APPLES: Efficiently Handling Spin-lock Synchronization on Virtualized Platforms

Jianchen Shan, Xiaoning Ding, and Narain Gehani

Abstract—Spin-locks are widely used in software for efficient synchronization. However, they cause serious performance degradation on virtualized platforms, such as the Lock Holder Preemption (LHP) problem and the Lock Waiter Preemption (LWP) problem, due to excessive spinning by virtual CPUs (VCPUs). The excessive spinning occurs when a VCPU waits to acquire a spin-lock. To address the performance degradation, hardware facilities, such as Intel PLE and AMD PF, are provided on processors to preempt VCPUs when they spin excessively. Although these facilities have been predominantly used on mainstream virtualization systems, using them in a manner that achieves the highest performance is still a challenging issue. There are two core problems in using these hardware facilities to reduce excessive spinning. One is to determine the best time to preempt a spinning VCPU (i.e., the selection of spinning thresholds). The other is which VCPU should be scheduled to run after the spinning VCPU is descheduled. Due to the semantic gap between different software layers, the virtual machine monitor (VMM) does not have information about the computation characteristics on VCPUs, which is needed to address the above problems. This makes the problems inherently challenging. We propose a framework named AdPtive Pause-Loop Exiting and Scheduling (APPLES) to address these problems. APPLES monitors the overhead caused by excessive spinning and preempting spinning VCPUs, and periodically adjusts spinning thresholds to reduce the overhead. APPLES also evaluates and schedules "ready" VCPUs in a VM by their potential to reduce the spinning incurred by the spin-lock synchronization. The evaluation is based on the causality and the time of VCPU preemptions. The implementation of APPLES incurs only minimal changes to existing systems (about 100 lines of code in KVM). Experiments show that APPLES can improve performance by 3 $\sim$ 49 percent (14 percent on average) for the workloads with frequent spin-lock operations.

Index Terms—Virtualization, multi-core, cloud computing, spin-lock synchronization, lock holder preemption, scheduling

1 Introduction

In the cloud, the number of virtual CPUs (VCPUs) in a virtual machine keeps increasing to effectively leverage the computing power of multicore processors on the host computer. For example, most VM instances in Amazon EC2 have multiple VCPUs; and a m4.10xlarge instance can have as many as 40 VCPUs [1]. One challenge in hosting large VMs (VMs with multiple VCPUs) is how to efficiently control the excessive spinning incurred by spin-lock synchronization in VMs. Excessive spinning can significantly degrade application performance and reduce system throughput.

Spin-locks are often used in VMs when waiting is expected to be brief. However, because the virtual machine monitor (VMM) schedules VCPUs to make them share physical CPUs (PCPUs), a VCPU may be preempted by another VCPU; and thus the VCPUs waiting for the preempted VCPU must spin for long time unexpectedly. A widely noticed problem caused by spin-lock synchronization is the Lock-Holder Preemption (LHP) problem [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. The LHP problem is caused when a VCPU holding a spin-lock is preempted and other VCPUs waiting for the lock perform excessive spinning before the lock holder VCPU is rescheduled and releases the lock. Another problem is Lock-Waiter Preemption (LWP) problem, which is caused when ticket spin-lock is used. With a ticket spin-lock, lock waiters queue up when waiting for the lock. Thus, when a VCPU waiting for a ticket spin-lock is preempted, other VCPUs waiting for the same lock after it in the queue will have to spend more time spinning [17]. The LHP and LWP problems can cause significant performance degradation.

A general approach to reducing the excessive VCPU spinning caused by spin-lock synchronization is to use hardware to monitor spinning VCPUs and stop them from spinning excessively. Following this approach, spinning suppression hardware facilities (e.g., Intel Pause-Loop-Exiting (PLE) [18] and AMD Pause Filter (PF) [19]) are designed on processors, and VCPU scheduling is improved in VMMs to utilize these facilities. Specifically, with such hardware facilities, the VMM sets a spinning threshold for the time spent on continuous spinning on each processor. Then, the processor monitors the instructions being executed by the VCPU on it to detect spinning. It interrupts the VCPU and reports to the OS if spinning exceeds the threshold on the VCPU, so that the VMM can preempt the VCPU and schedule another VCPU on the processor.

Though such hardware facilities have been predominantly utilized in most virtualization systems to control VCPU spinning (e.g., Xen [20], KVM [21], VMWare ESX [22], etc.), using these facilities in a manner that achieves the highest performance is still a challenging issue, and receives little attention. Two core problems must be solved. One is how to determine the best time to preempt a spinning VCPU, i.e., *spinning threshold problem*. The other is which VCPU should be scheduled after the spinning VCPU is preempted, i.e., *candidate* VCPU *selection problem*. As the paper will show (in Sections 2 and 4), solutions to these problems have substantial impact on

• *The authors are with Computer Science Department, New Jersey Institute of Technology, Newark, NJ 07041.*
E-mail: {js622, xiaoning.ding, narain.gehani}@njit.edu.

Manuscript received 3 Apr. 2016; revised 28 Oct. 2016; accepted 1 Nov. 2016. Date of publication 4 Nov. 2016; date of current version 14 June 2017.
Recommended for acceptance by X. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2625249

performance. Existing systems use some empirical solutions, leading to suboptimal performance.

Due to the semantic gap between the VMM and the software layers inside VMs, the VMM lacks adequate information that is required to effectively address these two problems. This makes these problems difficult to tackle. In the spinning threshold problem, it is difficult for the VMM to differentiate whether a VCPU is spinning normally (e.g., waiting for a spin-lock to be released shortly) or is spinning excessively (e.g., waiting for a spin-lock held by a preempted VCPU). In the candidate VCPU selection problem, the VMM does not have the information on the operations inside VMs, and cannot identify which VCPU is blocking the progress of other VCPUs and making them spinning. Thus, it is difficult to schedule VCPUs in a manner to minimize the overhead of synchronizations.

To make effective utilization of the spinning-suppression hardware facilities equipped on processors, the paper proposes a framework named AdaPtive Pause-Loop Exiting and Scheduling (APPLES). The framework has two components. One component is for addressing the spinning threshold problem, named Adaptive Pause-Loop Exiting (APLE). Instead of struggling with identifying whether spinning VCPUs are waiting for preempted VCPUs, APLE measures the overhead caused by wasteful spinning and wasteful VCPU switches. Wasteful spinning is the spinning stopped by processors when spinning thresholds are reached, since it does not contribute to lock acquisition. Wasteful VCPU switches are incurred by preempting spinning VCPUs and rescheduling other VCPUs. The APLE component periodically measures the overhead of these wasteful operations and adjusts spinning thresholds to minimize this overhead.

The second component in APPLES is a heuristic VCPU scheduling (HVS) mechanism. It is to address the candidate VCPU selection problem for VCPU scheduling. HVS evaluates and schedules “ready” VCPUs in a VM based on whether the scheduling of the VCPUs can effectively reduce spinning. For example, scheduling a spin-lock holder VCPU reduces the spinning of the VCPUs waiting for the lock. The evaluation and scheduling is based on two heuristics: casualty and preemption-time heuristics. The causality heuristic uses the reasons why the VCPUs are preempted. It categorizes “ready” VCPUs into two categories: resource-waiter VCPUs, which have been preempted because of the depletion of their time slices and are waiting for CPU resources to resume execution, and lock-waiter VCPUs, which are waiting for a spin-lock and have been preempted because of excessive spinning. It ranks and schedules resource-waiter VCPUs before lock-waiter VCPUs. The preemption time heuristic ranks VCPUs based on the time when the VCPUs are preempted. When a VCPU is preempted, it is timestamped. HVS schedules resource-waiter VCPUs from the ones with later timestamps, and then schedules lock-waiter VCPUs from the ones with smaller timestamps.

The advantages of APPLES are multi-fold. First, it provides a holistic solution to the effective utilization of spinning-suppression hardware facilities on modern processors, with one component APLE to stop current VCPU spinning when it becomes excessive and the other component HVS to reduce potential VCPU spinning in the future. APLE and HVS can work independently or be combined together to achieve higher performance. Second, APPLES, particularly APLE, directly targets end-to-end performance. Reducing the overhead increases efficiency, since it makes more resources available to the operations that directly contribute to the execution of the workloads. Finally, the implementation of APPLES incurs only minimal modification to existing VMM designs. The implementation based on KVM added about 100 lines of new code to four existing source files. This may help APPLES be quickly adopted by existing virtualization systems.

The contributions of our work are as follows. First, the paper systematically reviews and analyzes the problem faced by almost all mainstream virtualization systems-how to utilize hardware facilities to address the LHP and LWP problems efficiently to achieve high performance. Second, the paper proposes an innovative approach, named APPLES, that controls and utilizes the hardware facilities so as to effectively reduce excessive spinning caused by spin-lock synchronization with low overhead. Third, we implemented APPLES in KVM and tested its performance on a 16-core system. We show that, with APPLES, the performance can be improved by up to 49 percent.

2 Background and Motivation

In this section, we first introduce the problems caused by spin-locks in virtual machines. Then, we introduce the hardware facilities in processors for dealing with these problems, and explain how existing virtualization systems utilize these facilities, using Kernel-based Virtual Machine (KVM) and Intel PLE support as examples. We show that these hardware facilities must be better utilized by VMMs to achieve higher performance.[1]

2.1 Problems Caused by Spin-Locks in VMs

Spin-locks are usually used to protect short critical sections. While waiting to acquire a spin-lock, a thread repeatedly checks the availability of the lock, because the waiting is expected to be brief. With spinning, a lock can be acquired as soon as it is released. At the same time, because the thread does not block itself, the costly overhead associated with context switches is avoided.

Ticket spin-lock is a special type of spin-lock that guarantees the order of lock acquisitions to provide fairness and avoid starvation among lock requests. A ticket spin-lock uses a queue to manage the requests for the lock and schedules the requests accordingly. Thus, a lock waiter cannot acquire the lock until the lock waiter before it on the queue releases the lock.

In a virtualized environment, because of the scheduling of VCPUs, a thread running on a VCPU may not be able to continuously make progress as it does on a PCPU. When a VCPU is preempted, the thread running on it also stops. Thus, if a thread is holding a spin-lock and the VCPU is preempted, the spin-lock cannot be released quickly until the VCPU is rescheduled. Thus, other threads waiting for the lock have to spin for unexpected long time. This is the lock holder preemption (LHP) problem. The spinning causes a live-lock situation, where spinning VCPUs hold CPU resources and wait for the lock, and the lock holder VCPU waits for CPU resources to resume execution. If the spinning cannot be stopped promptly, system throughput may be significantly reduced.

1. Although the description in this section and the APPLES design later in the paper are mainly based on Intel PLE, they can be applied directly or with slight modification to the systems with AMD PF or other similar hardware utilities, which detect and stop spinning based on the thresholds set by the VMM.

With ticket spin-lock, the situation is more complex. The live-lock situation may be caused by not only lock-holders but also lock-waiters. When the VCPU of a lock waiter is preempted, all the subsequent lock waiters on the queue have to spin for unexpected long time until the lock waiter is rescheduled, even though the lock itself may be released during the spinning. This is defined as the lock waiter preemption (LWP) problem.

2.2 Hardware Facilities in Processors to Control Excessive VCPU Spinning

Modern processors provide hardware support for virtualization to reduce overhead. On these processors, PLE and other similar facilities are designed to control excessive VCPU spinning. With PLE, a processor first detects spinning VCPUs by examining the instructions executed by the VCPUs. On X86 architecture, spin-lock primitives usually repeatedly call PAUSE instructions to implement spinning. To detect spinning, the processor checks the intervals (in number of CPU cycles) between consecutive PAUSE instructions executed by a VCPU. For a spinning VCPU, the intervals are very short, since the VCPU only checks the condition for stopping spinning between PAUSE instructions. Thus, the processor compares the lengths of the intervals against a pre-set parameter *PLE_gap*. If the lengths do not exceed *PLE_gap*, it determines that the VCPU is spinning. If no PAUSE instruction is executed in an interval of *PLE_gap*, it determines that the spinning stops.

When the spinning is continuing, the processor needs to determine whether the spinning should be stopped. For this purpose, it keeps track of the length of the spinning by counting the number of cycles spent on PAUSE instructions and the intervals between PAUSE instructions. If the length of the spinning exceeds a pre-set spinning threshold *PLE_window*, the processor will trigger a VM_EXIT to stop the spinning and transfer the control to the VMM, so that the VMM can deschedule the spinning VCPU and reschedule another VCPU.

AMD Pause Filter (PF) functions similar to the Intel PLE. It also checks intervals between consecutive PAUSE instructions and considers PAUSE instructions with intervals smaller than *PAUSE Filter Threshold* to be in the same loop. It interrupts and reports to the OS the spin loops exceeding *PAUSE Filter Count* intervals. Both *PAUSE Filter Threshold* and *PAUSE Filter Count* are pre-set by software. For AMD PF, *PAUSE Filter Count* acts as the spinning threshold. Because of the similarity, we pick one—Intel PLE for our APPLES design and experiments. But APPLES applies equally well to the systems with AMD PF.

2.3 The Utilization of the Hardware Facilities in VMM

With PLE, when the two parameters *PLE_gap* and *PLE_window* are set, the processor detects and interrupts spinning VCPUs autonomously. The VMM controls the PLE facility by adjusting these parameters. It is relatively easy to find an adequate value for *PLE_gap* since PAUSE instructions are called much more frequently in spin-locks than in other scenarios. For example, KVM sets *PLE_gap* to 128 cycles by default, which proves to be effective in practice. Thus, the paper does not discuss the adjustment of *PLE_gap* parameter, and focuses only on how to find an adequate value for the spinning threshold *PLE_window*.

Besides adjusting the parameters, the VMM must also handle VM_EXITs caused by PLE facilities. The VMM takes the chances to preempt spinning VCPUs, put them onto the "ready" VCPU list, and schedule other VCPUs. For example, when a spinning VCPU of a VM is preempted, KVM examines the "ready" VCPUs in the same VM. If it can find a "ready" VCPU, which was not preempted due to spinning, KVM schedules the VCPU. For brevity, this case is called "successful yielding", since the spinning CPU "yields" the processor to a VCPU that can make progress. Otherwise, KVM reschedules the VCPU that is just preempted. This case is called "unsuccessful yielding".

The adjustment of spinning thresholds and the selection of VCPUs to schedule in are two key problems that a VMM must solve to make effective utilization of the hardware facilities controlling VCPU spinning. As we will show later using KVM as an example, these two problems are challenging, and many ad-hoc methods have been tested in existing VMMs. However, workloads with frequent spin-lock synchronization still suffer substantial performance degradation on virtualized platforms.

2.3.1 *Methods to Adjust Spinning Thresholds in KVM*

In the past, KVM would use a system-wide spinning threshold and set it to a fixed value selected empirically based on the normal spinning time under some typical workloads. The spinning time is measured when the VCPUs running the workloads are not preempted. Thus, a threshold can be set slightly higher than the normal spinning time, and any spinning longer than this threshold is considered as abnormal, indicating the occurrence of LHP or LWP problems.

A problem with a fixed spinning threshold was noticed. When physical CPUs (PCPUs) are under-subscribed, preempting spinning VCPUs cannot improve the utilization of PCPUs, and thus incurs unnecessary overhead. The overhead can be very high with large VMs. For example, experiments have shown that it takes 369s to boot a 80-VCPU VM with PLE enabled, while it takes only 25s with PLE disabled [23].

To improve the performance when physical CPUs (PCPUs) are not over-subscribed, attempts have been made to dynamically adjust spinning thresholds. The objectives are to increase the spinning threshold when PCPUs are under-subscribed and to restore the threshold when PCPUs are over-subscribed. For example, one of such attempts increases the threshold on "unsuccessful yieldings" and decreases it on "successful yieldings" [24]. The rationale is that "successful yieldings" indicate that there have been some non-spinning VCPUs preempted (i.e., PCPUs are over-subscribed) and "unsuccessful yieldings" indicate that there is not a "ready" VCPU waiting to be scheduled (i.e., PCPUs are under-subscribed).

In the latest design, KVM maintains a spinning threshold for each VCPU. If a VCPU is preempted and switched out because the VCPU runs out of the time slice, KVM determines that the VCPU is sharing a PCPU with other VCPUs, and quickly reduces the threshold of the VCPU to improve the utilization of the PCPU. This is to deal with the situation in which the PCPU is over-subscribed. For the situation in which the PCPU is under-subscribed, increasing the spinning threshold helps improving performance because this reduces the interruption to VCPU execution. Thus, when a VCPU is preempted because it spins and reaches the spinning threshold, KVM gradually increases its spinning threshold [25].

The above methods improve the performance when PCPUs are under-subscribed. But CPU over-subscription is a common practice [26]. These methods cannot appropriately adjust spinning thresholds when PCPUs are over-subscribed. We provide a quantitative illustration of the above problem using a few representative experiments. We select two benchmarks, *ebizzy* and *dbench*, and run them on a 16-core machine. (Please refer to Section 4 for benchmark description and machine configuration.) We use two 16-VCPU VMs. For each benchmark, we run two instances of the benchmark in parallel on the two VMs, one on each VM, and collect the performance reported by the benchmark (throughput for *dbench* and execution time for *ebizzy*). We first run the benchmarks using the default KVM setting with the mechanism adjusting spinning thresholds enabled. We use this configuration as baseline. Then, we disable the mechanism. We use the same spinning threshold for the VCPUs in the two VMs and vary the spinning threshold from 512 cycles to 32,768 cycles. We repeat the experiments for each of the different threshold values, and show the normalized performance relative to that with the baseline configuration in Fig. 1.

The figure clearly shows that the performance of the two benchmarks changes with the threshold. The performance of benchmark *dbench* varies from 0.98 to 1.17, and the performance of *ebizzy* varies from 0.88 to 1.13. At the same time, different benchmarks achieve the best performance with different spinning thresholds (4,096 cycles for *dbench* and 16,384 cycles for *ebizzy*). The experiments show that, to achieve optimal performance, spinning thresholds must be carefully tuned based on workloads. However, the current KVM system cannot adjust the thresholds adequately, leading to suboptimal performance.

2.3.2 Candidate VCPU Selection in KVM

When a VCPU is preempted because the spinning threshold is reached, the VMM must select a VCPU and schedule it on the vacated computing core. In KVM, a directed yield approach is used [27]. All the VCPUs in the same VM form a circle. The KVM searches the VCPUs other than the VCPU that is just preempted, following the circle. During the search, only "ready" VCPUs are considered, which include two types of VCPUs—the VCPUs preempted due to the depletion of time slices and the VCPUs preempted due to excessive spinning. For brevity, we call the first type of VCPUs *resource-waiter* VCPUs and the second type of VCPUs *lock-waiter* VCPUs.[2]

Resource-waiter VCPUs are more likely to make progress than the lock-waiter VCPUs after rescheduled (i.e., granted with resources). In KVM, resource-waiter VCPUs are more preferred than lock-waiter VCPUs. When a resource-waiter VCPU is found, it is selected to run unconditionally. But, when a lock-waiter VCPU is found, it is selected to run if it

2. It is possible that a VCPU depletes its time slice while it is spinning and waiting for a spin-lock. In such a case, the VCPU is "misclassified" as a resource-waiter VCPU. However, the possibility of such "misclassification" is slim, because spinning is capped by the spinning threshold and is very brief (usually shorter than 10 microseconds), and a time slice is much longer (at least a few milliseconds). At the same time, with existing hardware support, the VMM is not aware of VCPU spinning until it reaches the spinning threshold. Thus, it is not able to put spinning VCPUs into the lock-waiter category if they are preempted before they reach the spinning threshold.

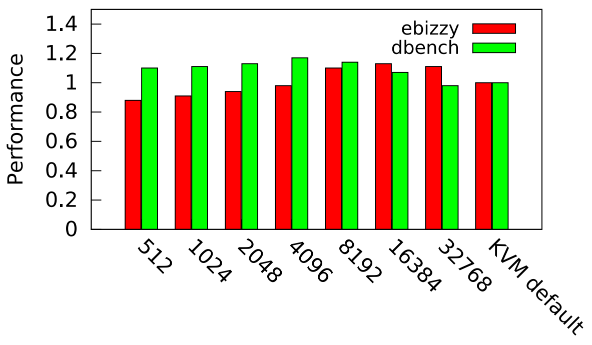


Fig. 1. The normalized performance of *ebizzy* and *dbench* when the spinning threshold is varied from 512 cycles to 32,768 cycles, relative to the performance with the *default* KVM configuration.

is labeled as "checked"; otherwise, it is labeled as "checked" to be selected next time. The "checked" label is removed when the VCPU is scheduled.

When a VCPU is selected, its location in the circle is marked. Later, when more spinning VCPUs are preempted, new searches will start from this location. Thus, consecutive searches will traverse the circle and schedule resource-waiter VCPUs in the first round. Then, in the second round, lock-waiter VCPUs are also considered, because the mechanism assumes that the preempted lock holders have been scheduled in the first round and the lock-waiter VCPUs can continue to make progress when scheduled. If there are not VCPUs ready to run, KVM reschedules the VCPU that is just preempted (i.e., "unsuccessful yielding").

The main problem of the method is with the quality of the candidate VCPUs selected by the method. First, it checks the VCPUs in a VM based on the order in which they are organized in the circle, instead of the possibility of the VCPUs being the causes of excessive spinning. Excessive spinning is usually caused by waiting for preempted VCPUs, which are either holding spin-locks or waiting in ticket spin-lock queues before other VCPUs. These preempted VCPUs should be rescheduled before the VCPUs waiting for them and other VCPUs making new requests for the same spin-lock, so as to avoid additional spinning. Thus, quickly rescheduling these VCPUs is the most effective method to prevent excessive spinning.

The current method in KVM cannot select VCPUs that are more likely to reduce excessive spinning. Even worse, though the method gives a slightly higher priority to resource-waiter VCPUs, there is still a high probability that lock-waiter VCPUs are selected and they continue spinning after being rescheduled. This further decreases the quality of the candidate VCPUs selected. This problem is caused because there may be concurrent searches from the same location on the VCPU circle of a VM—a search starting earlier labels lock-waiter VCPUs as "checked" and another search starting later selects a "checked" VCPUs as a candidate VCPU before the earlier search finds and reschedules a resource-waiter VCPU.

This problem is as shown in Fig. 2. In a VM with 8 VCPUs, VCPU #0 and VCPU #1 run on two different PCPUs. VCPU #0 is preempted and then VCPU #1 is preempted before a VCPU is selected to replace VCPU #0. Thus, both the PCPUs (i.e., the ones running and then preempting these two VCPUs) start searching from the same location (marked as "start point" in the figure). The PCPU preempting VCPU #0 starts earlier than the PCPU preempting VCPU #1. It checks

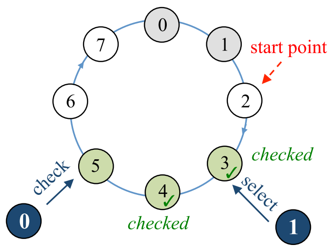


Fig. 2. Candidate VCPU selection in KVM.

VCPU #3 and VCPU #4, which are lock-waiter VCPUs, and labels them as "checked". Thus, the PCPU preempting VCPU #1 can select VCPU #3 as candidate VCPU and schedule it. With the low quality of VCPU candidates, potential VCPU spinning cannot be effectively reduced.

3 APPLES Design and Implementation

As we have introduced earlier, to utilize the spinning suppression hardware facilities equipped on current processors, there are two problems that must be solved: 1) the VMM must carefully adjust spinning thresholds for VMs; and 2) when the hardware facilities preempt a spinning VCPU, the VMM must select an appropriate VCPU to occupy the vacated PCPU. To address these problems, APPLES uses two components: Adaptive Pause-Loop Exiting (APLE) to dynamically adjust spinning threshold; and Heuristic VCPU Selection (HVS) to select candidate VCPUs when spinning VCPUs are preempted.

In this section, we introduce each component by first analyzing the problems and challenges and then describing its design. After that, we introduce the implementation of APPLES based on KVM and Linux.

3.1 APLE for Adjusting Spinning Thresholds

The adjustment of spinning threshold must make a difficult trade-off between different types costs and benefits, which makes it a challenging problem. On one hand, setting high thresholds increases excessive spinning and leads to low resource utilization. On the other hand, setting low thresholds may interrupt normal spinning prematurely. Spinlocks are used to protect short critical sections. Spinning ensures that a lock can be acquired as soon as it is released. At the same time, since spinning is expected to be brief, it incurs lower overhead than blocking, which is considered to be expensive because of the high cost of the context switches associated with blocking operations. Interrupting normal spinning increases synchronization overhead since it actually turns spin-based synchronizations into block-based synchronizations. If spinning thresholds are set too low, the VCPUs that are spinning normally may be preempted prematurely just before the lock holder is about to release the lock, incurring costly context switches between VCPUs. This can significantly increase synchronization overhead and reduce system throughput.

3.1.1 Possible Approaches and APLE Basic Idea

When setting spinning thresholds, the VMM struggles between two conflicting objectives. One is to stop VCPU spinning as early as possible in case spinning VCPUs are waiting for other VCPUs temporarily preempted. The other is to avoid stopping VCPU spinning too early for efficient synchronization in case suspended VCPUs are not blocking spinning VCPUs from making progress.

An intuitive approach for adjusting spinning thresholds is to first determine the amount of time that a VCPU usually spends on spinning when the lock holding VCPU is not preempted and then set the thresholds slightly higher than this amount. However, due to the semantic gap between system layers, it would be "mission impossible" to estimate the amount online. There are several reasons. First, the VMM does not have information about lock operations in virtual machines. Thus, it is not possible for the VMM to *predict* the amount of spinning time (e.g., by profiling and modeling the execution of the workloads). Second, the VMM is not aware of VCPU spinning until it is notified when a processor stops the spinning exceeding the threshold. Thus, it is not possible for the VMM to determine adequate thresholds by *measuring* the amount of spinning.[3] Finally, when a processor stops a spinning VCPU, though the VMM knows the amount of spinning, it cannot determine whether the VCPU is waiting for a preempted VCPU or not. Thus, it still cannot *estimate* the amount of spinning when the LHP or LWP problem does not happen.

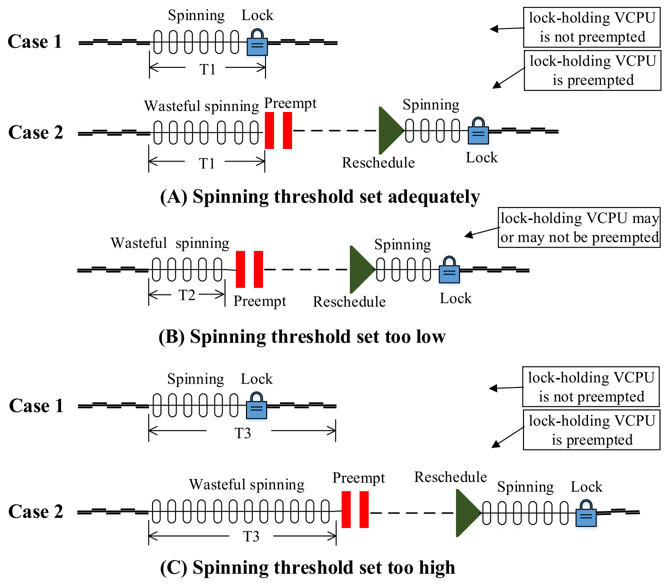
APLE is based on the following observations. If spinning thresholds are set too low, some overhead is caused because the time spent on spinning is wasted and extra time is used on descheduling spinning VCPUs and rescheduling other VCPUs. The overhead decreases if the thresholds are increased. If spinning thresholds are set too high, spinning VCPUs are preempted late. Overhead is caused by excessive spinning and descheduling and rescheduling VCPUs. The overhead decreases if smaller thresholds are used. Thus, optimal thresholds can be approached by varying the thresholds and choosing those leading to lower overhead.

APLE assumes that each workload runs in a VM and assigns a spinning threshold to each VM. It does not use a system-wide spinning threshold for all the VMs on the same physical machine, because different workloads have different locking behaviors and different spinning time before getting a lock. A threshold that achieves optimal performance for some workloads may cause serious performance degradation for other workloads. It does not use a per-VCPU spinning threshold because VCPUs sharing the same lock have similar locking behavior, e.g., every VCPU spins for longer time for longer critical section to finish. APLE also dynamically adjusts spinning thresholds to respond to workload changes in VMs.

3.1.2 Wasteful Spinning and Wasteful VCPU Switches

We use the LHP problem as an example to explain the rationale behind APLE. In Fig. 3, we compare the executions of a VCPU under three different scenarios: (a) when the spinning threshold is adequately set (Fig. 3A; (b) when the spinning threshold is set too low (Fig. 3B); and (c) when the spinning threshold is set too high (Fig. 3C). In the middle of the execution, the VCPU requests a spin-lock that is currently held by another VCPU (not shown in the figure). Thus, it spins before it enters the critical section. However, the spinning incurs different overhead depending on the spinning threshold and whether the lock-holding VCPU has been preempted or not.

3. The spinning time may be measured with the collaboration from guest OSs [28], which is not available on public cloud.



Case 1
Spinning
Lock
T1
lock-holding VCPU is not preempted
lock-holding VCPU is preempted
Case 2
Wasteful spinning
Preempt
Spinning
T1
Reschedule
Lock
(A) Spinning threshold set adequately
lock-holding VCPU may or may not be preempted
Wasteful spinning
Spinning
T2
Preempt
Reschedule
Lock
(B) Spinning threshold set too low
Case 1
Spinning
Lock
T3
lock-holding VCPU is not preempted
lock-holding VCPU is preempted
Case 2
Wasteful spinning
Preempt
Reschedule
Spinning
Lock
T3
(C) Spinning threshold set too high

Fig. 3. The overhead from wasteful spinning and wasteful VCPU switches under three scenarios, using the LHP problem as an example. The figure only shows the VCPU requesting a spin-lock. The lock-holding VCPU is not shown in the figure, but its status is shown in the boxes. A "pause" symbol (parallel vertical bars) indicates that the corresponding VCPU is preempted.

As illustrated in Fig. 3A, with the spinning threshold adequately set ($T1$), if the lock holding VCPU is not preempted, the spinning will not be interrupted before the lock is acquired. The spinning is considered *normal spinning*. In this case, the execution is exactly the same as that on a physical machine, and there is no overhead incurred. However, if the lock holding VCPU is preempted, the spinning will be stopped when it reaches the threshold, and the spinning VCPU is preempted. When the VCPU is rescheduled later, it still needs to spin and wait for the release of the lock. Since the spinning before the VCPU is preempted does not lead to a lock acquisition, it is considered *wasteful spinning*. Compared to the execution on a physical machine, the execution on the virtual machine incurs additional overhead due to the VCPU switch (i.e., descheduling the spinning VCPU and rescheduling another VCPU). Thus, the VCPU switch is a *wasteful VCPU switch*.

As illustrated in Fig. 3B, if the spinning threshold is set too low ($T2$), the VCPU may be stopped prematurely, even when the lock holding VCPU is not preempted. This incurs the overhead through wasteful spinning and wasteful VCPU switches. Compared to the scenario shown in Fig. 3A (the spinning threshold is adequately set), setting the threshold too low increases the chance that the spinning VCPU is preempted. If the spinning VCPU is preempted, next time when it is scheduled, the lock may still not be available, though the lock-holder may have changed. Thus, the VCPU must start over to wait for the release of the lock. With a low threshold, it may be preempted prematurely again. It is possible that the VCPU is descheduled and rescheduled multiple times before it gets the lock, incurring more wasteful spinning and VCPU switches.

If the spinning threshold is set too high ($T3$), as shown in Fig. 3C, the execution is similar to that in scenario (A), when the lock-holding VCPU is not preempted. But, if the lock-holding VCPU is preempted in the case when the spinning threshold is set higher than that in scenario (A), the VCPU spins for longer time before its is preempted. Compared to scenario (A), the spinning incurs higher overhead from wasteful spinning.

Among these three scenarios, no matter whether the threshold is set too low or too high, higher overhead will be caused, compared to an adequately set threshold. Therefore, the overhead can be a reliable indicator of the level of the threshold.

Both wasteful spinning and wasteful VCPU switches are visible to and handled by the VMM. Thus, their overhead can be accurately measured in the VMM with low cost. This is one of the advantages of APLE. Specifically, the overhead of wasteful spinning can be determined by spinning thresholds and the number of times the thresholds reached. The overhead of each VCPU switch is the time between the corresponding VM_EXIT and VM_ENTRY events.

3.1.3 The Calculation of Inefficiency as a Metric

To adjust the threshold, APLE measures the overhead caused by wasteful spinning and wasteful VCPU switches for each VM. However, the amount of overhead cannot be directly used in the adjustment, because the overhead is affected by the factors other than the spinning threshold. For example, the resources allocated to a VM change over time on a over-committed system. With more resources (e.g., more PCPUs) allocated to a VM, the workload on it makes faster progress and incurs higher overhead at the same time.

APLE calculates *inefficiency*, which is the ratio between the time spent on wasteful spinning and wasteful VCPU switches and the PCPU time consumed by the VCPUs. APLE calculates inefficiency periodically and uses it as the metric for the adjustment. Each time period is called an *epoch*. In each epoch, APLE collects the CPU time allocated to the VM. It also maintains a counter counting PLE events, which it resets at the beginning of each epoch. Each time spinning reaches the threshold, in the VM_EXIT event handler (for PLE events), APLE increments the counter, and timestamps the beginning and end of PLE event handling. At the end of each epoch, APLE calculates the overhead of wasteful spinning by multiplying the spinning threshold with the value in the counter, and calculates the overhead of VCPU switches by adding the time spent by PLE event handling. Then, it divides the sum of the two types of overhead by the total CPU time allocated to the VM, the result being the inefficiency of the VM in the epoch.

3.1.4 APLE Algorithm

To achieve the best performance, with APLE, the VMM periodically measures the inefficiency, and adjusts the spinning threshold to minimize the inefficiency using the APLE Algorithm below.

When a VM is launched, this algorithm sets an initial value of the desired threshold T_d (e.g., 8,192 in our experiments). While running, the VM tries the desired threshold and the thresholds slightly lower and slightly higher than the desired threshold, once for an epoch. For fast adjustment, the difference between these thresholds δ cannot be too small. However, to keep the threshold close to the optimal value, δ cannot be too large either. Based on our experiments, a value between 512 and 2,014 works best for the adjustment. At the end of each epoch, APLE calculates the inefficiency of the epoch. When these epochs with different thresholds finish, APLE compares the inefficiency of these epochs. It uses the threshold of the epoch with the smallest inefficiency to update the the desired threshold. Then, the desired threshold is used for the next round of adjustment.

Algorithm 1. APLE Algorithm

```
$T_d$: desired spinning threshold of a VM
$T_0$: initial spinning threshold of the VM
$T_u$: upper bound for the spinning threshold of the VM
$T_l$: lower bound for the spinning threshold of the VM
$T_d \leftarrow T_0$
while the VM is running do
    set the spinning threshold of the VM to $T_d$
    wait for the finish of an epoch $E_1$, and calculate the
    inefficiency of the VM in $E_1$
    set the spinning threshold of the VM to $min(T_u, T_d + \delta)$
    wait for the finish of an epoch $E_2$, and calculate the
    inefficiency of the VM in $E_2$
    set the spinning threshold of the VM to $max(T_l, T_d - \delta)$
    wait for the finish of an epoch $E_3$, and calculate the
    inefficiency of the VM in $E_3$
    compare the inefficiency of epochs $E_1$, $E_2$, and $E_3$
    $T_d \leftarrow$ the spinning threshold of the epoch with smallest
    inefficiency
```

Epoch lengths vary dynamically based on the frequency at which VCPUs are preempted due to excessive spinning (i.e., the frequency of VM_EXITs incurred by PLE events on Intel platforms). Specifically, each epoch corresponds to a fixed number of spinning VCPU preemptions. For example, in our experiments, an epoch corresponds to 1,000 preemptions of spinning VCPUs. Actual epoch lengths vary for different workloads. When the VM rarely uses spin-locks or the server is under-subscribed, spinning VCPU preemptions are rare, and thus epochs are long time intervals; when the VM is spinlock-intensive and is competing for CPU resources with other VMs, spinning VCPU preemptions are frequent, and thus epochs are short time intervals. With short epochs, APLE can quickly respond to execution phase changes. With long epochs, APLE can minimize runtime overhead. At the same time, this way of setting epoch lengths also guarantees that there are enough sample events in each epoch so that the *inefficiency* can be reliably calculated.

3.2 Heuristic VCPU Scheduling

The selection of candidate VCPUs has direct impact on performance. Excessive spinning is usually caused by waiting for preempted VCPUs. As explained in Section 2, for the best performance, these VCPUs should be rescheduled as quickly as possible to avoid additional spinning on the VCPUs that are currently waiting for them or may wait for them in the future before they are rescheduled. Specifically, if excessive spinning is caused by the LHP problem, the VCPU holding the spin-lock should be selected and rescheduled first; if excessive spinning is caused by the LWP problem, the VCPU waiting at the beginning of the ticket-lock queue should be rescheduled first. However, due to the semantic gap between the VMM and VMs, the VMM does not have information to diagnose the root causes of the excessive spinning or distinguish such VCPUs from other preempted VCPUs. This make VCPU selection a challenging problem.

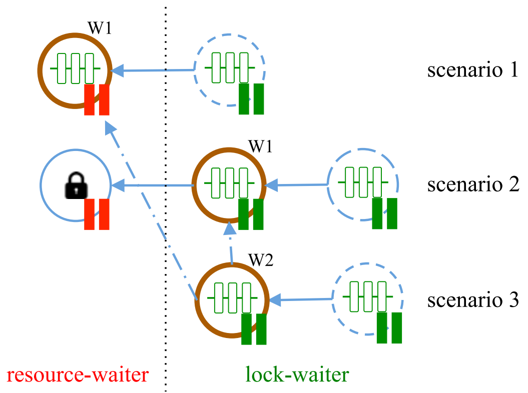


Fig. 4. Three different scenarios of the LWP problem. The preempted ticket-lock waiter in each scenario is illustrated using a solid circle in thick line. A "pause" symbol in red color indicates that the corresponding VCPU is preempted due to the depletion of its time slice, and a "pause" symbol in green color indicates that the corresponding VCPU is preempted due to excessive spinning.

The paper proposes a Heuristic-based VCPU Scheduling algorithm to address candidate VCPU selection problem. The HVS algorithm assumes that the spinning thresholds have been appropriately set. Thus, spinning VCPUs will not be preempted prematurely. While HVS can be implemented to work independently, it achieves better performance when utilized together with APLE, as we will show in Section 4.

The basic idea behind HVS is to evaluate and rank VCPUs based on the possibility and effectiveness to reduce spinning if they are rescheduled immediately. Similar to the methods in KVM, we first categorize "ready" VCPUs into two categories. *Resource-waiter VCPUs* are those preempted because of the depletion of their time slices and are waiting for CPU resources to resume execution; and *lock-waiter VCPUs* are those waiting for a spin-lock and preempted because of excessive spinning. A natural reason for such categorization is that resource-waiter VCPUs are ready to make progress and rescheduling them before lock-waiter VCPUs causes less spinning. A more important reason is that HVS needs to rank and schedule VCPUs in these two categories in different ways, as we will explain below.

HVS ranks the VCPUs in the same VM based on two heuristics. One is the *causality heuristic*, which schedules resource-waiter VCPUs before lock-waiter VCPUs. The rationale of the heuristic is that, when there are VCPUs preempted due to excessive spinning, they are directly or indirectly waiting for other VCPUs that have been preempted due to the depletion of time slices (i.e., resource-waiting VCPUs).

In a LHP problem, a spinning VCPU is preempted when it is waiting for the preempted lock holder, which can be found in the resource-waiter category. The cases with spin-lock holders spinning in critical sections are rare.

When the spinning-suppression hardware facilities are used, the LWP problem becomes more complex. As shown in Fig. 4, in a LWP problem, a ticket-lock waiter may be preempted in a few scenarios. First, a ticket-lock waiter is preempted because it ran out of its time slice. In this case, the ticket-lock waiter can be found in the resource-waiter category. Second, a ticket-lock holder has been preempted, and thus the ticket-lock waiter spins before it is preempted due to excessive spinning. In this case, the ticket-lock holder must be scheduled first, which is in the resource-waiter category. The ticket-lock waiter is in the lock-waiter category.

Third, a ticket-lock waiter $W2$ is preempted because another ticket-lock waiter $W1$ located before it in the queue has been preempted. In this case, $W1$ should be scheduled before $W2$. $W1$ is in the resource-waiter category (as in the first scenario), or is in the lock-waiter category waiting for another VCPU in the resource-waiter category (as that in the second scenario). $W2$ is in the lock-waiter category.

Based on the analysis above, no matter whether the excessive spinning problem is caused by preempted lock holder or preempted ticket-lock waiter, a VCPU in the resource-waiter category should be scheduled before the VCPUs waiting for it in the lock-waiter category are scheduled. However, due to the semantic gap between the VMM and VMs, the VMM cannot identify which resource-waiter VCPUs are blocking other VCPUs from making progress. Thus, a safe choice is to schedule all the resource-waiter VCPUs before scheduling the lock-waiter VCPUs.

The other heuristic, *preemption-time heuristic*, is used to rank the VCPUs in each category. When a VCPU is preempted, it is time-stamped. The timestamps keep increasing. HVS ranks resource-waiter VCPUs with larger preemption timestamps before the ones with smaller timestamps; and ranks lock-waiter VCPUs with smaller preemption timestamps before the ones with larger timestamps. This heuristic is based on the following observations.

When a VCPU (A) is preempted due to excessive spinning and resource-waiter VCPUs are examined, the resource-waiter VCPU (B) causing A to spin is more likely to be the one that is preempted recently. Critical sections and normal spinning in spin-lock synchronizations are much shorter than time slices. They are usually shorter than a few microseconds, while time slices are longer than a few milliseconds. Thus, the chance that spin-lock holders or spin-lock waiters are preempted due to depleted time slices is small if spin-locks are not frequently requested; and LHP and LWP problems are usually incurred by the workloads with frequent spin-lock synchronizations; for example, each VCPU many request a spin-lock multiple times in a time slice. Therefore, when A is preempted, B must have been preempted recently, later than the time when last time A requests the lock.

When all the "ready" VCPUs are lock-waiter VCPUs and there is still a VCPU being preempted due to spinning, the VCPU must be waiting for another VCPU, which shares the same ticket-lock with it and has been preempted earlier due to spinning. This corresponds to the third scenario in the LWP. Because all the VCPUs in the same VM use the same spinning threshold, the order in which the VCPUs are preempted by the hardware facilities reflects the order in which they request the ticket lock, which in turn determines their positions in the queue. Thus, these VCPUs should be scheduled in the same order as they are preempted.

Based on these two heuristics, the HVS maintains two lists, resource-waiter list and lock-waiter list, to organize resource-waiter VCPUs and lock-waiter VCPUs, respectively. HVS ranks the VCPUs on each list based on their preemption timestamps and ranks the resource-waiter VCPUs higher than lock-waiter VCPUs. When a VCPU needs to be selected, it just selects the VCPU with the highest rank.[4]

4. Though preferentially scheduling spinning VCPUs with larger preemption timestamps degrades performance (as shown in Fig. 13), the "misclassification" of spinning VCPUs as resource-waiter VCPUs hardly hurts performance, due to its low possibility of happening.

3.3 APPLES Implementation

We have implemented APPLES based on KVM and Linux. The implementation of APLE in KVM adds only about 80 lines of source code to 4 existing files, and the implementation of HVS adds about 30 lines of source code to one existing file. Most changes are made inside the PLE event handler of KVM. Other changes are mainly to collect event times and other needed information (e.g., the preemption time of VCPUs, the number of times that the VCPUs have been preempted due to excessive spinning in each epoch, etc).

Every time when the spinning-suppression hardware detects excessive VCPU spinning, the PLE event handler is called to handle this issue. Inside the handler, APPLES first uses HVS to select a candidate VCPU. Then, it checks whether an epoch is finished or not. If an epoch is finished, it adjusts the spinning threshold and changes the Virtual Machine Control Structure (VMCS) of the VCPU accordingly, before it schedules in the VCPU.

One issue we addressed in the implementation is to adapt HVS to the method currently used in KVM to reschedule VCPU candidates. Linux and KVM uses virtual run time to schedule VCPUs. When a VCPU runs, its virtual run time increases monotonically. When the virtual run time exceeds any other VCPU's virtual run time by a time quantum (usually very small), the VCPU is preempted. In KVM, when a spinning VCPU (A) is preempted and another VCPU (B) is selected, it uses a "yield_to" mechanism to temporarily boost the priority of B, such that B can be rescheduled as soon as possible. The virtual run time of B still keeps increasing. In HVS, the latest preempted VCPU is selected first. Since the latest preempted VCPU already has a large virtual run time (larger than that of any other VCPUs when it is preempted). Thus, it may be preempted again shortly after it is rescheduled. Then, it may be selected again by HVS when another spinning VCPU is preempted, though it is not blocking the progress of other VCPUs. This forms a loop preventing HVS from selecting VCPUs that can effectively reduce spinning. In the loop, a VCPU is selected by HVS repeatedly as a candidate VCPU. Since its virtual run time is large and keeps increasing, it is preempted shortly every time when it is scheduled, giving it a higher probability of being selected again by HVS. This not only lowers the quality of the candidate VCPUs selected by HVS, but also reduces the chances of other VCPUs getting rescheduled, and may cause starvation problem in the worst case.

To address this issue, the implementation in KVM prevents a VCPU from being selected as a candidate VCPU repeatedly. For this purpose, the implementation marks a VCPU as "yielded" when it is selected as a candidate VCPU. When a VCPU is preempted because its virtual run time is too large, its "yielded" mark is checked. If there is not a "yielded" mark, the VCPU is put onto the resource-waiter list. Otherwise, the mark is removed, and the VCPU is put onto a "yielded" VCPU list. The VCPUs on the "yielded" list are selected by HVS as candidate VCPUs when the resource-waiter list and lock-waiter list are empty. They may also be selected to run when their virtual run time is surpassed by that of other VCPUs.

4 Evaluation

This section evaluates APPLES with a collection of multithreaded benchmarks. We first present the overall performance of APPLES. Then, for each component in APPLES, we carry out experiments to show its performance advantage and study in detail how it improves performance.

4.1 Experimental Setup

We conducted our experiments on a Dell PowerEdge R720 server with 64 GB of DRAM and two 2.40 GHz Intel Xeon E5-2665 processors. Each processor has 8 cores. There are 16 cores in total. On the server, we created 4 VMs with 16 VCPUs. Each VM has 16 GB of memory. The VMM is KVM [21]. The host OS and the guest OS are Ubuntu version 14.04 with the Linux kernel version updated to 3.19.8. The VCPUs in each VM are one-to-one pinned to physical cores. Our experiments show that the benchmarks achieve better performance under this configuration than they do without pinning the VCPUs. CPU power management can reduce the performance of the applications running in VMs [29]. To prevent such performance degradation, in the experiments, we disabled the C states other than C0 and C1 of the processors, which have long switching latencies.

To evaluate APPLES, we used the benchmarks in PARSEC 3.0 suite [30], including native PARSEC benchmarks and SPLASH2X benchmarks in the suite. We attach a prefix ‘p.’ before the name of each native PARSEC benchmark, and attach a prefix ‘s.’ before the name of each SPLASH2X benchmark, in order to differentiate these two sets of benchmarks. We also refer to native PARSEC benchmarks as PARSEC benmarks for brevity. These benchmarks are mainly for testing multicore processor designs in computer architecture area. Most of them are computation-intensive and require minimal system support. Therefore, we also selected a few other applications that have been frequently used to study LHP and LWP problems. *Ebizzy* [31] is multi-threaded and generates workloads similar to those on common web application servers. *Dbench* [32] is derived from an industry-standard benchmark *NetBench*. It is a utility that tests the ability of a file system to service requests from clients. *Hackbench* [33] is a multi-threaded benchmark designed to test Unix-socket (or pipe) performance. *Kernbench* [34] is a CPU and memory intensive benchmark that measures and compares the time used to compile Linux kernels. We selected these applications not only because they incur frequent system operations, but also because they are representative workloads in diverse application domains.

We compiled the PARSEC and SPLASH2X benchmarks using gcc with the default settings of the *gcc-pthreads* configuration in PARSEC 3.0. We built other benchmarks using the make files/scripts coming with the benchmark packages. The gcc compiler and the libraries required by the benchmarks are stock software components in the Ubuntu Linux distribution. We used the *parsecmgmt* tool in the PARSEC package to run the PARSEC and SPLASH2X benchmarks with native input. In the experiments, we set the number of threads in each benchmark equal to 32. We ran each experiment five times and report the average result.

We ran the benchmarks using the default KVM configuration and use their performance as the baseline performance. Since different benchmarks may use different metrics (e.g., throughputs and execution times) and the absolute performance numbers vary widely across benchmarks, we normalize the performance measured in the experiments against the baseline performance. Thus, the baseline performance is always 1. To be consistent, we use large values to represent higher performance. Thus, if a benchmark reports throughput, we present its normalized throughput in the paper. If a benchmark reports execution time, we present its speedup in the paper. For brevity, we use “performance” to refer to both normalized throughput and speedup.

The LHP and LWP problems happen when PCPUs are over-subscribed. Thus, we launch multiple VMs. We run multiple instances of the same benchmark in parallel on the VMs, one on each VM. different configurations, and compare the performance. In our experiments with APLE enabled, for all the VMs, the initial value of the desired threshold T_d is 8,192 cycles. The lower bound T_l is 4,096 cycles (the same as that in the default KVM setting). The upper bound T_u is 32,768 cycles, and δ is 1,024 cycles.

4.2 Overall Performance of APPLES

In this section, we first show the performance advantage of APPLES over the stock KVM for spinlock-intensive benchmarks. Then, we compare the overhead of APPLES and the stock KVM.

For each benchmark, we launch two VMs and run two instances of the benchmark in parallel, one on each VM. We first run the benchmark using the stock KVM with PLE disabled. Then, we enable the PLE support, and run the benchmark with the stock KVM and APPLES, respectively. We also manually set the PLE_window to 512 cycles, collect inefficiency values during its execution, and average the inefficiency values. If the average inefficiency value is greater than 5 percent, the benchmark is considered to be spinlock-intensive. We use spinlock-intensive benchmarks to evaluate the effectiveness of APPLES and non-spinlock-intensive benchmarks to test the overhead of APPLES.

Fig. 5 shows the performance of spinlock-intensive benchmarks and their average performance under three scenarios, i.e., (1) with the PLE support turned off, (2) with the stock KVM (PLE enabled), and (3) with APPLES (PLE enabled). APPLES performs consistently better than the stock KVM for these benchmarks. Compared to the stock KVM, APPLES improves the performance of the benchmarks by 14 percent on average and up to 49 percent.

Among these benchmarks, *ebizzy*, *dbench*, *hackbench*, and *kernbench* incur the most frequent spin-lock operations. Their performance suffers significantly from LHP and LWP problems. Though the PLE support in the stock KVM can significantly improve their performance, APPLES is more effective and can further improve performance. For *p.canneal*, *p.bodytrack*, *p.raytrace*, and *p.streamcluster*, with the stock KVM, enabling PLE even degrades their performance, because the stock KVM cannot set spinning thresholds adequately and preempts spinning VCPUs prematurely. APPLES can avoid this problem. It achieves similar (for *p.raytrace*) or higher (for the other three benchmarks) performance, relative to that with the PLE support turned off. For the remaining benchmarks, the spin-lock operations in their executions are not as frequent as those in the first four benchmarks. With PLE support, the stock KVM improves their performance moderately, and APPLES can improve the performance by larger percentages.

APPLES improves performance through the synergistic collaboration of APLE and HVS, which significantly reduces the total cost incurred by excessive spinning and preempting spinning VCPUs. We use *ebizzy* as an example to illustrate how APPLES with its components reduces the cost

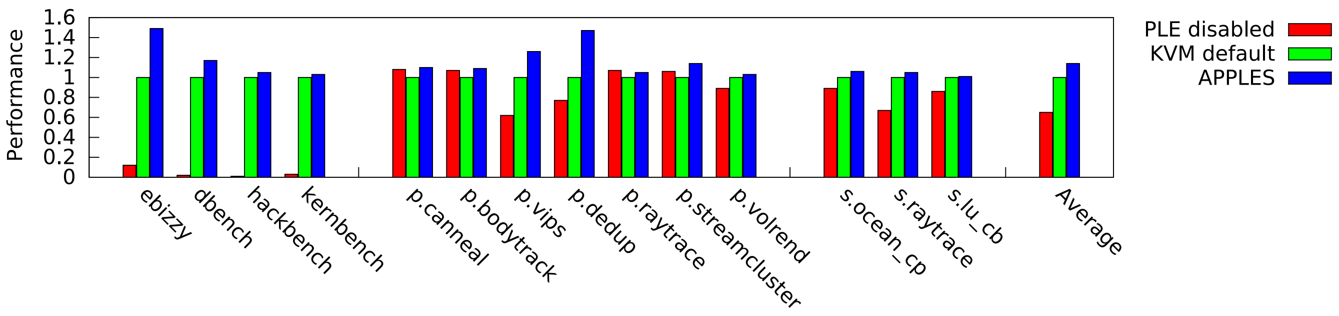


Fig. 5. Normalized performance of the spinlock-intensive benchmarks with *KVM* and *APPLES* (PLE support enabled) and PLE support disabled, when two VMs co-run. Prefixes 'p.' in benchmark names stand for PARSEC benchmarks, and prefixes 's.' stand for SPLASH2X benchmarks.

and how the performance is affected by the reduction of the cost. To test one component of APPLES, we disable the other component and use the default mechanism in KVM.

As shown in Fig. 6, compared to the stock KVM, the performance of *ebizzy* is improved by 34 percent with APLE alone, and is improved by 9 percent with HVS alone. With APLE and HVS combined, the performance can be improved by 49 percent. The percentage of improvement with APPLES is even higher than the sum of percentages of improvement with APLE and HVS alone. This is because HVS is more effective with APLE than it with the default mechanism in KVM to adjust spinning threshold, as we will show later in Section 4.4. The figure also compares the average inefficiency values of *ebizzy* executions under these scenarios. APLE and HVS can reduce inefficiency by 32 and 18 percent respectively, and reduce it by 58 percent when combined, relative to the stock KVM. It is evident that performance is improved when the inefficiency decreases.

We measure the overhead of APPLES with two sets of experiments. The first set studies its overhead on under-subscribed systems. For this purpose, we launch one VM, and run spinlock-intensive benchmarks in the VM. On a system that is under-subscribed, each VCPU gets a dedicated physical core. Thus, lock holder/waiter VCPUs would not be preempted, and there is no need to preempt spinning VCPUs. Descheduling and rescheduling spinning VCPUs degrades performance. Thus, the performance measured with the PLE support disabled represents the best performance these benchmarks can achieve. The performance degradation caused by KVM and APPLES enabling and handling PLE events represents their overhead. On average, the benchmarks achieve similar performance with APPLES and the stock KVM, and the performance difference is not noticeable (less than 2 percent). Compared to the executions with PLE support disabled, these benchmarks show only

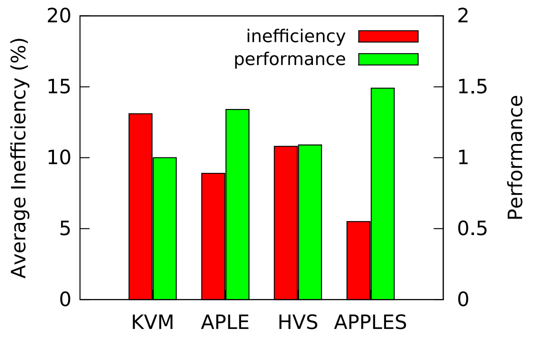


Fig. 6. Normalized performance and average inefficiency of *ebizzy* with *KVM*, *APLE*, *HVS*, and *APPLES* when two VMs co-run.

slightly lower performance with KVM and APPLES (1~2 percent on average and up to 8 percent for *kernbench*). The overhead of APPLES is similar to that of the stock KVM and is acceptable when the system is under-subscribed.

The second set of experiments study the overhead of APPLES on over-subscribed systems. We launch two VMs, on which we run the benchmarks that do not incur frequent spinlock operations. Fig. 7 shows the performance of these benchmarks and their average performance. We use performance tested with the stock KVM with PLE support disabled as baseline performance. Both APPLES and the stock KVM show similar performance as that with PLE support disabled (difference < 1 percent), indicating that their overhead is very low for the benchmarks that rarely incur spinlock operations.

4.3 APLE Performance

To study in detail how APLE improves system performance, we enable APLE and disable HVS in APPLES. We select seven spinlock-intensive applications for the study. We select *ebizzy*, *dbench*, *hackbench*, and *kernbench*, because they are more spinlock-intensive than other benchmarks, and their performance is more sensitive to the management of PLE facility. We select *p.raytrace* and *p.streamcluster*, because we want to investigate the reasons why APPLES can maintain and improve the performance while the stock KVM degrades their performance when PLE support is enabled. Benchmark *p.dedup* is selected because its performance is most sensitive to the management of PLE facility among the remaining benchmarks, which are not as spinlock-intensive as the first four benchmarks.

We carry out experiments to compare APLE against the mechanism which uses a fixed system-wide spinning threshold. Since a benchmark shows different performances with different spinning thresholds, we repeat experiments and test different spinning thresholds from 512 cycles to 32768 cycles to get a scope of performance variation. Thus, we can find the *"best"* performance and the *"worst"* performance that the benchmark can achieve by selecting different *fixed* spinning thresholds. In this section, we use *"best"* to represent the case in which the selected spinning threshold leads to the best performance, and use *"worst"* to represent the case in which the selected spinning threshold leads to the worst performance.

Please note that the "best" and "worst" performances are only those achieved with fixed spinning thresholds. They do not represent the real best and worst performance that can be achieved with any possible methods. However, we use the "best" performance and the "worst" performance to show the potential of adjusting the spinning threshold and

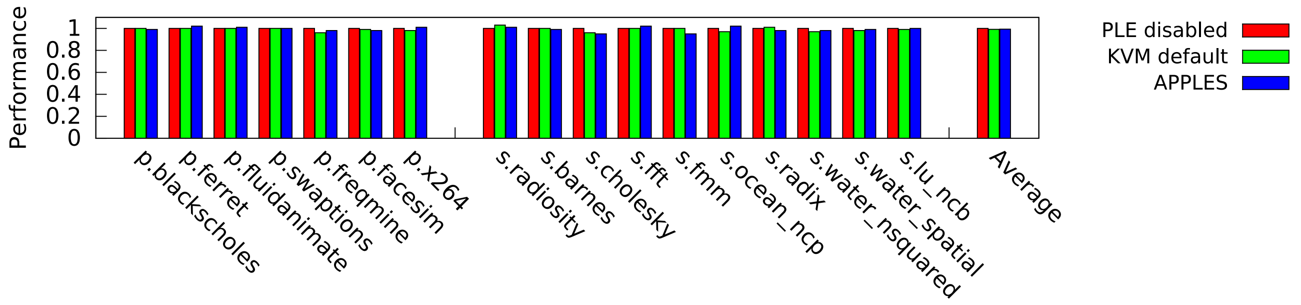


Fig. 7. Normalized performance of the non-spinlock-intensive benchmarks with *KVM* and *APPLES* (PLE support enabled) and PLE support disabled, when two VMs co-run. Prefixes ‘p.’ in benchmark names stand for PARSEC benchmarks, and prefixes ‘s.’ stand for SPLASH2X benchmarks.

how much performance degradation could be caused if the spinning threshold was not adequately set.

We also want to compare the “best” and “worst” performances against the performance that can be achieved with the dynamic method in APLE, and show the necessity for adjusting the spinning threshold dynamically based on workloads. During the execution of a benchmark, there may be different phases. A threshold leading to good performance in one phase may lead to bad performance in another phase. Thus, it is possible that, with a spinning threshold adjusted dynamically, a benchmark achieves better/worse performance than the “best”/“worst” performance achieved with a fixed threshold used across different phases.

Fig. 8 shows the performance of these benchmarks when two VMs co-run. The stock KVM cannot achieve the best performance. Especially, with *p.streamcluster*, *kernbench* and *p.raytrace*, it even achieves lower performance than the “worst” performance obtained with a fixed spinning threshold level. In contrast, APLE can achieve better performance than *“best”*—the best performance that can be obtained by smartly selecting a fixed spinning threshold. The average performance achieved with APLE is 1.13, and the average performance achieved by smartly selecting a fixed spinning threshold (i.e., *“best”*) is 1.10. APLE improves the performance of *ebizzy* by the largest percentage (34 percent relative to the stock KVM and 19 percent relative to *“best”*). For *p.raytrace* and *p.streamcluster*, the “best” performance is achieved when the thresholds are high (32,768 cycles). The stock KVM degrades performance because it sets the thresholds too low, such that spinning VCPUs are preempted prematurely. APPLES avoids this problem since premature VCPU preemptions increase wasteful VCPU switches and thus inefficiency. The figure also shows that, when selecting a wrong spinning threshold level, the performance can be degraded by 16 percent on average and up to 46 percent (for *ebizzy*), relative to that with spinning thresholds adequately set by APLE.

Fig. 9 shows the performance of the benchmarks when four VMs co-run. Compared to the executions with two VMs, the performance difference between the stock KVM, *“best”*, and APLE is much smaller. However, if the spinning thresholds are set inadequately, application performance still can be significantly reduced. For example, with *dbench*, the performance difference between *“best”* and *“worst”* is 19 percent when two VMs co-run, and the difference is increased to 35 percent when four VMs co-run.

To illustrate the correlation between system performance and the inefficiency level and to show how adjusting spinning threshold can reduce inefficiency and improve system performance, we use *ebizzy* as an example, and compare the average inefficiency values along with normalized performances achieved by KVM, *“best”*, *“worst”*, and APLE when two VMs co-run. The average inefficiency is the average of the inefficiency values measured in the epochs of the two VMs during the two instances of *ebizzy* run in parallel in the VMs.

As shown in Fig. 10, in general the average inefficiency reduces when the spinning threshold is increased from 512 cycles to 16,384 cycles. This is because the overhead of wasteful VCPU switches caused by preempting spinning VCPUs prematurely can be reduced with larger spinning thresholds. Meanwhile, with the decreasing of the average inefficiency, the performance is improved accordingly. However, when the spinning threshold is further increased, the average inefficiency increases, since the overhead of wasteful spinning starts to dominate, and thus the performance is degraded.

Fig. 10 also clearly shows that, with a fixed spinning threshold, the “best” performance is achieved when the average inefficiency is minimized by smartly selecting the

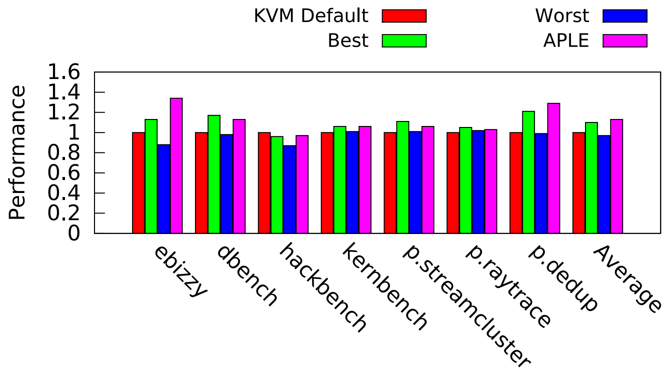


Fig. 8. Normalized performance of the benchmarks with *KVM*, *“best”*, *“worst”*, and APLE when two VMs co-run.

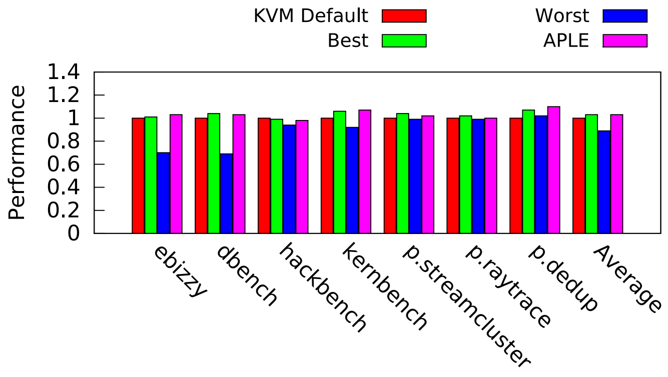


Fig. 9. Normalized performance of the benchmarks with *KVM*, *“best”* and *“worst”*, and *APLE* when four VMs co-run.

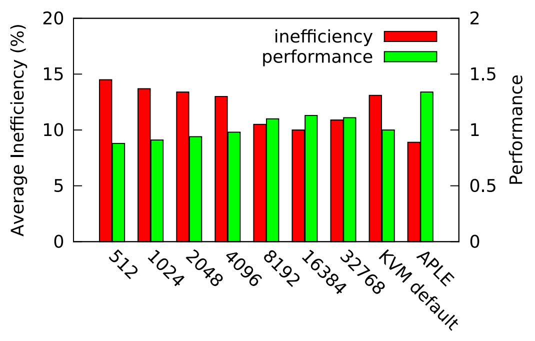


Fig. 10. Normalized performance and average inefficiency of *ebizzy* when a system-wide spinning threshold is changed from 512 cycles to 32,768 cycles, and when the stock *KVM* and APLE is used to adjust the spinning threshold. Two VMs are used.

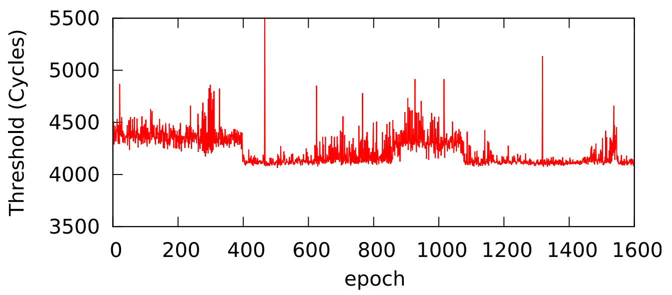


Fig. 11. Spinning threshold adjusted by the stock KVM when two VMs co-run.

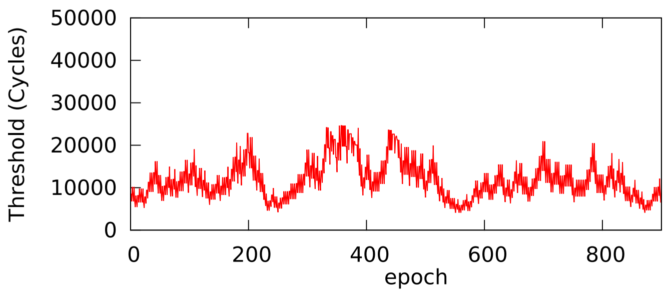


Fig. 12. Spinning threshold adjusted by APLE when two VMs co-run.

spinning threshold (16,384 cycles in this case). The default KVM mechanism cannot achieve the best performance since it cannot effectively reduce inefficiency. In contrast, APLE reduces the average inefficiency by 32 percent, relative to the stock KVM. Moreover, compared to *"best"*, APLE reduces the average inefficiency by 11 percent, which is the reason why APLE can achieve even higher performance than *"best"*.

In the above experiments, we also collected the spinning thresholds during the execution of the *ebizzy* instances.[5] Figs. 11 and 12 show how spinning thresholds are adjusted respectively for the scenarios with default KVM mechanism and APLE. With APLE, there are about 900 epochs in the execution, while with KVM default mechanism there are about 1,600 epochs. This is because fewer VM_EXITs are incurred by PLE events with APLE. With the default KVM mechanism,

5. The default KVM mechanism does not use epochs and sets a spinning threshold for each VCPU. For fair comparison, we define epoch in the same way as in APLE (i.e., 1000 VM_EXITs caused by PLE events), and collect the average spinning threshold of all the VCPUs in a VM for each epoch.

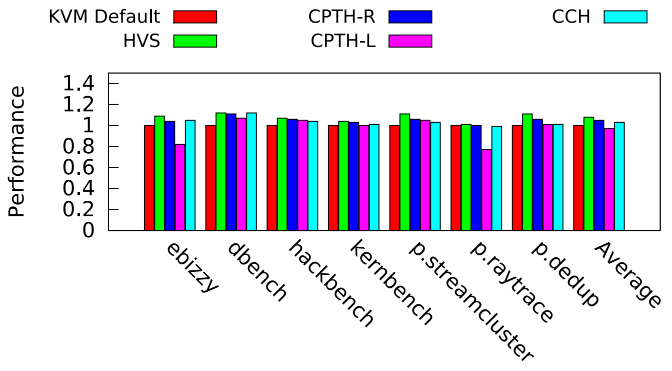


Fig. 13. Normalized performance of the benchmarks with the stock KVM, HVS, and three variants of HVS when two VMs co-run. The default mechanism in KVM is used to adjust spinning thresholds.

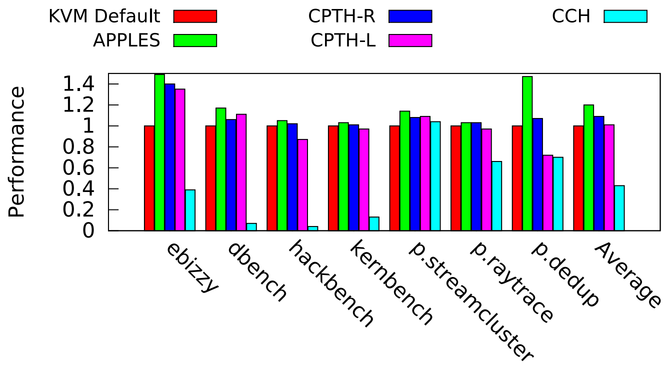


Fig. 14. Normalized performance of the benchmarks with the stock KVM, HVS, and three variants of HVS when two VMs co-run. APLE is used to adjust spinning thresholds for HVS and its variants.

the spinning threshold sticks round 4,200, which leads to the poor performance similar to the one with fixed spinning threshold of 4,096. This shows that the stock KVM cannot effectively adjust the spinning threshold to achieve optimal performance. However, with APLE, the spinning threshold changes steadily around 12,000, which leads to the performance that is even better than "best" performance achieved with fixed spinning threshold of 16,384.

4.4 HVS Performance

In this section, we want to understand how the heuristics in HVS help improve the performance. For this purpose, we have implemented three variants of HVS, which intentionally avoid selecting the candidate VCPUs suggested by a heuristic. The name of the variants and their differences with HVS are:

- Counter Preemption-Time Heuristic on Resource-waiters (CPTH-R): when selecting a candidate VCPU from resource-waiter VCPUs, the one with the smallest preemption timestamp is selected.
- Counter Preemption-Time Heuristic on Lock-waiters (CPTH-L): when selecting a candidate VCPU from lock-waiter VCPUs, the one with the largest preemption timestamp is selected.
- Counter Causality Heuristic (CCH): lock-waiter VCPUs are selected before resource-waiter VCPUs.

We select the same set of benchmarks as we do for testing APLE, and compare the performance of HVS with its variants when we run the benchmarks in two VMs. Figs. 13 and 14 show their performance, relative to the stock KVM. In Fig. 13, the data was obtained with the default mechanism in KVM to adjusting spinning threshold. In Fig. 14, the data was obtained with APLE adjusting spinning thresholds.

As shown in Fig. 13, HVS performs slightly better than its variants when the default mechanism in KVM adjusting spinning threshold. The average performance is 1.08, 1.05, 0.97, and 1.03 for HVS, CPTH-R, CPTH-L, and CCH, respectively. This indicates that the heuristics used in HVS do help improving the performance, but they are not very effective. The figure also shows that the preemption-time heuristic applied to lock-waiter VCPUs helps improving performance by the largest percentage. This is because, for ticket spin-locks, the order in which lock-waiter VCPUs are scheduled has great impact on performance.

We were surprised to see that the causality heuristic is not as effective as the preemption-time heuristic. The benchmark *dbench* even shows the same performance on HVS and CCH. Our investigation shows that the default mechanism in KVM tends to adjust the spinning thresholds to very low values (Figs. 11 and 12), which preempt spinning VCPUs prematurely even when LHP or LWP does not happen. Thus, scheduling resource-waiter VCPUs first and scheduling lock-waiter VCPUs first do not make much difference on performance. Note that lock-waiter VCPUs are preempted when their spinning thresholds are reached. Thus, if spinning thresholds are set too low, lock-waiter VCPUs are preempted even when they may get the locks shortly.

Thus, we repeated the experiments with APLE adjusting spinning thresholds of the VMs. As shown in Fig. 14, the heuristics in HVS become more effective when spinning thresholds are adequately set. Not using these heuristics leads to serious performance degradation. The average performance is 1.20, 1.09, 1.01, and 0.43 for HVS, CPTH-R, CPTH-L, and CCH, respectively. Specifically for the causality heuristic, scheduling lock-waiter VCPUs before resource-waiter VCPUs reduces the performance of *dbench* and *hackbench* by more than 10x. This on one hand shows the importance of VCPU selection and confirms the effectiveness of causality heuristic, and on the other hand demonstrates the effectiveness of APLE on selecting adequate spinning threshold to accurately and promptly identify VCPUs waiting for preempted lock holders or preempted lock waiters.

5 Related Work

A large number of studies have been focused on the lock holder preemption (LHP) and the lock waiter preemption (LWP) problems. Various solutions have been proposed to reduce performance degradation. Software-only solutions include sophisticated VCPU scheduling algorithms [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], improved synchronization primitives [17], and paravirtualization [13], [14]. On current platforms, using spinning-suppression hardware facilities, such as Intel PLE and AMD PF, has been dominantly utilized on mainstream virtualization systems and become a *de facto* standard solution [15], [16], [18], [19]. Our work does not provide an alternative solution to the LHP and LWP problem. Instead, it improves the solutions with hardware facilities.

Targeting the problem of setting spinning thresholds for the hardware facilities, there are studies showing that spinning thresholds must be adjusted based on workloads to achieve best performance [28], [35]. Besides APLE [36], there are some other efforts to adjust spinning thresholds dynamically. Zhang, Dong, and Duan [28] proposed a profiling method that collects the average spin-lock cycles in guest OSs and uses the information to adjust spinning thresholds. This approach requires the VMM to have detailed and important information about guest OSs, such as OS symbol tables, which should not be exposed to the VMM for security reasons on the systems shared by multiple users, e.g., public clouds. This seriously limits the scope of the solution. Thimmappa [24] proposed a method to adjust the spinning threshold based on whether or not the resources freed by preempting spinning VCPUs can be reallocated to other VCPUs for them to make progress. Recently, KVM implemented a method to dynamically grow/shrink the spinning threshold for each VCPU [25]. These two methods focus mainly on improving the performance when the system is under-committed.

Targeting the problem of which VCPUs should be scheduled to replace spinning VCPUs, besides the directed yield method currently used in KVM [27], and another Linux online patch [37], which relies on modified guest OSs to label VCPUs holding spin-locks, we cannot find any research on selecting VCPUs for the efficient utilization of spinning suppression hardware facilities.

The trade-off between busy waiting (spinning) and blocking in synchronization primitives is a classic yet challenging problem, and has been intensively studied under different scenarios [38], [39], [40], [41]. The problem we target in this paper also needs to make a trade-off between busy waiting and blocking. But, compared to the problems targeted in previous studies, the problem in this paper is more challenging, since the VMM has limited information and cannot directly control the spinning in synchronization primitives.

6 Conclusion

Mainstream virtualization systems rely on hardware facilities, such as Intel PLE and AMD PF, to alleviate the performance degradation due to excessive VCPU spinning. However, it is still a challenging issue to effectively control these facilities to minimize overhead and maximize throughput, which requires the knowledge on the locking behaviors of guest systems that is unavailable at the VMM level, due to the semantic gap between the host and the guests. Ineffective utilization of these hardware facilities may even cause performance degradation.

The paper addresses this issue with a holistic solution named APPLES. The two components in it solve two core problems in the utilization of the hardware facilities. Specifically, one component APLE maintains an adequate VCPU spinning threshold for each VM, in order to promptly detect and preempt VCPUs when they spin excessively. The key idea is to measure the execution efficiency of each VM and adjust the threshold in a way to maximize the efficiency. The other component HVS carefully selects VCPUs and schedules them in an order required by efficient synchronization. The key idea is to evaluate and rank VCPUs based on the causality and time of VCPU preemptions.

Our experiments show that APPLES can improve system performance by as much as 49 percent. Its implementation incurs minimal modification to existing virtualization system designs. We seek the adoption of APPLES in commercial and open-source virtualization systems.

Acknowledgments

We thank all the reviewers for their constructive comments and suggestions. This work is supported in part by the US National Science Foundation under grants CNS 1409523 and CCF 1617749.

REFERENCES

[1] (2016). "Amazon EC2 instance types," [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[2] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines," in *Proc. 3rd Conf. Virtual Mach. Res. Technol. Symp.*, 2004, pp. 43–56.

[3] C. Yu, L. Qin, and J. Zhou, "A lock-aware virtual machine scheduling scheme for synchronization performance," *J. Supercomputing*, pp. 1–13, 2015.

[4] Drummonds, "Co-scheduling SMP VMs in VMware ESX server," 2008. [Online]. Available: http://communities.vmware.com/docs/DOC-4960

[5] H. Kim, S. Kim, J. Jeong, and J. Lee, "Demand-based coordinated scheduling for SMP VMs," in *Proc. 18th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2013, pp. 369–380.

[6] X. Song, J. Shi, H. Chen, and B. Zang, "Schedule processes, not VCPUs," in *Proc. 4th Asia-Pacific Workshop Syst.*, 2013, pp. 1:1–1:7.

[7] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for SMP VMs?" in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 257–272.

[8] L. Zhang, Y. Chen, Y. Dong, and C. Liu, "Lock-Visor: An efficient transitory co-scheduling for MP guest," in *Proc. 41st Int. Conf. Parallel Process.* 2012, pp. 88–97.

[9] B. Wang, et al., "Efficient consolidation-aware VCPU scheduling on multicore virtualization platform," *Future Generation Comput. Syst.*, vol. 56, pp. 229–237, 2016.

[10] J. Ahn, C. H. Park, and J. Huh, "Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2014, pp. 394–405.

[11] S. Wu, Z. Xie, H. Chen, S. Di, X. Zhao, and H. Jin, "Dynamic acceleration of parallel applications in cloud platforms by adaptive time-slice control," in *IEEE Int. Parallel Distrib. Process. Symp.* 2016, pp. 343–352.

[12] H. Mitake, T.-H. Lin, Y. Kinebuchi, H. Shimada, and T. Nakajima, "Using virtual CPU migration to solve the lock holder preemption problem in a multicore processor-based virtualization layer for embedded systems," in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.* 2012, pp. 270–279.

[13] K. Raghavendra, S. Vaddagiri, N. Dadhania, and J. Fitzhardinge, "Paravirtualization for scalable kernel-based virtual machine (KVM)," in *Proc. IEEE Int. Conf. Cloud Comput. Emerging Markets* 2012, pp. 1–5.

[14] S. Kashyap, C. Min, and T. Kim, "Opportunistic spinlocks: Achieving virtual machine scalability in the clouds," *ACM SIGOPS Operating Syst. Rev.*, vol. 50, no. 1, pp. 9–16, 2016.

[15] T. Friebel and S. Biemueller, "How to deal with lock holder preemption," *Xen Summit North America*, 2008.

[16] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Hardware support for spin management in overcommitted virtual machines," in *Proc. 15th Int. Conf. Parallel Archit. Compilation Techn.* 2006, pp. 124–133.

[17] J. Ouyang and J. R. Lange, "Preemptable ticket spinlocks: Improving consolidated performance in the cloud," in *Proc. 9th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.* 2013, pp. 191–200.

[18] M. Righini, "Enabling Intel® virtualization technology features and benefits," Intel White Paper. Retrieved Jan. 15 (2010): 2012.

[19] AMD Incorporation, "AMD64 architecture programmer's manual volume 2: System programming." 2006.

[20] P. Barham, et al., "Xen and the art of virtualization," in *Proc. 19th ACM Symp. Operating Syst. Principles*, 2003, pp. 164–177.

[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, pp. 225–230.

[22] C. A. Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Operating Syst. Rev.*, vol. 36, pp. 181–194, 2002.

[23] A. Theurer, "KVM and big VMs," 2012. [Online]. Available: http://www.linux-kvm.org/images/5/55/2012-forum-Andrew-Theurer-Big-SMP-VMs.pdf

[24] R. K. T, (2012). "[patch rfc 1/1] KVM: Add dynamic PLE window feature," [Online]. Available: https://lkml.org/lkml/2012/11/11/14

[25] R. Krčmář, (2014). "[patch v3 7/7]KVM: VMX: Optimize ple_window updates to VMCS," [Online]. Available: https://lkml.org/lkml/2014/8/21/456

[26] S. D. Lowe, (2013). "Best practices for oversubscription of cpu, memory and storage in vSphere virtual environments," [Online]. Available: https://software.dell.com/whitepaper/best-practices-for-oversubscription-of-cpu-memory-and-storage-in-vsphe823172/

[27] K. Raghavendra, "Virtual CPU scheduling techniques for kernel based virtual machine (KVM)," in *Proc. IEEE Int. Conf. Cloud Comput. Emerging Markets* 2013, pp. 1–6.

[28] J. Zhang, Y. Dong, and J. Duan, "ANOLE: A profiling-driven adaptive lock waiter detection scheme for efficient MP-guest scheduling," in *Proc. IEEE Int. Conf. Cluster Comput.* 2012, pp. 504–513.

[29] VMware, 2013. [Online]. Available: http://www.vmware.com/resources/techresources/10205

[30] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *Proc. 5th Annu. Workshop Model. Benchmarking Simul.*, Jun. 2009, http://www-mount.ece.umn.edu/~jjyi/MoBS/2009/MoBS_2009_Advance_Program.html

[31] R. R. Branco and V. Henson, "ebizzy-0.3," 2013. [Online]. Available: http://sourceforge.net/projects/ebizzy/

[32] P. Russell and M. Nordstrom, "dbench-measure disk throughput for simulated netbench run," 2005. [Online]. Available: http://manpages.ubuntu.com/manpages/raring/man1/dbench.1.html

[33] I. Molnar and Z. Yanmin, "Latest version of hackbench," 2008. [Online]. Available: http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c

[34] C. Kolivas, "Kernbench v0.50," 2009. [Online]. Available: http://ck.kolivas.org/apps/kernbench/kernbench-0.50/

[35] B. Huang and M. Zhu, "Research on necessity of adjusting PLE configuration," in *Proc. Int. Conf. Cloud Comput. Internet Things*, 2014, pp. 45–48.

[36] J. Shan, X. Ding, and N. Gehani, "APLE: Addressing lock holder preemption problem with high efficiency," in *Proc. IEEE 7th Int. Conf. Cloud Comput. Technol. Sci.*, 2015, pp. 242–249.

[37] L. Bin, (2014). "Enhancement for PLE handler in KVM," [Online]. Available: https://lkml.org/lkml/2014/3/3/356

[38] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry, "Decoupling contention management from scheduling," in *Proc. 15th Edition ASPLOS Archit. Support Program. Languages Operating Syst.* 2010, pp. 117–128.

[39] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein, "Optimal strategies for spinning and blocking," *J. Parallel Distrib. Comput.*, vol. 21, no. 2, pp. 246–254, 1994.

[40] B. He, W. N. Scherer III, and M. L. Scott, "Preemption adaptivity in time-published queue-based spin locks," in *Proc. 12th Int. Conf. High Performance Comput.* 2005, pp. 7–18.

[41] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki, "A new look at the roles of spinning and blocking," in *Proc. 5th Int. Workshop Data Manage. New Hardware* 2009, pp. 21–26.

Jianchen Shan received the BS and MS degrees in 2008 and 2011 both from Shanghai University, China. He is currently working toward the PhD degree in the Department of Computer Science, New Jersey Institute of technology. His research interests include parallel and distributed computing and cloud computing.

Xiaoning Ding received the PhD degree in computer science and engineering from the Ohio State University. He is an assistant professor with New Jersey Institute of Technology. His interests include the area of experimental computer systems, such as distributed systems, virtualization, operating systems, and storage systems.

Narain Gehaini received the PhD degree in computer science from Cornell University. He is currently a professor of computer science with New Jersey Institute of Technology. Previously, he was with Bell Labs. He has worked extensively in programming languages, software, and databases. He has authored several software systems, holds several patents, and has written many books and numerous papers in computer science.