some critical task inside a VM that leads to an application performance anomaly, which we enumerate below:

Lock holder preemption (LHP) problem occurs when a vCPU holding a lock gets preempted and all waiters waste CPU cycles for the lock. Most of the prior works [8, 14, 42, 44] have focused on non-blocking primitives such as spinlocks.[1] On the other hand, LHP also occurs in blocking primitives such as `mutex` [28] and `rwsem` [26, 31], which the prior works have not identified. However, LHP accounts up to 90% preemptions for blocking primitives in some of the memory intensive applications that have short critical sections.

Lock waiter preemption (LWP) problem stems when the very next waiter is preempted just before acquiring the lock, which occurs due to the strict FIFO ordering of spinlocks [14, 42]. Fortunately, this problem has been mostly mitigated in existing spinlock design [19, 20], as the current implementation allows waiters to steal the lock before joining the waiter queue. We do not see such a problem in blocking primitives because the current implementation is based on the test-and-set (TAS) lock—an unfair lock, which inherently mitigates LWP.

Blocked-waiter wakeup (BWW) problem occurs mostly for blocking primitives in which the latency to wake up a waiter to pass the lock is quite high. This issue severely degrades the throughput of applications running on a high core count [16], even in a native environment. Moreover, it is evident in both under- and over-committed VM scenarios. For example, the BWW problem degrades the application scalability up to 1.6× (refer Figure 6).

Readers preemption (RP) problem is a new class of problem that occurs when a vCPU holding a read lock among multiple readers gets preempted. This problem impedes the forward progress of a VM and also increases the latency of the write lock. For instance, various memory-intensive workloads have sub-optimal throughput as RP accounts to at most 20% of preemptions. We observe this issue in various read-dominated memory-intensive workloads in which the readers are scheduled out.

RCU reader preemption (RRP) problem is a type of RP

[1]Non-blocking locks, both holders and waiters, do not schedule out. However, the para-virtualized interface converts spinlocks to blocking locks (only waiters) with hypercalls [6, 20] to overcome LHP/LWP issues.

problem that occurs when an RCU reader is preempted, while holding the RCU read lock [33]. Because of RRP, the guest OS suffers from an increased quiescence period. This issue can increase the memory footprint of the application, and is responsible for 5% of preemptions.

Interrupt context preemption (ICP) problem happens when a vCPU that is executing an interrupt context gets preempted. In particular, this problem is different from prior works that focus on interrupt delivery [12, 43] rather than interrupt handling. This issue occurs in cases such as TLB shootdowns, function call interrupts, rescheduling interrupts, IRQ work interrupts, etc. in every commodity OS. For example, we found that Apache web server, an interrupt-intensive workload, suffers from the ICP problem as it accounts to almost 18% of preemptions for evaluated workloads (refer Figure 3).

2.2 Prior Approaches

Some of the prior studies mitigate LHP and LWP problems by relaxed co-scheduling [45], balancing vCPUs to physical CPUs [41] with IPIs as a heuristic [17], or using hardware features [34]. Meanwhile, others designed a para-virtualized interface [8, 14, 42, 44] to only tackle the LHP and LWP problem for spinlocks. Besides these, one radical design focused on scheduling VM's processes than vCPUs by hot plugging vCPUs on the basis of load on the VM [3, 40]. Unfortunately, all of these prior works address the double scheduling problem either *partially* that misses other preemption problems, or take a radical path that is not only difficult to adopt in practice but can have significant overhead, in terms of scaling for machines with almost 100 physical cores. Because their approach involves 1) the detection of response to the double scheduling in the form of hypercalls and interrupt injection [3], and 2) explicit task migration from idle vCPUs to active vCPUs. On the contrary, our approach does simple memory operations and exploits the vCPU scheduling boundary to notify the hypervisor for scheduling decisions without any explicit task and vCPU migration: a lightweight approach even at high core count.

2.3 The Case for *An Extra Schedule*

As mentioned before, OS critical sections are the ones that define the forward progress of an application for which the OS is responsible. For instance, let us take an example of two threads competing to acquire a lock to update contents of a file. If the lock holder, which is updating the file, is preempted, the other waiter will waste CPU cycles. There are several critical operations that affect the application scalability [16, 25], and OS performs such operations either by acquiring a lock or executing an interrupt context (I/O processing, TLB shootdowns, etc.). In particular, a delay in processing of these critical sections can result in a severe performance anomaly such as a convoy effect [14, 16], or decreased network through-

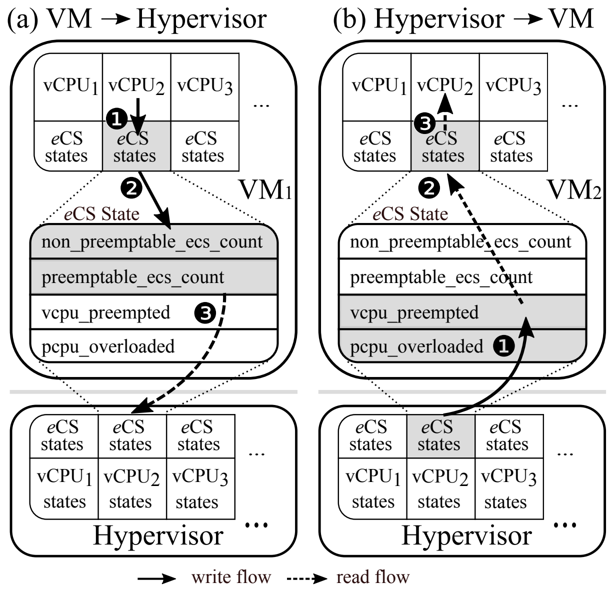

Figure 2: Overview of the information flow between a VM and a hypervisor. Each vCPU has a per-CPU state that is shared with the hypervisor, denoted as *e*CS state. Figure (a) shows how the $vCPU_2$ relays information about an *e*CS to the hypervisor. On entering a critical section or an interrupt context (❶), $vCPU_2$ updates the non_preemptable_ecs_count (❷). After a while, before scheduling out $vCPU_2$, the hypervisor reads its *e*CS state (❸), and allows it run for one more schedule to mitigate any of the double scheduling problems. Figure (b) shows how the hypervisor shares the information whether a vCPU is preempted or a physical CPU is overloaded, at the schedule boundary. For instance, the hypervisor marks vcpu_preempted, while scheduling out a vCPU; or updates pcpu_overloaded flag to one if the number of active tasks on that physical CPU is more than one. Both try to further mitigate LWP and BWW problems.

put for applications such as web servers (refer Figure 1). Hence, unlike prior approaches, we propose a simple and an intuitive approach, *i.e.*, now a VM hints the hypervisor about a critical section that enables the hypervisor to let a vCPU execute for a pre-defined time slot (schedule). This extra schedule is sufficient to complete a critical section because 1) most critical sections are very fine-grained, and have a time granularity of several microseconds [42], while 2) the granularity of a single schedule is in the order of milliseconds, which is sufficient enough to complete a critical section. For instance, an extra schedule decreases the preemption count by 85–100% (Figure 3). This approach is not only practical but also critical to apply on machines with large core count. However, the extra schedule introduces *unfairness* in the system, which we address by designing a simple, *zero-overhead schedule penalization algorithm* that tries to maintain the *eventual fairness* in the system by leveraging the CFS [27] that tries to maintain fairness in the system.

Hint	Lightweight Para-virtualized API	Description
VM → Hypervisor	`void activate_non_preemptable_ecs(cpu_id)`	Increase the *e*CS count for a vCPU with cpu_id by 1 for a non-preemptable task
	`void deactivate_non_preemptable_ecs(cpu_id)`	Decrease the *e*CS count for a vCPU with cpu_id by 1 for a non-preemptable task
	`void activate_preemptable_ecs(cpu_id)`	Increase the *e*CS count for a vCPU with cpu_id by 1 for a preemptable task
	`void deactivate_preemptable_ecs(cpu_id)`	Decrease the *e*CS count for a vCPU with cpu_id by 1 for a preemptable task
Hypervisor → VM	`bool is_vcpu_preempted(cpu_id)`†	Return whether a vCPU with cpu_id is preempted by the hypervisor
	`bool is_pcpu_overcommitted(cpu_id)`	Return whether a physical CPU, running a vCPU with cpu_id, is over-committed

Table 1: Set of para-virtualized APIs exposed by the hypervisor to a VM for providing hints to the hypervisor to mitigate double scheduling. These APIs provide hints to the hypervisor and VM via shared memory. A vCPU relies on the first four APIs to ask for an extra schedule to overcome LHP, LWP, RP, RRP, and ICP. Meanwhile, a vCPU gets hints from the hypervisor by using the last two APIs to mitigate LWP and BWW problems. The cpu_id is the core id that is used by tasks running inside a guest OS.
†Currently, `is_vcpu_preempted()` is already exposed to the VM in Linux.

3 Design

A hypervisor can mitigate various preemption problems, if it is aware of a vCPU executing a critical section. We denote such a hypervisor-aware critical section as an *enlightened critical section* (*e*CS), that can be executed for one more schedule. *e*CS is applicable to all synchronization primitives and mechanisms such as RCU and interrupt contexts. We now present our lightweight APIs that act as a cross-layer interface for annotating an *e*CS and later focus on our notion of an extra schedule and our approach to maintain eventual fairness in the system.

3.1 Lightweight Para-virtualized APIs

We propose a set of six *lightweight para-virtualized APIs* to bridge the semantic gap that both VM and hypervisor use for conveying information between them. These APIs rely on four variables (refer Figure 2) that are *local* to each vCPU. They are exposed via shared memory between the hypervisor and a VM and the notification happens via simple read and write memory operations. A simple memory read is sufficient for the hypervisor to decide on scheduling because 1) it tries to execute each vCPU on a separate pCPU, 2) and it requires knowing about an *e*CS only at the schedule boundary, thereby removing the cost of polling and other synchronous notifications [3]. To consider an OS critical section as an *e*CS, we mark the start and unmark the end of a critical section, which lets the hypervisor know about an *e*CS. However, a process in an OS can be of two types. First is the non-preemptable process that can never be scheduled out. Such a process is either an interrupt or a kernel thread running after acquiring a spinlock. Another one is the preemptable task such as a user process or a process with blocking lock. Hence, we introduce four APIs (VM → Hypervisor) to separately handle these two types of tasks. The last two APIs (Hypervisor → VM) provide the hypervisor context to the VM, which a lock waiter can use to mitigate the LWP problem or yield the vCPU to other hypervisor tasks or vCPUs in an over-committed scenario. Figure 2 illustrates those four states:

- **`non_preemptable_ecs_count`** maintains the count of active non-preemptable *e*CSs, such as non-blocking locks, RCU reader, and interrupt contexts. It is similar to the preemption count of the OS.
- **`preemptable_ecs_count`** is similar to the preemption count variable of the OS, but it only maintains the count of active preemptable *e*CSs, such as blocking primitives, namely, `mutex` and `rwsem`.
- **`vcpu_preempted`** denotes whether a vCPU is running. It is useful for handling the BWW problem in both under- and over-committed scenarios.
- **`pcpu_overloaded`** denotes whether a physical CPU, executing that particular vCPU, is over-committed. Lock waiters can use this information to address the BWW problem in an over-committed scenario.

Figure 2 presents two scenarios in which the schedule context information is shared between a vCPU and the hypervisor. Figure 2 (a) shows how a vCPU, *i.e.*, entering an *e*CS, shares information with the hypervisor. During entry (❶), $vCPU_2$ first updates its corresponding state (`non_preemptable_ecs_count` or `preemptable_ecs_count`) (❷) and continues to execute its critical section. Meanwhile, the hypervisor, before scheduling out $vCPU_2$, checks $vCPU_2$'s *e*CS states (❸) and allows it to run for extra time if certain criteria are fulfilled (§3.2); otherwise, it schedules out $vCPU_2$ with other waiting tasks. When $vCPU_2$ exits the *e*CS, it decreases the *e*CS state count, denoting the end of critical section. Figure 2 (b) illustrates another scenario that addresses the BWW problem. in which the hypervisor updates the *e*CS states: `pcpu_overloaded` and `vcpu_preempted` while scheduling in and out $vCPU_2$, respectively, at each schedule boundary (❶). We devise a simple approach—virtualized scheduling-aware spinning (*e*SchdSpin)—that enables efficient scheduling aware waiting for both blocking and non-blocking locks (§4). That is, $vCPU_2$ reads both states (❷) and decides whether to keep spinning until the lock is acquired if the pCPU is not overloaded (❸), else it yields, which allows the other vCPU (in VM_2) or a hypervisor's task to progress forward by doing some useful task, thereby mitigating the double scheduling problems.

3.2 Eventual Fairness with Selective Scheduling

As mentioned before, the hypervisor relies on its scheduler to figure out whether a vCPU is executing an *e*CS. That is, when a vCPU with a marked *e*CS is about to be scheduled out, the hypervisor scheduler checks the value of *e*CS count variables (Figure 2). If any of these values are greater than zero, the hypervisor lets the vCPU run for an extra schedule. However, vCPU rescheduling introduces two problems: 1) How does the hypervisor handles a task with *e*CS, which the guest OS can preempt or schedule out? 2) How does it ensure the system fairness?

We handle an *e*CS preemptable task with `preemptable_ecs_count` counter APIs, which differentiate between a preemptable task and a non-preemptable task. We do so because the guest OS can schedule out a preemptable task. In this case, the hypervisor should avoid rescheduling that vCPU because 1) it will result in false rescheduling, and 2) it can hamper the VM performance. We address this issue inside the guest OS, *i.e.*, before scheduling out an *e*CS-marked task inside a guest OS, we save the value of `preemptable_ecs_count` to a task-specific structure and reset the counter to zero. Later, when the task is rescheduled again by the guest OS, we restore the `preemptable_ecs_count` with the saved value from the task-specific structure, thereby mitigating the false scheduling.

With vCPU rescheduling, we introduce unfairness at two levels: 1) An *e*CS marked vCPU will always ask for rescheduling on every schedule boundary.[2] 2) By rescheduling a vCPU, the hypervisor is unfair to other tasks in the system. We resolve the first issue by allowing the hypervisor to reschedule an *e*CS-marked vCPU only once during that schedule boundary as rescheduling extends the boundary. At the end of schedule boundary, the hypervisor schedules other tasks to avoid the starving other tasks or VMs and addresses indefinite rescheduling. In addition, the hypervisor also keeps track of this extra reschedule information and runs other vCPUs for longer duration and inherently balances the running time, an equivalent to vCPU penalization. Thus, our approach selectively reschedules and penalizes a vCPU rather than balancing the extra reschedule information across all cores, which will result in an unnecessary overhead of synchronizing all runtime information of rescheduling. We call our approach as the *local CPU penalization* approach, as we only penalize a vCPU that executed an *e*CS, thereby ensuring *eventual fairness* in the system. Moreover, our local vCPU scheduling is a form of selective-relaxed co-scheduling of vCPUs depending on what kind of tasks are being executed, while without maintaining any synchronization among vCPUs, unlike prior approaches [41, 45].

4 Use Case

The double scheduling phenomenon introduces the semantic gap in three places: 1) from a vCPU to a physical CPU that results in LHP, RP, and ICP problems; 2) from a pCPU to a vCPU; and 3) from one vCPU to another in a VM, both suffer from LWP and BWW problems. Table 2 shows how to use our APIs to address these problems.

[2]Such a VM can be either an I/O or an interrupt-intensive VM that spends most of its time in the kernel, or even a compromised VM.

API	LHP	RP	RRP	ICP	LWP	BWW
`activate_non_preemptable_vcs()`	✓	✓	✓	✓	-	-
`deactivate_non_preemptable_vcs()`	✓	✓	✓	✓	-	-
`activate_preemptable_vcs()`	✓	✓	-	-	-	-
`deactivate_preemptable_vcs()`	✓	✓	-	-	-	-
`is_vcpu_preempted()`	-	-	-	-	✓	✓
`is_pcpu_overcommitted()`	-	-	-	-	-	✓

Table 2: Applicability of our six lightweight para-virtualized APIs that strive to address the symptoms of double scheduling.

Component	Lines of code
*e*CS annotation	60
*e*CS infrastructure	800
Scheduler extension	150
Total	1,010

Table 3: *e*CS requires small modifications to the existing Linux kernel, and the annotation effort is also minimal: 60 LoC changes to support the 10 million LoC Linux kernel that has around 12,000 of lock instances with 85,000 lock invocations.

LHP, RP, RRP, and ICP problem. To circumvent these problems, we rely on the VM → hypervisor notification because a vCPU running any spinlocks, read-write locks, `mutex`, `rwsem`, or an interrupt context is already inside the critical section. Thus, we call `activate_*()` and `deactivate_*()` APIs for annotating critical sections. For example, the first two APIs are applicable to spinlocks, read-write locks, RCU, and interrupts, and the next two are for `mutex` and `rwsem`. (refer Table 2).

LWP and BWW problem. The LWP problem occurs in the case of FIFO-based locks such as MCS and Ticket locks [23]. However, unfair locks, such as `qspinlock` [6], `mutex` [29], and `rwsem` [37], do not suffer from this problem, and are currently used in Linux. The reason is that they allow other waiters to steal the lock, while suffering from the issue of starvation. On the other hand, all of these locks suffer from the BWW problem because the cost to wake up a sleeping in a virtualized environment varies from 4,000–10,000 cycles. as a wake-up call results in a VMexit, which adds an extra overhead to notify a vCPU to wake up a process. This problem is severe for blocking primitives because they are non-work conserving in nature [16], *i.e.*, the waiters schedule out themselves, even if a single task is present in the run queue of the guest OS. We partially mitigate this issue by allowing the waiters to spin rather than sleep if a single task is present in the run queue of the guest scheduler (SchdSpin). However, this approach is non-cooperative when multiple VMs are running. Thus, to avoid unnecessary spinning of waiters, we rely on our `is_pcpu_overcommitted()` API that notifies a waiter to only spin if the pCPU is not over-committed. We call this approach the virtualized scheduling-aware spinning approach (*e*SchdSpin).

5 Implementation

We realized the idea of *e*CS by implementing it on the Linux kernel version 4.13. Besides annotating various locks and interrupt contexts with *e*CS, we specifically modified the scheduler and the para-virtual interface of the KVM hypervisor. Our changes are portable enough to apply on the Xen hypervisor too. The whole modification consists of 1,010 lines of code (see Table 3).

Lightweight para-virtualized APIs. We share the information between the hypervisor and a VM with a shared memory between them, which is similar to the `kvm_steal_time` [4] implementation. For instance, each VM maintains a per-core *e*CS states, and the hypervisor maintains per-vCPU *e*CS states for each VM.

Scheduler extension. We extend a scheduler-to-task notification mechanism, `preempt_notifier` [18], for identifying an *e*CS-marked vCPU at the schedule boundary. Our extension allows the scheduler to know about the task scheduling requirement and decide scheduling strategy at the schedule boundary. For example, in our case, the extension reads the `non_preemptable_ecs_count` and `preemptable_ecs_count` to decide the scheduling strategy for the vCPU. Besides this, we rely on the notifier's in and out APIs to set the value of `vcpu_preempted` and `pcpu_overloaded` variables.

We implemented our vCPU rescheduling decision in the `schedule_tick` function [36]. The `schedule_tick` function performs two tasks: 1) It does the bookkeeping of the task runtime, which is used for ensuring the fairness in the system. 2) It also is responsible for setting the rescheduling flag (`TIF_NEED_RESCHED`) if there is more than one task on that run queue, which is used by the scheduler to schedule out the task if the reschedule flag is set. We implemented the rescheduling strategy by bypassing the setting up of the reschedule flag in case the `preempt_notifier` check function returned true, meanwhile updating the runtime statistics of the vCPU.

Annotating locks for *e*CS. We mark *e*CS by using the non-preemptable APIs for non-blocking primitives, preemptable ones for `mutex` and `rwsem`. Our annotation comprises only 60 LoC that covers around 12,000 lock instances with 85,000 lock API calls in the Linux kernel that has 10 million LoC for the kernel version 4.13.

6 Evaluation

We evaluate our approaches by answering the following questions:

- What is the overhead of an *e*CS annotation and the scheduler overhead to read the values? (§6.1)
- Does *e*CS helps in an over-committed case? (§6.2)
- How does *e*CS impact the scalability of a VM? (§6.3)
- How do our APIs address the BWW problem? (§6.4)
- Does our schedule penalization approach maintain the eventual fairness of the system? (§6.5)

Experimental setup. We extended `VBench` [13] for our evaluation. We chose four benchmarks: Apache web server [7], Metis [21], Psearchy from Mosbench, and Pbzip2 [9]. The Apache web server serves a 300 bytes static page for each request that is generated by WRK [10]. Both of them are running inside the VM to remove the network wire overhead and only stress the VM's kernel components. We choose Apache to stress the interrupt handler to emphasize the importance of *e*CS for an interrupt context. Metis is a map-reduce library for a single multi-core server that mostly stresses the memory allocator (spinlock) and the page-fault handler (`rwsem`) of the OS. Similar to Metis, Psearchy is an in-memory parallel search and indexer that stresses the writer side of the `rwsem` design. In addition, we also choose Pbzip2—a parallel compression and decompression program—because we wanted to use a minimally kernel-intensive application. Moreover, none of these workloads suffer from performance degradation from any known user space bottleneck in a non-virtualized environment. We use memory-based file system, `tmpfs`, to isolate the effect of I/O. We further pin the cores to circumvent vCPU migration at the hypervisor level to remove the jitter from our evaluation.

We evaluate our *e*CS approach against the following configurations: 1) PVM is a para-virtualized VM that includes unfair `qspinlock` implementation, which mitigates LWP and BWW issues, and it is the default configuration since Linux v4.5. 2) HVM is the one without para-virtualization support and also includes unfair `qspinlock` implementation. Both PVM and HVM are not *e*CS annotated. Note that we could not compare other prior works because they are not open sourced [3, 45] and are very specific to the Xen hypervisor [42]. We evaluate these configuration on an eight socket, 80-core machine with Intel E7-8870 processors. Another point is that the current version of KVM partially addresses the BWW problem that can occur from the user space [22].

6.1 Overhead of *e*CS

We evaluate the cost of our lightweight para-virtualized APIs on various blocking and non-blocking locks, and RCU. Table 4 enumerates the overhead of the sole API cost including the cost of executing a critical section with a simple microbenchmark that executes an empty critical section to quantify the impact of *e*CS API on these primitives in both lowest (1 core) and highest contention (80 core) scenarios. *1 core* denotes that a thread is trying to acquire a critical section, whereas *80 core* denotes that 80 threads are competing. We observe that *e*CS adds an overhead of almost 0.9–18.4 ns in low contention, whereas negligible overhead in high contention scenario, except RCU. For RCU, the empty critical section

Critical sections	Time (ns)			
	1 core		80 core	
	W/o *e*CS	W/ *e*CS	W/o *e*CS	W/ *e*CS
API cost	–	16.4	–	16.4
`spinlock`	31.2	44.8	4,782.3	4,772.9
`rwlock` (read)	32.0	38.8	2,418.2	2,519.4
`rwlock` (write)	27.4	45.8	4,363.3	4,784.5
`mutex`	33.5	34.4	49,116.4	48,125.7
`rwsem` (read)	35.6	36.6	2,588.8	2,737.0
`rwsem` (write)	33.3	38.1	7,055.7	7,150.1
`RCU`	9.8	19.7	9.8	19.8

Table 4: Cost of using our lightweight para-virtualized APIs with various synchronization primitives and mechanism. *1 core* and *80 core* denote the time (in ns) to execute an empty critical section with one and 80 threads, respectively. Although, our approach slightly adds an overhead on a single core count, there is no performance degradation for our evaluated workloads.

suffers from almost twice the overhead because both `RCU`'s lock/unlock operations do a single memory update on the `preempt_count` variable for a preemptable kernel. Even though our APIs add an overhead in the low contended scenario, we do not observe any performance degradation for any of our evaluated workloads.

6.2 Performance in an Over-committed Scenario

We evaluate the performance of the aforementioned workloads in an over-committed scenario by running two VMs in which each `vCPU` from both VMs share a physical CPU. Figure 3 (i) shows the throughput of these workloads for `PVM`, `HVM`, and *e*`CS`; (ii) shows the number of unavoidable preemptions that we capture while running these workloads when a `vCPU` is about to be scheduled out for *e*`CS`; and (iii) represents the percentage of types of observed preemptions, namely, `LHP` for blocking (`B-LHP`) and non-blocking (`NB-LHP`) locks, `RP`, `RRP`, `ICP` problems that we observe for the *e*`CS` configuration, including both avoided and unavoided preemptions.

Apache. *e*`CS` outperforms both `PVM` and `HVM` by 1.2× and 1.6×, respectively (refer (t:a) in Figure 3). Moreover, our approach reduces the number of possible preemptions by 85.8–100% (refer (n:a)) because of our rescheduling approach. We cannot completely avoid all preemptions because of our schedule penalization approach, as some of the preemptions occur consecutively. Even though *e*`CS` adds overhead, especially to `RCU`, it still does not degrade the scalability for four reasons: 1) We address the `BWW` problem, which allows for more opportunities to acquire the lock on time; 2) both hypervisor → VM APIs allow cooperative co-scheduling of the VMs; 3) our extra schedule approach avoids 85.8–100% of captured preemptions with the help of our VM → hypervisor APIs; and 4) the APIs overhead partially mitigates the highly contended system at higher core count by acting as a back-off mechanism. Another interesting observation is that we observe almost every type of preemption (refer Figure 3 (p:a)) because of serving the static pages, which involves blocking locks for the socket connection and `softirq` and spinlocks use for the interrupts processing. In particular, the number of preemptions is dominated by `LHP` for non-blocking and blocking locks, followed by `ICP` and then `RP`. We believe that the `ICP` problem will further exacerbate with optimized interrupt delivery mechanisms [12, 43]. `PVM` is 1.36× faster than `HVM` at 80 cores because of the support of para-virtualized spinlock (`qspinlock` [20]) as well as the asynchronous page fault mechanism that decreases the contention [30].

The major bottleneck for this workload is the interrupt injection, which can be mitigated by proposed optimized methods [12, 43]. In addition, Figure 4 (b) presents the latency CDF for the Apache workload at 80 cores in both under- and over-compression case. We observe that *e*`CS` not only maintains almost equivalent latency as that of `PVM` in an under-committed case, but also decreases in the over-committed case by 10.3–17% and 9.5–27.9% against `PVM` and `HVM`, respectively.

Psearchy mostly stresses the writer side of `rwsem` as it performs 20,000 small and large mmap/munmap operations along with stressing the memory allocator for inode operations, which mostly idles the guest OS because of the non-work conserving blocking locks [16]. Figure 3 (t:b) shows the throughput, in which *e*`CS` outperforms both `PVM` and `HVM` by 2.3× and 1.7×, respectively. The reason is that we 1) partially mitigate the `BWW` problem with our *e*SchdSpin approach, and 2) decrease the number of preemptions by 95.7–100% with an extra schedule (refer (n:b)). In addition, our *e*SchdSpin approach decreases the idle time from 65.4% to 45.2%, as it allows waiters to spin than schedule out themselves, which severely degrades the scalability in a virtualized environment, as observed for both `PVM` and `HVM`. This workload is dominated by mostly blocking and non-blocking locks, as they account to almost 98% preemptions (refer (p:b)). We also observe that `HVM` outperforms `PVM` by 1.33× because the asynchronous page fault mechanism introduces more `BWW` issue as it schedules out a `vCPU` if the page is not available, which does not happen for `HVM`.

Metis is a mix of both page fault and mmap operations that stress both the reader and the writer of the `rwsem`. Hence, it also suffers from the `BWW` problem, as we observe in Figure 3 (t:c). *e*`CS` outperforms `PVM` and `HVM` by 1.3× at 80 cores because of the reduced `BWW` problem and decreased preemptions that account to 91.4–99.5% (Figure 3 (n:c)). Note that the reader preemptions are 20%, thereby illustrating that readers preemptions is possible for read-dominated workloads, which has not been observed by any prior works. We do not observe any difference in the throughput of `HVM` and `PVM`.

Pbzip2 is an efficient compression/decompression work-

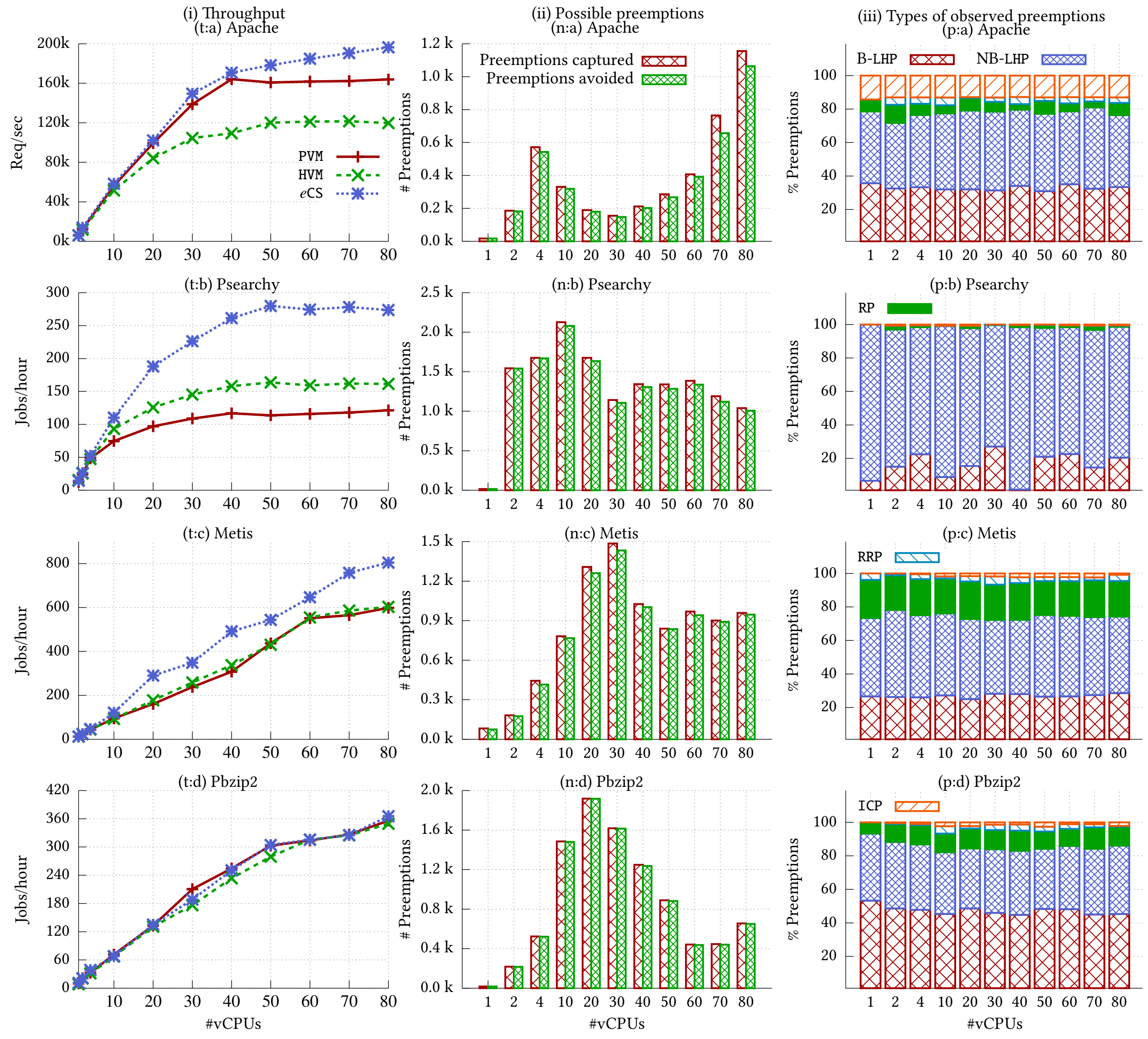

Figure 3: Analysis of real-world workloads in an over-committed scenario, *i.e.*, two instances of VM are executing the same workload. Column (i) represents the scalability of selected workloads in three settings: PVM, HVM, and with *e*CS annotations. Column (ii) represents the number of preemptions caught and prevented by the hypervisor with our APIs. Column (iii) represents the type of preemptions caught by the hypervisor (refer Table 4). By allowing an extra schedule, our approach reduces preemptions by 85–100% and improve scalability of applications by up to 2.5×, while observing almost all types of preemptions for each workload.

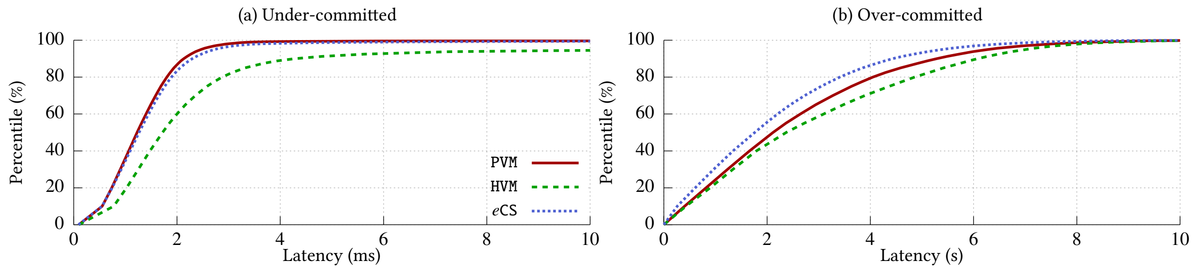

Figure 4: CDF of the latency of requests for the Apache web server workload in both under- and over-committed scenarios at 80 cores. It clearly shows the impact of *e*CS in the over-committed scenario, while having minimal impact in the under-committed case.

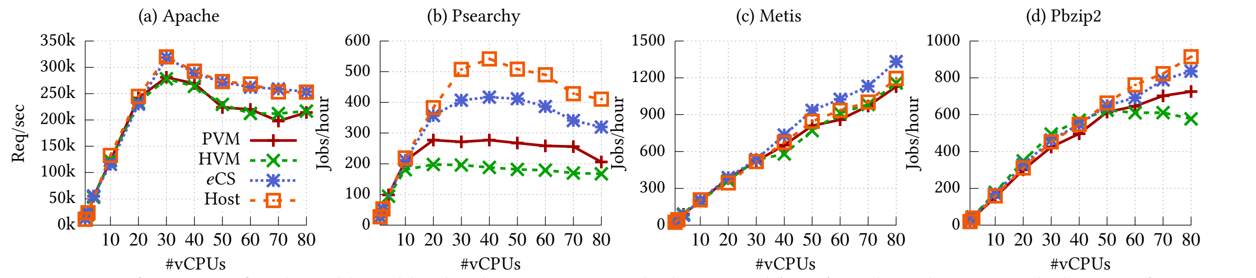

Figure 5: Performance of real-world workloads when running on the bare metal (Host), and inside a VM with three configurations: `PVM`, `HVM`, and with *e*CS annotations. In this scenario, only one VM is running. We use Host as the baseline for the comparison because we consider Host to have almost optimal performance.

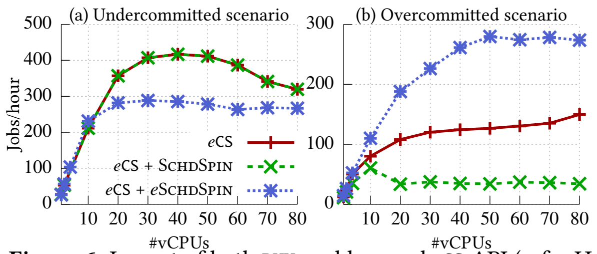

Figure 6: Impact of both `BWW` problem and *e*CS API (refer Hypersivor → VM in Table 1) on Psearchy in both under- and over-committed scenarios.

load that spends only around 5% of the time in the kernel space. Figure 3 (t:d) shows that the performance of *e*CS is similar to `PVM` and `HVM`, while decreasing the number of preemptions by 98.4–100% (refer (n:d)). We do not observe any performance gain in this scenario because 1) these preemptions may not be too critical to affect the application scalability, and 2) the overhead of our APIs, which do not provide any gains even after decreasing the preemptions. Similar to the other workloads, `LHP` dominates the preemption, followed by `RP`, `ICP`, and `RRP`.

In summary, our APIs not only reduce preemptions by 85–100%, but also improve the scalability of applications that use these synchronization primitives up to 2.5×, while no observable overhead on these applications. Moreover, we found that these preemptions occur for almost every type of primitives, specifically in the case of blocking synchronization primitives, read locks (Metis and Pbzip2), and interrupts (*e.g.*, TLB operations, packet processing etc.). In addition, most of the workloads still suffer from the `BWW` problem because of them being non-work conserving. We partially address this problem with the help of our *e*SchdSpin approach. One point to note is that we do not observe too many preemptions, as shown by prior works [42], because the current Linux kernel has dropped the FIFO-based Ticket `spinlock` and has replaced it with a highly optimized unfair queue-based lock [20] that mitigates the problem of `LHP` and `LWP`.

6.3 Performance in an Under-committed Case

We evaluate our *e*CS approach against `PVM` and `HVM` configurations in which a VM is running to show the impact of both APIs and *e*SchdSpin approach. We also include bare-metal configuration (Host) as a baseline (Figure 5). We observe that *e*CS addresses the `BWW` problem, and outperforms both `PVM` and `HVM` in the case of Apache (1.2× and 1.2×), Psearchy (1.6× and 1.9×), Metis (1.2× and 1.3×), and Psearchy (1.2× and 1.4×), while having almost similar latency for the Apache workload (Figure 4 (a)). Likewise, *e*CS performance is similar to that of bare-metal, except for the Psearchy workload.

For Apache, our APIs act as a back-off mechanism to improve its scalability, as the system is heavily contended. The throughput degrades after 30 cores because of the overhead of process scheduling, socket overhead, and inefficient kernel packet processing. Besides this, both Psearchy and Metis suffer from the `BWW` problem, which we improve with our *e*SchdSpin approach that results in better scalability as well as reduction in the idling of VMs. In particular, we decrease the idle time of Psearchy and Metis by 25% and 20%, respectively, by using our approach. One point to note is that blocking locks are based on the `TAS` lock, whose throughput severely degrades with increasing core count because of the increase cache-line contention, which we observe after 40 cores for Psearchy for all configurations. We also find that the Host is still 1.4× faster than *e*CS because *e*SchdSpin only partially mitigates the `BWW` problem, while introducing excessive cache-line contention, which we can circumvent with NUMA-aware locks [16]. For Pbzip2, we observe that *e*CS performs equivalent to the Host, while outperforming `PVM` and `HVM` after 60 cores, because Pbzip2 spends the least amount of time in the kernel space (5%), and starts to suffer from the `BWW` problem only after 60 cores, which our *e*SchdSpin easily tackles.

6.4 Addressing `BWW` Problem via *e*CS

We evaluate the impact of the `BWW` problem on Psearchy in both under- and over-committed scenarios. Figure 6 (a) shows that our scheduling-aware spinning approach (marked as *e*CS + SchdSpin) improves the throughput of Psearchy by 1.5× and 1.2× at 40 and 80 cores, respectively, in an under-committed scenario. SchdSpin approach allows a blocking waiter, both reader and writer, to actively spin for the lock if the number of tasks in the run queue is one, else the task schedules itself out. This approach is similar to the scheduling-aware parking/wake-up strategy [16], which we applied to the stock `mutex` and `rwsem`.

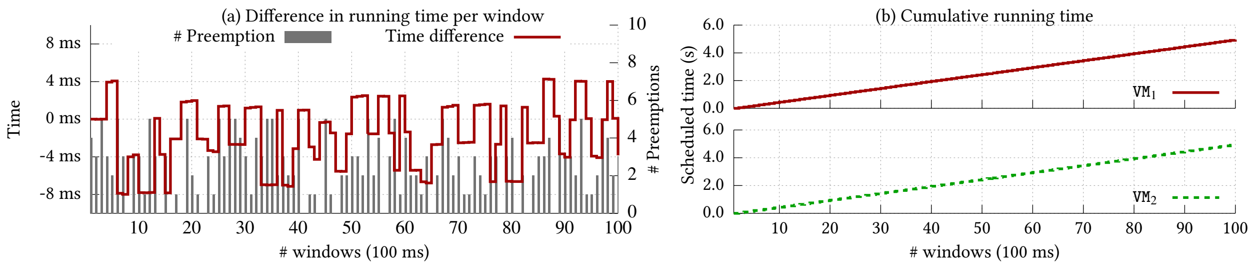

Figure 7: Fairness in *e*CS. Running time of a vCPU of two co-scheduled VMs (VM_1 and VM_2) with *e*CS annotations for a period of 10 seconds with 100 ms window granularity while executing a kernel intensive task (reading the contents of a file) that involves read side of rwsem. (a) shows the difference in running time of vCPU per window granularity as well as the number of preemptions occurring per window, while (b) illustrates the cumulative running time, and shows that the hypervisor maintains eventual fairness in the system, even if VM_2 is allowed extra schedules. Both VMs get 4.95 seconds to run.

As mentioned before, the reason for such an improvement is that the current design is not scheduling aware, as the waiter parks itself if it is unable to acquire the lock. With our approach, we try to mitigate this performance anomaly and allow the applications to scale further. Unfortunately, the scheduling-aware approach is inefficient in the case of the over-committed scenario, as shown in Figure 6 (b). The reason is that current waiters are guest OS agnostic, which leads to wasting CPU resources and resulting in more LHP and LWP problems, thereby degrading the scalability by almost 4.4× (marked *e*CS + SchdSpin in (b)) against a simple *e*CS configuration that still suffers from the BWW problem. We overcome this issue by using our `is_pcpu_overcommitted()` API that allows the SchdSpin approach to spin only when there is no active task on the pCPU's run queue; otherwise, the waiter is scheduled out when more than one task are in the run queue of the pCPU. By using our API (marked *e*CS + *e*SchdSpin), we outperform the baseline *e*CS approach by 1.8× and the *e*CS + SchdSpin approach by 8×.[3]

6.5 System Eventual Fairness

We now evaluate whether we are able to achieve eventual fairness while allowing *e*CS annotated VMs to obtain an extra schedule followed by local vCPU penalization. To evaluate the fairness, we run a simple micro-benchmark in two VMs (marked VM_1 and VM_2). VM_1 is a non-annotated VM, whereas VM_2 is an *e*CS annotated one. This micro-benchmark indefinitely reads the content of a file that stresses the read side of the rwsem and spends around 99% of the time in the kernel without scheduling out the task, thereby prohibiting the guest OS from doing any halt exits. Figure 7 (a) shows the time difference between two VM runtimes that we measure at every 100 ms window for each VM as well as the number of preemptions for VM_2 in that window. Figure 7 (b) shows the cumulative runtime of the VMs. We observe from Figure 7 (a) that even after allowing for extra schedules, the CFS scheduling policy balances out these extra schedules, which does not affect the runtime difference between VM_1 and VM_2. For example, at the end of one second window, marked 10, we observe that the number of extra schedules that the hypervisor granted VM_2 was 34 (34 milliseconds of extra time), but the runtime difference between VM_1 and VM_2 is 7.8 ms, which becomes -1.9 ms at the end of two seconds, while VM_2 received a total of 54 extra schedules (54 milliseconds). Hence, the extra schedule approach followed by our local vCPU penalization ensures that none of the tasks running on that particular physical CPU suffers from the fairness issue, also referred as eventual fairness. Moreover, Figure 7 (b) shows that both VMs get almost equivalent runtime in a lockstep fashion with both VMs getting almost 4.95 seconds at the end of 10 seconds.

7 Discussion

Our *e*CS approach addresses the problem of preemptions and BWW in both under- and over-committed scenarios by annotating all synchronization primitives and mechanisms in the kernel space. However, besides these primitives, kernel developers have to manually annotate a critical section if they want to avoid the preemptions while introducing their own primitives. One approach could be that the hypervisor can read the instruction pointer (IP) to figure out an *e*CS, but the guest OS must provide a guest OS symbol table to resolve the IP. In addition, the current design of *e*CS only targets the kernel space of a guest OS, and it is still agnostic of the user space critical sections such as pthread locks. Hence, we would like to extend our approach to the user space critical sections to further avoid the preemption problem, as we believe that *e*CS is a natural fit for multi-level scheduling. However, we need to communicate the scheduling hint down to the lowest layer effectively, which requires designing of the *e*CS composability extensions.

Our annotation approach does not open any security vulnerability because our approach is based on the paravirtualized VM, and it is similar to other approaches that share the information with the hypervisor [4, 19]. By using our virtualized scheduling-aware spinning ap-

[3]We have used *e*CS + *e*SchdSpin approach for our evaluation against PVM and HVM in §6.2 and §6.3.

proach (*e*SchdSpin), we partially mitigate the BWW problem. However, our Hypervisor → VM APIs expose scheduling information of the pCPU, but they only tell if a pCPU is overloaded or a vCPU is preempted. In addition, a VM cannot misuse this information as it will be later penalized by the hypervisor. There is also very slight possibility of priority inversion problem with our extra schedule approach. However, the window of that hypervisor-granted extra schedule is too small to incur priority inversion and performance, unlike co-scheduling approaches [41, 45] in which the scheduling window is in the order of several milliseconds.

8 Related work

The double scheduling phenomenon is a recurring problem in the domain of virtualization, which seriously impacts the performance of a VM. There have been comprehensive research efforts to mitigate this problem.

Synchronization primitives in VMs. Uhlig *et al.* [44] demonstrated the spinlock synchronization issue in a virtualized environment, which he addressed with synchronous hints to the hypervisor, and was later replaced by para-virtual hooks for the spinlock [8] for notifying the hypervisor to block the vCPU after it has exhausted its busy wait threshold. Meanwhile, other problems such as LWP [32], the BWW problem [5, 39], and RCU readers preemption problems were found. Gleaner [5] that addressed the BWW problem implemented a user space solution to handle tasks among a varying number of vCPUs, by manipulating tasks' processor affinity in the user space, which is difficult to maintain at runtime as it must accurately track each task launch and deletion. However, our *e*SchdSpin approach is user agnostic and mitigates the problem to certain extent for large core count.

Taebe *et al.* [42] addressed the LHP/LWP issue by exposing the time window from the hypervisor to the guest OS, which leverages this information that enables a waiter to either spin or join the waiting queue. However, their solution is not applicable to CFS [27] scheduler of Linux as it does not expose the scheduling window information. Their solution is orthogonal to our approach as we want the hypervisor to take a decision than the VM. Waiman Long [20] designed and implemented qspinlock that inherently overcomes the problem of LWP by exploiting the property of the TAS lock in the queue-based lock. It works by allowing the other waiters to steal the lock before joining the queue without disrupting the waiters' queue. However, qspinlock is still prone to LHP. Meanwhile, by annotating various locks as *e*CS, we confirm these problems, and further identify new sets of problems such as RP and ICP, and provide a simple solution to address the double scheduling phenomenon.

Partial handling of scheduling overhead in VMs. There have been several studies on virtualization overhead because of the software-hardware redirection [1, 39] and co-scheduling issues [17, 41, 45]. For example, VMware relies on relaxed co-scheduling [45] to mitigate double scheduling problem, in which vCPUs are scheduled in batches and the stragglers are synchronized within a predefined threshold. Besides this, other works have proposed balanced vCPU scheduling [41] or even IPI based demand scheduling [17]. However, these co-scheduling approaches suffer from CPU fragmentation. On the contrary, our approach neither introduces any CPU fragmentation nor it needs to synchronize the global scheduling information for all the vCPU of a VM because each vCPU is locally penalized by the hypervisor rather than synchronizing them among other vCPUs.

Song *et al.* [40] proposed the idea of dynamically adjusting vCPUs according to available CPU resources, while allowing guest OS to schedule its tasks. They used the approach of vCPU ballooning, which avoided the problem of double scheduling and was later extended by Cheng *et al.* [3] by designing a lightweight hotplug vCPU mechanism. Although their approach is effective in case of small VMs, it is complementary to our approach and may not scale effectively for large SMP VMs because of the overhead of migrating tasks from one vCPU to another as well as the frequent rescheduling of the targeted vCPUs. *e*CS, on the other hand, does not suffer from any explicit IPI and migration-specific tasks, as it only adds an overhead of a simple memory operations for a scheduling decision.

9 Conclusion

Double scheduling phenomenon is a well-known problem in the domain of virtualization that leads to several symptoms in the form of LHP, LWP, and BWW. We identify that it not only is limited to non-blocking locks, but also is applicable to blocking locks and reader side of locks. We present a single shot solution with our key insight: if a certain key component of a guest OS is allowed to proceed further, the guest OS will make forward progress. We identify these critical components as synchronization primitives and mechanism such as spinlocks, mutex, rwsem, RCU, and even interrupt context, which we call *enlightened critical sections* (*e*CS). We annotate *e*CS with our lightweight APIs that expose whether a VM is executing a critical section, which the hypervisor uses to provide an extra schedule at the scheduling boundary, thereby allowing the guest OS to progress forward. In addition, by leveraging the hypervisor scheduling context, a VM mitigates the effect of BWW problem with our simple virtualized spinning-aware spinning strategy. With *e*CS, we not only decrease the spurious preemptions by 85–100% but also improve the throughput of applications up to 1.6× and 2.5× in an under- and over-committed scenario, respectively.

References

[1] 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 XII, pages 2–13, New York, NY, USA, 2006. ACM.

[2] Amazon. Amazon Web Services, 2017. https://aws.amazon.com/.

[3] L. Cheng, J. Rao, and F. C. M. Lau. vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 2:1–2:14, London, UK, Apr. 2016. ACM.

[4] G. Costa. Steal time for KVM, 2011. https://lwn.net/Articles/449657/.

[5] 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'14, pages 73–84, Berkeley, CA, USA, 2014. USENIX Association.

[6] J. Fitzhardinge. Paravirtualized Spinlocks, 2008. http://lwn.net/Articles/289039/.

[7] T. A. S. Foundation. APACHE HTTP Server Project, 2017. https://httpd.apache.org/.

[8] T. Friebel. How to Deal with Lock-Holder Preemption. Technical report, Xen Summit, 2008.

[9] J. Gilchrist. Parallel BZIP2 (PBZIP2), Data Compression Software, 2017. http://compression.ca/pbzip2/.

[10] W. Glozer. wrk - a HTTP benchmarking tool, 2017. https://github.com/wg/wrk.

[11] Google. Compute Engine, 2017. https://aws.amazon.com/.

[12] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. Eli: Bare-metal performance for i/o virtualization. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS XVII, pages 411–422, London, UK, Mar. 2012. ACM.

[13] S. Kashyap. Vbench, 2015. https://github.com/sslab-gatech/vbench.

[14] S. Kashyap, C. Min, and T. Kim. Scalability In The Clouds! A Myth Or Reality? In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys)*, pages 5:1–5:7, New York, NY, USA, July 2015. ACM.

[15] S. Kashyap, C. Min, and T. Kim. Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds. *SIGOPS Oper. Syst. Rev.*, 50(1):9–16, Mar. 2016.

[16] S. Kashyap, C. Min, and T. Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 603–615, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6.

[17] 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, pages 369–380, New York, NY, USA, 2013. ACM.

[18] A. Kivity. sched: add notifier for process migration, 2009. https://lwn.net/Articles/356536/.

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

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

[21] Y. Mao, R. Morris, and F. M. Kaashoek. Optimizing MapReduce for Multicore Architectures. Technical report, MIT CSAIL, 2010.

[22] D. Matlack. Message Passing Workloads in KVM, 2015. http://www.linux-kvm.org/images/a/ac/02x03-Davit_Matalack-KVM_Message_passing_Performance.pdf.

[23] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

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

[25] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 71–85, Denver, CO, June 2016. USENIX Association.

[26] I. Molnar. Linux rwsem, 2006. http://www.makelinux.net/ldd3/chp-5-sect-3.

[27] I. Molnar. [patch] Modular Scheduler Core and Completely Fair Scheduler [CFS], 2007. https://lwn.net/Articles/230501/.

[28] I. Molnar and D. Bueso. Generic Mutex Subsystem, 2016. https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

[29] I. Molnar and D. Bueso. Generic Mutex Subsystem, 2017. https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

[30] G. Naptov. KVM: Add asynchronous page fault for PV guest., 2009. https://lwn.net/Articles/359842/.

[31] O. Nesterov. Linux percpu-rwsem, 2012. http://lxr.free-electrons.com/source/include/linux/percpu-rwsem.h.

[32] 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 '13, pages 191–200, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1266-0.

[33] A. Prasad, K. Gopinath, and P. E. McKenney. The RCU-Reader Preemption Problem in VMs. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 265–270, Santa Clara, CA, July 2017. USENIX Association.

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

[35] SAP. SAP HANA 2.0 SPS 02, 2017. http://hana.sap.com/abouthana.html.

[36] V. Seeker. Process Scheduling in Linux, 2013. https://www.prism-services.io/pdf/linux_scheduler_notes_final.pdf.

[37] A. Shi. [PATCH] rwsem: steal writing sem for better performance, 2013. https://lkml.org/lkml/2013/2/5/309.

[38] Q. Software. Demystifying CPU Ready (%RDY) as a Performance Metric, 2017. http://www.actualtechmedia.com/wp-content/uploads/2013/11/demystifying-cpu-ready.pdf.

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

[40] X. Song, J. Shi, H. Chen, and B. Zang. Schedule Processes, Not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 1:1–1:7, New York, NY, USA, 2013. ACM.

[41] O. Sukwong and H. S. Kim. Is Co-scheduling Too Expensive for SMP VMs? In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, pages 257–272, New York, NY, USA, Apr. 2011. ACM.

[42] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont. The Lock Holder and the Lock Waiter Pre-emption Problems: Nip Them in the Bud Using Informed Spinlocks (I-Spinlock). In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 286–297, New York, NY, USA, Apr. 2017. ACM.

[43] C.-C. Tu, M. Ferdman, C.-t. Lee, and T.-c. Chiueh. A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 1–15, New York, NY, USA, 2015. ACM.

[44] 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, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[45] VMware. The CPU Scheduler in VMware ESX 4.1. Technical report, VMware, 2010.

[46] VMware. Best Practices for Oversubscription of CPU, Memory and Storage in vSphere Virtual Environments, 2017. https://communities.vmware.com/servlet/JiveServlet/previewBody/21181-102-1-28328/vsphere-oversubscription-best-practices%5b1%5d.pdf.

[47] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.