Schedule Processes, not VCPUs*

Xiang Song, Jicheng Shi, Haibo Chen, Binyu Zang
Institute of Parallel and Distributed Systems
School of Software, Shanghai Jiao Tong University

Abstract

Multiprocessor virtual machines expose underlying abundant computing resources to applications. This, however, also worsens the double scheduling problem where the hypervisor and a guest operating system will both do the CPU scheduling. Prior approaches try to mitigate the semantic gap between the two levels of schedulers by leveraging hints and heuristics, but only work in a few specific cases. This paper argues that instead of scheduling virtual CPUs (vCPUs) in the hypervisor layer, it is more beneficial to dynamically increase and decrease the vCPUs according to available CPU resources when running parallel workloads, while letting the guest operating system to schedule vCPUs to processes. Such a mechanism, which we call vCPU ballooning (VCPU-Bal for short), may avoid many problems inherent in double scheduling. To demonstrate the potential benefit of VCPU-Bal, we simulate the mechanism in both Xen and KVM by assigning an optimal amount of vCPUs for guest VMs. Our evaluation results on a 12-core Intel machine show that VCPU-Bal can achieve a performance speedup from 1.5% to 57.9% on Xen and 8.2% to 63.8% on KVM.

*This work was supported by China National Natural Science Foundation under grant numbered 61003002 and a Foundation for the Author of National Excellent Doctoral Dissertation of PR China.

APSys '13, July 29-30 2013, Singapore, Singapore

1 Introduction

System virtualization has been widely used in many commercial usage scenarios such as server consolidation, multi-tenant cloud and virtual appliances. With virtualization, individual workloads can be deployed on separate virtual machines (VM) hosted on the same machine for performance isolation and resource sharing. With the increasing computing capability of underlying processors by adding more cores, a single VM can be configured with multiple virtual CPUs (vCPU) to take advantage of such abundant computing resources.

In virtualized environments, there is a double scheduling phenomenon: 1) the OS schedules processes on vCPUs and 2) the hypervisor schedules vCPUs on physical CPUs (pCPU). Due to the semantic gap between the two schedulers (i.e., most operations inside a guest OS are opaque to the hypervisor), double scheduling introduces new problems that not exist in no-virtualized environments. For example, a spinlock holder will be preempted even after disabling the kernel preemption (e.g., use spin_lock_irq in Linux kernel) in a virtualized environment [16]. This will significantly increase the synchronization latency as another vCPU may wait for the same lock at the same time. Such a problem can be severe for SMP VMs running parallel applications, resulting in significant performance degradation.

Unfortunately, state-of-the-art vCPU schedulers, such as completely fair scheduler (CFS) in KVM [1] and credit scheduler in Xen [4], are oblivious to the double scheduling problem. To this end, researchers have proposed a number of approaches to bridging this semantic gap, including relaxed co-scheduling [2, 17], balance scheduling [15], and demand-based coordinated scheduling [9]. However, these approaches either introduce other problems when reducing the synchronization latency (e.g., co-scheduling and demand-based coordinated scheduling) or do not solve the problem completely (e.g., balance scheduling).

This paper argues that instead of trying to bridge the semantic gap using some specific heuristics and hints, it would be more beneficial to eliminate this gap by minimizing the vCPU number of each VM to exclude the hypervisor from vCPU scheduling as much as possible. To this end, we propose VCPU-Bal, a vCPU ballooning scheme that dynamically adjusts the number of runnable vCPUs of each VM according to its VM weight to minimize the runnable vCPUs. With the minimal number of runnable vCPUs, the hypervisor can assign each vCPU to an exclusive physical CPU in many cases, which may eliminate the double scheduling problem. Besides, even when there is more vCPUs than available pCPUs, by minimizing number of runnable vCPUs, VCPU-Bal can help other schedule schemes such as co-scheduling and balanced scheduling minimize the overhead of double scheduling. Further, as commodity operating systems and some applications do not scale well with a large number of cores [6, 7], especially on the virtual environments [14, 13] where the vCPUs are asymmetric in essence, reducing the amount of runnable vCPUs in a VM can help avoid reaching the scalability bottleneck.

As currently a complete implementation of the proposed scheme is still on the way, we simulate VCPU-Bal by assigning an optimal amount of vCPUs and evaluate it against the default and an affinity-based scheduling using four applications from PARSEC and two applications from the Phoenix benchmark suite. On a 12-core machine, the average performance improvement for PARSEC applications over the state-of-the-art schedule scheme is 13.6% ranging from 1.5% to 25.9% on Xen and 25.5% ranging from 8.4% to 49.2% on KVM. The average performance improvement of histogram and wordcount from Phoenix test suite is 52.7% and 57.9% respectively on Xen and 54.4% and 63.8% respectively on KVM.

The rest of this paper is organized as follows. The next section discusses the scheduling trap at the presence of double scheduling and examines existing proposals, which motivate the design of VCPU-Bal. Section 3 describes the design the VCPU-Bal mechanism, followed by the evaluation in section 4. Section 5 relates VCPU-Bal with previous work. Finally, we conclude this paper with a brief description of our future work.

2 VCPU Scheduling Trap

Double scheduling can cause significant performance degradation of parallel applications running on a guest VM. To illustrate such an overhead, we have conducted a simple performance evaluation using the streamcluster application from PARSEC on an Intel machine with 12 cores. We evaluate its performance under two settings: 1) Single-VM case: we launch a guest VM with 8 GByte memory and 12 vCPUs, which runs a streamcluster instance; and 2) Two-VM case: we launch two guest VMs each configured with 8 GByte memory and 12 vCPUs, each of which runs a streamcluster instance. In the two-VM case, there are 24 vCPUs in total, which is twice the number of physical cores that may cause serious double scheduling problem. Figure 1 shows the result. Although the computing resource of the each guest VM in the two-VM case is 50% of the computing resource of the guest VM in the single-VM case, the execution time of streamcluster in two-VM case is 2.6X longer than that in single-VM case on KVM and 2.1X longer on Xen. This is because double scheduling introduces extra performance overhead on synchronization operations inside the kernel. As shown in the figure, the accumulated time spent in guest kernel in the two-VM case is 5.2X longer than that in single-VM case on KVM and 16.7X longer on Xen.

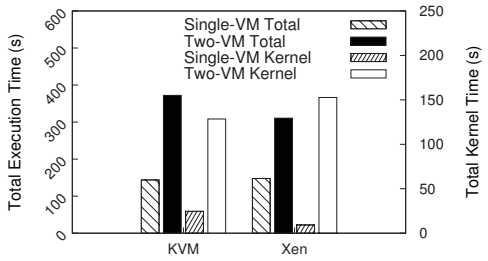

Figure 1: Performance of streamcluster in single-VM and two-VM cases.

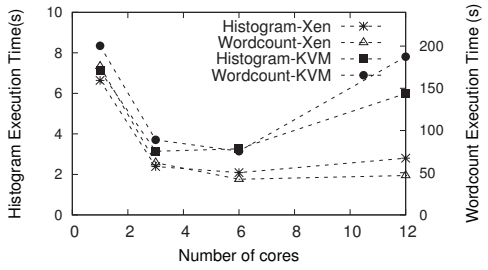

Figure 2: Scalability of histogram and wordcount on Xen and KVM.

Further, many operating systems and applications do not scale well with large number of cores, especially on virtual environments [14, 13]. Actually Amdahl's law [8] has limited the speedup of increasing the number of CPU cores at the presence of serial sections. Figure 2 shows a simple evaluation on the scalability of histogram and wordcount on both Xen and KVM. It can be seen that both histogram and wordcount start to encounter scalability problem when the number of cores exceeds 3.

2.1 Issues with Existing VM Scheduling

Due to the semantic gap between the OS scheduler and the hypervisor scheduler, double scheduling introduces many problems. Table 1 summarizes the problems of different scheduling strategies in virtualized environments.

State-of-the-art scheduling schemes, such as CFS used by KVM and credit scheduler used by Xen, are usually oblivious of synchronization operations inside an SMP VM. Hence, it may preempt a vCPU regardless of what it is running. This results in two basic problems introduced by double scheduling: VCPU preemption and VCPU stacking.

VCPU Preemption: Whenever a lock holder vCPU is preempted by the hypervisor and switched to another vCPU, the synchronization time spent by another vCPU waiting for the lock will be extended significantly. Figure 3 shows an example. The lock holder vCPU0 on pCPU0 is preempted after it enters the critical section at T1. Meanwhile, vCPU1 on pCPU1 is waiting for the lock holder to exit the critical section. However, vCPU0 is rescheduled on pCPU0 on T2 only after the execution of another VM finishes. Thus, vCPU1 has to wait *Tlock_wait* to enter the critical section. Here, the synchronization time spent by vCPU1 is extended by Tpreempt. Further, during this period, vCPU1 may occupy pCPU1 busy-waiting for the lock holder, which will waste lots of CPU cycles.

VCPU Stacking: Normally, a scheduler allows vCPUs to be scheduled on any pCPUs. This will cause the vCPU stacking problem that the lock waiter is scheduled before the lock holder on the same pCPU. Figure 4 shows an example. The lock holder on vCPU0 is preempted when it enters the critical section at T1. Unfortunately, the lock waiter on vCPU1 is scheduled before vCPU0 on the same pCPU (pCPU0) until T2. Thus, vCPU1 has to wait Tlock_wait to enter the critical section. Here, the synchronization time spent by vCPU1 is extended by Ttask1 plus Ttask2.

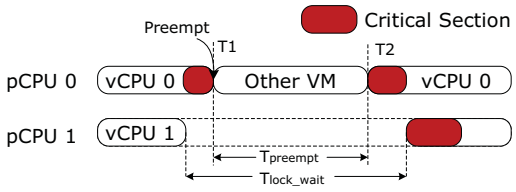

Figure 3: Synchronization latency caused by vCPU Preemption.

Co-scheduling: To solve these two problems, researchers proposed relaxed co-scheduling [2, 17] schemes. It schedules vCPUs of the same SMP VM simultaneously. A simple way is to find a time slice that has a sufficient number of available pCPUs to run all tasks. This can help avoid the vCPU stacking problem and reduce the computing resources wasted due to vCPU preemption. However, it introduces CPU fragmentation and priority inversion.

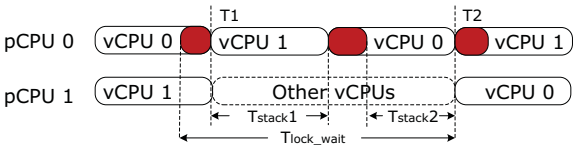

Figure 4: Synchronization latency caused by vCPU stacking.

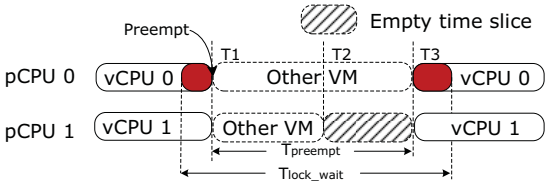

Figure 5: Synchronization latency in co-scheduling.

CPU Fragmentation: Figure 5 shows an example of CPU fragmentation. When the lock holder vCPU0 is preempted at T1, the lock waiter will also be preempted at T1. Although vCPU1 is runnable on pCPU1 at T2, it cannot be scheduled until T3. Thus, the computing resources between T2 and T3 are wasted.

Priority Inversion: Priority invention is where a vCPU with higher priority is scheduled after a vCPU with lower priority. For example, co-scheduling vCPUs may occupy all pCPUs at the current time slice, when an emergent vCPU comes to the run queue. The emergent vCPU has to wait until the end of this time slice.

Balance Scheduling: Balance scheduling [15] tries to spread vCPUs of a VM on different pCPUs without scheduling the vCPUs at the same time. By avoiding placement of sibling vCPUs on the same pCPU runqueue, it can avoid the vCPU stacking problem. However, it still has vCPU preemption problem.

Demand-based Scheduling: Demand-based scheduling [9] bridges the semantic gap between the double schedulers by taking inter-processor interrupts (IPIs) as a guidance of coordination demands [1] and proposing a IPI-driven co-scheduling and delayed preemption schedule scheme. However, it may cause priority inversion problem due to its urgent-vCPU-first scheduling mechanism which may preempt a high priority vCPU for an urgent vCPU. Further, this scheme is specific to the IPI-based coordination implementation of the guest OS, and it lacks support of spin-based synchronization detection.

[1] Most coordination demands in kernel such as TLB shootdowns and synchronization operations are followed by IPIs.

	Credit	CFS	Co-schedule	Balance	Demand-based	VCPU-Bal
vCPU Preemption	yes	yes	no	yes	no	no
vCPU Stacking	yes	yes	no	no	no	no
CPU Fragmentation	no	no	yes	no	no	no
Priority Inversion	no	no	yes	no	yes	no
Sync. Impl. Sensitive	no	no	no	no	yes	no
Spinlock	yes	yes	no	yes	yes	no

Table 1: A summary of possible issues with different VM schedulers.

2.2 Motivating VCPU ballooning

Although there is a semantic gap between the OS scheduler and the hypervisor scheduler, double scheduling is not necessary in many cases, where we can let the guest OS schedule its processes and exclude the hypervisor from vCPU scheduling. Instead, the hypervisor cooperate with the guest OS to decide an optimal amount of vCPUs assigned to guest OS. By dynamically adjusting the total number of runnable vCPUs, we may directly assign each vCPU to a pCPU exclusively. To do so, we should minimize the number of runnable vCPUs of each VM, and instead rely on the process scheduler in guest OS to do the scheduling. Here, we propose VCPU-Bal, a vCPU ballooning scheme, to achieve this. VCPU-Bal has the following benefits:

- In many cases, when each pCPU has only one vCPU scheduled on it, there will be no vCPU preemption and vCPU stacking problems for VCPU-Bal. Further, it neither introduces problems as shown in Table 1.
- VCPU-Bal can work for any kinds of guest VMs and is applicable to most virtual environments. Further, it can work in collaboration with other scheduler schemes such as co-scheduling and balanced scheduling and help them ease the overhead of double scheduling.
- By minimizing the runnable vCPUs of a VM, VCPU-Bal can help the VM avoid reaching the scalability bottleneck.

3 Design of VCPU-Bal

This section describes a general design of VCPU-Bal, which is composed of three parts: 1) a manager inside the hypervisor or the management VM makes vCPU balloon and schedule decisions; 2) a kernel module inside the guest OS dynamically adjusts runnable vCPUs according to the ballooning commands from the manager; and 3) a VCPU-Bal scheduler inside the hypervisor schedules vCPUs according to the schedule decisions.

VCPU-Bal Manager: VCPU-Bal manager makes vCPU balloon and schedule decisions based on the runnable guest VMs and the underlying hardware resource statistics. VCPU-Bal manager decides the number of pCPUs assigned to a VM according to its weight share. The following expression shows how the number of pCPUs of VMa, N_{pCPU}, is calculated. In the expression, W_{VMi} means the weight of VMi and T_{pCPU} means the total number of pCPUs.

$$N_{pCPU} = \frac{W_{VMa} * T_{pCPU}}{\sum_0^n W_{VMn}}$$

Suppose we have two running VMs, VM1 and VM2, with the weight of 512 and 256 respectively. The underlying hardware has 12 pCPUs. The VCPU-Bal manager will assign VM1 8 pCPUs and VM2 4 pCPUs, and tell VM1 and VM2 to balloon their runnable vCPUs to 8 and 4 respectively. Then, VCPU-Bal manager will try to schedule vCPUs from different VMs to different pCPUs and tell this scheduling decision to the VCPU-Bal scheduler. For example, it will assign vCPUs of VM1 to pCPU 0-7 and vCPUs of VM2 to pCPU 8-11.

VCPU Balloon Module: The vCPU balloon module inside the guest OS is responsible to dynamically online or offline vCPUs according to the ballooning commands. The simplest way to realize the ballooning is to HLT the victim vCPU when it is offline and reactivates it when it is online. When a vCPU is set offline, it will first clear its runqueue and then invoke the HLT instruction to enter into deep sleep, which will cause the hypervisor to schedule it off the runnable vCPU list. The sleeping vCPU will only be responsive to virtual hardware events, such as IPIs or timer interrupts. To reactivate it, an interrupt can be used to exit this vCPU from the HLT state. Although this method is simple, its implementation requires some modifications to the existing OS scheduler. Further, the offline vCPU is still available to the guest OS that it can be the target of IPIs and other hardware events. Thus, an offline vCPU can be accidentally woken up by events, such as global TLB shotdowns, resulting unnecessary schedule requirements of this vCPU and even causing double scheduling problems, such as the CPU initiating the TLB shotdown command should busy-wait until its all recipient CPUs acknowledge the IPI for the sake of TLB consistency.

Another way to realize the ballooning is to use the

CPU hotplug mechanism provided in existing OSes. However, there is no need to send commands to the underlying bus to actually offline the victim vCPU. When a vCPU is set offline, its runqueue and the events registered on it will be cleared; all hardware interrupts will be masked for it; and it will be masked as invisible to the OS. When a vCPU is set online, it will be set available to the OS again. Its runqueue will be activated and the interrupt mask will be cleared. Although this method seems more complicated than the "HLT" solution, it can be realized by reusing most functions provided by CPU hotplug mechanism and the offline vCPU is unavailable to the guest OS, which will not be the target of IPIs or other events. However, as the hot-plug operation is relatively heavyweight compared to "HLT", a more lightweight approach to hotplugging a vCPU is necessary to reduce the ballooning overhead. Fortunately, recent research has provided operating system support for fast CPU hotplugging, with a reduction of latency from hundreds of milliseconds to several microseconds [12]. This can greatly help reduce the cost of VCPU ballooning.

VCPU-Bal Scheduler: VCPU-Bal scheduler is responsible to schedule vCPUs according to VCPU-Bal manager's schedule decision. It can be built upon the vanilla hypervisor scheduler by simply pinning each vCPU on its dedicated pCPUs. As each pCPU will usually have only one vCPU scheduled on it, the hypervisor is excluded from scheduling only when the guest VM explicit relinquish the control by yielding the CPU (e.g., calling *hlt*). When there are two or more vCPUs assigned to a pCPU, it will schedule the vCPUs using the vanilla hypervisor scheduler.

Challenges and Solutions: One problem VCPU-Bal faces is that most applications do not take CPU hotplug into design consideration and assume that the number of available CPUs does not change during execution. If the number of vCPUs changes dynamically, such applications may not make use of all available CPUs when the number of vCPUs is increased or they have too many threads when vCPUs are decreased. Here, we believe VCPU ballooning does not happen frequently, as the creation, deletion and migration of a VM do not happen frequently in multi-tenant clouds. Thus, we depend on application itself to adapt to the number of vCPU changes. We can categorize most applications into two types: 1) server applications that will spawn more threads than available CPUs (e.g., web server, database); and 2) computational applications that will check the number of available CPUs before its execution and spawn appropriate number of threads to make good use of CPU resources (e.g., PARSEC). For applications from the first category, they are usually long running and spawn more threads than CPUs in case of asynchronous I/O. They can tolerant changes with the number of vCPUs as when vCPUs decreases they can act as usual and when vCPUs increases they still have more threads than available vCPUs. For applications from the latter one, they are usually short running and spawns as many threads as CPUs. When the number of vCPUs changes, the mismatch between the number of threads and vCPUs will not last long.

The other problem VCPU-Bal faces is that N_{pCPU} may be a fraction. In the ideal cases, N_{pCPU} of each VM can be an integer and it is possible to find a one-on-one mapping between vCPUs and pCPUs. In such cases, only the process scheduler in guest OS is needed to do the scheduling, which eliminates the double scheduling problem. On other cases, when N_{pCPU} of a VM is a fraction, where there will be more than one vCPUs assigned to a pCPU, VCPU-Bal can still optimize the overall performance by minimizing the number of runnable vCPUs in collaboration with other existing schedule schemes. For example, for balanced scheduling, with less vCPUs in each VM, a lock holder vCPU is less likely to be preempted. For co-scheduling, there will be less CPU fragments, as it is easier to find a time slice for a VM with less vCPUs.

4 Evaluation

To illustrate the potential benefit of VCPU-Bal, we simulate VCPU-Bal by assigning an optimal amount of vCPUs and comparing its performance with the CFS scheduler on KVM and the credit scheduler on Xen as well as an affinity-based scheduler. The affinity-based scheduling has similar performance of balanced scheduling when vCPUs from the same VM are pinned on different pCPUs. All evaluations were conducted on an Intel machine with two 1.87 Ghz Six-Core Intel Xeon E7 chips equipped with 32 Gbyte memory. We use Debian GNU/Linux 6.0, Xen 4.1.2 and the management VM with Linux kernel version 3.2.6. The KVM version is kvm-qemu 1.2.0 with the host VM kernel version 3.2.6.

We launched two guest VMs using hardware-assisted virtualization each configured with 8 Gbyte memory. Both VMs have the same weight. When evaluating the CFS and credit schedulers, both VMs are configured with 12 vCPUs. For the affinity-based scheduler, both VMs are also configured with 12 vCPUs with each vCPU from the same VM pinned on a different pCPU. When evaluating VCPU-Bal, each VM is configured with 6 vCPUs (each is pinned on a different pCPU). We use four applications (canneal, dedup, swaptions, streamcluster) from PARSEC with the native input set and two applications (histogram and wordcount) from Phoenix with the input size of 2 GByte to compare the performance of different schedulers.

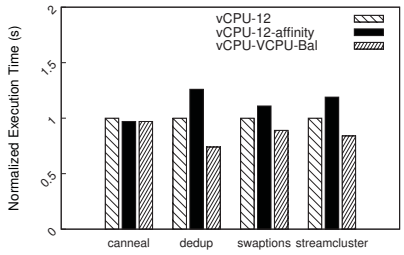

Figure 6: *Performance of four applications from PARSEC on Xen.*

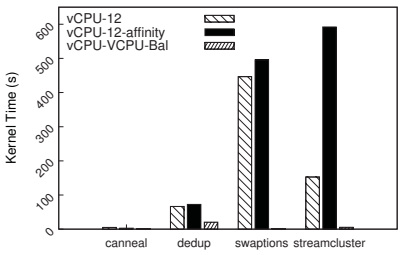

Figure 7: *Kernel time of four applications from PARSEC on Xen.*

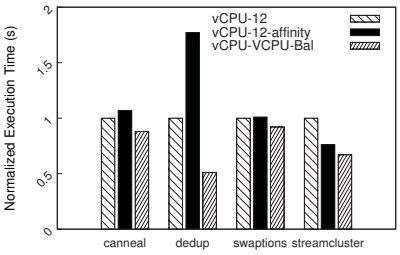

Figure 8: *Performance of four applications from PARSEC on KVM.*

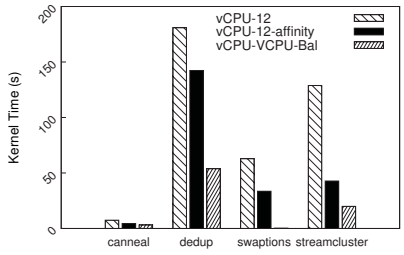

Figure 9: *Kernel time of four applications from PARSEC on KVM.*

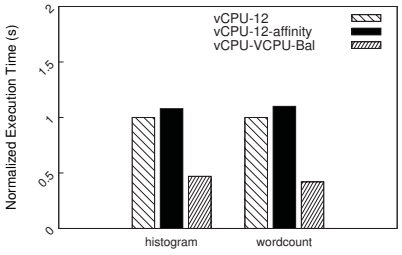

Figure 10: *Performance of histogram and wordcount on Xen.*

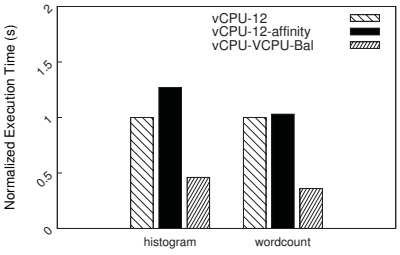

Figure 11: *Performance of histogram and wordcount on KVM.*

PARSEC applications: Figure 6 shows the normalized execution time of four applications on Xen using the credit scheduler, affinity-based scheduler and VCPU-Bal. It can be seen that VCPU-Bal improves the performance of dedup, swaptions and streamcluster by up to 25.9% compared to credit scheduler with an average of 17.6% speedup. This is mainly due to the reduced execution time spent in kernel for synchronization operations, as shown in Figure 7. VCPU-Bal reduces the kernel time of these three applications by 3.3X, 458.1X and 27.9X respectively over credit scheduler. For canneal, as it employs lock-free synchronizations for its synchronization strategy [5], it does not suffer from double scheduling problem in Xen. Thus, VCPU-Bal introduces only a small amount of performance improvement for it. On the other side, affinity-based scheduler introduces more performance overhead than improvements for three out of four applications over credit scheduler.

Figure 8 shows the normalized execution time of four applications on KVM using CFS, affinity-based scheduler and VCPU-Bal. It can be seem that VCPU-Bal improve the application performance by up to 49.2% compared to CFS with an average of 25.5% speedup. The performance improvement of dedup, swaptions and streamcluster are mainly due to the reduced execution time spent in kernel for synchronization operations as shown in Figure 9. VCPU-Bal reduces the kernel time of these three applications by 6.5X, 241.8X and 6.5X respectively. While, for canneal, as it encounter the scalability problem when the number of underlying cores exceeds 6, VCPU-Bal improves its performance by helping it avoid the scalability bottleneck. On the other side, affinity-based scheduler can only improve the performance of streamcluster compared to CFS, but reduce the performance of other three applications.

Histogram and Wordcount:Figure 10 and Figure 11 shows the normalized execution time of histogram and wordcount on Xen and KVM using vanilla scheduler (credit scheduler for Xen and CFS for KVM), affinity-based scheduler and VCPU-Bal. VCPU-Bal reduces the execution time of histogram and wordcount by 52.7% and 57.9% respectively compared to credit scheduler on Xen and reduces execution time of histogram and wordcount by 54.4% and 63.8% respectively compared to CFS on KVM. The performance improvement is due to two reasons: 1) the execution time of histogram and wordcount spent in kernel is reduced by 3.0X and 10.4X respectively on Xen and 4.0X and 7.3X respectively on KVM; and 2) VCPU-Bal helps both histogram and wordcount avoid reaching the scalability bottlenecks, as they perform worse with 12 cores than 6 cores on both Xen and KVM, as shown in Figure 2. On the other side, affinity-based scheduler cannot improve the performance of histogram and wordcount on Xen and KVM.

Discussion: It should be acknowledged that the above results are still quite preliminary yet. On one hand, the startup cost of CPU hotplugs is not included when running the parallel workloads. In our future work, we

plan to design and implement a lightweight CPU hotplug mechanism to let the OS be friendly to VCPU-Bal. On the other hand, we currently only test the performance on a 12-core machine, where the performance gap in a larger scale multicore machine would be larger as the parallel workload will experience more severe scalability problem.

5 Related Work

In the past, lots of work tries to mitigate the semantic gap between multiprogrammed parallel applications and the OS scheduler [10, 11, 3] that is similar to the semantic gap between VM and VM scheduler. However, the interface between VM and hypervisor is more transparent than that between application and OS. Further, there exists a double scheduling problem in virtual environments. Scheduler activation [3] tries to mitigate this semantic gap by allowing user-level application to positively require or delete CPUs assigned to it. Although VCPU-Bal can take the same idea of scheduler activations by letting the guest VM require or delete its CPUs positively, there are three problems: 1) as most applications are not designed to explicitly require or delete CPUs from the OS, it is hard for a VM to decide whether it needs to increase or decrease its vCPUs; 2) the schedule activation interfaces are too complex to integrate into virtual environments; and most importantly 3) the computing resources a guest VM get is decided by its service level agreement (SLA) not its requirement. Thus, VCPU-Bal let the hypervisor to decide the pCPU assignment.

The semantic gap between the OS scheduler and the hypervisor scheduler introduces the double scheduling problem. Uhlig et al. [16] identifies this problem as the lock-holder preemption problem. They proposed a locking-aware scheduler to mitigate the semantic gap, which requires the guest OS to provide hits on whether a vCPU is holding a spinlock. However, this technique requires a guest OS being full knowledge of all lock-holders in kernel and user space which requires argumenting the guest OS as well as user applications which may not be feasible in commodity OSes. Other work tries to mitigate the problem without the intervention of the guest OS through co-scheduling [2, 17], balance scheduling [15] and demand-based scheduling [9]. However, these approaches either cause other problems or do not solve the problem completely.

6 Conclusion and Future Work

This paper argued to avoid hypervisor scheduling as much as possible by dynamically adjusting the amount of vCPUs to a guest VM according to its weight and available physical CPUs. Experiments using simulated scenarios confirmed the potential benefits of such an approach. In our future work, we plan to fully implement the approach on both Xen and KVM and evaluate the performance benefit in a more realistic setting.

References

[1] Kvm. kernel based virtual machine. http://www.linux-kvm.org/.

[2] The CPU Scheduler in VMware ESX 4.1. http://www.vmware.com/files/pdf/techpaper/VMW_vSphere41_cpu_schedule_ESX.pdf.

[3] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 53–79.

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proc. SOSP* (2003).

[5] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The parsec benchmark suite: Characterization and architectural implications. In *Proc. PACT* (2008).

[6] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of linux scalability to many cores. In *Proc. OSDI* (2010).

[7] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scalable address spaces using rcu balanced trees. In *Proc. ASPLOS* (2012).

[8] HENNESSY, J. L., AND PATTERSON, D. A. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011.

[9] KIM, H., KIM, S., JEONG, J., LEE, J., AND MAENG, S. Demand-based coordinated scheduling for smp vms. In *Proc. ASPLOS* (2013).

[10] KONTOTHANASSIS, L. I., WISNIEWSKI, R. W., AND SCOTT, M. L. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems 15*, 1 (1997), 3–40.

[11] OUSTERHOUT, J. K. Scheduling techniques for concurrent systems. In *Proc. ICDCS* (1982), pp. 22–30.

[12] PANNEERSELVAM, S., AND SWIFT, M. M. Chameleon: operating system support for dynamic processors. In *Proc. ASPLOS* (2012).

[13] SONG, X., CHEN, H., CHEN, R., WANG, Y., AND ZANG, B. A case for scaling applications to many-core with os clustering. In *Proc. EuroSys* (2011).

[14] SONG, X., CHEN, H., AND ZANG, B. Characterizing the performance and scalability of many-core applications on virtualized platforms. Tech. rep., 2011.

[15] SUKWONG, O., AND KIM, H. S. Is co-scheduling too expensive for smp vms? In *Proc. Eurosys* (2011).

[16] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards scalable multiprocessor virtual machines. In *Proc. Virtual Machine Research and Technology Symposium* (2004).

[17] WENG, C., WANG, Z., LI, M., AND LU, X. The hybrid scheduling framework for virtual machine systems. In *Proc. VEE* (2009).