User-driven Scheduling Of Interactive Virtual Machines

Bin Lin Peter A. Dinda Dong Lu
{binlin, pdinda, donglu}@cs.northwestern.edu
Department of Computer Science, Northwestern University

***Abstract*— We are developing a distributed computing system, Virtuoso, which presents virtual machines (VMs) as its fundamental abstraction to end users. Long-running noninteractive VMs may coexist on the same host used to support VMs being used by highly interactive users. We must simultaneously provide high average computation rates to the non-interactive VMs while keeping the users of the interactive VMs happy. We report here an initial work on using direct user feedback to achieve this balance. The user is provided with a (physical or logical) button that can be pressed when he feels his machine is responding inadequately. In response, the scheduler boosts the priority of his VMs relative to the other VMs in the system. The priority then declines with time. The goal of the control algorithm driven by this mechanism is to maintain a targeted average time between button presses while simultaneously delivering a high compute rate to the other VMs.**

I. Introduction

Virtual machines (VMs) can dramatically simplify the use of distributed resources to run diverse applications with high performance because they provide a low level of abstraction that is conducive both to providers of resources and their users. VMs allow dynamic multiplexing of users onto physical resources at the granularity of a single user per operating system session, thereby supporting per-user VM configuration and isolation from other users sharing the same physical resource. For distributed computing, VMS have advantages in software security, customization, resource control, site-independence and many other aspects. We have described the complete case for wide-area distributed computing using virtual machines in a previous paper [5]. We are now building a system, Virtuoso, for distributed computing on VMs that has the following model:

- The user receives what appears to be a new computer or computers on his network at very low cost. The user can install, use, and customize the operating system, environment, and applications with full administrative control.
- The user chooses where to execute the virtual machines. Checkpointing and migration is handled efficiently through Virtuoso. The user can delegate these decisions to the system.
- A service provider need only install the VM management software to support a diverse set of users and applications.
- Monitoring, adaptation, resource reservation, and other services are retrofitted to existing applications at the VM level with no modification of the application code, resulting in broad application of these technologies with minimal application programmer involvement.

We currently use VMWare GSX Server [19] as our virtual machine monitor (VMM). Our interface is web-based and emulates the site of a hardware vendor such as Dell or IBM [15]. An important element of our system is layer 2 virtual network, VNET, which creates the "networking illusion" needed for the first element of the model and which we are now extending to be the basis of the fourth element of the model [18], [7]. Connectivity to a VM's console is provided using a VNC applet embedded in a web page. From there, the user can bootstrap to direct connectivity using any protocol supported by the OS (SSH [16], X [21], and Windows Remote Desktop [13], for example).

A resource provider can have a host participate in Virtuoso by registering with the front end of the system and running the Virtuoso software on the host. The front end can then start new VMs on or migrate existing VMs to the host.

While a VM can support a very wide range of applications, we particularly want to be able to gracefully handle long-running non-interactive applications, such as scientific simulations, parallel programs, and grid computing applications, as well as interactive applications, such as desktop productivity applications and games. VMs running noninteractive applications and VMs running interactive applications may have to coexist on the same host machine. *How can we schedule or control the interactive VMs so that their users remain happy while not over-penalizing the noninteractive VMs?*

In ongoing work [9], we have been studying the tolerance of the interactive user for contention for CPU, memory, and disk resources. At the present time, we have completed a controlled user study on this topic, and are running an Internet-wide user study that is open to all.[1] For these studies, we have developed the Understanding User Comfort System (UUCS), a Windows client/server system that executes carefully controlled resource borrowing according to a profile and accepts user feedback as it is doing so. The feedback mechanism is a physical or

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, EIA-0130869, and EIA-0224449, and by a gift from VMWare, Inc. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

[1]Consult http://comfort.cs.northwestern.edu for more information.

logical button that the user presses to express discomfort. In response, the system immediately halts its borrowing and waits for a controlled period of time before trying a new profile. One of the purposes for characterizing user tolerance to resource borrowing is to inform the scheduling of interactive and non-interactive VMs.

User comfort with resource borrowing is highly dependent on the applications being used. Because of this complexity, determining the exact single scheduling goal for an interactive VM in a particular context is challenging. In response, we are exploring using direct user feedback in the scheduling process. This paper presents our initial results in applying direct user feedback, specifically the use of a "discomfort button" similar to that used in our user comfort work, to control the CPU use of interactive VMs.

The basic idea is to modulate the Linux "nice" level of an interactive VM's VMM in response to the passing of time and the user's button presses. VMWare GSX is a "type II" VMM [6], meaning that it executes as a process on an underlying operating system, Linux in our case. As such, we can control, to some extent, the priority of all the processes and the OS (Windows XP here) running in the VM by manipulating the nice level.

There is a tension between user participation and the average compute rate of non-interactive VMs; the more frequently we expect the user to press the button, the faster the non-interactive VMs can operate. We propose to let the administrator resolve this tension by setting a target mean interarrival time between button presses. One example of how an administrator might set the target interarrival time is in response to the cost of running the VM with more costly VMs having a longer interarrival time target, and the interactive VM user paying according to the interarrival time he is assigned. The more the user pays, the better interactivity he can get. The control system's goal is to manipulate the interactive VM's nice level to maintain this set interarrival time with as little variance as possible.

We explore three control algorithms, all of which are in their infancy. The first two are straightforward translations of the TCP Reno congestion control algorithm. For us a congestion event is the user button press and the equivalent to the congestion window is the linearized nice level of the interactive VM. Acknowledgments are replaced with the simple passage of time. After a button press, the priority of the interactive VM is maximized. It then declines exponentially in a slow start-like phase and eventually linearly increases at a rate r. Other VMs have the potential to run increasingly faster during this process. At some point, through the combination of the low priority of the interactive VM, the workload inside it, and the workload on the other VMs, the user becomes discomforted and the process repeats.

The third algorithm is also similar to TCP Reno, but here we also control the rate of linear increase from button press to button press, making r a function of time, with the goal of achieving the target interarrival time between button presses.

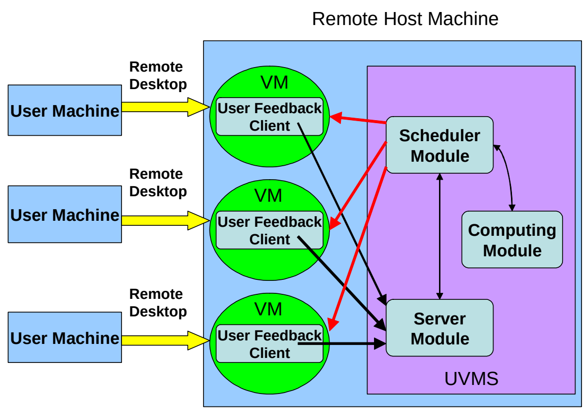

Fig. 1. Structure of UVMS.

There has been some related work on controlling CPU allocation by adjusting priorities and scheduler parameters [3], [10] or by the nice mechanism [14], but none has used direct user feedback to facilitate the scheduling of both interactive and non-interactive programs.

In the following, we begin by explaining the details of our framework, which we refer to as the User-driven Virtual Machine Scheduling or UVMS. We describe the extent of control we have using the nice mechanism, how we linearize this mechanism to simplify the remainder of the system, and how our three control algorithms work. Next, we describe very early performance results using the system. Although these results are largely anecdotal at this point, they are quite interesting. Using user feedback to schedule the interactive VMs, the batch machines can make much faster progress compared with simply running the batch machines at a low priority. However, it is unclear whether the user feedback rate is sufficient to control the interarrival time. In the absence of such control, the system may too frustrating for the end user.

II. Mechanism

We have prototyped a mechanism to explore the user-driven scheduling of interactive VMs. This section describes our experimental framework, the control mechanism we use, and three algorithms that react to user feedback using the control mechanism.

A. *Overall framework*

User-driven Virtual Machine Scheduling, as shown in Figure 1, involves a remote host machine and local client machines. The remote host machine, one of the machines in our cluster, runs VMs. We install Windows XP Professional in a VM created using VMWare GSX Server 2.5. VMWare is configured to use bridged networking, which makes the Windows VM appear as a new and independent machine in the cluster. To access the VM from local machines, we enable the Remote Desktop service of Windows XP professional (also known as Terminal Services.)

The local client machine can be any machine away from the cluster. We use a remote desktop client to connect to the

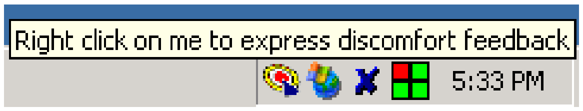

Fig. 2. Client interface.

remote Windows VM. Note that there are various methods to connect to a remote VM's display, such as X11 and VNC. The reason why we use remote desktop is that it can achieve better interactivity compared with other methods. Note that multiple VMs can be hosted simultaneously in the same remote host machine. In the current state of this research, we only study the case of a single VM. Underneath VMWare, the host machine runs Red Hat Linux 9.0 with a uniprocessor kernel to simplify the analysis. The host machine is an IBM x335 (2 GHz Xeon, 1.5 GB RAM).

Our UVMS control system consists of a client that runs in the VM and a scheduler that runs on top of Linux in the host machine. We modified the UUCS client, developed in the Understanding User Comfort Project [9], [8], to be our UVMS client. The UVMS client can monitor user's activities and capture user discomfort feedback. Figure 2 shows the most basic graphical interface of the UVMS client as it appears in the toolbar of the Windows VM. A user can express discomfort, either by clicking on the tray icon or by simply pressing a hot-key (e.g. F11).

UVMS runs under Linux on the host machine, side by side with the VMs. It consists of three modules:

- Server Module
- Priority Scheduler Module
- Computing Module

The client synchronizes with the server module whenever it starts running, ends, or captures user discomfort feedback.

The priority scheduler module is responsible for applying control algorithms to set the priority of the VMs. It also records realtime information about the VM process and the scheduler itself, including process id, priority, running time, and so on. The data we collect is stored in text-based form.

To simulate the non-interactive VMs competing for CPU in the system, the computing module launches and monitors a computing process which keeps running a unit busy loop (a loop which finishes computing in a certain unit of time, e.g. 0.01 seconds). We measure the amount of computation as the number of unit busy loops finished.

B. Control mechanism

By default, processes in Linux are scheduled using the SCHED_OTHER policy. In this policy, each process has a dynamic priority level. Linux ranks the processes based on their priorities and executes the highest priority process. Processes have an initial static priority that is called the nice value. This value ranges from -20 to 19 with a default of zero. The smaller the nice value, the higher the priority. The dynamic priority is calculated as a function of the nice value and the task's interactivity (e.g. sleeping time of a process versus time it spends in runnable state) [12]. The user influences the system's scheduling decisions by changing the process's nice value [11].

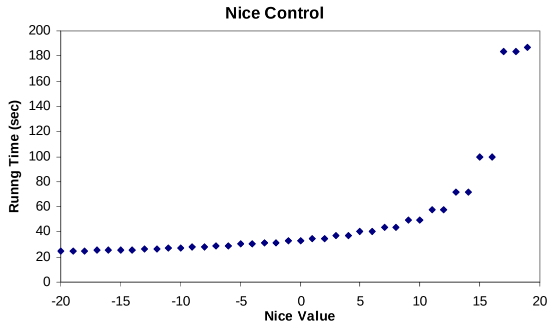

Fig. 3. Nice control experiment 1.

We control the process of the VM running in the remote host machine (the core process used by the VMware virtual machine monitor (VMM) for the execution path within the VM). The UVMS scheduler monitors the execution of the VM process and adjusts its priority at run time through the nice mechanism, based on user feedback. As a first step, we did two experiments to validate the control capability of our scheduler.

Experiment 1: In this experiment, we measure the running time of a busy loop program in the VM that is similar to the computing module, with different nice values of the VM process. Note that we run the competing computing module process with a nice value of 0 to compete with the VM process for CPU. We sweep the nice value of the VM process from -20 to 19. For each value, we run the program 15 times and calculate the average. As shown in Figure 3, with nice value -20, the average running time is 24.32 seconds. As we increase the nice value from -20 to 19, the running time gradually increases to about 50 seconds and then abruptly steps to 185.99 seconds.

We repeat the experiment three times with different unit running time of the computing module. We find the same trend in all graphs. This experiment shows that by controlling the nice value of the VM process, we can influence the performance of the programs inside the VM by a factor of around 8.

Note that, surprisingly, the nice control mechanism does not behave linearly.

Experiment 2: In this experiment, we study the sensitivity of programs inside the VM to the nice value of VM process. We increase the nice value of the VM process from -20 to 19 at set intervals. We increase the nice value by 1 every 5 seconds. At the same time, we record, at a fine granularity, the running time of a unit computation inside the VM. From Figure 4, we can see that with nice value -20, the running time of the unit computation is 5 microseconds. The running time increases smoothly as the nice value of the VM increases. As before, we are running a competing compute module process with nice value equal to 0 in the host machine to compete with the VM process for CPU.

We repeat the experiment three times, with different unit

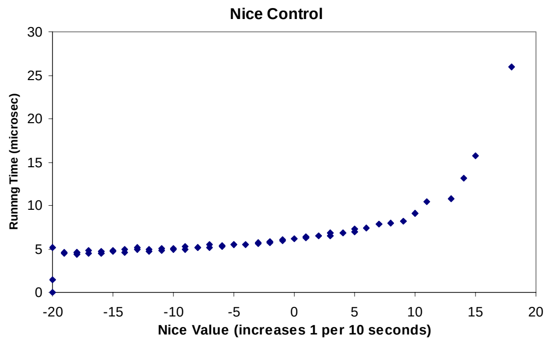

Fig. 4. Nice control experiment 2.

computations in the computing program and with different intervals of increasing priority. We find the same trend in all graphs.

C. Control model

How does a CPU intensive process's performance change with its nice value and nice values of other competing processes? In this section, we set up a simple yet effective model to describe a CPU intensive process's performance as a function of its nice value and other competing processes' nice values.

We assume that the process's nice value ranges between -20 and 19, as is reported in the man page that comes with current Linux kernel 2.4.20-8. Note that the possible difference of the nice value range between platforms won't effect our model as long as we know the range on a specific platform. For modeling convenience, we map the nice value from [-20, 19] to [1, 40]. This mapping can be converted back easily. In the paper, we call the mapped nice value *normalized nice value*.

Let P_x be the percentage of CPU dedicated to process x, and T_{fx} be the execution time of process x given $P_x = 1$. Then we have

$$T_x = \frac{T_{fx}}{P_x} \tag{1}$$

where T_x is the execution time of process x. Equation 1 holds for any CPU intensive applications with deterministic compute cycle requirements.

Assume there are m CPU intensive processes running in the system, each with nice value n_1, n_2, n_3 ... n_m. Then P_x can be modeled using

$$P_x = \frac{40 - n_x}{\sum_{i=1}^{m}(40 - n_i)} \tag{2}$$

where x is between 1 and m. We call $40 - n_x$ the *complementary nice value* of process x. Equation 2 means that the percentage of CPU cycles that process x can achieve equals the ratio of its complementary nice value over the sum of all CPU intensive process's complementary nice values.

Combining Equation 1 and Equation 2, we derive

$$T_x = \frac{T_{fx} \times \sum_{i=1}^{m}(40 - n_i)}{40 - n_x} \tag{3}$$

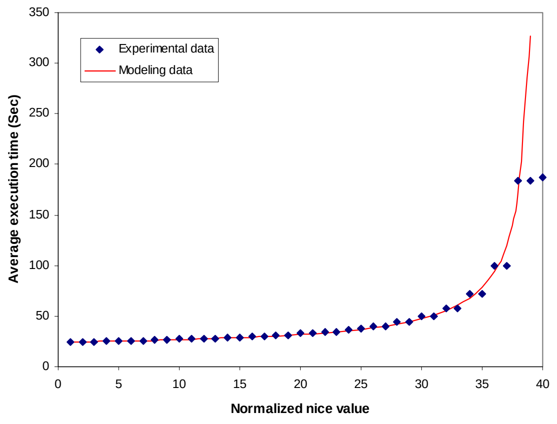

Fig. 5. Model evaluation experiment I.

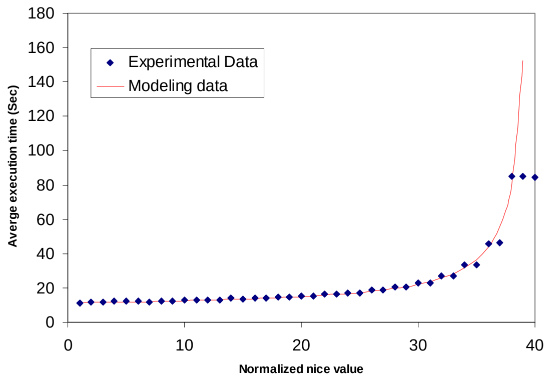

Fig. 6. Model evaluation experiment II.

We did experiments to validate our model. Figures 5 and 6 show two examples of the experimental data versus predicted data given by the model. The experiment data is from experiment 1 as we discussed in Section II-B. Clearly, we can see the model produces satisfactory prediction results until the normalized nice value exceeds 38, where the experimental data flattens out while our model shoots much higher.

D. Control algorithm

We seek a scheduling algorithm that balances the comfort of interactive VM users and the progress of non-interactive VMs. In other words, we want to maximize the CPU usage of the non-interactive VMs, modeled in our framework with the compute module, subject to a constraint on the discomfort of the the interactive VM users. Our innovation is to have an interactive VM user directly report his discomfort.

We borrow a simple but well-known algorithm as our starting point. The TCP congestion control algorithm [17], [20], [1], [4] is designed to adapt the size of the congestion window (and hence the send rate) dynamically to the available bandwidth in the path. For us a congestion event is the user button press and the equivalent to the congestion window is the nice value of the VM. Acknowledgments are replaced with the

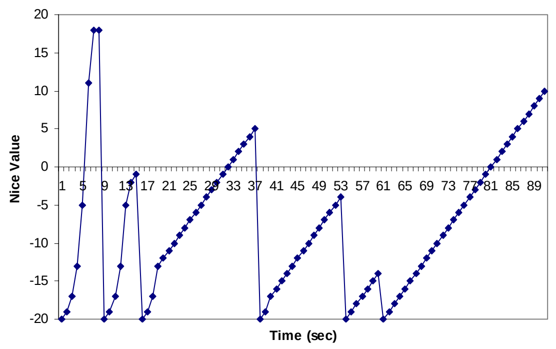

Fig. 7. TCP control algorithm.

simple passage of time. In effect, the priority of the interactive VM starts out at maximum, and then declines with the passage of time until the user presses the button, at which point the priority is restored to the maximum and the cycle repeats.

Each of our algorithms has two state variables

- Current control value, r (this is the nice level of the VM)
- Threshold, r_t

and three major components:

- Slow Start
- Additive-Increase, Multiplicative-Decrease
- Reaction to the user feedback

We begin with an algorithm with two control parameters:

- Slow Start speed, α
- Additive-Increase speed, β.

Slow Start: If $r < r_t$, we increase r exponentially fast with time (e.g. 2^{α}), assuming that the performance of the VM is less likely to be affected under low nice values (i.e. high priorities).

Additive-Increase, Multiplicative-Decrease: If no user feedback is received and $r \geq r_t$, we increase r linearly with time, $r \propto \beta t$.

Reaction to the user feedback: When the user expresses his discomfort at level r we immediately set $r_t = r/2$, and set r to the initial (lowest) priority.

Figure 7 illustrates the execution of the algorithm. On top of this general TCP Reno lookalike, we implemented three extended control algorithms based on nonlinear and linear control schemes. Experimental results of our algorithms will be discussed in Section III.

1) Nonlinear control scheme: By nonlinear, we mean that changing a function input does not proportionally change the output. As discussed in Section II-B, by directly manipulating the nice value (40 levels from -20 to 19) of the VM process, we can nonlinearly influence the performance of the programs inside the VM by factor of around 8. Based on this scheme, we apply the general TCP control algorithm directly, using r as the nice level, and as a result, we get our nonlinear control algorithm.

2) Linear control scheme: As discussed in Section II-C, Equation 3 models the impact of the nice value on the compute

Control value	Nice value	Experimental data	Normalized model value
1	-20	24.36	1.00
2	-7	29.14	1.25
3	-1	32.98	1.48
4	3	37.08	1.72
5	5	40.00	1.89
6	7	43.94	2.13
7	9	49.51	2.45
9	11	57.71	2.94
11	13	71.75	3.75
16	15	99.56	5.38
28	17	183.61	10.25

Fig. 8. Linearization.

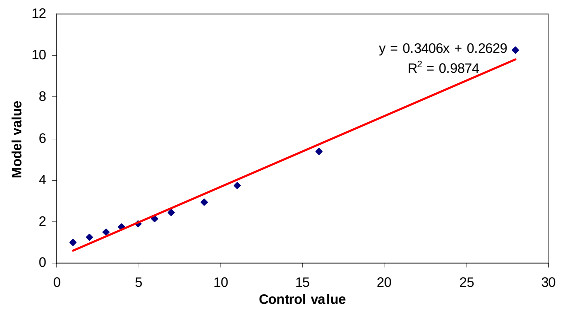

Fig. 9. Linearization.

rate of a process. Using this model, we can linearize our control mechanism, as shown in Figure 8, by these means:

- From Figure 3, we can see that the initial 2/3 of the curve is almost linear. We divide this part as evenly as possible into 7, assigning control values 1 to 7. These values ensure distinguishable differences in the running time.
- For the tail of the curve, we can see that the running time increases very quickly with even small changes in the nice values. To preserve the smallest granularity of the nice value changes, we divide this part into 4 and assign to it discontinuous control values 9, 11, 16 and 28.
- The time intervals between control values being changed reflect the differences between the corresponding control values.

We use discontinuous control values here to show that we adapt the time intervals of applying control values to the growing Y axis difference between two consecutive control values.

Figure 9 shows that with the above linearization to 11 control values and the adaptation of intervals, we can achieve very good linear control. And we did experiments to evaluate the TCP control algorithm based on this linear control scheme.

The linear control scheme has the same control parameters as the nonlinear scheme. The r value of the nonlinear scheme is transformed via the mapping derived above before it is applied.

3) Adaptive control scheme: Through experiments, we observe that the Additive-Increase and Multiplicative-Decrease phase in our TCP control algorithm is most often dominated by the linear increase, while the interarrival time between button presses is quite varied. Recall that we would ideally want the user to only have to press the discomfort button at relatively periodic and deterministic points. We extended our TCP control algorithm to better adapt to user feedback, to control not only the impact of background processes, but also the degree of user attention necessary.

In the adaptive algorithm, certain control parameters become state variables:

- Rate of increase, ρ
- Slow Start speed, $\alpha = f(\rho)$
- Additive-Increase speed, $\beta = g(\rho)$

Adaptive reaction to the user feedback: The rate of increase ρ controls the rate of exponential and linear increase from user feedback to user feedback. In addition to our original TCP control algorithm, we introduce an adaptive function to control the rate of increase:

$$\rho_{i+1} = \rho_i \left(1 + \gamma \times \frac{T_i - T_{AVI}}{T_{AVI}}\right) \tag{4}$$

Here ρ_i is the rate of increase. T_i is the latest interarrival time between user feedback. T_{AVI} is the target mean interarrival time between user feedback, expected by the user or set by the administrator. γ controls the sensitivity to the feedback. We applied this adaptive algorithm to the linearized control scheme.

III. Experiments

Using the UVMS, we addressed the questions posed in the introduction, and we compared the various control algorithms described in the previous section. At present, we have studied only a single user, so our results are preliminary, but interesting and promising.

A. Experimental setup

One of the authors (Lin) volunteered to be our guinea pig. He used his own desktop in his office to connect to the remote Windows VM using the Windows Remote Desktop client in full screen mode. The user used this VM as his desktop during the day. The only difference from his physical desktop was the existence of the user feedback button. The UVMS client spoke to the UVMS scheduler as described earlier. UVMS recorded the system information as well as the user feedback, which was later used to generate the results.

We did three experiments corresponding to the three control algorithms we discussed in Section II-D. The duration of each experiment was approximately 1 hour.

The user's activities included typical tasks he performs daily, for example:

- Browsing with Internet Explorer
- Word processing using Microsoft Word
- Presentation preparation using Microsoft Powerpoint
- Listening to MP3 music using Microsoft Media Player
- Playing games like DemonStar [2] (a 2D shooting game)

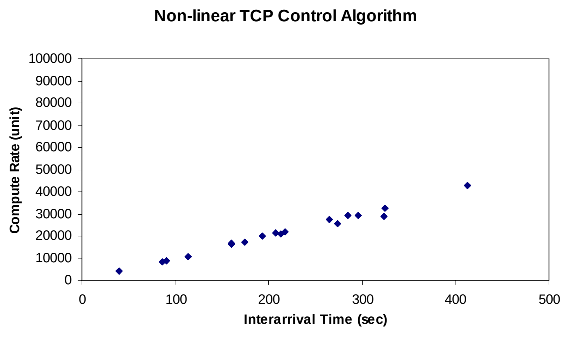

Fig. 10. Experiment I: nonlinear control scheme.

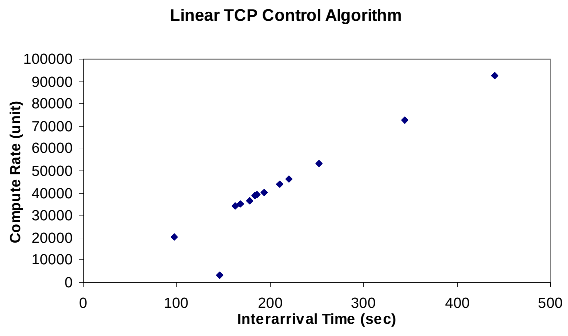

Fig. 11. Experiment II: linear control scheme.

B. Metrics

Recall that our goal is to simultaneously provide high average computation rates to the non-interactive VMs while keeping the users of the interactive VMs happy. We use two metrics to evaluate the algorithms.

The *interarrival time between user feedbacks* is the interval between two consecutive user feedbacks. This measurement helps us understand how the user feels (e.g. comfort, happiness) when interacting with the VM. Ideally, the user would prefer that such feedbacks are far between on average with very low jitter. We will consider both the average interarrival time and its standard deviation.

The *compute rate* is the rate of computation of the competing process launched by the computing module of UVMS. This metric represents the computation done by other VMs and non-VM processes running in the same host machine. As we mentioned before, we measure the amount of computation as the number of unit busy loops finished. We would like this rate to be as high as possible.

C. Experimental results and analysis

We show results for our three control algorithms.

1) Experiment I: nonlinear control scheme.: Figure 10 shows the relationship between the compute rate of the non-interactive process and the interarrival times of the user feedbacks. Each data point in the figure represents the interarrival time between two consecutive and the amount of computation accomplished in the interval.

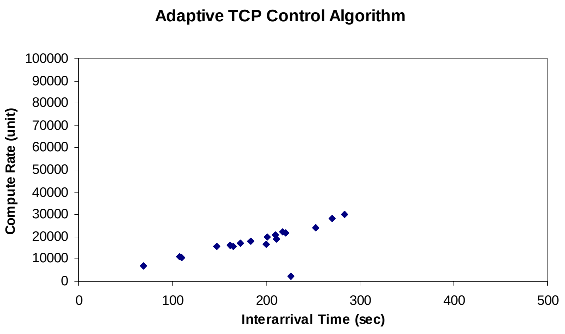

Fig. 12. Experiment III: adaptive control scheme.

The average interarrival time between user feedback is 213.17 seconds, with a standard deviation of 97.37 seconds. The average nice value at which user feels discomfort is 12.12, with a standard deviation of 6.89. The average compute rate is 21223.61 units, with a standard deviation of 9804.84 units.

2) Experiment II: linear control scheme.: Figure 11 shows the relationship between compute rate and interarrival time. The average interarrival time between user feedbacks is 213.89 seconds, with a standard deviation of 89.28 seconds. The average nice value at which user feels discomfort is 14.67, with a standard deviation of 2.06. The average compute rate is 42824.62 units per interarrival, with a standard deviation of 21981.35 units.

Does the linear control scheme *really* work better than the nonlinear control scheme? Although both experiments have almost the same average interarrival time between user feedbacks with similar standard deviations, the user felt more comfortable in Experiment I. More specifically, in Experiment I, the user felt that the machine always slowed down gradually, while in Experiment II, he often felt discomforted by the abrupt slowdown of the machine. One possible reason for this may be that in the additive-increase phase of Experiment I, we increase the nice value by 1 in certain interval (e.g. 10 seconds) using nonlinear control of 40 levels of nice values, while we use discontinuous control values with limited levels in the other two experiments.

Observe that in both Experiment I and II, the standard deviation of the interarrival time is very large. While it is clear that the system was able to react quickly to provide better performance to the user, the points in time at which the user had to express discomfort showed a lot of jitter, depending mostly on what applications he was using. In other words, it was difficult for the button pressing to become a habit.

Based on this observation, we did experiment III, testing the adaptive TCP control algorithm, which explicitly tries to reduce the variance of the discomfort interarrival error (i.e., the selected interarrival time minus the actual discomfort interarrival time of the user).

3) Experiment III: adaptive control scheme: Figure 12 shows the relationship with compute rate and time interval. The target interarrival time (T_{AVI}) is set to 240 seconds (based on the user experience in Experiment I and II). The control parameter γ is set to 0.5, which makes the algorithm very sensitive to user feedback.

The average interarrival time between user feedback is now 189.44 seconds, with a standard deviation of 56.60 seconds. The average nice value at which user feels discomfort is 10.28, with a standard deviation of 2.78. The average compute rate is 17610.56, with a standard deviation of 6980.43.

As we can see, Experiment III achieved a much lower deviation of interarrival time, compared with Experiment I and II. The user felt discomforted at more predictable points in time, as we hoped for.

What compute rate we can deliver to other VMs and non-VM processes? In all three experiments, we can see that the larger the interarrival time is, the higher the compute rate is. The reason is the larger the interarrival time of user feedback is, the higher the average nice value (lower priority) the interactive VM process has, and the more CPU time other processes will get. However, a large interarrival time also means that user can tolerate the slowdown of the machine for a long time without being discomforted.

To further study how high a compute rates we can achieve by using UVMS, we calculated the accumulated compute cycles in Experiment I through III. We also ran two more experiments to calculate the accumulated compute rate of the computing process when competing with VM process without UVMS scheduler running. The nice value of the VM process was -20 in one experiment and -1 in another one. Note that in all the experiments we did, the nice value of the computing process was always -20 (highest priority).

Figure 13 summarizes our experimental results. In Experiment II, when user had to tolerate being discomforted at any time (highly variable interarrival time), we can deliver the highest compute rate to other processes. The compute rate is almost 5 times the compute rate without UVMS scheduler running. In Experiment III, when user felt relatively more comfortable by expressing feedback in fixed intervals, the compute rate we could deliver is a little bit lower while it is still 3 times the compute rate without the UVMS scheduler running.

IV. Observations

Figure 13 clearly illustrates that it is possible to deliver higher compute rates to non-interactive VMs when interactive VMs are scheduled with the use of direct user feedback. However, is this technique likely to to be practical? This rests on essentially two questions: will an interactive user tolerate providing the feedback, and can the adaptive control algorithm work for long interarrival times?

In our motivating scenario, the user indicates he is willing to press the button periodically. The more he pays, the longer the expected interval between button presses. Unfortunately, there appears to be a tension here. With a long expected interval, the adaptive algorithm gets very little measurement input, making it harder for any algorithm to do well. Conversely, the algorithm is likely to do better with short intervals, but these are lower paying customers.

	Accumulated compute rate	Average compute rate	Std. Dev. compute rate	Average interarrival	Std. Dev. interarrival
Experiment I (nonlinear scheme)	382025	21223.61	9804.84	213.17	97.37
Experiment II (linear scheme)	556720	42824.62	21981.35	213.89	89.28
Experiment III (adaptive scheme)	316990	17610.56	6980.43	189.44	56.60
VM nice value: -20	116987	N/A	N/A	N/A	N/A
VM nice value: -1	133650	N/A	N/A	N/A	N/A

Fig. 13. Comparison of compute rates and interarrival times.

Although we believe it is critical for a user to be able to asynchronously indicate to the system that he is displeased, we increasingly believe that having a single bit of information from the user is insufficient. We are now exploring how to support interactivity through real-time mechanisms, letting the user interrupt the system at any time to request a new schedule for any of his VMs.

V. Conclusions and future work

We have described the initial design of a scheduling system UVMS, that uses direct user feedback to balance between providing high average computation rates to the non-interactive VMs while keeping the users of the interactive VMs happy. We showed the extent of control we have using the nice mechanism, how we linearize this mechanism to simplify the remainder of the system, the design of control algorithms and how our three control algorithms work. We also described very early experimental results using the system.

Although our results are largely anecdotal at this point, they are promising. Using feedback it is possible to provide interactive performance while noninteractive VMs progress much faster than would otherwise be possible. However, it appears that more information is needed from the user, perhaps in the form of a real-time schedule.

References

[1] Brakmo, L. S., O'Malley, S. W., and Peterson, L. L. Tcp vegas: new techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications* (1994), pp. 24–35.

[2] Demon Star. Demonstar. http://www.demonstar.co.uk/.

[3] Epema, D. H. J. Decay-usage scheduling in multiprocessors. *ACM Trans. Comput. Syst. 16*, 4 (1998), 367–415.

[4] Fall, K., and Floyd, S. Simulation-based comparisons of tahoe, reno and sack tcp. *SIGCOMM Comput. Commun. Rev. 26*, 3 (1996), 5–21.

[5] Figueiredo, R., Dinda, P. A., and Fortes, J. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)* (May 2003).

[6] Goldberg, R. Survey of virtual machine research. In *Survey of Virtual Machine Research, IEEE Computer, pp 34-45* (June 1974).

[7] Gupta, A., and Dinda, P. A. Inferring the topology and traffic load of parallel programs running in a virtual machine environment. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing (JSPPS 2004* (June 2004). To Appear.

[8] Gupta, A., Lin, B., and Dinda, P. A system for studying user comfort with resource borrowing. Tech. Rep. NWU-CS-04-28, Department of Computer Science, Northwestern University, 2003.

[9] Gupta, A., Lin, B., and Dinda, P. Measuring and understanding user comfort with resource borrowing. In *Proceedings of the 13th IEEE International Symposium on High-Performance Distributed Computing* (June 2004).

[10] Hellerstein, J. L. Achieving service rate objectives with decay usage scheduling. *IEEE Transactions on Software Engineering 19*, 8 (August 1993), 813–825.

[11] Linux. Linux man pages : nice.

[12] Love, R. *Linux Kernel Development*. SAMS, Developer Library Series, Sept 2003.

[13] Microsoft Corporation. Remote desktop connection software. http://www.microsoft.com.

[14] Moruzzi, C., and Rose, G. Watson share scheduler. *Proc. Large Installation Systems Administrator Conf.* (1991), 129–133.

[15] Shoykhet, A., Lange, J., and Dinda, P. Virtuoso: A system for virtual machine marketplaces. Tech. Rep. NWU-CS-04-39, Department of Computer Science, Northwestern University, July 2004.

[16] SSH Communications Security. Ssh secure shell. www.ssh.com.

[17] Stevens, W. Tcp slow start, congestion avoidance, fast retransmit and fast recovery algorithms, 1997.

[18] Sundararaj, A., and Dinda, P. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM 2004)* (May 2004). Earlier version available as Technical Report NWU-CS-03-27, Department of Computer Science, Northwestern University.

[19] VMWare Corporation. Vmware gsx server. http://www.vmware.com/.

[20] Wang, Z., and Crowcroft, J. Eliminating periodic packet losses in the 4.3-tahoe bsd tcp congestion control algorithm, 1992.

[21] X.org. X windows. http://www.x.org/.