Is Co-scheduling Too Expensive for SMP VMs?

Orathai Sukwong
Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA, USA
osukwong@ece.cmu.edu

Hyong S. Kim
Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA, USA
kim@ece.cmu.edu

Abstract

Symmetric multiprocessing (SMP) virtual machines (VMs) allow users to take advantage of a multiprocessor infrastructure. Despite the advantage, SMP VMs can cause synchronization latency to increase significantly, depending on task scheduling. In this paper, we show that even if a SMP VM runs non-concurrent applications, the synchronization latency problem can still occur due to synchronization in the VM kernel.

Our experiments show that both of the widely used open source hypervisors, Xen and KVM, with the default schedulers are susceptible to the synchronization latency problem. To remediate this problem, previous works propose a co-scheduling solution where virtual CPUs (vCPUs) of a SMP VM are scheduled simultaneously. However, the co-scheduling approach can cause CPU fragmentation that reduces CPU utilization, priority inversion that degrades I/O performance, and execution delay, leading to deployment impediment. We propose a *balance scheduling* algorithm which simply balances vCPU siblings on different physical CPUs without forcing the vCPUs to be scheduled simultaneously. Balance scheduling can achieve similar or (up to 8%) better application performance than co-scheduling without the co-scheduling drawbacks, thereby benefiting various SMP VMs. The evaluation is thoroughly conducted against both concurrent and non-concurrent applications with CPU-bound, I/O-bound, and network-bound workloads in KVM. For empirical comparison, we also implement the co-scheduling algorithm on top of KVM's Completely Fair Scheduler (CFS). Compared to the synchronization-unaware CFS, balance scheduling can significantly improve application performance in a SMP VM (e.g. reduce the average TPC-W response time by up to 85%).

Categories and Subject Descriptors D.4.1 [**Process Management**]: Scheduling.

General Terms Algorithms, Experimentation, Performance.

Keywords Virtualization, Synchronization.

1. Introduction

Virtualization provides a flexible computing platform for cloud computing (e.g. Amazon EC2) and server consolidation. It helps to maximize physical resource utilization and simplify system and infrastructure management. Virtualization mainly consists of a software layer between an operating system (OS) and hardware called a *hypervisor*, and a software version of a machine called a *virtual machine* (VM) or *guest*. Examples of hypervisors are VMware ESXi [VMware2010c], Xen [Barham2003] and KVM [KVM2008]. Like a real machine, a VM can run any application, OS or kernel without modifications. A VM can be configured with different hardware settings, such as the number of virtual CPUs (vCPUs) and the size of hard disk and memory. A VM with multiple vCPUs, which behave identically, is called a symmetric multiprocessing (SMP) VM. As a rule of thumb, a SMP VM should not have more vCPUs than available physical CPUs [VMware2010a], a practice followed in this paper.

Virtualization can cause problems which do not exist in a non-virtualized environment. For instance, a spinlock (used for kernel synchronization) in a non-virtualized environment is assumed to be held for a short period of time and does not get preempted. But a spinlock held by a VM can be preempted due to vCPU preemption [Uhlig2004], vastly increasing synchronization latency and potentially blocking the progress of other vCPUs waiting to acquire the same lock. Combined with preemptible synchronization in a concurrent application inside a SMP VM, the synchronization latency problem in the VM can be severe, resulting in significant performance degradation.

Deducing from the results of our experiments (Section 5.3.1), the default schedulers of several current hypervisors (Xen and KVM) still seem to be unaware of the synchronization latency problem. To reduce synchronization latency, previous works propose a co-scheduling solution where vCPUs of a SMP VM are scheduled simultaneously. Recently proposed systems [Weng2009, Bai2010] selectively apply co-scheduling only to SMP VMs running concurrent applications because a non-concurrent application has no application synchronization and thus may not significantly benefit from co-scheduling. In this paper, we show that a SMP VM running a non-concurrent application, such as a single-threaded, synchronization-free and I/O-bound task, can also benefit from co-scheduling due to synchronization in the guest OS.

Nonetheless, co-scheduling can still cause CPU fragmentation, priority inversion [Lee1997] and execution delay. These drawbacks can hinder deployment of various SMP VMs. For example, VMware's co-scheduling solution [VMware2010b] tries to maintain synchronous progress of vCPU siblings by deferring the advanced vCPUs until the slower ones catch up. This can be too rigorous for a SMP VM running a minimal synchronization application, as shown in Section 5.3.1.

We propose the *balance scheduling* algorithm which provides application performance similarly to or better than that of the traditional co-scheduling approach without the co-scheduling drawbacks. The concept of balance scheduling is simple – balancing vCPU siblings on different CPUs without precisely scheduling the vCPUs at the same time. This is easily accomplished by dynamically setting CPU affinity of vCPUs so that no two vCPU siblings are in the same CPU runqueue. We implement the balance scheduling algorithm on top of KVM's scheduler (Completely Fair Scheduler – CFS [Molnar2007]) in the Linux kernel.

For empirical comparisons, we also implement a co-scheduling algorithm, called dynamic time-slice (DT) co-scheduling, based on CFS to avoid the impact of different resource optimizations found in different hypervisors. DT co-scheduling should perform similarly to classic co-scheduling, despite the differences in implementation. Our co-scheduling implementation is based on CFS with dynamic time slice, while the previous implementation relies on a scheduler with static time slice. Compared to the co-scheduling algorithm [Ousterhout1982], DT co-scheduling has less computational complexity and does not incur CPU fragmentation. The expected synchronization latency of the DT co-scheduling algorithm is theoretically the lower-bound of the co-scheduling algorithm (details in Section 5.2.2).

Because VMs can run many types of programs, we extensively evaluate the scheduling algorithms against both non-concurrent and concurrent applications with various degrees of synchronization. We also test them with different workloads (CPU-bound, I/O-bound and network-bound). The empirical results show that balance scheduling can significantly improve application performance (e.g. reducing the average TPC-W response time by up to 85% compared to CFS). Balance scheduling also yield similar or better application performance (e.g. up to 8% higher X264 throughput) than co-scheduling without the drawbacks of co-scheduling, thus benefiting many SMP VMs.

We also evaluate balance scheduling against affinity-based scheduling [Vaddagiri2009]. Both balance scheduling and affinity-based scheduling similarly manipulate each vCPUs' CPU affinity. Unlike affinity-based scheduling (static configuration), balance scheduling can potentially adapt to load changes. Balance scheduling dynamically sets CPU affinity before a scheduler assigns a runqueue to a vCPU, allowing the vCPU to run on the least-loaded CPU where there is no vCPU siblings. Load is measured as the number of runnable tasks in a per-CPU runqueue.

This paper makes the following contributions:

- We show that a SMP VM running a non-concurrent application can also suffer from the synchronization latency problem due to synchronization in the guest OS.
- We propose the *balance scheduling* algorithm and present its performance analysis. We also compare the computational complexity of the balance scheduling and co-scheduling algorithms.
- We implement the balance scheduling and co-scheduling algorithms on top of CFS for empirical comparison. We theoretically and empirically show that our co-scheduling implementation is a refined variation of classic co-scheduling.
- We perform a thorough evaluation on the balance scheduling, co-scheduling and affinity-based scheduling algorithms, in addition to CFS.

The rest of this paper is organized as follows. Section 2 elaborates on the synchronization latency problem. Section 3 describes the co-scheduling approach. Section 4 presents our proposed balance scheduling algorithm. Section 5 discusses the evaluation. Section 6 describes related work. Section 7 is the conclusion.

2. Synchronization in SMP VMs

2.1 Lock Primitive

In a concurrent program, a lock primitive is used to provide synchronization among concurrent threads. Different OSes may support different types of locking. Typically there are two major types of lock primitives [Fischer2005].

Semaphore/Mutex (non-busy-wait). The thread that is waiting for this lock can be blocked and go to sleep, allowing the scheduler to context switch to another runnable thread. This lock primitive is normally used in applications where synchronization may take long to complete (e.g. waiting to receive a network packet).

Spinlock (busy-wait). A spinlock is used when synchronization is expected to take only a short amount of time. Thus, it is inefficient to perform context switching. The lock-waiter thread keeps spinning CPU cycles until it successfully acquires the lock. Spinlocks are simple and usually used in kernel. An OS kernel typically does not preempt a kernel thread which is holding a spinlock. With virtualization, a spinlock in a VM may be preempted due to vCPU preemption.

2.2 Synchronization latency

Synchronization latency is the amount of time it takes a thread to successfully acquire a lock. Synchronization latency in a SMP VM is simply the lock latency experienced by vCPUs of a VM. There are two causes of synchronization latency: task scheduling and preemption or blocking. The hypervisor scheduler can preempt vCPUs at any time, regardless of what they are executing.

Synchronization latency depends on task scheduling when two or more vCPUs simultaneously want the same lock and this lock is blocked or preempted, as shown in Figure 1B. Otherwise, the latency is equal to or less than the amount of time it takes the lock-holder thread to finish synchronization and release the lock (T_H) as shown in Figure 1A.

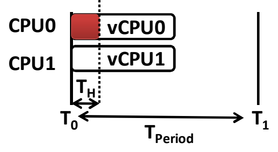

Figure 1A. Synchronization latency without preemption.

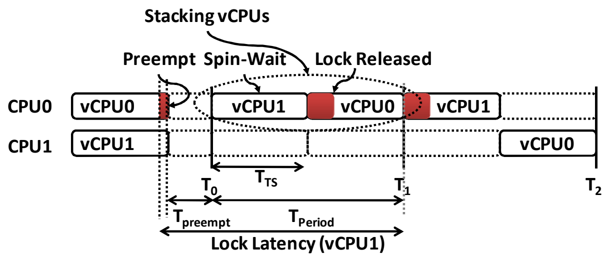

Figure 1B. Synchronization latency with preemption.

Normally, a hypervisor scheduler, such as CFS or Xen's Credit Scheduler [Yaron2007], allows vCPUs to be scheduled to run on any CPU. It is possible that the lock-waiter thread can be scheduled before the lock-holder thread when a lock-holder thread is preempted, as shown in Figure 1B. We call this task scheduling situation *vCPU stacking*. In the worst case scenario, vCPU1 has to wait $T_{preempt}$ + T_{period}, as opposed to $T_{preempt}$ + T_H. $T_{preempt}$ is measured from the time that vCPU0 is preempted until one of these vCPUs is re-scheduled, and T_{TS} is a time slice of a vCPU, assuming all time slices are the same. Normally, T_H is in the order of microseconds and T_{period} is in the order of milliseconds. The worst case latency may increase to several milliseconds. When waiting for a spinlock, many CPU cycles will also be wasted.

3. Co-scheduling

Ousterhout proposed a co-scheduling algorithm [Ousterhout1982] that schedules a set of concurrent threads simultaneously to reduce synchronization latency. Several previous works [VMware2008, Weng2009, Bai2010] apply co-scheduling to SMP VMs. As shown in Figure 2, co-scheduling can significantly reduce synchronization latency (from $T_{preempt}$ + T_{period} to $T_{preempt}$ + $T_{H'}$). Note that co-scheduling cannot prevent preemption and eliminate $T_{preempt}$, as shown in Figure 2.

A simple way to co-schedule a set of tasks is finding a time slice that has a sufficient number of available physical CPUs to run all tasks, assuming every time slice has the same size. These tasks are delayed until such a time slice is found. This approach causes CPU fragmentation and priority inversion [Lee1997, VMware2008].

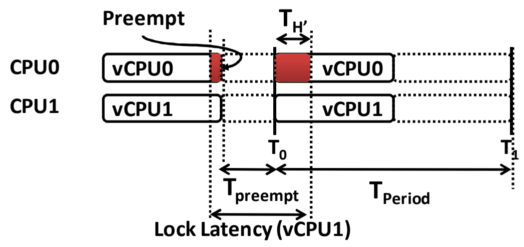

Figure 2. Synchronization latency with co-scheduling.

3.1 CPU fragmentation

As shown in Figure 3, with the co-scheduling approach, vCPU0 and vCPU1 cannot be scheduled until T_1, although both become runnable at T_0 because there is only one CPU idle at T_0. This is called *CPU fragmentation*, which can reduce CPU utilization and also delay the vCPU execution.

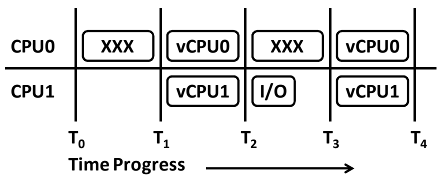

Figure 3. CPU fragmentation in co-scheduling.

3.2 Priority Inversion

Priority inversion is where a higher priority task is scheduled after a lower priority task. For example, an I/O-bound job is given a priority to run whenever it is ready. However, it cannot run because all CPUs are allocated to the co-scheduled tasks. This problem can adversely affect interactive or I/O-bound jobs, and under-utilize other resources (e.g. disks). As seen in Figure 3, when an I/O-bound job is ready between T_0 and T_1, the I/O job has to wait until T_2 because the scheduler already assigns the slot T_1 on both CPUs to vCPU0 and vCPU1, given that both vCPUs are runnable since T_0. The longer the time slice of vCPU1 (T_2-T_1), the longer the disk sits idle and the higher the I/O latency, for example.

4. Balance Scheduling

4.1 Description

To alleviate the synchronization latency problem, we propose the *balance scheduling* algorithm which balances vCPU siblings on different physical CPUs without precisely scheduling the vCPUs simultaneously. It is simply achieved by dynamically setting CPU affinity of vCPUs so that no two vCPU siblings are in the same CPU's runqueue. Unlike co-scheduling, it does not incur CPU fragmentation, priority inversion or execution delay.

4.2 Severity of the vCPU-Stacking Problem

Balance scheduling can be considered a probabilistic type of co-scheduling. It increases the chance of vCPU siblings being scheduled simultaneously by reducing the likelihood of the vCPU-stacking situation (described in Section 2.2). To estimate the probability of the vCPU-stacking occurrence, we empirically measure how often KVM's CFS scheduler places vCPU siblings in the same CPU's runqueue when running one or more CPU-intensive SMP VMs. We run three experiments: one, two and three four-vCPU VMs with our CPU-bound workload (described in Section 5.1) in a four-CPU host. Each CPU runqueue is examined every 700 microseconds to see what tasks are in the queues by inspecting */proc/sched_debug*. We then count the number of samples where the runqueue has two, three and four vCPU siblings in the same runqueue.

As shown in Table 1, the risk of vCPU siblings being stacked grows as the number of VMs increases (the runqueue size also increases). When only one VM is running in the host, the chance that more than one vCPU sibling will be running sequentially is not significant (~6%). When the number of VMs increases to two, the chance substantially increases to 43.13%. Stacking vCPUs can undermine an illusion of synchronous progress of vCPUs, expected from the guest OS [VMware2010]. Without this illusion, the guest OS may malfunction or panic.

# VMs	# vCPUs in the same runqueue			> 1 vCPU in the same runqueue
	2	3	4	
1	5.518%	0.045%	0.001%	5.564%
2	31.903%	10.717%	0.507%	43.127%
3	29.730%	12.091%	4.111%	45.932%

Table 1. The probability of vCPU-stacking.

4.3 Computational Complexity Analysis

We compare the computational complexity of balance scheduling and co-scheduling. Assuming each time slice is the same, the pseudo code of the co-scheduling algorithm for scheduling *k* vCPUs of a SMP VM is described in Algorithm 1. According to the pseudo code, the computational complexity of the co-scheduling algorithm is O(*NR*) where *N* is the number of physical CPUs and *R* is the runqueue size.

The pseudo code of the balance scheduling algorithm is shown in Algorithm 2. The computational complexity of balance scheduling is O(*N*) because the number of vCPUs is always less than or equal to the number of CPUs. By fixing *N*, the complexity of balance scheduling and co-scheduling becomes O(1) and O(*R*) respectively. Therefore, balance scheduling has less computational complexity than co-scheduling.

Algorithm 1: Co-scheduling

```
for each time slot i
    available_cpus ← 0
    for each CPU j
        if time slot i on CPU j is idle
            increment available_cpus by 1
        end if
        if available_cpus ≥ k
            assign vCPUs to available CPUs
            return
        end if
    end for each
end for each
```

Algorithm 2: Balance Scheduling

```
all_cpus ← set of all physical CPUs
if (task T has not been assigned a runqueue)
    and (task T is a vCPU)
    VMID ← Parent PID of task T
    used_cpus ← {}
    for each vCPU v of VMID
        add CPU that v is on in used_cpus
    end for each
    CPUS of task T ← all_cpus – used_cpus
end if
```

4.4 Performance Analysis

We theoretically show the synchronization latency improvement in balance scheduling compared to CFS, with the different numbers of available physical CPUs. We also estimate the impact on application performance.

As mentioned earlier, task scheduling can affect the lock latency when the lock is needed by two or more vCPUs and also preempted. We calculate the expected lock latency of balance scheduling and CFS using the equations in Appendix A. The following assumptions are made: each task in a runqueue has the same weight, each runqueue has the same size, the average lock holding time is one microsecond and two vCPUs need to acquire the same lock simultaneously. As shown in Figure 4A, the expected lock latency increases as the runqueue size grows. Intuitively, when the runqueue size is one (only one vCPU in the runqueue), balance scheduling and CFS are practically the same. The expected lock latency also lowers as the number of CPUs increases due to decrease in the vCPU-stacking probability. Balance scheduling can reduce the expected latency more than CFS as balance scheduling avoids vCPU stacking. As shown in Figure 4B, balance scheduling can significantly improve the expected lock latency compared to CFS (more than 14.4% for four CPUs), when the runqueue size is less than six. The experiments in Section 5.3.6 show that the average runqueue size is practically about 4-6, even if a host has many threads.

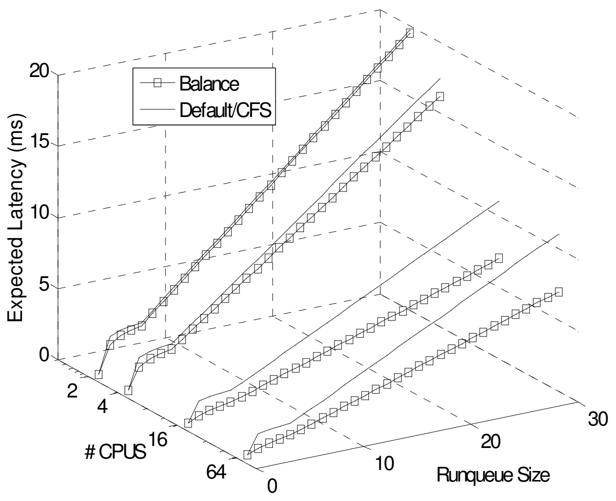

Figure 4A. The expected lock latency in balance scheduling and CFS.

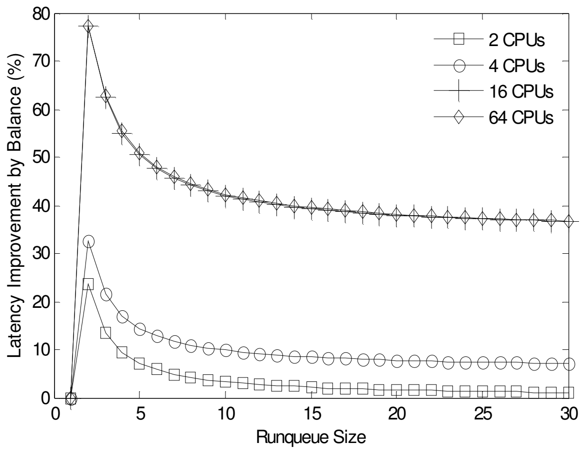

Figure 4B. The expected lock latency improvement in balance scheduling compared to CFS.

Quantifying variation in application performance due to the change in synchronization latency is difficult. Conceptually, the impact of lock latency on application performance should be similar to a step function. As long as lock latency does not exceed a threshold leading to an operation timeout, a change in application performance should appear insignificant. Otherwise, the change can be substantial. For example, an application has to send five TCP packets. We assume the application takes 100 locks per second and the TCP average response time is 10 milliseconds or greater. TCP transmission timeout is 200 milliseconds by default in Linux. If the lock latency increases from one to two microseconds without any TCP retransmission, the response time of each packet will be increased by at most 20 microseconds, which is 0.2% or less increase in the average response times. But if the lock latency exceeds the threshold causing TCP timeout and a retransmission, then the average response time becomes 50.004 milliseconds ((50 + 200 + 0.02)/5) or greater, which is a 400% increase in the average response time. Balance scheduling is designed to reduce the likelihood that the lock latency becomes exceedingly high. High lock latency usually occurs when a scheduler stacks vCPUs. As shown in Section 5.3.2, balance scheduling causes no TCP retransmission in TPC-W, but CFS does.

5. Evaluation

We extensively evaluate how balance scheduling, co-scheduling, affinity-based scheduling and CFS (KVM's default scheduler) improve application performance. The experiments are conducted with applications ranging from single-threaded and synchronization-free applications to concurrent applications with different degrees of synchronization. The applications also carry different types of workloads (CPU-bound, I/O-bound and network-bound) in various scenarios (combinations of SMP and non-SMP VMs run concurrently in the host).

5.1 Experimental Setup

All experiments run on a physical machine with Intel Core2 Quad CPU Q8400 2.66GHz and 4 GB of RAM with 1Gbit Network card. The physical host runs Fedora Linux kernel 2.6.33 with QEMU 0.11.0. The guest OSes are either Fedora 12 or 13. The selected applications are Pi, HackBench, X.264, Compile, TPC-W, Dell DVDstore, BZip2, Tar, TTCP, Ping, Bonnie++, our synthesized disk and CPU workloads, and our multiple-independent-process workload. Where relevant we use the fourth extended file system (ext4) [Mathur2007] in the experiments.

Pi [Yee2010] is a multi-threaded and CPU-bound program entirely fitting in the memory. It calculates 100,000,000 digits of pi using the Chudnovsky Formula. We use the computing time as a performance metric.

HackBench [HackBench2008] is a multi-threaded program measuring Unix-socket (or pipe) performance. We run HackBench using four threads with 10,000 loops. The completion time (seconds) is used as a metric.

X.264 [Phoronix2010] is a multi-threaded and CPU-bound application, which performs H.264/AVC video encoding. It reports the average throughput in frames per second.

Compile is a compilation test on *libvirt* library using *rpmbuild* tool (a multi-process program). We measure the amount of time it takes to compile (in seconds).

TPC-W [TPC2000] is a transactional web benchmark using multiple web interactions to simulate a retail store's activities. We use Apache HTTP server version 2.2.14 for the proxy server, Tomcat5 version 5.5.27 for the web server and MySQL version 5.1.44 for the database server. These servers are multi-threaded applications.

Dell DVD Store (DVDstore) [Dell2007] is an open source simulation of an online ecommerce site. We use MySQL server 5.1.45 and Apache HTTP Server 2.2.15 for the database and web servers running in the same VM. 100 clients with five-second thinking time concurrently connect from another physical machine located on the same network for three minutes. The average response time is used as a performance metric.

BZip2 and Tar are single-thread data compressor programs. We use *BZip2* to compress a 460 MB file and use Tar to decompress a 1.1GB file (*Untar*). We measure the time it takes to complete the task.

TTCP [TTCP1996] is a single-thread socket-based application that measures TCP and UDP throughput (kB/s) between two systems.

Ping is a single-thread and network-bound program that sends ping packets to another machine located on the same network.

Bonnie++ [Coker2001] is an I/O benchmark measuring hard drive and file system performance. By default, it creates one thread for each test, except the seek test that uses three threads.

Our disk-bound workload is a single-thread and disk-I/O-bound program of our own creation that sequentially creates, writes and deletes small files on a local disk. We measure the time it takes to finish the job.

Our CPU-bound workload is a single-thread and CPU-bound program primarily consuming only CPU resources with minimal memory footprint and I/O usage. It runs infinite loops with simple additions.

Our multiple-independent-process workload consists of multiple processes that independently run a finite number of loops with simple arithmetic calculations.

The host CPU utilization is collected using *dstat* [Wieërs2010]. The I/O statistics are gathered from */sys/block/vda/stat*. We record the runqueue size of each physical CPU by sampling */proc/sched_debug* every second. The sample average is the average runqueue size. We quantify application performance improvement by calculating a performance speed-up. The speed-up metric of a scheduling algorithm ($SpeedUp_{SCHED}$) is computed using the following equation, where $Perf_{SCHED}$ is the application performance result achieved by the scheduling algorithm and $Perf_{CFS}$ is the application performance result achieved by CFS.

$$SpeedUp_{SCHED} = \frac{Perf_{SCHED} - Perf_{CFS}}{Perf_{CFS}}$$

We create seven experiments for the evaluation. Experiment 1 shows the degree of the synchronization problem in several hypervisors (Xen, VMware and KVM). To eliminate a different resource optimization factor in different hypervisors, we use only KVM hypervisor for the rest of the experiments. Experiment 2 and 3 quantify synchronization latency improvement and efficiency of CPU resources by each scheduling algorithm respectively. Experiment 4 measures performance improvement in both concurrent and non-concurrent applications. Experiment 5 assesses the scalability of the scheduling algorithms. Experiment 6 determines the scheduling performance and CPU runqueue sizes, when the machine hosts many VMs. Experiment 7 shows the performance of SMP and non-SMP VMs running in the same host.

5.2 Implementation

KVM is seamlessly integrated into the Linux kernel. It has a loadable kernel module providing the core of virtualization, and relies on existing Linux kernel modules for the rest of the functionalities (e.g. a scheduler). In KVM, a VM is a regular Linux process with vCPU processes, which require a modified QEMU for device emulation.

We implement the balance scheduling and co-scheduling algorithms based on CFS. Unlike its predecessors, CFS dynamically calculates a time slice for each runnable task. The time slice is calculated as follows, where N_T is the number of tasks in a runqueue, *MinPeriod* is the minimum period and *MinSlice* is the minimum time slice.

$$MinTasks = \frac{MinPeriod}{MinSlice}$$

$$Period = \begin{cases} MinPeriod & \text{if } N_T \le MinTasks \\ MinSlice \times N_T & \text{if } N_T > MinTasks \end{cases}$$

$$Time\ Slice = \frac{Period}{N_T}$$

In version 2.6.33 of the Linux kernel, by default the minimum time slice is one millisecond, the minimum period is five milliseconds and all tasks in a runqueue have the same weight. The time slice calculation and scheduling decision are made independently on each runqueue (one per CPU). CFS implements a runqueue as a red-black tree [Cormen2001], sorted by each task's *vruntime* (virtual runtime in nanoseconds). The scheduler always selects the task with the smallest *vruntime* to run next.

5.2.1 Balance scheduling

The balance scheduling algorithm can be easily implemented. We modify CFS to dynamically set the *cpus_allowed* field in each vCPU's *task_struct* so that no two vCPU siblings are in the same runqueue. The c*pus_allowed* field indicates a set of CPUs that this task can run on. This *cpus_allowed* setting is done before a runqueue is chosen for a vCPU.

5.2.2 Co-scheduling

The classic co-scheduling algorithm (details in Section 4.3) is designed with a static-time-slice assumption. This design cannot be applied to CFS due to its dynamic time slice calculation. In CFS, the second tasks in the different runqueues may not be scheduled at the same time, for example.

We create our version of co-scheduling, called *dynamic time slice* (*DT*) *co-scheduling*. To schedule vCPUs simultaneously, we first modify CFS so that it never inserts any two vCPU siblings in the same runqueue, like in balance scheduling. We then force the scheduler to schedule all runnable vCPU siblings simultaneously. However, this step only occurs when the scheduler normally selects the first vCPU sibling from a runqueue. As a result, we still preserve fairness among VMs without keeping track of vCPUs' runtime. To force the scheduler to context switch to the chosen vCPU, we call the *resched_cpu* function with the CPU ID. This function sets *TIF_NEED_RESCHED* flag on the current task and then sends an *smp_send_reschedule* inter-processor interrupt (IPI) to the targeted CPU. We modify the *pick_next_entity* function in *sched_fair.c* so that it can choose the targeted vCPU, instead of the lowest *vruntime* task. Unlike the previous co-scheduling approach, our DT co-scheduling algorithm does not incur CPU fragmentation and execution delay. However, DT co-scheduling may shorten the time slice of the current task due to premature preemption and incur additional context switching.

Our DT co-scheduling algorithm is a refined version of the previous co-scheduling algorithm. It has less computational complexity than the previous co-scheduling algorithm (O(N) versus O(NR)). Its expected synchronization latency is the lower-bound of the previous co-scheduling algorithm. The expected synchronization latency of DT co-scheduling is $T_H + T_{INT+CTX}$ where T_H is the lock-holding time and $T_{INT+CTX}$ is the amount of time it takes to send an IPI and perform context switching. Due to CPU fragmentation, the expected latency of the previous co-scheduling algorithm is $T_H + \sum_{i=1}^{\infty} P_i T_{TS}(i-1)$. T_{TS} is a size of time slice. P_i is the probability of having sufficient CPUs to run all vCPU siblings at time slice i and $\sum_{i=1}^{\infty} P_i = 1$. $T_{INT+CTX}$ is normally in the order of microseconds and T_{TS} is in the order of milliseconds. Therefore, $\sum_{i=1}^{\infty} P_i T_{TS}(i-1) \geq T_{TS} > T_{INT+CTX}$. Moreover, the empirical results show that the DT co-scheduling algorithm can improve application performance by up to 6% compared to the previous co-scheduling algorithm. Please see Appendix B for more details. Hence, our DT co-scheduling algorithm should be adequate for the comparative evaluation.

5.2.3 Affinity-based scheduling

We use the *virsh vcpupin* command to modify the CPU affinity of vCPUs. At the beginning of each experiment, we bind each vCPU to a CPU in such a way that the number of vCPUs per physical CPU is relatively the same and vCPU siblings cannot be assigned to the same physical CPU.

5.3 Experimental Results

5.3.1 Experiment 1

Experiment 1 shows the degree of the synchronization problem in several current hypervisors. We run two CPU-intensive workloads: HackBench (intensive synchronization) and the multiple-independent-process workload (no application synchronization) in a four-vCPU VM along with three one-vCPU VMs running our CPU-bound workload.

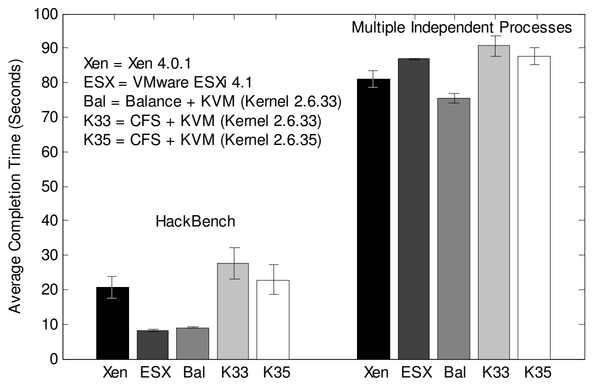

Figure 5. The average completion time with 95% confidence interval assuming the normal distribution.

As shown in Figure 5, Xen (using the Credit scheduler) and KVM (CFS) have higher completion times than balance scheduling on HackBench due to their synchronization-unaware schedulers. They treat all vCPU siblings as independent entities. Although VMware ESXi's scheduler uses a co-scheduling algorithm to mitigate the synchronization problem, their algorithm can be too restrictive for certain applications that barely incur synchronization. As shown in Figure 5, VMware's scheduler has the lowest completion time on HackBench (9.65% less than balance scheduling approach) because VMware's scheduler maintains synchronous progress of vCPU siblings. However, this also causes VMware's scheduler to complete the multiple-independent-process workload (14.88%) slower than the balance scheduling approach. VMware's scheduler stops the advanced vCPUs until the slow vCPUs catch up [VMware2010b], resulting in vCPU-execution delay. Note that we confine the comparison to the CPU-only tests since different hypervisors may have different optimizations on other resources (e.g. network and disk I/O).

We also create a Windows version of HackBench to test on a VM with a different guest OS (i.e. Windows Server 2008). The results are consistent with our findings from the Fedora guest OS. The Windows VM spends 20.23 and 10.46 seconds using CFS/KVM and balance scheduling/KVM respectively, while the Fedora VM spends 20.58 and 9.02 seconds. These results suggest that the balance scheduling approach can benefit other guest OSes than Linux.

5.3.2 Experiment 2

The goal of this experiment is to show the improvement on the synchronization latency by different scheduling algorithms. We run TPC-W benchmark using three four-vCPU VMs for the proxy, web and database servers. The maximum of 250 clients concurrently connect from another physical machine to the proxy server. The average and 90th percentile response times experienced by the clients are reported. We also use *ftrace* [Edge2009] to monitor the amount of time that the vCPUs of the proxy server take to execute the *spin_lock* function, and use *SystemTap* [RedHat2010] to monitor TCP retransmissions in the VMs.

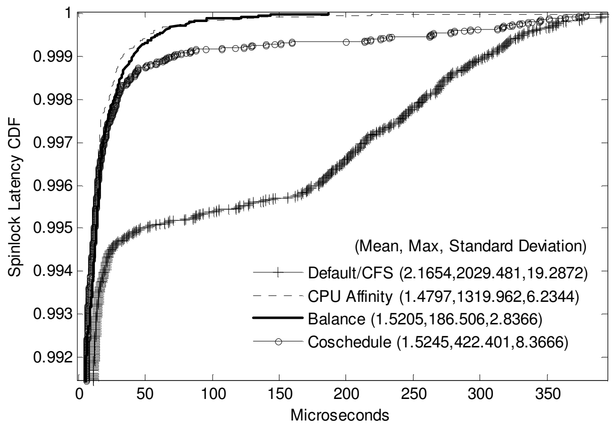

Figure 6A. The spinlock latency CDF of the proxy server.

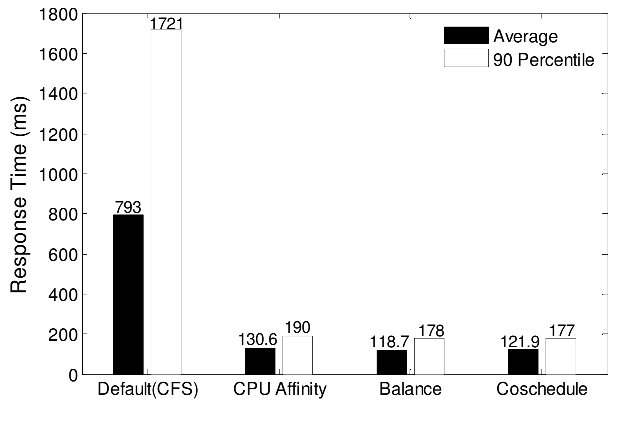

Figure 6B. The response time statistics of TPC-W.

As shown in Figure 6A, balance scheduling, affinity-based scheduling and co-scheduling can similarly improve the average spinlock latency compared to CFS (decreased by 29.78%, 31.67% and 29.60% respectively). The significant increase in the spinlock latency caused by CFS can trigger a TCP timeout leading to TCP retransmissions. The retransmissions in the proxy server can cause disruptions in the subsequent servers (the web and database servers) and eventually affect the overall response time. From the experiment, we find 1,363 retransmissions between the proxy and web servers, 399 retransmissions from the web to database servers, and 38 TCP retransmissions between the clients and the proxy server with CFS, while there is no retransmission with the other scheduling algorithms. These retransmissions severely degrade the TPC-W performance. In balance scheduling, affinity-based scheduling and co-scheduling, the average response time is reduced by 85.04%, 83.53% and 84.62% respectively, as shown in Figure 6B. These results show that balance scheduling can significantly improve the synchronization latency and application performance, compared to CFS. Balance scheduling also performs similarly to co-scheduling (achieving about the same average and 90th percentile response times).

5.3.3 Experiment 3

The synchronization latency problem not only degrades application performance, but also wastes CPU resources due to unnecessary CPU spinning. This experiment shows the improvement in processing efficiency by the different scheduling algorithms. We run the Bonnie++ benchmark in a four-vCPU VM along with a two-vCPU VM running the CPU-bound workload. The two-vCPU VM is used to simulate a background workload. For each I/O test, Bonnie++ reports I/O throughput and CPU utilization in the VM. We use these metrics to calculate throughput per CPU utilization, which is then used to compute the speed-up. This speed-up metric indicates the I/O processing efficiency.

Even though Bonnie++ spawns only a single thread for each test (except the seek test), it can encounter the synchronization problem due to intensive disk I/O processing in the guest OS. Balance scheduling, affinity-based scheduling and co-scheduling can help reduce excessive CPU cycles caused by synchronization-unaware scheduling, thereby having more CPU cycles for useful work. As shown in Figure 7, balance scheduling, affinity-based scheduling and co-scheduling significantly increases the I/O processing efficiency by up to 40%, 45% and 53% for the read operation (SeqInput); 70%, 73% and 63% for the write operation (SeqOutput) and 374%, 439% and 382% for the seek operation respectively. The I/O latency is also improved. As shown in Figure 8, balance scheduling, affinity-based scheduling and co-scheduling reduce the I/O read latency by 48%, 23% and 35%, compared to CFS. The I/O write latency is not improved as much as the read latency due to the disk being a bottleneck. The gain by each scheduling algorithm, excluding CFS, can be varied at each trial depending on cache performance.

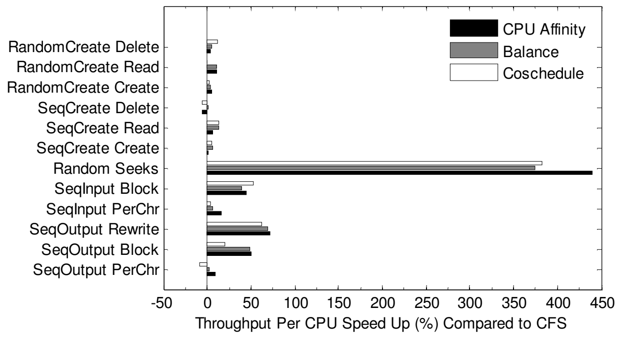

Figure 7. The speed up of I/O throughput per CPU utilization of Bonnie++benchmark.

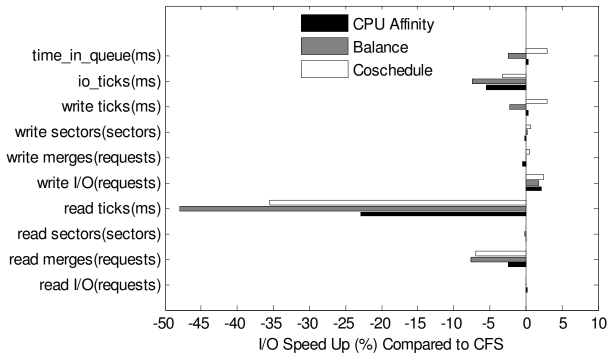

Figure 8. The I/O statistics in the Bonnie VM.

5.3.4 Experiment 4

This experiment shows how much each scheduling algorithm can improve the performance of applications ranging from single-threaded programs without any locking, to multi-threaded programs with different degrees of application synchronization. We also test the algorithms with different types of workloads (CPU-bound, I/O-bound and network-bound). We run two SMP VMs in the host: one four-vCPU VM running an application, except for TTCP, and one two-vCPU VM running our CPU workload. This two-vCPU VM is for simulating a background workload. For TTCP, we run two four-vCPU VMs in the host: one for a TTCP transmitter and the other for a TTCP receiver.

For the multi-threaded applications (Pi, HackBench, X.264, Compile, and DVDstore), affinity-based scheduling, balance scheduling and co-scheduling similarly improve the application performance by up to 85% compared to CFS, as shown in Figure 9. The improvement varies due to the degree of synchronization in the SMP VM. HackBench incurs intensive synchronization due to socket sharing in the guest VM kernel, as opposed to Pi, which incurs a relatively small degree of application synchronization.

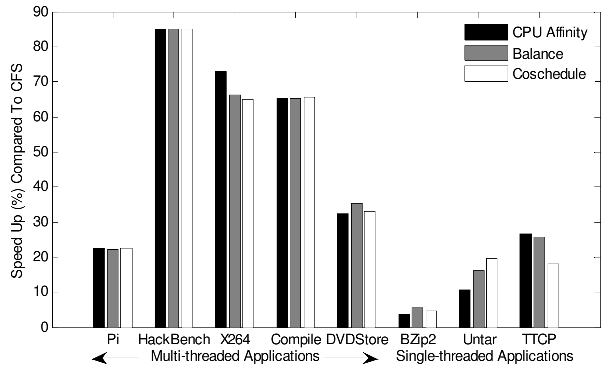

Figure 9. The performance improvement of different applications using affinity-based, balance and co-scheduling.

Modern kernels are capable of servicing multiple applications simultaneously. To understand the impact of kernel synchronization on application performance, we run two independent (synchronization-free) processes of the disk workload in the four-vCPU VM. As shown in Figure 10, balance scheduling, affinity-based scheduling, and co-scheduling reduce the completion time by 35%, 32%, and 31% compared to CFS, respectively, due to file system synchronization. The improvement in file system performance increases I/O aggregation as indicated by the 20% reduction in the average I/O write requests. These results suggest that synchronization can incur in a VM despite running synchronization-free applications. Balance scheduling reduces the completion time by 5% compared to co-scheduling due to additional context switching.

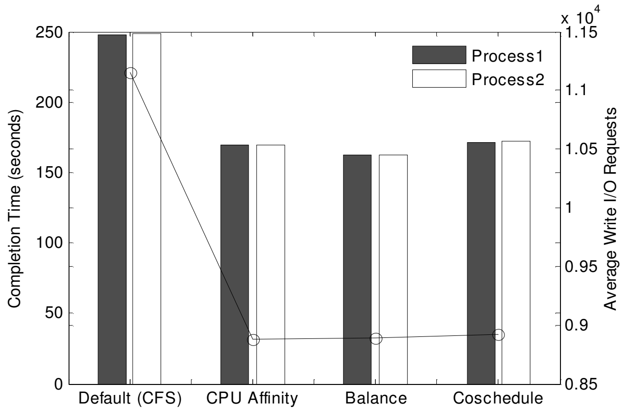

Figure 10. Performance of multiple disk-I/O processes in a SMP VM.

We also run a single-threaded application (no application synchronization) in the VM to understand the effect of synchronization in the guest VM kernel. As shown in Figure 9, balance scheduling, affinity-based scheduling and co-scheduling improve TTCP performance by 26%, 27% and 18%, Untar performance by 16%, 11% and 20%, and BZip2 performance by 6%, 4% and 5% compared to CFS respectively. The improvement depends on the degree of synchronization in the VM kernel. TTCP mainly relies on the guest kernel for network processing, while Untar processes in both user and kernel spaces (for I/O processing) and BZip2 mainly runs in the user space with minimal kernel assistance. For TTCP, balance scheduling has 6% higher TTCP throughput than co-scheduling due to additional context-switching. For Untar and BZip2, the completion times vary over 20 trials because the improvement due to their small degree of kernel synchronization can be outweighed by cache performance.

Overall, the results show that balance scheduling, affinity-based scheduling and co-scheduling can benefit any application that incurs synchronization in either application or kernel inside a SMP VM. The performance improvement depends on the degree of synchronization in the VM. Balance scheduling can improve application performance up to 6% more than co-scheduling due to additional context-switching.

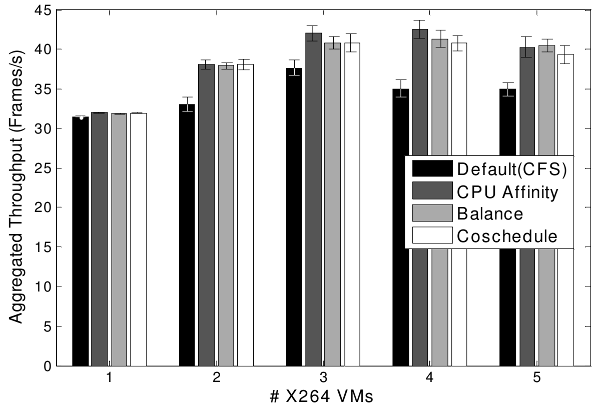

Figure 11A. The aggregated throughput of all X.264 VMs with 95% confidence interval.

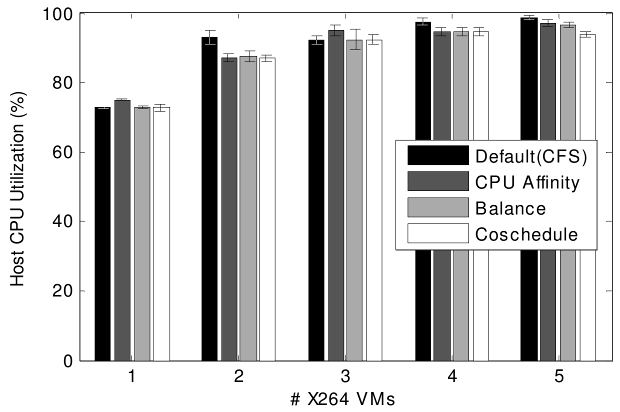

Figure 11B. The CPU utilization on the host with 95% confidence interval.

5.3.5 Experiment 5

This experiment assesses the scalability of each scheduling algorithm as the number of VMs increases. We keep adding more four-vCPU VMs running X.264 until reaching the host's maximum CPU capacity. We use the aggregated throughput of all VMs as a performance metric.

As shown in Figure 11A and B, balance scheduling, affinity-based scheduling and co-scheduling scale better than CFS due to the synchronization latency problem. Their X.264 throughputs increase as the number of VMs and the CPU utilization increases, when the host runs between one to three VMs. The host reaches its maximum capacity, when running 3-4 VMs. As shown in Figure 11A, affinity-based scheduling achieves 3% higher X264 throughput than balance scheduling due to better cache performance. When the host has five VMs, the thrashing effect starts to take place. Performance of all scheduling algorithms decreases, while the CPU utilization does not. As seen in Figure 11A and B, balance scheduling yields up to 4% higher in the X264 throughput than co-scheduling with about the same amount of CPU resources due to additional context switching.

5.3.6 Experiment 6

As discussed in Section 4.4, the performance of balance scheduling theoretically declines as the runqueue size grows. In this section, we show that in practice the average runqueue (per CPU) does not exceed six even if the four-CPU host has more than 24 threads. We run 14 four-vCPU VMs in the host: one X.264 VM and the rest (13 VMs) running the CPU workload with a *CPULimit* program [Marletta2010]. *CPULimit* is used to control CPU usage in the VMs. The maximum CPU usage of the 13 VMs is 8%, bounded by the maximum CPU capacity in the host. 14 VMs is the maximum number of VMs we can run concurrently due to the memory capacity. We measure the X.264 throughput and the runqueue size of each host CPUs.

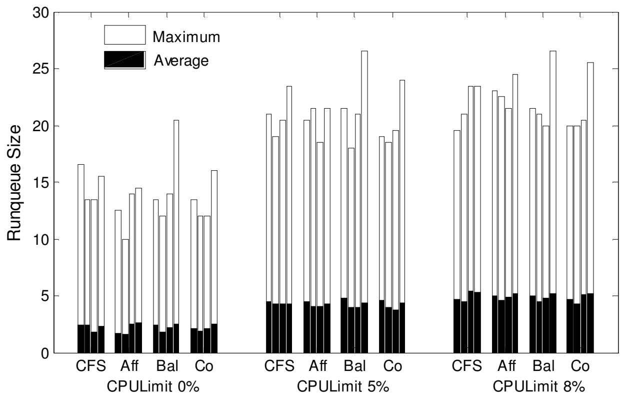

Figure 12. The average and maximum runqueue size of four physical CPUs by each scheduling approach.

In this experiment, there are 56 vCPU threads, in addition to other threads (e.g. QEMU and system threads), alive in the host. One may expect to have at least 14 tasks per runqueue. In fact, a runqueue contains only runnable threads, not threads that are blocked or sleeping. As shown in Figure 12, as the CPU usage in the 13 VMs increases, the average runqueue size of each CPU increases but still remains less than six, although the maximum runqueue at a certain moment can go up to 26. Due to the limited number of runnable threads in the runqueue, balance scheduling still performs very well even if the host has many VMs running. As shown in Figure 13, balance scheduling improves the X.264 throughput by up to 82.69%, 3.8%, and 4.2%, compared to CFS, affinity-based scheduling and co-scheduling respectively.

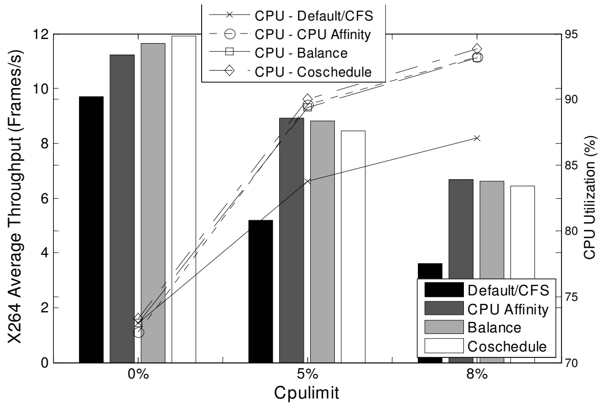

Figure 13. The X.264 performance and the host CPU utilization.

5.3.7 Experiment 7

This experiment shows how the scheduling algorithms affect performance of both SMP and non-SMP VMs running in the same host. We run X.264 in four-vCPU VMs and Ping in a one-vCPU VM. Ping sends an ICMP packet to another machine every one millisecond for 300,000 times. It goes to sleep after a packet is sent. We record the X.264 throughput and the standard deviation of Ping response times (jitter). High jitter can cause an undesirable effect, for example unusable video rendering.

As shown in Figure 14, balance scheduling has better Ping jitter (up to 41%) and more aggregated X.264 throughput (up to 8%) than affinity-based scheduling due to global load balancing. In balance scheduling, the Ping vCPU should always run in the least-loaded CPU, but it is not always the case in affinity-based scheduling. By default, the load balancer is triggered every 60 milliseconds. It is possible that a CPU has more load than the others for a certain period of time. By design, balance scheduling allows a vCPU to move to the least-loaded CPU every time it wakes up, given that the CPU does not have its siblings. In the affinity-based scheduling, the vCPU has to run on the same CPU. Hence, balance scheduling can better adapt to load changes than affinity-based scheduling. The benefit of load adaptation decreases as the number of the available CPUs for vCPU siblings decreases.

Balance scheduling has better the X.264 throughput (up to 8%) and Ping jitter (up to 2.5%) than co-scheduling due to priority inversion and additional context switching. Balance scheduling yields (up to 12%) higher aggregated X.264 throughput than CFS due to the synchronization latency problem. It also has similar or (up to 27%) higher Ping jitter than CFS. These results suggest that balance scheduling can effectively schedule both SMP and non-SMP VMs without suffering from priority inversion and global load balancing.

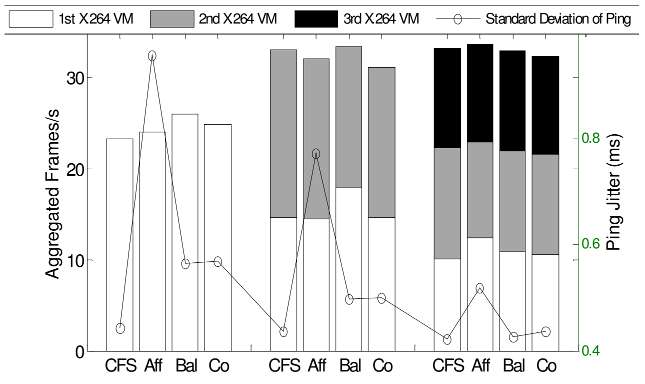

Figure 14. The X.264 performance in SMP VMs and Ping jitter performance in non-SMP VMs.

5.3.8 Discussion

Application performance degradation in a SMP VM depends on the degree of synchronization in both applications and OS inside the VM. As shown in Experiment 3 and 4, a SMP VM running synchronization-free applications (no application locks) can also suffer from the synchronization latency problem because the guest OS is capable of concurrent processing.

For example, the file system in guest OS can process multiple read/write requests simultaneously to reduce the latency perceived by users. Synchronization is required to provide concurrent modifications on the file system structure. In Experiment 4, we simultaneously run two independent disk-I/O processes which continuously create, read and write a number of files in the same directory. In the file system, a file or directory is represented by an *inode* which can be identified by a unique number within a file system. An inode contains file information, such as physical locations of file data, permission, and file size. An inode for a directory also has a list of inodes, identifying files in the directory. When two processes concurrently create new files in the same directory, they need to be synchronized in order to access and update the directory inode.

Similar to the file system, the network processing in guest OS also requires synchronization. For instance, when the networking layer and a device driver access a buffer simultaneously, a lock must be held prior to the access. A buffer (a block of memory) is used to store network packets. In Experiment 4, we run a single-thread network application, TTCP, in the SMP VM. TTCP continuously sends a number of TCP packets to another VM, which will send TCP ACK packets back upon receiving TCP packets. For the packet transmission, the networking layer creates packets and places them in the buffer. The device driver removes packets from the same buffer and sends to the network. The networking layer and the device driver require synchronization to access the shared buffers. Hence, even if a SMP VM runs a synchronization-free and network-bound application, the synchronization is still required in the guest OS.

Due to task scheduling, synchronization latency in a SMP VM can significantly increase, adversely affecting application performance. By design, the co-scheduling algorithm should work exceptionally well with synchronization-intensive applications because it synchronizes the execution of vCPU siblings, as shown in Figure 2. It would be futile to schedule the vCPU siblings in different time slots, if they often contend on the same lock. However, if the synchronization is barely required in a SMP VM, forcing vCPU siblings to be scheduled simultaneously can result in vCPU-execution delay, leading to application performance degradation. As shown in Figure 3, if the vCPUs mostly execute independent jobs, each vCPU should be able to run as soon as a CPU becomes available without unnecessary delay.

Unlike co-scheduling, balance scheduling does not force vCPU siblings to be scheduled simultaneously. It just balances vCPU siblings on different physical CPUs to increase a chance of the vCPUs being scheduled simultaneously. Balance scheduling never delays vCPU execution. Hence, minimal synchronization applications should benefit from balance scheduling more than co-scheduling. As shown in Experiment 1, balance scheduling has the shortest completion time (13% better than co-scheduling) on the multiple-independent-process workload (no application synchronization). But VMware's co-scheduling solution has the smallest completion time (10% better than balance scheduling) on HackBench (synchronization-intensive application). Theoretically, balance scheduling should be preferable to co-scheduling as the degree of synchronization in a SMP VM decreases. We also evaluate co-scheduling and balance-scheduling against other concurrent applications with different degree of synchronization (TPC-W, DVDstore, Compile and X264). As shown in Experiment 2 and 4, balance scheduling can improve application performance similarly to DT co-scheduling with a possible few percentage gain (e.g. reduce the average response times of TPC-W and DVDstore by up to 3%). Overall, balance scheduling exhibits a promising capability in alleviating the synchronization latency problem without the co-scheduling drawbacks.

Balance scheduling can also significantly improve application performance, compared to synchronization-unaware schedulers, such as Xen and CFS. As shown in Experiment 1, balance scheduling can complete HackBench and the multiple-independent-process workload 6% and 56% quicker than Xen/Credit scheduler does, respectively. Balance scheduling can reduce the average TPC-W response time by 85% compared to CFS. It also improves the I/O processing efficiency by up to 40% and 70% for the disk-I/O read and write operations, compared to CFS. Additionally, balance scheduling can effectively schedule more SMP VMs than CFS. As shown in Experiment 5, balance scheduling increases the aggregated X264 throughput as the number of VMs increases (up to four VMs). With CFS, the X264 throughput increases, when the number of VMs increases up to three VMs. Then, the X264 throughput starts to drop. The X264 throughputs by CFS are also consistently less than the throughputs by balance scheduling (up to 15%). The reason is that CFS wastes more CPU cycles due to the synchronization latency problem.

Moreover, balance scheduling can potentially adapt to load changes, unlike affinity-based scheduling (static configuration). As shown in Experiment 7, balance scheduling can improve Ping jitter up to 41%, compared to affinity-based scheduling.

6. Related Work

In the past, without virtualization, Ousterhout [Ousterhout1982] proposed a co-scheduling algorithm which schedules concurrent threads simultaneously to reduce application synchronization latency. Lee et al. [Lee1997] show that the co-scheduling algorithm can cause CPU fragmentation, which reduces CPU utilization, and priority inversion, which reduces I/O performance and other resource utilization. Later works [Feitelson1992, Wiseman2003] try to improve on the co-scheduling algorithm.

With virtualization, the synchronization latency problem becomes severe; spinlocks in a guest OS can get preempted. This never happens in a non-virtualized environment. Uhlig et al. [Uhlig2004] identify this problem as lock-holder preemption (LHP) in SMP VMs. They propose several techniques to prevent LHP. The techniques require augmenting guest OS or installing a special-crafted device driver, and thus may not be feasible in commodity OSes (e.g. Windows). Balance scheduling does not prevent LHP, but alleviates effect of LHP. Even if spinlocks in a SMP VM are no longer preempted, application locks can still benefit from balance scheduling.

To mitigate the synchronization latency problem in SMP VMs, previous works [VMware2008, Weng2009, Bai2010] propose a co-scheduling solution where vCPU siblings are scheduled simultaneously. Unlike co-scheduling, balance scheduling only balances vCPU siblings on different physical CPUs without forcing the vCPUs to be scheduled at the same time. Balance scheduling can be easily implemented and significantly improve application performance without the complexity and drawbacks found in co-scheduling (CPU fragmentation, priority inversion and execution delay).

VMware developed several versions of co-scheduling for VMware ESXi. The first version, called *strict co-scheduling*, is included in VMware ESX 2.x [VMware2008]. Due to CPU fragmentation, VMware created *relaxed co-scheduling* (ESX 3.x) where all vCPU siblings are stopped and only the lagging vCPUs are started simultaneously when they are out of synchronization. The relaxed co-scheduling is further refined in ESX 4.x [VMware2010b] – stopping only advanced vCPUs, instead of all vCPUs. Balance scheduling is similar to the relaxed co-scheduling in a sense that the scheduling operation is per vCPU. But balance scheduling never delays execution of a vCPU to wait for another vCPU in order to maintain synchronous progress of vCPU siblings. Balance scheduling is also simpler. No discrepancy accruing in progress of vCPU siblings is required. To avoid the co-scheduling drawbacks, Weng et al. [Weng2009] limit co-scheduling to a SMP VM with a concurrent application, unlike balance scheduling which does not share any co-scheduling drawbacks, thereby benefiting both concurrent and non-concurrent SMP VMs.

Jiang et al. [Jiang2009] propose several techniques to improve KVM performance, such as temporarily increasing the priority of vCPUs and approximately co-scheduling vCPU siblings by changing their scheduling class from SCHED_OTHER (default scheduling class in CFS) to SCHED_RR (real-time scheduling class). Changing the priority of vCPUs can affect the fairness and performance of other VMs; unlike balance scheduling which never changes scheduling class or priority of vCPUs.

AMD [Langsdorf2010] and Intel [Intel2010] also provide architectural support for heuristically detecting contended spinlocks so that the hypervisor can de-schedule them to reduce excessive CPU cycle use. They add additional fields in the VM data structure (Pause-Filter-Count in AMD and PLE_Gap and PLE_Window in Intel). For example, in Intel, PLE_Gap is an upper bound on the amount of time between two successive executions of PAUSE in a loop. PLE_Window is an upper bound on a guest allowed for a PAUSE loop. According to KVM's codes, PLE_Gap is set to 41 and PLE_Window is 4096. It means that this approach can detect a spinning loop that lasts around 55 microseconds on a 3GHz CPU. As mentioned earlier, the synchronization problem incurs not only by synchronization in applications inside a VM, but also synchronization in the guest kernel. As shown in Figure 6A, most spinlocks in VMs last less than 50 microseconds. Hence, this support should help cease application locks rather than spinlocks in kernel. However, the values of PLE_Gap and PLE_Window should not be too small due to the cost of VM_EXIT, (4-5K cycles [Zhang2008], depending on CPU architectures). VM_EXIT can also cause performance loss due to transition cost (VM exit, VM reads, VM writes, VM entry, and TLB flushing cost).

7. Conclusion

Despite the benefit of parallel processing, SMP VMs can also increase synchronization latency significantly, depending on task scheduling. In this paper, we show that a SMP VM running non-concurrent applications can also need synchronization for concurrent processing in the guest OS.

To mitigate the synchronization problem, previous works have proposed a co-scheduling solution, which rigorously maintains synchronous scheduling of vCPU siblings. This approach can be too expensive for SMP VMs with minimal synchronization due to delay in vCPU execution. We propose the balance scheduling algorithm, which simply balances vCPU siblings on different physical CPUs without strictly scheduling the vCPUs simultaneously. Balance scheduling can improve performance of concurrent SMP VMs similarly to co-scheduling without the co-scheduling drawbacks (CPU fragmentation, priority inversion and execution delay). Unlike co-scheduling, balance scheduling can also effectively schedule SMP VMs with minimal synchronization; thereby benefiting many SMP VMs. In practice, most applications, including concurrent applications, should not demand intensive synchronization. Minimal synchronization usage is encouraged in concurrent applications to promote parallelism. Synchronization serves as the bottleneck in parallel execution. Yet, it is still necessary in many concurrent applications. Additionally, a number of existing and legacy applications are still non-concurrent.

Acknowledgments

We would like to thank Akkarit Sangpetch for insightful discussions on the implementation issues and John Lanyon for comments. We would like to also thank our shepherd, Jacob Gorm Hansen, and the anonymous reviewers for helpful comments on the paper.

Appendix

A. *Expected lock latency calculation*

We use Eq. 1 and 2 to calculate the expected lock latency of CFS and balance scheduling respectively. T_H is the average lock holding time. |RQ| is a runqueue size. |VW| is the number of vCPUs that want to acquire the same lock, and |CPU| is the number of available physical CPUs.

$$Eq.1 \quad Expected\ Latency_{default} = P_{stack}\left[\frac{\binom{|RQ|}{|VW|}(|VW|-1)!|CPU|T_P+|CPU|\alpha}{|CPU|\binom{|RQ|}{|VW|}|VW|!}\right] + (1-P_{stack})\left[\frac{T_P\left(\left(\sum_{i=2}^{|RQ|}|CPU|\binom{(i-1)|CPU|}{|VW|-1}(|VW|-1)!\right)-\binom{|RQ|}{|VW|}|CPU|(|VW|-1)!\right)}{\beta} + \frac{\sum_{i=2}^{|VW|}\binom{|CPU|}{i}|RQ|\left(\prod_{k=i+1}^{|VW|}((|RQ|-1)|CPU|-(k-i-1))\right)T_H}{\beta} + \frac{|CPU|\alpha}{\beta}\right], \quad where\ \alpha =$$

$$\left(\sum_{i=1}^{|RQ|-(|VW|-1)} \binom{|RQ|-i}{|VW|-1} (|VW|-1) i T_{TS}\right), \beta = \binom{|CPU||RQ|}{|VW|} |VW|! - |CPU| \binom{|RQ|}{|VW|} |VM|! = \left(\frac{(|CPU||RQ|)!}{(|CPU||RQ|-|VW|)!} - \frac{|CPU||RQ|!}{(|RQ|-|VW|)!}\right), and\ |CPU| \geq |vCPU| \geq |VW|$$

$$Eq.2\ \ Expected\ Latency_{balance} = \frac{T_P\left(\left(\sum_{i=2}^{|RQ|}|CPU|\binom{(i-1)|CPU|}{|VW|-1}\right) - \binom{|RQ|}{|VW|}|CPU|(|VW|-1)!\right)}{\beta} + \frac{\sum_{i=2}^{|VW|}\binom{|CPU|}{i}|RQ|\left(\prod_{k=i+1}^{|VW|}((|RQ|-1)|CPU| - (k-i-1))\right)T_H}{\beta} + \frac{|CPU|\alpha}{\beta}$$

B. Analysis of our DT scheduling

The computation complexity of DT co-scheduling is O(*N*) where *N* is the number of CPUs, according to the pseudo code in Algorithm 3.

Algorithm 3: DT Co-scheduling

```
all_cpus ← set of all physical CPUs
if task T is a vCPU
    if task T is not assigned a runqueue
        VMID ← Parent PID of task T
        used_cpus ← { }
        for each vCPU v of VMID
            add CPU that v is on in used_cpus
        end for each
        CPUs of task T ← all_cpus – used_cpus
    else if task T is the first vCPU of the VM to be
          scheduled
        for each vCPU sibling v of task T
            if v is not currently scheduled
                send reschedule interrupt
                context switch to v
            end if
        end for each
    end if
end if
```

We also run three multi-threaded applications (Pi, HackBench and DVDstore) to compare the performance of our DT co-scheduling and the co-scheduling in [Ousterhout1982]. We mimic the co-scheduling on KVM by changing vCPUs' scheduling class from SCHED_OTHER (CFS) to SCHED_RR (RT scheduling) with the priority of 20. RT tasks have higher priority than CFS tasks. By default, the RT period is 1 second and the RT runtime is 950 milliseconds. This reserved RT runtime is given to RT tasks first and the rest is allocated to CFS tasks. We experiment with four combinations of RT runtime and period: 15ms/30ms, 28ms/30ms, 500ms/1000ms and 950ms/1000ms (default). As shown in Figure 15, our DT co-scheduling improves DVDStore (I/O and network-intensive) performance at least 6% better than the co-scheduling. DT co-scheduling improves HackBench and Pi performance at least 0.7% and 0.3% better than the co-scheduling respectively. These results show that our DT co-scheduling can perform similarly or better than the co-scheduling without tuning the time slice and period parameters.

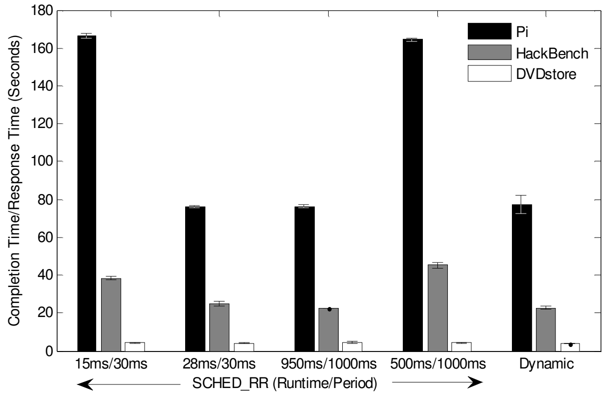

Figure 15. The comparison of application performance between the SCHED_RR-based co-scheduling and our DT co-scheduling.

References

[Bai2010] Y. Bai, C. Xu, and Z. Li. "Task-aware based co-scheduling for virtual machine system", In Proceedings of the 2010 ACM Symposium on Applied Computing. SAC '10. ACM, New York, NY, 181-188.

[Barham2003] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. 2003. Xen and the art of virtualization. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, 164-177.

[Coker2001] R. Coker. Bonnie++ version 1.03. http://www.coker.com.au/Bonnie++/, 2001.

[Cormen2001] T. H. Cormen,, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. Chapter 13: Red-Black Trees, pp. 273–301.

[Dell2007] Dell, Inc. The DVD Store Version 2. http://www. dell techcenter.com/page/DVD+Store, December, 2007.

[Edge2009] J. Edge. A look at ftrace. http://lwn.net/Articles/ 322666/, March, 2009. (accessed August 2010).

[Feitelson1992] D. Feitelson, L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. Journal of Parallel and Distributed Computing, 1992.

[Fischer2005] G. Fischer, C. Rodriguez, C. Salzberg, S. Smolski. Linux Scheduling and Kernel Synchronization. Nov 11, 2005. Prentice Hall Professional.

[HackBench2008] HackBench, http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c, September 2008.

[Intel2010] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 2, June 2010.

[Jiang2009] W. Jiang, Y. Zhou,, Y. Cui, W. Feng, Y. Chen, Y. Shi, and Q. Wu. CFS Optimizations to KVM Threads on Multi-Core Environment. In Proceedings of the 2009 15th international Conference on Parallel and Distributed Systems. ICPADS2009.

[KVM2008] Qumranet. KVM. Kernel Based Virtual Machine. http://www.linux-kvm.org/, September, 2008.

[Langsdorf2010] M. Langsdorf. Patchwork: Support Pause Filter in AMD processors. https://patchwork.kernel.org/ patch/48624/ (accessed May 2010).

[Lee1997] W. Lee, M. Frank, V. Lee, K. Mackenzie and L. Rudolph, Implications of I/O for Gang Scheduled Workloads, Job Scheduling Strategies for Parallel Processing, pp. 215-237, 1997.

[Marletta2010] A. Marletta. CPU Usage Limiter for Linux. http://cpulimit.sourceforge.net/ (accessed August 2010).

[Mathur2007] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier. The new ext4 filesystem: current status and future plans. *Proceedings of the Linux Symposium*. Ottawa, ON, CA: Red Hat. 2007.

[Molnar2007] I. Molnar. CFS design. http://people.redhat.co m/mingo/cfs-scheduler/sched-design-CFS.txt, May 2007.

[Ousterhout1982] J. Ousterhout, "Scheduling Techniques for Concurrent Systems,"Proc. 3rd International Conference on Distributed Computing Systems, October 1982.

[Phoronix2010] Phoronix Test Suite. X.264 Benchmark. http: //www.phoronix-test-suite.com/index.php?k=downloads (accessed September 2010)

[RedHat2010] Red Hat, IBM, Hitachi, and Oracle. SystemTap. http://sourceware.org/systemtap/

[TPC2000] TPC. Transaction Processing Performance Council. TPC-W: A transactional web e-Commerce benchmark. http://www.tpc.org/tpcw/, January 2000.

[TTCP1996] TTCP Utility. Test TCP (TTCP) Benchmarking Tool and Simple Network Traffic Generator. http://www .pcausa.com/Utilities/pcattcp.htm, 1996.

[Uhlig2004] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. “Towards scalable multiprocessor virtual machines”, In Proceedings of the 3rd Conference on Virtual Machine Research and Technology Symposium - Volume 3, 2004. USENIX Association, Berkeley, CA.

[Vaddagiri2009] S. Vaddagiri, B.B. Rao, V. Srinivasan, A.P. Janakiraman, B. Singh, and V.K. Sukthankar. Scaling software on multi-core through co-scheduling of related tasks. In Linux Symp., pages 287–295, 2009.

[VMware2008] Drummonds. VMware, Inc. Co-scheduling SMP VMs in VMware ESX server. May 2, 2008. http://communities.vmware.com/docs/DOC-4960.

[VMware2010] VMware, Inc. VMware vSphere 4: The CPU Scheduler in VMware ESX 4 White Paper. http://www.vmware.com/files/pdf/perf-vsphere-cpu_scheduler.pdf (accessed September 2010).

[VMware2010a] VMware, Inc. Performance best practices for VMware vSphere 4.0. VMware ESX 4.0 and ESXi 4.0. http://www.vmware.com/pdf/Perf_Best_Practices_vSphere4.0.pdf (accessed September 2010)

[VMware2010b] VMware, Inc. VMware vSphere 4: The CPU scheduler in VMware ESX 4.1, September 2010. http://www.vmware.com/files/pdf/techpaper/VMW_vSphere41_cpu_schedule_ESX.pdf (accessed September 2010).

[VMware2010c] VMware, Inc. VMware vSphere Hypervisor (ESXi). http://www.vmware.com/products/vsphere-hypervisor/index.html. (accessed September 2010).

[Weng2009] C. Weng, Z. Wang, M. Li, and X. Lu. “The hybrid scheduling framework for virtual machine systems”, In Proceedings of the 2009 ACM SIGPLAN /SIGOPS international Conference on Virtual Execution Environments. VEE '09. ACM, New York, NY, 111-120.

[Wieërs2010] D. Wieërs. Dstat: Versatile resource statistics tool. http://dag.wieers.com/home-made/dstat/.

[Wiseman2003] Y. Wiseman , D. Feitelson, Paired Gang Scheduling, IEEE Transactions on Parallel and Distributed Systems, v.14 n.6, p.581-592, June 2003.

[Yaron2007] Yaron. Xen Wiki. Credit Scheduler. http://wiki.xensource.com/xenwiki/CreditScheduler November, 2007. (accessed August 2010).

[Yee2010] Yee, J. A y-cruncher-A Multi-Threaded Pi-Program. http://www.numberworld.org/y-cruncher/, August 2010.

[Zhang2008] X. Zhang, Y. Dong. Optimization Xen VMM Based on Intel Virtualization Technology. International Conference on Internet Computing in Science and Engineering, 2008 (ICICSE’08).