The lock holder and the lock waiter pre-emption problems: nip them in the bud using informed spinlocks (I-Spinlocks)

Boris Teabe, Vlad-Tiberiu Nitu, Alain Tchana, Daniel Hagimont

The lock holder and the lock waiter pre-emption problems: nip them in the bud using informed spinlocks (I-Spinlock)

Boris Teabe Vlad Nitu Alain Tchana Daniel Hagimont

Toulouse University, France

first.last@enseeiht.fr

Abstract

In native Linux systems, spinlock's implementation relies on the assumption that both the lock holder thread and lock waiter threads cannot be preempted. However, in a virtualized environment, these threads are scheduled on top of virtual CPUs (vCPU) that can be preempted by the hypervisor at any time, thus forcing lock waiter threads on other vCPUs to busy wait and to waste CPU cycles. This leads to the well-known Lock Holder Preemption (LHP) and Lock Waiter Preemption (LWP) issues.

In this paper, we propose I-Spinlock (for Informed Spinlock), a new spinlock implementation for virtualized environments. **Its main principle is to only allow a thread to acquire a lock if and only if the remaining time-slice of its vCPU is sufficient to enter and leave the critical section.** This is possible if the spinlock primitive is aware (informed) of its time-to-preemption (by the hypervisor).

We implemented I-Spinlock in the Xen virtualization system. We show that our solution is compliant with both paravirtual and hardware virtualization modes. We performed extensive performance evaluations with various reference benchmarks and compared our solution to previous solutions. The evaluations demonstrate that I-Spinlock outperforms other solutions, and more significantly when the number of core increases.

Keywords spinlocks; multi-core; scheduler; virtual machine

1. Introduction

The last decade has seen the widespread of virtualized environments, especially for the management of cloud computing infrastructures which implement the Infrastructure as a Service (IaaS) model. In virtualized systems, a low level kernel called the hypervisor is running on the real hardware and provides the illusion of several hardwares (called virtual machines - VM) on top of which guest operating systems can be run.

In operating systems, spinlock is the lowest-level mutual exclusion mechanism for synchronizing access to shared data structures in the kernel on multicore architectures. Its implementation has a significant impact on performance and scalability of operating systems and applications [28, 29]. In native Linux systems, spinlock's implementation relies on the assumption that the locking primitive is not preemptable on its execution core.

However, in a virtualized environment, even if the guest OS scheduler does not allow preemption within the locking primitive, the guest OS (and more precisely its vCPUs) can be preempted by the hypervisor at any time, leading to the well-known Lock Holder Preemption (LHP) and Lock Waiter Preemption (LWP) issues. The LHP problem arises when a lock holder running on a vCPU is preempted by the hypervisor, thus forcing other lock waiter threads running on other vCPUs from the same VM to perform useless spinning and to waste CPU cycles. The LWP problem arises when a lock waiter (with a ticket, tickets enforcing an order among waiters) is preempted, thus forcing other waiters with higher ticket numbers to busy wait even if the lock was freed.

Several research works addressed these issues. Many of them consist in detecting the problematic busy waiting threads and scheduling the vCPUs of these threads out, in order to reduce wasted cycles. [1] proposes to monitor *pause* instructions from the hypervisor in order to detect busy waiting vCPUs. [5, 6, 7] introduce a hypercall in paravirtual systems, enabling the lock primitive to release the processor when busy waiting is detected. [8, 9] propose to enforce the simultaneous execution (coscheduling) of all vCPUs of the same VM, thus avoiding LHP and LWP. [10, 11, 25] propose to reduce the time-slice so that a preempted vCPU is rescheduled quickly. All these works tolerate LHP and LWP, but aim at limiting the side effects of LHP and LWP. Comparatively, our solution aims at limiting and almost cancelling the occurrences of LHP and LWP.

DOI: http://dx.doi.org/10.1145/3064176.3064180

In this paper, we propose I-Spinlock (for Informed Spinlock), a new spinlock implementation for virtualized environments. **The main principle is to only allow a thread to acquire a lock if and only if its time-to-preemption is sufficient to wait for its turn (according to its ticket number), enter and leave the critical section.** Therefore, I-Spinlock prevents LHP and LWP instead of detecting them and limiting their effects. In order to implement it, the spinlock primitive has to be aware (informed) of its time-to-preemption (provided by the hypervisor). We implemented I-Spinlock in the Xen virtualization system [31]. We performed extensive performance evaluations with various reference benchmarks (Kerbench, Pbzip2 and Ebizzy) and compared our solution to previous solutions. First, the evaluation results show that I-Spinlock does not compromize fairness and liveness and it does not incur any overhead on performance. Second, I-Spinlock is able to almost cancel LHP and LWP, leading to significant performance improvements compared to standard Linux ticket spinlocks. Finally, we compared our prototype with four existing solutions: *para-virtualized ticket spinlock* [6], *preemptable ticket spinlock* [5], *time slice reduction* solution [10] and *Oticket* [7]. The obtained results show that I-Spinlock outperforms all these solutions: up to 15% for para-virtualized ticket spinlock, up to 60% for preemptable ticket spinlock, up to 25% for time slice reduction, and up to 8% for Oticket when the number of core increases.

In summary, this paper makes the following contributions:

- We introduce I-Spinlock, a new spinlock implementation which avoids LHP and LWP in both types of virtualization (paravirtual and hardware-assisted);
- We implemented I-Spinlock in the Linux kernel and the Xen virtualization system;
- We performed extensive evaluations of I-Spinlock with reference benchmarks and compared it with four exiting solutions. The evaluations show that I-Spinlock outperforms exiting solutions.

The rest of the article is organized as follows. Section 2 presents the necessary background to understand our contributions and our motivations. A review of the related work is presented in Section 3. Section 4 presents our contributions while Section 5 presents evaluation results. The conclusion is drawn in Section 6.

2. Background and Motivations

2.1 Spinlocks in OSes

Spinlocks are the lowest-level mutual exclusion mechanism in the kernel [12]. They have two main characteristics: busy waiting [16] (the blocked thread spins while waiting for the lock to be released) and non pre-emption of both the blocked threads and the one holding the lock. This is why spinlocks are generally used for short duration locking. The critical section is supposed to be rapidly freed by the holder thread running on a different core. Spinlocks have a great deal of influence on safety and performance of the kernel. Therefore, several studies have investigated their improvement in Linux kernels [28, 29, 24], which is the research context of this paper. The rest of this section presents the two main spinlock improvements in Linux systems, embodied in kernel versions $\leq$ 2.6.24 and kernel versions $\geq$ 2.6.25.

Spinlocks in kernel versions $\leq$ 2.6.24. In these versions, a spinlock is implemented with an integer (hereafter noted l) which indicates whether the lock is available or held. The lock is acquired by decrementing and reading the value of l (with an atomic instruction). If the returned value is zero, the thread acquires the lock. Otherwise, the thread is blocked. l is set to one when the thread releases the lock. During the blocked state, the thread spins testing l. It stops spinning when l becomes positive. The main shortcoming of this implementation is unfairness. Indeed, there is no way to ensure that the thread which has been waiting the longest time obtains the lock first. This can lead to starvation. This issue was addressed in Linux kernel versions greater than 2.6.24.

Spinlocks in kernel versions $\geq$ 2.6.25. In these versions, spinlocks are implemented with a data structure called *ticket spinlock*, which includes two integer fields namely *head* and *tail*. These fields in a lock are respectively initialized to one and zero. Each time a thread attempts to acquire the lock, it first increments the value of tail and then makes a copy of it. This operation is called taking a ticket and the ticket number is the value of this copy. The acquisition of the lock by a thread is allowed if its ticket number is equal to the lock's head. The release of the lock is followed by the incrementation of its head. This way, the lock is acquired in a FIFO (First In First Out) order.

2.2 Scheduling in virtualized systems

In a virtualized environment, the underlying hardware is managed by the hypervisor, which is in charge of multiplexing hardware resources. For CPU management, the scheduler of the hypervisor is responsible for assigning virtual CPUs (vCPUs) to physical CPUs (pCPUs), see Fig. 1. Generally, this allocation is realized in a round-robin way. When scheduled on a pCPU, a vCPU runs during a period of time (called time slice, e.g. 30ms in Xen [22]) before being preempted. As a result, scheduling in a virtualized system is performed at two levels: guest OS-level and hypervisor-level. The latter has the last word in the sense that it controls the hardware. Fig. 1 illustrates a situation where VM1's thread t1 is given access to the processor, leaving VM2's thread (t2) out, although it has been scheduled in by VM2's scheduler. This situation seriously impacts guest OS performance [11, 10], especially when they run applications which perform a lot of spinlocks. The next section presents the issues related to this situation. For the sake of conciseness, the

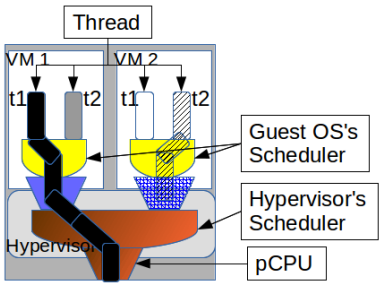

Figure 1. Scheduling in a virtualized system: This figure shows that scheduling in a virtualized system is performed at two levels: the guest OS level and the hypervisor level. The latter has the last word in the sense that it controls the hardware. For instance, VM1's thread t1 is given access to the processor, leaving VM2's thread (t2) out, whereas it is scheduled in by VM2's scheduler.

rest of the document assimilates a vCPU to the thread it runs.

2.3 Spinlock issues in virtualized systems

2.3.1 Description

To improve the performance of spinlock applications, the OS ensures that a thread holding a lock cannot be preempted. This caution is ineffective in a virtualized system since vCPUs are in their turn scheduled atop pCPUs, which are managed by the hypervisor. The hypervisor makes no distinction between vCPUs which are performing spinlocks and the others. Two issues can result from this situation, namely: the lock holder preemption problem and the lock waiter preemption problem. Before presenting these two problems, we first introduce some notations. Let us consider $V = \{v_1, .., v_n\}$ the list of vCPUs which attempt to acquire the same lock. V is sorted by ascending order of vCPU's ticket numbers. It implies that v_1 is the vCPU which actually holds the lock, according to the ticket spinlock assignation policy (FIFO).

The Lock Holder Preemption (LHP) problem. It occurs each time v_1 is scheduled out by the hypervisor, meaning that all v_i, $i > 1$, will consume their entire time-slice to carry out busy waiting until v_1 is rescheduled-in again.

The Lock Waiter Preemption (LWP) problem. The LWP problem occurs each time a v_i, $1 < i < n$, is scheduled-out and all v_k, $k < i$ have already used the lock. This situation prevents all v_j, $j > i$, to obtain the lock although it is free. They will consume their entire time-slice to carry out busy waiting until v_i is scheduled-in and can free the lock. This is because ticket spinlock enforces that the lock is acquired in a FIFO order, see Section 2.1.

2.3.2 Assessment

We realized a set of experiments in order to assess the above problems. We experimented with two benchmarks namely kernbench [17] and pbzip2 [14]. Each experiment consists in running the benchmark in two VMs on the same pool of CPUs with various numbers of vCPUs on our Dell machine (see Section 5.1 experimental environment informations). The two VMs run atop the same physical machine. We collected three metrics:

- the total number of *Problematic Spin Preemption* (noted PSP). It is the addition of the number of LHP and the number of LWP during the experiment.
- the total number of CPU cycles used for spinning. It corresponds to the wasted CPU time.
- the total number of CPU cycles used by the experiment. It corresponds to the execution time of the benchmark.

Benchmarks	#vCPUs	#PSP	Spinning cycles	Total cycles	(%)
Kernbench	4	2151	$2.223E8$	$2.3E11$	0.01
	8	17057	$5.452E11$	$6.3E11$	87.9
	12	74589	$6.70E11$	$7.7E11$	87.01
Pbzip2	4	140	$1.3E8$	$2.67E11$	0.4
	8	1542	$5.1E10$	$1.94E11$	26.1
	12	40114	$5.4E11$	$6.41E11$	84.5

Table 1. Impact of the LHP and the LWP problems on two representative benchmarks (Kernbench and Pbzip2 benchmarks): *#PSP* is the total number of Problematic Spin Preemption (#LHP+#LWP) during the experiment, while *Spinning cycles* and *Total cycles* are respectively the total CPU cycles burned due to spinning and the total CPU cycles consumed by the execution. (%)=$\frac{Spinningcycles}{Total\ cycles} \times 100$.

A LHP is detected if a thread acquires a lock in a quantum and releases it in a different quantum. A LWP is detected whenever a thread takes a ticket for a lock in a quantum and acquires the lock in a different quantum. The results of these experiments are reported in Table 1. Several observations can be made. First, the total number of PSP is significant. Second, it increases with the number of vCPUs. Third, the VM can spend a significant time spinning (up to 87.9%). Previous research studies [11, 10] led to the same conclusions. The next section presents the related work.

3. Related work

Several research studies have investigated the LHP and the LWP issues. The basic idea behind all these studies is to detect problematic spinning situations in order to limit them. According to the proposed detection approach, the related work can be organized into three categories.

3.1 Hardware approaches

Description. Hardware based solutions were first introduced in [1]. The authors proposed a way for detecting spinning vCPUs by monitoring the number of pause instructions (the "Pause-Loop Exiting" in Intel Xeon E5620 processors [19] and "Pause Filter" in AMD processors) executed by the guest OS in busy waiting loops. They consider that a vCPU which executes a high number of pause instructions is spinning. Therefore, the hypervisor can decide to preempt such

vCPUs.

Limitations. Unfortunately, even with this feature in place, it remains difficult to accurately detect spinning vCPUs [10]. In addition, even though the hypervisor has detected and preempted a spinning vCPU, it is tricky to decide on the next vCPU which should get the processor [20]. Indeed, it should be a vCPU which really needs the processor (not for spinning). [10] demonstrated that hardware based solutions have mixed effects: they improve some application's performance while degrading others.

3.2 Para-virtualized approaches

Description. Para-virtualized approaches try to address the issues by modifying the spinlock implementations directly in the guest OS kernel. In [2], the authors focused on the LHP problem. The guest OS kernel is modified (new hypercalls are introduced) in order to inform the hypervisor that a lock is acquired. Thus, the hypervisor prevents its preemption for the duration of the lock (it aims at implementing at the hypervisor level the non pre-emption characteristic of spinlocks in native kernels). [5] focused on the LWP problem and presented *preemptable ticket spinlocks*, a new ticket spinlock implementation. Preemptable ticket spinlocks allow an out of order lock acquisition when the earlier waiters are preempted. [6, 4] also proposed another spinlock implementation called *para-virtualized ticket spinlocks* (based on ticket spinlock) which addresses both the LHP and the LWP problems. Unlike typical ticket spinlock implementations, a para-virtualized ticket spinlock consists of two phases namely active and passive. When a vCPU tries to acquire a lock, it enters the active phase first. In this phase, the vCPU takes a ticket number and starts spinning. The number of spinning iterations is limited (a parameter configured at kernel compilation time). If the vCPU does not acquire the lock during these iterations, it moves to the passive phase. In this phase, the vCPU performs a yield instruction in order to release the processor (the vCPU is put in a sleep state). This instruction is para-virtualized, meaning that its execution is performed at the hypervisor-level. Subsequently, the hypervisor schedules-out the spinning vCPU. The latter will only be scheduled-in when the vCPU preceding it (according to the ticket number order) releases the lock. Notice that the lock release primitive is also para-virtualized, allowing the hypervisor to know when a lock is freed. Oticket [7] showed that this solution does not scale when the number of vCPUs sharing the same lock becomes too large. This comes from the utilization of hypercalls (for lock release and yield operations) which is known to introduce a significant overhead. To minimize the number of hypercalls, [7] proposed to increase the duration of the active phase for some vCPUs, i.e., those whose ticket number is close to the one of the vCPU which actually holds the lock.

Limitations. First, some of these solutions [2, 5] do not address the two issues at the same time. Second, almost all of these solutions tolerate LHP and LWP, and prefer to suspend vCPUs that would be spinning otherwise. Suspended vCPUs do not work for the VM and therefore impact its performance. Also, the increase of vCPU wake up operations may significantly degrade overall performance [30, 7]. Third, most of these solutions are only applicable to para-virtualized VMs. Hardware-assisted VMs are not taken into account because these solutions need to invoke some functions implemented in the hypervisor, and this is done via hypercalls which are not present in Hardware-assisted VMs.

3.3 Hypervisor scheduler based approach

Description. Solutions in this category advocate to take action in the hypervisor scheduler, rather than the guest OS. Solutions based on co-scheduling [8, 9, 26, 27] are situated in this category. It consists in enforcing the simultaneous execution of all vCPUs of the same VM. The preemption of a VM's vCPU implies the preemption of all its vCPUs. By doing so, the LHP and the LWP problems cannot occur. Another hypervisor-level solution is proposed in [10, 11, 25]. They propose to reduce the time-slice so that a preempted vCPU does not wait for a long time before being rescheduled-in. By this way, the spinning duration is reduced.

Limitations. Although co-scheduling addresses both the LHP and the LWP problems, it introduces several other issues (CPU fragmentation, priority inversion, and execution delay) [21]. These issues are more penalizing than the LHP and the LWP issues. Concerning time-slice reduction techniques, they increase the number of context switches, which is significantly degrading performance for CPU intensive applications [10, 11].

3.4 Position of our work and similar approach in non-virtualized system

In this paper, we adopt a different approach. Instead of trying to minimize the side effects caused by the LHP and the LWP issues, we try to prevent their occurrence as done by [32]. [32] proposed to minimize undesirable interactions between kernel level scheduling and user-level synchronization by providing each virtual processor with a "two-minute warning" prior to preemption. This two-minute warning avoids acquiring a spin-lock near the end of the virtual processor quantum, by yielding the processor voluntarily rather than acquiring a lock after the warning is set. If the warning period exceeds the maximum length of a critical section, the virtual processor will generally avoid preemption while holding a spin-lock. Furthermore, if the warning is triggered while the lock is held, a flag is positioned to indicate to the other threads that they should not spin for the lock but rather block.

We use a similar approach in the context of virtualized systems. Our solution works with both modified (para-virtualized) and unmodified (HVM) guest OSes.

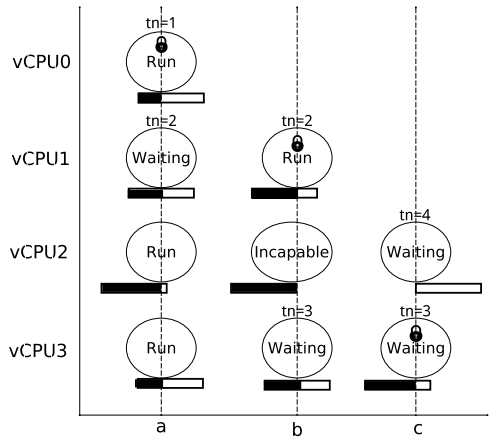

Figure 2. I-Spinlock illustration. We consider a VM configured with four vCPUs which try to access the same lock. The x-axis shows the different steps of I-Spinlock execution. The y-axis shows the consumption of each vCPU's quantum.

4. Informed Spinlocks (I-Spinlock)

This paper addresses both the LHP and the LWP issues. To this end, we introduce a new spinlock implementation call Informed-Spinlock (I-Spinlock for short).

4.1 Description

Before the presentation of our idea, let us summarize the two issues we address. The LHP issue occurs when the vCPU of a thread holding a lock is preempted by the hypervisor scheduler (hereafter scheduler). The LWP issue occurs when the vCPU of a thread having a ticket is preempted and all its predecessors obtained the lock. Therefore, we can say that a problem occurs when the vCPU of a thread holding either a lock or a ticket is preempted. In order words, the LHP and the LWP issues are philosophically similar. **Their origin is the fact that a thread is allowed to acquire a lock/ticket while the remaining time-slice of its vCPU is not sufficient for completing the critical section.** Therefore, the basic (but powerful) idea we develop in this paper consists in avoiding this situation. **To this end, we propose to only allow a thread to acquire a ticket if and only if the remaining time-slice of its vCPU is sufficient to enter and leave the critical section.** Notice that this constraint on ticket acquisition (instead of lock acquisition) is sufficient because a thread takes a ticket before acquiring a lock.

To achieve its goal, I-Spinlock introduces a new metric (associated with each vCPU) called *lock completion capability (lock_cap* for short) which indicates the capability of its thread to take a ticket, wait its turn, acquire the lock, execute the critical section and release the lock before the end of the quantum. Notice that the quantum is set by the hypervisor. Given a vCPU v, its $lock\_cap$ is computed as follows:

$$lock\_cap(v) = r\_ts(v) - (lql + 1) \times csd \quad (1)$$

where $r\_ts(v)$ is the remaining time-slice of v before its preemption, lql is the total number of threads waiting for the lock (they already have tickets) and csd is the critical section duration. Having this formula, I-Spinlock works as follows. Every time a thread attempts to take a ticket, the guest OS kernel first computes the $lock\_cap$ of its vCPU. If the obtained value is greater than zero (the vCPU has enough time before its preemption), the thread is allowed to take a ticket. Otherwise, it is not allowed and it has to wait for the next quantum of its vCPU. Such threads are called *incapable threads* in I-Spinlock. Our approach is similar to the one presented in [32] for non virtualized system, called "two-minute warning"[1]. I-Spinlock follows the same direction by applying this approach to virtualized environments. Figure 2 illustrates the functioning of I-Spinlock stage by stage with a simple example. To this end, we consider a VM configured with four vCPUs (noted vCPU0-vCPU3). We assume that csd is the quarter of the quantum length. Initially (stage a), the lock is free. vCPU0 and vCPU1 threads attempt to take a ticket. vCPU0's thread is the first to take the ticket, thus the lock. vCPU1's thread takes the second ticket number because its $lock\_cap$ indicates that its remaining time-slice is enough for completing the critical section before being preempted. It enters a busy waiting phase. vCPU2's thread, followed by vCPU3's thread, attempt to take a ticket. Since $lock\_cap(vCPU2)$ shows that it is not able to complete the critical section within its remaining time-slice, its thread is not allowed to take a ticket. Subsequently, vCPU2's thread is marked *incapable* (Section 4.2 details the actions taken by *incapable* threads). This is not the case for vCPU3's thread which takes ticket number 3 (stage b). At stage c, vCPU2 starts a new quantum, resulting into a positive value for its $lock\_cap$. Then vCPU2's thread can take a ticket.

We can see that, theoretically, a kernel which uses I-Spinlock does not suffer from neither the LHP nor the LWP issues. However, implementing this solution raises two main challenges which are summarized by the following questions:

- $r\_ts(v)$: how can the guest OS be informed about the remaining time-slice of its vCPUs, knowing that this information is only available at the hypervisor-level?
- csd: how can the guest OS estimate the time needed to complete a critical section, knowing that they are not all identical?

Availability of the remaining vCPU time-slice ($r\_ts(v)$) in the guest OS. We don't consider the hypervisor as a completely black box as it is traditionally the case, see Fig. 4. I-Spinlock implements a memory region shared between the hypervisor and the guest OS, resulting in a gray box hypervisor. This way, the hypervisor can share information with guest OSes. In our case, this information is limited

[1] [32] provided to user-level threads two-minute warning before preemption.

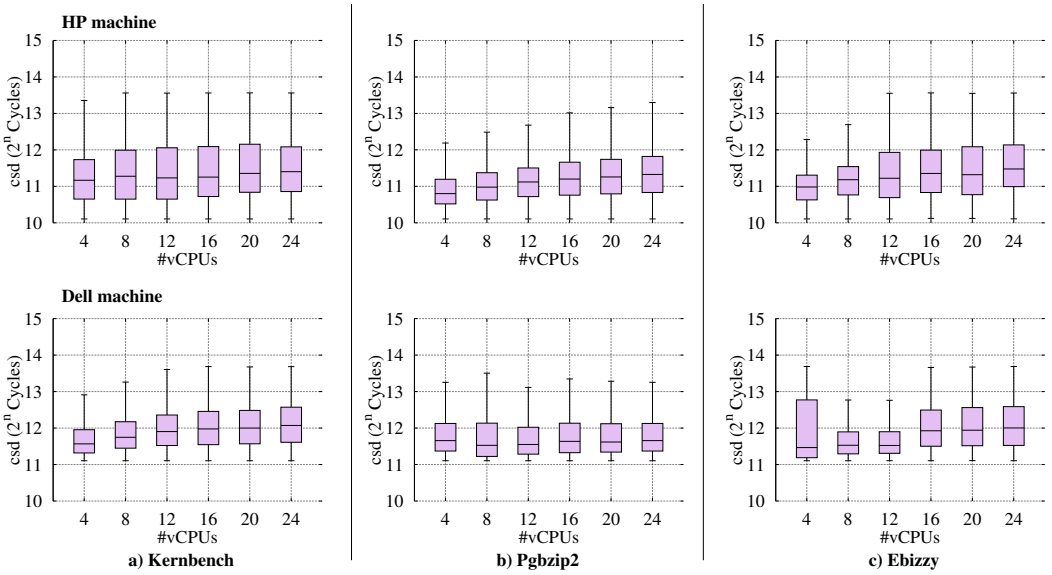

Figure 3. Estimation of the critical section duration (csd). We have experimented with three reference benchmarks namely Kernbench, Pgbzip2, and Ebizzy. These experiments have been carried out on several machine types and almost the same results have been obtained. Here we present the results for two of them namely DELL (top) and HP (bottom). The average csd is about 2^{12} Cycles, corresponding to what is reported in the literature [3]. In order to cover all cases, I-Spinlock uses 2^{14} Cycles as the estimated csd.

to the vCPU end of quantum time. This is not critical for the security of neither the hypervisor nor VMs. Therefore, each time a vCPU is scheduled-in, the hypervisor scheduler set into the shared memory region the exact time when the vCPU will be scheduled-out. This is called the *preemption time*. By doing so $r\_ts(v)$ can be computed in the guest OS each time a thread wants to take a ticket. The computation of $r\_ts(v)$ is given by the following formula:

$$r\_ts(v) = preemption\_time - actual\_time; \quad (2)$$

where $actual\_time$ is the actual time (when the thread is taking the ticket).

Computation of the critical section duration (csd). As previous research studies [3, 2], we estimate csd by calibration. To this end, we realised a set of experiments with different benchmarks (kernbench, Ebissy and Pgbzip2, presented in Section 5) with various numbers of vCPUs. The experiments have been carried out on different modern machine types and we observed the same results. Figure 3 presents the obtained results for two machine types: DELL and HP (their characteristics are presented in Section 5). First, we acknowledge that the average csd is about 2^{12} cycles. This corresponds to what was reported by other studies, realized on other machine types [3, 2]. In order to cover all critical section durations, I-Spinlock uses 2^{14} cycles as the estimated csd (corresponds to 6μsec on our machines). This value is larger than the maximum measured csd (Figure 3). By doing so, we minimize the number of false negative among *incapable* threads. Intensive evaluations confirmed that this value works well, also for other machine types (see Section 5). In addition, We experimented a machine learning solution which dynamically computes csd based on previous values. The evaluation results showed no improvement.

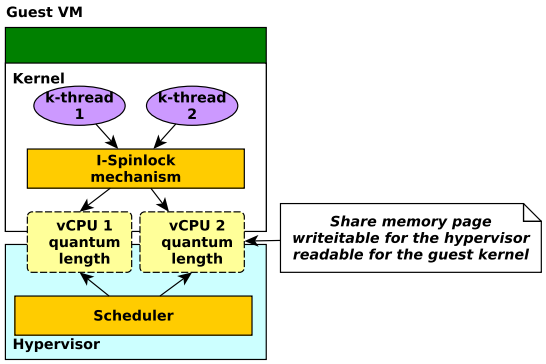

Figure 4. Availability of the remain vCPU time-slice ($r\_ts(v)$) in the guest OS. I-Spinlock uses a share memory between the hypervisor and the guest OS. The former is granted the write right while the latter has only the read right.

4.2 Implementation

Although our solution can be applied to every virtualization system, this section presents the implementation of a prototype within Xen 4.2.0 [31] (the most popular open source virtualization system) and Linux kernel version 3.13.0. This implementation involves both the hypervisor and the guest

OS kernel. The former is concerned by the implementation of the shared memory region while the latter is concerned by the implementation of I-Spinlock itself. Our ambition is to provide an implementation which works with both HVM (Hardware Virtual Machine) and PV (ParaVirtual) guest OSes. The HVM implementation is called HVM I-Spinlock while the PV implementation is called PV I-Spinlock.

4.2.1 Shared memory implementation

Our implementation takes advantage of the shared memory mechanism which already exists in Xen (as well as other virtualization systems). I-Spinlock especially exploits the $share\_info$ data structure.

Xen share_info data structure. Xen uses a shared memory page (called $share\_info$) to provide VMs with some hardware informations such as the memory size, necessary for the kernel boot process. Each VM has its own $share\_info$ which is used once by the hypervisor (at VM creation time) and the guest OS (at kernel boot time). The way in which the mapping of $share\_info$ is established differs between PV and HVM. In PV, $share\_info$ is allocated by the hypervisor and appears at a fixed virtual address in the guest kernel's address space. The corresponding machine address is communicated to the guest kernel through the xen_start_info data structure. In HVM mode, the guest has full control over its physical address space. It can allocate the $share\_info$ data structure in any of its physical page frames. The chosen physical address is communicated from the guest kernel to the hypervisor.

Utilization of share_info in I-Spinlock. Instead of adding new additional shared pages between the hypervisor and the guest OS, we exploit the unused part of $share\_info$ as follows. The hypervisor scheduler is modified so that each time a vCPU is scheduled-in, its end of quantum time is stored in the $share\_info$ data structure of its VM. The end of quantum time is expressed as a number of CPU cycles. Each vCPU has a dedicated entry in $share\_info$. The entire modification we performed in the hypervisor consists of about 10 lines of codes.

4.2.2 Ticket and Lock acquisition

Our implementation acts as an amelioration of ticket spinlocks in Linux. The listing of Fig. 5 presents the patch that should be applied to the kernel, interpreted as follows. Routine $arch\_spin\_lock$ is invoked every time a thread attempts to perform a spinlock. The vCPU end of quantum time, stored in the shared page by the hypervisor, is read at line 16. Using this information, the remaining time-slice ($r\_ts(v)$) is computed at line 18, according to equation 2. The number of vCPUs waiting for the lock (lql) is computed at line 20. lql is obtained by subtracting the tail value of the lock from its head value. The $lock\_cap$ of the vCPU is computed at line 22. If it is lower than zero, the thread is marked *incapable*. An *incapable* thread can take two possible paths depend-

```
1 #define AVERAGE_HOLD_TIME (1 << 15)
2 /*Get the actual time*/
3 +uint64_t rdtsc(void)
4 +{
5 +   unsigned int hi, lo;
6 +   __asm__volatile("rdtsc" : "=a" (lo), "=d" (hi));
7 +   return ((uint64_t)hi << 32) | lo;
8 +}
9
10 void arch_spin_lock(arch_spinlock_t *lock)
11 {
12    struct __raw_tickets inc = {.tail = TICKET_LOCK_INC
          };
13    /*Share_info page*/
14 +  struct shared_info *s = HYPERVISOR_shared_info;
15    /*Compute the remain time_slice*/
16 +  uint64_t time=rdtsc();
17 +  uint32_t cpu= cpuid();
18 +  uint64_t remaining_slice= s->vcpu_info[cpu].
       time_slice-time;
19    /*Compute the number of threads waiting for the
          lock*/
20 +  uint8_t dist = lock->tickets.head - lock->tickets.
       tail;
21    /*Compute the vCPU's lock_cap*/
22 +  int64_t lock_cap= remaining - (dist + 1) *
       AVERAGE_HOLD_TIME ;
23    /*Decide if the thread is allowed to take a ticket
          */
24 +  if(lock_cap < 0)
25 +  {
26    /*The thread is not allowed. It should wait the
          next quantum to take a ticket*/
27 +   #ifdef CONFIG_HVM_SPINLOCKS
28
29 +    u64 slice_ID = s->vcpu_info[cpu].slice_ID;
30 +    do
31 +    {
32 +         cpu_relax();
33 +      }while(slice_ID==s->vcpu_info[cpu].slice_ID);
34 +    #else
35 +    __halt_cpu(cpu);
36 +    #endif
37
38 +  }
39    inc = xadd(&lock->tickets, inc);
40    if(likely(inc.head == inc.tail))
41      goto out;
42    for (;;)
43    {
44      do
45      {
46        if (ACCESS_ONCE(lock->tickets.head)==inc.tail)
47                goto out;
48        cpu_relax();
49      }
50    out:
51    barrier();
52 }
```

Figure 5. Implementation of I-Spinlock in the guest OS. This is the patch that should be applied to the Linux kernel. This implementation is compatible with both HVM and PV modes.

ing on the virtualization mode. It performs a busy waiting in the case of HVM guest OSes (lines 29-33) or it releases the processor in the case of PV guest OSes (line 35). The release of the processor is possible in the PV mode because its supports the hypercall framework. Therefore instead of entering in a busy waiting phase (HVM mode), the *incapable* thread invokes a hypercall (PV mode) which tells the hypervisor to schedule-out the vCPU. By doing so PV I-Spinlock avoids the waste of CPU cycles unlike HVM I-Spinlock. The evaluation results show that this optimisation makes PV I-Spinlock more efficient than HVM I-Spinlock.

One may think that our solution could lead to no improvement if the vCPU is pre-empted before the end of its time-slice (e.g. the hypervisor preempts the vCPU in order to treat an IO interrupt). This issue can be easily handled by informing the VM with the minimal duration of the quantum instead of the time-slice (this parameter is called *ratelimit* in Xen).

5. Evaluations

This section presents the evaluation results of I-Spinlock. We evaluate the following aspects:

- starvation and unfairness risks: the capability of I-Spinlock to avoid both starvation and unfairness.
- effectiveness: the capability of I-Spinlock to address both the LHP and the LWP issues.
- genericity: the capability of I-Spinlock to take into account both PV and HVM guest OSes.
- overhead: the amount of resources consumed by I-Spinlock.
- scalability: the capability of I-Spinlock to deal with heavy lock contention guest OSes.
- positioning: the comparison of I-Spinlock with other solutions.

We first present the experimental environment before the presentation of the evaluation results.

5.1 Experimental environment

To demonstrate the robustness of I-Spinlock, it has been evaluated with different benchmark types, running on different machine types. We used both micro- and macro-benchmarks that have been used in many other works [5] (they perform a lot of spin-locks).

Benchmarks.

- A micro-benchmark, developed for the purpose of this work. It starts n threads which attempt to access x times the same memory area which is the critical session. The number of threads is the number of vCPUs of the VM. The performance of the application is given by two metrics namely the average execution time of all threads and the standard deviation. This benchmarks is built to perform a lot of parallel locking, so to produce a lot of lock contention.

Experimental procedure and configurations	
We run two instances of the same benchmark in two VMs, both in either Hardware-assisted (HVM) or Para-virtual configurations. We use HVM VMs when evaluating HVM IS and PV VMs when evaluating PV IS. All vCPUs share the same pool of physical CPUs (pCPUs) during the experiments.	
Abbreviations	
TSL	Ticket Spinlock
HVM IS	HVM I-Spinlock
PV IS	PV I-Spinlock
PTS	Preemptable Ticket Spinlock
TS	Time Slice
PV TS	Para virtual Ticket Spinlock
OT	Oticket

Table 2. Experimental setup: experimental procedure and configurations, and adopted abbreviations.

- Kernbench [17] is a CPU intensive benchmark which consists of parallel Linux kernel compilation processes. Its performance is given by the duration of the entire compilation process (expressed in seconds).
- Pbzip2 [14] is a parallel implementation of bzip2 [15], the block-sorting file compressor command found in Linux systems. It uses pthreads and achieves near-linear speedup on SMP machines. Its performance is given by the duration of the compression (in seconds).
- Ebizzy [18] generates a workload resembling common web application server workloads. It is highly threaded, has a large in-memory working set, and allocates and deallocates memory frequently. Its performance is measured as the sustained throughput (records/second).

Hardware.

The experiments were carried out on two machine types running the same Linux distribution (ubuntu server 12.04):

- Dell: 24 cores (hyper threaded), 2.20 GHz Intel Xeon E5-2420, 16 GB RAM, 15 MB LLC.
- HP: 8 cores, 3.2 GHz Intel core i 7, 16 GB RAM, 8 MB LLC

The results obtained from the two machines are almost the same. Therefore, this section only presents the DELL machine results.

Experimental procedure.

Unless otherwise specified, each experiment is realized as follows. We run, two instances of the same benchmark in two VMs. The execution is then repeated with different numbers of vCPUs. All vCPUs share the same pool of physical CPUs (pCPUs). All benchmarks are configured to use an in-memory filesystem (*tmpfs*) to alleviate the side effect of I/O operations. Also all the experiments were repeated 5 times. Table 2 summarizes the abbreviation use in this section.

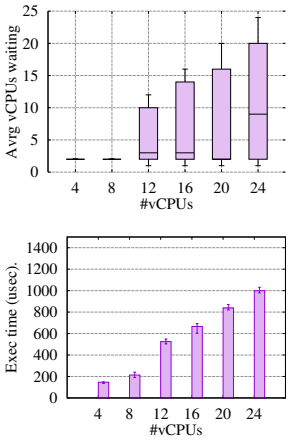

Figure 6. Evaluation of the starvation and the fairness risks. This experiment was realized with the micro-benchmark. (top) The average number of vCPUs of the same VM waiting for the lock (lql). (bottom) The average execution time of all threads, including the standard deviation. These results show that I-Spinlock ensures fairness and avoids starvation, as the standard deviation of execution times is kept small, even if lock contention is high.

5.2 Starvation and unfairness risks

Ticket spinlock uses the first-come first-served (FCFS for short) lock allocation policy to avoid starvation in the one hand and to ensure fairness on the other hand. Knowing that I-Spinlock breaks down the FCFS policy, one could legitimately ask how I-Spinlock deals with starvation and fairness. This section discusses and evaluates these aspects (we will see in Section 5.5.1 that this is an issue for other solutions).

The FCFS policy is implemented in ticket-spinlocks by imposing each thread to take a ticket before trying to acquire a lock. This is not completely broken down in I-Spinlock because it is still applied to threads except those which are marked *incapable*. Therefore, only *incapable* threads are concerned by starvation and unfairness risks. We explain in the rest of this paragraph that this situation is not problematic. To this end, we show that in I-Spinlock, an *incapable* thread will not wait for a long time before taking a ticket. Indeed, the combination of two factors allow I-Spinlock to naturally avoid starvation and unfairness. First, when the *incapable* vCPU gets back to the processor, it is allocated a new quantum and the *incapable* thread will run first. This is because the guest OS scheduler cannot preempt the *incapable* thread since it is in the spinlock routine. This is the traditional functioning of the kernel scheduler towards threads which perform spinlocks. The second factor is the fact that the quantum length of a vCPU is far greater than the duration of the critical section (csd), resulting to a positive value of $lock\_cap$ at the beginning of any quantum. Indeed, the quantum length is in the order of millisecond (e.g. 30msec in Xen [22] and 50msec in VMware [23]) while csd is in the order of microsecond (2^{14} cycles which corresponds to 6μsec on our machine), see Fig. 3 in Section 4.1). Therefore, the probability to have a negative $lock\_cap$ at the beginning of the vCPU quantum is practically nil. It could only be otherwise in the case of a VM with $\frac{quantum\ length \times 1000}{6}$ vCPUs having a ticket (corresponding to the value of lql in equation 1). This represents about 5000 active vCPUs in a Xen guest OS and about 8333 vCPUs in a VMware guest OS, which is practically impossible (it is rather in the order of hundreds). To validate this demonstration we experimented with the micro-benchmark which generates a lot of lock contention. We are interested in three metrics namely: the value of lql, the average execution time of all threads and the standard deviation. These results are shown in Fig. 6. We can make the following observations. First, due to the number of lock contention, lql can be as large as the number of vCPUs (Fig. 6 top). Second, in each experiment, all threads run within almost the same duration (Fig. 6 bottom), since the standard deviation is almost nil. This means that threads fairly acquire the lock within a reasonable latency. In summary, I-Spinlock guarantees that a previous *incapable* thread will always get the ticket during the next quantum of its vCPU, thus avoiding starvation and unfairness.

5.3 Effectiveness

The metric which allows to evaluate the effectiveness of I-Spinlock is the number of *Problematic Spin Preemption* (PSP). We collected the value of this metric in three situations: standard ticket spinlock (the base line noted TS, HVM I-Spinlock (noted HVM IS), and PV I-Spinlock (PV IS). Figure 7 presents the results of these experiments. The top curve presents the normalized (over the base line) performance of the studied benchmarks while the bottom curve presents the total number of PSP. We can see on the bottom curve that with ticket spinlocks, the impact of both the LHP and the LWP issues increases with the number of vCPUs (corresponding to heavy lock contention guest OSes). I-Spinlock does not suffer from these issues (the number of PSP is far much smaller), resulting into no performance degradation (top curve). We can also notice that the number of PSP with I-Spinlock is kept constant when the number of vCPUs increases, which demonstrates the scalability of I-Spinlock. PV IS performs a little better (with less PSP) than HVM IS because *incapable* threads waste CPU cycles in the HVM mode while they release the processor in the PV mode (see Section 4.2.2). Finally, notice that I-Spinlock cannot totally prevent PSP as several threads may compute their $lock\_cap$ at the same time and their lql values may be erroneous.

5.4 I-Spinlock overhead

We evaluated the overhead of both HVM IS and PV IS in terms of the number of CPU cycles each of them needs. To

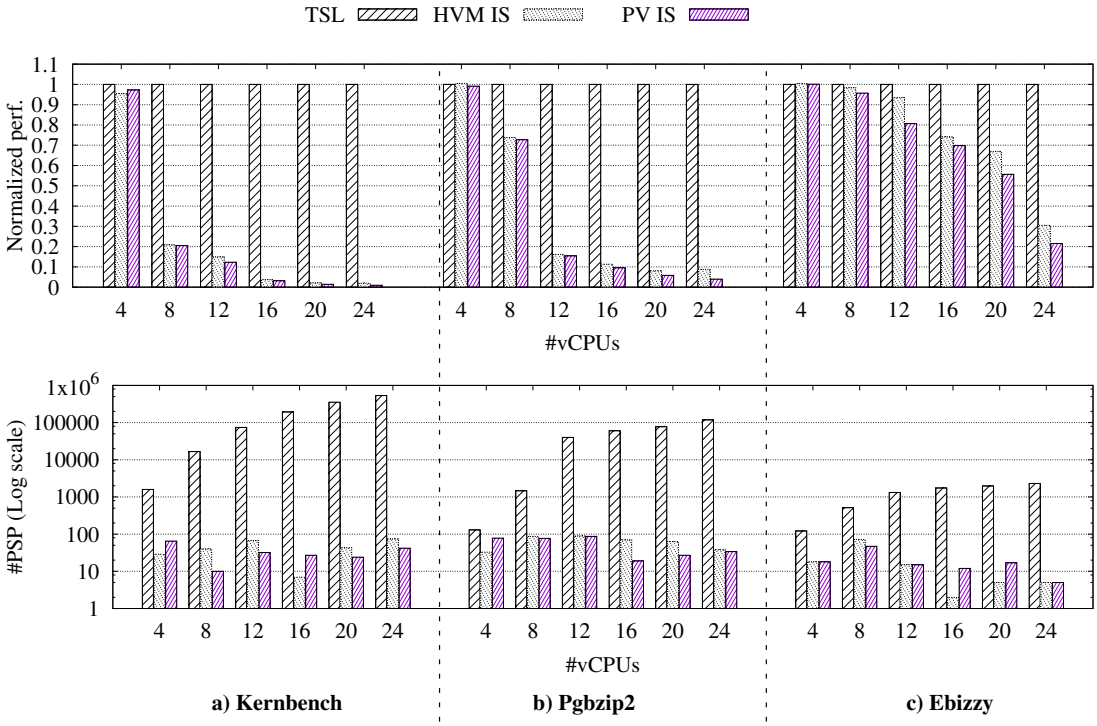

Figure 7. Effectiveness of I-Spinlock. We evaluate the performance and the number of Problematic Spin Preemption (PSP) for three benchmarks with three spinlock implementations: standard ticket spinlock (the base line), HVM I-Spinlock, and PV I-Spinlock. The top curve presents the normalized (over the base line) performance while the bottom curve presents the total number of PSP. We can see that both HVM I-Spinlock and PV I-Spinlock significantly reduce the number of PSP with a important impact on performance. Also, the number of PSP is kept constant when the number of vCPUs increases, which demonstrates the scalability of I-Spinlock.

this end, we ran a single kernbench in a 24 vCPUs VM. We ran the VM alone in order to avoid the LHP and LWP issues (which are not the purpose of the experiment here). Figure 8 (a) shows the number of CPU cycles needed for acquiring a lock while Figure 8 (b) depicts the normalized execution time of the benchmark. The normalization is realised over the ticket spinlock results. We can observe that the number of CPU cycles used by either HVM IS or PV IS is slightly higher than the number of CPU cycles used by ticket spinlock. This comes from the additional operations introduced by I-Spinlock, see the implementation in Section 4.2. From Figure 8 (b), we can see that the impact of this overhead is negligible because both I-Spinlock and ticket spinlock lead to the same results here. In summary, the overhead of our solution is almost nil.

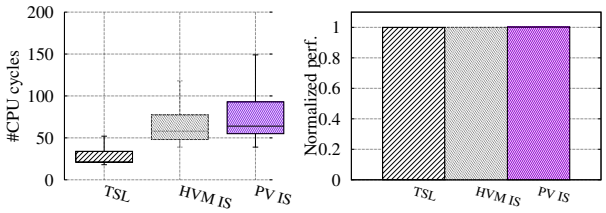

Figure 8. Overhead estimation. (a) We evaluated the number of CPU cycles needed to acquire a lock in both I-Spinlock and ticket spinlock. (b) We can see that the former is slightly larger than the latter But this slight difference does not incur any overhead.

5.5 Comparison with other solutions

We compared I-Spinlock with existing solutions, discussed in the related work section.

5.5.1 HVM compatible solutions

We compared HVM IS with the two main existing HVM compatible solutions namely preemtable ticket spinlock (noted PTS) [5] and time sliced [10] (noted TS). Figure 9 top presents the normalized performance of kernbench, normalized over HVM IS. Let us start by analysing PTS's results. PTS provides almost the same performance as HVM IS with small VM sizes (4 and 8 vCPUs). However, it does not scale with bigger VMs (#vCPUs $>$ 12). This is explained by the fact that PTS allows an out of order lock acquisition when the earlier waiters are preempted. Therefore, the more the number of vCPUs (increasing lock contention), the more there are threads trying to acquire the same lock simultaneously without established order. This brings PTS back

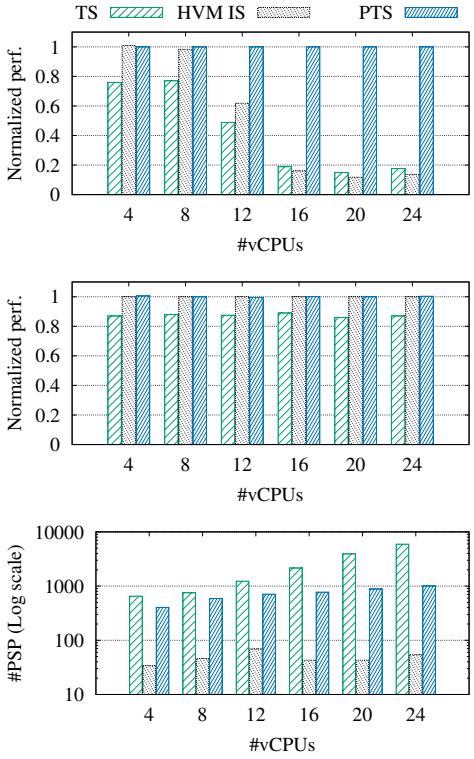

Figure 9. Comparison of HVM I-Spinlock (HVM IS) with two existing HVM compatible solutions: PTS [5] and TS [10]. (top) Performance of the kernbench VM (in which solutions are implemented). (middle) Performance of a collocated (CPU bound, the y-cruncher [13] benchmark) VM, which can suffer from spinlock solutions which are applied to kernbench. (bottom) Total number of PSP. HVM IS provides the best performance without impacting collocated VMs.

to the old implementations of spinlock (see Section 2.1) where the fastest thread will always acquire the lock, resulting to possible starvations. Also, PTS does not address the LHP problem. Regarding TS [10], we observe that it slightly outperforms HVM IS when the number of vCPU is low (#vCPUs<16). We can observe that the #PSP does not have a huge effect on TS. This is because with a small time slice, the time wasted spinning by vCPUs due to PSP is reduced. However as we said in the related work (Section 3), TS's approach consists in reducing the vCPU time slice. This increases the number of context switches, which is known to be negative for CPU bound VMs [11, 25]. Fig. 9 middle shows the performance collocated of a VM which runs several instances of a CPU bound application (y-cruncher [13]). We observe a performance degradation of up to 13% for the CPU bound application when TS [10] is used. In summary, HVM IS provides better results compared to both PTS [5] and TS [10].

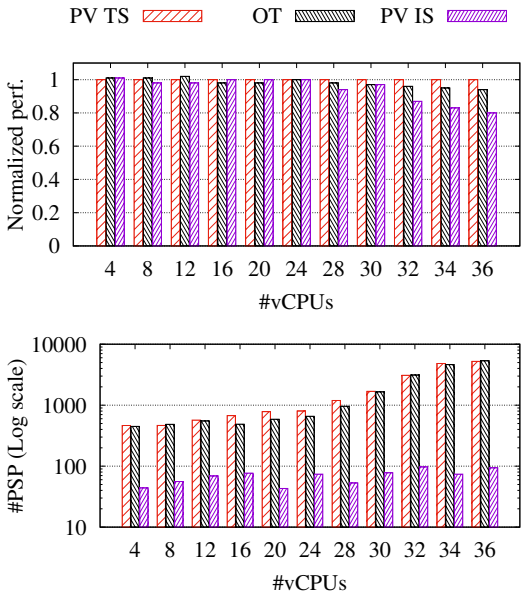

Figure 10. Comparison of PV I-Spinlock (PV IS) with two existing PV compatible solutions (PV TS [6] and OT [7]). (top) The normalized performance of kernbench (normalized over PV TS). (bottom) The number of PSP for each solution.

5.5.2 PV solutions

We also compared PV IS with the two main existing PV compatible solutions namely para-virtualized ticket spinlock [6] (noted PV TS) and Oticket [7] (noted OT). Figure 10 top presents the normalized performance of kernbench, normalized over PV TS. We can notice that all solutions provide almost the same results with low contention scenarios (low number of vCPUs). This is not the case with heavy lock contention scenarios where only our solution provides better results. This is explained by the fact that the number of PSP increases linearly with the number of vCPUs in other solutions while it is almost constant in PV IS. The increase of the number of PSP incurs more hypercall (during lock release and yield) which significantly impact performance for PV TS and OT [7].

6. Conclusion

This article presented I-Spinlock, a new spinlock implementation targeting virtualized systems. In such environments, spinlocks are victims of two major flaws, lock holder preemption (LHP) and the lock waiter preemption (LWP), which lead to wasted CPU cycle and performance degradation. I-Spinlock is able to avoid both LHP and LWP in both hardware virtualization (HVM I-Spinlock) and paravirtualization (PV I-Spinlock). The main principle of I-Spinlock is to allow a thread to acquire a ticket if and only if the remaining time-slice of its vCPU is sufficient to wait for its turn (according to its ticket number), enter

and leave the critical section, thus preventing any LHP or LWP. Both versions of I-Spinlock (HVM I-Spinlock and PV I-Spinlock) were implemented in the Xen system. In order to demonstrate its effectiveness, we experimented and evaluated I-Spinlock with several benchmarks (Kerbench, Pbzip2, Ebissy). We compared I-Spinlock with the ticket spinlock, paravirtualized ticket spinlock and time slice reduction approach, and showed that our solution outperforms those solutions.

Acknowledgements

We sincerely thank Katryn S McKinley and the anonymous reviewers for their feedback.

References

[1] P. M. Wells, K. Chakraborty, G. S. Sohi, "Hardware support for spin management in overcommitted virtual machines," *Proceedings of the 15th international conference on Parallel architectures and compilation techniques (PACT),* 2006 .

[2] V. Uhlig, J. LeVasseur, E. Skoglund, U. Dannowsk, "Towards Scalable Multiprocessor Virtual Machines," *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium (VM),* 2004

[3] T. Friebel, "How to deal with lock-holder preemption," *Presented at the Xen Summit North America,* 2008.

[4] K. Raghavendra, J. Fitzhardinge, "Paravirtualized ticket spinlocks," May 2012

[5] J. Ouyang, J. R. Lange, "Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud," *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE),* 2013

[6] J.Ouyang, "https://lwn.net/Articles/556141/, "

[7] S. Kashyap, C. Min, T. Kim, "Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds," *ACM SIGOPS Operating Systems Review,* 2016

[8] J. Ousterhout, "Scheduling techniques for concurrent systems," *IEE Distributed computer System,* 1982

[9] VMware, "I. Vmware(r) vsphere(tm): The cpu scheduler in vmware esx(r) 4.1," 2010

[10] J. Ahn, C. H. Park, J. Huh, "Micro-Sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems," *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO),* 2014

[11] T. Boris, A. Tchana, D. Hagimont, "Application-specific quantum for multi-core platform scheduler," *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys),* 2016

[12] J. Corbet, "https://lwn.net/Articles/267968/,"

[13] y-cruncher: A Multi-Threaded Pi-Program, "http://www.numberworld.org/y-cruncher/"

[14] J. Gilchrist, "http://compression.ca/pbzip2/," 2015

[15] bzip2, " http://www.bzip.org/," 2015

[16] R. Love, "Linux Kernel Development, Third edition," 2005

[17] Kernbench, "http://freecode.com/projects/kernbench," 2009

[18] Ebizzy, "http://sourceforge.net/projects/ebizzy/," 2009

[19] Intel, "Intel 64 and IA-32 Architectures Software Developers Manual, *Software Developers Manual, Intel,* 2010.

[20] K. T. Raghavendra, "Virtual Cpu Scheduling Techniques for Kernel Based Virtual Machine (Kvm)," *Proceeding of the Cloud Computing in Emerging Markets (CCEM)* 2013

[21] O. Sukwong, H. S. Kim, "Is co-scheduling too expensive for smp vms," *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys),* 2011

[22] Credit Scheduler, "http://wiki.xen.org/wiki/Credit Scheduler", consulted on September 2015

[23] The CPU Scheduler in VMware vSphere 5.1, "https://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf,"

[24] J. M. Mellor-Crummey, M. L. Scott "Algoritms for scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems (TOCS),* 1991

[25] S. Wu, Z. Xie, H. Chen, S. Di, X. Zhao, H. Jin "Dynamic Acceleration of Parallel Applications in Cloud Platforms by Adaptive Time-Slice Control", *IEEE International Parallel and Distributed Processing Symposium (IPDPS),* 2016

[26] C. Weng, Q. Liu, L. Yu, and Minglu, "Dynamic Adaptive Scheduling for Virtual Machines," *Proceedings of the 20th international symposium on High performance distributed computing (HPDC),* 2013

[27] A. Menon, J. R. Santos, Y. Turner, "Diagnosing Performance Overheads in the Xen Virtual Machine Environment", *Proceedings of the second ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE),* 2005

[28] Mckenney, PE. Appavoo, J. Kleen, A. Krieger, O. Russel, R. Sarma and Soni, "Read-Copy Update," *In Ottawa Linux Symposium (OLS),* 2002

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

[30] X. Ding, Phillip, B. Gibbons and M. Kozuch, J. Shan, "Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications," *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (ATC),* 2014

[31] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, Alex Ho, R. Neugebauer, I. Pratt, A. Warfield, "Xen and the art of virtualization," *Proceedings of the nineteenth ACM symposium on Operating systems principles Pages (SOSP)* 2003

[32] D. Marsh, L. Scott, J. LeBlanc, P. Markatos, "First-Class User-Level Threads," *Proceedings of the Thirteenth ACM symposium on Operating systems principles Pages (SOSP)* 1991