2015 IEEE 7th International Conference on Cloud Computing Technology and Science

APLE: Addressing Lock Holder Preemption Problem with High Efficiency

Jianchen Shan Xiaoning Ding Narain Gehani
Department of Computer Science, New Jersey Institute of Technology
Email: {js622, xiaoning.ding, narain.gehani}@njit.edu

***Abstract*—On virtualized platforms, Lock Holder Preemption (LHP) is known as a serious problem, which makes virtual CPUs (VCPUs) spin excessively while waiting for locks and seriously degrades performance. To address this problem, hardware facilities, such as Intel PLE and AMD PF, are provided on processors to preempt spinning VCPUs. Though 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.**

The core issue in dealing with the LHP problem is to determine the best time to preempt spinning VCPUs (i.e., spinning thresholds). Due to the semantic gap between different software layers, the virtual machine monitor (VMM) does not have the information about whether a VCPU is spinning normally (i.e., waiting for a lock to be released quickly) or is spinning excessively (i.e., waiting for a lock which is currently held by a preempted VCPU and cannot be released quickly). Thus, it cannot determine adequate thresholds for preempting spinning VCPUs to achieve high performance. Preempting spinning VCPUs late wastes system resources. Preempting them prematurely incurs costly context switches between VCPUs and delays lock acquisition.

The paper addresses the issue of preempting spinning VCPUs with an end-to-end approach named Adaptive PLE (APLE). APLE monitors the execution efficiency of each VM by collecting the overhead incurred by wasteful spinning and wasteful VCPU switches. Then, it periodically adjusts the spinning threshold to reduce the overhead and increase the execution efficiency of the VM. The implementation of APLE incurs only minimal changes to existing systems (about 80 lines of code in KVM). The experiments with multicore workloads show that APLE can improve throughput by up to 68%.

Keywords*—virtualization; virtual CPU scheduling; lock-holder preemption; pause-loop exiting;

I. Introduction

In the cloud, virtual machines (VMs) with multiple virtual CPUs (VCPUs) have become increasingly popular. At the same time, the number of VCPUs in the same VM keeps increasing. For example, Amazon EC2 now provides VM instances with 40 VCPUs running on Intel Xeon E5-2676 v3 processors with hyperthreading [1]. On each VM, the "guest" operating system (OS) manages a set of VCPUs and schedules application threads on these VCPUs. In the cloud, a physical machine is often shared by multiple VMs. To coordinate the sharing of physical CPUs (PCPUs), on the host machine, the virtual machine monitor (VMM) independently schedules VCPUs onto the PCPUs.

Though a VCPU can run concurrently with other VCPUs, due to the time-sharing of PCPUs, it may be descheduled by the VMM to release the PCPU to another VCPU. It stops making progress until it is rescheduled. Unfortunately, this makes the behavior of the VCPUs not always match the behavior of PCPUs, which are expected to make continuous progress. Since the OS is designed and optimized for PCPUs, the mismatch between the VCPU abstraction and PCPU behavior introduces great challenges to synchronization and causes serious performance issues, particularly for multithreaded applications running on multicore VMs. One such issue is known as the Lock-Holder Preemption (LHP) problem, which has been extensively studied [2]–[13].

The LHP problem is caused when a VCPU is descheduled from the host platform while it is holding a spinlock. Since the lock-holding VCPU cannot proceed to release the lock, other VCPUs waiting on the lock cannot make progress until the lock-holding VCPU is rescheduled to release the lock. These VCPUs spin in spinlock primitives. The spinning (i.e., busy waiting) occupies physical resources and prevents the lock holding VCPU from being rescheduled quickly. Thus, the LHP problem causes excessive spinning, which significantly reduces system throughput.

To handle the LHP problem, hardware facilities are designed on processors to detect and stop VCPUs from excessive spinning, e.g., Intel Pause-Loop-Exiting (PLE) [14] and AMD Pause Filter (PF) [15]. Now, most virtualization systems (e.g., Xen, KVM, VMWare ESX, etc.) rely on these facilities. With such hardware facilities, the VMM sets a threshold for the time spent on continuous spinning on each processor. Then, the processor monitors the instructions being executed by each VCPU to detect spinning and stops the spinning if it exceeds the spinning threshold, so that the VMM can reallocate the processor to another VCPU that can make progress.

Despite the wide adoption of such hardware facilities, it is still a challenging issue to utilize these facilities to achieve high performance. On a physical machine equipped with such facilities, the VMM must carefully set spinning thresholds for VMs. On one hand, setting the thresholds high increases excessive spinning and leads to low resource utilization. On the other hand, spinlocks are often used to protect short critical sections in guest OSs and spinning ensures that a lock can be acquired as soon as it is released. If spinning thresholds are set too low, spinning VCPUs may be preempted prematurely. This delays lock acquisition and incurs costly context switches between VCPUs, which can significantly increase synchronization overhead and reduce system throughput. Thus, the VMM struggles between two conflicting objectives. One is to stop VCPU spinning as early as possible in case the lock holding VCPU has been preempted. The other is to avoid stopping VCPU spinning too early for efficient spinlock synchronization

978-1-4673-9560-1/15 $31.00 © 2015 IEEE
DOI 10.1109/CloudCom.2015.100

in case the lock holding VCPU is running and is about to release the lock. However, due to the semantic gap between software layers, the VMM does not have the information about whether a VCPU is holding a spinlock or whether the lock holding VCPU is running. Thus, it is challenging for the VMM to set spinning thresholds adequately.

To address this challenging issue of achieving high performance with hardware utilities like PLE, the paper proposes Adaptive PLE (APLE), with which the VMM can control the hardware facilities and deal with the LHP problem efficiently by dynamically adjusting spinning thresholds[1]. Instead of struggling with identifying whether lock-holding VCPUs are preempted, 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. APLE periodically measures the overhead caused by the wasteful operations and adjusts spinning thresholds to minimize this overhead.

The advantages of APLE are three-fold. First, 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. Second, since both wasteful spinning and wasteful VCPU switches are visible to and handled by the VMM, their overhead can be accurately measured in the VMM with low cost. 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. Third, the implementation of APLE incurs only minimal modification to existing VMM designs. Our implementation based on KVM adds only about 80 lines of source code to 4 existing files. This may help APLE be quickly adopted by existing virtualization systems.

The contributions of our work are as follows. Firstly, the paper systematically reviews and analyzes the problem faced by almost all mainstream virtualization systems — how to utilize hardware facilities to address the LHP problem efficiently to achieve high performance. Secondly, the paper proposes an innovative approach named APLE that controls the hardware facilities so as to minimize the overhead caused by wasteful spinning and wasteful VCPU switches. Thirdly, we implemented APLE in KVM and tested its performance on a 16-core system. We show that, with APLE, the performance can be improved by 68%.

II. Background and Motivation

In this section, we first introduce the hardware facilities for dealing with the LHP problem. Then, we explain how the existing virtualization systems utilize these facilities, using the support in KVM for Intel PLE as an example. At the end of the section, we present some performance results of a few standard benchmarks to show that the hardware facilities must be better controlled by VMMs to achieve better performance.

[1]Although the APLE design in the paper is based on Intel PLE, it can also be used on systems with AMD PF or other similar hardware utilities, which detect and stop spinning based on the thresholds set by the VMM.

A. Intel PLE and Other Similar Facilities

Modern processors provide hardware support for virtualization to reduce overhead. On these processors, PLE and other similar facilities are designed to deal with the LHP problem. With PLE, a processor first detects spinning VCPUs by examining the instructions executed by the VCPUs. On X86 architecture, spinlock 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 that can make progress.

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 preempts 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 use Intel PLE in APLE design and experiments. But the proposed methodology can be applied to the systems with AMD PF.

B. PLE Support in KVM

With PLE, when the two parameters *PLE_gap* and *PLE_window* are set, the processor detects and preempts spinning VCPUs autonomously. The VMM controls the PLE facility by selecting/adjusting the values of the parameters. It is relatively easy to find an adequate value for *PLE_gap* since PAUSE instructions are called much more frequently in spinlocks than other scenarios. By default, KVM sets *PLE_gap* to 128 cycles, which proves to be effective in practice. However, it is difficult to find an adequate value for the spinning threshold *PLE_window*, as we will show in the next subsection. Our work focuses on adjusting the spinning threshold.

Besides setting the two parameters, the VMM also needs to handle VM_EXITs caused by PLE facilities. Specifically for KVM, when a spinning VCPU of a VM is preempted, KVM examines the "ready" VCPUs in the same VM. If there is 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".

In the past, KVM would use a system-wide spinning threshold and set it to a fixed value selected empirically based on the spinning time under some typical workloads. However, the spinning time changes with workloads and with VM sizes (VCPU counts). A value that achieves optimal performance for some workloads may cause serious performance degradation for other workloads. On large VMs, even when lock holding VCPUs are not preempted, VCPUs may spend more time on spinning than the selected spinning threshold and may be preempted prematurely. This can significantly degrade performance. 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 [16].

The problem with a fixed spinning threshold has been noticed in the KVM community. There are attempts to dynamically adjust spinning thresholds. They mainly focus on improving the performance when PCPUs are under-subscribed. Since a lock holding VCPU will not be preempted when PCPUs are under-subscribed, spinning thresholds should be set high enough to prevent spinning VCPUs from being preempted prematurely.

For example, an attempt increases the threshold on "unsuccessful yieldings" and decreases it on "successful yieldings" [17]. The reasoning is that, if PCPUs are under-subscribed, only "unsuccessful yieldings" can take place when spinning VCPUs are preempted. Thus, the threshold should be lifted on "unsuccessful yieldings" to prevent spinning VCPUs from being preempted frequently. When PCPUs are over-subscribed, "successful yieldings" indicate that there are some non-spinning VCPUs preempted. These VCPUs may be lock holders. Reducing the spinning threshold helps getting them rescheduled quickly.

In the latest design, KVM maintains a spinning threshold for each VCPU and increases the thresholds of spinning VCPUs when they are preempted, and resets the thresholds to a default value when there are switches between different VCPUs [18]. When PCPUs are under-subscribed, there are no VCPU switches, and increasing the thresholds can prevent spinning VCPUs from being preempted prematurely. When the system is over-subscribed with multiple VMs, the VMM must switch VCPUs dynamically to time-share PCPUs. Thus, it resets spinning thresholds to deal with the LHP problem[2].

While these attempts can reduce the performance degradation caused by handling PLE events when PCPUs are under-subscribed, they do not appropriately adjust spinning thresholds when PCPUs are over-subscribed. For example, with the first attempt, when PCPUs are over-subscribed, "successful yieldings" may happen even when spinning thresholds are set too low and lock holding VCPUs are running. In such a case, further reducing the threshold degrades performance. With the current implementation, thresholds may vary widely when PCPUs are over-subscribed.

[2]The current KVM design also allows users to set the speed in which spinning thresholds are reduced. However, there is not any guidelines on how to set the speed, and resetting the thresholds is considered as the best choice and is selected as the default method.

C. Some Motivating Experiments

We now provide a quantitative illustration of the above problem using a few representative experiments. We select two benchmarks, *dbench* and *streamcluster*, and run them on a 16-core machine. (Please refer to Section IV 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 *streamcluster*). 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 32768 cycles. We repeat the experiments for each of the different threshold values, and show the normalize performance relative to that with the baseline configuration in Figure 1.

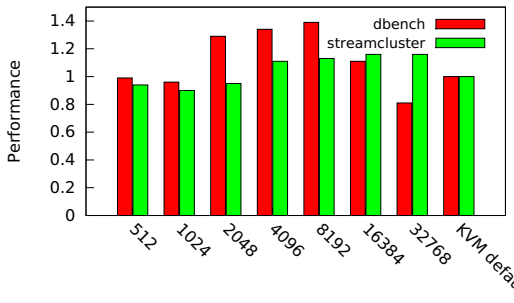

Fig. 1. The normalized performance of *dbench* and *streamcluster* when the spinning threshold is varied from 512 cycles to 32768 cycles, relative to the performance with the *default* KVM configuration.

The figure clearly shows that the performance of the two benchmarks changes with the threshold. The performance of benchmark *dbench* varies from 0.81 to 1.39, and the performance of *streamcluster* varies from 0.90 to 1.16. At the same time, different benchmarks achieve the best performance with different spinning thresholds (8192 cycles for *dbench* and 16384 cycles for *streamcluster*). 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 and achieves suboptimal performance.

III. APLE Design and Implementation

In this section, we first explain the rationale behind APLE. Then we introduce the algorithm in APLE for adjusting spinning thresholds.

A. Possible Approaches and APLE Basic Idea

An intuitive approach for adjusting spinning thresholds on a system 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. Firstly, 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). Secondly, 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 lock which is currently held by a preempted VCPU or not. Thus, it still cannot *estimate* the amount of spinning when the LHP problem does not happen.

APLE follows a trial and error approach. It 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 reduces 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 reduces if smaller thresholds are used. Thus, optimal thresholds can be approached by varying the thresholds and choosing those leading to lower overhead.

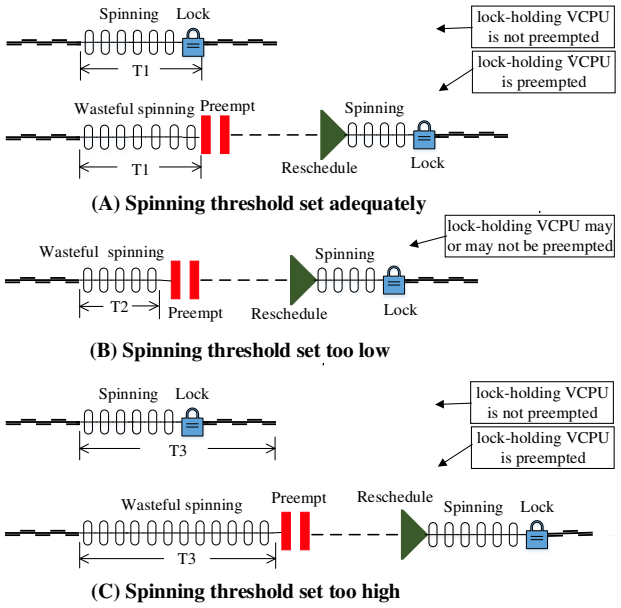

Fig. 2. The overhead from wasteful spinning and wasteful VCPU switches under three scenarios.

B. Wasteful Spinning and Wasteful VCPU Switches

To explain the rationale behind APLE, we use Figure 2 to compare the executions of a VCPU under three different scenarios: (a) when the spinning threshold is adequately set (Figure 2(A); (b) when the spinning threshold is set too low (Figure 2(B)); and (c) when the spinning threshold is set too high (Figure 2(C)). In the middle of the execution, the VCPU requests a spinlock 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.

As illustrated in Figure 2(A), 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 Figure 2(B), 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 Figure 2(A) (the spinning threshold adequately set), setting the threshold too low increases the chance that the spinning VCPU is preempted. The VCPU may have to be descheduled and rescheduled multiple times before it gets the lock. Thus, more wasteful spinning and wasteful VCPU switches are incurred.

If the spinning threshold is set too high ($T3$), as shown in Figure 2(C), 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.

C. The Calculation of Inefficiency as a Metric

APLE assumes that each workload runs in a VM. Since different workloads have different locking behaviors, APLE assigns a spinning threshold to each VM. 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.

[3]The spinning time may be measured with the collaboration from guest OSs [19], which is not available on public cloud.

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.

D. 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.

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
end while

When a VM is launched, this algorithm sets an initial value of the desired threshold T_d (e.g., 8192 in our experiments). While the VM is running, it tries the desired threshold and the thresholds slightly lower and slightly higher than the desired threshold, one 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 2014 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.

In APLE, the length of an epoch is not determined by clock time. Instead, it is determined by the number of times VCPUs are preempted (i.e., the number of VM_EXITs for PLE events on Intel Platforms). For example, in our experiments, an epoch corresponds to 1000 VCPU preemptions. With this method, when the VM rarely uses spinlocks, epochs are long time intervals; when the VM is spinlock-intensive, epochs are short time intervals. By using short epochs, APLE can quickly respond to execution phase changes. With long epochs, APLE can minimize runtime overhead. At the same time, this method also guarantees that there are enough sample events in each epoch so that the *inefficiency* can be reliably calculated.

IV. Experiments

To test the performance of APLE, we have implemented APLE in KVM. The implementation to the stock Linux kernel added only about 80 lines of code in 4 existing files. In our experiments, for all the VMs, the initial value of the desired threshold T_d is 8192 cycles. The lower bound T_l is 4096 cycles (the same as that in the default KVM setting). The upper bound T_u is 32768 cycles, and δ is 1024 cycles.

We conducted our experiments on a Dell PowerEdge R720 server with 64GB of DRAM and two 2.40GHz Intel Xeon E5-2665 processors. Each processor has 8 cores. On the server, we created 4 VMs with 16 VCPUs. Each VM has 16GB of memory. The VMM is KVM [20]. The host OS and the guest OS are Ubuntu version 14.04 with the Linux kernel version updated to 3.19.8. CPU power management can reduce the performance of the applications running in VMs [21]. 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.

We selected six benchmarks, *streamcluster*, *dedup*, *raytracer*, *dbench*, *kernbench*, and *ebizzy*, for evaluation. Among these benchmarks, *streamcluster*, *dedup*, and *raytracer* are from PARSEC 3.0 benchmark package [22]. All six benchmarks are introduced below. We selected these benchmarks because they incur frequent spinlock operations on VMs. Without PLE support, their executions on VMs suffer significant performance degradation due to the LHP problem. With PLE support, their performance on virtual machines are sensitive to spinning threshold levels.

- *streamcluster* is a program to solve the online clustering problem. For a set of points provided in a stream, it looks for a number of medians, so that the points can be clustered based on their nearest centers. It measures the clustering quality by calculating the sum of squared distances.
- *dedup* compresses a data stream with a combination of global and local deduplication. It uses a pipelined programming model to mimic real-world deduplication, which is widely used in new generation backup storage systems.
- *raytracer* renders a three-dimensional scene onto a two-dimensional image plane using optimized ray tracing. A hierarchical uniform grid is used to represent the scene for efficient access, and early ray termination and antialiasing are implemented in the benchmark.

- *ebizzy* [23] is a multi-threaded program that generates workloads similar to those on common web application servers.
- *dbench* [24] 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.
- *kernbench* [25] is a CPU and memory intensive benchmark that measures and compares the time used to compile Linux kernels.

We compiled the PARSEC 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 benchmarks with native input. In the experiments, we set the number of threads in each benchmark equal to 32. We run each experiment five times and report the average result.

We run the benchmarks using the default KVM configuration and use the 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 numbers 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.

We first launch one VM and run the benchmarks in the VM. We run each benchmark under three different scenarios: (1) with the default mechanism in KVM to adjust spinning thresholds and default configuration, (2) with APLE method to adjust spinning threshold, and (3) with the PLE support disabled. When only one VM is launched, the performance measured under scenario 3 represents the best performance these benchmarks can achieve. On average, the benchmarks achieve similar performance with APLE and the stock KVM (scenarios 1 and 2), and the performance difference is not noticeable (less than 2%). Compared to their executions under scenario 3, these benchmarks show slightly lower performance under the first two scenarios (1%∼2% on average and up to 8% for *kernbench*). The experiments show that, for a few benchmarks (e.g., *kernbench*), processing PLE events may still cause some performance degradation, though the degradation is not large.

Then, we launch two VMs. We run two instances of each benchmark in parallel on the two VMs, one on each VM. We run each benchmark under three different scenarios: (1) with the ***default*** mechanism in KVM to adjust spinning thresholds and default configuration, (2) with ***APLE*** method to adjust spinning thresholds, and (3) with both the default KVM mechanism to adjust spinning thresholds and APLE disabled. In scenario 3, we repeat the experiments for different spinning thresholds from 512 cycles to 32768 cycles. A benchmark shows different performance with different spinning thresholds. Thus, we can find the ***best*** performance and the ***worst*** performance that the benchmark can achieve by selecting different spinning thresholds. We use the best performance to show the potential of adjusting spinning threshold, and use the worst performance to illustrate how much performance degradation could be caused if the spinning threshold was not adequately set.

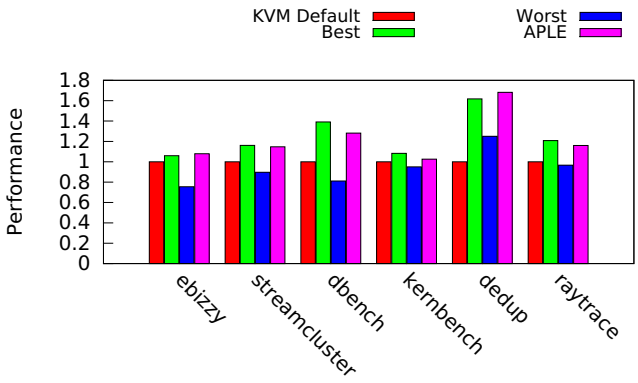

Fig. 3. Normalized performance of the benchmarks in different scenarios when 2 VMs co-run ("best" and "worst" are both under scenario 3 but with different levels of spinning thresholds)

Figure 3 show the performance of these benchmarks under these scenarios. As shown in the figure, the stock KVM cannot achieve optimal performance. Especially, with *dedup*, it even achieves lower performance than the "worst" performance obtained with a fixed spinning threshold level. Generally, APLE achieves similar performance as *"best"* — the best performance that can be obtained by smartly selecting a fixed spinning threshold for each VM. The average performance achieved with APLE is 1.21, and the average performance achieved by smartly selecting a fixed PLE_window (i.e., "best") is 1.23. Compared to the stock KVM, APLE improves the performance of *dedup* by the largest percentage (68%). The figure also shows that, when selecting a wrong spinning threshold level, the performance can be degraded by 27% on average and up to 42% (for *dbench*), relative to that when spinning thresholds are adequately set.

We also repeated the above experiments with four VMs. Specifically, for each benchmark, we run four instances of each benchmark in parallel on the four VMs, one on each VM. We run the benchmark under three different scenarios as described above, and show the performance of the benchmarks under these scenarios in Figure 4.

Compared to the executions with 2 VMs, the performance difference between APLE, the stock KVM ("default"), and the best performance ("best") reduces with 4 VMs. This is because, with more VMs, performance factors other than spinning (e.g., contention for memory and I/O bandwidth) start to dominate. Compared to the stock KVM, APLE improves the performance of the benchmarks by 3% on average. It improves the performance of *raytrace* by the largest percentage (8%). The average performance of APLE is not as high as "best". The average performance achieved with APLE is 1.03, the best performance the benchmarks can achieve is 1.09. This shows that there is still some space for APLE to further improve performance.

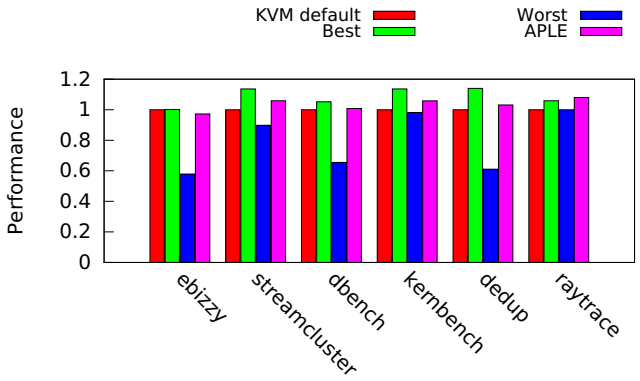

Fig. 4. Normalized performance of the benchmarks in different scenarios when 4 VMs co-run ("best" and "worst" are both under scenario 3 but with different levels of spinning thresholds)

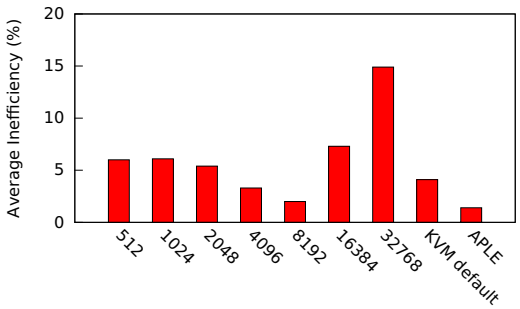

Fig. 5. The average inefficiency of *dbench* in different scenarios when 2 VMs co-run

The performance improvements achieved with APLE are by reducing inefficiency. We use *dbench* as an example to illustrate how APLE reduces inefficiency. Figure 5 compares the average inefficiency values for the different scenarios described above (i.e., scenarios with fixed PLE_window sizes, with the default KVM settings, and with APLE). Each value in the figure is the average of the inefficiency values measured in the epochs of the two VMs during two instances of *dbench* run in parallel in the VMs in the corresponding scenario. As shown in the figure, the average inefficiency reduces when PLE_window is increased from 512 cycles to 8192 cycles. This is because the overhead of wasteful VCPU switches caused by preempting spinning VCPU prematurely can be reduced with larger PLE_window sizes. However, when the PLE_window size is further increased, the average inefficiency increases, since the overhead of wasteful spinning starts to dominate. The default KVM mechanism cannot achieve the best performance since it cannot effectively reduce inefficiency. In contrast, APLE reduces the average inefficiency by 50%, relative to the stock KVM.

In the above experiments, we also collected the PLE_window sizes during the execution of the *dbench* instances[4]. Figures 6 and 7 show how PLE_window sizes 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 1900 epochs. This is because fewer VM_EXITs are incurred by PLE events with APLE. With the default KVM mechanism, the PLE_window size swings back and forth in a

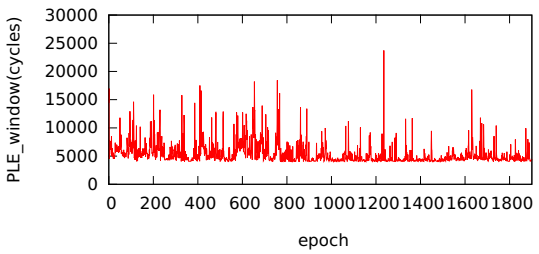

Fig. 6. Spinning threshold (PLE_window) adjusted by the default mechanism in KVM when two VMs co-run

wide range between 4096 and 25000. However, with APLE, the PLE_window size changes steadily around 8192, which leads to "best" performance.

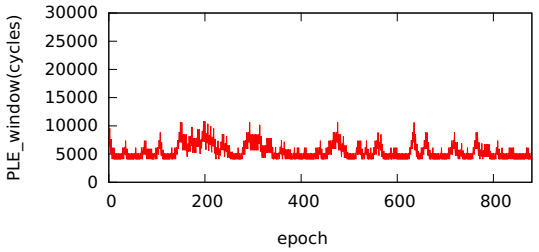

Fig. 7. Spinning threshold (PLE_window) adjusted by APLE when two VMs co-run

V. Related Work

A large number of studies have been focused on the lock holder preemption (LHP) problem. Various solutions have been proposed to reduce performance degradation, for example, sophisticated VCPU scheduling algorithms [2]–[7], VCPU migration [8], improved synchronization primitives [9], paravirtualization [10], [11], and hardware facilities to detect and preempt spinning VCPUs [12]–[15]. On current platforms, using hardware facilities, such as Intel PLE and AMD PF, has become a *de facto* standard solution. Our work does not provide an alternative solution to the LHP problem. Instead, it improves the solution with hardware facilities, which has been dominantly utilized on mainstream virtualization systems.

Targeting the solutions using hardware facilities, there are studies showing that spinning thresholds must be adjusted based on workloads to achieve best performance [19], [26]. There are also some efforts to adjust spinning thresholds dynamically. Zhang, Dong, and Duan [19] proposed a profiling method that collects the average spinlock 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 [17], [27] proposed a method to adjust the spinning threshold based on whether the freed resources can be reallocated to a

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

VCPU that can make progress when a spinning VCPU is preempted. Recently, KVM implemented a method to dynamically grow/shrink the spinning threshold for each VCPU [18]. These two methods mainly focus on improving the performance when the system is under-committed. When a spinning VCPU is preempted, the selection of a VCPU to occupy the freed resources may also impact performance. Thimmappa [28] studied this problem and proposed a method to quickly select a candidate that is likely to be the lock holder. The method has been accepted into Linux kernel and is orthogonal to APLE.

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 [29]–[32]. 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.

VI. Conclusion

Almost all mainstream virtualization systems rely on hardware facilities, such as Intel PLE and AMD PF, to alleviate performance degradation caused by the lock holder preemption problem. 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 APLE, which measures the execution efficiency of each VM and controls the hardware facilities to maximize the efficiency. Our studies show that APLE can effectively control the hardware facilities to improve performance. The methodology in APLE can be applied to any virtualization system based on hardware assisted virtualization techniques. Its implementation incurs minimal modification to existing virtualization system designs. As future work, we aim to test APLE with more extensive workloads and seek its adoption in mainstream virtualization systems.

VII. Acknowledgment

This research was supported by the National Science Foundation (NSF) under Grants No. CNS 1409523 and NJIT faculty seed grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] "Introducing m4 instances and lower amazon EC2 instance prices." [Online]. Available: http://aws.amazon.com/about-aws/whats-new/2015/06/introducing-m4-instances-and-lower-amazon-ec2-instance-prices/

[2] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines." in *VM 2004*, pp. 43–56.

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

[4] H. Kim, S. Kim, J. Jeong, and J. Lee, "Demand-based coordinated scheduling for SMP VMs," in *ACM ASPLOS 2013*, pp. 369–380.

[5] X. Song, J. Shi, H. Chen, and B. Zang, "Schedule processes, not VCPUs," in *APSys 2013*, pp. 1:1–1:7.

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

[7] L. Zhang, Y. Chen, Y. Dong, and C. Liu, "Lock-Visor: An efficient transitory co-scheduling for MP guest," in *ICPP 2012*.

[8] 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 *RTCSA 2012*, pp. 270–279.

[9] J. Ouyang and J. R. Lange, "Preemptable ticket spinlocks: Improving consolidated performance in the cloud," in *ACM VEE 2013*, pp. 191–200.

[10] "Paravirtual spinlocks." [Online]. Available: http://lwn.net/Articles/289039/

[11] K. Raghavendra, S. Vaddagiri, N. Dadhania, and J. Fitzhardinge, "Paravirtualization for scalable kernel-based virtual machine (KVM)," in *CCEM 2012*.

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

[13] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Hardware support for spin management in overcommitted virtual machines," in *PACT 2006*. ACM, pp. 124–133.

[14] M. Righini, "Enabling Intel® virtualization technology features and benefits," Intel, Tech. Rep., 2010.

[15] AMD, "AMD64 architecture programmers manual volume 2: System programming."

[16] 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

[17] R. K. T, "[patch rfc 1/1] kvm: Add dynamic ple window feature." [Online]. Available: https://lkml.org/lkml/2012/11/11/14

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

[19] J. Zhang, Y. Dong, and J. Duan, "ANOLE: A profiling-driven adaptive lock waiter detection scheme for efficient mp-guest scheduling," in *CLUSTER 2012*, pp. 504–513.

[20] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, 2007, pp. 225–230.

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

[22] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *MoBS 2009*, June.

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

[24] 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

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

[26] B. Huang and M. Zhu, "Research on necessity of adjusting ple configuration," in *CCIOT 2014*, 2014.

[27] R. K. Thimmappa, "Adjusting pause-loop exiting window values," Apr 2015, US Patent 9,021,497.

[28] K. Raghavendra, "Virtual CPU scheduling techniques for kernel based virtual machine (KVM)," in *CCEM 2013*, pp. 1–6.

[29] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry, "Decoupling contention management from scheduling," in *ASPLOS 2010*, pp. 117–128.

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

[31] B. He, W. N. Scherer III, and M. L. Scott, "Preemption adaptivity in time-published queue-based spin locks," in *High Performance Computing–HiPC 2005*. Springer, pp. 7–18.

[32] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki, "A new look at the roles of spinning and blocking," in *DaMoN 2009*, pp. 21–26.