Efficient Consolidation-aware VCPU scheduling on Multicore Virtualization Platform

Bei Wang*, Yuxia Cheng*, Wenzhi Chen*, Qinming He*, Yang Xiang*[§],
Mohammad Mehedi Hassan[#], and Abdulhameed Alelaiwi[#]

*College of Computer Science and Technology
Zhejiang University, Hangzhou, China
Email: {*wangbei, rainytech, chenwz, hqm*}*@zju.edu.cn*

[§]School of Information Technology
Deakin University, Melbourne, Australia
Email: *yang.xiang@deakin.edu.au*

[#]College of Computer and Information Sciences
King Saud University, Riyadh, Kingdom of Saudi Arabia
Email: *{mmhassan, aalelaiwi}@ksu.edu.sa*

Abstract

Multicore processors are widely used in today's computer systems. Multicore virtualization technology provides an elastic solution to more efficiently utilize the multicore system. However, the Lock Holder Preemption (LHP) problem in the virtualized multicore systems causes significant CPU cycles wastes, which hurt virtual machine (VM) performance and reduces response latency. The system consolidates more VMs, the LHP problem becomes worse. In this paper, we propose an efficient consolidation-aware vCPU (CVS) scheduling scheme on multicore virtualization platform. Based on vCPU over-commitment rate, the CVS scheduling scheme adaptively selects one algorithm among three vCPU scheduling algorithms: co-scheduling, yield-to-head, and yield-to-tail based on the vCPU over-commitment rate because the actions of vCPU scheduling are split into many single steps such as scheduling vCPUs simultaneously or inserting one vCPU into the run-queue from the head or tail. The CVS scheme can effectively improve VM performance in the low, middle, and high VM consolidation scenarios. Using real-life parallel benchmarks, our experimental results show that the proposed CVS scheme improves the overall system performance while the optimization overhead remains low.

Keywords: multicore, virtualization, Lock Holder Preemption, vCPU scheduling, consolidation

1 Introduction

Multicore processors are commonly deployed in computer systems from high-end servers to power efficient embedded devices. With the increasing number of processing cores integrated into the system, how to efficiently utilize the multicore processing power becomes a big challenge. To more flexibly utilize physical resources, virtualization technology is widely used in today's cloud computing environment. Virtualization technology enables multiple virtual machines (VMs) concurrently run in one physical machine, which provides an elastic resource provisioning method.

However, multiple VMs consolidated into the same physical machine will contend for shared CPU resources [1]. In a typical VM consolidation scenario, one physical core usually has multiple virtual CPUs (vCPUs) running on it (named vCPU overcommitment) [2]. The virtual machine monitor (VMM), which provides a virtual abstraction of machine hardware for each guest operating system (OS), schedules the vCPUs based on their time slice and priority [3]. The VMM has little

awareness of the code being executed inside each vCPU [4]. Therefore, the vCPUs that are holding spinlocks may be preempted [5] by other vCPUs due to the VMM's vCPU scheduling. The vCPUs that are waiting for the spinlocks have to spin for a much longer time when the vCPUs holding the locks are preempted. This is known as lock holder preemption (LHP) problem [6-9], which decreases VM performance and reduces system scalability.

In the non-virtualized environment, the LHP problem can be avoided [10, 11] by preventing the lock holder from being preempted until the lock holder releases the lock. The spinlock is designed to wait for a very short time and is used in the situations where context switches to yield CPU time slices are deemed as more costly than spinning. The lock holder can be easily detected by the native OS kernel. But when the OS is running inside a VM, the VMM cannot easily figure out whether the vCPU is holding the spinlock.

To address the LHP problem, researchers have proposed multiple solutions. The VM co-scheduling technique [5, 12] was proposed to simultaneously co-schedule all or part of the vCPUs in one VM onto physical cores to avoid the LHP problem. However, co-scheduling has some weaknesses [13, 24], such as CPU utility fragmentation [14, 26] and increased system latency. Para-virtualization technique provides another solution to address the LHP problem by modifying the spinlock primitives in the guest OS to cooperate with the underlying VMM [4, 15, 16]. In this way, the VMM can detect the lock holder and avoid preempting the vCPU that are holding the spinlock. But modifying the primitive spinlock cannot be easily applied in those proprietary OSes such as Windows and Mac OS X [17].

The precise lock holder detection in the full virtualization environment is currently not available [1, 8]. An alternative hardware assisted technology was introduced to approximately detect the lock waiter, such as Intel PAUSE Loop Exiting (PLE) [18] and AMD PAUSE Filter (PF) [19]. The PLE/PF mechanism is designed to detect the guest OS execution of the PAUSE instruction that is used in the spin loop code. By detecting the potential lock waiter, the VMM can then make the lock waiter vCPU yield its CPU cycles and donate them for other vCPUs [17]. Therefore, the overall system throughput can be improved. However, the vCPU yield policy can lead to unfairness for the donating vCPUs and impact their response latency [7, 8].

In this paper, we propose an efficient consolidation-aware vCPU scheduling (CVS) scheme that considers different VM consolidation scenarios. The CVS scheme takes advantages of the low overhead hardware assisted PLE mechanism to detect lock waiter vCPUs. Based on vCPU over-commitment rate, the CVS scheduling selects one algorithm among three vCPU scheduling algorithms: co-scheduling, yield-to-head, and yield-to-tail. The main contributions are described as follows:

(1) We evaluate three vCPU scheduling algorithms in the multicore virtualized system under different VM consolidation scenarios. We observe that different scheduling algorithms have performance advantages and disadvantages under different vCPU over-commitment rate settings. We analyze the behavior of three vCPU scheduling algorithms and find the most suitable algorithm in a certain VM consolidation scenario.

(2) After analyzing the characteristics of three vCPU scheduling algorithms, we propose an efficient consolidation-aware vCPU scheduling (CVS) scheme that adaptively selects the most suitable vCPU scheduling algorithm online based on the vCPU over-commitment rate. The CVS scheme dynamically changes vCPUs scheduling strategies according to CPU run queue status and the positions of the lock waiter vCPU and the lock holder vCPU. The CVS can more effectively

address the LHP problem and improve VM performance than the single policy vCPU scheduling algorithms.

(3) We extensively evaluate the proposed CVS scheme in the multicore virtualized platform. We run the real-life parallel benchmarks in the multicore VMs. The evaluation results demonstrate that our proposed CVS scheme can effectively improve the overall VM performance, and the optimization overhead remains low.

The rest of this paper is organized as follows. Section 2 describes the motivation of this paper and shows the experimental observations. Section 3 presents our proposed consolidation-aware vCPU scheduling scheme in detail and describes its implementation in the Xen virtualized platform. Section 4 shows the evaluation results. Finally, we conclude this paper in section 5.

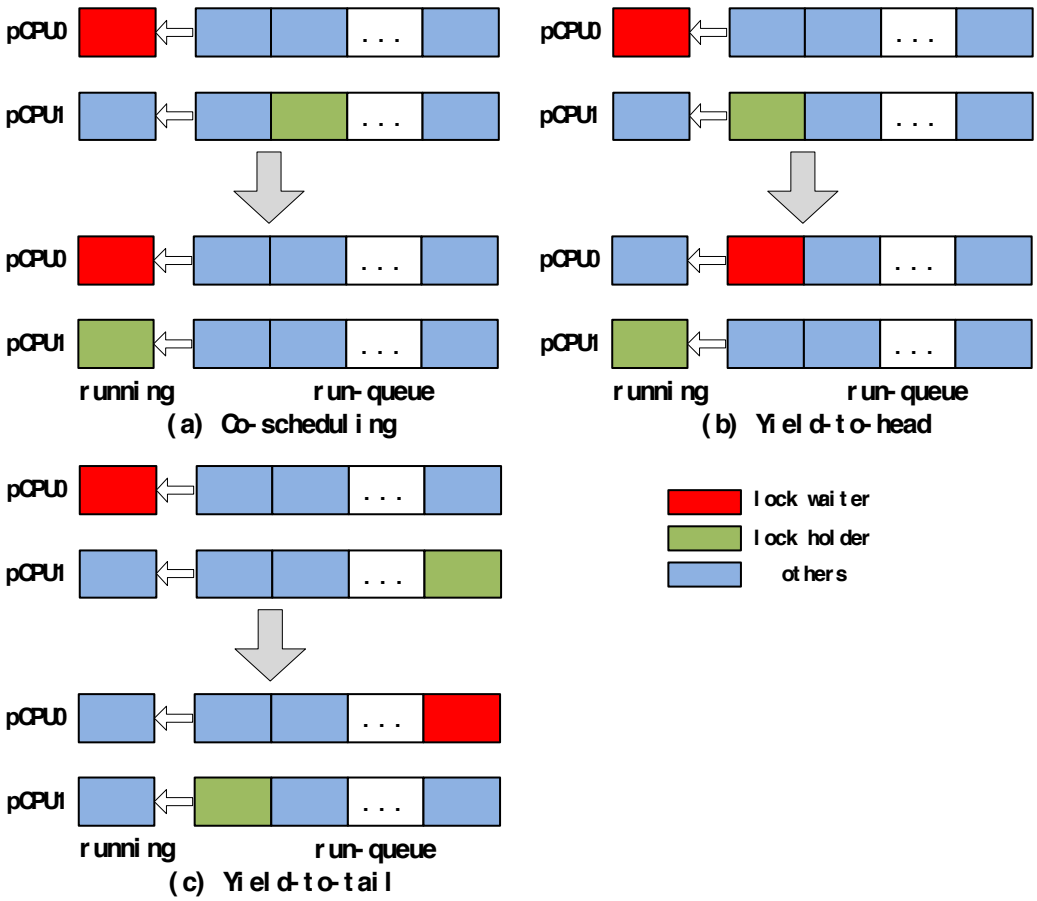

Fig. 1. Three vCPU scheduling algorithms: (a) Co-scheduling. (b) yield-to-head. (c) yield-to-tail

2 Motivation

The lock holder preemption (LHP) problem in the virtualized environment can be mitigated via three different vCPU scheduling algorithms: co-scheduling, yield-to-head (YTH), and yield-to-tail (YTT), because the actions of vCPU scheduling are split into many single steps such as scheduling vCPUs simultaneously or inserting one vCPU into the run-queue from the head or tail.

The co-scheduling algorithm, shown in Fig. 1(a), schedules all vCPUs of a VM synchronously on physical cores. The YTH algorithm detects the lock waiter vCPU and inserts it to the head of the corresponding CPU run queue. The YTH algorithm can be illustrated with Fig. 1(b). The lock waiter vCPU yields its CPU time slice to other vCPUs and waits in the head of CPU run queue for the next round. The YTT algorithm is the default scheduling methods for Xen Hypervisor. As shown in Fig.

1(c), YTT detects the lock waiter vCPU and inserts it to the tail of the corresponding CPU run queue. In the YTT algorithm, the lock waiter vCPU also yields its CPU time slice to other vCPUs but waits in the tail of CPU run queue for the next round.

Algorithm	YTT	YTH	co-sched
Overhead	Low	Low	High
Performance	Low	Mid	High
Degradation	Low	Mid	High

Table 1. The properties of three algorithms.

From the analysis above, we fill Table 1 with the properties of three algorithms. Table 1 shows that three algorithms have their own advantages and disadvantages. To study the performance characteristics of these algorithms, we implemented three algorithms in the Xen hypervisor and evaluated their performance using parallel benchmarks. We run 4 VM guests with 3GB simultaneously and concurrently using the Kernbench [20] as the workload. Each VM guest has the same number of vCPUs. We gradually increase the number of vCPUs in each guest to increase the probability of the LHP problem. Since only the number of CPUs or VMs could not reflect the contention for CPU resources between VMs, we consider the vCPU over-commitment rate (VOR) to represent the current system's VM configurations. VOR is calculated as the following equation:

$$\mathrm{VOR} = \frac{\sum_{i=1}^{n} VN(vm_i)}{Total\ Number\ of\ pCPUs}, \quad (1)$$

where $VN(vm_i)$ represents the number of vCPUs in the i*th* VM. For example, if the system has 8 physical CPUs (pCPUs) and 4 VMs with each configured with 2 vCPUs run on the system, then the VOR value equals to (2+2+2+2)/8. The higher the value of VOR, the higher probability the LHP problem occurs.

Fig. 2 shows the total execution time of the Kernbench running inside 4 VMs under three vCPU scheduling algorithms: YTT, YTH, and co-scheduling. When VOR varies from [1.00, 1.75], [1.75, 3.00], to [3.00, 4.00], the best performance algorithm varies from co-scheduling, YTH, to YTT respectively.

When the value of VOR falls in [1.00, 1.75], the co-scheduling algorithm outperforms both the YTH and YTT algorithms. However, the performance of co-scheduling algorithm deteriorates drastically when the value of VOR is between 1.5 and 2.0, and remains relatively stable after 2.0. Fig. 2 shows that the user time and system time both increase because of the CPU utility fragmentation problem that the co-scheduling algorithm brings. In the co-scheduling algorithm, all the other vCPUs need to wait for the readiness of the last vCPU, which results in the increase of user time. And all the vCPUs need to be coordinated which results in the increasing of system time.

When the value of VOR falls in [1.75, 3.00], the YTH algorithm outperforms the co-scheduling and the YTT algorithms. The performance of YTH gets close to linear with the CPU reuse ratio. Fig. 3 shows the system time rather than user time increases due to the lock waiting in the kernel before the lock is released.

When the value of VOR falls in [3.00, 4.00], the best performance algorithm is YTT. The performance of YTT remains relatively stable when the CPU reuse ratio is more than 2. Because

the YTT algorithm can effectively reduce the lock waiter's spinning time in the high vCPU over-commitment scenario.

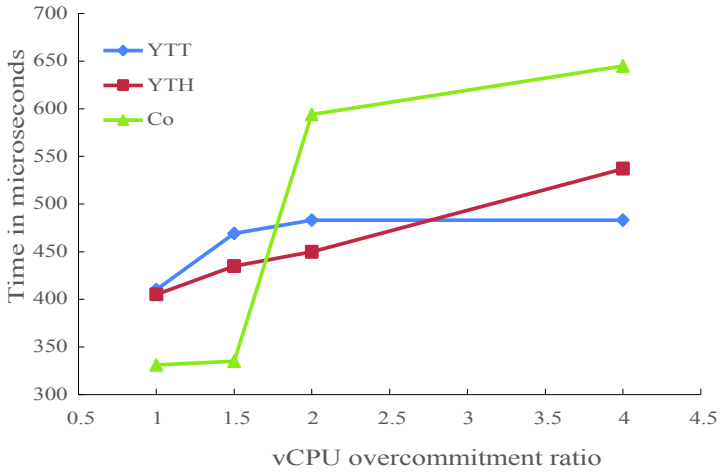

Fig. 2. The total execution time of all the Kernbench benchmarks on 4 VM guests. It is recorded on host OS.

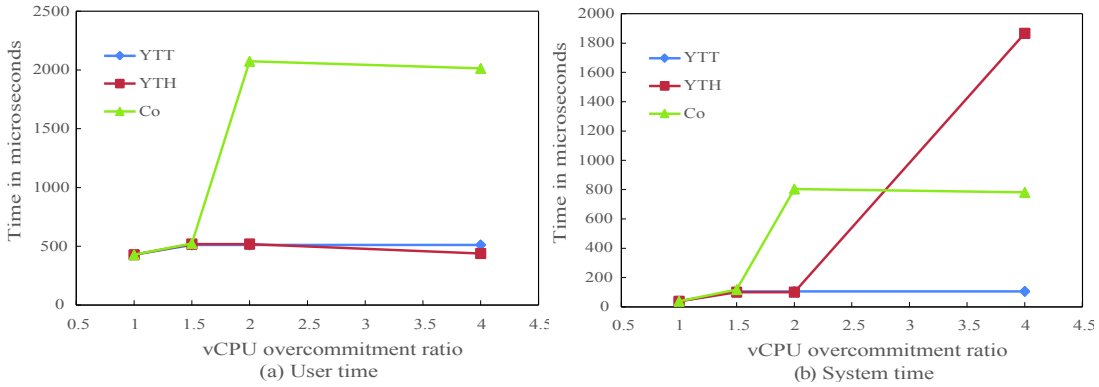

Fig. 3. The user time and system time of all the Kernbench benchmarks on 4 VM guests. They are reported by the Kernbench in the VM guests.

From the experimental results, we make the following observations:

(1) In the low vCPU overcommitment rate scenario, the co-scheduling algorithm outperforms the other two algorithms. When the VMM detects the lock waiter, the co-scheduling algorithm can effectively co-schedule the lock holder and the lock waiter can soon acquire the lock.

(2) The performance of co-scheduling algorithm deteriorates when the vCPU overcommitment rate increases. Therefore, in a certain VOR value interval, the YTH algorithm performs better than the co-scheduling algorithm. Because the YTH algorithm can reduce the CPU fragmentation problem.

(3) In the high vCPU overcommitment rate scenario, the YTT algorithm outperforms both the co-scheduling and the YTH algorithms. When the VOR value increases, the average length of CPU run queue increases. In the YTT algorithm, the lock waiter is inserted into the tail of CPU run queue and when it gets scheduled again the lock waiter can very likely acquire the lock.

Because the lock holder can have higher probability got scheduled and release the lock before the lock waiter gets scheduled again.

Therefore, a good vCPU scheduling scheme should consider the current system's vCPU overcommitment rate when addressing the LHP problem. When the VMM detects the lock waiter, the vCPU scheduler that intelligently selects the best scheduling policy can achieve better performance than just using a single scheduling algorithm. This observation motivates the design of our proposed CVS scheme.

3 Design and Implementation

3.1 Overview

To exploit more performance opportunities, we design an efficient consolidation-aware vCPU scheduling (CVS) scheme to adaptively select appropriate vCPU scheduling algorithms online. The proposed CVS scheme takes advantage of the co-scheduling, yield-to-head (YTH), and yield-to-tail (YTT) algorithms. Based on the system's online vCPU overcommitment rate, the CVS intelligently selects one of the three vCPU scheduling algorithms to address the locker hold preemption problem timely. Due to the total number of runnable vCPUs in a virtualized system varies dynamically, the CVS scheme captures this variation online and makes scheduling adjustment accordingly. In the low VM consolidation rate scenario, which means the vCPU overcommitment rate is relatively low, the CVS invokes the vCPU co-scheduling algorithm when the PLE mechanism detects the locker waiter. With the vCPU overcommitment rate changed, the CVS alleviates CPU fragmentation problem for less co-scheduling frequency. In the high VM consolidation rate scenario, the CVS invokes the vCPU YTT algorithm to mitigate the LHP problem. And, when the vCPU overcommitment rate lies between the low and high VM consolidation scenarios, the CVS invokes the vCPU YTH algorithm. In some extent the YTH algorithm is a performance tradeoff between the co-scheduling and the YTT algorithms.

3.2 Consolidation-aware vCPU scheduling

Consolidation-aware vCPU scheduling (CVS) scheme consists of the following three major steps:

(1) The hardware assisted PLE mechanism continually monitors the vCPU execution of the PAUSE instructions. When the PLE detects the lock waiter vCPU, it sends a PLE VMEXIT signal to the virtual machine monitor (VMM).

(2) When the VMM receives an PLE VMEXIT interrupt from the PLE, the VMM calculates the vCPU overcommitment rate and determines the suitable vCPU scheduling algorithm.

(3) After choosing the suitable scheduling algorithm, the VMM invokes the corresponding scheduling functions to address the detected LHP problem.

The CVS scheme classifies the VM consolidation scenario into three categories: low, middle, and high. To determine the exact VM consolidation scenario, we calculate the runnable vCPU overcommitment rate online. The vCPU over-commitment rate (VOR) is calculated as Eq. (1). Since the total number of running vCPUs equals to the number of active threads in the physical machine, we can infer the average length of each CPU run queue and each vCPU's expected position in the CPU run queue from the value of VOR.

The average length of each CPU run queue:

$$Len_{avg} = \mathrm{VOR}, \tag{2}$$

The vCPU's expected position in the CPU run queue:

$$Pos_{exp} = (1 + Len_{avg})/2 = (1 + VOR)/2 \quad (3)$$

The vCPU's expected distance to the head of the CPU run queue:

$$Dist_{exp} = Pos_{exp} - 1 = (1 + VOR)/2 - 1 = (VOR - 1)/2 \quad (4)$$

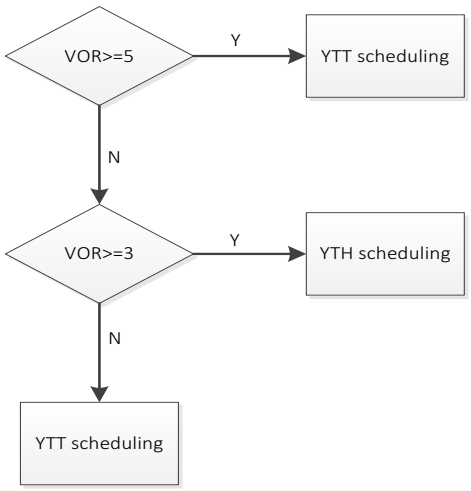

Fig. 4. CVS selects the scheduling algorithm by the value of VOR

Therefore, based on the vCPU's expected distance to the head of the CPU run queue, we can select the appropriate scheduling algorithms accordingly. When the VMM detects the lock waiter, shown in Fig. 4, the vCPU scheduling algorithm selection process is shown as follows:

(1) If the current system's vCPU run queue meets the requirement: $Dist_{exp} < 1$, then CVS selects the co-scheduling algorithm. In this scenario, $Dist_{exp} = (VOR - 1)/2 < 1 \rightarrow VOR < 3$, which means the vCPU overcommitment rate is low. The co-scheduling algorithm performs better than the other algorithms in the low VM consolidation environment.

(2) If the current system's vCPU run queue meets the requirement: $1 \leq Dist_{exp} < 2$, then CVS selects the YTH algorithm. In this scenario, $1 \leq (VOR - 1)/2 \leq 2 \rightarrow 3 \leq \mathrm{VOR} < 5$, which means the lock waiter vCPU can be re-scheduled only once to acquire the lock using the YTH algorithm. Because the value of $Dist_{exp}$ is smaller than 2, the corresponding lock holder has a very high probability to be scheduled during the time the lock waiter is sitting in the head of the CPU run queue. Therefore, when $3 \leq \mathrm{VOR} < 5$, CVS selects the YTH algorithm can effectively address the LHP problem while reduce the CPU fragmentation problem occurred in the co-scheduling algorithm.

(3) If the current system's vCPU run queue meets the requirement: $Dist_{exp} \geq 2$, then CVS selects the YTT algorithm. In this scenario, $(VOR - 1)/2 \geq 2 \rightarrow \mathrm{VOR} \geq 5$, which means the vCPU overcommitment rate is high. If the VMM still invokes the YTH algorithm to schedule lock waiter vCPUs, then the lock waiter vCPUs will have a very low probability of acquiring the lock. Because the lock holder may be situated in the end part of CPU run queue which cannot be scheduled before the lock waiter gets scheduled again in the head of run queue. Thus, the YTH algorithm will waste CPU time slice in the high VM consolidation scenarios. Therefore, the CVS selects the YTT algorithm that inserts the lock waiter vCPUs into the tail of CPU run queue. When the lock waiter vCPU is re-scheduled, it will have a high probability of acquiring the lock. Because the lock holder vCPU is scheduled and releases the lock before the lock waiter vCPU gets re-scheduled again in the YTT algorithm.

Through adaptively selecting the appropriate vCPU scheduling algorithms according to online

VM consolidation rate, the CVS scheme can efficiently exploit the performance opportunities in the virtualized multicore systems.

3.3 Implementation

There are two methods of implementing the proposed scheduling algorithm: hacking Xen Credit Scheduler or adding an additional mechanism to the trigger function of PLE. We implement the CVS in Xen virtualization platform using the latter implementation method. Considering the following two reasons: (1) Hacking Xen Credit Scheduler may destruct the fairness of the VM scheduling, while our implementation does not cause such side-effect. (2) The LHP problem may not occur in some cases, especially when the workloads in the VMs are not parallel applications. In this case, hacking Xen Credit Scheduler increases more costs than our implementation, because it needs to check whether LHP problem occurs and decide when to do co-scheduling. While in our implementation we utilize the hardware to decide when to do co-scheduling, it reduces the decision costs of scheduling moment.

Fig. 5 is the CVS scheduling flow. When the lock holder is de-scheduled, the lock waiter will wait until the timeout. Then the PLE hardware detects the timeout and triggers VMEXIT signal, which calls the corresponding handler function. The handler function chooses an appropriate algorithm to do the corresponding scheduling job.

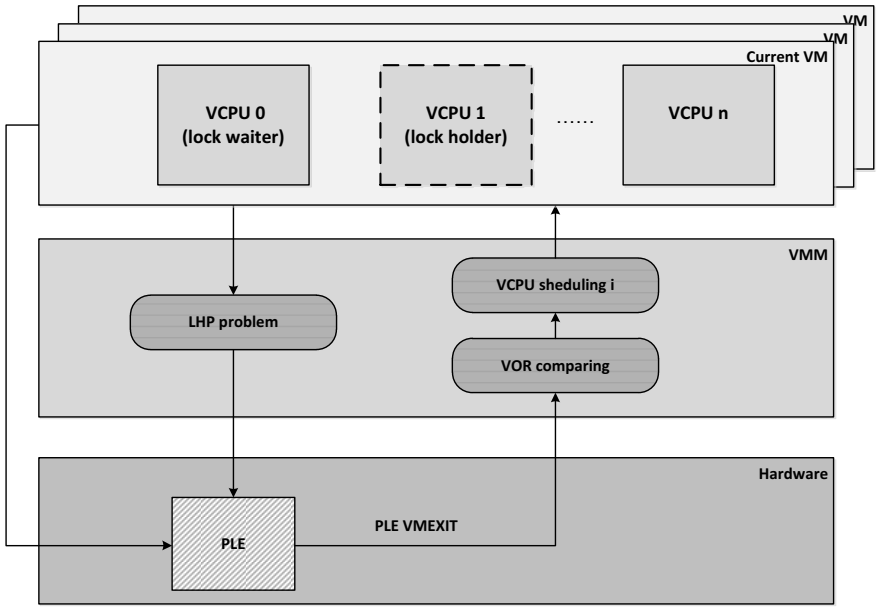

Fig. 5. The architecture of the CVS scheduling architecture

4 Performance Evaluation

Our testbed system is Dell T710 with 2 quad-core CPUs (Intel Xeon 5620) and 24GB DDR3 RAM. The operating system is Ubuntu 12.04 Server x64 with Xen 4.1.2 and Linux kernel 3.2.0. The kernel of Guest OS is the original kernel 3.2.0-32. Each VM is configured with 8 vCPUs and 2GB of memory.

We select three benchmarks to evaluate the proposed CVS scheme and other scheduling algorithms. The benchmarks are listed as follows:

(1) SysBench. We run the OLTP test mode of SysBench [27] to evaluate the impact of CVS on the database systems.

(2) Kernbench. We use Kernbench to evaluate the impact of CVS on the real jobs which is I/O-intensive and lock-intensive because multi-compiling threads compete for the inode-lock and frequently access files.

(3) NPB. NPB [28] is a collection of MPI benchmarks as follows: BT, CG, EP, IS, LU, MG, SP. We compile BT, EP, IS, MG in Class B and others in Class B.

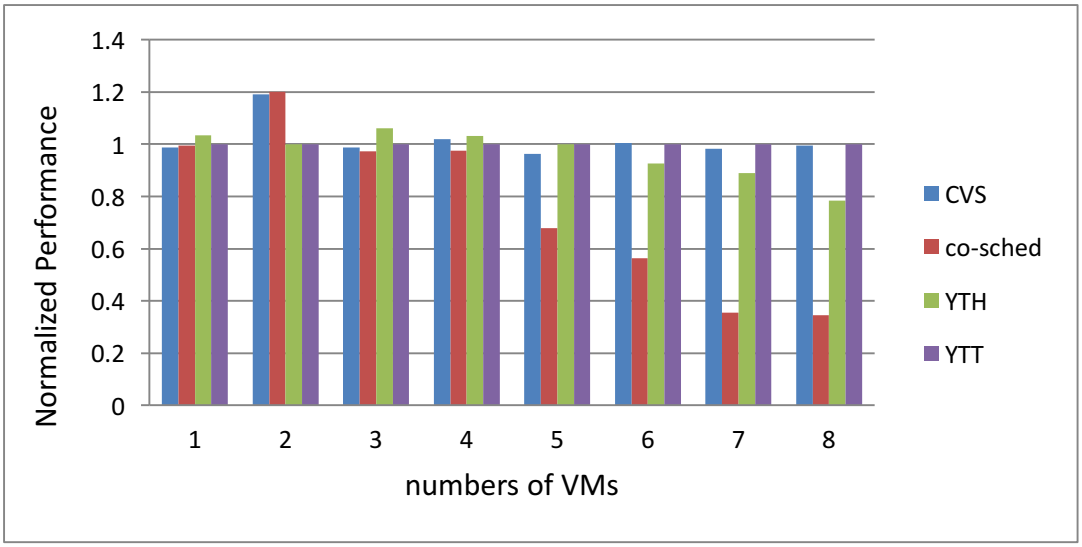

Fig. 6. The performance of SysBench benchmark under different vCPU scheduling algorithms.

In the first experiment, we execute the benchmarks in one VM. All the other VMs run CPU BOMBs (the synthetic CPU intensive micro-benchmark). We launch two types of VMs. One VM is called benchmark-VM with benchmarks running on it. All the other VMs running CPU BOMBs that are named Bomb-VM. These experiments test the effect of CVS and other scheduling algorithms when the CPU contention VMs are increasing.

All benchmarks and CPU BOMBs are configured with 8 threads (the number of vCPUs). The numbers of BOMB-VMs are increasing from 0 to 7. It is clear that the VOR (vCPU overcommitment rate) value equals the numbers of running VMs. We select YTT as the baseline because it is the default scheduling for handling LHP problems in Xen hypervisor.

First, we conduct the SysBench benchmark experiment. As is shown in Fig. 6, the CVS algorithm matches the best performance in 7 settings. But when the numbers of VM is 3, the performance of CVS is 15% less than YTT. This is because a delay of each connection to database is set in SysBench. When VOR is smaller than 3, CVS chooses the co-scheduling algorithm. When VOR is between 3 and 5, CVS adaptively chooses the YTH algorithm. And when VOR is higher than 5, CVS predicts that the YTT algorithm will achieve better performance. The result shows that the proposed CVS algorithm dynamically chooses the suitable vCPU scheduling algorithms according to the different VM consolidation scenarios, and thus can improve system performance.

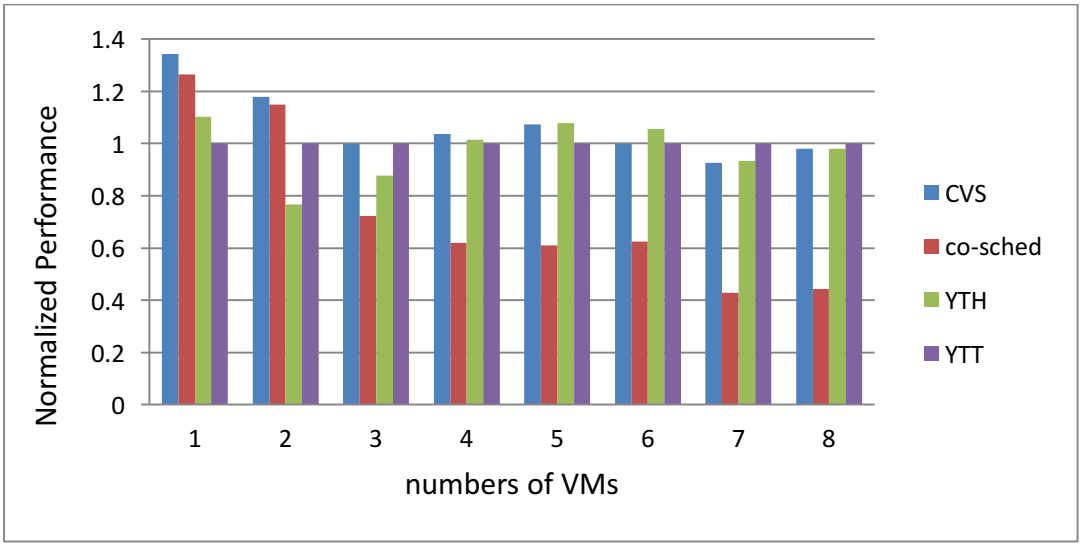

Fig. 7. The performance of KernBench benchmark under different vCPU scheduling algorithms.

Second, we conduct the KernBench benchmark experiment. As is shown in Fig. 7, the performance of co-scheduling algorithm degrades as the vCPU overcommitment rate increases. This demonstrates the performance issues incurred by the co-scheduling algorithm due to CPU fragmentation. Therefore, under the kernel building usage scenario, CVS adaptively chooses the the YTH algorithm when the VM consolidation rate exceeds 3. In this experiment, the YTH and YTT scheduling algorithms have very close performance results, because the KernBench is the I/O intensive benchmark. When in the high VM consolidation rate scenario, the performance bottleneck shifts to the resource contention of I/O resource.

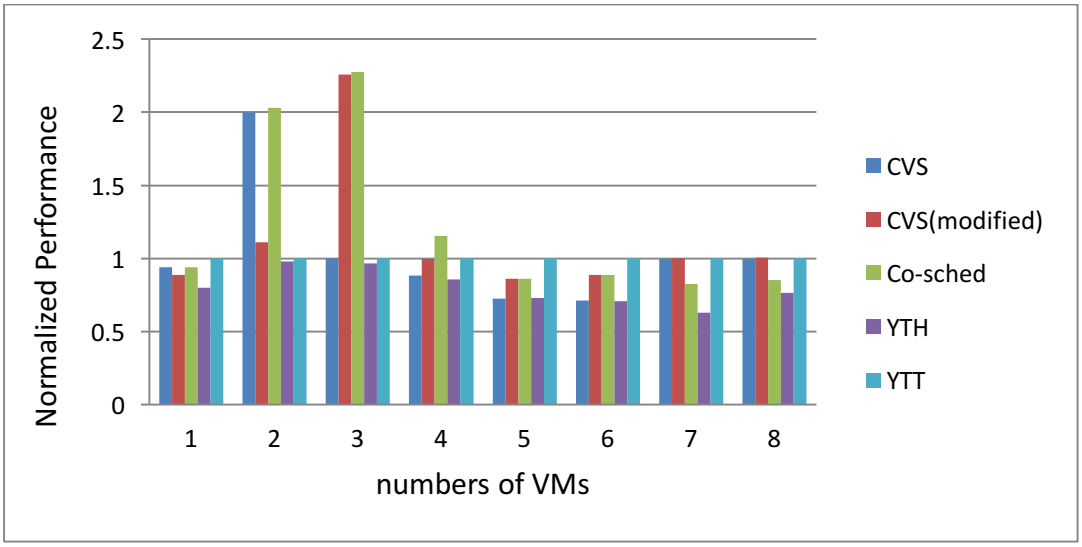

Fig. 8. The performance of NPB benchmarks under different vCPU scheduling algorithms.

We run all the benchmarks of NPB by the order of BT, CG, EP, IS, LU, MG and SP. The whole running time of NPB is recorded from the host. As is shown in Fig. 8, the performance of the co-scheduling algorithm under the NPB benchmark is much better compared to the SysBench and KernBench benchmarks. It is reasonable because the NPB benchmark suite is a parallel high performance computing benchmark and the threads which have MPI communications need to be scheduled timely. From this experiment, we found that the YTH algorithm is unfriendly to the MPI

applications. Therefore, we modified the CVS algorithm through bypassing the YTH algorithm. When the VM consolidation rate is under 4, the CVS algorithm selects the co-scheduling algorithm and in other circumstances CVS selects the YTT algorithm. As the experiment result shows, the modified CVS performance better than the original CVS algorithm under the MPI parallel applications.

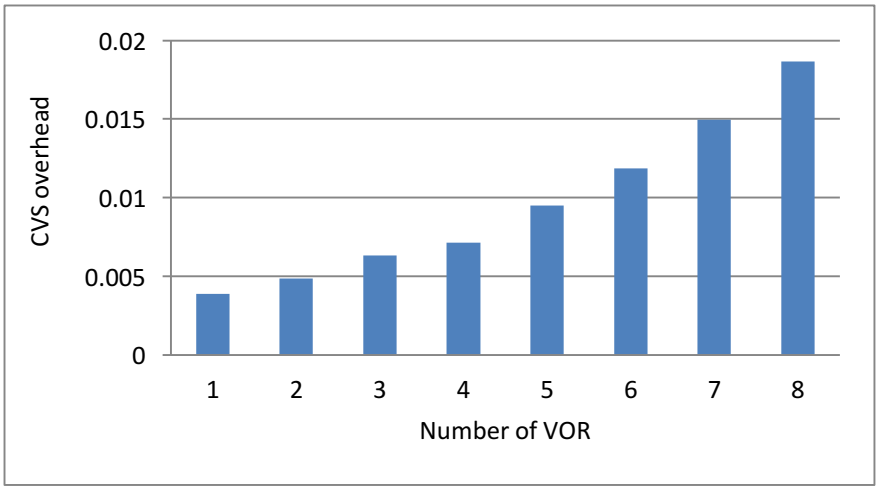

Fig. 9. The CPU overhead of CVS with different numbers of VOR.

We evaluate the performance overhead of the CVS optimization scheme. As the Fig. 9 shows, the extra CPU consumption incurred by the CVS optimization code remains relatively low, as the total number of vCPUs in the system increases, the CPU consumption increases slightly. This is because that the major overhead of the CVS optimization scheme consists of monitoring the number of runnable vCPUs in the system and the frequency of PLE VMEXIT signal. Therefore, the CPU consumption is higher in the high VM consolidation scenario than in the low VM consolidation scenario.

5 Related Work

There has been significant research interest on managing the synchronization overhead in parallel programs. Previous researchers [10, 11] have studied the Lock Holder Preemption (LHP) problem in parallel applications. Ousterhout [21] proposed a thread co-scheduling algorithm that schedules multiple threads belonging to the same application simultaneously to address the LHP problem. Although the traditional co-scheduling algorithm improves synchronization efficiency, it suffers from CPU utility fragmentation, priority inversion and increased system latency [14]. To improve the classic co-scheduling algorithm, other less strict co-scheduling approaches [22, 4, 23] select the communicating processes in advance.

In the virtualized environment, the semantic gap between the VMM and guest OSes brings more challenges to the scheduling problem. Uhlig et al. [5] proposed two methods of inferring lock holders for full-virtualization and para-virtualization environment. In the full-virtualization scenario, the method assumes that the lock holders are always in the OS kernel mode. Therefore, based on monitoring user-kernel mode switches, the VMM can infer the possible lock holders and prevent the lock-holder vCPU from being preempted. In the para-virtualization scenario, the modified guest OS kernel marks the running vCPU when the vCPU enters into a critical section protected by locks. Thus, the VMM can explicitly detect the lock holder vCPU. Both methods have their weaknesses. First, the solution to the full-virtualization is not applicable to all scenarios [1]. Libraries with

synchronization in user mode escape the monitoring of LHP by avoid switching the user-kernel mode. Second, the solution to the para-virtualization needs to modify the code of guest kernel. But the modification of code is not always feasible for proprietary OSes.

Hardware assisted technologies were proposed to address the LHP problem in the virtualized platform. Dong et al. [17] proposed a light weight yielding approach that leverages the Intel's processor's PLE (Pause Loop Exit) [18] feature to approximately infer lock waiters. The PLE mechanism (similarly, AMD's Pause Filter [19]) is designed to detect long time spinning by monitoring the PAUSE instruction continuously. A vCPU that spins for a predefined threshold will cause a VMEXIT from the guest OS to the VMM and yield the CPU time slice to other VMs. The PLE technique incurs expensive VMEXIT operation. Chakraborty et al. [2] proposed another hardware assisted approach to infer lock waiters using low-level hardware performance counters to collect performance statistics of STORE instructions. However, hardware assistance is only effective in full-virtualization. Further, the constant parameters of hardware are not suitable for the different configurations and workloads. Zhang et al. [7] proposed a runtime lock waiter detection approach that leverages the PLE feature to preempt vCPUs adaptively.

To reduce the CPU fragmentation and performance degradation of vCPU scheduling, some approaches were proposed. Zhang et al. [8] explored multiple co-scheduling policies and transitorily co-schedules in all potential lock holders to bypass the guest spin lock loop. Sukwong et al. [13] proposed the balance scheduling algorithm by balancing vCPU siblings on different CPUs without forcing the vCPUs to be scheduled simultaneously. Rao et al. [1] proposed Flex, a vCPU scheduling scheme that enforces fairness at VM-level and improves the efficiency of hosted parallel applications. Ding et al. [9, 25] pointed out that the wakeup path of blocked-waiter can significantly increase the execution time of virtualized multithreaded applications. They proposed a corresponding approach to lessen the frequency that vCPUs enter/exit idle loops. None of these studies identify that the scheduling policy needs change with the increasing vCPU overcommitment ratio.

6 Conclusion

The lock holder preemption (LHP) problem in the virtualized multicore systems causes significant performance degradation. Existing solutions to address the LHP problem includes: vCPU co-scheduling, yield-to-head, and yield-to-tail scheduling algorithms implemented in the full-virtualization environment; spinlock modification in the para-virtualization environment; and the hardware assisted virtualization technologies like the PLE mechanism provides more efficient method to address the LHP problem.

In this paper, we proposed the consolidation-aware vCPU scheduling (CVS) scheme to improve VM performance in the virtualized multicore systems. We first conducted a series of experiment to show that different vCPU scheduling algorithms have advantages and disadvantages under different VM consolidation scenarios. Then, based on the experimental result observations, we proposed the CVS scheme that adaptively selects the appropriate vCPU scheduling algorithms. According to VOR and the positions of lock waiter and lock holder in the CPU run queue, the CVS scheme dynamically adjust vCPU scheduling strategies. Comparing with co-scheduling algorithm, the CVS reduces CPU fragmentation problem in the dynamical VM consolidation scenario for less co-scheduling frequency. The CVS scheme takes advantage of the PLE mechanism and is implemented in the Xen full virtualization environment without changing the Credit scheduler code. We extensively evaluated the proposed CVS scheme using real-life parallel benchmarks. The

experimental results showed that the CVS scheme can effectively improve VM performance and incurs little performance overhead.

7 Acknowledgements

This research was supported by the National Key Technology R&D Program of the Chinese Ministry of Science and Technology under Grant No. 2012BAH94F03 and the Key Science and Technology Innovation Team Fund of Zhejiang under Grant No. 2010R50041.

References

[1] Jia Rao, Xiaobo Zhou, Towards fair and efficient SMP virtual machine scheduling, in: Proc. of PPoPP, 2014.

[2] K. Chakraborty, P. M. Wells, G. S. Sohi, Supporting over-committed virtual machines through hardware spin detection. IEEE Trans. Parallel Distrib. Syst. 23 (2) (2012).

[3] W. Jiang, Y. Zhou, et al. CFS Optimizations to KVM threads on multi-core environment, in: Proc. of ICPDS, 2009.

[4] A. C. Arpaci-Dusseau, Implicit co-scheduling: coordinated scheduling with implicit information in distributed systems, ACM Trans. Comput. Syst. 19(3) (2001).

[5] V. Uhlig, J. Levasseur, E. Skoglund and U. Dannowski, Towards scalable multiprocessor virtual machines, in: Virtual Machine Research and Technology Symposium, USENIX, 2004.

[6] Hitoshi Mitake, Tsung-Han Lin, et al. Using virtual CPU migration to solve the lock holder preemption problem in a multicore processor-based virtualization layer for embedded systems, in: Proc. of ERTCSA, 2012.

[7] Jian Zhang, Yaozu Dong, Jiangang Duan, ANOLE: A profiling-driven adaptive lock waiter detection scheme for efficient MP-guest scheduling, in: Proc. of Cluster, 2012.

[8] Lei Zhang, Yu Chen, Yaozu Dong, Chao Liu, Lock-visor: an efficient transitory co-scheduling for MP guest, in: Proc. of ICPP, 2012.

[9] X. Ding, P. B. Gibbons, M. A. Kozuch, J. Shan, Gleaner: mitigating the blocked-waiter wakeup problem for virtualized multicore applications, in: Proc. of ATC, 2014.

[10] D. L. Black, Scheduling support for concurrency and parallelism in the Mach operating system. IEEE Computer. 23 (5) (1990) 3543.

[11] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors, in: Proc. of PPoPP, 1995.

[12] Y. Yu, Y. Wang, H. Guo, X. He, Hybrid co-scheduling optimizations for concurrent applications in virtualized environments, in: Proc. of NAS, 2011.

[13] O. Sukwong, H. S. Kim. Is co-scheduling too expensive for SMP VMs? In: Proc. of EuroSys, 2011.

[14] W. Lee, M. Fran, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for gang scheduled workloads, in: Proc. of IPPS, 1997.

[15] H. Kim, S. Kim, J. Jeong, J. Lee, S. Maeng, Demand-based coordinated scheduling for SMP VMs, in: Proc. of ASPLOS, 2013.

[16] K.T. Raghavendra, S. Vaddagiri, N. Dadhania, J. Fitzhardinge, Paravirtualization for scalable kernel-based Virtual Machine (KVM), in: Proc. of CCEM, 2012.

[17] Yaozu Dong, Xudong Zheng, et al. Improving virtualization performance and scalability with advanced hardware accelerations, in: Proc. of IISWC, 2010.

[18] Intel Corporation, Intel 64 and IA-32 architecture software developer's manual, December, 2014.
[19] AMD Corporation, MD64 architecture programmers manual volume 2: System programming, 2014.
[20] C. Kolivas, Kernbench, December 2009, http://mirror.sit.wisc.edu/pub/linux/kernel/people/ck/apps/kernbench/.
[21] J. Ousterhout. Scheduling techniques for concurrent systems, in: Proc. of ICDCS, 1982.
[22] Y. Wiseman, D. Feitelson. Paired gang scheduling, IEEE Trans. Parallel Distrib. Syst. 14 (6) (2003) 581.
[23] P. Sobalvarro, S. Pakin, W.E. Weihl, and A. A. Chien, Dynamic co-scheduling on workstation clusters, in: Proc. of JSSPP, 1998.
[24] Chuliang Weng, Qian Liu, Lei Yu, and Minglu Li, Dynamic adaptive scheduling for virtual machines, in: Proc. of HPDC, 2011.
[25] X. Ding, P. B. Gibbons, M. A. Kozuch, A hidden cost of virtualization when scaling multicore applications, in: Proc. of HotCloud, 2013.
[26] X. Song, J. Shi, H. Chen, B. Zang, Schedule processes, not VCPUs. In Proc. of APSys, 2013.
[27] MySQL AB, SysBench manual, 2010.
[28] H. Jin, M. Frumkin and J. Yan, The OpenMP implementation of NAS Parallel Benchmarks and its performance, NASA Ames Research Center, Tech. Rep. NAS-99-011, 2003.