Poris: A Scheduler for Parallel Soft Real-Time Applications in Virtualized Environments

Song Wu, Member, IEEE, Like Zhou, Huahua Sun, Hai Jin, Senior Member, IEEE, Xuanhua Shi, Member, IEEE

Abstract—With the prevalence of cloud computing and virtualization, more and more cloud services including parallel soft real-time applications (PSRT applications) are running in virtualized data centers. However, current hypervisors do not provide adequate support for them because of soft real-time constraints and synchronization problems, which result in frequent deadline misses and serious performance degradation. CPU schedulers in underlying hypervisors are central to these issues. In this paper, we identify and analyze CPU scheduling problems in hypervisors. Then, we design and implement a parallel soft real-time scheduler according to the analysis, named Poris, based on Xen. It addresses both soft real-time constraints and synchronization problems simultaneously. In our proposed method, priority promotion and dynamic time slice mechanisms are introduced to determine when to schedule virtual CPUs (VCPUs) according to the characteristics of soft real-time applications. Besides, considering that PSRT applications may run in a virtual machine (VM) or multiple VMs, we present parallel scheduling, group scheduling and communication-driven group scheduling to accelerate synchronizations of these applications and make sure that tasks are finished before their deadlines under different scenarios. Our evaluation shows Poris can significantly improve the performance of PSRT applications no matter how they run in a VM or multiple VMs. For example, compared to the Credit scheduler, Poris decreases the response time of web search benchmark by up to 91.6%.

Index Terms—virtualization; soft real-time; parallel; scheduling

1 INTRODUCTION

Soft real-time applications are very common in real-world, which are allowed to miss a few deadlines as long as their performance is acceptable, such as media player. As we have entered the multicore era, many soft real-time applications use parallel programming models to fully utilize multicore processors, which can reduce deadline misses and possibly shorten response time. We call this kind of applications as parallel soft real-time ones, abbreviated as PSRT applications. Examples include cloud-based video streaming, real-time transcoding, and real-time stream computing.

Meanwhile, an increasing number of applications are running in clouds because of their flexibility and cost-effectiveness. Virtualized cloud data centers, such as Amazon’s Elastic Compute Cloud (EC2) [1], use VMs to host different applications from customers on the same hardware platform. However, when running in virtualized environments, PSRT applications do not behave well and only obtain inadequate performance [2–4]. This problem is mainly due to the fact that CPU schedulers in underlying hypervisors are unaware of the characteristics of PSRT applications running in VMs.

PSRT applications should finish their tasks before deadlines due to their soft real-time constraints. A PSRT application can begin to execute only when the VCPU of its hosting VM is scheduled. However, CPU schedulers in underlying hypervisors do not consider the soft real-time constraints of PSRT applications, which cause frequent deadline misses. For example, Xen’s default Credit scheduler [5] is a proportional fair share CPU scheduler, and does not support the VM hosting PSRT applications which needs to be scheduled before their deadlines.

More importantly, PSRT applications may miss deadlines even if CPU schedulers adopt soft real-time scheduling strategies optimized for single-threaded soft real-time applications. The main reason is that these strategies are asynchronous CPU scheduling without considering the parallel feature of PSRT applications, which is also used by hypervisors such as Xen [6]. When PSRT applications need to synchronize between threads or processes, they may often wait for inactive VCPUs or VMs for synchronization because VCPUs of a VM or different VMs are not active at the same time under these strategies. Besides, when a PSRT application runs in a VM with multiple VCPUs, inconsistent active states of VCPUs causes synchronization problems in a VM, such as lock-holder preemption (LHP) problem [7].

Moreover, synchronization problems among VMs, which are much more complicated than the problems in a VM, may appear when PSRT applications run in multiple VMs. The complexity lies in two aspects. On one hand, if these VMs are hosted by a physical machine (PM), the total number of VCPUs of these co-located VMs is probably greater than the number of PCPUs [8]. As a result, it is impossible to schedule all these VCPUs simultaneously to make their active states consistent.
On the other hand, if these VMs are distributed across multiple PMs, it is a challenge to schedule these VMs synchronously across PMs. Both synchronization problems in a VM and among VMs result in a waste of CPU time on waiting for synchronization and doing useless work, such as busy-waiting on preempted spinlocks, and cause deadline misses of PSRT applications.

Despite the existing research [2][7][9–13] on CPU scheduling in virtualized environments, they only focus on how to support single-threaded soft real-time applications or multi-threaded applications running in a VM. For instance, Lee et al. [2] introduce laxity to denote the deadline when a VM should be scheduled, but VCPUs are scheduled asynchronously. Co-scheduling [12][13] and balance scheduling [11] schedule VCPUs in a round-robin way, but they do not consider soft real-time constraints of PSRT applications. Moreover, all of them are not aware of VMs that run the same PSRT application.

In summary, PSRT applications are more complicated than single-threaded soft real-time applications, and have to face soft real-time constraints and synchronization problems in virtualized environments. In this paper, we first analyze the causes of performance degradation of these applications, and argue that both soft real-time constraints and synchronization problems should be considered simultaneously to support these applications. Then, we design and implement a parallel soft real-time scheduler according to the analysis, named Poris, based on Xen. Priority promotion and dynamic time slice mechanisms are proposed to determine when to schedule VCPUs. Furthermore, we consider the situations that PSRT applications run in a VM or multiple VMs, and propose parallel scheduling, group scheduling and communication-driven group scheduling to decide how to schedule VCPUs, which address synchronization problems.

The main contributions of this paper are as follows.

- We identify the scheduling problems in virtual machine monitor (VMM) schedulers when PSRT applications run in a VM or multiple VMs, and analyze the problems.
- We present different solutions to determine when to schedule VCPUs according to the characteristics of PSRT applications by dividing them into event-driven ones and time-driven ones. Then, we design parallel scheduling, which schedules all the VCPUs of a RT-VM simultaneously, to address synchronization problems in a VM.
- We present group scheduling to address synchronization problems among VMs when the VMs running the same PSRT applications are co-located on a PM, and propose communication-driven group scheduling to support PSRT applications if the hosted VMs are distributed across multiple PMs.
- We implement a prototype in the Xen hypervisor based on the algorithm, named Poris, and verify its effectiveness through various applications. The experimental results show that Poris can guarantee the performance of PSRT applications no matter how they run in a VM or multiple VMs.

2 BACKGROUND

Xen is a widely used open-source hypervisor. Its default Credit scheduler is a proportional fair share scheduler. Each VM is assigned a parameter called weight, and CPU resources (or credits) are distributed to VCPUs of VMs in proportion to their weight at the end of each accounting period (default 30ms).

There are three kinds of VCPU priorities in the Credit scheduler: boost, under, and over. The VCPU priority is recalculated based on the credits at the end of the accounting period. When a VCPU exhausts its credits, its priority is set to over. If it still has credits, its priority is set to under. Boost priority is introduced to reduce the latency of I/O processing. If a blocked VCPU receives external events, its priority is promoted to boost. Then, the scheduler is invoked, and the VCPU will preempt the current running VCPU (unless its priority is boost).

VCPUs are always inserted into the tail of PCPUs’ queue, and the scheduler picks up the VCPU from the head of queue to execute. VCPUs with the same priority are scheduled in round-robin manner. Besides, the Credit scheduler supports symmetric multiprocessing (SMP) platforms well. When a PCPU becomes idle or its run queue has no VCPU with boost or under priority, it checks its peer PCPUs’ run queues to see if there is a higher-priority VCPU. If so, it steals that VCPU.

3 PROBLEM ANALYSIS

In this section, we first use scenarios to show the behaviors of different scheduling strategies in face of PSRT applications. Then, we analyze the scheduling problems resulting in performance degradation.

3.1 Scenarios

A basic task model of PSRT applications is shown in Fig.1. It consists of \( n+1 \) threads (or processes) \( t_0 \sim t_n \). \( t_0 \) listens user requests and forwards the requests to the other \( n \) threads \( t_1 \sim t_n \), which process the requests in parallel. When these threads finish their tasks and return results to \( t_0 \), it responds user requests.

In this section, we discuss two scenarios: single-VM scenario and multi-VMs scenario. The difference between these scenarios lies in the number of VMs used to host a PSRT application. In these scenarios, a PM has a quad-core processor, and each VM has two VCPUs \( v_{ij} \) represents the \( j \)th VCPU of the \( i \)th VM. In the single-VM scenario, the PSRT application, which consists of
three threads \((t_0 \sim t_2)\), is running in a VM. \(t_0\) and \(t_1\) run on \(v_{00}\), \(t_2\) runs on \(v_{01}\). Because the VM needs at most two PCPUs, for the convenience of description, only two PCPUs are shown in this scenario. In the multi-VMs scenario, for simplicity, the PSRT application, which has two more threads \((t_3\) and \(t_4)\), is running in two VMs co-located on a PM. \(t_3\) and \(t_4\) run on \(v_{10}\) and \(v_{11}\) respectively. Assume that no load balance is happened in guest OSes before the task is finished. If all VCPUs are online at the same time, the PSRT application can respond user requests in a time slice.

In the following, we describe the scheduling sequences of VCPUs when the hypervisor uses different scheduling strategies.

**Soft Real-Time Scheduling** In the soft real-time scheduling, the VM running the PSRT application is scheduled immediately when requests arrive, but its VCPUs are scheduled in an asynchronous way. The single-VM scenario is shown in Fig.2(a). Asynchronous scheduling may cause VCPUs of a VM to be located in the same PCPU’s run queue. \(v_{00}\) is scheduled immediately when requests arrive, and may be descheduled when the thread on \(v_{00}\) is in a critical section. Then, \(v_{01}\) is scheduled, but the thread on \(v_{01}\) cannot enter the critical section. It may try to enter the critical section in a busy-waiting way if the critical section is protected by spinlocks. \(t_2\) on \(v_{01}\) can only process the requests after thread on \(v_{00}\) releases locks. Then \(t_0\) can respond the requests, but it misses the deadline. The multi-VMs scenario is similar to the single-VM scenario, which is shown in Fig.3(a). Even if VCPUs of a VM is not on the same PCCPU’s run queue, a VCPU may also waste time on entering critical sections. \(t_0\) can respond the requests when \(t_3\) and \(t_4\) have finished their work and sent the results to \(v_{00}\). However, it misses the deadline too.

**Co-scheduling** Co-scheduling schedules VCPUs synchronously, but in a round-robin manner. The single-VM scenario is shown in Fig.2(b). Because co-scheduling can address the LHP problem, critical sections are not shown in the figure. Co-scheduling schedules \(v_{00}\) and \(v_{01}\) at the same time, which can guarantee that \(t_0 \sim t_2\) can synchronize and complete their tasks in a short time. However, co-scheduling does not consider the soft real-time constraints of PSRT applications. Then, \(v_{00}\) and \(v_{01}\) may be scheduled too late, which causes deadline misses. The multi-VMs scenario is shown in Fig.3(b). \(v_{01}\) and \(v_{01}\) may be in the same run queue, and \(v_{10}\) and \(v_{11}\) can only be scheduled after \(v_{01}\). After \(t_3\) and \(t_4\) have finished their work, \(v_{00}\) is scheduled again. Then, \(t_0\) can respond the requests, but misses its deadline. Therefore, when these applications run in multiple VMs, VCPUs belonging to different VMs are scheduled in an asynchronous way under co-scheduling, which may increase the task finish time, and cause deadline misses.

**Parallel Soft Real-Time Scheduling** The last scheduling strategy is parallel soft real-time scheduling that all the VCPUs can be scheduled synchronously and immediately when the requests arrive. If PSRT applications run in multiple VMs, all the VCPUs of these VMs are scheduled synchronously. The single-VM scenario and multi-VMs scenario are shown in Fig.2(c) and Fig.3(c). Because the parallel soft real-time scheduling makes all the VCPUs online at the same time, threads can synchronize and process the requests in a short time. The response time of the request is very short and does not miss deadline. Besides, because all the VCPUs are online at the same time, the parallel soft real-time scheduling supports other task models too.

### 3.2 Analysis

In this subsection, we analyze the scheduling problem, and find the factors that affect the performance of PSRT.
applications, which help us to design a CPU scheduler to support these applications.

The PCPs of the PM are denoted by \( P = \{ P_1, P_2, ..., P_p \} \), and the number of PCPs is \( |P| \). The VMs running a PSRT application are denoted by \( V = \{ V_1, V_2, ..., V_v \} \), and \( |V| \) denotes the number of VMs where the PSRT application is deployed. The weight of \( V_i \) assigned in the proportional fair share scheduler is denoted by \( \omega(V_i) \). The set of VCPUs of the \( i \)-th VM is denoted by \( C(V_i) = \{ v_{i1}, v_{i2}, ..., v_{i|C(V_i)|} \} \), and \( |C(V_i)| \) denotes the number of VCPUs of \( V_i \). We also define some variables as follows, which are illustrated in Fig. 4:

- \( T_p \): scheduling period, denoting the accounting period, such as 30ms in the Credit scheduler.
- \( T_e(v_{ij}) \): execution time, denoting the time taken by the \( j \)-th VCPU of the \( i \)-th VM to do useful work, typically equals to the length of time slice. For simplicity, we assume that each VCPU of a VM executes for the same time period. Thus, we use \( T_e(V_i) \) instead of \( T_e(v_{ij}) \) in the following analysis.
- \( T_{ws}(v_{ij}) \): wasting time, denoting the time taken by the \( j \)-th VCPU of the \( i \)-th VM doing useless work. It is caused by the asynchronous scheduling algorithm which makes the VCPU spend this time period on synchronous operations.
- \( T_a(V_i) \): wait time, denoting the time period from the arrival of a request to the start of the first VCPU slice of the \( i \)-th VM.
- \( T_i(V_i) \): scheduling time lag in the \( i \)-th VM, denoting the lag on scheduling time between the first scheduled VCPU and the last scheduled VCPU in a round (a round means each VCPU is scheduled once).
- \( T_{ib}(V_i, V_j) \): scheduling time lag between the \( i \)-th VM and the \( j \)-th VM, denoting the lag on scheduling time between the end of the last scheduled VCPU of the \( i \)-th VM and the \( j \)-th VM in a round.
- \( T_s(V_i) \): next scheduled time of a VCPU of the \( i \)-th VM, which hosts the thread to summarize results.
- \( T_c \): gathering time, denoting the time taken by a thread to summarize results sent by other threads and respond to requests.

Fig. 4. The definition of some variables used in analysis

Normally, a hypervisor’s scheduler, such as the Credit Scheduler, schedules VCPUs asynchronously without considering the soft real-time constraints of PSRT applications. Therefore, the response time of these applications is

\[
\max_{1 \leq i,j \leq |V|, i \neq j} \{ T_{ws}(V_i) + T_i(V_i) + T_e(V_i) + T_{ib}(V_i, V_j) + T_n(V_i) + T_s \}. 
\]

The soft real-time scheduling reduces \( T_{ws}(V_i) \), which can improve the performance of these applications. However, the improvement is limited, because these applications still spend lots of time on synchronization. Similarly, because the co-scheduling does not consider the synchronization between the VMs, it only eliminates the time spending on synchronization and scheduling time lag in a VM caused by asynchronous scheduling. The response time of these applications changes into

\[
\max_{1 \leq i,j \leq |V|, i \neq j} \{ T_{ws}(V_i) + T_i(V_i) + T_{ib}(V_i, V_j) + T_n(V_i) + T_r \},
\]

but it is still long. The parallel soft real-time scheduling reduces \( T_{ws}(V_i) \) and eliminates \( T_i(V_i) \). Besides, because it considers the synchronization among VMs, \( T_{ib}(V_i, V_j) \) can be eliminated if \( \sum_{i=1}^{|V|} |C(V_i)| \leq |P| \). Otherwise, \( T_{ib}(V_i, V_j) \) can be reduced to some extent. Therefore, the parallel soft real-time scheduling can improve the performance of PSRT applications significantly.

Moreover, asynchronous scheduling results in a waste of CPU time doing useless work. The relation of \( \omega(V_i) \), \( T_p \), \( T_e(v_{ij}) \), and \( T_{ws}(v_{ij}) \) can be given by this equation:

\[
\omega(V_i) = \sum_{j=1}^{|C(V_i)|} \left( T_e(v_{ij}) + T_{ws}(v_{ij}) \right) / T_p \tag{1}
\]

From (1), we can see that the parallel soft real-time scheduling can increase the actual useful execution time of VCPUs because they eliminate or reduce \( T_{ws}(v_{ij}) \).

**4 Design**

We choose the Xen hypervisor to design Poris because it is a widely used open-source hypervisor. In this section, we describe the system overview of Poris first. Then, we describe main components of Poris in detail. Finally, we propose the parallel soft real-time scheduling algorithm, which consists of all the components.

**4.1 Overview**

Poris is a parallel soft real-time scheduler, which addresses soft real-time constraints and synchronization problems simultaneously. The scheduling problem can be divided into two sub-problems: 1) when to start scheduling RT-VCPUs? (For simplicity, we call the VM hosting soft real-time applications as RT-VM, and the VCPU of RT-VM as RT-VCPU.) 2) how to schedule RT-VCPUs of VMs that host the same PSRT application? The
overview of Poris is shown in Fig.5. The left part including \textit{priority promotion} and \textit{dynamic time slice} addresses the first sub-problem, and the right part including \textit{parallel scheduling}, \textit{group scheduling} and \textit{communication-driven group scheduling} solves the second one.

Soft real-time applications can be divided into event-driven ones and time-driven ones according to their characteristics. Event-driven soft real-time applications are executed when external events arrive. Time-driven ones are executed periodically. Some soft real-time applications may have both characteristics. The starting time of scheduling these applications is different, because they have different characteristics. We present \textit{priority promotion} and \textit{dynamic time slice} mechanisms to satisfy the requirements of both types of soft real-time applications. The \textit{priority promotion} mechanism promotes the priorities of RT-VCPUs and preempts the current running VCPU when these VCPUs receives I/O events, which can guarantee the performance of event-driven soft real-time applications. The \textit{dynamic time slice} mechanism adjusts time slice used by the CPU scheduler dynamically. If the system has RT-VMs, the CPU scheduler uses short time slice to schedule time-driven soft real-time applications periodically before their deadline. Otherwise, the scheduler uses long time slice to reduce overheads introduced by short time slice.

When a PSRT application runs in a RT-VM, hypervisors’ CPU schedulers may cause synchronization problems in a VM, such as LHP problem. We present \textit{parallel scheduling}, which schedules all the VCPUs of a RT-VM simultaneously, to address these problems. Accordingly, when it runs in multiple RT-VMs, synchronization problems among VMs appear, which are much more complicated. The challenges caused by synchronization problems among VMs lie in two aspects: 1) when these RT-VMs are hosted by a PM, it is possible that the total number of VCPUs of these co-located RT-VMs is greater than the number of PCPUs [8]; 2) When these RT-VMs are distributed across multiple PMs, an effective synchronization method among hypervisors is needed. For the first challenge, we present \textit{group scheduling}, which divides a group of RT-VMs running the same PSRT application into one or more subgroups according to the number of VCPUs of these RT-VMs and the number of PCPUs. Then, it schedules all the VCPUs of a subgroup simultaneously and schedule subgroups belonging to a group in a round-robin manner, to address synchronization problems among VMs co-located on a PM. For the second challenge, we present \textit{communication-driven group scheduling} to support PSRT applications. When applications are running in different VMs, synchronization between processes always needs to be accomplished through network. As a result, \textit{communication-driven group scheduling} uses packet reception events as the implicit control messages to trigger local group scheduling.

In summary, the \textit{priority promotion} and \textit{dynamic time slice} mechanisms determine when to schedule RT-VCPUs. It triggers \textit{parallel scheduling}, \textit{group scheduling} and \textit{communication-driven group scheduling} to meet deadlines of PSRT applications. The combination of them can address soft real-time constraints and synchronization problems simultaneously, which guarantees the performance of PSRT applications.

### 4.2 Priority Promotion

The Credit scheduler provides a mechanism to achieve low I/O response latency by using the \textit{boost} priority. However, this mechanism only promotes the priorities of blocked VCPUs. If a VCPU in the run queue receives external events, the priority of the VCPU is not promoted, and then the VCPU is not scheduled immediately. As a result, this mechanism cannot satisfy the demands of event-driven soft real-time applications. Besides, if the priorities of many VCPUs are promoted to \textit{boost}, the VM hosting soft real-time applications cannot be scheduled in a timely fashion, which probably results in missing deadlines of these applications. Although soft real-time applications can tolerate few deadline misses, if there are too many deadline misses, the performance of soft real-time applications is not guaranteed. In this subsection, we introduce \textit{priority promotion} mechanism to achieve timely scheduling for RT-VCPUs.

The \textit{priority promotion} mechanism introduces \textit{real-time} priority in the CPU scheduler, which is the highest priority. If a RT-VCPU with \textit{under} priority in the run queue receives an external event, the priority of the RT-VCPU is promoted to \textit{real-time}, and the RT-VCPU is inserted into the tail of the \textit{real-time} run queue. The RT-VCPU preempts current running VCPU if the priority of RT-VCPU is higher than that of current running VCPU, and its priority degrades to \textit{under} when the RT-VCPU is descheduled. Therefore, the RT-VM can be scheduled immediately when external events arrive, which satisfies the demands of event-driven soft real-time applications. Meanwhile, for other VCPUs, \textit{boost} priority is given in the original manner of the Credit scheduler.

### 4.3 Dynamic Time Slice

Time-driven soft real-time applications are more complicated than event-driven ones, as we do not know when to schedule the RT-VMs exactly. Typically, these applications can tolerate several milliseconds latency, such as media players and streaming media applications. In this subsection, we introduce \textit{dynamic time slice} to satisfy the demands of time-driven soft real-time applications.

Because the time-driven soft real-time applications can respond to requests only when the RT-VM hosting these applications is scheduled, the scheduling latency of the RT-VM determines the response time of these applications. Short time slice can reduce scheduling latency, which decreases response time of these applications. However, reducing the time slice increases the number of context switches (and cache flushes), leading to performance degradation of CPU-intensive and memory-intensive applications.
As a result, dynamic time slice is proposed to support time-driven soft real-time applications while minimizing the impact of non-real-time applications. If there is no RT-VM in the system, long time slice (i.e. 30ms) is used to schedule VCPUs. Otherwise, the time slice of scheduler is set to the short mode. For the convenience of describing the calculation of time slice, we first define some variables as follows:

- $S$: the length of time slice the scheduler used.
- $LTS$: long time slice (i.e. 30ms in the Credit scheduler).
- $N_R$: the total number of RT-VCPUs in the system.
- $N_V$: the number of VCPUs per PCPU, which is the ratio of total number of VCPUs and total number of PCPUs.
- $L$: the expected latency of soft real-time applications, which represents the soft deadline.
- $WCSL$: worst case scheduling latency.

The Credit scheduler schedules VCPUs with the same priority in FCFS manner. If each VM has runnable tasks in it, it occupies the entire CPU slice allotted to it. Therefore, $WCSL$ can be given by this equation:

$$WCSL = (N_V - 1) \times S \quad (2)$$

Besides, in order to guarantee the performance of soft real-time applications, $L$ and $WCSL$ must meet the following relations:

$$L \geq WCSL \quad (3)$$

Therefore, derived from (2) and (3), $S$ can be calculated by (4):

$$S = \begin{cases} 
  LTS & N_R = 0 \\
  LTS & N_R > 0 \text{ and } N_V = 1 \\
  \frac{L}{(N_V - 1)} & N_R > 0 \text{ and } N_V > 1 
\end{cases} \quad (4)$$

In our proposed method, a VM can be set as a RT-VM manually by administrators, so $N_R$ and $N_V$ can be calculated easily by the scheduler. However, it is a challenge to find the value of expected latency, $L$. This is because different soft real-time applications have different expected latency, which is an implicit characteristic of these applications.

Like many other related researches [14][15], we design an experiment to determine the approximate value of the expected latency. How to find the precise value of the expected latency of various applications is beyond the scope of this paper.

Because VoIP is a typical soft real-time application, we use MyConnection Server (MCS) [16] to conduct an experiment to find approximate value of the expected latency. We choose the VoIP test of MCS to evaluate its performance when different time slices are used in the Credit scheduler. Four VMs are used to conduct this experiment. One VM runs MCS. The others run CPU-intensive applications. The physical machine has a dual-core CPU. Each VM has one VCPU. All the VCPUs of these VMs are pinned to the same PCPU, and the VCPU of Domain0 is pinned to the other PCPU. We simulate a remote client in the real world by introducing a random delay between 20ms to 40ms using tc command. Because there are many VoIP connections in the real world, 50 connections are concurrently generated by the client to simulate such environment. The experimental results are shown in Table 1. Mean Opinion Score (MOS), an important metric to evaluate the quality of VoIP, is a performance metric reported by MCS. It is expressed as a single number in the range 1 to 5, where 5 represents the highest perceived audio quality. If MOS is greater than or equal to 4, it means that the VoIP service has good quality.

From the test results, we can see that if the time slice is lower than 5ms, MOS is 4, and there is no packet discard. However, short time slice affects the performance of CPU-intensive and memory-intensive applications. Therefore, the time slice with the value of 5ms is good enough to guarantee the quality of VoIP while minimizing the impact on other applications. $N_V$ in the experiment is 4, so the value of $L$ is calculated as 15ms according to (4). Accordingly, when we implement Poris, 15ms is used as the default expected latency. If users know the expected latency of soft real-time applications, we also provide an interface to users to set the value of expected latency.

### 4.4 Parallel Scheduling

We present parallel scheduling to address synchronization problems in a VM. It schedules all the VCPUs of a RT-VM simultaneously. (As to non-real-time VMs, the scheduler uses default scheduling strategy, i.e. asynchronous scheduling, to schedule their VCPUs.) Such scheduling strategy can eliminate the synchronization problems of PSRT applications while minimizing the impact on non-real-time applications. In this subsection, we introduce the design of parallel scheduling first.

Then, we discuss the VCPU migration problem in parallel scheduling and propose an approach named affinity exchange to address this problem.

The design of parallel scheduling is as follows. First, the scheduler distributes all the VCPUs of a RT-VM across different PCPUs. Second, when a VCPU is scheduled, if it is the first scheduled VCPU of a RT-VM, the priorities of other VCPUs, belonging to the RT-VM, are promoted to real-time. The scheduler reinserts the VCPUs into the proper position of corresponding PCPUs’ run queue (mostly, it is the head of the run queue), and soft interrupts are sent to these PCPUs to trigger rescheduling.

<table>
<thead>
<tr>
<th>Time slice</th>
<th>Upstream jitter</th>
<th>Upstream packet loss</th>
<th>Packet discards</th>
<th>MOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>30ms</td>
<td>7.8ms</td>
<td>9.5%</td>
<td>3.0%</td>
<td>1.0</td>
</tr>
<tr>
<td>15ms</td>
<td>6.7ms</td>
<td>7.2%</td>
<td>0.9%</td>
<td>1.0</td>
</tr>
<tr>
<td>5ms</td>
<td>5.0ms</td>
<td>0.1%</td>
<td>0%</td>
<td>4.0</td>
</tr>
<tr>
<td>3ms</td>
<td>4.8ms</td>
<td>0%</td>
<td>0%</td>
<td>4.0</td>
</tr>
</tbody>
</table>

### TABLE 1

Performance Metrics of VoIP Tests under the Credit Scheduler with Various Time Slices
Like balance scheduling [11], we distribute all the VCPUs of a RT-VM across different PCPUs by setting their CPU affinity dynamically, which enables the mapping and unmapping of a specific VCPU to a PCPU or a range of PCPUs.

In the Credit scheduler, if the priority of the next VCPU in a PCPU’s run queue is over, the scheduler may steal a VCPU with higher priority from peer PCPUs. Because the priorities are implemented as different regions of the same run queue, the VCPU with the priority of over is still in the run queue. Besides, because the Credit scheduler supports work-conserving mode, the VCPU with the priority of over may get extra CPU time. Parallel scheduling does not allow a run queue to hold two VCPUs of the same VM. Thus, if a RT-VCPU with the priority of over is in a PCPU’s run queue, it cannot steal a RT-VCPU with higher priority, belonging to the same RT-VM, from another PCPU’s run queue. We call this problem as the VCPU migration problem, and these two VCPUs as conflicting VCPUs. We present an approach named affinity exchange, which exchanges the CPU affinity of conflicting VCPUs, to address the VCPU migration problem. By doing so, the conflicting VCPUs change their running PCPUs, and the VCPU with higher priority has a chance to run.

4.5 Group Scheduling

A PSRT application may run in multiple VMs, such as real-time stream computing applications. Hypervisors’ CPU schedulers may cause synchronization problems in a VM and among VMs in this case, but our solution described in Section 4.4 only addresses the synchronization problems in a VM. In this subsection, we introduce group scheduling to solve synchronization problems among VMs co-located on a PM.

In the group scheduling, we introduce two hierarchies: group and subgroup. The group is used to manage all the RT-VMs running the same PSRT application. Because the total number of VCPUs in a group is probably greater than the number of PCPUs, we introduce the subgroup and ensure that the number of VCPUs in a subgroup must not exceed the number of PCPUs. Then, we can schedule all the VCPUs of a subgroup simultaneously and schedule subgroups belonging to a group in a round-robin order, which can reduce or eliminate $T_{lb}(V_i, V_j)$. In the following, we describe the group scheduling in detail.

When some RT-VMs are set as a group, we need to divide the group into one or more subgroups according to number of VCPUs in the group and the number of PCPUs. We preferably put all the VCPUs of a RT-VM into a subgroup, because we can eliminate the synchronization problems in a VM by doing so. The division is a one-dimensional packing problem. We can use current existing algorithms to divide groups. And then, we distribute all the VCPUs of a subgroup across different PCPUs by setting their CPU affinity.

From the definition in Section 3.2, we can infer that synchronization costs among VMs in a group is

$$\max_{1 \leq i,j \leq |V|, i \neq j} T_{lb}(V_i, V_j) + T_n(V_i) + T_r.$$  

For simplicity, $\max_{1 \leq i,j \leq |V|, i \neq j} T_{lb}(V_i, V_j)$ is abbreviated as $C_{sync}$. Because the Credit scheduler schedules VCPUs with the same priority in FCFS manner, $C_{sync}$ can be given by this equation:

$$C_{sync} = (N_V - 1) \times S$$  

In the group scheduling, if a RT-VCPU is scheduled, the priorities of other VCPUs, belonging to each subgroup, are promoted to real-time. Then, all the VCPUs of the group are located at the head of run queues. All the VCPUs of a subgroup can be scheduled simultaneously and subgroups are scheduled one by one. Therefore, $C_{sync}$ is reduced, and can be given by (6). From the equation, we can infer that the group scheduling can eliminate $C_{sync}$, $T_n(V_i)$ and $T_r$ if the total number of VCPUs in a group is less than or equal to the number of PCPUs.

$$C_{sync} = \left(\sum_{i=1}^{|V|} \frac{|C(V_i)|}{|P|} - 1\right) \times S$$  

Besides, we design a mechanism, named multi-round, to further reduce $C_{sync}$ and eliminate $T_n(V_i)$ and $T_r$. $C_{sync}$ can be given by (7). It is only used when a group has multiple subgroups. We assume that each VCPU in a subgroup runs for a period time (i.e. $T_{round}$), and $T_{round}$ is much less than $S$. Therefore, each subgroup can run $\frac{S}{T_{round}}$ rounds, and the total execute time for each VCPU is $S$. As to other VCPUs, the time slice is still $S$.

$$C_{sync} = \left(\sum_{i=1}^{|V|} \frac{|C(V_i)|}{|P|} - 1\right) \times T_{round}$$  

Fig.6 illustrates the basic idea of group scheduling and multi-round mechanism. A PSRT application runs in a group of four VMs (VM1~4), and each VM has two VCPUs. In the Credit scheduler, because time slice is 30ms by default, $C_{sync}$ can be very high according to (5). Our group scheduling can reduce it significantly. The group is divided into two subgroups in the group scheduling, which is shown in the left side of Fig.6. We assume that the time slice is 5ms under the policy without multi-round. We can see that $C_{sync}$ is 5ms according to (6). We assume that $T_{round}$ is 1ms under the policy with multi-round. So the time slice with the length of 5ms can let the subgroups run 5 rounds. $C_{sync}$ becomes 1ms under this policy according to (7).

4.6 Communication-driven Group Scheduling

Group scheduling only solves synchronization problems among VMs co-located on a PM. When PSRT applications run on multiple VMs, these VMs may be distributed across multiple PMs. For example, real-time
stream computing systems always need many VMs to compute. As a result, group scheduling is ineffective in this situation. We present communication-driven group scheduling for virtualized environments to address synchronization problems among VMs distributed across multiple PMs.

When applications are running in a VM, inter-process communication and synchronization is relatively simple, such as shared memory, semaphores and pipes. When applications are running in different VMs, synchronization between processes always needs to be accomplished through network. In the traditional cluster environment, researchers present communication-driven co-scheduling to improve the performance of distributed systems [17], which monitors the communication behaviors between processes and co-schedules these processes. In this paper, we present communication-driven group scheduling, which is triggered by packet reception events generated by hypervisors, to support PSRT applications. The basic idea behind this strategy is to keep RT-VMs that need to be synchronized on-line at the same time.

In virtualized environments, when a process wants to synchronize with other processes in a VM located at other PMs, the time taken by this synchronous operation is a sum of network latency, VCPU scheduling latency after packets arrive and process scheduling latency after the VCPU is scheduled. As a result, RT-VMs running the same PSRT applications are typically located at a cluster because of low network latency, and what we can do in hypervisors is to optimize the VCPU scheduling latency after packets arrive. A possible way to eliminate this latency is to schedule corresponding VMs synchronously, which needs to synchronize time between PMs and predefine the VCPU scheduling sequences. However, it is a challenge to do this in distributed environments and requirements of applications are changed dynamically. An alternative way to support this scenario is to reduce this latency, which is the objective of our proposed communication-driven group scheduling. When a RT-VM receives packets sent by other RT-VMs in the same cluster, it is probably that the packets contain synchronous operations. Then, communication-driven group scheduling schedules corresponding RT-VMs in this PM immediately and simultaneously, i.e. through group scheduling. In this case, communication-driven group scheduling needs to analyze each packets received by RT-VMs to check whether packets are sent by RT-VMs in the same cluster or users. However, as described in Section 4.2, Poris promotes the priorities of RT-VCPUs when they receive I/O events, such as user requests, which also triggers group scheduling. Therefore, communication-driven group scheduling does not distinguish packets sent by other RT-VMs in the same cluster or sent by users in Poris, which eliminates overheads introduced by analyzing packets.

In summary, communication-driven group scheduling is an implicit coordinated scheduling strategy, which does not require explicit control messages. As a result, it has good scalability.

### 4.7 Put Them Together

We present parallel soft real-time scheduling algorithm, which consists of all the components described in previous sections, to schedule VCPUs. In order to simplify the description of the algorithm and scheduler implementation, we treat the scheduling of a RT-VM as a special case in group scheduling: a group has only one subgroup, and the subgroup has only one RT-VM.

When some RT-VMs are set as a group, we use a bin-packing algorithm to divide the group into one or more subgroups, and distribute the VCPUs of each subgroup across different PCPUs. Then, we use priority promotion and dynamic time slice mechanisms to determine when to trigger parallel scheduling, group scheduling, and communication-driven group scheduling to schedule RT-VCPUs.

The pseudo-code of the algorithm is shown in Algorithm 1. The scheduler first calculates time slice according to (4) (line 1~10). If the current running VCPU is running in multi-round mode, we need to reduce its round number before inserting it into run queue (line 12~14). If the next VCPU is a RT-VCPU and is not in multi-round mode, the priorities of other runnable VCPUs of the group are promoted to real-time. And the VCPUs of each subgroup are scheduled simultaneously through sending soft interrupts to corresponding PCPUs (line 16~31). As to the RT-VCPU in multi-round mode, its time slice is set to \( T_{\text{round}} \) (line 32~36). Priority promotion mechanism invokes this algorithm to realize preemption if a VCPU with real-time priority is higher than that of the current running VCPU. As a result, communication-driven group scheduling consists of three steps. First, a RT-VM receives packets sent by other RT-VMs; Second, priority promotion mechanism promotes the priority of its VCPU to real-time and invokes this algorithm; Finally, local group scheduling is activated to schedule corresponding RT-VCPUs.

### 5 Scheduler Implementation

We implement Poris based on the Credit scheduler of Xen-4.0.1. Initially, Poris treats all the VMs as non-real-time ones, and uses default expected latency to calculate time slice. We add a new command `xm sched-rt` into the Xen management tool. Through the command, the type of a VM and expected latency can be set directly by
system administrators, and a set of VMs can be set as a
group. The modifications of the scheduler are as follows.

First, we add two structures to represent groups
and subgroups: csched_group and csched_subgroup.
csched_group maintains a list of csched_subgroup, and
csched_subgroup maintains a list of domains.

Second, we add a new priority (CSCHED_PRI_TS_RT)
as the real-time priority, which is the highest priority in
Poris, to implement priority promotion mechanism. We
modify the VCPU and PCPU operating functions to
count \(N_R\) and \(N_V\), which are used to calculate time slice.
The calculation happens when a VM is set as RT-VM, the
expected latency is changed by users, or the number of
VCPUs or PCPUs changes.

Finally, when we distribute VCPUs of a subgroup
across different PCPUs by setting their CPU affinity, a
variable named used_cpus needs to be maintained by
csched_subgroup. The CPU affinity of each VCPU is
calculated according to used_cpus. Then, we modify
csched_schedule(), which selects the next VCPU from the
run queue of a PCPU, to realize group scheduling. When a
VCPU of a group is scheduled, if other VCPUs of the
group are runnable, the priorities of the runnable
VCPUs are promoted to CSCHED_PRI_TS_RT, and soft
interrupts are sent to corresponding PCPUs, triggering
rescheduling on the particular PCPUs. Then all runnable
VCPUs of a subgroup are co-scheduled. If a group con-
tains multiple subgroups, multi-round is used to schedule
RT-VCPUs. We set \(T_{round}\) as 1 millisecond in Poris.
When RT-VCPUs are descheduled, their priorities are
degraded to under. We implement affinity exchange, which
is invoked by csched_runq_steal(), to eliminate the VCPU
migration problem when conflicting VCPUs appear. As
to communication-driven group scheduling, it uses packet
reception events, which trigger priority promotion mech-
anism to promote the priority of corresponding VCPU,
as a clue to invoke local group scheduling.

In summary, Poris does not need to modify guest OSEs
and inherits all the advantages of the Credit scheduler.

Algorithm 1: Parallel Soft Real-Time Scheduling Al-

\begin{verbatim}
Input: run-queue information of the PCPU where
the scheduler resides
Output: scheduling decision
1 if \(nr\_rt\_vcpus > 0\) then
2 \(vcpus\_per\_pcpu \leftarrow \text{ceil}(nr\_vcpus/nr\_pepus)\);
3 if \(vcpus\_per\_pcpu <= 1\) then
4 \(t\_slice \leftarrow \text{long\_time\_slice} \);
5 else
6 \(t\_slice \leftarrow \text{latency}/(vcpus\_per\_pcpu - 1)\);
7 end
8 else
9 \(t\_slice \leftarrow \text{long\_time\_slice} \);
10 end
11 next \(\leftarrow \text{get\_first\_elem}(runq)\);
12 if \(current\_round > 0\) then
13 \(current\_round \leftarrow current\_round - 1\);
14 end
15 insert(current);
16 if \(is\_rt\(4\) then
17 next\_pri \leftarrow \text{real\_time} ;
18 foreach subgroup in group(next) do
19 foreach vcpu in vm(subgroup) do
20 if runnable(vcpu) then
21 remove(vcpu);
22 vcpu\_pri \leftarrow \text{real\_time} ;
23 if \(nr\_subgroups(vcpu) > 1\) then
24 vcpu\_round \leftarrow t\_slice/t\_round ;
25 end
26 insert(vcpu);
27 raise\_softirq(cpus);
28 end
29 end
30 end
31 if \(next\_round > 0\) then
32 \(next\_runtime \leftarrow t\_round \);
33 else
34 \(next\_runtime \leftarrow t\_slice \);
35 end
36 return next;
\end{verbatim}

6 PERFORMANCE EVALUATION

In this section, we evaluate our system’s performance us-
ing various workloads and try to answer these questions:
1) does Poris support multi-threaded soft real-time appli-
cations in client-side virtualization? 2) does Poris support
multi-threaded soft real-time applications in server-side
virtualization? 3) does Poris support distributed soft real-
time applications? 4) what is the impact introduced by
Poris? In the following, we first describe the experimental
environment, and then present the experimental results.

6.1 Experimental Setup

For the scheduler setup, we ensure that each VM receives
equal CPU resources by assigning the same weight to the
VMs, and default expected latency is used to calculate
time slice. In addition, the schedulers allow VMs to use
idling CPU resources beyond their given allocation (i.e.
work-conserving mode).

Our evaluations are conducted on two types of ma-
chines with different configurations, which are used to
show Poris can cover either client-side virtualization or
server-side virtualization. The first machine (called Ma-
chine I) is comprised of a dual-core 2.6GHz Intel CPU,
2GB memory, 500GB SATA disk and 100Mbps Ethernet
card. The second machine (called Machine II) consists of
two quad-core 2.4GHz Intel Xeon CPUs, 24GB memory,
1TB SCSI disk and 1Gbps Ethernet card. The configu-
rentions of Machine I and Machine II are representative
configurations of PC and server. We use Xen-4.0.1 as the
hypervisor and CentOS 5.5 distribution with the Linux-2.6.31.8 kernel as the OS. Unless otherwise specified, all the configurations of VMs running on Machine I are as follows: 2VCPU and 256MB memory. All the configurations of VMs running on Machine II are as follows: 8VCPU and 1GB memory.

We always run the testing applications under interference configurations to verify the effectiveness of Poris in the real multi-tenant cloud environment. CPU-hungry loop application is adopted as the CPU-intensive workload, and filebench [18] is used as the I/O-intensive workload. In the experiments, there are two interfering workload configurations: CPU-intensive interfering configuration and mixed interfering configuration. The first configuration means that all interfering VMs run CPU-intensive workloads. Accordingly, the latter configuration means that some interfering VMs run CPU-intensive workloads, and some run I/O-intensive workloads.

6.2 Experiments in Client-side Virtualization

In this test, as Media player is a common soft real-time application in client side, we want to show Poris is effective in client-side virtualization by the experiments with Media Player. The QoS of Media player relies on the timely display of each video frame. We choose MPlayer 1.1 [19], an open-source cross-platform media player, to conduct a preliminary experiment to show Poris is effective to guarantee the performance of Media player. Displayed Frame-Per-Second (FPS) is used to evaluate the performance of the media player. We play low resolution (640*480) and high resolution (1280*720) video files with the length of 60 seconds, which are encoded with H.264. The frame rate of them is 29.411FPS and 23.809FPS, respectively. In order to measure the frame rate of video player, we modify MPlayer to record the timestamp of every displayed frame except dropped ones. By post-processing the record, we obtain real frame rates as time progresses.

We use four VMs to conduct this test. As MPlayer is a typical application in PC, all the VMs run on Machine I. One VM runs MPlayer with multithreaded decoding enabled. This VM is set as RT-VM. The others run different workloads to compete with MPlayer. The tests are conducted under different interfering workload configurations. The results are shown in Fig.7 and Fig.8.

From these figures, it is clear that the playback time of movies under the Credit scheduler is much longer than that under Poris (i.e. 60 seconds). The reason is that MPlayer 1.1 running in a VM does not drop any frames. In other words, if it cannot decode and display a frame timely, a short pause happens on the video playback. From the test results, we find that Poris supports the smooth playback of movies with different resolution very well whenever MPlayer competes with CPU-intensive applications or I/O-intensive applications. Compared with the Credit scheduler, Poris achieves up to 135.94% and 95.31% in playing low resolution movie and high resolution movie according to the average FPS, respectively.

On one hand, as MPlayer uses two threads to decode video files in this test and the Credit scheduler uses asynchronous scheduling to schedule VCPUs, the decoding progress of MPlayer under the Credit scheduler is very slow. On the other hand, because the Credit scheduler does not support soft real-time applications well, the decoded frames cannot be displayed in a timely fashion. Poris addresses both synchronization problems and soft real-time Constrains of PSRT applications. The decoding progress of MPlayer under Poris is very fast, and the decoded frames can be displayed in a timely manner.

6.3 Experiments in Server-side Virtualization

The above experiment shows Poris can guarantee the performance of MPlayer well compared with the Credit scheduler. In this part, we will extend the application type from MPlayer to a series of PSRT programs, and compare the performance of Poris with that of more related schedulers in server-side.

PARSEC benchmark suite [20] contains 13 multi-threaded programs from many different areas, such as computer vision, video encoding, financial analytics, animation physics and image processing. All of them are parallel programs, and some of them have soft real-time constraints. For example, fluidanimate simulates the underlying physics of fluid motion for real-time animation purposes with SPH algorithm. streamcluster computes an approximation for the optimal clustering of a stream of data points, and the data set has to be processed under real-time conditions.

In this test, we run PARSEC benchmark in a VM with eight VCPUs, and set the VM as RT-VM. We specify the thread parameter of PARSEC benchmark as eight threads, and choose native data set as the input set. We use three other VMs running CPU-intensive workloads to compete with the RT-VM. All of them are running on Machine II. For comparison, we also implement soft real-time scheduler (RS) and parallel scheduler (PS) under Xen-4.0.1 based on the descriptions in Section 4.2, 4.3 and 4.4 respectively. Fig.9 shows the results where the bars are normalized execution time.

As can be seen from the test results, for each individual benchmark, the performance of Poris is the best among these schedulers. This is because Poris addresses real-time constraints and synchronization problems simultaneously. The performance of Poris is up to 44.12%, 28.02% and 41.28% better than Credit, PS and RS respectively. Moreover, we can observe that, Poris can improve the performance of not only PSRT applications, but also parallel non-real-time applications to some extent.

6.4 Experiments with Distributed Soft Real-Time Applications

In the previous experiments, all the applications are running in a RT-VM. In this test, we run PSRT applications
in multiple VMs and want to show the importance of addressing the synchronization problems among VMs.

1) **The configurations and placement of RT-VMs.** A number of PMs (1–4) with the same configuration of Machine II are used to host RT-VMs. Each PM runs four RT-VMs and three interfering VMs running CPU-intensive applications simultaneously. All the RT-VMs in a PM are set as a group, and we set the number of VCPUs of each RT-VM in the group as 2 VCPUs and 4 VCPUs respectively, which represent different cases. For simplicity, we call these cases as 2VCPUs/RT-VM and 4VCPUs/RT-VM. The difference between these cases is whether the total number of RT-VCPUs exceeds the number of PCPUs.

2) **PSRT applications.** We use two PSRT applications to conduct experiments in this section, that are web search benchmark in CloudSuite [21] and a spark streaming program (WordCount in this test). The first is a representative of traditional PSRT applications, while spark streaming [22] is an emerging real-time streaming processing framework.

Web search is one of the most frequent used Internet services in our daily life. User experience depends on the response time of search engine. Many optimizations have been adopted by search engines to enhance user experience. For instance, Google Instant [23] can show results as you type. Besides, with the prevalence of social network, real-time search becomes a trend [24]. In these cases, web search is an PSRT application, which requires low response time and searches results in parallel.

Spark streaming is a real-time streaming processing framework, which enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Streams are everywhere, such as twitter streams and log streams. As a result, spark streaming is an attractive tool to process these streams in real-time. WordCount, which is implemented on top of spark streaming, counts the number of words in the input streams.

3) **Scheduling approaches.** We compare Poris to two other scheduling approaches as follows:
   - Credit: the default CPU scheduler of Xen hypervisor.
   - Poris s: a parallel soft real-time scheduler which is not aware of multiple VMs described in our preliminary work [25]. It only addresses synchronization problems in a single VM.

4) **Experimental results** Fig.10 shows the average response time of the web search benchmark. With the increase of RT-VMs, the average response time of the web search benchmark is increased very slightly under Poris. However, it is increased sharply under the Credit scheduler, especially when the number of RT-VMs is 16. If the total number of RT-VCPUs is the same in both cases, the average response time is smaller in the case of 4VCPUs/RT-VM. This is because that local interdomain communication latency is smaller than network latency across PMs [26]. In the case of 2VCPUs/RT-VM, compared with the Credit scheduler and Poris s, Poris achieves up to 91.6% and 71.6% reduction in average response time respectively. Accordingly, in the case of 4VCPUs/RT-VM, Poris achieves up to 90.6% and 66.3% reduction in average response time respectively.

Fig.11 shows the performance of WordCount under different cases. The performance metric of this application is the processing rate of input streams. Several
these CPU schedulers, and Poris applications need more CPU resources, such as the web search benchmark, consolidating PSRT applications are sensitive to communication latency, such as the web search benchmark, consolidating PSRT applications are sensitive to communication latency. PSRT applications are sensitive to communication latency, such as the web search benchmark, consolidating PSRT applications are sensitive to communication latency. Third, if the total number of VCPUs of these RT-VMs is the same in both cases, the performance of WordCount in the case of 2VCPUs/RT-VM is better than the other case. The main reason is that $C_{\text{sync}}$ cannot be eliminated if the number of RT-VCPUs in a group is greater than the number of PCPUs. In the case of 2VCPUs/RT-VM, compared with the Credit scheduler and Poris, $s$, Poris improves the processing rate by up to 53.1% and 37.7% respectively. Accordingly, in the case of 4VCPUs/RT-VM, Poris improves the processing rate by up to 66.7% and 42.7% respectively.

From Fig.10 and Fig.11, we can also observe that if PSRT applications are sensitive to communication latency, such as the web search benchmark, consolidating RT-VMs on a small portion of PMs is better. If PSRT applications need more CPU resources, such as WordCount, it is better to distribute these RT-VMs across more PMs.

In summary, Poris addresses both soft real-time constraints and synchronization problems, which improves the performance of PSRT applications significantly. On the contrary, neither of them is considered by the Credit scheduler. As a result, it results in poor performance for these applications. Because Poris ignores synchronization problems among VMs, it does not behavior well when PSRT applications run in multiple VMs. Moreover, since Poris does not need global controls to schedule RT-VMs distributed across multiple PMs, it has better scalability.

6.5 Experiments with Non-real-time Workloads

In this test, we use three applications with different characteristics to represent non-real-time workloads, and study the impact of Poris on them. Kernel compilation consumes a large amount of CPU resources, which is a CPU-intensive application. We use multiple threads to collaboratively compile Linux-3.12.5 kernel source. The second non-real-time workload is Postmark [27], which provides workload that generates random I/O operations on multiple small files. It is an I/O-intensive application. The last one is the STREAM benchmark [28], which is a memory-intensive application.

We use the same configurations as Section 6.4 except that evaluation is conducted on a PM because these non-real-time workloads can only run in a VM. Besides, as to the three interfering VMs, one VM is used to run non-real-time workloads while the others are still interfering VMs. Four RT-VMs run the web search benchmark. We send search requests to the web search benchmark and measure the performance of non-real-time workloads under Poris and the Credit scheduler. As to the non-real-time workloads, we set the threads number of kernel compilation as the VCPU number of the VM (i.e. 8), the transactions of Postmark as 400000, and NTIMES of the STREAM benchmark as 200. Because this test is used to show the impact of Poris on non-real-time workloads, we only concern the performance metrics of non-real-time applications. Specially, the performance metric of the STREAM benchmark is memory bandwidth of copy operation. The test results are shown in Fig.12.

There are two major factors that affect the performance of non-real-time workloads under Poris. One is preemptions caused by real-time priority. The other is short time slice, which increases context switches. From the test results, we can observe that Poris almost has no impact on kernel compilation and Postmark, and even increases the performance of Postmark in the case of 2VCPUs/RT-VM. The reasons are as follows. First, although kernel compilation is a CPU-intensive application, it also has lots of disk I/O operations. Second, short time slice can improve I/O performance. Finally, there are less RT-VCPUs in the case of 2VCPUs/RT-VM, which causes less preemptions. Both influence factors introduced by Poris may increase cache misses, but it only introduces a little interferences to the STREAM benchmark (less than 11.5%) in both cases. This is because when we design Poris we always consider how to minimize the impact on non-real-time applications.

In summary, because Poris promotes the priorities of RT-VCPUs temporarily and uses dynamic time slices, the interferences of Poris on non-real-time workloads are slight and acceptable. As a result, it can be applied to either private or public clouds.

7 Related Work

VMMs always schedule VCPUs of a VM asynchronously, which breaks some assumptions of operating systems. For example, operating systems always use spinlocks to protect critical sections in kernel. The asynchronous scheduling in VMM causes the lock-holder preemption
Fig. 12. The impact of Poris on non-real-time workloads

Besides, there are some studies about parallel real-time scheduling in traditional environment [30–33]. However, virtualization introduces another layer, which causes additional problems, such as two-level scheduling problems and LHP problems. And the interference among VMs is not a trivial issue. So, designing a new CPU scheduling algorithm for PSRT applications in virtualized environments is more complicated and has to face some new challenges.

8 Conclusion

In this paper, we identify the scheduling problems in virtualized environments when PSRT applications run in a VM or multiple VMs, and find existing CPU scheduling mechanisms do not fit for PSRT applications. Aiming at both the soft real-time constraints and synchronization problems, we design and implement a parallel soft real-time scheduler based on Xen, named Poris, which introduces priority promotion and dynamic time slice mechanism to determine when to schedule VCPUs. If PSRT applications run in a VM, parallel scheduling is used to address synchronization problems in a VM. Accordingly, if they run in multiple VMs, group scheduling and communication-driven group scheduling are activated to solve synchronization problems among VMs co-located in a PM or distributed across multiple PMs respectively. We conduct various experiments to validate the effectiveness of Poris. The experimental results show that Poris guarantees the performance of PSRT applications well no matter how they run in a VM or multiple VMs and outperforms traditional schedulers significantly.

Acknowledgements

The research is 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.2015AA011402, and Research Fund for the Doctoral Program of MOE under grant No.20110142130005.

References


