Dynamic Adaptive Scheduling for Virtual Machines

Chuliang Weng[‡], Qian Liu[‡], Lei Yu[‡], and Minglu Li[‡][§]
[‡]Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
[§]Shanghai Key Laboratory of Scalable Computing and Systems
{clweng, piano, anycom, mlli}@sjtu.edu.cn

ABSTRACT

With multi-core processors becoming popular, exploiting their computational potential becomes an urgent matter. The functionality of multiple standalone computer systems can be aggregated into a single hardware computer by virtualization, giving efficient usage of the hardware and decreased cost for power. Some principles of operating systems can be applied directly to virtual machine systems, however virtualization disrupts the basis of spinlock synchronization in the guest operating system, which results in performance degradation of concurrent workloads such as parallel programs or multi-threaded programs in virtual machines.

Eliminating this negative influence of virtualization on synchronization seems to be a non-trivial challenge, especially for concurrent workloads. In this work, we first demonstrate with parallel benchmarks that virtualization can cause long waiting times for spinlock synchronization in the guest operating system, resulting in performance degradation of parallel programs in the virtualized system. Then we propose an adaptive dynamic coscheduling approach to mitigate the performance degradation of concurrent workloads running in virtual machines, while keeping the performance of non-concurrent workloads. For this purpose, we build an adaptive scheduling framework with a series of algorithms to dynamically detect the occurrence of spinlocks with long waiting times, and determine and execute coscheduling of virtual CPUs on physical CPUs in the virtual machine monitor. We have implemented a prototype (ASMan) based on Xen and Linux. Experiments show that ASMan achieves better performance for concurrent workloads, while maintaining the performance for non-concurrent workloads. ASMan coscheduling depends directly on the dynamic behavior of virtual CPUs, unlike other approaches which depend on static properties of workloads and manual setting of rules. Therefore, ASMan achieves a better trade-off between coscheduling and non-coscheduling in the virtual machine monitor, and is an effective solution to this open issue.

Categories and Subject Descriptors

D.4.1 [**OPERATING SYSTEMS**]: Process Management—*scheduling, synchronization*

General Terms

Algorithms, Performance

Keywords

Virtualization, Scheduling, Synchronization, Adaptive strategy, Performance

1. INTRODUCTION

Virtualization [1][2] is currently becoming popular. It allows the functionality of multiple standalone computer systems to be aggregated into a single system, to exploit the increased processing capacity of recent hardware. Examples of system virtualization include VMware [3][4], Xen [5][6], KVM [7], Hyper-V [8], VirtualBox [9], User Mode Linux [10]. Differing from the traditional system software stack, a virtual machine monitor (VMM) is introduced into the system. The VMM virtualizes the physical CPUs (PCPUs), providing virtual CPUs (VCPUs), on which the guest operating system runs.

Generally a virtual machine (VM) with multiple VCPUs is seen by the guest operating system as a symmetric multiprocessing (SMP) system, on which multi-process or multi-threaded programs run. However, the VCPUs in a virtual SMP system are not continuously online, which is different from the non-virtualized scenario. When the workload in a VM is a non-concurrent application, the VMM schedules VCPUs of the VM asynchronously. This method is beneficial as it simplifies the implementation of the CPU scheduling in the VMM, and can deliver near native performance for non-concurrent applications running in VMs. Therefore, it is widely adopted in the implementation of VMMs such as Xen. However, when the workload in a VM is a concurrent application with synchronization operations, this scheduling method may give poor performance.

To mitigate this kind of performance degradation, coscheduling (alternatively, gang scheduling [11]) is applied to the CPU scheduling in the VMM in our previous work [12] and VMware [13]. For this method, the system administrator is expected to have sufficient knowledge about workloads in each VM to manually set the VMs' types with the help of tools provided by the VMM. The VMM can then schedule VCPUs of the same VM together. However, coscheduling also introduces additional overhead to the virtualized system, and may reduce the system performance if some VMs' types are incorrectly set. Furthermore, for concurrent applications, a considerable portion of program code may still run asynchronously without resulting in performance degradation. Therefore, to minimize both the negative influence of virtualization on synchronization and the additional overhead of coscheduling in the VMM, an ideal scenario is that the VMM schedules VCPUs of a VM to PCPUs asynchronously when no synchronization operation is in progress between threads or processes in the VM, and schedules them synchronously when those threads or processes have to synchronize with each other. Unfortunately, existing methods use static coscheduling so the burden of distinguishing workloads' properties is imposed on the user.

Moreover, the total number of VCPUs in a virtualized system is usually larger than the number of PCPUs, and the scheduler in the VMM maps VCPUs into PCPUs in a time-shared manner. When multiple VMs run simultaneously on top of the VMM, it is not a good practice to allow a VM with a heavy workload to appropriate the CPU resource of other VMs without any limitation. Therefore, a fair share of CPU should be guaranteed by the VMM. Proportional share fairness between VMs is used in VMMs; for example, Xen uses *weight* and VMware *entitlement*. In short, each VM should get CPU time in proportion according to the strategy. Consequently, coscheduling should also keep this kind of proportional share fairness in the virtualized system.

Spinlocks and semaphores are widely used as synchronization primitives in modern operating systems. They are used to resolve contention between threads or processes, using either busy waiting (spinning) or non-busy waiting (blocking). Synchronization operations in concurrent applications are sometimes mapped internally to spinlocks or semaphores in the operating system kernel, especially when the competition exists. Blocked threads or processes are descheduled from VCPUs, and state-of-the-art VMMs detect the idle status of a VCPU and keep proportional share fairness. Consequently, virtualization has little negative impact on synchronization using semaphores. However, virtualization disrupts the normal operating system policy that a thread or process holding spinlocks should not be preempted. As a result, spinlocks may have a long waiting time in the virtualized system (see Section 2.2 for details), and then we define them as over-threshold spinlocks (see Section 4.2 for details).

The motivation of this paper is to improve the performance of concurrent applications in virtualized systems. We argue that the impact of virtualization on spinlocks is a major cause of performance degradation in concurrent applications. To mitigate this, we use demand-driven coscheduling to bring the VCPUs of a VM online when necessary, rather than static coscheduling according to the type of VMs, adopted in our previous work [12]. This avoids excessive waiting time in spinlocks while reducing the additional overhead of coscheduling. In summary, the main contributions of this paper are as follows.

- We study the impact of virtualization on synchronization in concurrent applications with experiments. Results show that virtualization has a significant influence on the waiting time of spinlocks in the guest operating system, especially when the VCPU online rate is relatively low. They also demonstrate that frequent occurrence of over-threshold spinlocks results in severe performance degradation of a concurrent workload running in a VM.
- The concept of *VCPU Related Degree* (VCRD) is presented, which dynamically depicts the relatedness of VCPUs in a VM. When the VCRD of a VM is high, its VCPUs will be coscheduled to PCPUs. Otherwise, its VCPUs are scheduled asynchronously. We propose an algorithm to automatically adjust the VCRD of a running VM based on the detection of over-threshold spinlocks.
- We propose an adaptive scheduling framework for the VMM which implements dynamic coscheduling based on VCRD, while also guaranteeing fairness among VMs in the system. Our work attempts to address synchronization in virtualized systems while avoiding unnecessary overhead for coscheduling.
- A prototype has been implemented based on Xen and Linux; the modifications to Xen and the Linux kernel are highly localized and relatively small. We have evaluated the working prototype on standard benchmarks, and the results show that our approach improves the performance of concurrent applications while maintaining the performance of non-concurrent applications in the virtualized system.

The rest of this paper is organized as follows. The next section introduces the background, and Section 3 presents our scheduling framework while Section 4 proposes scheduling algorithms. Section 5 describes performance evaluation. Section 6 provides a brief overview of related work, and Section 7 concludes the paper.

2. BACKGROUND

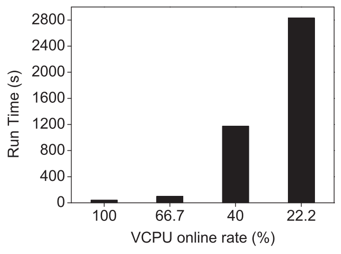
In this section, we briefly describe system virtualization, and then present measurements to demonstrate that the negative impact of virtualization on synchronization in concurrent programs is severe.

2.1 Virtualization

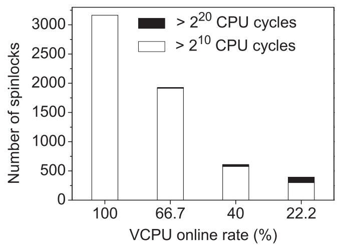
System virtualization provides a complete system environment, in which many processes from multiple users with different operating systems can coexist. This kind of system virtualization was first developed during the 1960s and early 1970s, and it is the origin of the term *virtual machine* [2]. Today, system virtualization is becoming quite popular. Virtualization provides an additional layer (VMM) between the running operating system and the underlying hardware. The VMM manages the hardware resources and exports them to guest operating systems running on them. As a result, the VMM is in full control of allocating PCPUs to the guest operating system.

In current systems such as VMware and Xen, a VM with multiple VCPUs appears to the guest OS as a SMP system. All VCPUs behave identically and any process can execute on any VCPU. The number of VCPUs of a VM is related to the number of PCPUs in the system, and is not more than this number. Concurrent applications such as multi-threaded programs or parallel programs can run directly on a virtual SMP system. From the viewpoint of end users, a virtual SMP system is similar to a physical SMP system. To the VMM, a VCPU may be sometimes online, and sometimes offline. A VCPU may also be migrated from one PCPU to another PCPU, as a result, it runs on a PCPU for some time slices at this moment, and runs on another PCPU at the next moment. A VCPU could also be pinned to a fixed PCPU, and consequently it will only be mapped to the fixed PCPU when it is online.

2.2 Synchronization in virtualization



(a)

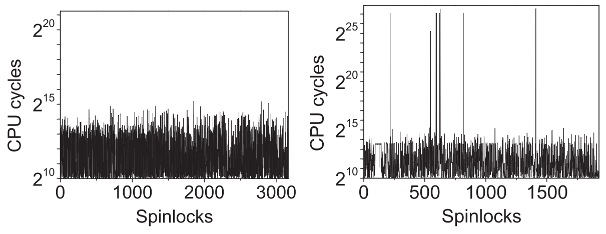


(b)

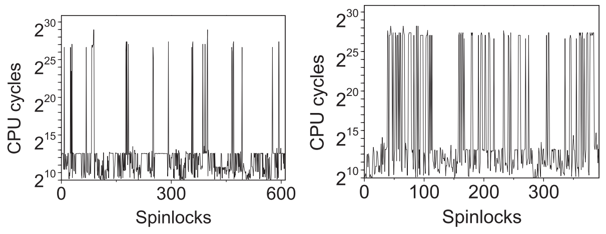
Figure 1: Parallel benchmark LU runs on a virtual SMP system with 4 VCPUs. The run time of LU is shown in (a), and the statistical information (waiting time) on spinlocks is shown in (b).

In a concurrent application, synchronization libraries are usually used to implement synchronization operations between threads or processes. These libraries leverage functions of underlying synchronization primitives provided by the operating system. Synchronization primitives are used to resolve contention between threads or processes, by either busy waiting (spinning) or non-busy waiting (blocking). Specifically, they are spinlock and semaphore widely used in operating system kernels. Taking GNU OpenMP as an example, the runtime library (libgomp) is essentially a wrapper around the POSIX threads library, and synchronization APIs are implemented using atomic instructions and futex system calls. Futex is implemented by the hybrid of spinning and blocking, and then synchronization in parallel applications may involve spinlocks or semaphores in kernel, particularly when there is competition in the lock. With spinlocks, threads wait actively monitoring locks, and respond to lock handoffs very quickly, because it requires no context switching. However, polling for locks is highly resource-intensive. On the other hand, with semaphores waiting threads are blocked, then rescheduled once the required lock is available. Generally speaking, spinlocks are adopted for very short critical sections, otherwise semaphores are used.

Unlike CPUs in real SMP systems, virtual CPUs are not always online, so synchronization primitives designed for real SMP systems may negatively affect performance. To evaluate the impact of virtualization on synchronization in the



(a) (b)



(c) (d)

Figure 2: Parallel benchmark LU runs on a virtual SMP system with 4 VCPUs. The detailed spinlock waiting time is shown in (a)-(d), where the VCPU online rate is: (a) 100%, (b) 66.7%, (c) 40%, (d) 22.2%.

guest operating system, we have run the concurrent benchmark LU from the NAS parallel benchmark suite [14] on a virtualized system (see Sections 5.1 and 5.2 for details on our experimental setup).

As depicted in Figure 1 (a), the run time (in seconds, s) of LU increases quickly as the VCPU *online rate* decreases. The *online rate* of a VCPU denotes the percentage of time of the VCPU being mapped to a PCPU (see Section 4.1 for its formal definition). We also instrument the spinlock code and the semaphore code in the Linux kernel to measure their waiting times through the high-resolution timer provided by Linux. We monitor waiting times of spinlocks and semaphores in the virtual SMP system over a 30 second period while the LU benchmark is testing. The waiting times of all semaphores are less than 2^{16} CPU cycles, even when the VCPU online rate is 22.2%. For spinlocks, we collect information on those spinlocks, whose waiting times are larger than 2^{10} CPU cycles. The statistical information on spinlocks is shown as Figure 1 (b), and the details of spinlock waiting time are shown in Figure 2.

When a concurrent workload runs on a VM, we observe that: (1) the number of spinlocks in a fixed time interval decreases along with the VCPU online rate; (2) the majority of spinlocks have waiting time less than 2^{15} CPU cycles; (3) the percentage of waits taking more than 2^{25} CPU cycles increases quickly as the VCPU online rate decreases; (4) the long waits usually occur in some neighboring spinlocks.

Therefore, virtualization has a severe performance impact on concurrent applications in the virtualized system. As experiments are performed in a fixed length of time (30 seconds) and the CPU time obtained by a VM is in proportion to its online rate, a smaller part of LU is executed in a VM with a lower online rate, and correspondingly the number of spinlocks is smaller. However, the long waiting time is introduced by virtualization into some spinlocks in the guest operating system when concurrent applications run on a vir-

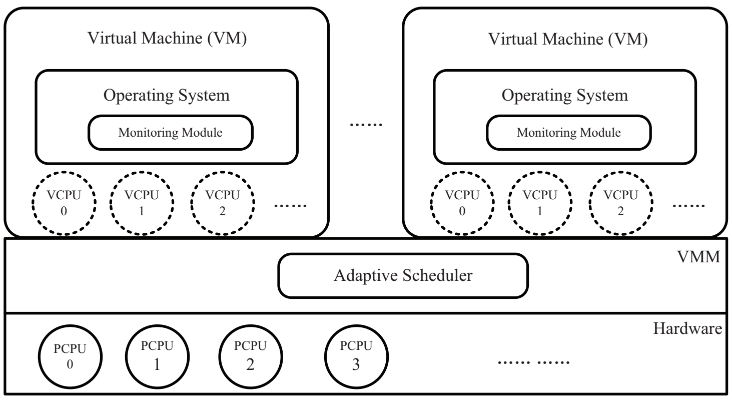


Figure 3: Scheduling framework

tual SMP system. Moreover, the frequency of occurrence of long waits in spinlocks increases quickly when the VCPU online rate decreases. As a result, concurrent applications have a significant performance degradation in the virtualized system.

3. SCHEDULING FRAMEWORK

In this section, we will propose a scheduling framework for addressing performance degradation induced by virtualization, and introduce its implementation (ASMan).

3.1 VCRD

Virtualization disrupts the hypothesis of spinlocks in traditional operating systems. That is, a thread or process holding a spinlock is not preempted. To mitigate the negative impact of virtualization on synchronization, an effective approach is to make the VCPUs of a VM like the CPUs of a physical machine from the viewpoint of the scheduler in the operating system. Therefore, when a concurrent application runs on a VM, the VCPUs should be online at the same time. That is, VCPUs of the VM should be coscheduled. However, coscheduling may also introduce some additional overhead. As a result, intuitively the VMM coschedules VCPUs of a VM when concurrent applications run on the VM (static coscheduling), otherwise, the VMM can schedule those VCPUs asynchronously when the workloads are non-concurrent applications.

To further reduce the overhead of coscheduling, we propose a dynamic instead of static coscheduling. We adopt the concept of *VCPU Related Degree* (VCRD) to reflect the related degree of VCPUs in a VM. When spinlocks in a VM have long waiting times, the VCRD of the VM is set as `HIGH`, and all VCPUs should be online at the same time. When spinlocks in a VM behave normally as in a traditional operating system, the VCRD of the VM is set as `LOW`, and then the online time of each VCPU could be absolutely asynchronous.

Unpredictable and irregular process or thread behavior over short time scales leads to large variations in load. Unlike static coscheduling, the VCRD of a VM may change from `HIGH` to `LOW` or from `LOW` to `HIGH` in the execution period of a single concurrent program running on the VM. Threads or processes in a concurrent program can still run independent from each other when they are not in critical sections. As a result, VCPUs need to be coscheduled to PCPUs only at the time of synchronization in the program. In the dynamic coscheduling approach, the VCRD of a VM depends on the dynamic behaviour of VCPUs instead of the static properties of workloads.

3.2 Framework

As depicted in Figure 3, multiple VMs run simultaneously on a single physical system, and each VM is treated as a virtual SMP system with multiple VCPUs. Although the VMM is in control of allocating physical resources, it lacks knowledge of processes in the guest operating systems. We therefore introduce a *Monitoring Module* to bridge this semantic gap [15]. The Monitoring Module runs in the guest operating system kernel in each VM, and is responsible for detecting spinlocks with long waiting times. Based on this information, the Monitoring Module updates the VCRD of the corresponding VM and reports it to the VMM.

It is the *Adaptive Scheduler* in the VMM that schedules VCPUs of VMs to PCPUs. A good scheduler in the VMM needs to guarantee that VCPUs from multiple VMs can multiplex PCPUs fairly and efficiently. To mitigate the influence of virtualization on synchronization in VMs, the Adaptive Scheduler makes VCPUs of a VM be online at the same time when its VCRD is `HIGH`, and attempts to make VCPUs of the VM act like CPUs of a physical machine from the viewpoint of the scheduler in the guest operating system. When the VCRD of a VM is `LOW`, the Adaptive Scheduler just maps them to PCPUs asynchronously. In either case, the scheduler manages load balancing between PCPUs and fairness of CPU share between VMs.

3.3 Implementation

We have implemented a working prototype of the proposed scheduling framework, called ASMan (**A**daptive **S**cheduling **Man**ager). In this subsection, we will discuss the implementation briefly.

Xen [5] is adopted as the VMM in ASMan, and our Adaptive Scheduler is implemented based on the default scheduler in Xen, that is, the *Credit Scheduler*. Moreover, Linux is the guest operating system. We choose Linux and Xen because of their broad acceptance and the availability of their opensource code.

Credit Scheduler. The scheduler in the VMM should keep fairness in resource sharing among VMs. An effective

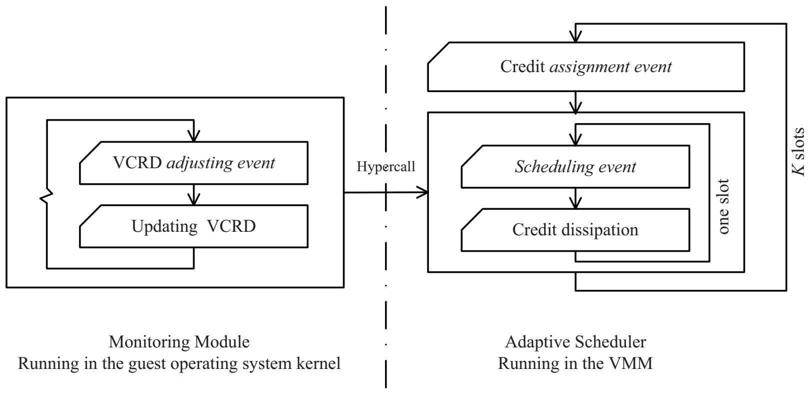


Figure 4: Overview of dynamic adaptive scheduling

and widely adopted method is *proportional share fairness* in which the CPU usage of a VM is in proportion to the weight that has been assigned to it. The *weight* of a VM is an integer parameter associated with it, and the *weight proportion* of a VM is the value of its weight divided by the total of all VMs' weights in the system. The scheduler gives each VCPU a fraction of total system CPU time depending on its *weight proportion*. As the latest scheduler in Xen, the Credit Scheduler adopts a proportional share strategy for share fairness, featuring automatic workload balancing of VCPUs across PCPUs [16]. Before a PCPU goes idle, it will find any runnable VCPU in the run queue of the other PCPUs. This approach guarantees that no PCPU is idle when there exists a runnable VCPU in the system. However, the characteristics of dynamic workloads in VMs are not further considered in this scheduler. We argue that these factors are not negligible when the system scale becomes large.

Monitoring Module. To monitor spinlocks with long waiting times, the Monitoring Module is implemented in the guest operating system kernel in each VM. We insert code into the spinlock code in the kernel, in order to collect the number of spinlocks and the waiting time for each. The waiting time is measured by the high-resolution timer provided by Linux. Upon detecting a spinlock whose waiting time is longer than a certain threshold (2^{δ} CPU cycles), the Monitoring Module adjusts the VCRD from `LOW` to `HIGH` and estimates the lasting time (see Algorithm 1), based on the locality of synchronization. After updating the VCRD, the Monitoring Module sends its information to the Adaptive Scheduler by invoking the hypercall `do_vcrd_op` implemented by us. A hypercall is a software trap mechanism from a VM to the VMM provided by Xen, just as a system call is a software trap from an application to the operating system kernel.

Adaptive Scheduler. The Adaptive Scheduler is modified from the Credit Scheduler, and *Credit* is still used in the Adaptive Scheduler. Similarly to the Credit Scheduler, the Adaptive Scheduler uses 30 ms (millisecond) time slices for the PCPU allocation. The Credits of all runnable VMs are recalculated at 30 ms intervals. They mainly depend on weights which the administrator assigns to VMs. The basic unit time of scheduling is 10 ms, and the Credit of a running VCPU is decreased every 10 ms.

Unlike the Credit Scheduler, our scheduler also uses VCRD information which is added to the data structure associated with each VM in the VMM. Also the hypercall `do_vcrd_op` is implemented in the Adaptive Scheduler, and is used to update the VCRD of a VM. The VCRD of a VM is `LOW` by default, and the priority of VCPUs in the run queue of a PCPU is based on the unused Credit allocated to each VCPU. When the VCRD of a VM changes from `LOW` to `HIGH`, VCPUs in the virtual SMP system will be relocated on the physical SMP system (see Algorithm 3), so that these VCPUs are in run queues of different PCPUs in the system, and will be coscheduled to those PCPUs. The Adaptive Scheduler uses inter-processor interrupts (IPI) to coschedule multiple VCPUs of a VM to PCPUs (see Algorithm 4). As a special type of interrupt, an IPI may be invoked by one processor to interrupt another processor in physical SMP systems; it is usually used to implement cache coherency synchronization.

4. SCHEDULING STRATEGY

This section presents a scheduling strategy based on the above scheduling framework. We begin with an overview of the strategy, followed by specifics regarding scheduling algorithms.

4.1 Overview

The time is subdivided into a sequence of fixed-length *slot*s (the basic time unit for scheduling), and each PCPU is allocated to at most one VCPU within each slot. When a VM is created, its VCPUs will be inserted into run queues of PCPUs.

As depicted in Figure 4, a spinlock with a long waiting time in a VM triggers a VCRD *adjusting event*. The Monitoring Module running in the guest operating system kernel in the VM changes its VCRD from `LOW` to `HIGH`, and estimates the lasting time. Then the Monitoring Module sends the VCRD information to the Adaptive Scheduler with a hypercall. After the estimated lasting time, the Monitoring Module changes the VCRD of the VM from `HIGH` back to `LOW` and again notifies the scheduler.

At *assignment event*s with a certain interval of K slots, the bootstrap PCPU (i.e., the master boot PCPU) calculates the total Credit of the system according to the number of PCPUs and the length of the interval, and then allocates Credits to VCPUs of each VM in proportion to their weights. The number of Credit of a VCPU comprises the new allocated number and the last residual number.

At a *scheduling event*, a VCPU with the maximal Credit in the run queue of a PCPU will be mapped to the PCPU. If the VCRD of its VM is `HIGH`, the PCPU sends an inter-processor interrupt (IPI) to other PCPUs, whose run queues include VCPUs of the same VM. These PCPUs will temporarily raise the priority of those VCPUs, so that all VCPUs in the VM can be coscheduled. A mutex lock is used to guarantee that only one PCPU can launch an IPI for coscheduling at each *scheduling event*.

Formally, a physical computer includes a group of homogenous PCPUs, denoted by $P = \{P_0, P_1, ..., P_{|P|-1}\}$, and the number of PCPUs is $|P|$. The VMs running on the physical computer are denoted by $V = \{V_0, V_1, ..., V_{|V|-1}\}$, and $|V|$ denotes the number of VMs in the system. In the ith VM, $C(V_i) = \{v_{i0}, v_{i1}, ..., v_{i(|C(V_i)|-1)}\}$ denotes its set of VCPUs, and $|C(V_i)|$ denotes the number of VCPUs in VM V_i. The weight of a VM is an integer parameter associated with it in Xen. The weight proportion of VM V_i is denoted by $\omega(V_i)$, which represents the proportion of CPU time consumed by the VM. $\omega(V_i)$ can be calculated according to Equation (1), and $\sum\limits_i \omega(V_i) = 1$.

$$\omega(V_i) = \frac{\text{The weight of VM } V_i}{\sum\limits_{k=0}^{|V|-1} \text{The weight of VM } V_k} \tag{1}$$

In addition, the Credit obtained by a VM is equally distributed among its VCPUs, and the VCPU *online rate*, the percentage of time of the VCPU being mapped to a PCPU, of VM V_i is calculated according to Equation (2).

$$\text{VCPU online rate of VM } V_i = \frac{|P| \times \omega(V_i)}{|C(V_i)|} \tag{2}$$

4.2 Locality of synchronization

Locality of reference is a fundamental principle of computing with many applications, and it is widely used to improve system performance [17][18]. The principle of locality also exists in synchronization, because a synchronization variable may be accessed in an immediate sequence by processes or threads in a concurrent application, which is also demonstrated by Figure 2 in Section 2.2. According to observations in Section 2.2, the overhead induced by virtualization exists mainly in the synchronization primitive spinlock. Therefore, when we refer to synchronization in virtualization hereafter, it is confined to spinlocks in the guest operating system.

We extend a very general model [17][19] to describe locality of synchronization in concurrent programs. We say a spinlock is *over-threshold* when its waiting time is larger than 2^δ CPU cycles (in this paper, $\delta = 20$). As depicted in Figure 5, L_i denotes the ith locality of synchronization, and X_i denotes its lasting time, and Z_i denotes the interval of the beginning of L_i and L_{i+1}. We also have $\mathbf{X} = \{X_0, X_1, ..., X_i, ...\}$. Then we specify the locality of synchronization in concurrent programs as follows, that is, (i) over-threshold spinlocks frequently occur in a locality of synchronization, and never occur outside any locality of synchronization. Moreover, (ii) the lasting time X_i of L_i has some relationship to L_{i-1} because they may have some synchronization variables in common, and (iii) L_i and L_{i+j} tend to become uncorrelated if j becomes larger.

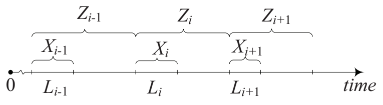


Figure 5: The locality model for synchronization

Then we can utilize the above locality model to avoid over-threshold spinlocks by coscheduling. When an over-threshold spinlock is detected in a VM, it is considered to the beginning of a locality of synchronization L_i. To mitigate the overhead of synchronization induced by virtualization, VCPUs of the VM will be coscheduled to PCPUs so that VCPUs in the virtual SMP system act similarly to CPUs in a physical SMP system, and the duration of coscheduling is X_i. However, actually the value of X_i is not known when an over-threshold spinlock is detected. Therefore, we will present an algorithm to estimate the lasting time in the following subsection.

4.3 Adjusting algorithm

An over-threshold spinlock in a VM triggers a VCRD adjusting event. At each adjusting event, we can calculate the interval between it and the last adjusting event in the VM. For estimating the duration of the next coscheduling of VCPUs in the VM, we modify a learning algorithm [20], which uses the above properties (ii) and (iii) of locality.

Firstly, consider the ith locality of synchronization L_i shown as Figure 6, which is experimentally abstracted by us from Figure 2. x_i denotes the estimated value of the lasting time of L_i, and z_i is the actual interval of L_i and L_{i+1}. If $x_i > X_i$, we have $z_i - x_i < Z_i - X_i$ as $z_i = Z_i$, and it means that VCPUs of the VM are coscheduled longer than the necessary time length (over-coscheduling), and unnecessary overhead is introduced. On the other hand, if $x_i < X_i$, an over-threshold spinlock will immediately occur once this round of coscheduling ends (under-coscheduling). Consequently, the overhead of synchronization induced by virtualization is still not avoided, and the value of $z_i - x_i$ is very small, less than a threshold $\triangle$. As a result, the ideal value of x_i is X_i, which is the goal of the learning algorithm.

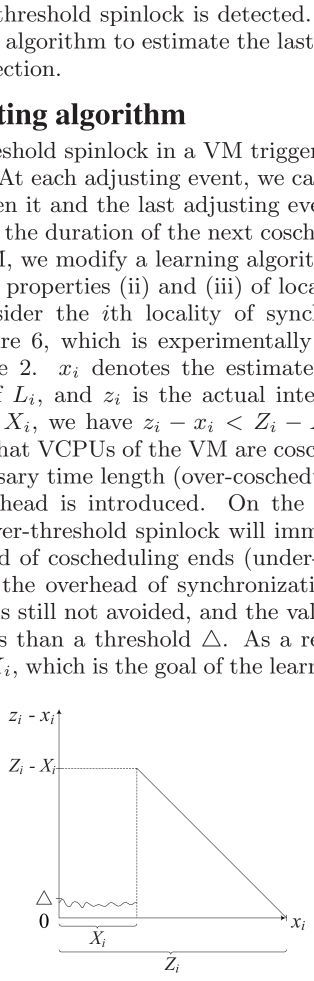


Figure 6: $(z_i - x_i) \propto x_i$

Next we turn our attention to the learning algorithm. The modified learing algorithm is calibrated with four parameters, a recency parameter r, a scaling parameter $s(0)$, an experimentation parameter e, and a parameter N denoting the number of possible values x of $\mathbf{X}$. At adjusting events, the Monitoring Module running in each VM executes the learning algorithm (Algorithm 1). At the beginning, an equal propensity $q_x(0) = s(0)A/N$ is assigned to each of all possible values of $\mathbf{X}$, and A is the statistical average value of possible values of $\mathbf{X}$. At the first two adjusting events, the Monitoring Module probabilistically selects feasible amounts x_0 and x_1 as the estimates for the following intervals. At later adjusting events, the Monitoring Module updates the corresponding parameters according to its estimation experience of the last interval, which is determined by the value of $z_i - x_i$. Once the estimated lasting time x_{i+1} is determined, the Monitoring Module updates the VCRD of the VM from `LOW` to `HIGH`, and notifies the Adaptive Scheduler with a hypercall. After the estimated time x_{i+1} without accident, the Monitoring Module changes the VCRD of the VM from `HIGH` to `LOW`, and notifies the Adaptive Scheduler.

Algorithm 1 VCRD adjusting algorithm for each VM

```
1: for VCRD adjusting event $i+1$ ($i > 0$) do
2:    for each propensity $q_x$ do
3:       $q_x(i+1) \leftarrow (1-r)q_x(i) + U(x, x_i, i, N, e)$;
4:    end for
5:    A possible value $x$ is chosen, and denoted as $x_{i+1}$, where,
      $q_{x_{i+1}}(i+1) = \max_x q_x(i+1)$;
6:    The lasting time of coscheduling is estimated as $x_{i+1}$;
7:    VCRD = HIGH;
8:    The new value of VCRD is sent to the Adaptive Scheduler;
9:    if no over-threshold spinlock occurs in the interval $x_{i+1}$
      then
10:      VCRD = LOW;
11:      The new value of VCRD is sent to the Adaptive Scheduler;
12:   else
13:      The next adjusting event is invoked;
14:   end if
15: end for
```

In Algorithm 1, $U(x, x_i, i, N, e)$ is an updating function reflecting the experience gained from the properties of locality, which is shown as Algorithm 2.

Algorithm 2 Updating algorithm for $U(x, x_i, i, N, e)$

```
1: for each $x \in \mathbf{X}$ do
2:    if $z_i - x_i \leq \Delta$ then
3:       if $x > x_i$ then
4:          $U(x, x_i, i, N, e) \leftarrow 1 - e$;
5:       else
6:          $U(x, x_i, i, N, e) \leftarrow q_x(i)\frac{e}{N-1}$;
7:       end if
8:    else
9:       if $x = x_i$ then
10:         $U(x, x_i, i, N, e) \leftarrow \frac{z_i - x_i}{z_{i-1} - x_{i-1}}(1-e)$;
11:      else
12:         $U(x, x_i, i, N, e) \leftarrow q_x(i)\frac{e}{N-1}$;
13:      end if
14:   end if
15: end for
```

4.4 Assignment algorithm

At assignment events, Credit is allocated to VMs according to the weights assigned to them, and the Credit ($Cred_{inc}$) obtained by a VM is equally distributed among its VCPUs. The Credit value of VCPU v_{ij} is $Cred(v_{ij})$. The dissipation of Credit per slot is denoted as $Cred_{unit}$, and the time length of the allocation interval is K slots. In addition, the run queue of P_i is denoted by $runq(P_i)$. Then the Credit assignment algorithm in ASMan is shown as Algorithm 3. When the VCRD of a VM changes into `HIGH`, its VCPUs have to be relocated and inserted into the run queues of different PCPUs, so that they can be coscheduled.

Algorithm 3 Credit assignment algorithm

```
1: The interval of the allocation event is $K$ slots;
2: The total Credit of the system is $Cred_{total} \leftarrow |P| \times Cred_{unit} \times K$;
3: for each VM $V_i$ do
4:    $Cred_{inc} \leftarrow Cred_{total} \times \omega(V_i)$;
5:    for each VCPU $v_{ij}$ in VM $V_i$ do
6:       $Cred(v_{ij}) \leftarrow Cred(v_{ij}) + Cred_{inc} \div |C(V_i)|$;
7:    end for
8:    if its VCRD changes from LOW to HIGH then
9:       for each VCPU $v_{ij}$ in VM $V_i$ do
10:         while $v_{ij} \in runq(P_k)$ and $v_{ij'} \in runq(P_k)$ do
11:            Choose a PCPU $P_{k'}$, so that there does not exist
               any VCPU $v_{il}$ ($l = 0, ..., |C(V_i)| - 1$) in the run
               queue of $P_{k'}$;
12:            VCPU $v_{ij'}$ is inserted into $runq(P_{k'})$ from
               $runq(P_k)$;
13:         end while
14:      end for
15:   end if
16: end for
```

4.5 Scheduling algorithm

At scheduling events, the Adaptive Scheduler chooses a VCPU for each PCPU in its run queue. The scheduling algorithm is described as Algorithm 4, and coscheduling is based on VCRD tuned adaptively by the Monitoring Module, rather than on the type of a VM set by the administrator in [12]. In addition, $VC(P_i)$ denotes the head VCPU in the run queue of PCPU P_i, and $vcrd(v_{ij})$ and $vcrd(V_i)$ both denote the VCRD of VM V_i.

Algorithm 4 Scheduling algorithm

```
1: for each PCPU $P_k$ do
2:    if $Cred(VC(P_k)) < 0$ then
3:       Determine $v_{IJ} \in runq(P_{k'})$ ($k' = 0, ..., |P| - 1, k' \neq k$), and we have $Cred(v_{IJ}) = \max_{k'} Cred(VC(P_{k'}))$, and
         $runq(P_k) \cap C(V_I) = \phi$ if $vcrd(V_I) =$ HIGH;
4:       VCPU $v_{IJ}$ is migrated to $runq(P_k)$ and scheduled to $P_k$;
5:    else if $Cred(VC(P_k)) >= 0$
      and $vcrd(VC(P_k)) =$ HIGH then
6:       Lookup all other VCPUs in the same VM as $VC(P_k)$,
         and determine their PCPUs;
7:       PCPU $P_k$ sends Inter-Percossor Interrupt (IPI) to these
         PCPUs to coschedule all VCPUs in this VM;
8:    else
9:       VCPU $VC(P_k)$ is scheduled to $P_k$;
10:   end if
11: end for
```

According to the Credit value of the VCPU at the head of the run queue of a PCPU, a scheduling decision is made, shown in Algorithm 4. If the scheduled VCPU belongs to a VM whose VCRD is `HIGH`, then the other VCPUs in that VM will be coscheduled to the corresponding PCPUs. If the Credit value of the VCPU at the head of the run queue of a PCPU is less than 0, it will not be scheduled in this slot, because its Credit had been used up in this period. A VCPU from the run queue of another PCPU may be migrated to the PCPU by the balancing mechanism. If no VCPU can be chosen for a PCPU in Algorithm 4, the idle VCPU from the idle VM (just as an idle task in the operating system) will be scheduled to the PCPU.

5. PERFORMANCE EVALUATION

To study performance comprehensively, we first test the performance of concurrent workloads on a single VM in the virtualized system, and compare ASMan with the default scheduler in Xen. Then we test the performance of concurrent and non-concurrent workloads running simultaneously on multiple VMs, and analyze the benefit and influence of the dynamic coscheduling on the performance of the whole system and individual applications.

5.1 Experimental methodology

We have selected benchmarks where both concurrent and non-concurrent workloads are available. This has enabled us to comprehensively study the performance of the VMM scheduler. Specifically, we used the NAS parallel benchmarks 2.3 [14][21], SPECjbb2005 and SPEC CPU2000 [22].

The NAS parallel benchmarks (OpenMP C Versions) are based on OpenMP, and are used here as concurrent workloads with multiple threads. We use SPECjbb2005, which is a Java program emulating a 3-tier system, to analyze the overall performance of ASMan. Since all of the three tiers of the modeled system are implemented in the same JVM, it will not generate I/O or network traffic.

SPEC CPU2000 provides a comparative measure of the compute intensive performance for a system. For testing the impact of coscheduling on non-concurrent workloads, the SPEC *rate* metric of SPEC CPU2000 is adopted [22], and we measure the throughput of a VM by running multiple copies of a benchmark simultaneously.

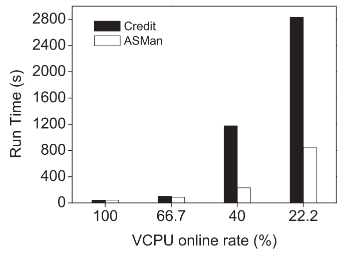
All experiments are executed on a Dell Precision T5400 workstation, with dual quad-core Xeon X5410 CPUs and 8GB of RAM. The virtualized system runs Xen 3.3.0, and all VMs run the Fedora Core 6 Linux distribution with the Linux 2.6.18 kernel.

5.2 Testing a single VM

In this testing scenario, a variety of benchmarks are run on VM V_1, which is configured with 4 VCPUs and 1024MB memory. Besides, the administrator VM V_0 (i.e. Domain-0 in Xen) is configured with 8 VCPUs and 1024MB memory, and its weight is fixed as 256, with no workload on it.

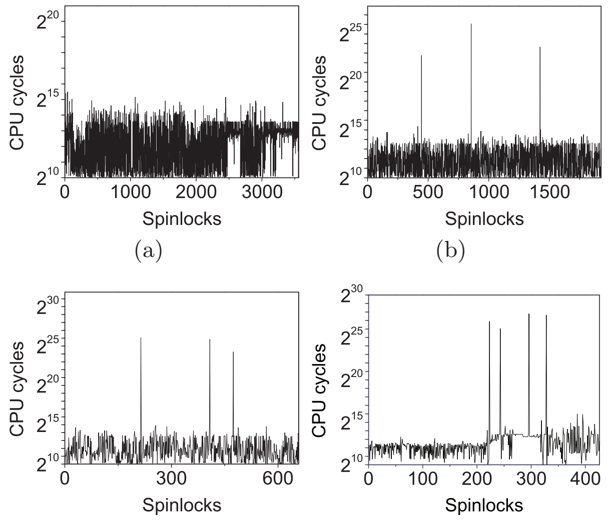
In both the default Xen with the Credit Scheduler and the modified Xen with ASMan, VM V_1 is configured to use the non work-conserving mode [23]. That is, the CPU time obtained by the VM is strictly in proportion to its weight, and it cannot receive any extra CPU time. Taking Amazon EC2 [24] for example, when a VM with 1 EC2 Compute Unit (equaling 1.0-1.2 GHz 2007 Opteron) has to be created for users on a physical server with current mainstream CPUs (such as Xeon X5680, 3.33GHz), the VCPU online rate may be about 30%. Therefore, in experiments the weight of VM V_1 is set to be 256, 128, 64, 32, respectively, consequently the VCPU online rate of the VM is 100%, 66.7%, 40%, 22.2%, respectively, according to Equations (1) and (2). The performance of benchmarks running on VM V_1 in the default Xen with the Credit Scheduler is denoted by `Credit`, while the performance of benchmarks tested in the modified Xen with ASMan is denoted by `ASMan`.

When the NAS parallel benchmarks run in VM V_1, the problem size for workloads is configured as *Class A* [21], and the other parameters are the default values. As these parallel benchmarks are CPU-bound workloads, they are not likely to run with more threads than VCPUs. Therefore, the number of the threads of each parallel benchmark is set as 4, responding to the number of VCPUs in VM V_1.



(a)

Figure 7: The run time of LU in VM V_1.



(d)

Figure 8: When LU runs in VM V_1, the detailed spinlock waiting time (also lasting for 30 seconds) in the modified Xen with ASMan is shown in (a)-(d) where the VCPU online rate is: (a) 100%, (b) 66.7%, (c) 40%, (d) 22.2%.

The run time of LU in VM V_1 with the two schedulers is shown as Figure 7, and the detailed spinlock waiting time in the modified Xen with ASMan as Figure 8. When the VCPU online rate is 100%, the concurrent workload running on a virtual SMP system gives similar performance with the two schedulers. There is an expected increase in the run time of the concurrent workload with ASMan when the VCPU online rate decreases. However, with the Credit Scheduler performance seriously deteriorates when the VCPU online rate decreases. As there are many synchronous communication operations in LU, the frequency of the VCRD of the VM being HIGH is also increasing while the VCPU online rate decreasing. However, comparing Figure 8 (a)-(d) with Figure 2 (a)-(d), it is observed that the adaptive dynamic coscheduling in ASMan avoids many over-threshold spinlocks and mitigates performance degradation introduced by virtualization to some extent.

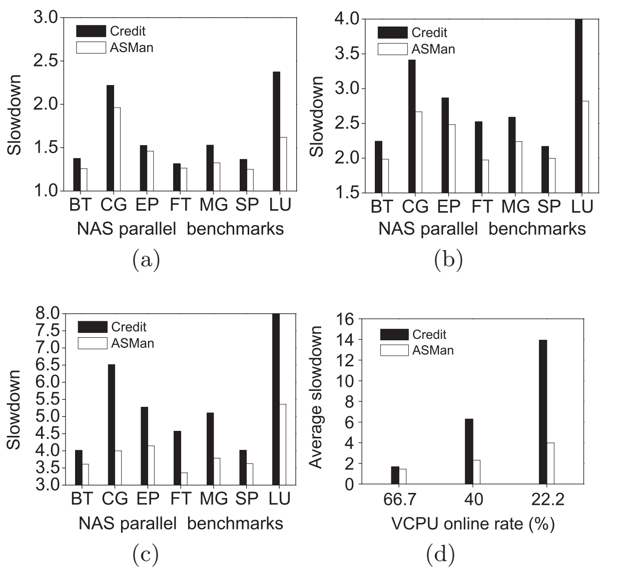


Figure 9: Slowdowns of NAS parallel benchmarks in VM V_1, where the VCPU online rate is: (a) 66.7%, (b) 40%, and (c) 22.2%; the average slowdown of all benchmarks is shown in (d).

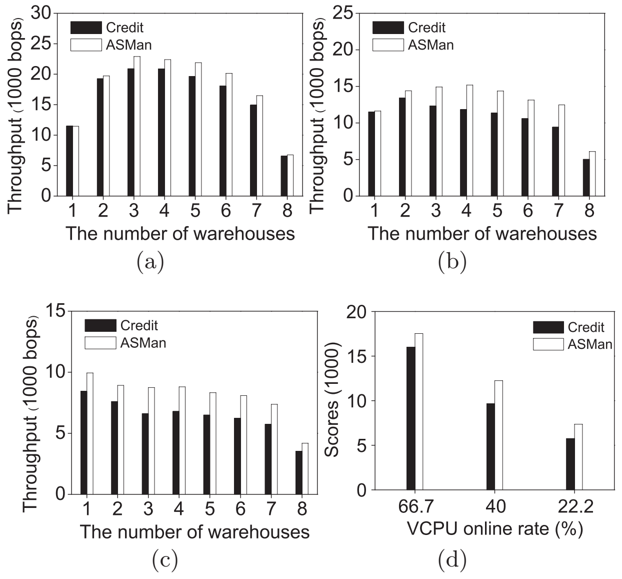


Figure 10: The throughput of SPECjbb2005 in VM V_1, where the VCPU online rate is: (a) 66.7%, (b) 40%, (c) 22.2%. The SPECjbb2005's score is shown in (d).

We also run other benchmarks in NAS parallel benchmarks on VM V_1 while the configuration is the same as LU, and test their run times while varying the VCPU online rate. In order to provide an intuitive comparison, we define the *slowdown* of a benchmark running on a VM with a VCPU online rate ($< 100\%$) as the ratio of its run time to the run time of the same benchmark running on the same VM scheduled by the Credit Scheduler with the VCPU online rate equaling 100%. As depicted in Figure 9, ASMan outperforms the Credit Scheduler in all aspects while varying benchmarks and the VCPU online rate. To be more specific, ASMan can save up to 70% of the average slowdown of all kinds of the NAS parallel benchmarks when the VCPU online rate is 22.2%.

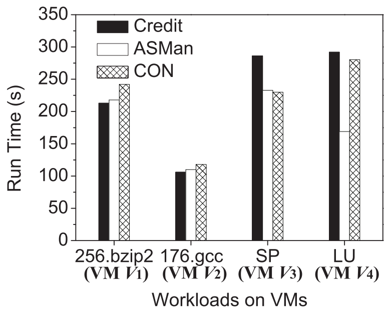
Moreover, we test the performance of the virtual SMP system (VM V_1) with SPECjbb2005, which can be used to measure the performance of CPUs, caches, memory hierarchy and the scalability of shared memory processors [22]. In experiments, a single JVM instance is adopted, and the number of warehouses increases gradually from 1 to 8. The throughput measurement is shown as Figure 10 (a)–(c), and the SPECjbb2005's score of VM V_1 is shown in Figure 10 (d). The score of each case is the average value of those throughput measurements when the number of warehouses is not less than 4 (the number of VCPUs) [22], and a higher score indicates that the VM is able to handle more Java requests and thus deliver better performance. Experimental results show that ASMan can improve throghput by up to 26% for SPECjbb2005 when the VCPU online rate is relatively low.

5.3 Testing multiple VMs

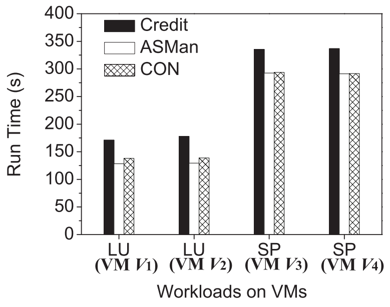
We now study the performance of workloads running simultaneously on multiple VMs. In this scenario, there are 6 VMs, VM V_1, VM V_2, ..., and VM V_6, which are configured with 4 VCPUs and 1024MB memory, respectively. We run various combinations of VMs, and their weights are fixed at 256. Besides, VM V_0 is also configured as in Section 5.2 without workload on it. In experiments, VMs are configured to use the work-conserving mode [23]. That is, the shares are merely guarantees, and a VM is eligible to receive extra CPU time if other VMs are blocked or idle. Moreover, we also compare the static coscheduling method in our previous work [12] with the adaptive dynamic coscheduling in this paper. In the static coscheduling method, one has to manually set VMs' types by the specified user interface [12]. A VM with concurrent workloads is classified as the concurrent VM, then VCPUs of the VM are coscheduled to PCPUs, and its measurement is labeled by `CON`.

There are two kinds of workloads on these VMs. One is the concurrent workload, either SP or LU from the NAS parallel benchmarks. The problem size of the two parallel benchmarks is configured as *Class A*, and the number of threads of each benchmark is set as 4. The other kind is the high-throughput workload, and 176.gcc and 256.bzip2 in SPEC CPU2000 are chosen. The SPEC rate metric [22] is adopted to measure the performance of a VM by running 4 copies of a benchmark simultaneously in the VM. As there is no synchronization between multiple copies of a benchmark, it can be treated as a kind of high-throughput workload, through which we test the impact of coscheduling on high-throughput workloads running simultaneously with concurrent workloads.

One run of a benchmark may differ from another, so multiple benchmarks will not finish at the same time when they simultaneously start to run on corresponding VMs. Therefore, we run each benchmark repeatedly with a batch program. The number of repetitions is set large enough that all other benchmarks are still running when each benchmark finishes its 10th round. The result is the average value of the run times of its first 10 rounds for each benchmark. More-



(a)



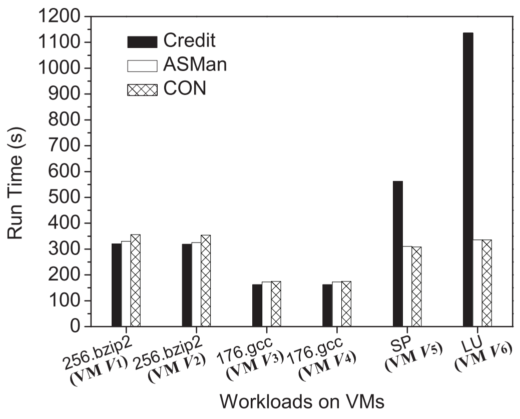
(b)

Figure 11: Four benchmarks run simultaneously on four VMs, respectively.

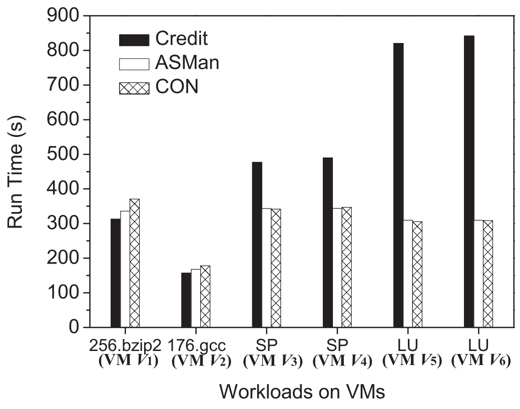
over, for each benchmark, the coefficient of variation, the ratio of the standard deviation to the mean, of its run times is less than 10%. Then the average value could be used to compare the performance.

There are four VMs in the first combination, on which two high-throughput benchmarks (176.gcc and 256.bzip2) and two concurrent benchmarks (LU and SP) run simultaneously, and the result is shown in Figure 11 (a). The second combination also includes four VMs, and four concurrent benchmarks run simultaneously on the four VMs, and the result is illustrated in Figure 11 (b). It is observed that the coscheduling of VCPUs on PCPUs has impact on the performance of the high-throughput workloads according to Figure 11 (a), and the dynamic coscheduling is beneficial to avoid unnecessary overhead of the static coscheduling, so that the run time of 176.gcc and 256.bzip2 in ASMan is less than that in CON. Figure 11 also indicates that both static and dynamic coscheduling improve the performance of concurrent workloads in VMs, either with all concurrent workloads or with mixed workloads.

To further investigate the benefits of coscheduling, we test more combinations of VMs running simultaneously in the system. As depicted in Figure 12 (a), the third combination includes six VMs, four running high-throughput benchmarks and two concurrent benchmarks. The last combination also has six VMs, but with two high-throughput benchmarks and four concurrent benchmarks running simultaneously on those VMs, as shown in Figure 12 (b). Compared with the Credit Scheduler, coscheduling can save up to 45% of the run time for SP and 70% of the run time for LU in Figure 12 (a), and save up to about 30% of the run time for SP and about 60% of the run time for LU in Figure 12 (b).



(a)



(b)

Figure 12: Six benchmarks run simultaneously on six VMs, respectively.

Moreover, compared with the Credit Scheduler, the performance degradation of high-throughput workloads in ASMan is less than 8% at worst, while it is 18% at worst in CON. It is over-coscheduling on concurrent workloads in the static coscheduling that causes more performance degradation of high-throughput workloads in CON.

5.4 Discussion

The Credit Scheduler is a non-coscheduling method, and implements proportional share fairness in Xen. When there are high-throughput workloads in VMs, it achieves good performance. However, the characteristics of concurrent applications should be considered in the virtualized system, otherwise the virtual SMP system can introduce significant overhead and cause performance degradation. This result has been validated by experiments. Compared with the Credit Scheduler, coscheduling mitigates the performance degradation of concurrent workloads while increasing the run time of high-throughput workloads to some extent. It is because a VM with a high-throughput workload is given less extra CPU time by the load balancing mechanism when coscheduling exists in the VMM. Therefore, it is a trade-off between improved performance and additional cost. However, experiments demonstrate that the performance gain for concurrent workloads is far greater than the performance loss for high-throughput workloads.

As the dynamic coscheduling avoids additional overhead caused by over-coscheduling with the static coscheduling, high-throughput workloads obtain better performance in `ASMan` than that in `CON`, while the performance of concurrent workloads is improved significantly both in `ASMan` and in `CON`. Moreover, in ASMan coscheduling depends directly on the dynamic properties of VCPUs and is executed automatically, rather than depending on static properties of workloads and being set manually in [12]. Therefore, the dynamic coscheduling in ASMan achieves a better trade-off between coscheduling and non-coscheduling in the VMM.

It should also be noted that the Monitoring Module is used in `ASMan` to bridge the semantic gap [15]. As a result, the guest operating system kernel has to be modified in `ASMan`. It is still an open issue to monitor the VCRD of a VM from outside the VM. However, the VMM may find hints from running statuses of CPUs to determine the VCRD of a VM, which will be our future work.

6. RELATED WORK

Synchronization is an important issue in operating systems, and there are many research efforts to deal with it. In particular, to avoid memory and interconnect contention by typical implementations of busy-waiting, a scalable algorithm for spinlocks is presented in [25], which generates $O(1)$ remote references at each lock acquisition, while it is independent of the number of processors that attempt to acquire the lock. Multi-Processor Restartable Critical Sections (MP-RCS) are introduced in [26]. With MP-RCS, user-level threads precisely know on which processor they are executing, so that they can safely manipulate CPU-specific data such as malloc metadata while locks or atomic instructions are never required. If preemption or migration occurs, upcalls are used to abort and restart the interrupted critical sections.

The problem of over-threshold spinlocks in virtualizaiton is defined as lock-holder preemption in [27]. To avoid lock-holder preemption, Volkman's system delays the preemption of a VCPU running in kernel space. However, the solution is only suitable for workloads that do not possess cross-processor scheduling requirements. [28] proposes a system that tolerates lock-holder preemption but tries to avoid long waits, however it just performs a preliminary attempt with kernbench.

The negative influence of virtualization on synchronization in concurrent workloads is discussed in our previous work [12], and a hybrid scheduling framework is presented to deal with the performance degradation of concurrent workloads in the virtualized system. A VM is classified as a concurrent VM when concurrent workloads run on it, and its VCPUs will be coscheduled. This implements static coscheduling of VCPUs. We have compared it with ASMan through experiments, and ASMan achieves better performance. Moreover, to use the static system, one needs enough knowledge about workloads to manually determine VMs' types.

VMware ESXi/ESX was the industry's first x86 "bare-metal" hypervisor [4], and coscheduling is used in its products [13] which is an add-on software module. VMkernel always coschedules VCPUs of a multi-VCPU VM, although it adopts a relaxed coscheduling to allow VCPUs to be scheduled on a slightly skewed basis. However, it still implements static coscheduling, and the coscheduling strategy is fixed in the module. As an open-source product, the Credit Scheduler [16] is the current default scheduler in Xen [5], and this scheduler tries to maximize the throughput of the system by automatic workload balancing of VCPUs across PCPUs, and does not attempt to improve the performance of concurrent workloads by coscheduling.

There are some application-aware scheduling methods for the VMM. A communication-aware scheduler is proposed in [29], and the SEDF scheduler in Xen counts the number of received or sent packets by each VM and preferentially schedules I/O-intensive VMs. To improve I/O responsiveness while keeping CPU fairness, a partial boosting mechanism is adopted in [30]. If a VCPU has at least one inferred I/O-bound task and an event is pending for the VCPU, the VMM initiates partial boosting for the VCPU regardless of its priority.

In addition, there are a lot of research efforts for scheduling on multi-processor or distributed systems. Gang scheduling (coscheduling) [11] is one of the major strategies for scheduling parallel jobs. Gang scheduling [31][32][33] tries to schedule related threads or processes to run simultaneously on different processors. When any of processes in a related group is scheduled, all of them will be scheduled for execution so that they can communicate efficiently. Scheduling is an important technique to address shared resource contention, and a comprehensive analysis is given for multi-core processors in [34]. Scheduling of VCPUs on PCPUs in the VMM is actually solving the contention on PCPUs between VMs, with the goal of maximizing the performance of the whole system and individual applications.

7. CONCLUSION

Virtualization has a negative influence on synchronization in guest operating systems, and this issue is especially serious when concurrent workloads such as multi-threaded or multi-process programs run in the virtualized system.

In this work, we demonstrate that virtualization can cause long waiting times for spinning synchronization, resulting in performance degradation of concurrent workloads in the virtualized system. We propose an adaptive dynamic coscheduling method to mitigate the problem, while avoiding unnecessary overhead for coscheduling. We present an adaptive scheduling framework and implement a prototype ASMan. The required code changes are highly localized and relatively small. We then compare the performance of ASMan with existing methods in a series of experiments. Results show that ASMan mitigates the negative impact of virtualization on spinning synchronization and improves the performance of concurrent workloads in the virtualized system, while keeping the performance of non-concurrent workloads.

Virtualization software is usually deployed on multi-core systems. To further improve the performance of virtualized systems while avoiding the modification of guest operating systems, the properties of the underlying architecture such as LLC (last-level cache) and the out-of-VM VCRD monitoring will be considered in our future work.

8. ACKNOWLEDGMENTS

This work was supported partly by National Natural Science Foundation of China (No. 60970008, 90715030 and

60503043), and National Key Basic Research and Development Plan (973 Plan) of China (No. 2007CB310900). We would like to thank our anonymous reviewers and our shepherd, Peter Dinda, for their thoughtful comments and suggestions.

9. REFERENCES

[1] P. H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983.

[2] J. E. Smith and R. Nair. *Virtual Machines: Versatile platforms for systems and processes*. Elsevier, USA, 2005.

[3] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[4] VMware. http://www.vmware.com/.

[5] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A.Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, October 2003.

[6] Xen. http://www.xen.org/.

[7] KVM. http://www.linux-kvm.org/page/main_page.

[8] Hyper-V. http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx.

[9] Virtualbox. http://www.virtualbox.org/.

[10] J. Dike. *User Mode Linux*. Prentice Hall, 2006.

[11] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling - a status report. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–16, 2004.

[12] 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)*, pages 111–120, 2009.

[13] VMware. VMware vSphere 4: The CPU scheduler in VMware ESX 4, 2009. http://www.vmware.com/files/pdf/perf-vsphere-cpu_scheduler.pdf.

[14] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, 1991.

[15] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

[16] Credit Scheduler. http://wiki.xensource.com/xenwiki/credit scheduler.

[17] P. J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

[18] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[19] J. R. Spirn and P. J. Denning. Experiments with program locality. In *Proceedings of the Fall Joint Computer Conference*, pages 612–621, 1972.

[20] A. Roth and I. Erev. Learning in extensive form games: Experimental data and simple dynamic models in the intermediate term. *Games and econmic behavior*, (8):164–212, 1995.

[21] NAS Parallel Benchmarks. http://www.nas.nasa.gov/resources/software/npb.html.

[22] Standard Performance Evaluation Corporation. http://www. spec.org/.

[23] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *ACM SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007.

[24] Amazon EC2. http://aws.amazon.com/ec2/.

[25] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[26] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *Proceedings of the 3rd international symposium on Memory management*, pages 163–174, 2002.

[27] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, 2004.

[28] T. Friebel and S. Biemueller. How to deal with lock holder preemption, 2008. http://www.amd64.org/fileadmin/user_upload/pub/2008-Friebel-LHP-GI_OS.pdf.

[29] S. Govindan, J. Choi, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and Co.: Communication-aware CPU management in consolidated Xen-based hosting platforms. *IEEE Transactions on Computers*, 58(8):1111–1125, 2009.

[30] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for I/O performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments (VEE)*, pages 101–110, 2009.

[31] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, 1982.

[32] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[33] D. G. Feitelson and M. A. Jette. Improved utilization and responsiveness with gang scheduling. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 238–261, 1997.

[34] Z. Sergey, B. Sergey, and F. Alexandra. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 129–142, 2010.