Defeating Network Jitter for Virtual Machines

Luwei Cheng, Cho-Li Wang
Department of Computer Science
The University of Hong Kong
{lwcheng, clwang}@cs.hku.hk

***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. Existing isolation solutions address heavily on resource proportionality such as CPU, memory and I/O bandwidth, but seldom focus on 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 and illustrates the causes of unpredictable network latency in virtualized execution environments. We decouple the design goals of resource proportionality from resource provisioning rate, and adopt divide-and-conquer strategy to defeat network jitter for VMs: (1) in VMM CPU scheduling, we differentiate self-initiated I/O from event-triggered I/O, and individually map them into periodic and aperiodic real-time domains to schedule them together; (2) in network traffic shaping of VMs, we introduce the concept of smooth window to smooth network latency and apply closed-loop feedback control to maintain network resource consumption. 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 while effectively maintain resource proportionality.**

I. Introduction

Modern data centers are increasingly adopting virtualization software for the purpose of server consolidation, flexible resource management and better fault tolerance. 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 Monitor (VMM, also called hypervisor) such as VMware [1] and Xen [2], orchestrates sophisticated resource controls to CPU, memory and I/O allocations. The burgeoning of various types of cloud applications such as 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. Since these applications are typically I/O intensive with special requirements for I/O latency, arbitrary sharing of resource infrastructure leads to significantly variable service rate to VMs, and thus violates QoS metrics. Nowadays, running forty to sixty VMs per physical host is not rare [3], which means that on a physical machine with like eight CPU cores, there will be as many as ten VMs sharing one physical core on average. With hardware becoming more and more powerful, the consolidation level will be much higher, which makes the I/O problems more challenging. This inevitable trend requires more effective I/O isolation techniques to provide predictable I/O latency for VMs.

Cloud based media streaming services such as Amazon CloudFront [4], use a global network of edge locations to deliver streaming content. Researches [5], [6] have shown that for media streaming, constant 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 will make the commonly used buffer mechanism ineffective and significantly degrade the received video quality. Research [7] has also pointed out that for TCP protocol which adopts adaptive control mechanism, large jitter in network latency can significantly affect TCP performance, because TCP congestion control algorithm heavily relies on network latency prediction to control the TCP window size.

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 proportion promise does not sufficiently guarantee performance isolation, as resource provisioning with different time granularity can significantly affect VM's responsiveness to I/O. Even the VM is allocated with adequate resources such as CPU time and network bandwidth, large I/O latency will still happen if the resource is provisioned at inappropriate moments. So in order to achieve performance isolation, the problem is not only *how much* resource each VM gets, but more importantly whether the resource is provisioned in a *timely* manner. In our research, we consider that the two design goals should be orthogonal and do not interfere each other.

In this paper, we systematically analyze and illustrate the causes of network latency in VM-hosted platforms, which is jointly caused by VMM CPU scheduling and network traffic shaping. According to different data delivery model, we characterize VM's I/O type as self-initiated I/O and event-triggered I/O. To address the non-deterministic scheduling delay in VMM CPU scheduling, we individually map the two types of VMs into periodic and aperiodic real-time

domains to schedule them together. Specifically, we propose a double runqueue design for each physical CPU, which can provide real-time scheduling and meanwhile maintain CPU time proportional share. In network traffic shaping for VMs, we introduce the concept of smooth window to mitigate the network jitter caused by varied packet sending delays. And in order to guarantee that 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 contribution of this paper can be identified as: (1) we systematically address network jitter problem for VMs, which is quite important to achieve satisfactory QoS for cloud applications like audio/video streaming; (2) Our solutions decouple the design goals of resource proportional share from resource provisioning rate; (3) we implement our solutions in Xen 4.1.0 and Linux 2.6.32.13. (4) we conduct meaningful evaluations using both real-life applications and low-level benchmarks, and the results prove the effectiveness of our solutions.

The remainder of this paper is organized as follows. In Section II, we give a systematic analysis of network latency in VM-hosted platforms, and in Section III, we present our design and architectures. Implementation is discussed in Section IV and evaluation is presented in Section V. The related work is discussed in Section VI. Finally, we conclude our research in Section VII.

II. Systematic Analysis of Network I/O latency

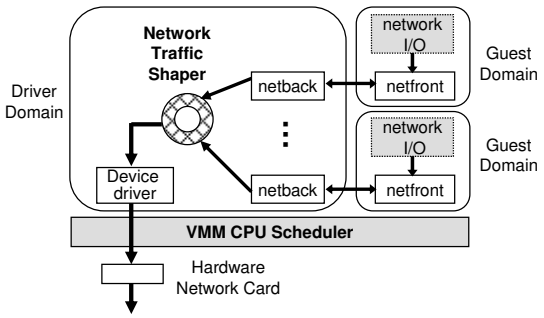


Figure 1. The path of network I/O from VM to outside clients

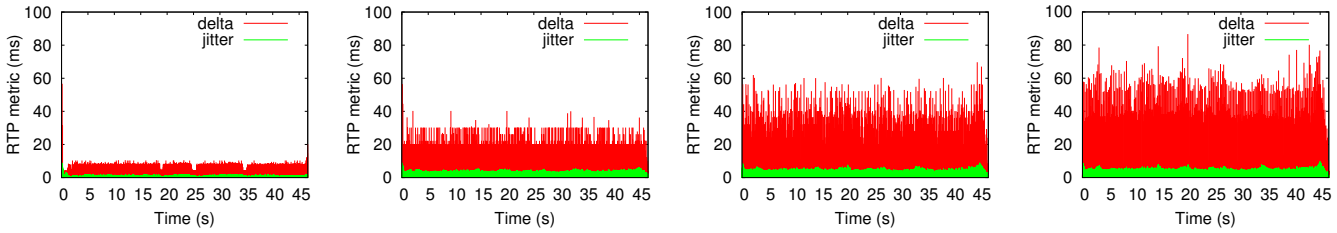
In this section, we take Xen [2] as the example, and systematically analyze that in VM-hosted platforms, what factors and how they affect the network latency perceived by end users. In Figure 1, we illustrate the path that network I/O packets have traversed from VM to end users. Specifically, the I/O from VM will firstly be sent to its front-end driver when the guest domain is scheduled, and then be handed over to the corresponding back-end drivers in the driver domain (domain 0), through event channel and shared memory mechanism [2] provided by Xen. When the driver domain is scheduled by VMM CPU scheduler, the I/O packets from each VM are further transferred to the network traffic shaper for rate limiting. It should be noted that both guest domains and the driver domain suffer scheduling delays caused by VMM CPU scheduler. If the I/O packet comes before the VM is scheduled, it has to wait until the VM gets CPU cycles. During the VM's scheduling time slice, batch of I/O packets can be handled. Since the driver domain acts as the I/O proxy for all guest domains, only when the driver domain is scheduled the traffic shaper inside it can take effect. For each packet in network traffic shaper, if the VM's remaining network resource is enough, the packet will be immediately delivered to the hardware network card with almost no delay. Otherwise if the allocated bandwidth has been consumed by previous packets and is not enough for the current packet, the packet will be inevitably delayed to wait for resource be replenished in next allocation epoch.

It can be observed that in order to address I/O latency problem, the strategy of divide-and-conquer should be adopted and the solutions should proposed in both VMM CPU scheduler and network traffic shaper. 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 handled immediately with almost no delay. So 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 latency should be 'reduced' in an best-effort way or in the user-specified manner, because if the latency 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 scheduled network packets so it can explicitly control the delay of each packet, thus it is possible for us to apply smoothing policy.

A. 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 [2]. 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 adopts boost mechanism [8] 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.

I/O Triggered by External Events. This type of I/O is identified as that the end users are not only the I/O receiver



(a) 1ms (b) 10ms (c) 20ms (d) 40ms

Figure 2. The effect of different scheduling delays on self-initiated I/O

but meanwhile, they are also I/O initiator. 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's pending I/O for VM, so it can take advantage of this knowledge to schedule the VM as soon as possible. Once the VM is scheduled, it can immediately satisfy the users' I/O requests by sending back HTTP replies including the specified files.

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 VM that it needs to be scheduled, so the I/O delay can be completely 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.

Self-initiated I/O from Inside VM. This type of I/O has no external trigger source but must be issued in a timely manner, thus we call it "self-initiated I/O". 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 the applications built above UDP protocol, such as RTP[1] based media streaming. The end users are only I/O receivers and never explicitly 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 cat not be received in the expected moment, the QoS and user experience will be seriously affected. Unlike the I/O triggered by external events, this type of I/O is actually self-triggered from inside VM. The VMM CPU scheduler has no knowledge of when the VM should serve the clients, but the user-perceived I/O latency completely replies on when the VM is scheduled. If the VM yields the CPU time (the idle process in guest operating system), it can only reply on system virtual interrupts (such as VIRQ_TIMER in Xen) to make the VMM CPU scheduler aware that it needs to be scheduled again.

To illustrate and verify 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 2. 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.

B. The Deficiency of Xen's CPU Scheduler

In Xen's credit scheduler, it introduces a boost mechanism [8] to improve the 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 time and stay in block state most of time. The assumption may hold in process scheduling in traditional operating system, but may not always be true in virtual machine scheduling.

With applications encapsulated in VM, more than one processes/threads exist in guest operating system. Take streaming application for example, one I/O bound thread is responsible for sending data frames and consumes little CPU time, meanwhile another thread performs encoding/decoding functionality and is CPU bound. 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 has 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

[1]RTP protocol: http://www.ietf.org/rfc/rfc3550.txt

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. So 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.

C. 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 control the consumed network bandwidth of each VM. However, traffic shaping is always achieved by delaying packets, which has significant effect on user-perceived network latency. Xen implements token bucket algorithm [9] to perform rate limiting among VMs.

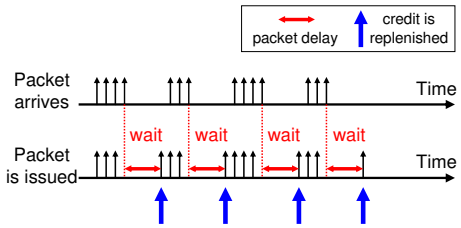


Figure 3. The network packet delay in token-bucket traffic shaping

As shown in Figure 3, token bucket algorithm works in the way that if the remaining tokens (credits) 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. This could cause large variation in network latency and thus leads to network jitter.

III. Proposed Solution

We introduce our new resource isolation methods, which decouple the design goal of resource provisioning rate from resource proportionality. Our approach is combined with two components which work together to smooth the network latency. In VMM CPU scheduler, we map the event-triggered I/O domain to aperiodic domain and map self-initiated I/O domain to periodic domain, and schedule them together in a real time way. In network traffic shaper, we introduce smoothing window to control the delay of network packets and apply closed-loop feedback control theory to adaptively adjust smoothing window to limit bandwidth consumption.

A. Credit-Independent real time CPU Scheduling

As explained in Section II A, 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 into aperiodic 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 domains, which means that the scheduler will periodically wake them up to serve I/O.

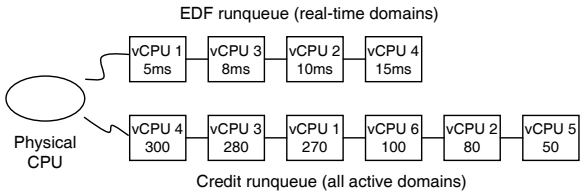


Figure 4. The double runqueue design for per physical CPU core

We introduce the double-runqueue design for each physical CPU core, as shown in Figure 4. 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 non-real-time domains and can only stay in Credit runqueue. In order 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 won't affect the scheduling latency of real-time domains.

Each physical CPU runs a thread to periodically poll the vCPUs' deadlines in EDF runqueue. If the first vCPU's deadline has reached, it will immediately preempt the current running vCPU. It is possible that the real-time vCPU is picked before its deadline from the Credit runqueue, in that case after the vCPU is descheduled, it will be re-inserted into EDF runqueue with a new deadline labeled.

Credit Reservation for I/O. As analyzed in section II B, 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 operating system, I/O bound processes usually consume little CPU time and have higher priority than CPU bound processes. But if we simply allow all credits consumed by its CPU bound processes and make the VM in the situation of credit shortage, 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 defeat the possible credit shortage which may prevents 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 epoch, 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 epoch, the remaining credits will be added to its next accounting epoch.

B. Domain Placement Policy

Since both I/O requests from guest domains and I/O replies to guest domains must traverse the driver domain, the scheduling delay to the driver domain has much more serious effect on the I/O latency, compared with the scheduling delay of guest domains. The negative effect of driver domain scheduling on I/O latency has been pointed out in [8], [10] and [11]. We adopt the similar approaches and propose a domain placement policy for multi-core platform. We classify the domains as three different types: the driver domain, real-time guest domains and non-real-time 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 with the driver domain on the same physical CPU core, so as to avoid competition for scheduling opportunities.

C. Latency Smoothing in Network Traffic Shaping

As explained in Section II C, the original token-bucket algorithm mainly focuses on bandwidth maintaining, regardless of the delays of network packets. The minimum packet delays can be zero and the maximum delay can be as high as replenishing period, which cause very large network jitter.

Smooth Window and Feedback Control. To guarantee that the network delay does not largely vary, the *smooth window* $w = [d_{min}, d_{max}]$ is introduced in our design. The imposed delay value d_i on each packet P_i, must be within the range of smooth window: $d_i \in w$. We convert discrete packet flow 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 $r_i \in [\frac{s_i}{d_{min}}, \frac{s_i}{d_{max}}]$. 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 [12] to construct a Proportional-Integral-Derivative (PID) controller, as illustrated in Figure 5.

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 (P 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's a sustained positive error and vice versa.

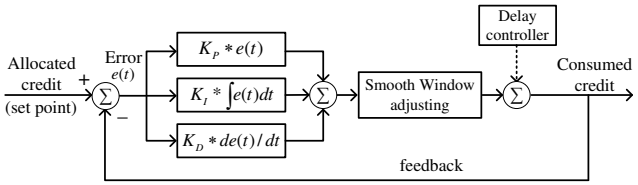
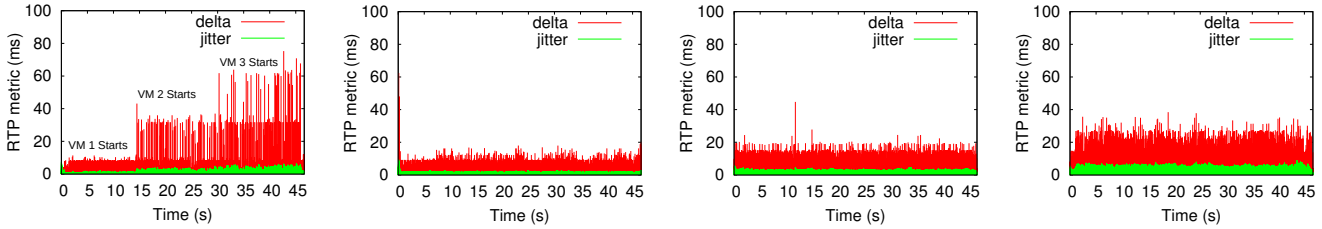


Figure 5. Closed-loop feedback control in network traffic shaper

IV. 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.

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



(a) Xen default (b) deadline = 3ms (c) deadline = 5ms (d) deadline = 8ms

Figure 6. The effect of our VMM CPU scheduler on RTP video streaming

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.

V. 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 an advertisement video from YouTube.com as the example, and then use VLC media player[2] to deliver the streaming data from hosted VM to the testing clients, based on RTP protocol. The network packets are decoded using Wireshark[3] for RTP stream 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 *Ping* and *Netperf* in the evaluation.

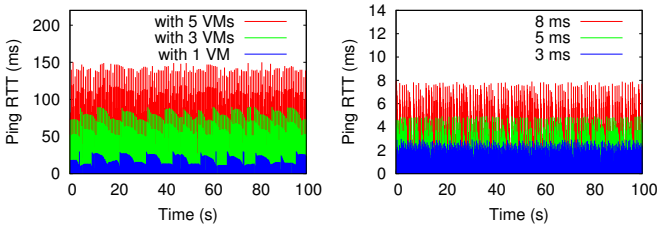
A. VMM CPU Scheduling on Controlling I/O Latency

We first evaluate the effect of our new VMM CPU scheduler on reducing latency of self-initiated I/O. In Figure 6, 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 in Figure 6 (a), 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 6 (b), (c) and (d), we use our new CPU scheduler and set VM 1 to be periodic real-time domain with deadline set at 3ms, 5ms and 8ms 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.

[2]VLC: http://www.videolan.org/

[3]Wireshark: http://www.wireshark.org/

We then evaluate the latency behaviors of event-trigger I/O under our new VMM CPU scheduler. We set up the stress test by individually running the testing VM together with five, three and one CPU intensive VMs on one physical CPU core. It should be noted that the testing VM is also CPU intensive. On the client side, we use ping with 0.1s interval to measure the ICMP latency to the testing VM. With Xen's default CPU scheduler in Figure 7 (a), the ICMP latency is significantly affected by the number of co-located VMs. For example, with 5 VMs running together with the testing VM, the ping latency can be as high as 150ms. This can be explained that Xen's CPU scheduler uses 30ms time slice, and with six VMs co-running on the same CPU core, the maximum waiting time of each VM is 5×30 ms. With our new VMM CPU scheduler in Figure 7 (b), we set the testing VM to be aperiodic real-time domain with deadline set at 3ms, 5ms and 8ms respectively. The testing VM also runs together with five CPU intensive VMs on one physical core. Results show that the network latency can be well controlled under the user-defined deadline requirements, and is not affected by co-located VMs.

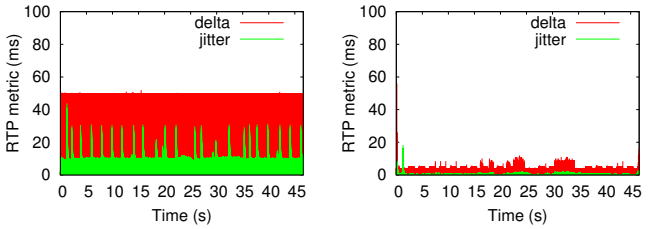


(a) Xen default (b) Our CPU scheduler

Figure 7. The effect of our VMM CPU scheduler on ping

B. Feedback Control on Smoothing I/O Latency

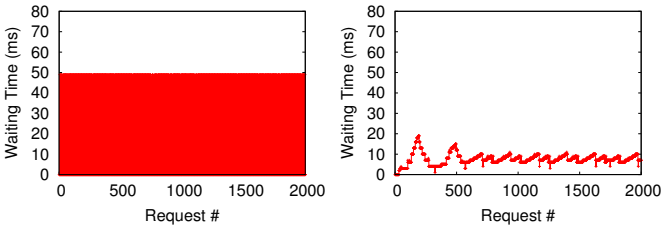
To evaluate the effect of our feedback control on reducing video streaming jitter, we use a 256MB VM as the streaming server which runs alone at a dedicated physical core. The VM's network bandwidth was set at 2Mb/s, which is consistent with the output rate of the video we use. Figure 8 (a) shows that with Xen's default setting which replenishes credit to VM at every 50ms, the RTP metric of VM shows very high jitter. Comparably in Figure 8 (b), since our feedback control method with smooth window can automatically smooth the delay of network packets, the RTP performance is significantly improved.



(a) Xen default (b) with feedback control

Figure 8. Feedback control on smoothing RTP video streaming jitter

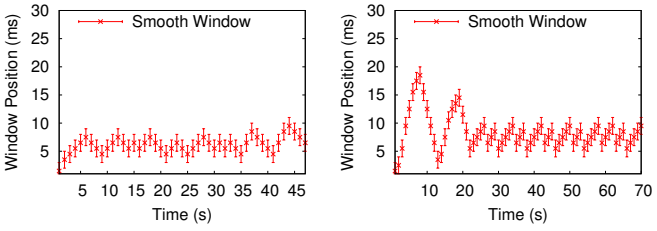
We also evaluate the smoothing effect on HTTP request waiting time in Figure 9. In client side, we use ApacheBench to send 2000 HTTP requests to the VM for a 8KB file. The experiment lasted for about 70 seconds. Results in Figure 9 (a) shows that with Xen's default setting, the waiting time jitters significantly which could range from 0ms to 50ms. While with our solution in Figure 9 (b), the waiting time can be greatly smoothed.



(a) Xen default (b) with feedback control

Figure 9. Feedback control on smoothing HTTP requests waiting time

Figure 10 shows the auto-adjustment of smooth window position during the above experiments. With feedback control, the smooth window periodically slides itself in an automatic and adaptive way, to provide smoothed latency and meanwhile maintain bandwidth consumption.

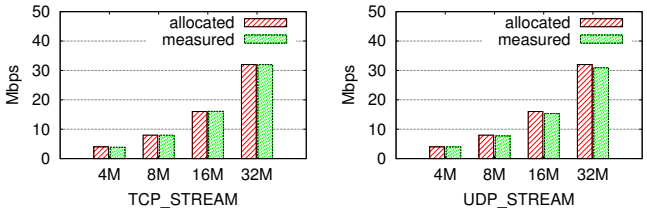


(a) RTP test (b) HTTP test

Figure 10. Automatic adjustment of smooth window position

C. Resource Proportionality Maintaining

We use Netperf to evaluate the effect of VM network rate limiting of our feedback control in Figure 11. The testing VM is allocated with 4Mbps, 8Mbps, 16Mbps and 32Mbps respectively. Each test lasted for 60 seconds and were conducted for three times to get the average value. 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 11. Network bandwidth shaping test

In Figure 12, we evaluate CPU time proportionality of our VMM CPU scheduler. On a single physical core, we firstly booted three VMs (VM1, VM2 and VM3), and after a short while we individually started another two VMs (VM4 and VM5). VM1 and VM2 were then stopped after running for certain time. All VMs are CPU intensive and allocated with the same relative CPU proportion. Results show that our CPU scheduler performs fairly on allocating CPU time.

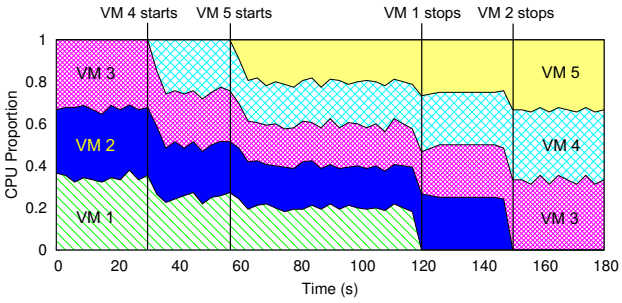


Figure 12. CPU time proportional share

VI. Related Work

Significant effort has been paid to address the I/O performance of virtual machines in recent years. Some researchers propose task-level solutions to map I/O bound tasks of VM directly to physical CPU, such as [13], [14]. 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 [8] and [15]. 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 mode and non-work-conserving mode. Working-conserving approaches allow clients to consume more that their allocations when there are idle resources, thus improve the resource utilization. Examples can be found in SFQ [16], WFQ [17] and mClock [18]. However, the major disadvantage is that it may cause large variation in VM's received resource allocation and introduces non-deterministic factor to network behavior. Non-work-conserving solutions force the

client to consume no more than its allocation even there are idle resources, such as leaky-bucket [19] and token-bucket [9] algorithm. Due to the predictable network bandwidth of non-work-conserving approaches, they are largely adopted in real-life systems. For instance, Linux implements Hierarchy Token Bucket (HTB) algorithm and Xen also adopts token-bucket algorithm to achieve rate limiting. For latency smoothing in network traffic shaping, there is comparatively lesser work. Some solutions use special hardware such as SR-IOV devices [20] or assign the VM dedicated device [21] to guarantee its I/O performance. However, these approaches are expensive and complicate the common functionalities such as live migration and checkpointing as sacrifice.

VII. Conclusions

Our paper addresses the network jitter problem in virtualized execution environment. We adopt a systematic analysis approach to identify the causes of network jitter. Our solutions decouple the design goals of resource proportionality and resource provisioning rate. We implement our solutions in Xen and Linux driver domain. Though the implementation is done in para-virtualization (PV), our solutions are generic and can be also applied to hardware virtualization (HVM) in that: first, VMM CPU scheduler treats HVM guests as the same as PV guests; second, the feedback control method in network traffic shaper can be easily extended to other similar network management utilities such as Linux's HTB and Open vSwitch [3]. The evaluations with both application-level benchmarks and low-level benchmarks prove the effectiveness of our solutions.

VIII. Acknowledgments

We thank the anonymous reviewers for their comments. 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). Thanks also to Xen's author Keir Fraser, for his kindness in answering our questions about Xen's network rate limiting.

References

[1] C. A. Waldspurger, "Memory resource management in vmware esx server," in *Proceedings of the 5th symposium on Operating Systems Design and Implementation*, vol. 36, New York, NY, USA, 2002, pp. 181–194.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, vol. 37, New York, NY, USA, 2003, pp. 164–177.

[3] P. Ben, P. Justin, K. Teemu, A. Keith, C. Martin, and S. Scott, "Extending networking into the virtualization layer," in *HotNets-VIII*, New York, NY, USA, 2009.

[4] Cloudfront: http://aws.amazon.com/cloudfront/.

[5] M. J. Karam and F. A. Tobagi, "Analysis of delay and delay jitter of voice traffic in the internet," *Computer Networks*, vol. 40, pp. 711–726, 2002.

[6] L. Zhang, L. Zheng, and K. S. Ngee, "Effect of delay and delay jitter on voice/video over ip," *Computer Communications*, vol. 25, pp. 863–873, 2002.

[7] M. Kesavan, A. Gavrilovska, and 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, 2010, pp. 27–38.

[8] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and 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*, vol. 45, New York, NY, USA, 2010, pp. 97–108.

[9] P. P. Tang and T.-Y. Tai, "Network traffic characterization using token bucket model," in *IEEE International Conference on Computer Communications*, vol. 1, New York, NY, USA, 1999, pp. 51–62.

[10] D. Ongaro, A. L. Cox, and 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, 2008, pp. 1–10.

[11] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and 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, 2007, pp. 126–136.

[12] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms*," *Real-Time Systems*, vol. 23, pp. 85–126, 2002.

[13] H. Kim, H. Lim, J. Jeong, H. Jo, and 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, 2009, pp. 101–110.

[14] R. Rivas, A. Arefin, and 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, 2010, pp. 676–683.

[15] J. Nieh and M. S. Lam, "A smart scheduler for multimedia applications," *ACM Transactions on Computer Systems*, vol. 21, pp. 117–163, 2003.

[16] P. McKenney, "Stochastic fairness queueing," in *IEEE International Conference on Computer Communications*, vol. 2, San Francisco, CA , USA, 1990, pp. 733–740.

[17] J. Bennett and H. Zhang, "Wf2q: worst-case fair weighted fair queueing," in *IEEE International Conference on Computer Communications*, vol. 1, 1996, pp. 120–128.

[18] A. Gulati, A. Merchant, and 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, 2010, pp. 1–7.

[19] H. Chao, "Design of leaky bucket access control schemes in atm networks," in *IEEE International Conference on Communications*, vol. 1, Denver, CO, USA, 1991, pp. 180–187.

[20] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," in *IEEE 16th International symposium on High Performance Computer Architecture*, 2010, pp. 1–10.

[21] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for i/o virtualization," in *USENIX 2008 Annual Technical Conference*, Berkeley, CA, USA, 2008, pp. 29–42.