Adaptive Dynamic Priority Scheduling for Virtual Desktop Infrastructures

Jinho Hwang and Timothy Wood
Department of Computer Science, George Washington University, Washington, DC
Email: {jinho10, timwood}@gwu.edu

***Abstract*—Virtual Desktop Infrastructures (VDIs) are gaining popularity in cloud computing by allowing companies to deploy their office environments in a virtualized setting instead of relying on physical desktop machines. Consolidating many users into a VDI environment can significantly lower IT management expenses and enables new features such as “available-anywhere” desktops. However, barriers to broad adoption include the slow performance of virtualized I/O, CPU scheduling interference problems, and shared-cache contention. In this paper, we propose a new soft real-time scheduling algorithm that employs flexible priority designations (via utility functions) and automated scheduler class detection (via hypervisor monitoring of user behavior) to provide a higher quality user experience. We have implemented our scheduler within the Xen virtualization platform, and demonstrate that the overheads incurred from co-locating large numbers of virtual machines can be reduced from 66% with existing schedulers to under 2% in our system. We evaluate the benefits and overheads of using a smaller scheduling time quantum in a VDI setting, and show that the average overhead time per scheduler call is on the same order as the existing SEDF and Credit schedulers.**

***Index Terms*—Xen, scheduler, virtual desktop infrastructure, desktop virtualization, cloud computing**

I. Introduction

Cloud computing infrastructure has seen explosive growth in the last few years as a source of on-demand storage and server power. Beyond simply being used to run web applications and large data analytic jobs, the cloud is now being considered as an efficient source of resources for desktop users. Virtual Desktop Infrastructure (VDI) systems seek to utilize network connected virtual machines to provide desktop services with easier management, greater availability, and lower cost.

Businesses, schools, and government agencies are all considering the benefits from deploying their office environments through VDI. VDI enables centralized management, which facilitates system-wide upgrades and improvements. Since the virtualized desktops can be accessed through a thin terminal or even a smartphone, they also enable greater mobility of users. Most importantly, companies can rely on cloud hosting companies to implement VDI in a reliable, cost-effective way, thus eliminating the need to maintain in-house servers and support teams.

To offer VDI services at a low cost, cloud providers seek to massively consolidate desktop users onto each physical server. Alternatively, a business using a private cloud to host VDI services may want to multiplex those same machines for other computationally intensive tasks, particularly since desktop users typically see relatively long periods of inactivity. In both cases, a high degree of consolidation can lead to high resource contention, and this may change very quickly depending on user behavior. Furthermore, certain applications such as media players and online games require high quality of service (QoS) with respect to minimizing the effects of delay. Dynamic scheduling of resources while maintaining high QoS is a difficult problem in the VDI environment due to the high degree of resource sharing, the frequency of task changes, and the need to distinguish between actively engaged users and those which can handle higher delay without affecting their quality of service.

Existing real-time scheduling algorithms that consider application QoS needs [1–4] use a fixed-priority scheduling approach that does not take into account changing usage patterns. Similarly, the scheduling algorithms included in virtualization platforms such as Xen [5] provide only coarse grain prioritization via weights, and do not support dynamic adaptation. This has a particularly harmful effect on the performance of interactive applications, and indicates that Xen is not ready to support mixed virtual desktop environments with high QoS demands.

For this paper we have enhanced the Xen virtualization platform to provide differentiated quality of service levels in environments with a mix of virtual desktops and batch processing VMs. We have built a new scheduler, D-PriDe[1], that uses utility functions to flexibly define priority classes in an efficient way. The utility functions can be easily parameterized to represent different scheduling classes, and the function for a given virtual machine can be quickly adjusted to enable *fast adaptation*. Utilities are also simple to calculate, helping our scheduler make decisions efficiently even though it uses a smaller scheduling quantum.

Our utility driven scheduler is combined with a monitoring agent built inside the hypervisor that enables automatic *user behavior recognition*. In a VDI consisting of a hypervisor and multiple VMs, the hypervisor is unaware of the types of applications running in each VM. However, knowledge of application behavior is important to the scheduler responsible for allotting system resources, e.g., to distinguish between VMs that have a user actively connected to them and ones which do not have any user interaction and thus are more tolerant

[1] D-PriDe stands for **D**ynamic-**Pri**ority **De**sktop Scheduler.

to service delays. In order to recognize user behavior and group VMs into scheduling classes, the proposed scheduler uses information obtained by the management domain about packets transmitted between the guest domains (VMs) and the external network.

This paper has the following main contributions:

1) A utility function-based scheduling algorithm that assigns VM scheduling priority based on application types, where fast adaptation is accomplished via linear functions with a single input argument.
2) A classification system that determines application type based on networking communication, and dynamically assigns VM scheduling priority using this information.
3) Experimental results that justify using smaller scheduling quanta than the quanta that are used in existing algorithms.

The remainder of this paper is organized as follows: Section II provides background on the standard schedulers in Xen, and on other system issues related to scheduling. Section III describes how to modify the Xen network structure to enable detection of VM scheduler classes. Section IV introduces the proposed utility-driven VDI scheduling algorithm. Section V presents results from several experiments. Finally, Sections VI and VII discuss related work, conclusions, and possible directions of future research.

II. Background

In this section, we provide background information on Xen schedulers, the VDI protocol, and hardware-related issues.

A. Xen Schedulers

The Xen hypervisor is used by many companies in the cloud computing business, including Amazon and Citrix. We describe the evolution of Xen's scheduling algorithms from the Borrowed Virtual Time (BVT) and Simple Earliest Deadline First (SEDF), to the currently used Credit algorithm [6].

BVT [7] is a fair-share scheduler based on the concept of virtual time. When selecting the next VM to dispatch, it selects the runnable VM with the smallest virtual time. Additionally, BVT provides low-latency support for real-time and interactive applications by allowing latency sensitive clients to "warp" back in virtual time and to gain scheduling priority. The client effectively borrows virtual time from its future CPU allocation.

The BVT scheduler accounts for running time in terms of a minimum charging unit, typically the frequency of clock interrupts. Each runnable domain i receives a share of CPU in proportion to its weight weight_i, and the virtual time of the currently running Dom_i is incremented by its running time divided by weight_i. This scheduler only supports work-conserving mode (WC-mode). In this mode, an idle CPU (with no runnable domains) can be consumed by a domain that does not normally have claim to that CPU. By contrast, in non work-conserving mode (NWC-mode), each domain is restricted to its own CPU share, even if another CPU is idle. The inability of BVT to support NWC-mode limits its usage in a number of environments, and has led to the development of another Xen Scheduler, SEDF [6].

SEDF uses real-time algorithms to deliver performance guarantees. Each domain Dom_i specifies its CPU requirements with a tuple (s_i, p_i, x_i), where the slice s_i and the period p_i together represent the CPU share that Dom_i requests: Dom_i will receive at least s_i units of time in each period of length p_i. The boolean flag x_i indicates whether Dom_i is eligible to receive extra CPU time (in WC-mode). SEDF distributes this slack time in a fair manner after all runnable domains receive their CPU share. For example, one can allocate 30% CPU to a domain by assigning either (3 ms, 10 ms, 0) or (30 ms, 100 ms, 0). The time granularity in the definition of the period impacts scheduler fairness.

For each domain Dom_i, the scheduler tracks an additional pair (d_i, r_i), where d_i is a time at which Dom_i's current period ends, also called the deadline. The runnable domain with the earliest deadline is selected to be scheduled next. r_i is the remaining CPU time of Dom_i in the current period. SEDF, however, is unable to perform global load balancing on multiprocessors. The Credit algorithm, discussed next, addresses this shortcoming.

Credit[1] is Xen's latest proportional share scheduler featuring automatic load balancing of virtual CPUs across physical CPUs on a symmetric multiprocessing (SMP) host [8]. Before a CPU goes idle, Credit considers other CPUs in order to find a runnable VCPU, if one exists. This approach guarantees that no CPU idles when runnable work is present in the system. Each VM is assigned a weight and a cap. If the cap is 0, then the VM can receive extra CPU (in WC-mode). A non-zero cap (expressed as a percentage) limits the amount of CPU a VM receives in NWC-mode. The Credit scheduler uses a 30 ms time quantum for CPU allocation. A VM (VCPU) receives 30 ms of CPU throughput before being preempted by another VM. Once every 30 ms, the priorities (credits) of all runnable VMs are recalculated. The scheduler monitors resource usage every 10 ms.

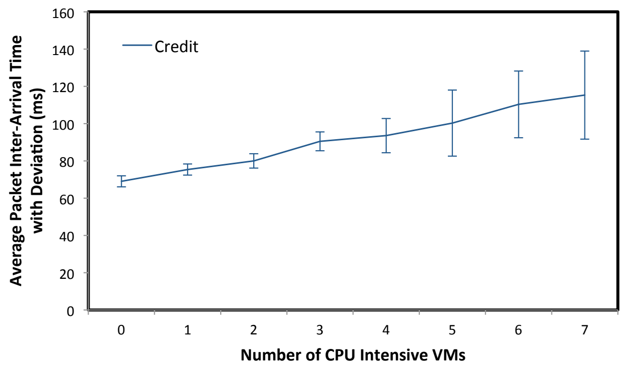


Fig. 1. The Credit scheduler causes the performance of a desktop VM to become increasingly variable as more interfering VMs are added to the machine.

Existing Scheduler Limitations: To demonstrate the per-

[1] Credit-based cpu scheduler, http://wiki.xensource.com/xenwiki/CreditScheduler

formance issues seen when using these schedulers, Figure 1 shows how the time between screen updates for a desktop virtualization client (measured by inter-packet arrival time) changes when adjusting the number of computationally intensive VMs causing interference. We see that with the Credit scheduler, the background VMs can increase the delay between VDI client updates by up to 66%. Further, the standard deviation of arrival times can become very large, making it impossible to offer any kind of QoS guarantees.

The existing scheduling algorithms satisfy fairness among VMs, but they are not well designed to handle latency sensitive applications like virtual desktops, nor do they provide support for dynamic changes of VM priorities. While the Credit scheduler used in our motivating experiment could be tweaked to give a higher weight to the VDI VM, this would only increase the total guaranteed share of CPU time it is allocated, not affect the frequency with which it is run. We propose D-PriDe, a scheduler that confronts these issues by using a low overhead priority scheduling algorithm that allocates VMs on a finer time scale than Credit. In addition, D-PriDe can detect when users log on or off of a desktop VM, allowing it to dynamically adjust priorities accordingly.

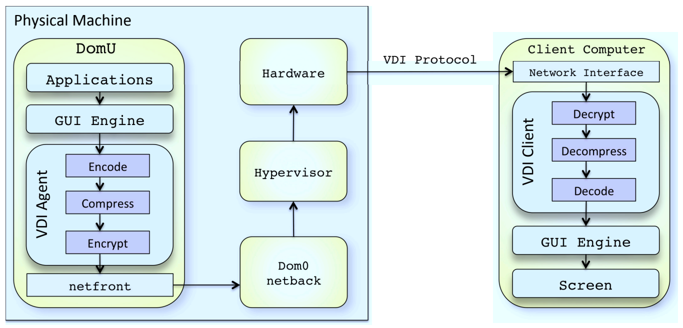


Fig. 2. Xen Networking Diagram with Simplified VDI Flow

B. VDI Protocol

The Internet Engineering Task Force [9] has recently drafted a preliminary VDI protocol problem statement. This document defines the VDI protocol to entail remote desktop access from either a PC, tablet computer, or mobile phone as a client device. The OS and applications of the desktop run on virtual machines, which are deployed in data centers. Mainstream commercial VDI products include Microsoft's RDS, Citrix's XenDesktop, Redhat's Enterprise Virtualization for Desktops, and VMware's VMware View. An open-source alternative, virtual network computing (VNC), uses the remote framebuffer protocol.

Figure 2 shows a Xen networking diagram with simplified VDI flow. Each guest domain (VM) has a VDI agent (server) running as a daemon and each client has a VDI client that connects to the VDI agent. A VDI service has high QoS requirements due to the streaming process, which involves encoding/compressing/encrypting the screen in the VDI agent and decrypting/decompressing/decoding the data in the VDI client. Streaming performance is directly related to user experience. In the Xen system, packets from a VDI agent are produced by $netfront$ in a VM, then delivered to $netback$ in Dom0, which sends them to the network interface hardware. The packets are transmitted on events such as screen changes, key strokes, or mouse clicks.

The base standard specification for the VDI protocol, ITU-T T.120 [10] specifies the terminal resource management for telematic services. Also, T.121 provides a template for T.120 resource management, which developers are encouraged to use as a guide for building VDI-related application protocols.

C. Cache Affinity

Cache affinity means keeping a VM within the same group of CPUs sharing cache at a certain level. It is important to keep a VM in the same cache group because cache misses, which involve retrieving data from memory, are very expensive. Sometimes, a scheduler considering SMP degrades system performance by moving VMs too often, resulting in cache trashing (cache data of a new VM replaces cache data of the resident VM) and future cache misses. CPU schedulers consider cache architecture in order to avoid cache trashing. In our scheduler design, we migrate VMs with lower scheduling classes to available CPUs first, so that VMs with higher scheduling classes are not affected by cache trashing.

III. Scheduler Class Detection

In a virtualized system such as Xen, the hypervisor is responsible for managing access to I/O devices, thus it has the capability of monitoring all network traffic entering and leaving a virtual machine. D-PriDe uses information about the network traffic sent from a VM to determine its scheduling class. D-PriDe uses packet information to distinguish between two main priority classes: VMs which have an active network connection from one or more desktop users, and those which are either being used for batch processing or have no connected users. If there are virtual machines detected that have online users, then they are granted a higher priority class and the system is switched to use a finer grain scheduling quantum, allowing interactive applications to achieve higher quality of service levels.

In Xen, the management domain is called Dom0, and we term a particular guest domain as DomU. D-PriDe modifies the scheduling operation hypercall (hypercall number 29) to enable cooperation between Dom0 and the hypervisor. As depicted in Figure 3, when DomU attempts to send a network packet, it is prepared in the $netfront$ driver and then handed off to the $netback$ driver to be fully processed. At this point, D-PriDe can inspect the packet and determine whether it matches the characteristics of an active VDI connection (e.g., based on port number). Dom0 then must make a hypercall so that the Xen scheduler will determine which virtual machine to schedule next. D-PriDe modifies this call so that it passes priority class information along with the hypercall. Thus whenever a VDI packet is detected in the outbound TCP buffer of a virtual machine, the Xen scheduler will elevate the virtual machine's priority level; if a timeout occurs before another VDI packet is seen, the priority level is reduced.

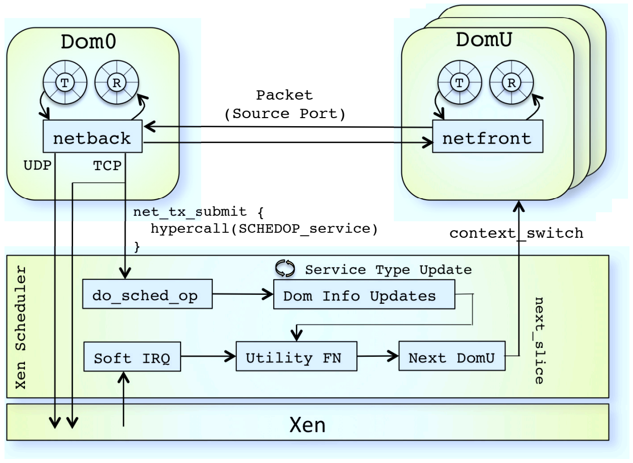


Fig. 3. D-PriDe Scheduler Architecture

D-PriDe places top emphasis on providing a positive user experience, and assigns scheduling classes to clients in order to schedule jobs with proper scheduling priority. We define three different VM scheduling classes as follows:

- Online Active (ONA): Client is actively using VD, and applications are running.
- Online Inactive (ONI): Client has VD connection and applications are running, but client is currently in idle mode (i.e., no VD packets are sent to the client).
- Offline (OFF): Client is not connected, but applications may be running.

Once the scheduling hypercall with SCHEDOP_service option is called, the scheduling class is updated for the corresponding DomU. The hypervisor stores the scheduling class value itself in the domain's meta data. If the scheduling class does not update for a long period of time (e.g., 10 seconds), it will degrade to a lower scheduling class, and the utility value of this VM will decrease. This situation occurs when no outbound VDI traffic leaves DomU.

Xen uses soft interrupt requests (IRQs) as a scheduling trigger. The soft IRQ is not interrupted by hardware, so it does not have a preset time period. When initializing, the scheduler registers the soft IRQ with the $schedule()$ function through $open\_softirq$. This can be adjusted to control the time quantum between scheduling calls. D-PriDe uses a quantum of 5 ms if there are any ONA or ONI priority VMs, and a quantum of 30 ms (the default of the Credit scheduler) otherwise.

IV. Utility Driven Priority Scheduling

A utility function enables fast calculation and reduces scheduling overhead. When the proposed scheduler is called, utility values for VMs are compared, and the VM with the largest utility value is returned to the scheduler. This section describes how we use a VM's priority and time share to determine its utility, and how the utility functions are used to make scheduling decisions.

A. *Time Share Definition*

Consider a hypervisor with a set of N VMs. The proposed algorithm schedules VMs according to their current utility values. Each VM has its own scheduling class, which is ONA, ONI, or OFF. Each VM $x \in N$ is assigned a time slot whenever the Xen hypervisor uses soft IRQ to trigger a scheduling event. The duration of time slots is not fixed because the time granularity of soft IRQ can range from tens of microseconds to tens or thousands of milliseconds. This irregularity makes hard real time scheduling difficult.

The scheduling algorithm selects a VM at time slot t. Based on its received time (CPU utilization) and its delay (time since last scheduling event), each VM is assigned a utility value. We define $tr_x(t)$ as a moving average of the received time assigned to a VM x at time slot t over the most recent time period of length t_0.

If VM x has been in the system for at least time period t_0,

$$tr_x(t) = tr_x(t-1) + \frac{s_x(t)h_x(t)}{t_0} - \frac{tr_x(t-1)}{t_0}, \qquad (1)$$

and if VM x enters the system at time u_x such that $(t-u_x) < t_0$,

$$tr_x(t) = \frac{\sum_{j=0}^{u_x} s_x(t-j)h_x(t-j)}{t_0}, \qquad (2)$$

where $s_x(t)$ is the time period from time slot $t-1$ to time slot t of VM x, and $h_x(t) = 1$ if VM x is scheduled from time slot $t-1$ to time slot t and $h_x(t) = 0$ otherwise. If VM x is scheduled at time t, $tr_x(t)$ increases. Otherwise, $tr_x(t)$ decreases by $\frac{tr_x(t-1)}{t_0}$. Intuitively, if $tr_x(t)$ increases, the utility value decreases and VM x will have fewer chances to be scheduled in subsequent time slots.

In addition, we consider the situation when a high priority VM x is scheduled consecutively for a long period of time. In order to maintain fairness to other VMs, VM x is not scheduled until $tr_x(t)$ decreases. Therefore, we need one more dimension to distribute the scheduling time evenly for VMs. We define the scheduling delay $td_x(t)$ as

$$td_x(t) = \frac{now(t) - p(x)}{t_0}, \qquad (3)$$

where $now(t)$ is the current scheduling time value at time slot t and $p(x)$ is the last scheduled time. $td_x(t)$ is employed in order to avoid the case when a VM receives enough CPU utilization at first, but is not later scheduled until the average utilization becomes small by Equation (1).

Together with Equations (1) and (3), we define the composite time unit (time share) containing CPU utilization $tr_x(t)$ and delay $td_x(t)$ as

$$t_x(t) = \frac{tr_x(t)}{td_x(t)+1}. \qquad (4)$$

$t_x(t)$ decreases if, during the time period t_0, the delay increases or the average utilization decreases. We use this equation in our definition of the utility function, as describing in Section IV-D.

B. CPU Allocation Policy

We now introduce policies that recognize different CPU allocation time-based types. These policies define rules that govern relationships between VM scheduling classes. Let $C(x)$ denote the scheduling class of VM x, as determined by our detection method described in Section III. Given two VMs x and y, $C(y) < C(x)$ means that scheduling class $C(x)$ has a higher preference value than $C(y)$. Note that VMs in the same scheduling class have the same guaranteed (or minimum) time period. Let $T(x)$ denote the guaranteed time share of VM x, and let $T(x) = T(y)$ when $C(x) = C(y)$. For any two VMs $x, y \in N$, we define the following policy rules:

- Policy Rule 1: In any time slot t, VM $x \in N$ with time share $t_x(t) < T(x)$ has a higher scheduling priority than any other VM $y \in N$ with scheduling class $C(y) < C(x)$. Hence, a VM y can be scheduled if and only if every VM x such that $C(y) < C(x)$ has time share $t_x(t) \geq T(x)$.
- Policy Rule 2: In any time slot t, VM $x \in N$ with $t_x(t) \geq T(x)$ has a lower scheduling priority than any other VM $y \in N$ with scheduling class $C(y) < C(x)$ and time share $t_y(t) < T(y)$. This means that once the utilization guarantees of all VMs in a particular scheduling class are satisfied, the scheduling priority shifts to VMs with lower scheduling classes.
- Policy Rule 3: In any time slot t, if all VMs meet their guaranteed time share, the remaining time must be distributed such that for any two VMs $x, y \in N$, the time ratio $t_x(t)/t_y(t) = \alpha_{C(x),C(y)}$, where $\alpha_{C(x),C(y)}$ is an arbitrary number given as a part of the policy rules.

C. Scheduling Algorithm

The scheduling algorithm is based on a marginal utility function that takes into account scheduling class. Given a VM x with scheduling class $C(x)$ and time share $t_x(t)$, let $f_{C(x)}(t_x(t))$ denote the marginal utility function.

In each time slot t, the scheduling algorithm selects and schedules VM x_i^* of CPU i such that

$$x_i^* = argmax_{x \in N_i}\{f_{C(x)}(t_x(t))\}, \tag{5}$$

where N_i is the set of VMs in CPU i. Accordingly, $h_x(t)$ is set to 1 for the selected VM and to 0 for all other VMs. $\forall x \in N_i$, time share $t_x(t)$ is updated according to Equation (4).

D. Marginal Utility Function

Suppose k different VM scheduling classes $C_1, ..., C_k$ such that the guaranteed minimum time share for VMs in scheduling class C_i is denoted by T_i, where $T_1 < ... < T_k$. The k scheduling classes are given a preference order that is independent from the minimum time share requirement. If $T_i < T_j$ for some i, j such that $1 \leq i, j \leq k$, C_i may have a higher preference than C_j.

Let C_i and C_j be arbitrary VM types with $T_i < T_j$. Assuming that C_j has a higher preference than C_i, we define

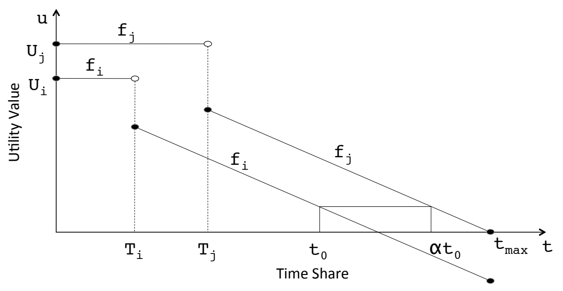


Fig. 4. Marginal Utility Function: $f_j = -t + t_{max}$; $f_i = -\alpha t + t_{max}$

marginal utility functions f_i and f_j for types C_i and C_j, respectively, as

$$f_j(t) = \begin{cases} U_j & \text{if } 0 \leq t < T_j \\ -t + t_{max} & \text{if } T_j \leq t \leq t_{max} \end{cases} \tag{6}$$

and

$$f_i(t) = \begin{cases} U_i & \text{if } 0 \leq t < T_i \\ -\alpha_{C_j,C_i} t + t_{max} & \text{if } T_i \leq t \leq t_{max} \end{cases} \tag{7}$$

where U_i and U_j are constants defined such that $U_j t_{min} > U_i t_{max}$ and $0 < t_{min} < t_{max}$. Policy Rule 1 is satisfied even if a VM in scheduling class C_j has a low time share. Similarly, $f_j(T_j)$ is defined with $U_i t_{min} > f_j(T_j) t_{max}$ in order to satisfy Policy Rule 2. Suppose the current utilization of a VM x in scheduling class C_i and that of a VM y in scheduling class C_j are t_0 and αt_0, respectively, where $\alpha = \alpha_{C_j,C_i}$. Then, $f_i(t_0) = f_j(\alpha t_0)$, as shown in Figure 4. α ratio can be easily extended to k different utility functions with k different scheduling classes. Hence, if the time shares are same for x and y, Policy Rule 3 will also be satisfied. When C_i has a higher preference than C_j, f_i and f_j can be similarly constructed with minor changes.

In practice, D-PriDe defines only three scheduling classes (ONA, ONI, and OFF), however, the utility function scheme described above could be used to support a much broader range of priority types. This could also be used to allow for differentiated priority levels within a scheduling class (i.e., multiple tiers within ONA), or to support a set of scheduling classes outside of the VDI domain.

E. CPU Migration

As the purpose of CPU migration is to balance loads among CPUs, D-PriDe tries to assign each CPU an equal number of VMs in each scheduling class. We do not take OFF VMs into consideration for CPU migration because OFF VMs are scheduled in a lower priority than ON (ONA and ONI) VMs. Once the scheduling algorithm selects a VM to schedule in the next time slot for a particular CPU, it will attempt to assign that VM to an idle CPU, if one exists. This process does not take long time because the D-PriDe scheduler keeps idle maps for the idle CPU currently as the Credit scheduler does. Migration proceeds in order of priority from low to high, thus minimizing cache trashing for higher priority VMs. This CPU migration scheme is similar to that of the Credit scheduler except the consideration of priority. The Credit scheduler migrates VMs when there is no OVER priority in a run-queue, whereas our migration scheme considers scheduling class order from lower scheduling class VMs to higher scheduling class VMs.

V. Experimental Evaluation

In this section, we analyze the D-PriDe scheduler's performance and overheads.

A. Experimental Setup

Hardware: Our experimental testbed consists of one server (2 cores, Intel 6700, 2.66 GHz, with 8GB memory and 8MB L1 cache) running Xen and one PC running VDI clients.

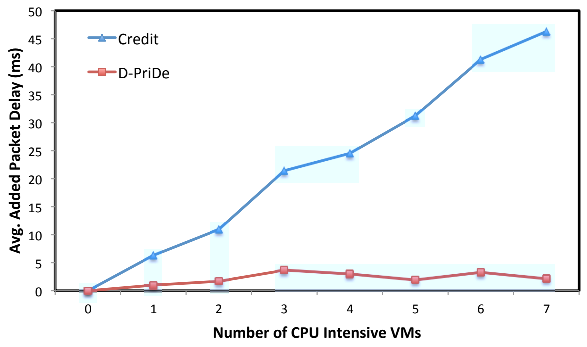
Xen and VM Setup: We use Xen version 4.1.2 with linux kernel version 3.1.1 for Dom0 and linux kernel version 3.0.9 for DomU. Xentop is used for measuring CPU utilization. We use a 5 ms quantum in all the experiments except Section V-F, where we experiment with other quanta.

VDI Environment Setup: We use tightVNC server (agent) with the JAVA-based tightVNC client (vncviewer). VDI clients, which connect to VM servers through vncviewer, are co-located in the same network with the server in order to prevent network packet delay. To measure packet inter-arrival time in a VDI client, we modify the packet receiving function $processNormalProtocol$ (located in the $VncCanvas$ class of vncviewer) by adding a simple statistics routine. We generate packets by playing a movie (23.97 frames per second and 640×480 video resolution) on the VDI agent. While the video is at 23.97 frames per second, in practice VNC delivers a slower rate because of how it recompresses and manages screen updates. A VM is called VD-VM when it is connected to a client and runs a video application, whereas a VM is called CPU-VM when it is or is not connected to a client and runs CPU intensive application such as a linux kernel compilation.

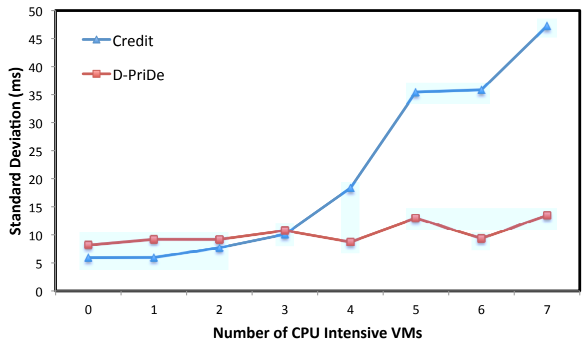
B. Credit vs. D-PriDe

We performed experiments for the existing Credit scheduling algorithm in a VDI setting, and found that packet inter-arrival time degraded when CPU-VMs ran in the background. Figure 5 and Figure 6 show the results when one VM runs a VDI agent connected to a VDI client and maximum seven CPU-VMs compile the linux kernel. We play a movie on the VD-VM in order to generate screen refreshments, so that the VDI agent on the VD-VM will send data to the VDI client. Watching a movie on the VDI client requires high QoS with respect to packet inter-arrival time. In order to measure packet inter-arrival time, we quantify the time difference between screen updates (a set of packets) from a client side.

When there are no interfering VMs, both schedulers see an average screen update interval time of 69ms (as shown for Credit in Figure 1). Figure 5(a) illustrates the average *additional* packet delay when CPU intensive VMs are added. For the Credit scheduler, as the number of CPU-VMs increases, the added packet inter-arrival time becomes large due to CPU interference. For the D-PriDe scheduler, however, the



(a) Added Packet Inter-Arrival Time



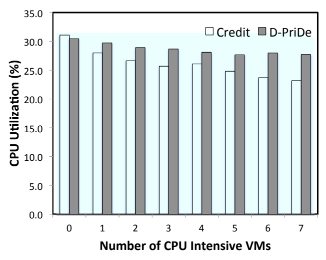
(b) Standard Deviation

Fig. 5. Packet Delay Comparison between the Credit scheduler and the D-PriDe scheduler for a VM playing a video (VD-VM) via VDI protocol, and maximum seven CPU intensive VMs running a linux kernel compile: (a) shows added packet delay defined as $added\_delay_i = packet\_delay_i - packet\_delay_0$ where i is the number of CPU-VMs; (b) describes a standard deviation for the packet delay.

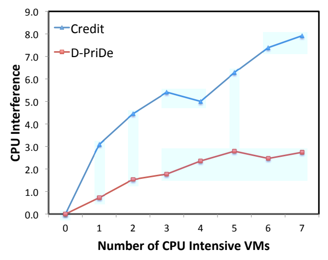
added packet inter-arrival time remains almost unchanged due to the priority-based scheduling. Figure 5(b) shows that the packet inter-arrival time fluctuation of the Credit scheduler becomes very high when many CPU intensive VMs run in the background, but the D-PriDe scheduler limits the standard deviation even though the number of CPU-VMs increases. In the worst case, the packet delay overhead of Credit is 66%, whereas the overhead of D-PriDe is less than 2%.

Figure 6(a) shows the CPU share given to the VD-VM. With no other VMs competing for a share, both schedulers allocate approximately 31% of one core's CPU time to the video streaming VM. When additional VMs are added, this share can decrease due to competition. However, when using a fair share scheduler we would not expect the VM to receive less than this base allocation until there are more than six VMs (i.e., our two CPU cores should be able to give six VMs equal shares of 33% each). In practice, imprecise fairness measures prevent this from happening, and the CPU dedicated to the VD-VM drops by over 7% when there are six or more VMs in the Credit scheduler as shown by Figure 6(b). The priority boost given to the VD-VM with D-PriDe prevents as much CPU time being lost by the VD-VM, with a drop of only 2.8% in the worst case.

Figure 7 shows that the cumulative density function of packet inter-arrival times in D-PriDe is more densely weighted



(a) CPU Utilization



(b) CPU Interference

Fig. 6. CPU Utilization Comparison between the Credit scheduler and the D-PriDe scheduler for a VM playing a video (VD-VM) via VDI protocol, and maximum seven CPU intensive VMs running a linux kernel compile: (a) and (b) illustrate CPU utilization and CPU interference of a VD-VM. CPU interference is defined as $cpu\_interference_i = cpu\_usage_i - cpu\_usage_0$.

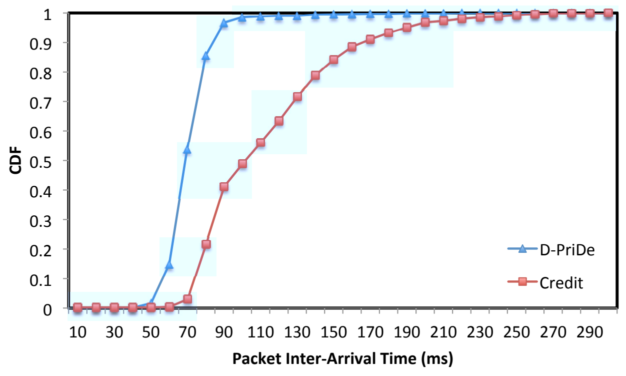


Fig. 7. Cumulative Density Function (CDF) for Packet Inter-Arrival Time from Credit and D-PriDe.

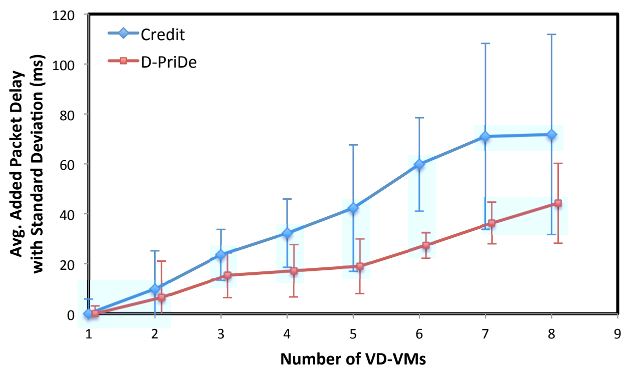


Fig. 8. Multiple VD-VMs: The figure shows the added packet delay defined as $added\_delay_i = packet\_delay_i - packet\_delay_0$ where i is the number of VD-VMs, with the standard deviation of the packet delay.

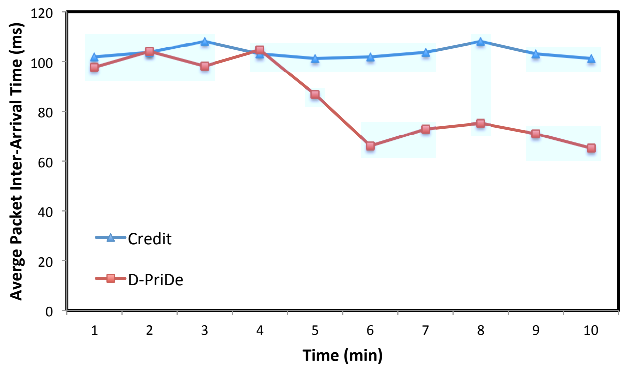


Fig. 9. Automatic Scheduling Class Detection in the D-PriDe scheduler improves the performance of an interactive user once competing VMs no longer have active client connections.

towards lower delays. The graph shows that 95% of screen update packets arrive within 90 ms for D-PriDe, whereas only 40% of packets arrive within 90 ms and takes as long as 190 ms to achieve 95% CDF with Credit. This guarantees the user experience when using D-PriDe is better than when using the credit scheduler.

C. Multiple VD-VMs

In this experiment, we run multiple VD-VMs simultaneously and show how competition between VD-VMs (of the same scheduling class) affects packet inter-arrival time and its standard deviation. The results of this experiment are shown in Figure 8. Since now all the VD-VMs are given the same high priority, we expect the packet delay to increase due to competition. However, the figure shows that D-PriDe still achieves better results than Credit. The primary reason is that D-PriDe uses a smaller quantum than Credit, which makes the scheduler respond quickly for the short sporadic requests. While D-PriDe cannot prevent competition between equivalently classed VMs, it still lowers the total overhead and keeps the deviation of the packet delay in a reasonably small cap.

D. Automatic Scheduling Class Detection

One characteristic of VDI setups is that users may have bursts of high interactivity followed by periods of idleness. The goal of D-PriDe is to automatically detect these events with help from the hypervisor, and adjust the priority of VD-VMs accordingly. To test D-PriDe's ability to detect and adjust scheduler classes, we consider an experiment where three VMs all initially have virtual desktop clients actively connected to them. The users of two of the VMs initiate CPU intensive tasks and then disconnect after a four minute startup period. The two CPU intensive VMs are assigned two VCPUs each so that they can saturate the CPU usage across all the cores, interfering with a video streaming task performed by the third VM. Figure 9 shows the average packet arrival rates for the third VM watching a video stream during the entire experiment. The two CPU intensive VMs get disconnected at 4 min for both Credit and D-PriDe schedulers. The Credit scheduler does not know anything about which users are performing interactive tasks, whereas the D-PriDe scheduler detects the scheduling class based on the user traffic so that it can adjust the priority of VMs. By minute 5, the two CPU VMs have been lowered from scheduler class ONA to ONI since no VDI packets have been detected by D-PriDe; they are further lowered to OFF after a timeout expires in minute 6 and the two VMs are considered low priority. This results in a decrease in packet inter-arrival times for D-PriDe, increasing the user perceived quality of service.

TABLE I
SCHEDULING OVERHEAD

Scheduler	**Average per call** (ns)	**Max** (ns)	**Min** (ns)	**Total** (μs)
D-PriDe	527	12057	32	1801
Credit	493	12082	64	874
SEDF	546	13201	56	645

E. Scheduling Overhead

We compare the scheduling overhead of the D-PriDe scheduler to the SEDF and Credit schedulers. We implement an overhead checker in $scheduler.c$, which reports the scheduler overhead (average time per call, maximum time, minimum time, and total scheduling time) through $xm\ dmesg$ every five seconds. Among eight VMs created, four VMs run VD services connected to a VDI client playing a video, and the other four VMs run a linux kernel compile in the background. Table I shows the overhead of the scheduling algorithms. Credit has the most efficient overhead time on average, but the average time difference between Credit and D-PriDe is 34 ns, which is negligible. Also, there is almost no difference in the maximum scheduling times of the Credit and D-PriDe schedulers. Since the time quantum of the D-PriDe scheduler is smaller than the Credit scheduler, the D-PriDe scheduler is called more frequently, resulting in greater total overhead. However, the absolute cost of scheduling remains small: in an average 5 second monitoring window only 0.036% of CPU time is spent on scheduling.

F. Quantum Effects

The Credit scheduler uses a coarse-grained scheduling quantum of 30 ms which does not perform well when VMs run applications requiring short, irregularly-spaced scheduling intervals (e.g., VD, voice, video, or gaming applications). In this experiment, we try a range of quanta in order to find a fine-grained quantum for the D-PriDe scheduler that yields good performance with respect to packet inter-arrival time and CPU utilization. All VMs are VD-VMs.

Figure 10 shows how scheduling quantum impacts packet delay from the clients perspective and CPU utilization on the server; in the best case we would like to minimize both, but lower time quantums typically improve client responsiveness at the expense of increased CPU overhead. We normalize the packet delay by the score achieved by the scheduler with a 30ms quantum (the default used by Credit), and normalize the total CPU utilization by the amount consumed with a very fine 1ms quantum. We run eight VD-VMs simultaneously with quantum times between 1 ms and 30 ms. The figure shows that average packet inter-arrival time increases when the quantum increases, whereas the CPU utilization decreases. The D-PriDe scheduler uses a time quantum of 5 ms, which provides a balance between packet inter-arrival time and CPU utilization. We have also tested the impact of the 5ms quantum when running CPU benchmarks inside competing VMs and found less than 2% overhead.

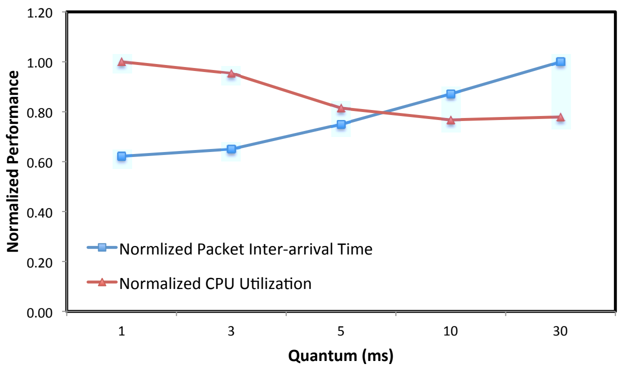


Fig. 10. Normalized Performance for Packet Inter-Arrival Time and CPU Utilization to show the best quantum to satisfy both criteria.

G. Discussion

Our results show that D-PriDe can significantly improve VDI performance and that our scheduling algorithm does not add significant overhead. When multiple background VMs are competing for resources with a VD-VM, D-PriDe lowers the interference impact from over 66% to less than 2% by using a finer grained time quantum than Credit and prioritizing the VM with an active desktop connection. When there are multiple VD-VMs running simultaneously, D-PriDe improves overall QoS by almost 40% and reduces the performance variability seen by clients. We have shown that the cost of making scheduling decisions in our system is comprable to other Xen schedulers, even though it provides a more powerful prioritization mechanism. Using a smaller scheduling time quantum reduces the additional packet delay seen by clients without incurring substantial overheads, and D-PriDe's ability to automatically detect when desktop users disconnect allows it to revert to a longer time quantum when high interactive performance is not required.

D-PriDe makes the assumption that virtual machines seeing workloads that involve frequent screen updates sent via VDI communication protocols are more important than other VMs. While we believe this assumption is valid for mixed environments hosting both virtual desktops and batch processing tasks, the framework provided by D-PriDe could be used in a variety of other situations as well. Utility functions provide a flexible way to assign priorities, and could be easily adapted for a situation such as running multiple scientific computing jobs with different priority levels. Similarly, D-PriDe's adaptation of scheduling parameters based on hypervisor observed behavior has many other uses. For example, the ruleset governing priority changes could instead be based on packet origin IP address, allowing a VM hosting a web application to automatically receive a priority boost whenever customers from a preferred network region arrive. We believe that resource management in the virtualization layer offers new approaches to QoS management that can be provided in a flexible, application agnostic way.

VI. Related Work

The deployment of soft real-time applications are hindered by virtualization components such as slow performance virtualization I/O [11, 12], lack of real-time scheduling, and shared-cache contention.

Certain scheduling algorithms [13, 14] use network traffic rates to make scheduling decisions. [13] modifies the SEDF scheduling algorithm in order to provide a communication-aware CPU scheduling algorithm to tackle high consolidation required circumstances, and conducts experiments on consolidated servers. [14] modifies the Credit scheduling algorithm by providing a task-aware virtual machine scheduling mechanism based on inference techniques, but this algorithm uses a large time quantum that is not conducive to interactive tasks. The network traffic rate approach in general is not suitable for VDI environments because high traffic rate does not directly imply high QoS demands.

Real-time fixed-priority scheduling algorithms [1, 3] are based on a hierarchical scheduling framework. RT-Xen [1] uses multiple priority queues that increase scheduling processing time by considering instantiation and empirical evaluation of a set of fixed-priority servers within a VMM. [3] proposes fixed priority inter-VM and reservation-based scheduling algorithms to reduce the response time by considering the schedulability of tasks. Instead of using SMP load balance, these algorithms dedicate each VM to a physical CPU. This approach can give better performance when a consistent level of CPU throughput is required, but results in degraded performance in a general VDI setting.

Soft real-time task scheduling algorithms [2, 4] have also been studied. [2] focuses on managing scheduling latency and controlling shared cache. This algorithm schedules VMs based on the laxity time in voice streaming applications, resulting in queue wait times of 2-5 ms and threshold delay of 2 ms. However, average scheduling delay of 2 ms is too high in a VDI setting, where delay is noticeable on the order of tens of microseconds when multiple virtual desktop applications are running. [4] assumes that VM types are set manually in advance, which is not possible in a dynamic VDI setting.

An inference technique-based scheduler has been proposed in [15]. The proposed scheduler is aware of task-level I/O-boundness using inference techniques, thereby improving I/O performance without compromising CPU fairness. The scheduler proposed in this paper aims to guarantee the fairness between VMs with the knowledge of task-level I/O-boundness, but the authors did not investigate interactive applications like VDI services.

VII. Conclusion

Virtualization and cloud computing promise to transform desktop computing by allowing large numbers of users to be consolidated onto a small number of machines. However, this goal cannot yet be achieved because most cloud hosting companies are not yet willing to schedule multiple VMs per CPU due to quality of service concerns; they prefer to buy additional server resources and to err on the side of caution. Our work tries to minimize VM interference in order to provide high-performing virtual desktop services even when the same machines are being used for computationally intensive processing tasks. D-PriDe's improved scheduling methods have the potential to increase revenue for hosting companies by improving resource utilization through server consolidation. We have shown that our scheduler reduces interference effects from 66% to less than 2% and that it can automatically detect changes in user priority by monitoring network behavior.

In the future, further tests of the proposed algorithm are needed in larger-scale systems (with more memory and a larger number of VMs) where hardware components such as cache and NUMA may impact experimental results.

References

[1] C. Lu S. Xi, J. Wilson and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," *EMSOFT*, 2011.

[2] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, "Supporting soft real-time tasks in the xen hypervisor," *VEE*, 2010.

[3] T. Cucinotta, G. Anastasi, and L. Abeni, "Respecting temporal constratins in virtualized services," *COMPSAC*, 2009.

[4] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: the xtratum approach," *EDCC*, 2010.

[5] 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," *ACM SOSP*, 2003.

[6] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *SIGMETRICS*, 2007.

[7] K. J. Duda and D. R. Cheriton, "Borrowed-virtual-time(bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler," *SOSP*, 1999.

[8] N. Nishiguchi, "Evaluation and consideration of the credit scheduler for client virtualization," *Xen Summit Asia*, 2008.

[9] S. Ma J. Wang and L. Liang, "Virtual desktop infrastructure problem statement," *IETF*, 2011.

[10] ITU-T T.120, "Data protocols for multimedia conferencing," *ITU-T*, 2007.

[11] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling i/o in virtual machine monitors," *VEE*, 2008.

[12] G. Liao, D. Guo, L. Bhuyan, and S. R. King, "Software techniques to improve virtualized io performance on multi-core systems," *ANCS*, 2008.

[13] S. Govindan, J. Choi, A. R Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co.: Communication-aware cpu management in consolidated xen-based hosting platforms," *VEE*, 2007.

[14] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware virtual machine scheduling for i/o performance," *VEE*, 2009.

[15] H. Kim, H. Lim, J. Jeong, H. Jo, J. Lee, and S. Maeng, "Transparently bridging semantic gap in cpu management for virtualized environments," *ELSEVIER*, 2009.