Synchronization-Aware Scheduling for Virtual Clusters in Cloud

Song Wu, *Member, IEEE,* Haibao Chen, Sheng Di, *Member, IEEE,*
Bingbing Zhou, Zhenjiang Xie, Hai Jin, *Senior Member, IEEE,* and Xuanhua Shi, *Member, IEEE*

Abstract—Due to high flexibility and cost-effectiveness, cloud computing is increasingly being explored as an alternative to local clusters by academic and commercial users. Recent research already confirmed the feasibility of running tightly-coupled parallel applications with virtual clusters. However, such types of applications suffer from significant performance degradation, especially as the overcommitment is common in cloud. That is, the number of executable Virtual CPUs (VCPUs) is often larger than that of available Physical CPUs (PCPUs) in the system. The performance degradation is mainly due to the fact that the current Virtual Machine Monitors (VMMs) are unaware of the synchronization requirements of the VMs which are running parallel applications.

In this paper, There are two key contributions. (1) We propose an autonomous synchronization-aware VM scheduling (SVS) algorithm, which can effectively mitigate the performance degradation of tightly-coupled parallel applications running atop them in overcommitted situation. (2) We integrate the SVS algorithm into Xen VMM scheduler, and rigorously implement a prototype. We evaluate our design on a real cluster environment with NPB benchmark and real-world trace. Experiments show that our solution attains better performance for tightly-coupled parallel applications than the state-of-the-art approaches like Xen's Credit scheduler, balance scheduling, and hybrid scheduling.

Index Terms—Virtualization, Virtual cluster, Synchronization, Scheduling, Cloud computing.

1 Introduction

Cloud computing has become a very compelling paradigm in optimizing resource utilization based on different user demands, especially due to virtualization technology. Virtualized cloud datacenter is increasingly being explored as an alternative to local clusters to run tightly-coupled parallel applications [1, 2], because of its flexibility and cost-effectiveness. However, the users still face the performance degradation problem when running such applications in cloud. This problem is mainly due to the fact that the current Virtual Machine Monitors (VMMs) are unaware of the synchronization requirements of the VMs which are running parallel applications.

Despite the existing research [3–6] on scheduling in virtualized environment, many researchers mainly focus on the *single* symmetric multiprocessing (SMP) VM that runs multi-thread applications with synchronization requirements. There is no existing research regarding the scheduling on virtual clusters that hosts tightly-coupled parallel applications. In fact, as the feasibility of running tightly-coupled parallel applications in cloud has been confirmed [7], more and more such applications are hosted by virtual clusters rather than single SMP VM, leading to more complex scheduling scenarios.

Moreover, in order to maximize cloud resource utilization, overcommitment (i.e., the number of executable Virtual CPUs (VCPUs) is larger than that of available Physical CPUs (PCPUs) in the system) is a fairly common phenomenon in cloud. For example, recent research from VMware shows that the average VCPU-to-core ratio is 4:1, based on the analysis of 17 real-world datacenters [8]. A statistical report based on Google data center [9, 10] shows that the requested resource amounts are often greater than the total capacity of Google data centers. Such overcommitted situation aggravates the performance degradation problem of parallel applications running in cloud.

This paper targets how to efficiently schedule virtual clusters hosting parallel applications in overcommitted cloud environment. We introduce a synchronization-aware approach for scheduling virtual clusters, which can effectively mitigate the performance degradation of tightly-coupled parallel applications running in overcommitted cloud environment.

The main contribution of this paper is two-fold:

- Based on experiments, we find that inter-VM communication can serve as a signal to detect the synchronization demands from the perspective of virtual cluster. We propose a *synchronization-aware VM scheduling (SVS)* algorithm based on such an observation. SVS algorithm can help VMM schedulers schedule suitable VMs on-line at runtime, avoiding the significant performance degradation of tightly-coupled parallel applications in virtual

- *S. Wu, H. Chen, Z. Xie, H. Jin, and X. Shi are with the Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Sci. and Tech., Huazhong University of Science and Technology, Wuhan 430074, China.*
  E-mail: wusong@hust.edu.cn.
- *S. Di is with Argonne National Laboratory, USA and INRIA, Grenoble, France.*
- *B. Zhou is with The University of Sydney, NSW 2006, Australia.*

clusters. Meanwhile, such a SVS algorithm suffers little overhead, because the information demanded, such as the statistics of received packets in VM is implicitly carried on demand in the communication messages of parallel applications.

- We integrate the SVS algorithm into Xen Credit scheduler, and rigorously implement a prototype. We evaluate our design on a real cluster environment using the well-known public NPB benchmark. Experiments show that our solution attains better performance for tightly-coupled parallel applications than the state-of-the-art approaches including Credit scheduler of Xen [11], balance scheduling [4], and hybrid scheduling [5].

The rest of this paper is organized as follows. We explain our motivation in detail in Section 2 followed by the description of synchronization-aware VM scheduling (SVS) algorithm in Section 3. Section 4 describes the Xen based prototype of our VMM scheduler with SVS algorithm for virtual clusters. Section 5 presents the performance evaluation results, by comparing our solution to other state-of-the-art approaches. We discuss the pros and cons of our SVS scheduler with a vision of the future work in Section 6. Section 7 summarizes the related work comprehensively. Finally, we conclude the paper in Section 8.

2 Design Motivation

In this section, we first analyze the asynchronous scheduling problem of virtual clusters running tightly-coupled parallel applications in overcommitted environment. We then discuss the disadvantage of existing solutions.

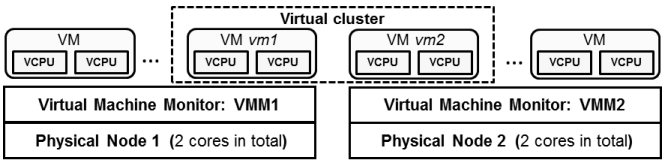


Fig. 1. Virtual cluster deployed among two physical nodes. Each VMM carries out scheduling asynchronously without considering the synchronization requirement of VMs belonging to the same virtual cluster.

Asynchronous VM scheduling method used by VMMs in multi-core physical nodes is inefficient for virtual clusters running parallel applications, when they require heavy communication in overcommitted environment.

We use Figure 1 to illustrate this problem. In this example, a 4-process tightly-coupled parallel application is running on a virtual cluster. This virtual cluster consists of two 2-VCPU VMs (*vm1* and *vm2*) that are hosted in two different physical machines (*node1* and *node2*). Suppose Xen is used as the VMM, adopting Credit scheduler [11] (a proportional-share scheduling policy). With Credit scheduler, each PCPU autonomously hosts a scheduling program and manage its own run-queue independently. That is, VCPUs in all run-queues of PCPUs are scheduled asynchronously on each physical machine. This kind of asynchronous scheduling policy usually cannot take over the lock-holder preemption problem [12]. For example, the VMM can preempt a VCPU holding a spinlock (assumed to be held for a short period of time and does not get preempted in a non-virtualized environment). This will significantly increase synchronization latency and block the progress of other VCPUs waiting to acquire the same lock. More details can be found in existing literatures [4, 5, 13].

Recently, most of existing work (e.g., co-scheduling methods of VMware [3], hybrid scheduling [5], and dynamic co-scheduling [6]) on VM-based scheduling is only designed for concurrent workload processing. That is, they aim to improve the performance for multi-thread applications with synchronization operations over SMP VM, instead of the parallel applications on virtual clusters. With the existing approaches, all VCPUs of a single SMP VM can be co-scheduled by VMM scheduler. For example, as shown in Figure 1, in a multi-processor physical machine (*node1*), VMM1 can interrupt the involved PCPUs by sending inter-processor interrupt (IPI) signals, and make them schedule two VCPUs of *vm1* at the same time, attaining a co-scheduling of VCPUs for *vm1*. Similarly, the VCPUs of *vm2* can also be co-scheduled by *VMM2*.

The key problem of these approaches is that all VMs inside a virtual cluster are scheduled asynchronously from the perspective of virtual cluster, which may degrade the performance of tightly-coupled parallel applications. As shown in Figure 1, since *VMM1* and *VMM2* make VM scheduling decisions autonomously, the probability of *vm1* and *vm2* (managed by different VMMs) being scheduled simultaneously is very low. That is, the existing scheduling methods for SMP VMs neglect the synchronization requirement of the VMs (belonging to the same virtual cluster) across physical machines. One straight-forward idea is to strictly co-schedule all VMs of a virtual cluster simultaneously among the involved VMMs via a global controller (which directs all involved VMMs when and which VMs to be scheduled). However, it is non-trivial to always guarantee such a strict co-scheduling for virtual cluster in fast-changing cloud system hosting various types of applications.

3 Our Approach

In this section, we first introduce the basic idea about the performance improvement of virtual clusters running tightly-coupled parallel applications in an overcommitted cloud environment. Then we propose the *synchronization-aware VM scheduling (SVS)* algorithm and describe how it works. At last, we analyze the design of SVS scheduler including cost, fairness, and scalability.

3.1 Basic Idea

Based on the above analysis, our objective is to design an autonomous efficient method which can schedule VMs

running tightly-coupled parallel applications according to the synchronization requirements. This issue is very tough because of no central coordination servers to be used. Fortunately, based on experiments in virtualized environment, we find that the inter-VM communication (e.g., the number of packets) can serve as a signal to detect the synchronization demands from the viewpoint of VM-level synchronization. That is, VMMs can make VM scheduling decisions based on this signal to satisfy the coordination demands of VMs belonging to the same virtual cluster. For example, as shown in Figure 1, the virtual cluster that is composed of *vm1* and *vm2* runs a tightly-coupled parallel application with four processes. Upon *vm1* receiving an amount of packets from outside between adjacent scheduling timestamps of *VMM1*, we can deduce that it probably requires synchronization among *vm1* and *vm2*. That is, *vm1* should be autonomously selected by *VMM1* to satisfy the potential synchronization requirements. Under this approach, all VMMs make VM scheduling decisions independently and no global controller among VMMs is needed.

In order to explore the correlation between the number of packets received by VM and the synchronization requirements, we investigate the Pearson Correlation Coefficients (PCC) [14] between the number of packets and the number of spinlocks (an indicator of synchronization requirement) which are obtained from the following experiment. Specifically, the PCC (denoted as r) can be computed by Formula (1), where n is the number of paired data (X_i, Y_i) recorded in the following experiment, X_i and Y_i are the number of packets and that of spinlocks recorded at the ith sampling step, respectively, $\overline{X}$ and $\overline{Y}$ refer to the mean value of X_i and Y_i, respectively. The correlation coefficient r ranges from - 1 to 1. A value that is close to 1 implies a fairly strong positive relationship between the two items, while a value of 0 means non-correlation between them.

$$r = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}} \quad (1)$$

In the experiment, four 8-VCPU VMs are used to run a set of MPI programs with 32 processes in parallel. We choose three benchmark programs (called *is*, *ep*, and *lu*) from NPB suite of version 2.4 [15], as they exhibit three typical types of parallel executions: communication intensive application with little computation (*is*); CPU intensive application with little communication (*ep*); and the one that lies in between them (*lu*). For each VM, the number of packets and that of spinlocks are recorded every 120 milliseconds (multiplying the 30ms of Xen Credit scheduler by the number of VMs in this experiment) over 60 seconds.

The statistical data is shown in Table 1. It is observed that the average Pearson Correlation Coefficients (PCC) between the number of packets and that of spinlocks in tightly-coupled parallel applications (i.e., *lu* and *is*) are 0.89 and 0.97, respectively, while that value in computation-intensive application (i.e., *ep*) is only 0.17. This implies that the inter-VM communication is a fairly good signal to detect potential synchronization requirements.

TABLE 1
The Pearson Correlation Coefficient (PCC) between the number of packets a VM received and that of spinlocks during benchmarks running

Benchmarks	*lu*	*is*	*ep*
Pearson Correlation Coefficient (PCC)	0.89	0.97	0.17
Standard Deviation of PCC	0.094	0.046	0.24

Based on the finding about the correlation between the number of packets received by VM and the synchronization requirements, our basic idea is to devise a *synchronization-aware VM scheduling (SVS)* algorithm to help make scheduling decisions for VMMs. Specifically, for each VM of virtual cluster that runs tightly-coupled parallel application, we will count the number of packets received since its last de-scheduled moment, which can be recognized and counted by VMM itself. And then, we take the statistics of packets as one of four metrics (which will be introduced in Section 3.2) to select and schedule VMs. The schedulers of VMMs with our SVS algorithm can take into account the synchronization requirement among VMs, and autonomously determine which VM should be scheduled from their own perspective, so as to improve the performance of tightly-coupled parallel application running in virtual cluster. This contrasts the strict co-scheduling approach for virtual cluster, which needs global controller for VMMs to schedule all VMs of virtual cluster online and offline at the same time.

3.2 SVS Algorithm

Based on the observation presented above, SVS algorithm promotes the scheduling priority of the VM (running tightly-coupled parallel application) with the largest number of received packets counted from the last de-scheduled moment. Intuitively, when a tightly-coupled parallel application is running in a virtual cluster, the more packets a VM receives during the last scheduling period, the more urgent the synchronization requirement of this VM is. Hence, the objective of the SVS algorithm is to differentiate the priority of the VMs which receive different amounts of packets during the last scheduling period. Specifically, the VM receiving more packets will have higher chance to be promoted on its priority for satisfying synchronization requirement of VMs belonging to the same virtual cluster.

Note that we should also guarantee the fairness of the resource consumption among VMs when promoting the priorities for some of them, thus our SVS algorithm also takes into account three other metrics in addition to the number of packets: (1) the type of the application running on VM, (2) the remaining CPU shares (e.g., the remaining credit values of Xen), and (3) the original orders in the run-queue of PCPU.

Basically, there are four situations to deal with, according to the information of PCPU run-queue, as listed below.

- If there is no VM that runs tightly-coupled parallel application in the run-queue of PCPU, SVS algorithm will select the VM at the head of run-queue directly, just like the Credit scheduler of Xen does.
- If there exists only one VM that runs parallel application in the run-queue of PCPU, SVS algorithm will select that VM without any hesitation.
- If there are more than one VM, which runs some tightly-coupled parallel application in the run-queue of PCPU, SVS algorithm will select the VM that receives the largest number of packets counted from the last de-scheduled moment. Further more, if two VMs receive the same number of packets, their remaining CPU shares (e.g., remaining credit values of Xen) will be used to carry out the VM selection, and the more the remaining CPU shares, the higher the priority.
- If the VMs still cannot be differentiated, SVS algorithm will pick a VM from among all qualified VMs according to their original orders in the run-queue of PCPU.

TABLE 2
Variables and Functions in Algorithm

Variables and Functions	Description
v	a VCPU
$runq$	the run-queue of PCPU
$VM(v)$	the VM to which the v belongs
$pcpuSet$	a set of PCPUs
$vcpuSet$	a set of VCPUs
$vmSet$	a set of VMs
do_loadbalance()	migrate VCPU residing in the run-queue of neighboring PCPUs once there are no VCPUs in the local run-queue
get_first_element($runq$)	return the VCPU which is at the head of run-queue $runq$
get_parallel_vcpus_with_ maxPackets($runq$)	return VCPUs from $runq$ if parallel VMs they belong to receive maximum packets since their last de-scheduled
get_vcpus_with_ maxRemainingShares($vcpuSet$)	return VCPUs from VCPUs set $vcpuSet$ if they have the most remaining CPU shares
get_vcpu_with_ oOrderinRunq($vcpuSet$)	return a VCPU from VCPUs set $vcpuSet$ according to their original order in $runq$
get_pcpus_of_vm($VM(v)$)	return all PCPUs where the VCPUs of $VM(v)$ are allocated
get_pcpu(v)	return the PCPU which hosts v
get_running_vm($pcpuSet$)	return all VMs which are running in $pcpuSet$

After SVS scheduler determines which VM should be scheduled for running the parallel application, the VMM scheduler will schedule all VCPUs of the VM simultaneously by sending Inter-Processor Interrupt (IPI) signals to the involved PCPUs on the same physical machine. Unrestricted simultaneous scheduling all VCPUs for SMP VM through sending IPIs, however, may cause excessive numbers of preemptions due to repeatedly interrupting VMs, which results in serious performance degradation [16, 17]. To mitigate this problem with unexpected preemptions, we devise a VM preemption mechanism. For tightly-coupled parallel applications, it is very important to timely handle network packets, thus it is necessary to prevent the preemption during such a time period. In other words, the duration for handling packets by each VM can be treated as a threshold to determine whether the VM can be preempted. We calculate such a threshold value for each VM (denoted as $\mathcal{T}_{VM}$) based on the estimated time interval a VM will take to handle packets in next run, as shown in Formula (2).

$$\mathcal{T}_{VM} = Num_{VM}^{Packets} \times AveTime_{VM}^{OnePacket} \quad (2)$$

$\mathcal{T}_{VM}$ is a product of two metrics: 1) the number of packets (denoted as $Num_{VM}^{Packets}$)) that a VM received since its last de-scheduled moment (these packets should be handled during its next run), and 2) the time spent in handling one packet (denoted as $AveTime_{VM}^{OnePacket}$). Specifically, the first metric (the number of packets) for each VM can be counted in the driver domain of VMM (e.g., Domain 0 of Xen). Because the packets that a VM received since its last de-scheduled moment may have different size, the time spent in handling a packet may vary. We estimate the second metric by averaging the historic monitoring data about handling one packet. When the VM is scheduled to run for the first time, the $\mathcal{T}_{VM}$ is an empirical value by default. In Section 5, it is set to 2ms in our experiments, because the tick time of Xen Credit scheduler is 10ms.

The pseudo-code of SVS algorithm is described in Algorithm 1. We call a VM running non-parallel (parallel) application "non-parallel VM" ("parallel VM"). Similarly, we call its VCPUs the "non-parallel VCPU" ("parallel VCPU"). The variables and functions used in Algorithm 1 are described in Table 2. The SVS algorithm takes run-queue information as an input. If there are no runnable VCPUs (line 1), `do_loadbalance()` will be called to balance the VCPUs' load across PCPUs on a host with multiple processors. That is, once there are no VCPUs in the run-queue of a PCPU, this function migrates a VCPU residing in the run-queue of its neighboring PCPUs. (lines 2 and 3). Otherwise, if runnable VCPUs are only non-parallel ones, our algorithm will act like Credit scheduler of Xen, that is, it selects the VCPU which is at the head of run-queue (lines 4-6).

Four functions are used to get a suitable VCPU when runnable VCPUs include parallel ones (lines 8-19). First, it calls `get_parallel_vcpus_with_maxPackets()` to obtain VCPUs, which are put in $vcpuSet1$, from run queue *runq* if parallel VMs they belongs to receive maximum packets since their last de-scheduled (line 8). If there is more than one VCPU in $vcpuSet1$, function `get_vcpus_with_maxRemainingShares()` will be called to return VCPUs according to their remaining CPU shares (line 13). If more than one VCPU has the most remaining CPU shares, `get_vcpu_with_oOrderinRunq()` will be used to

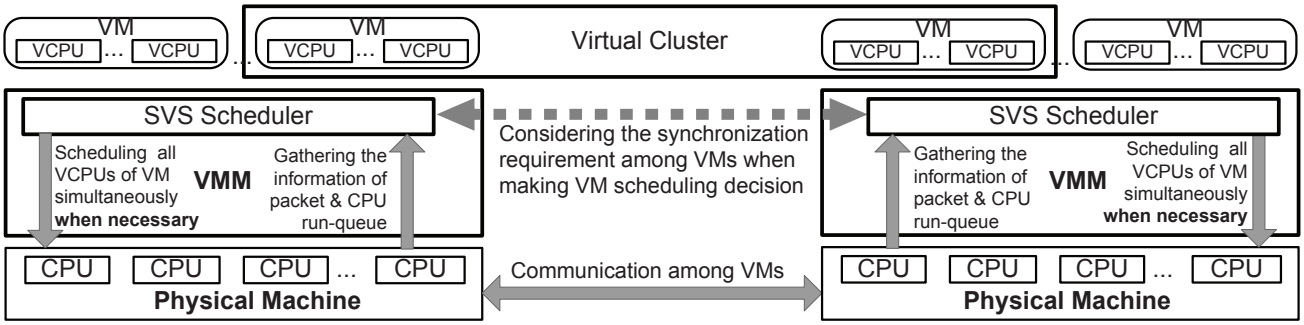


Fig. 2. Overview of SVS scheduler. With SVS algorithm, the scheduler can take into account the synchronization requirement among VMs which belong to the same cluster when scheduling VMs. Meanwhile, with the preemption mechanism, it can avoid over context-switch.

Algorithm 1 The synchronization-aware VM scheduling (SVS) algorithm

Input: run-queue information of the PCPU where the scheduler resides
Output: scheduling decision
1: **if** there are no runnable VCPUs **then**
2: $v =$ do_loadbalance();
3: $v.type = LOADBALANCE$;
4: **else if** runnable VCPUs are only non-parallel ones **then**
5: $v =$get_first_element($runq$);
6: $v.type = NONPARALLEL$;
7: **else** /* runnable VCPUs include parallel ones */
8: $vcpuSet1 =$get_parallel_vcpus_with_maxPackets($runq$);
9: **if** there is only one VCPU in $vcpuSet1$ **then**
10: $v =$get_first_element($vcpuSet1$);
11: $v.type = PARALLEL$;
12: **else** /* more than one VCPU has max. packets */
13: $vcpuSet2 =$get_vcpus_with_maxRemainingShares($vcpuSet1$);
14: **if** only one VCPU exists in $vcpuSet2$ **then**
15: $v =$get_first_element($vcpuSet2$);
16: $v.type = PARALLEL$;
17: **else** /* more than one VCPU has max. CPU shares */
18: $v =$get_vcpu_with_oOrderinRunq($vcpuSet2$);
19: $v.type = PARALLEL$;
20: **end if**
21: **end if**
22: **end if**
23: **if** $v.type = PARALLEL$ **then**
24: $pcpuSet =$get_pcpus_of_vm($VM(v)$);
25: $pcpuSet = pcpuSet -$ get_pcpu(v);
26: $vmSet =$get_running_vm($pcpuSet$);
27: **if** $VM(v)$ can preempt all VMs in $vmSet$ **then**
28: send Inter-Processor Interrupt (IPI) to these PCPUs in $pcpuSet$ and schedule VCPUs of $VM(v)$ to PCPUs simultaneously;
29: **else**
30: schedule v to PCPU;
31: **end if**
32: **else**
33: schedule v to PCPU;
34: **end if**

return a VCPU based on VCPUs' original order in run-queue *runq* (line 18).

After obtaining the selected VCPU v, scheduling decision is made based on the following two alternative decisions: (1) schedule all VCPUs of $VM(v)$ to their corresponding PCPUs simultaneously; or (2) only schedule VCPU v to PCPU. If $VM(v)$ runs a tightly-coupled parallel application and can preempt the running VMs, we choose the first scheduling decision, or we choose the second one otherwise. Specifically, VM preemption mechanism is used to determine whether $VM(v)$ can preempt the running VMs in $vmSet$ when it runs parallel application (line 27). And if satisfied, all VCPUs of $VM(v)$ are scheduled to their corresponding PCPUs simultaneously by sending Inter-Processor Interrupt (IPI) (line 28).

3.3 SVS Scheduler

Based on SVS algorithm, we design our SVS scheduler in this section, the overview of which is presented by giving an example of two nodes as shown in Figure 2.

The SVS scheduler monitors the communication states inside each VMM, and dynamically analyzes the statistics of received packets. The monitored communication state is driven by the running parallel application, and we call it *locally visible synchronization requirement information*. With such information, our SVS scheduler can take the synchronization requirement into consideration when scheduling VMs. Meanwhile, SVS scheduler suffers little overheads, because the coordination information demanded (i.e., the statistics of received packets in VM) is implicitly carried in the communication messages. As for the intra-VM scheduling, all VCPUs of each SMP VM can be scheduled at the same time by sending Inter-Processor-Interrupt (IPI) to involved PCPUs when demanded.

Cost analysis. The space complexity is $O(Q)$, and the computational complexity is $O(Qlog_2Q)$, where Q is the number of VCPUs in the run-queue of each PCPU. This is because SVS scheduler selects the scheduling object by sorting PCPU's run-queue with merge sort. As the value of Q grows, so does the computational complexity of SVS Scheduler. However, in real environment, the value of Q is very small. For example, VMware shows that the average value is 4 based on analyzing 17 real-world datacenters [8].

It should be noted that, if the selected VCPU and its siblings which need to be co-scheduled cannot preempt the VCPUs occupying the related PCPUs, SVS scheduler

will only schedule the selected VCPU to PCPU (lines 27-31 of Algorithm 1). That is, there is no extra cost in such situation, because SVS scheduler does not need to reselect a VCPU from the run-queue of PCPU.

Fairness guarantee. One important feature of our SVS scheduler is improving the performance of tightly-coupled parallel application by adjusting the scheduling order according to some key metrics, such as type of applications, the number of received packets, and remaining shares (e.g., credits in Xen). Our scheduler does not degrade the scheduling fairness of the overall system, because it adopts proportional-share scheduling strategy for VMs running different types of applications, which allocates CPU in proportion to the amount of shares (weights) that VMs have been assigned, to guarantee the fairness among VMs over time. This strategy is widely used by VMMs such as Xen. The key part of this strategy is the concept of weight, while the CPU time obtained by a VM is equally distributed among its VCPUs. In Section 5.4, an experiment is used to test the fairness among VMs.

Scalability. Due to our autonomous design, all VMMs with SVS scheduler make VM scheduling decisions based on their locally visible synchronization requirement information (e.g., the statistic about packets received by each VM), implying a higher scalability compared to the strict co-scheduling method of virtual cluster with centralized controller among VMMs. Such a feature makes it better suitable to the complex and fast-changing cloud environment. And we also evaluate the scalability of our method in Section 5.2.

4 IMPLEMENTATION

It is convenient to apply our design in practice. SVS scheduler just requires a simple modification to the VM scheduler in the VMM. It is generic and thus applicable to many VMMs (e.g., Xen, VMware ESX [18]).

The prerequisite of all co-scheduling algorithms is to know the type of workload running in VMs. That is, the scheduler must understand whether the workload is parallel application or not. Here, we adopt our previous method of inferring the type of application in virtualized system [19].

We implement a working prototype, called *sched_SVS*, by extending the Credit scheduler of Xen 3.2 because of its open-source codes. Before we describe the implementation of SVS scheduler, we give an overview of the working mechanism of Credit scheduler as follows.

Credit scheduler is currently the default scheduler in Xen. It allocates the CPU resources to VCPU according to the weight of the domain that the VCPU belongs to. It uses credits to track VCPU's execution time. Each VCPU has its own credits. If one VCPU has credits greater than 0, it gets UNDER priority. When it is scheduled to run, its credit is deducted by 100 every time it receives a scheduler interrupt that occurs periodically once every 10ms (called a tick). If one VCPU's credit is less than 0, its priority is set to OVER. All VCPUs waiting in the run-queue have their credits topped up once every 30ms, according to their weights. The higher weight a domain has, the more credits are topped up for its VCPUs every time. An important feature of the Credit scheduler is that it can automatically load-balance the VCPUs across PCPUs on a host with multiple processors. The scheduler on each PCPU can "steal" VCPUs residing in the run-queue of its neighboring PCPUs once there are no VCPUs in its local run-queue.

Since SVS scheduler is based on the Credit scheduler, SVS scheduler inherits its proportional fairness policy and multi-core support. SVS scheduler mainly consists of three modules (VCPU initial mapping, load balancing, synchronization-aware VM scheduling), as shown in Figure 3.

Through extending the VCPU initial mapping, load balancing, and scheduling mechanisms of Credit scheduler, we implement the SVS scheduler modules of VCPU initial mapping, load balancing, and synchronization-aware VM scheduling.

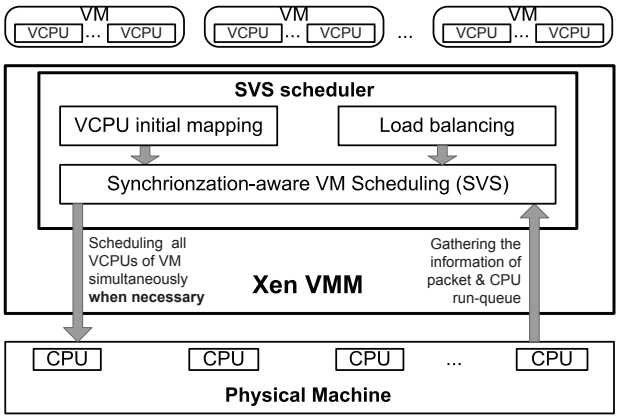


Fig. 3. Modules of SVS scheduler based on the Credit Scheduler of Xen.

VCPU initial mapping. When a VM is created, each VCPU of the VM will be inserted to the run-queue of a PCPU by this module. As for the type of SMP VMs that run tightly-coupled parallel application, we implement this module by balancing all VCPUs of a SMP VM to different run-queue of PCPU. That is, there are no two VCPUs of a VM associated to the same PCPU's run-queue when the VM is running. As for other types of VMs, we map their VCPUs like the Credit scheduler does. Through implementing such kind of mapping strategy, it not only can reduce the cost of co-scheduling VCPUs, but also resolve the VCPU-stacking problem presented in [4] (that is, the lock waiter is scheduled before the lock holder on the same PCPU) for tightly-coupled parallel application running on SMP VM.

Load balancing. With this module, SVS scheduler can automatically perform local balancing among its local run-queue and the ones of its neighbor PCPUs of the same physical node. Specifically, the VCPUs residing in

the neighbor PCPUs' run-queues can be dynamically migrated into the local run-queue on demand. Similar to VCPU initial mapping, we implement this module by extending the load balancing mechanism of Credit scheduler to guarantee that no two VCPUs of a VM exist in the same PCPU's run-queue, when making the decision of migrating VCPUs due to the same reasons.

Synchronization-aware VM scheduling. We implement this module based on SVS algorithm (described in Section 3.2) through extending Credit scheduler of Xen. For example, we add a variable (*domu_rx_packets*) and an array (*domu_handling_time[MAX_SIZE]*) for each VM to record information, where *MAX_SIZE* is the size of historic monitoring data about time spent in handling one packet. Specifically, *domu_rx_packets* is used to record the number of packets that a VM has received since its last de-scheduled moment, which is taken as one of metrics to schedule VM. The *domu_handling_time[MAX_SIZE]* make us look backwards last *MAX_SIZE* data, and the time used in computing the threshold of VM preemption mechanism is the average of all data in *domu_handling_time[MAX_SIZE].*

In Xen, the Xen-netback is responsible for receiving packets and forwarding them to corresponding VMs. We add an instruction in Xen-netback to increment the $domu\_rx\_packet$ variable for receiving every packet. The computational complexity of this operation is $O(1)$ - a constant, so it does not introduce extra overhead in the system at all. Similarly, the operation of array (i.e., $domu\_handling\_time[MAX\_SIZE]$) has no impact on the computational complexity of system either. Hence, the packet analysis suffers from little overhead in the system.

5 Performance Evaluation

We first describe our experimental methodology in Section 5.1, and then present experimental results about SVS approach in the following sections.

5.1 Experimental Methodology

(1) Experimental Platform. A private infrastructure is deployed in our university. It is used as our main experimental platform because commercial cloud providers (e.g., Amazon EC2) do not allow users to modify their VMM schedulers. In this experimental platform, there are 32 nodes with totally 256 cores connected by a 1Gbps Ethernet. Each physical node is equipped with two Intel Xeon E5345 quadcore CPU and 8 GB of DDR2 DRAM. These nodes run Xen 3.2, and all VMs (4-VCPU) run the CentOS5.5 Linux distribution with a Linux 2.6.18 kernel. Although open source cloud computing software (e.g., Eucalyptus [20] and CloudStack [21]) can be used to build and manage our private infrastructure, we do not adopt it because it will introduce extra problems of cloud resource scheduling (e.g., how to place or migrate VMs in cloud), which probably make our study more complex and affect the evaluation of our work in VMM. Note that in this paper, we focus on the scheduling problem in VMM rather than that of cloud scheduler.

(2) Scheduling approaches. On this platform, we compare our SVS approach to three other state-of-the-art scheduling approaches as follows:

- **CREDIT**: the default scheduler of Xen.
- **H**ybrid **S**cheduling (HS): It adopts two different algorithms correspondingly for two types of VMs - high-throughput VM and concurrent VM that are classified according to the type of application running on VM. For high-throughput VM, it uses proportional share scheduling algorithm. For concurrent VM, it uses co-proportional share scheduling algorithm, which co-schedules the VCPUs of SMP VM to the PCPUs in the physical node while guaranteeing that the CPU time is allocated to VMs in proportion to their weight values.
- **B**alance **S**cheduling (BS): To remediate the synchronization latency problem faced by the open source hypervisors (e.g., Xen) with the default schedulers, **B**alance **S**cheduling approach balances VCPU siblings of SMP VM on different PCPUs without strictly scheduling the VCPUs simultaneously. That is, it is a probabilistic co-scheduling approach.

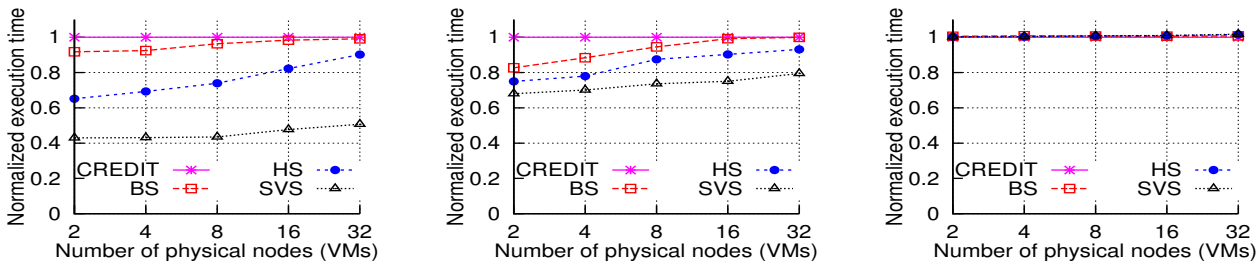
(3) Classification of experiments. There are many factors in the platform to affect the approaches' performance, such as the degree of overcommitment (that is, the ratio of VCPU-to-PCPU), the size and number as well as placement of virtual clusters running parallel applications, and the interference of other types of applications.

In order to evaluate the performance of approaches clearly, we first devise a test in Section 5.2 with some restrictions, e.g., using the given configuration (the size, number, and placement) of virtual clusters, which is split into two parts. The first one fixes the ratio of VCPU-to-PCPU (i.e., fixing the number of VMs hosted on each physical node), and investigates the performance with different scales (different numbers of physical nodes). The second one fixes the number of physical nodes and dynamically adjusts the ratio of VCPU-to-PCPU by changing the number of VMs hosted on each physical node at runtime.

Then, we release these restrictions about the configuration of virtual cluster in Section 5.3 to evaluate the performance of different scheduling approaches, where we synthesize the size and number of virtual clusters launched in cloud environment based on the job traces of a Linux cluster (Thunder) at Lawrence Livermore National Laboratory [22]. This test is also split into two parts. The first one only runs parallel applications. The second one adds web applications and disk-intensive applications as interference workload into the platform which are very common applications in cloud environment. At last, we evaluate the CPU fairness of our approach in Section 5.4.

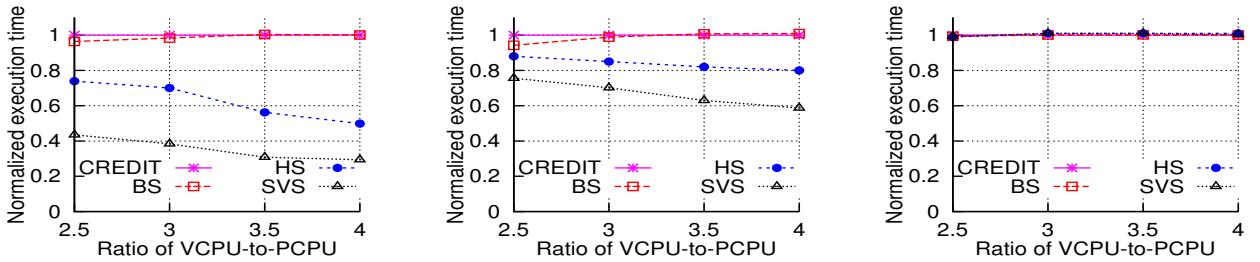
(4) Benchmarks. The benchmarks used in our experiments are *NPB suite*, *Web server*, and *Bonnie++* [23].

- *NPB suite* of version 2.4. A set of MPI programs de-



(a) *lu* (b) *is* (c) *ep*

Fig. 4. Performance comparison of approaches (CREDIT, BS, HS, and SVS) with fixed ratio of VCPU to PCPU when running benchmarks on 2, 4, 8, 16, and 32 nodes (VMs).



(a) *lu* (b) *is* (c) *ep*

Fig. 5. Performance comparison of approaches (CREDIT, BS, HS, and SVS) when running benchmarks with different ratios of VCPU-to-PCPU.

signed to help evaluate the performance of clusters and supercomputers.

- *Web server*. The average response time of Web server is measured by *httperf* [24].
- *Bonnie++*. A benchmark suite that aims at performing a number of simple tests of hard drive and file system performance.

In general, cloud users construct the virtual cluster for their own individual purposes, so we assume without loss of generality that each virtual cluster runs one application in all of our experiments.

5.2 Scenarios with Restrictions

The benchmark employed is *NPB suite* of version 2.4. And we select three benchmark programs (*is* with size A, *ep* with size C, and *lu* with size A) from *NPB* for experiment because they exhibit three typical types of parallel executions: communication intensive application with little computation (*is*); CPU intensive application with little communication (*ep*); and the one that lies in between them (*lu*).

5.2.1 Fixed Ratio of VCPU to PCPU

In this experiment, we scale the number of physical nodes from 2 to 32 (2, 4, 8, 16, and 32), and four 4-VCPU VMs are booted up on each physical node. The fixed VCPU-to-PCPU ratio is 2.5:1 when considering the number of VCPUs in privileged domain (domain 0). Four identical virtual clusters are built using all VMs in the platform, and the four VMs on each physical node belong to them separately. We run *lu* on these four virtual clusters simultaneously for ten times, and record the execution time of *lu* on each virtual cluster. The same test procedures also go to *is* and *ep*, respectively.

Figure 4 shows the average execution time of *lu*, *is*, and *ep* running on virtual clusters with different solutions BS, HS and SVS. Their execution times are all normalized by comparing to that of traditional approach CREDIT.

Based on Figure 4(a) and 4(b), it is clearly observed that our SVS approach exhibits the best performance and scalability for *lu* and *is*. For example, in absolute terms, the execution times of *lu* application under HS and BS approaches are longer than that under our SVS approach by 0.65/0.42=155% and 0.92/0.42=219% when the number of physical nodes (that is, the number of VMs in virtual cluster) is 2. The performance and scalability of HS is much better than that of BS. We analyze the key reasons as follows.

- For tightly-coupled parallel applications (e.g., *lu* and *is*), our SVS scheduler outperforms the other three approaches (BS, HS and CREDIT) and scales better because it considers the synchronization requirements of the VMs that belong to the same virtual cluster when making VM scheduling decisions. Moreover, with our designed SVS algorithm, VMMs can autonomously determine which VM should be scheduled.
- BS is a probabilistic co-scheduling approach [4], and the probability of co-scheduling VCPUs of virtual cluster will become lower and lower with increasing number of physical nodes (VMs of virtual cluster). Thus, BS has a slight performance gain over CREDIT when the number of physical nodes is small (e.g., 2),

while the performance gain is not clear with large number of nodes.

- Although HS co-schedules all VCPUs of SMP VMs on single physical node, all VMs belonging to the same virtual cluster are still scheduled asynchronously because involved VMMs neglect the synchronization requirements among VMs when making scheduling decision. Therefore, the performance and scalability of HS are between these of SVS and BS, that is, HS is better than BS, but worse than our SVS solution.

From Figure 4(c), we can observe that these four approaches have almost the same performance and scalability. The reason is that SVS and HS will gracefully degrade into CREDIT with respect to the CPU intensive applications with little communication (e.g., *ep*). In this situation, all of the three solutions will exhibit almost the same performance, which is extremely close to the classic CREDIT approach.

5.2.2 Varying Ratios of VCPU-to-PCPU

In this experiment, we dynamically adjust the ratio of VCPU-to-PCPU from 2.5 to 4 by changing the number of VMs hosted on each physical node of platform from 4 to 7. As the configuration of virtual clusters in Section 5.2.1, VMs on each physical node belong to different virtual clusters separately. Figure 5 presents the average execution time of *lu*, *is*, and *ep* when running on virtual clusters with BS, HS and SVS, respectively. The execution time is also normalized in comparison to that of CREDIT.

From Figure 5(a) and 5(b), we can easily observe that our SVS approach has the best performance for *lu* and *is* in scenarios with different ratios of VCPU-to-PCPU, and the performance of HS are between these of SVS and BS. Specifically, the performance of SVS and HS become better with increasing ratio of VCPU-to-PCPU, while the performance gain of BS over CREDIT is not obvious at all. The reasons behind these figures are as follows.

- As the ratio of VCPU-to-PCPU increasing, the probability of co-scheduling VCPUs of virtual cluster using BS approach becomes lower and lower. Therefore, the performance gain over CREDIT is not clear in such situation.
- HS outperforms BS and CREDIT because it can co-schedule the VCPUs of SMP VM that runs tightly-coupled parallel application. However, it is still worse than our SVS approach due to the fact that involved VMMs with HS neglect the synchronization requirements among VMs when making scheduling decision.

From Figure 5(c), we observe that these approaches have almost the same performance and scalability for *ep* as the ratio of VCPU-to-PCPU changing, which is due to the same reasons for Figure 4(c) in Section 5.2.1.

5.3 Scenarios without Restrictions

In the real work, it is very common for cloud systems (e.g., Amazon EC2) to host different types of applications with different sizes. Therefore, we devise a test to carry out further evaluation of scheduling approaches (i.e., CREDIT, HS, BS, and SVS), where we synthesize the size and number of virtual clusters launched in our platform based on the job traces of a Linux cluster (Thunder) at Lawrence Livermore National Laboratory [22]. And we adopt a random placement algorithm to place these virtual clusters in this test. Besides, the type of the application running on virtual clusters is selected randomly from several typical benchmarks. That is, different from experiment in Section 5.2, we release restrictions about the configuration of virtual clusters and the type of application running on them.

The test in this section is split into two parts. The first part (in Section 5.3.1) only runs different type of parallel applications. The other one (in Section 5.3.2) adds web applications and disk-intensive applications as interference workload into the platform, which are very popular applications in cloud environment. In these two parts, 128 4-VCPU VMs are hosted on 32 physical nodes (each node hosts four 4-VCPU VMs). The VCPU-to-PCPU ratio is 2.5:1 when considering the number of VCPUs in privileged domain (domain 0), which means a fairly overcommitted situation.

5.3.1 Mix of Parallel Applications

First, we evaluate the performance of approaches using a mix of different parallel applications. We will add non-parallel applications into the platform, and do further evaluation in Section 5.3.2.

(1) The configuration of virtual clusters. Different from the restricted configuration of virtual clusters in Section 5.2, we synthesize the size and number of virtual clusters based on the job traces of a Linux cluster (Thunder) at Lawrence Livermore National Laboratory (LLNL).

Specifically, we make the size of each virtual cluster in the platform lie in the range between 8 and 128 VCPUs. Such a setting is due to the fact that the number of CPUs required by majority (about 91.5% in Table 3) of jobs is no more than 128, according to the job traces of a Linux cluster (Thunder).

TABLE 3
The percentage of the number of jobs with different sizes (i.e., the number of CPUs required by jobs) based on the traces of Linux cluster (Thunder)

Size	*4*	*8*	*16*	*32*	*64*	*128*	*others*
Percentage	48.5%	12.2%	13.8%	4.7%	8.4%	3.9%	8.5%

Based on the total number of VMs in our platform, we also try to make the distribution of virtual clusters be consistent with the trace. For example, the number of virtual clusters with 8 VCPUs (12.2%) is about 3 times much more than that of ones with 128 VCPUs (3.9%). Specifically, from the 128 VMs, ninety are used to build the 10 virtual clusters with different sizes, and the other thirty-eight VMs act as independent VMs. Therefore, the

10 virtual clusters are organized as follows, according to the trace data in Table 3.

- Three 8-VCPU virtual clusters (labeled as *VC1* ∼ *VC3*, respectively), each of which has two 4-VCPU VMs.
- Three 16-VCPU virtual clusters (*VC4* ∼ *VC6*).
- One 32-VCPU cluster (*VC7*).
- Two 64-VCPU virtual clusters (*VC8* and *VC9*).
- One 128-VCPU virtual cluster (*VC10*).

(2) The placement of virtual clusters. The placement of virtual clusters is a non-trivial issue, and more details about various policies of virtual cluster placement can be found in [25]. This part is beyond the scope of this paper, so we just adopt a random placement algorithm in our experiment here to place these 10 virtual clusters and 38 independent VMs.

(3) Benchmarks. In this experiment, each of virtual clusters and independent VMs runs a randomly selected application from *lu.A*, *is.A*, and *ep.C* of NPB benchmark. One run of an application may differ from another, so multiple applications will not finish at the same time when they simultaneously start to run on corresponding virtual clusters and independent VMs. Therefore, we run each application repeatedly with a batch program. And the number of repetitions is set large enough that all other applications are still running when each application finishes its 10th round.

(4) Experimental results. Figure 6 shows the average execution time of applications running in each virtual cluster and independent VM with BS, HS and SVS normalized to that with CREDIT. The application names in the parentheses indicate which virtual cluster (VC) is running which application based on the random assignment. For each application, the coefficient of variation of its run times is less than 10%. Then the average value could be used to compare the performance.

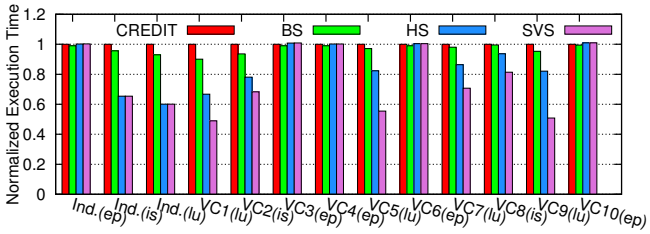


Fig. 6. Normalized execution time of applications (*lu*, *is*, and *ep*) running on ten virtual clusters of different sizes and independent VMs in our platform.

From Figure 6, we can see that the performance of *lu.A* (Ind.(lu)) and *is.A* (Ind.(is)) running on independent VM is improved by using HS and SVS approaches. And it confirms the effectiveness of co-scheduling methods (HS and SVS) for tightly-coupled parallel applications running on a single SMP VM. However, the performance of HS is not as good as that of SVS when *lu.A* and *is.A* run on virtual clusters (*VC1*, *VC2*, *VC5*, *VC7*, *VC8*, and *VC9*). The reason is that HS approach does not consider the synchronization requirements of related VMs that belong to the same virtual cluster when it co-schedules VCPUs of a VM.

For *ep*, these four approaches (SVS, HS, BS, and CREDIT) have the similar performance, because SVS, HS, and BS adopt similar strategy as CREDIT when they schedule VMs that host CPU intensive applications.

From Figure 4(a), 4(b), 5(a), and 5(b) in Section 5.2.1, and Figure 6 in this section, we can also find that SVS provides more performance gains with respect to *lu* than that with respect to *is*. That is, our SVS approach is more suitable for such type of applications as *lu*. The performance difference of SVS in different kinds of tightly-coupled parallel applications lies in their different characteristics of internal communication pattern, which will be investigated in our future work.

5.3.2 Mix of Parallel and Non-parallel Applications

In this experiment, we further evaluate the performance of approaches with parallel applications and non-parallel applications. Because web applications and disk intensive applications are very common in cloud environment, we take them as the representative of non-parallel applications.

(1) Benchmark. Three types of benchmarks are used in this experiment: 1) *web server*, the average response time of which is measured by *httperf*; 2) *Bonnie++*, a benchmark suite that aims at performing a number of simple tests of hard drive and file system performance; and 3) NPB suite of version 2.4.

(2) Experiment settings. The experiment environment is built following the settings described in subsection 5.3.1. The application running on each virtual cluster is randomly selected from *lu*, *is* and *ep* of NPB benchmark. Meanwhile, the application running on each independent VM is randomly selected from the set of *web server*, *Bonnie++*, *lu*, *is* and *ep*. For *web server*, we use *httperf* with default configuration to measure its average response time. For *Bonnie++*, the default configuration is used, and the performance metrics are Bonnie Sequential Output (BSO) (— Per Chr), Bonnie Sequential Output (BSO) (— Block), and Bonnie Sequential Input (BSI) (— Per Chr).

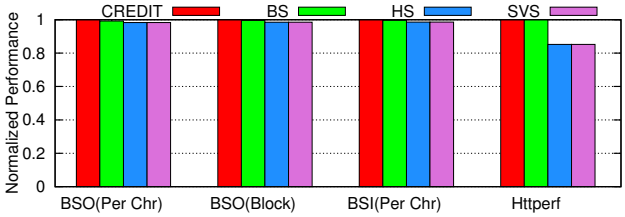


Fig. 7. The performance of scheduling approaches on web and disk intensive applications, respectively.

(3) Experimental results. The average throughput of *Bonnie++* and the average response time of *web server* with BS, HS and SVS approaches are normalized to that of them with CREDIT, respectively. The experimental results are shown in Figure 7.

We can see that the performances (SO (— Per Chr), SO (— Block), and SI (— Per Chr)) of *Bonnie++* with SVS, HS,

and BS approximate to those with CREDIT. The average response time of web server with SVS and HS is about 84% of those with CREDIT.

The reason behind the experimental results is that the average response time of non-parallel application with HS and SVS approaches becomes longer due to the scheduling priority declining, which results in the performance degradation of delay-sensitive application (such as web application) and has minimal impact on delay-insensitive application (such as *Bonnie*++). The findings about the NPB benchmarks running on virtual clusters with four approaches in this experiment are consistent with the conclusions of the experiment in Section 5.3.1. Therefore, the experiment results are omitted here.

5.4 Evaluation of CPU Fairness

The fairness of system is guaranteed by the modules of CPU time allocation and consumption in our SVS scheduler. In order to measure the fairness for CPU time allocation among VMs hosted in virtualized system, we adopt Jain's fairness index [26] which is a well-known indicator for this purpose.

$$\mathcal{J}(x_1, x_2, ..., x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} x_i^2} \quad (3)$$

Fairness index can be computed by Formula (3), where there are n VMs and x_i is the CPU time obtained by the ith VM. Its values are always in [0,1], and the higher its value, the higher fairness degree of the VM resource allocation is. When all VMs receive the same allocation, the fairness index will be equal to 1.

The Credit scheduler of Xen is a well-known fair scheduler. Therefore, we treat it as the baseline in our experiment. As described in Section 5.3.1, different applications may not finish at the same time when they simultaneously start to run on corresponding virtual clusters and independent VMs. Therefore, we use a batch program to run each application repeatedly. And the number of repetitions is set large enough that all applications are running during the experiment about CPU fairness.

TABLE 4
Results of CPU fairness with Jain's fairness index

Metrics	Appraoches	
	CREDIT	SVS
Jain's fairness index	0.999949737	0.999212079
Standard Deviation	0.0000220134	0.000351322

We first record the CPU time obtained by each VM every ten minutes on a randomly selected node using the same experiment as Section 5.3.2 during one day (24 hours). That is, we compute the CPU fairness among VMs every ten minutes, and we obtain 144 Jain's fairness indexes in total ((24 hours * 60 minutes per hour)/10 minutes). At last, we get the mean and standard deviation about these 144 Jain's fairness indexes, which are shown in Table 4. And we can observe that SVS scheduler has almost the same fairness as that of Credit scheduler.

6 Discussion and Future Work

In this section, we discuss the following questions and our future work.

(1) Can SVS be more precise, thus SVS can get better performance?

SVS schedules virtual cluster running tightly-coupled parallel applications by locally visible synchronization requirement information (e.g., the statistics of received packets in VM), rather than in an absolutely precise way (i.e., strict co-scheduling). The reason behind this design is that we want to make our method simple and practical, because strict co-scheduling through centralized controller is not cost-effective and introduces too much overhead. Though SVS has proved its prominent performance, we will further explore and characterize the relation model between local behavior of VMs (e.g., the average waiting time of spinlock, Inter-Processor Interrupt communications) and synchronization requirements among them, so as to make SVS adaptive to more complex scheduling scenario.

(2) How to choose a cloud infrastructure to run parallel applications, overcommitted or non-overcommitted?

Amazon EC2 has launched Cluster Compute Instances (CCI) [27] hosted on dedicated infrastructure to support parallel applications, where one physical node only hosts one CCI. Actually, there are always tradeoffs between performance and cost and thus choices between overcommitted and non-overcommitted systems for running parallel applications. Many practical factors, such as pricing policy, the type and size of parallel applications, VCPU-to-PCPU ratio, and customer satisfaction [28], can significantly influence users' choices. In order to provide users with more options, we focus on improving the deliverable performance of parallel applications in overcommitted cloud environment in this paper. And we will build a performance-cost tradeoff model to help users choose a suitable environment (overcommitted or non-overcommitted) for running different kinds of parallel applications.

7 Related Work

In this section, we provide an overview of the research work related to our study.

Since the Xen virtualization technology was developed, many performance studies [7, 29, 30] have been conducted to investigate the feasibility of using virtualized platforms to run parallel applications. Virtualization may cause problems which do not exist in traditional distributed systems. For instance, with dynamic [31, 32] or implicit co-scheduling [33–35], independent schedulers on each workstation coordinate parallel jobs through local events (e.g., spinlock) that occur naturally within the communicating applications in traditional distributed systems. However, a spinlock held by a VM

can be preempted due to VCPU preemption [29] in the virtualized environment, which vastly increases synchronization latency and potentially blocks the progress of other VCPUs waiting to acquire the same lock. Therefore, the solutions in traditional distributed environments do not perform very effectively for the scheduling of virtual clusters in overcommitted cloud environment.

The negative influence of virtualization on synchronization in parallel workloads is discussed in [5], and a hybrid scheduling framework is introduced to deal with the performance degradation of parallel workload running on a SMP VM. In [6] an adaptive dynamic co-scheduling approach is proposed to mitigate the performance degradation of parallel workload on a SMP VM. The co-scheduling solution of VMware [3] tries to maintain synchronous progress of VCPU siblings by deferring the advanced VCPUs until the slower ones catch up. A probabilistic type of co-scheduling named balance scheduling is provided in [4], the concept of which is to balance VCPU siblings on different CPUs without considering the precise scheduling of VCPUs at the same time. The authors of [36] present Flex, a VCPU scheduling scheme, to enforce fairness at VM-level, and to flexibly schedule VCPUs to minimize wasted busy-waiting time, so as to improve the efficiency of parallel applications running on SMP VM. A demand-based coordinated scheduling scheme for consolidated VMs that host multithreaded workloads is presented in [17]. Actually, these co-scheduling solutions are all targeted for SMP VMs rather than virtual clusters.

In [37] authors improve the performance of virtual MapReduce cluster through batching of I/O requests within a group and eliminating superfluous context switches. A communication-aware CPU scheduling algorithm is presented in [38] to alleviate the problem of CPU schedulers being agnostic to the communication behavior of modern, multi-tier, Internet server applications in virtualization-based hosting platforms. The authors of [39] propose VMbuddies, which presents a high-level synchronization protocol to guarantee application's SLA when the correlated multiple VMs are migrated across geographically distributed data centers concurrently. In [40] authors present Nephele, which is a High-Throughput Computing framework to explicitly exploit the dynamic resource allocation offered by today's IaaS clouds for both, task scheduling and execution. Based on this framework, they perform evaluations of MapReduce-inspired processing jobs on an IaaS cloud system and compare the results to the popular data processing framework Hadoop. In [41] authors explore to deploy a computing cluster on the top of a multicloud infrastructure, for solving loosely coupled Many-Task Computing (MTC) applications. In this way, the cluster nodes can be provisioned with resources from different clouds to improve the cost effectiveness of the deployment, or to implement high-availability strategies. Our work differs from these work [37–41], because we focus on improving the tightly-coupled parallel applications that run on virtual clusters in overcommitted cloud.

8 Conclusions

More and more users from academic and commercial communities are exploring cloud computing system as an alternative to local clusters to execute their tightly-coupled parallel applications. However, they still face the performance degradation problem, which results from that the synchronization requirement of VMs among the same virtual cluster running tightly-coupled parallel application is neglected by VMMs.

This paper targets the challenge of how to schedule virtual clusters hosting tightly-coupled parallel applications and mitigate performance degradation in overcommitted cloud environment. For this purpose, we analyze the inadequacy of existing solutions, and introduce a synchronization-aware VM scheduling (SVS) approach of virtual clusters. With our SVS approach, no extra communication cost is introduced in schedulers because of our synchronization-aware design. This approach is simple to apply in practice. Meanwhile, it allows participating VMMs to act autonomously, thus retaining the independence of VMMs. We implement a SVS scheduler with SVS algorithm, and compare it to existing methods such as CREDIT, BS and HS in a series of experiments. For tightly-coupled parallel application, our SVS approach improves the application performance significantly in comparison to the state-of-the-art approaches. For example, in absolute terms, the execution times of *lu* under HS and BS approaches are longer than that under our SVS approach by 162% and 188% when it runs on virtual cluster (*VC9*). Our SVS approach also maintains a high fairness among VMs. The Jain's fairness index of our approach is greater than 0.9992, which is very close to that of Credit scheduler (0.9999).

Acknowledgment

We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported by National Science Foundation of China under grant No.61232008 and No.61472151, National 863 Hi-Tech Research and Development Program under grant No.2013AA01A213, Chinese Universities Scientific Fund under grant No.2013TS094, Research Fund for the Doctoral Program of MOE under grant No.20110142130005. This work was also supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

[1] Thomas J Hacker and Kanak Mahadik. Flexible resource allocation for reliable virtual cluster computing systems. In *Proc. SC*, page 48. ACM, 2011.

[2] Thomas J Hacker and Kanak Mahadik. *Magellan Final Report*. U.S. Department of Energy (DOE), 2011.

[3] Vmware, inc. vmware vsphere 4: The cpu scheduler in vmware esx 4.1, http://www.vmware.com/files/pdf/techpaper/VMW_vSphere41_cpu_schedule_ESX.pdf.

[4] Orathai Sukwong and Hyong S Kim. Is co-scheduling too expensive for smp vms? In *Proc. EuroSys*, pages 257–272. ACM, 2011.

[5] Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu. The hybrid scheduling framework for virtual machine systems. In *Proc. VEE*, pages 111–120. ACM, 2009.

[6] Chuliang Weng, Qian Liu, Lei Yu, and Minglu Li. Dynamic adaptive scheduling for virtual machines. In *Proc. HPDC*, pages 239–250, 2011.

[7] Constantinos Evangelinos and Chris N. Hill. Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's ec2. In *In The 1st Workshop on Cloud Computing and its Applications (CCA)*, 2008.

[8] Vijayaraghavan Soundararajan and Jennifer M Anderson. The impact of management operations on the virtualized datacenter. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 326–337. ACM, 2010.

[9] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. SoCC*, page 7. ACM, 2012.

[10] Sheng Di, Yves Robert, Frédéric Vivien, Derrick Kondo, Cho-Li Wang, Franck Cappello, et al. Optimization of cloud task processing with checkpoint-restart mechanism. In *Proc. SC*, 2013.

[11] Credit scheduler. a proportional fair share cpu scheduler, http://wiki.xen.org/wiki/Credit_Scheduler/.

[12] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, 2004.

[13] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule processes, not vcpus. In *Proc. APSys*, page 1. ACM, 2013.

[14] Joseph Lee Rodgers and W Alan Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, 1988.

[15] Nas parallel benchmarks, http://www.nas.nasa.gov/Resources/Software/npb.html.

[16] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware virtual machine scheduling for i/o performance. In *Proc. VEE*, pages 101–110. ACM, 2009.

[17] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Demand-based coordinated scheduling for smp vms. In *Proc. ASPLOS*, pages 369–380. ACM, 2013.

[18] Vmware esx, http://www.vmware.com/products/esx/.

[19] Huacai Chen, Hai Jin, Kan Hu, and Jian Huang. Scheduling overcommitted vm: Behavior monitoring and dynamic switching-frequency scaling. *Future Generation Computer Systems*, 29(1):341–351, 2013.

[20] Eucalyptus, https://www.eucalyptus.com/.

[21] Cloudstack, http://cloudstack.apache.org/.

[22] Parallel workloads archive, http://www.cs.huji.ac.il/labs/parallel/workload/.

[23] Bonnie++. a benchmark suite aimed at performing hard drive and file system performance, http://www.coker.com.au/bonnie++/.

[24] Httperf. a tool for measuring web server performance, http://www.hpl.hp.com/research/linux/httperf/.

[25] Asser N. Tantawi. A scalable algorithm for placement of virtual clusters in large data centers. In *Proc. MASCOTS*, pages 3–10. IEEE, 2012.

[26] Raj Jain, Arjan Durresi, and Gojko Babic. Throughput fairness index: An explanation, feb. 1999. In *ATM Forum/99-0045*.

[27] High performance computing (hpc) on aws, http://aws.amazon.com/hpc-applications/.

[28] Junliang Chen, Chen Wang, Bing Bing Zhou, Lei Sun, Young Choon Lee, and Albert Y Zomaya. Tradeoffs between profit and customer satisfaction for service provisioning in the cloud. In *Proc. HPDC*, pages 229–238. ACM, 2011.

[29] Adit Ranadive, Mukil Kesavan, Ada Gavrilovska, and Karsten Schwan. Performance implications of virtualizing multicore cluster machines. In *Proceedings of the 2nd workshop on System-level virtualization for high performance computing*, pages 1–8. ACM, 2008.

[30] Sayaka Akioka and Yoichi Muraoka. Hpc benchmarks on amazon ec2. In *Proc. WAINA*, pages 1029–1034. IEEE, 2010.

[31] Patrick G Sobalvarro and William E Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Job Scheduling Strategies for Parallel Processing*, pages 106–126. Springer, 1995.

[32] Patrick G Sobalvarro, Scott Pakin, William E Weihl, and Andrew A Chien. Dynamic coscheduling on workstation clusters. In *Job Scheduling Strategies for Parallel Processing*, pages 231–256. Springer, 1998.

[33] Andrea C Dusseau, Remzi H Arpaci, and David E Culler. Effective distributed scheduling of parallel workloads. In *SIGMETRICS*, volume 24, pages 25–36. ACM, 1996.

[34] Andrea C Arpaci-Dusseau, David E Culler, and Alan M Mainwaring. Scheduling with implicit information in distributed systems. In *SIGMETRICS*, volume 26, pages 233–243. ACM, 1998.

[35] Andrea Carol Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 19(3):283–331, 2001.

[36] Jia Rao and Xiaobo Zhou. Towards fair and efficient smp virtual machine scheduling. In *Proc. PPoPP*, pages 273–286. ACM, 2014.

[37] Hui Kang, Yao Chen, Jennifer L Wong, Radu Sion, and Jason Wu. Enhancement of xen's scheduler for mapreduce workloads. In *Proc. HPDC*, pages 251–262. ACM, 2011.

[38] Sriram Govindan, Jeonghwan Choi, Arjun R Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Xen and co.: communication-aware cpu management in consolidated xen-based hosting platforms. *IEEE Transactions on Computers*, 58(8):1111–1125, 2009.

[39] Haikun Liu and Bingsheng He. Vmbuddies: Coordinating live migration of multi-tier applications in cloud environments. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2014.

[40] Daniel Warneke and Odej Kao. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):985–997, 2011.

[41] Rafael Moreno-Vozmediano, Ruben S Montero, and Ignacio M Llorente. Multicloud deployment of computing clusters for loosely coupled mtc applications. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):924–930, 2011.

Song Wu is a professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. from HUST in 2003. He is now served as the director of Parallel and Distributed Computing Institute at HUST. He is also served as the vice director of Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) of HUST. His current research interests include cloud computing, system virtualization, datacenter management, storage system, in-memory computing and so on.

Haibao Chen is currently working toward the PhD degree in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL) at Huazhong University of Science and Technology (HUST) in China. His research interests include parallel and distributed computing, virtualization, and resource scheduling on cloud computing.

Sheng Di received the masters degree (MPhil) from Huazhong University of Science and Technology in 2007 and the PhD degree from The University of Hong Kong in 2011. He is currently a post-doctor researcher at Argonne National Laboratory. His research interest includes optimization of distributed resource allocation and fault tolerance for Cloud Computing and High Performance Computing. His background is mainly on the fundamental theoretical analysis and system implementation. He is a member of the IEEE.

Bing Bing Zhou received the BS degree from Nanjing Institute of Technology, China and the PhD degree in Computer Science from Australian National University, Australia. He is currently an associate professor at the University of Sydney. His research interests include parallel/distributed computing, Grid and cloud computing, peer-to-peer systems, parallel algorithms, and bioinformatics. His research has been funded by the Australian Research Council through several Discovery Project grants.

Zhenjiang Xie is currently working toward the Master degree in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL) at Huazhong University of Science and Technology (HUST) in China. His research interests include virtualization and resource scheduling on cloud computing.

Hai Jin received the PhD degree in computer engineering from HUST in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He is currently the dean of the School of Computer Science and Technology at HUST. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of National 973 Basic Research Program Project of Cloud Security. He is a senior member of the IEEE and a member of the ACM. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.

Xuanhua Shi is a professor in Service Computing Technology and System Lab and Cluster and Grid Computing Lab, Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. degree in Computer Engineering from HUST in 2005. From 2006, he worked as an INRIA Post-Doc in PARIS team at Rennes for one year. His current research interests focus on the scalability, resilience and autonomy of large-scale distributed systems, such as peta-scale systems, and data centers.