QoS Scheduling for Latency-sensitive Cloud Applications

Luwei Cheng*, Cho-Li Wang

Department of Computer Science, The University of Hong Kong

Abstract

Virtualization based cloud computing hosts networked applications in virtual machines (VMs), and provides each VM the desired degree of performance isolation using resource isolation mechanisms. The current resource sharing methods for virtual machines (VMs) mainly focus on resource proportional share such as CPU amount, memory size and I/O bandwidth, whereas ignore the fact that I/O latency in VM-hosted platforms is mostly related to resource provisioning rate. Even the VM is allocated with adequate resources, if they can not be provided in a timely manner, problems such as network jitter will be very serious and significantly affect the performance of cloud applications like internet audio/video streaming. This paper systematically analyzes the causes of unpredictable network latency and proposes a combined solution to guarantee network performance isolation: (1) in the hypervisor, we design a proportional share CPU scheduling with real-time support to reduce scheduling delay for network packets; (2) in network traffic shaper, we introduce the concept of smooth window with feedback control to smooth the packet delay. We implement our solutions in Xen 4.1.0 and Linux 2.6.32.13. The experimental results with both real-life applications and low-level benchmarks show that our solutions can significantly reduce network jitter, and meanwhile effectively maintain resource proportionality.

1. Introduction

Cloud computing is a new computing paradigm to transform computing services into a utility, just as providing electricity in "pay-as-you-go" model. Virtualization technologies are increasingly adopted in modern data centers to host cloud applications. By giving virtual machines (VMs) the illusion of owning dedicated physical resources, multiple VMs can share the single physical infrastructure. In order to guarantee the performance isolation of co-located VMs, Virtual Machine Monitors (VMM, also called hypervisor) such as VMware [1], Xen [2] and KVM [3], orchestrate sophisticated resource controls to CPU, memory and I/O allocations.

Effective management of networking resource is fundamental to data centers, which offers significant benefit of performance predictability for applications. The burgeoning of various types of cloud applications like audio/video streaming, interactive online gaming and e-commerce, have fueled research interest to focus on the design of virtualization-based service provisioning with satisfactory Quality-of-Service (QoS) guarantee. For example, Amazon CloudFront [4] uses a global network of edge locations to deliver streaming content. Since these applications are typically I/O intensive with special requirements for I/O latency, arbitrary sharing of resource infrastructure can lead to significant performance interference among VMs. Nowadays, running forty to sixty VMs per physical host is not rare. With hardware becoming more and more powerful, the consolidation level will be much higher in the future, which makes the I/O problems more challenging. The inevitable trend requires more effective I/O isolation techniques to provide predictable I/O performance for VMs.

Unlike most other applications which are comparatively more tolerant of underlying platform performance, media applications are far more demanding. Early studies [5, 6, 7] showed that network delay with small variation is tolerable and does not affect user-received media quality. This is because the clients usually adopt buffer mechanism to store certain amount of media data before playing them. However, the network delay with large jitter (variation in packet arrival time) will make the commonly used buffer mechanism ineffective and significantly degrade the received video quality.

In this paper, we systematically analyze the causes of unpredictable network latency in VM-hosted platforms, in both technical discussion and experimental illustration. We identify that the I/O latency is jointly caused by VMM CPU scheduler and network traffic shaper, and then address the problem in these two parts. In our solutions, we consider the design goals of resource provisioning rate and resource proportionality as two orthogonal dimensions. In VMM CPU scheduler, we characterize VM's I/O as two types: self-initiated I/O and event-triggered I/O. We map them into periodic domains and aperiodic domains, and then propose an algorithm which supports both real-time

*Corresponding author. Tel: +852 2857 8463; Fax: +852 2559 8447

Email addresses: lwcheng@cs.hku.hk (Luwei Cheng), clwang@cs.hku.hk (Cho-Li Wang)

URL: http://www.cs.hku.hk/~lwcheng (Luwei Cheng), http://www.cs.hku.hk/~clwang (Cho-Li Wang)

scheduling and CPU time proportional. In network traffic shaper, we introduce the concept of smooth window to mitigate the network jitter caused by varied packet sending delays. Meanwhile, in order to guarantee maintain network bandwidth allocation is not violated, the closed-loop feedback control theory is applied to adaptively control the packet sending rate by dynamically adjusting the smooth window position.

The rest of this paper is organized as follows. In Section 2, we systematically analyze the causes of network delay in VM-hosted platforms, and in Section 3, we introduce our design and architectures. The Implementation is presented in Section 4 and evaluation results are shown in Section 5. Section 6 discuss related work and Section 7 concludes. In appendices, we give detailed descriptions of the algorithms.

2. Problem Analysis

We take Xen as the example to systematically analyze what factors affect the network latency perceived by end users in VM-hosted platforms. Figure 1 illustrates how guest domain transmits network data to the outside world under split-driver model. Specifically when the guest domain is scheduled (①), the I/O from VM will firstly be sent to its frontend driver (②); the frontend driver will transfers the ownership of the memory pages to its corresponding backend driver and notify the driver domain via event channel mechanism (③). When the driver domain is scheduled by VMM CPU scheduler (④), it will see the pending I/O and get data from shared memory (⑤); then the data will be handed over to the network traffic shaper for rate limiting (⑥), before sending it to real device driver for transmitting (⑦).

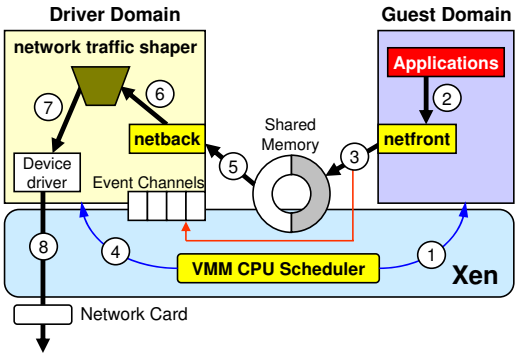


Figure 1: The path of outgoing I/O in VM-hosted platform

The data travel path from end users to virtual machine follows the reverse path, as shown in Figure 2. Compared with Figure 1, the main difference for ingress I/O is that network administrators seldom set restrictions on the server side for rate limiting. First, the requests from users are usually small network packets whereas the replied data from the server can be quite big, most commonly zone transfers; Second, since the number internet users is huge and they are mostly anonymous, in most cases it is unnecessary and also impractical to control the users' behaviors each and every.

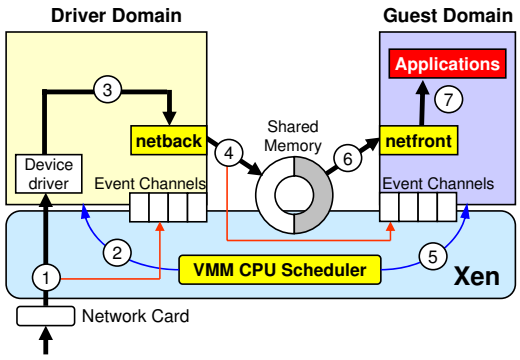


Figure 2: The path of ingress I/O in in VM-hosted platform

It can be seen that both guest domains and driver domain suffer scheduling delays from VMM CPU scheduler, which will be imposed on I/O packets' sending delays. For outgoing I/O from guest domains, since the driver domain acts as the I/O proxy, only when the driver domain is scheduled the traffic shaper inside it can take effect. In network traffic shaper, rate limiting is achieved by delaying packets: if the VM's bandwidth consumption does exceed its allocation in a certain period, the current packet can be processed immediately; otherwise, the packet has to be delayed for some time before transmission.

$$Lat_{total} = Lat_{vmm} + Lat_{shaper} \quad (1)$$

$$Lat_{vmm} = Lat_{dom0} + Lat_{domU} \quad (2)$$

The network bandwidth can be easily controlled in network traffic shaper, but network latency can be unpredictable as both VMM CPU scheduler and network traffic shaper contribute to it. The current resource sharing methods for VMs mainly focus on resource proportionality maintaining, whereas ignore the fact that I/O latency is mostly related to resource provisioning rate. The resource isolation with only quantitative promise does not sufficiently guarantee performance isolation, as resource provisioning with different time granularity can result in different response speed to VM's I/O. Even the VM is allocated with adequate resources such as CPU time and network bandwidth, large I/O latency can still happen if the resources are provisioned at inappropriate moments. So in order to achieve performance isolation, the problem is not only *how many* resources each VM gets, but more importantly whether the resources are provisioned in a *timely* manner.

From the perspective of VMM CPU scheduler, the scheduling entity it faces is virtual CPU (vCPU) and once the vCPU is scheduled, batches of I/O packets can be handled immediately with almost no delay. Therefore, the VM's I/O latency is actually VM's scheduling delay. Since VMM CPU scheduler has no direct control on I/O packets,

it is infeasible to 'smooth' the latency. More reasonably the scheduling delay should be 'reduced' in a best-effort way or in the user-specified manner, because if the delay can be reduced to a low level, the network jitter will also be low. In network traffic shaper, the situation is different in that it directly schedules network packets so it can explicitly control the delay of each packet, thus it is possible for us to apply smoothing policy.

2.1. Characterizing VM's I/O type

In virtualized environment, the notifications from VMM to VMs or between VMs are mostly delivered through event mechanism. Xen adopts event mechanism to replace hardware interrupt for asynchronous I/O delivering. The VM is marked with external event pending so it perceives the waited I/O. VMM CPU scheduler also takes advantage of event mechanism to make scheduling decisions. Xen's credit CPU scheduler adopts boost mechanism to accelerate I/O speed which favors to schedule the domain that receives external events. This works well for VMM to schedule the driver domain because all I/O events must be delivered to the driver domain first for proxy, no matter it is incoming I/O to guest domain or outgoing I/O from guest domain. However, the vulnerability of boost mechanism is that, not all I/O for guest domains are event-triggered. In the following two subsections, we characterize VM's I/O into two types: external event-triggered I/O and self-initiated I/O. We use real examples to illustrate the rationality of this classification.

2.1.1. I/O Triggered by External Events

This type of I/O is identified as that the end users are not only the I/O receivers but meanwhile, they are also I/O initiators. A very good example can be found in VM-hosted web servers. Each time when the users want to obtain web pages, files and etc, they will send out an HTTP request to the VM, and once the request arrives the VM is notified by receiving external events. In this way, the hypervisor knows that there is pending I/O for VM, so it can take advantage of this knowledge to schedule the VM as soon as possible. After the VM is scheduled, it can immediately satisfy the users' requests by sending back the specified files.

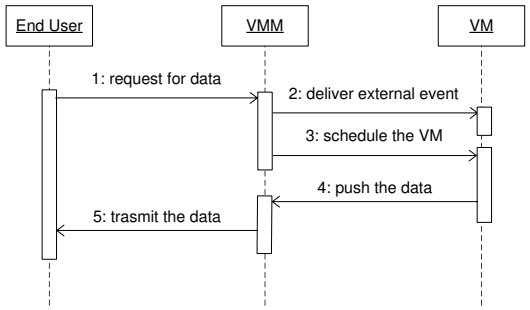


Figure 3: The data delivery model of event-triggered I/O

It is easy to control the latency of this I/O type because it follows the "request-reply" model. The end users will explicitly notify the hypervisor which VM needs to be scheduled, so the I/O delay can be precisely determined by controlling the scheduling delay of the VM. From the perspective of the VM, it needs to be scheduled in the real time way only when the external events are received. In other cases, the VMM CPU scheduler can simply focus on CPU time proportional share, regardless of the scheduling frequency of VMs.

2.1.2. Self-initiated I/O from Inside VM

This type of I/O has no external triggering source but it must also be issued in a timely manner. Examples can be found in applications for the purpose of controlling and monitoring: the server periodically sends instructions/requests to the clients to perform status polling, information updating and etc. Since the actions of the clients are totally driven by the server, if the instructions can not be issued by the server within the expected period, the job of the clients will inevitably be delayed. Another common example can be seen in UDP-based applications, such as RTP [8] media streaming. The end users are only I/O receivers and never tell the server which frames they currently need. But user-perceived video quality totally depends on the way that media data is delivered by the server. If the desired data frame does not arrive at the expected moment, the user experience will be seriously affected. Unlike event-triggered I/O, this type of I/O is actually "self-initiated" from inside VM. The VMM scheduler has no knowledge of when the VM should be scheduled, but the user-perceived I/O latency completely relies on how the VM is scheduled. Once the VM yields the CPU time (the idle process in guest OS), it can only rely on system virtual interrupts (such as VIRQ_TIMER in Xen) to make the VMM CPU scheduler aware that it needs to be scheduled again.

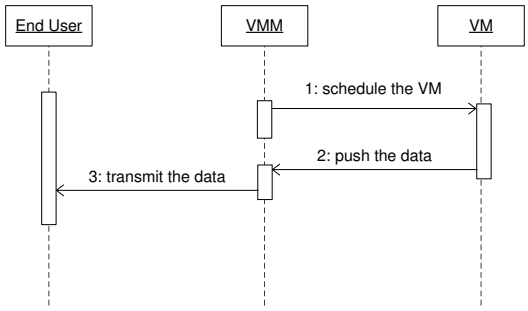
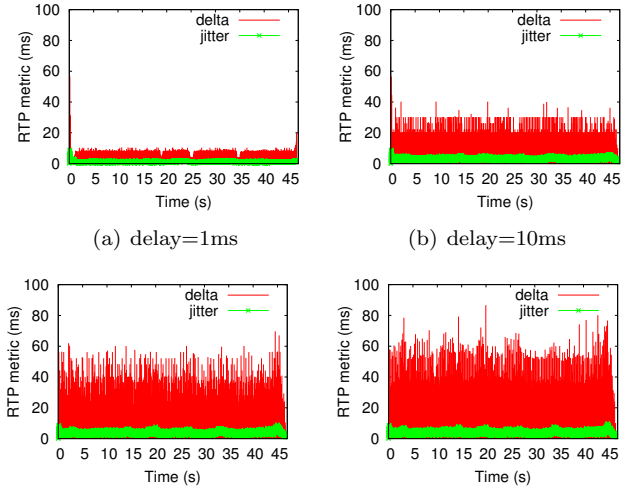


Figure 4: The data delivery model of self-initiated I/O

To illustrate the effect of different scheduling delays on self-initiated I/O, we use RTP video streaming as a case for evaluation. Since RTP streaming data are UDP packets and no external events from clients are involved during streaming period, it is typically self-initiated I/O. The VM runs alone on a dedicated physical core so that it owns the

whole CPU cycles of that core. In each test, when the VM voluntarily yields CPU time, we activated it again after every 1ms, 10ms, 20ms and 40ms respectively, as shown in Figure 5. During all four tests, the VM only consumes about 60% CPU time. Experimental results show that even the VM is provided with enough CPU resource, if the CPU cycles are not provisioned in a timely manner, the I/O performance will also be significantly affected.



(a) delay=1ms (b) delay=10ms

(c) delay=20ms (d) delay=40ms

Figure 5: The effect of different scheduling delays on self-initiated I/O

2.2. The Deficiency of Xen's Credit CPU Scheduler

Xen's credit scheduler introduces a boost mechanism to improve VM's I/O performance. The basic idea is to temporarily give the vCPU that receives external events a `BOOST` priority with preemption, which is higher than other vCPUs in `UNDER` and `OVER` state. However, the current implementation sets the limitation that the vCPU is boosted only when it is in block state and has not used up its credits. This is because it assumes that the I/O-intensive VMs usually consume little CPU cycles and stay in block state most of time. The assumption may hold in traditional process scheduling of OS, but may not always be true in virtual machine scheduling. With applications encapsulated in one VM, multiple processes/threads exist in guest OS. Take media streaming application for example, one I/O-bound thread is responsible for sending data frames and consumes little CPU time, meanwhile another CPU-bound thread performs encoding/decoding functionalities. So from the perspective of VMM CPU scheduler, the VM is both I/O-intensive and CPU-intensive. It is very possible that the vCPU is already in runqueue when I/O events arrive. In this case, the events can be handled only when the vCPU gets next scheduled, resulting in increased response time. Besides, when the VM yields the CPU time it may have used up its credit. Since the blocked VM stops earning credits, it may not get boosted due to credit shortage when it receives I/O events. It is not fair because the VM voluntarily yields CPU time in sacrifice of its own share, thus it should be compensated when it needs CPU cycles next time. Therefore, even for event-triggered I/O, the original CPU scheduler can not effectively schedule the VM to serve it, let alone self-initiated I/O.

2.3. Latency Caused by Network Traffic Shaper

In order to avoid performance interference among co-located VMs which share the same network resource, and also fit the "pay-as-you-go" model of cloud computing, the network traffic shaping (rate limiting) is widely adopted to shape network bandwidth of each VM. However, traffic shaping is always achieved by delaying packets, which has significant effect on user-perceived network latency.

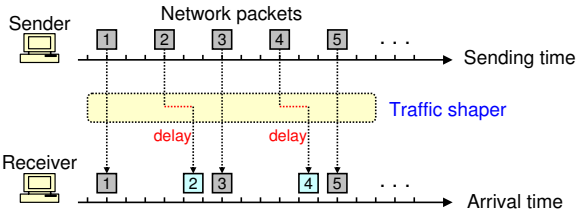


Figure 6: The unpredictable network delay in token-bucket algorithm

Xen implements token-bucket algorithm [9] to perform rate limiting among VMs. token bucket algorithm works in the way that if the remaining tokens (credits) in the bucket are enough to send the current packet, the packet will be issued immediately without delay. Whereas if the tokens are in lack, the packet has to be postponed for certain time to wait for tokens to be replenished. The major disadvantage is that it is bandwidth-oriented but not packet-oriented, which means that it works well in bandwidth maintenance but has no guarantee for the delay of each packet. As shown in Figure 6, with some packets delayed and some others not, the packets' arrival times can be quite different from their sending times. The large variation network latency leads to significant network jitter.

3. Proposed Solution

To provide network performance isolation, we introduce our new resource isolation methods which guarantee both resource provisioning rate and resource proportionality. Our approach includes two components: VMM CPU scheduler and network traffic shaper, which work together to smooth the network latency.

3.1. Proportional Share CPU Scheduling with Real-time Support

As explained in Section 2, for event-triggered I/O, the VMs need to be scheduled in the real-time way only when the external events are received, so we map this type of VM to aperiodic real-time domains; for self-initiated I/O, the VMs have no external notification for scheduling but they

must also be scheduled in the real-time way, therefore they are mapped to periodic real-time domains, which means that the scheduler will periodically wake them up to serve I/O.

3.1.1. Double-runqueue with Differential Time Slice

We introduce the double-runqueue design for each physical CPU core, as shown in Figure 7. EDF (Earliest Deadline First) runqueue is responsible to satisfy real-time VMs, sorted by their vCPUs' deadlines. Credit-runqueue takes the role to maintain CPU time proportionality, sorted by their remaining credits. For periodic domains, they can stay in both EDF-runqueue and Credit-runqueue. For aperiodic domains, only when external events are received they are considered as real-time domains and can enter EDF-runqueue; otherwise, they are regarded as normal domains which can only stay in Credit-runqueue. To avoid that the credit consumption of EDF-vCPUs affect the CPU time proportionality, the vCPUs from EDF-runqueue are assigned with small time slice whereas the vCPUs from Credit-runqueue will get long time slice. Since we allow the real-time vCPUs to preempt the others, the length of time slice that each vCPU receives will not affect the scheduling latency of real-time domains.

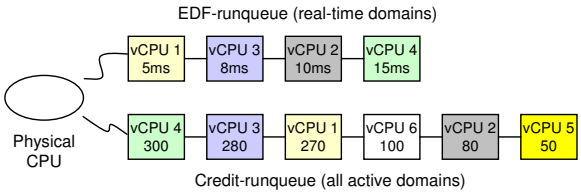


Figure 7: The double runqueue design for per physical CPU core

3.1.2. Credit Allocation

A very important cause of non-deterministic I/O latency is that the VM can not be scheduled to serve I/O due to credit shortage. In guest OS, I/O-bound processes usually have higher priority than CPU-bound processes and consume little CPU time. But if we simply allow all credits consumed by the CPU-bound processes and then put the VM in block state, even when the external events come, they can not be handled by I/O processes because the VM can not be scheduled by VMM. In order to avoid the possible credit shortage which may prevent the VM from serving the external events, we propose a credit reservation mechanism. Each VM will reserve certain amount of credits within the credit accounting period, and these credits can only be used when the VM receives I/O events in block state. To guarantee that CPU time proportionality is not affected, if the VM does not use up its reserved credits in the current accounting period, the remaining credits will be added to its next accounting period.

3.1.3. Domain Placement Policy

Since both ingress and outgoing I/O traffic of guest domains must traverse the driver domain, the scheduling delay of the driver domain has much more serious effect on the I/O latency than that of guest domains, which has been pointed out in [10, 11, 12]. We propose a simple domain placement policy by favoring the scheduling of driver domain in multi-core platform. Specifically, the domains are classified as three different types: the driver domain, real-time guest domains and normal guest domains. First, the driver domain can preempt any other domain but can not be preempted by the others. Second, real-time guest domains can not reside on the same physical core with the driver domain, so as to avoid competition for scheduling opportunities. The original load balancing mechanism is still kept to distribute vCPUs across all available physical CPU cores.

3.2. Latency Smoothing in Network Traffic Shaping

The original token-bucket algorithm mainly focuses on bandwidth maintaining, regardless of the delays of network packets. Our algorithm keeps the merits of limiting average bandwidth, but outperforms it by providing smoothed network latency as shown in Figure 8.

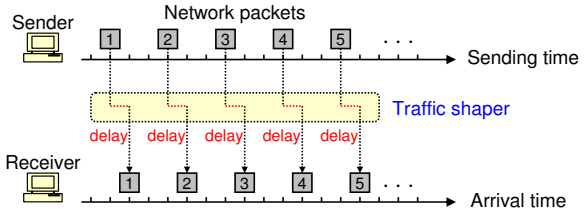


Figure 8: The proper way to delay packets

3.2.1. Smooth Window with Feedback Control

To guarantee that the network delay does not largely vary, the *smooth window* $w = [d_{min}, d_{max}]$ is introduced. The imposed delay value d_i on each packet p_i, must be within the range of smooth window: $d_i \in w$. The discrete packet flow is converted into continuous stream flow in the flowing way: for each packet p_i of size s_i, the equivalent credit consuming rate $r_i = \frac{s_i}{d_i}$, thus $r_i \in [\frac{s_i}{d_{max}}, \frac{s_i}{d_{min}}]$. So in order to guarantee that the bandwidth consumption does not exceed the limit, the average credit consumption rate must be no more than the credit replenish rate (derived from bandwidth allocation). However, due to unpredictable characteristics of bypassing packages (e.g. varied packet size and packet arriving speed), it is very hard to rule the relationship between package's delay and the credit consumption rate. If the packets are issued too fast with low delays, the high credit consumption rate will violate bandwidth allocation; whereas if they are issued too slowly with high delays, it will result in low bandwidth utilization. Therefore, to dynamically tune packets' delay level and the credit consumption rate, we adopt closed-loop feedback method to construct a Proportional-Integral-Derivative (PID) controller, as illustrated in Figure 10.

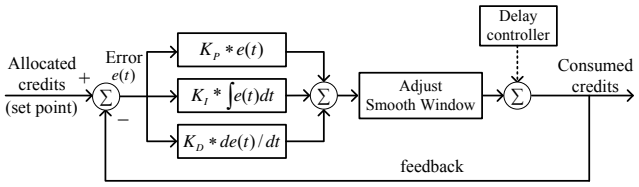


Figure 9: Closed-loop feedback control in network traffic shaper

The controller measures the error $e(t)$ between the consumed credit and allocated credit (set point). The proportional part reacts to the current value of the error, the integral part accounts for the recent history in error, and the differential part calculates the recent change in error. The weighted sum of these values is used as control input to adjust the position of smooth window ($w = [d_{min}, d_{max}]$). Specifically, during each window adjusting interval, we first use a pure proportional controller to perform a raw feedback control on the credit consumption rate: if the current consumed credit level is lower (greater) than the set point, the package's delay will be set to the d_{max} (d_{min}). In this way, the packets' delays won't largely vary since they are all within the smooth window range. However, such a raw adjustment may introduce overshoot (e.g. oscillation of credit level) in the long term. In next adjusting period, the smooth window position will be automatically adjusted according to the credit deviation level from the set point of last period. Therefore in general, the feedback controller corrects the credit over-consumption when there is a sustained positive error and vice versa.

3.2.2. Packet Filtering

We do not impose delay on every bypassing network packet. This is because for some types of packets are of small size and only for management purpose. For example, ARP who-has packet is 42 bytes and ARP reply packet is 60 bytes; ICMP echo packet is 98 bytes; TCP SYN packet is 74 bytes; TCP ACK and FIN are both 66 bytes. Though their sizes are quite small, delaying them can significantly affect the network performance perceived by the clients: (1) TCP SYN and ACK packets are quite important to establish TCP 3-way handshake connection; (2) for each application data transmission, an ACK packet needs to be sent for notification. Compared with the packets which carry the real application data such as RTP packet (1370 bytes) and normal TCP data packet (can be several kilobytes), these packets consume very few credits, so letting them pass immediately will not have too much negative effect on bandwidth shaping.

4. Implementation

Our VMM CPU scheduler is implemented in Xen 4.1.0. We firstly extend Xen tools to allow users to specify real-time domains with desired deadline requirements in VM configuration file. The vCPUs in EDF runqueue are sorted by their deadlines and likewise in Credit runqueue, they are sorted by the remaining credits. Each physical CPU involves a timer to periodically check the first vCPU's deadline in EDF runqueue. The timer period is currently set at 0.5ms so the scheduling error of each real-time domain won't exceed 0.5ms. In order to avoid that the scheduling behavior of EDF runqueue affects CPU time proportionality, each vCPU from EDF runqueue will receive only 0.5ms time slice while the vCPUs from Credit runqueue will receive 30ms time slice. Since we allow the EDF-vCPU to preempt the current vCPU, a long time slice for Credit runqueue won't cause scheduling delay for real-time domains. The credit accounting algorithm is also modified to allow each real-time domain to reserve certain amount of credits for I/O, under the circumstance that external events arrive when they are in block state. The original load balancing mechanism is still kept to distribute vCPUs across all available physical CPU cores. For non-real-time domains, the pervious BOOST mechanism is still kept to improve their I/O performance. But even they are boosted they can not enter EDF runqueue to compete for scheduling opportunity with real-time domains. So only when there are no real-time domains or the deadline of the first vCPU in EDF-runqueue is not reached, the boosted vCPU from Credit runqueue is scheduled.

The closed-loop feedback controller for network traffic shaping is implemented in Linux 2.6.32.13. The tick rate `HZ` in Linux kernel is modified from 100 to 1000, because with `HZ` set at 100 by default, the timer precision of `jiffies` is only 10ms which is too coarse to control the delay of network packets. The smooth window size and window adjusting interval are two tunable parameters, and choosing values for them is actually the tradeoff between latency smoothing level and bandwidth maintaining accuracy. In our current implementation, we set the window size to be 3ms with window adjusting interval of 1 second, which seem to work well for most cases in practice.

5. Performance Evaluation

The server we use to host virtual machines is equipped with two quad-core Intel Xeon 5540 2.53GHz CPUs, and 16GB physical memory. Several testing clients are connected with the server through a Gigabit Ethernet switch. For self-initiated I/O evaluation, we downloaded a video from YouTube.com as the example, and use *VLC media player* to deliver the streaming data from hosted VM to the testing clients, based on RTP protocol. The network packets are decoded using *Wireshark* for RTP stream quality analysis. For event-triggered I/O evaluation, we use *ApacheBench*, a web site stress test benchmark, to measure HTTP service quality. Besides the two application-level benchmarks, we also use low-level benchmarks such *Ping*, *Netperf* and *Iperf* in the evaluation.

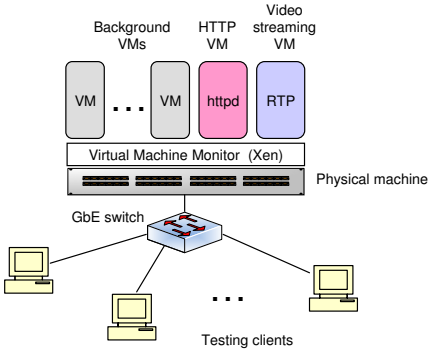


Figure 10: Experimental Setup

5.1. Network Jitter over Internet

We first evaluate network delay characteristics over Internet to see how serious network jitter is. If the Internet jitter is already quite heavy, even we solve the problem of unstable network delay in the VM-server side, the overall jitter perceived by Internet end users will still be intolerable. In Figure 11, the ping results with four different web sites show that the Internet jitter is relatively low.

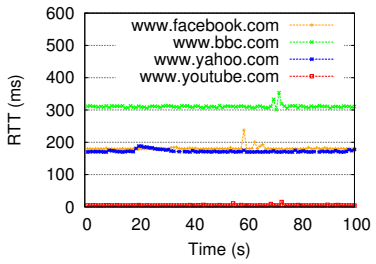


Figure 11: Internet jitter

5.2. Evaluation of our VMM CPU Scheduler

We first evaluate the effect of our new VMM CPU scheduler on reducing latency of self-initiated I/O. In Figure 12, VM 1 runs as streaming server with two CPU intensive VMs (VM 2 and VM 3) on the same physical core. Since when VM 1 runs alone, it consumes about 55% CPU time during streaming period. So in order to avoid that the streaming quality will be affected by insufficient CPU cycles, we allocate 60% CPU time to VM 1. The remaining 40% CPU time is evenly allocated between VM 2 and VM 3. With Xen's default CPU scheduler, it can be seen that after VM 2 starts, the RTP metric of VM 1 is significantly degraded; after VM 3 starts, the negative effect is even more serious.

For comparison in Figure 13, we use our new CPU scheduler and set VM 1 to be periodic real-time domain with deadline set at 3ms and 5ms respectively. During the whole video streaming period, VM 1 runs along with VM 2 and VM 3. It can be observed that the performance of VM 1 only depends on user-defined deadlines, and is not affected by co-located VMs.

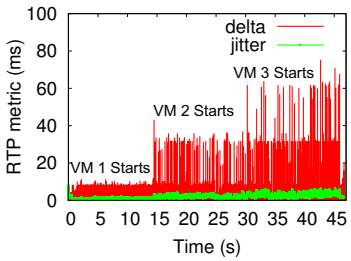
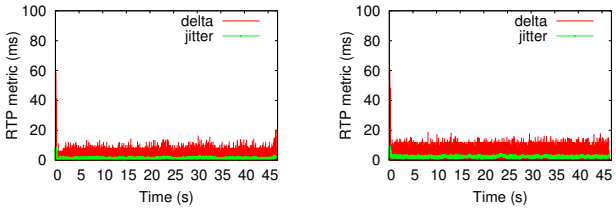


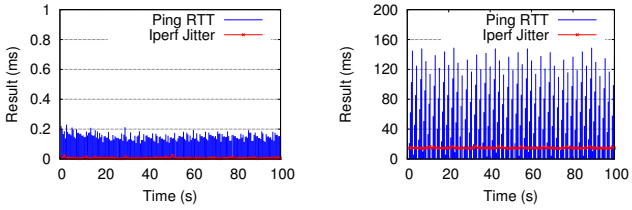
Figure 12: The effect of Xen's credit scheduler on RTP video streaming



(a) deadline=2ms (b) deadline=4ms

Figure 13: The effect of our VMM CPU scheduler on RTP video streaming

We then evaluate the network latency behaviors of event-trigger I/O under our new VMM CPU scheduler. On the client side, we use *Ping* with 0.5 second interval to measure the RTT latency to the testing VM. For each test, we also use *Iperf* to measure the network jitter (Iperf also uses RTP protocol to calculate network jitter). In Figure 14, we illustrate how VM will suffer unpredictable and long network latency in consolidation scenarios using Xen's credit scheduler: when the testing VM runs alone on the physical core, the ping latency is quite small as shown in Figure 14 (a); however, when the testing VM runs with five co-located CPU-intensive VMs, the ping latency increases significantly, as shown in Figure 14 (b). For network jitter, it also increases along with more co-located VMs. Take Figure 14 (c) for example, with five co-located VMs, the ping latency to the testing VM can be as high as 150ms. This can be explained that Xen's credit scheduler uses 30ms time slice to schedule VMs, and with six VMs running on the same CPU core, the maximum waiting time of each VM is 5×30 ms.

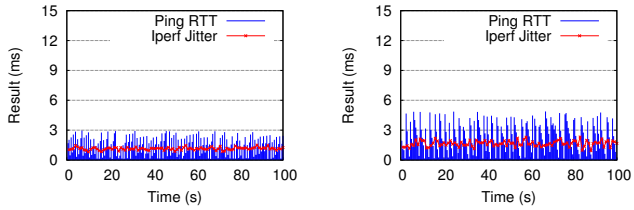


(a) run alone (b) run with 5 VMs

Figure 14: Evaluation results of Xen's credit CPU scheduler

We evaluate our new VMM CPU scheduler in Figure

15. We set the testing VM to be aperiodic real-time domain with deadline set at 3ms, 5ms and 8ms respectively. We setup the stress tests by running the testing VM with five CPU-intensive VMs on one physical core all the time. Results show that the network latency can be well controlled under the user-defined deadline requirements, and is not affected by co-located VMs. Accordingly, the network jitters are also much lower than that in Figure 14.



(a) deadline = 3ms (b) deadline = 5ms

Figure 15: Evaluation results of our new VMM CPU scheduler

In Figure 16, we evaluate the ability to keep CPU time proportionality of our VMM CPU scheduler. We run six VMs on a single physical core with each allocated the same relative weight. The test lasted for 1000 seconds. Results show that our CPU scheduler performs fairly on allocating CPU time.

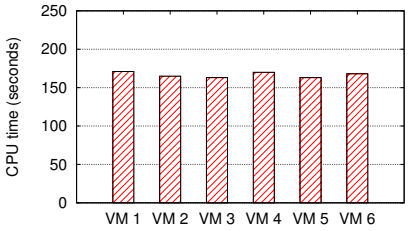
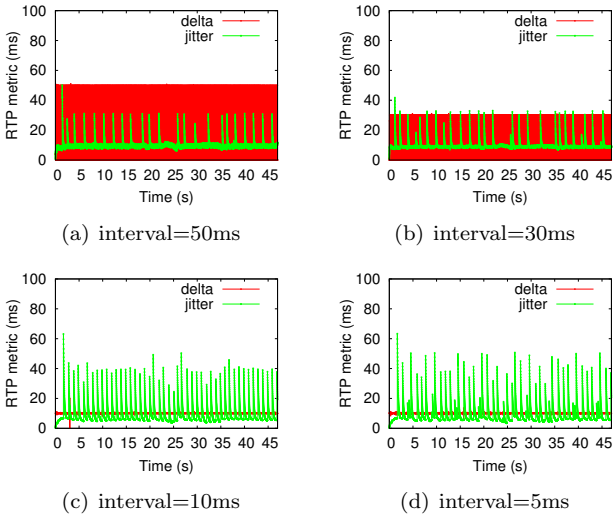


Figure 16: CPU time proportional share

5.3. Evaluation of our Network Traffic Shaper

In RTP video streaming tests, we use a 128MB VM as the streaming server, which runs alone at a dedicated physical core so that it won't be negatively affected by CPU scheduler. The VM's network bandwidth is set at 2Mb/s, which is consistent with the output rate of the video we set at VLC scripts. We first illustrate how video streaming suffers large jitter problem under Xen's traffic shaper, which adopts token-bucket algorithm. In Figure 17, we set the credit replenish interval to be 50ms, 30ms and 5ms respectively. Results show that when the interval tends to be small, RTP delta will decrease; however, RTP jitter does not decrease at all. For example, when the credit replenish interval is set at 5ms, the RTP jitter in Figure 17 (c) is even more serious than that in Figure 17 (a) and (b). This proves that without smoothed packet sending delays in token-bucket algorithm, only using small credit replenish interval has no benefit in improving video streaming quality.

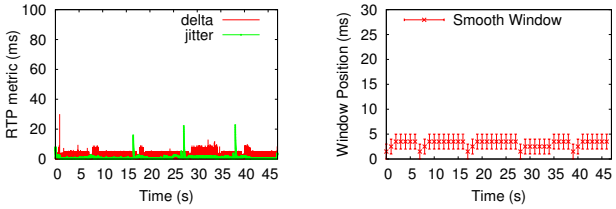


(a) interval=50ms (b) interval=30ms

(c) interval=10ms (d) interval=5ms

Figure 17: RTP video streaming using Xen's traffic shaper

We then use our new traffic shaper to smooth the network latency, and see how video streaming quality is improved. As shown in Figure 18 (a), with smoothed network latency, both RTP delta and RTP jitter are significantly reduced. Figure 18 (b) shows how smooth window position is automatically adjusted according to the dynamic bandwidth consumption. Since the packet sending delays are all within the smooth window size ($w = [d_{min}, d_{max}]$), the overall network jitter is greatly brought down.



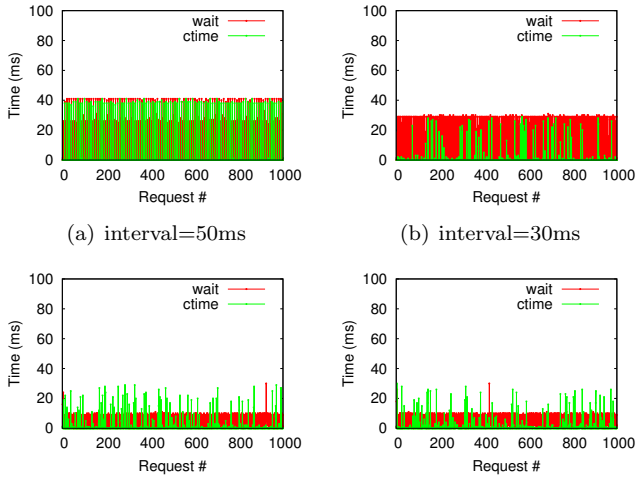
(a) after smoothing (b) auto-adjustment of smooth window position

Figure 18: Latency smoothing effect on RTP video streaming

We also evaluate the smoothing effect on improving HTTP service quality, in terms of HTTP connection time (denoted by `ctime`) and HTTP waiting time (denoted by `wait`). In Figure 19, we run a 128MB VM as HTTP web server. In the client side, we use *ApacheBench* to send 1000 HTTP requests with a mixed workload, including 10% 2KB file, 20% 4KB file, 40% 8KB file, 20% 16KB file and 10% 32KB file.

Results in Figure 19 show that with Xen's default setting: (1) HTTP connection time can be kept very low, but when credit replenish interval becomes smaller, HTTP connection time tends to increase as shown Figure 19; (2) HTTP waiting time can vary significantly with different credit replenish interval, the smaller the better;

The results with latency smoothing policy are shown

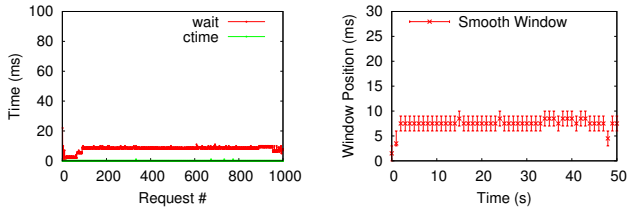


(a) interval=50ms (b) interval=30ms

(c) interval=10ms (d) interval=5ms

Figure 19: HTTP service quality using Xen's traffic shaper (mixed)

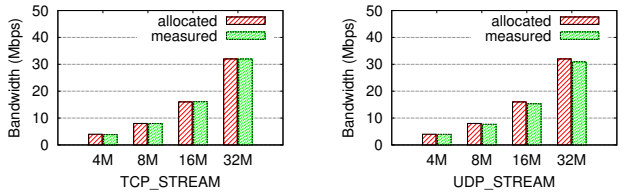
in Figure 20. It can be seen that our solution can *automatically* smooth HTTP waiting time and reduce HTTP connection time.



(a) after smoothing (b) auto-adjustment of smooth window position

Figure 20: Latency smoothing effect on HTTP service (mixed)

We use *Netperf* to evaluate the network rate limiting effect of our solution. The testing VM is allocated with 4Mbps, 8Mbps, 16Mbps and 32Mbps respectively, with each test lasting for 120 seconds. Figure 21 shows the evaluation results with a micro-view which are recorded by every two seconds. Experimental results with both TCP and UDP tests show that, our solution has very effective control on bandwidth shaping and meanwhile achieve high resource utilization.



(a) TCP bandwidth test (b) UDP bandwidth test

Figure 21: Network bandwidth shaping

6. Related Work

Significant effort has been paid to address the I/O performance of virtual machines in recent years. Hardware-based solutions such as [13] use SR-IOV devices to assign the VM dedicated device to guarantee its I/O performance. However, the special hardware devices are usually expensive and complicate the common functionalities such as live migration and checkpointing. Researches [14] and [15] proposed task-level solutions to map VM's I/O-bound tasks directly to physical CPU. The drawback is that additional hypercalls are needed and users have to explicitly tell VMM which tasks they want to map. Our approach is non-intrusive and do not need extra modification to guest OS. Besides, The philosophy of our method is different from other real-time schedulers such as [12] and [16]. First, instead of scheduling real-time domains in an best-effort way, we allow users to specify different level of real-timeness. Second, our solution decouples the goal of CPU time proportionality from CPU scheduling rate.

Resource sharing approaches can be classified as work-conserving (WC) mode and non-work-conserving (NWC) mode. WC approaches allow clients to consume more that their allocations when there are idle resources, thus improve the resource utilization. NWC solutions provides clients predictable network bandwidth by forcing them to consume no more than the allocations even when there are idle resources, such as leaky-bucket [17] and token-bucket [9] algorithm which are largely adopted in real-life systems. Research [18] use "virtual time" proposed in [19], but they focused on storage I/O instead of network I/O. Research [20] did the similar work to smooth network latency, however their solutions are incomplete as they did not address the VMM scheduling delays. Research [21] adopts play-back buffer to smooth network delays, but they do not guarantee unaffected bandwidth allocation; additionally, play-back buffer brings extra memory overhead.

7. Conclusions and Future Work

The paper systematically addresses the network jitter problem in VM-hosted platforms, which significantly affect network performance isolation for certain cloud applications. Our design guarantees both resource proportionality and resource provisioning rate. The solutions have been implemented and thoroughly evaluated for single host running multiple VMs. Our future work will explore their utilities in: (1) storage subsystems; (2) network performance isolation for large amount of distributed VMs in datacenters.

8. Acknowledgments

The authors would like to acknowledge the program members from UCC 2011 conference for their helpful comments and kind recommendation. This research is supported by a Hong Kong RGC grant HKU 7179/09E and a

HKU Basic Research grant (Grant No. 10401460), and also in part by a Hong Kong UGC Special Equipment Grant (SEG HKU09).

References

[1] C. A. Waldspurger, Memory resource management in vmware esx server, ACM SIGOPS Operating Systems Review 36 (2002) 181–194.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, ACM SIGOPS Operating Systems Review 37 (2003) 164–177.

[3] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, kvm: the linux virtual machine monitor, in: The 2007 Ottawa Linux Symposium, pp. 225–230.

[4] Amazon CloudFront: http://aws.amazon.com/cloudfront/.

[5] M. Claypool, J. Tanner, The effects of jitter on the peceptual quality of video, in: Proceedings of the 7th ACM International Conference on Multimedia, New York, NY, USA, pp. 115–118.

[6] M. J. Karam, F. A. Tobagi, Analysis of delay and delay jitter of voice traffic in the internet, Computer Networks 40 (2002) 711–726.

[7] L. Zhang, L. Zheng, K. S. Ngee, Effect of delay and delay jitter on voice/video over ip, Computer Communications 25 (2002) 863–873.

[8] RFC 3550: http://www.ietf.org/rfc/rfc3550.txt.

[9] S. Shenker, J. Wroclawski, General characterization parameters for integrated service network elements, RFC 2215 (1997).

[10] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, A. Sivasubramaniam, Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms, in: Proceedings of the 3rd International Conference on Virtual Execution Environments, New York, NY, USA, pp. 126–136.

[11] D. Ongaro, A. L. Cox, S. Rixner, Scheduling i/o in virtual machine monitors, in: Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, New York, NY, USA, pp. 1–10.

[12] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, S. Yajnik, Supporting soft real-time tasks in the xen hypervisor, in: Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, volume 45, New York, NY, USA, pp. 97–108.

[13] J. R. Santos, Y. Turner, G. Janakiraman, I. Pratt, Bridging the gap between software and hardware techniques for i/o virtualization, in: USENIX Annual Technical Conference, Berkeley, CA, USA, pp. 29–42.

[14] H. Kim, H. Lim, J. Jeong, H. Jo, J. Lee, Task-aware virtual machine scheduling for i/o performance., in: Proceedings of the 5th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, New York, NY, USA, pp. 101–110.

[15] R. Rivas, A. Arefin, K. Nahrstedt, Janus: a cross-layer soft real-time architecture for virtualization, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, New York, NY, USA, pp. 676–683.

[16] J. Nieh, M. S. Lam, A smart scheduler for multimedia applications, ACM Transactions on Computer Systems 21 (2003) 117–163.

[17] H. Chao, Design of leaky bucket access control schemes in atm networks, in: IEEE International Conference on Communications, volume 1, Denver, CO, USA, pp. 180–187.

[18] A. Gulati, A. Merchant, P. J. Varman, mclock: handling throughput variability for hypervisor io scheduling, in: Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation, Berkeley, CA, USA, pp. 1–7.

[19] J. Bennett, H. Zhang, Wf2q: worst-case fair weighted fair queueing, in: IEEE International Conference on Computer Communications, volume 1, pp. 120–128.

[20] M. Kesavan, A. Gavrilovska, K. Schwan, Differential virtual time (dvt): rethinking i/o service differentiation for virtual machines, in: Proceedings of the 1st ACM Symposium on Cloud Computing, New York, NY, USA, pp. 27–38.

[21] A.-O. Huthaifa, P. Christos, W. Francis, M. David, Smoothing delay jitter in networked control systems, Journal of Embedded Computing 4 (2010) 11–21.