An Improved Xen Credit Scheduler for I/O Latency-Sensitive Applications on Multicores

Lingfang Zeng [♯], Yang Wang [♭], Wei Shi [§], Dan Feng [♯]

[♯]*Wuhan National Laboratory for Optoelectronics*
School of Computer, Huazhong University of Science and Technology
[♭]*IBM Center for Advanced Studies (CAS Atlantic)*
University of New Brunswick, Fredericton, Canada E3B 5A3
[§] *Faculty of Business and Information Technology*
University of Ontario Institute of Technology, Ontario, Canada
E-mail: {lfzeng,dfeng}@hust.edu.cn, yangwang@ca.ibm.com, wei.shi@uoit.ca

***Abstract*—It has long been recognized that the Credit scheduler favors CPU-bound applications while for the latency-sensitive workloads such as those related to stream-based audio/video services, its performance is far from satisfactory. In this paper we present an improved Credit scheduler in Xen to facilitate such tasks on multicore platforms. To this end, we improve the Credit scheduler from three perspectives. First, given the identified *Simultaneous Multi-Boost* problem, we minimize the system response time by load balancing the virtual CPUs with the BOOST priority between the cores. Second, we address the *Premature Preemption* problem by monitoring the received network packets in the driver domain and deliberately preventing it from being prematurely preempted during the packet delivery to further reduce and stabilize the I/O latency. Finally, we optimize the frequency of CPU switch by utilizing time-variant slice instead of the existing long time-invariant one to adapt to the dynamic fluctuation of the number of virtual CPUs in the run queue associated with each physical CPU. Our empirical studies show that the proposed improvement can significantly improve the performance of the Credit scheduler for scheduling the I/O latency-sensitive applications.**

***Index Terms*—Xen virtual machine, Credit scheduling, SMP framework, multicore, I/O latency, Dynamic time slice**

I. Introduction

Xen virtualization is being widely adopted by many enterprises at present to underpin their business-oriented services due to its flexibility and efficiency in resource management and provision. With Xen virtualization, one can easily achieve quick resource re-allocation, ideal fault isolation, and increased flexibility, which are all critical factors to the success of service delivery.

Although the benefits of Xen are obvious, its complexity and overhead present additional challenges to adversely effect the overall performance to some applications. One typical example is the I/O-intensive applications with latency sensitivity such as those related to stream-based audio/video services. In general, these applications do not require a great amount of CPU time; rather they are very sensitive to latency. This observation indicates that resource multiplexing and scheduling among virtual machines in Xen is still poorly understood, or at least not fully studied in some regards.

In this paper, we study this VM scheduling problem in Xen [1] by improving its default *Credit scheduler* [2], [3] to minimize the response time of the latency-sensitive applications. The Credit scheduler is the most recent *work-conserving* (WC-mode) [3] scheduler in Xen, which is characterized by using a credit system to fairly share processor resources while minimizing the wasted CPU cycles. Hence, it is very intuitive for users to think about CPU allocations, which should be flexible enough to achieve global load balancing of virtual CPUs (VCPUs) across physical CPUs (PCPUs) on SMP hosts in order to achieve high performance for CPU-bound tasks.

Although the Credit scheduler is efficient to computation-intensive workloads, it could incur a certain amount of latency for I/O-intensive tasks due to the mismatched or inefficient designs in its VCPU allocation strategies, data distribution mechanisms, and time slice calculations. For example, VCPUs could be frequently switched in dealing with I/O packet distribution if a large amount of blocked I/O operations exist, including those in Dom0 which is also subjected to the same scheduling algorithm as other guest domains. This would adversely impact the overall system performance as the control domain (*Dom0*) back-ends the communication directly with I/O devices to complete all the I/O requests in the system. On the other hand, the fixed time slice the Credit scheduler algorithm assigns to each PCPU may also potentially cause frequent switching, cache miss or low hit rate, and sporadic task starvation [4].

We make contributions to improving the Credit scheduler with enhanced performance to the I/O latency-sensitive applications. First, we minimize the system response time by balancing the I/O intensive tasks of the VCPUs with BOOST priority (i.e., BOOST VCPUs) in multicore systems. Second, we reduce the VCPU dispatching time by monitoring the received network packets and deliberately prevent the scheduler from being prematurely preempted during the packet delivery. Finally, we optimize the frequency of CPU switch by utilizing time-variant slice instead of the existing long time-invariant one to adapt to the dynamic fluctuation of the number of VCPUs in the run queue of each PCPU. Our experimental results show that the proposed optimization can significantly improve the performance of the Xen's scheduling algorithm.

II. Background Knowledge

Xen virtualization is a layer of software, termed *Virtual Machine Monitor* (VMM) or *hypervisor* in Xen parlance, that abstracts the underlying hardware resources of a single physical server into multiple instances of virtual machines (VMs) (aka *Domains* in Xen) co-existing and co-executing simultaneously. Domain as the VM presents the virtualized resources such as CPUs, physical memory, network connections, and block devices as an illusion of a real machine to the overlying guest OS and applications.

A. Inter-Domain Communication in Xen

According to Xen, the domains in the same physical machine are not in the same functional class. Dom0 as a privileged domain (i.e., control domain) is created during the boot time to take the responsibility for hosting the application-level management software, including the control of other domains, termed DomUs in Xen (e.g., domain creation, domain termination). In particular, Dom0 can function as a driver domain if it hosts the device drivers and performs I/O operations on behalf of the DomUs[1]. In contrast, DomUs are unprivileged guest domains which as discussed present the virtualized resources to the overlying guest OS and applications. They cannot directly access the physical hardware on the machine. Rather, to accomplish an I/O operation (i.e., network and disk accesses), DomUs have to cooperate with Dom0 via two ring buffers: one for packet transmission and the other for packet reception.

B. Xen's Credit Scheduler

The Credit scheduler is a fair share algorithm based on the proportional scheduling. Each domain is assigned a *credit* value which is determined by the defined *weight* for the domain. The credit value represents the CPU share that the domain is expected to have. Therefore, the domains should have an equal fraction of processor resources if each domain is given the same number of credits. The credits of the running domains as the cost paid for the processor resources are deducted periodically (100 credits for every 10ms via scheduler interrupts, but a domain that runs for less than 10ms will not have any credits debited). Whenever its credit value is negative, the domain is in OVER priority, otherwise, in UNDER priority. Every so often (30ms), a system-wide accounting thread recomputes the credits earned by each running domain according to its weight.

By default, Xen allocates only one VCPU to each domain when the domain is created[2], and it contains general information related to scheduling and event channels. To support the numerous processes in the guest OS and applications, the VCPUs will be scheduled among the PCPUs, which is achieved by maintaining a local ready VCPU (domain) queue (i.e., run queue) for each PCPU. The queue is periodically (accounting period 30ms) sorted in such a way that VCUPs with UNDER priority (i.e., UNDER VCPUs) will always run ahead of VCPUs with OVER priority (i.e., OVER VCPUs), and the ordering within each priority is round-robin. The system always schedules the VCPU to run which is at the head of the queue. The selected VCPU will receive 30ms before being preempted to run another VCPU. VCPUs in OVER priority cannot be scheduled unless there is no UNDER VCPUs in the run queue. This implies a domain cannot use more than its fair share of the processor resources unless the processor(s) would otherwise have been idle. When a processor is idle or the processor's run queue has no more UNDER VCPUs, it would check other processors to find any eligible VCPUs to run on this processor, achieving the global load balancing.

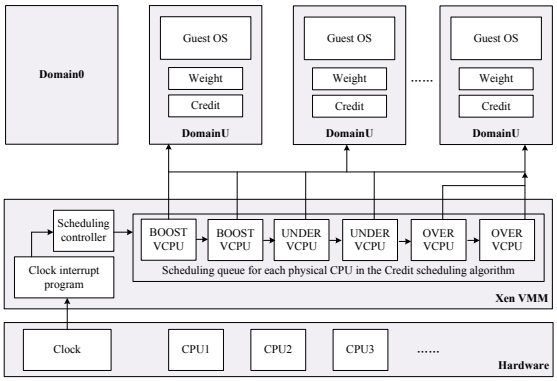


Fig. 1: The organization of the Credit scheduler (IDLE state is not illustrated here).

However, the original Credit algorithm is merely amenable to the compute-intensive workloads and biased against the I/O-latency sensitive applications. To address this problem and minimize the latency of I/O event handling, a boost mechanism is introduced into the Credit scheduler by adding a new priority BOOST to the system so that a VCPU with BOOST priority (i.e., BOOST VCPU) is allowed to preempt a running UNDER VCPU. The current Credit scheduler boosts the priority of the blocked VCPU in UNDER to BOOST after it is woke up by receiving an event over its event channel. The woke-up VCPU preempts the running VCPU at once rather than entering the run queue to compete with other domains. As a consequence, the response time of I/O-intensive tasks is reduced. As an illustrative example, Fig. 1 shows the organization of the Credit scheduler where four Domains are co-located in the same machine to multiplex the underlying physical resources for effective utilization.

III. Performance Analysis of Xen Credit Scheduler

In this section, we identify several not yet fully-studied sources of performance flaws in the current Credit scheduler that could slowdown the I/O-latency sensitive applications.

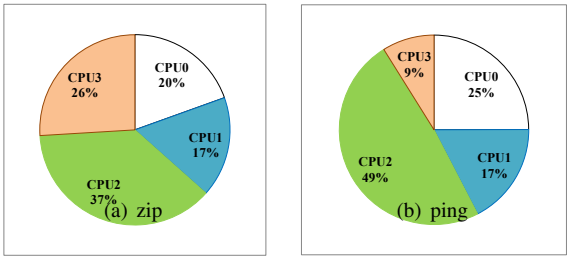
[1]Current Xen-based systems usually employ the *Isolated Driver Domain* (IDD) to conduct real I/O operations the reliability and safety of the system.

[2]Xen also allows a domain to have access to more than one CPU by allocating more than one VCPU. For easy presentation, we assume that each domain is configured by one VCPU. Thus VCPU and domain can be interchangeable in the discussion.

We refer to them as *Simultaneous Multi-Boosting*, *Premature Preemption*, and *Fixed Time Slice*.

1) Simultaneous Multi-Boosting (SMB): SMB refers to the phenomenon that multiple domains could be boosted at the same time. This phenomenon is quite often observed in I/O-intensive applications due to Xen's split driver model as we discussed.

The SMB problem has been recognized in the virtualizations [5], [6]. However, there are quite few research on this problem in the context of multicore platforms. Although the Credit scheduler is merited for its global load balancing, it does not distinguish the states of the VCPUs to balance the loads among the multiple PCPUs. As a consequence, it could be happened that for some PCPUs, there are more BOOST VCPUs waiting in the queue to be scheduled than the others, which results in the imbalance of the BOOST VCPUs as shown in Fig. 2 where the uneven distribution of the BOOST VCPUs for two I/O intensive benchmarks (ping and zip&gzip) on a quad-core platform with 32 test domains (excluding Dom0) are shown. In this paper, we intend to make a contribution on this problem for the multicore platforms.



Group	CPU0	CPU1	CPU2	CPU3
(a) zip (Linux-2.6.18)	112	98	215	149
(b) ping (1000Mbps Eth.)	146	102	285	52

Fig. 2: Distribution of the BOOST VCPUs among the physical CPUs. 32 guest domains are evenly divided into 4 groups, each having 8 domains concurrently running on Xen-3.4.2. Xentrace is used to create a record whenever a VCPU enters BOOST state in a time interval of 1s.

According to Xen I/O model, arriving packets are first delivered to the hypervisor where the associated physical interrupts are virtualized and sent to the driver domain via the event channel. The driver domain demultiplexes the packets after it receives the corresponding virtual interrupt, and then notifies the availability of one or more packets to a target guest domain (also via the event channel connected with the guest) which will be woke up by the hypervisor if it is blocked. Since sending an event will result in the scheduler tickle, the PP problem will be caused as the driver domain could be preempted by a guest domain even though it has multiple pending interrupts for the arriving packets not being processed. In this situation, the driver domain is prematurely scheduled out in the sense that some guest domains cannot timely receive their notices and have to wait till when the driver domain is re-scheduled by the hypervisor, rendering the whole system to be less responsive.

2) Fixed Time Slice (FTS): The Credit algorithm is by nature a sort of round-robin algorithm, a fixed time slice of 30ms is designed independent of the number of the VCPUs as well as their states to ensure the fairness among the domains. However, this may result in two problems: when the number of VCPUs in the run queue is small, the fixed time slice will cause frequent switches between the VCPUs in the run queue, increasing the system overhead. On the contrary, when the number of VCPUs is large, the fixed time slice will increase the response time of those VCPUs at the tail of the queue. In theory, for a particular time interval, the longer the time slice is, the fewer switch times and the shorter average turnaround of the scheduling are. However, for a latency-sensitive task, its response time is closely related to the number of tasks waiting in the run queue. Therefore, the fixed time slice might not be sufficiently adequate and dynamic time slice adapting to the number of VCPUs waiting in the run queue is probably desired. On the other hand, as the number of CPUs in the same physical machine is constantly increased on a large-scale. The number of VCPUs associated with each PCPU would be steadily decreased for more efficient consolidation. Therefore, the Credit scheduling algorithms need to consider and optimize the case where the run queue of PCPU has less VCPUs. Again, the fixed time slice is also not adequate.

IV. Enhanced Scheduler: Design and Implementation

Given the description of the identified flaws in Credit scheduler, in this section we present the design and implementation of our enhanced scheduler to address these issues.

A. Load Balancing of BOOST Domains (LB)

As we showed in the last section, the current scheduler could cause the SMB problem, which potentially leads to the uneven distribution of BOOST domains among the PCPUs. Although Credit scheduler can achieves global load balancing, it little effect on the balance of the BOOST domains, making the system less responsive as a whole. To address this problem, we perform a load balance for the BOOST VCPUs across all the PCPUs in the machine to improve the response speeds of the VCPUs. Our load balancing algorithm is shown as follows:

After a BOOST domain is woke up and inserted into a run queue, the hypervisor will tickle the scheduler to make a scheduling for the PCPU associated with that queue. However, before this action is performed, we first run a simple load balancing algorithm,

- Checks whether or not the current running VCPU on the corresponding PCPU is BOOST?
- If yes, selects the target PCPU based on some criteria where VCPUs can migrate (ties are broken arbitrarily). However, if the selected target and source PCPU are the same, then exits. Otherwise, inserts the VCPU into the run queue of the target PCPU.
- Otherwise, if the current running VCPU is not BOOST, triggers the PCPU scheduling as usual, and then exits.

In a multicore system, the target PCPU is selected according to the following criteria,

- Searches a free PCPU in accordance with the order of the same core, the same socket, different sockets. If found, then returns the PCPU number.
- Searches the PCPU not running a BOOST VCPU in accordance with the order of the same core, the same socket, different sockets. If found, then returns the PCPU number.
- Finds out the PCPU whose run queue includes the least BOOST VCPUs, then returns the PCPU number.

The rationale behind these criteria is the spatial locality of the computations. For example, if two PCPU share the same core, they can also share the same local cache.

B. Prevention of Premature Preemption (PPP)

This section introduces our design and implementation of the algorithm to prevent the driver domain from being prematurely scheduled out (i.e., the PP problem) in the process of the network packet dispatches. The basic idea of this algorithm is straightforward. When there are multiple pending packets for dispatching, the driver domain will inform the underlying hypervisor of this fact, which as a response will not tickle the scheduler until the driver domain dispatches all these packets and yields.

1) Dispatching vector: The core data structure of this algorithm is a *dispatching vector* of size equal to the number of the PCPUs in the platform. The vector is created in the hypervisor at the boot time and then shared with the driver domain. Each bit represents a PCPU, and the bit value is set by the driver domain and used by the hypervisor. The value of 1 indicates the corresponding PCPU is dispatching packets while the value of 0 denotes that the PCPU is not doing such work.

2) Prevention algorithm: The algorithm runs inside the driver domain whenever a packet is arriving or has been dispatched to the target guest,

- Depending on whether or not the number of I/O virtual interrupts in the event channel is greater than one, the algorithm notifies the hypervisor by setting or clearing the corresponding bit in the vector via a hypercall.
- When the scheduler is tickled by the hypervisor, it first examines the value of the bit that corresponds to the PCPU running the domain driver.
- If the value has been set to 1, then scheduling on the corresponding PCPU is disabled, which means the driver domain cannot be preempted while dispatching packets. Otherwise, scheduling is performed as usual and the driver domain could be preempted.

Our algorithm is simple yet efficient to achieve the goal as only one bit vector is shared between the driver domain and the hypervisor, and the shared bit vector is only updated by the domain driver, incurring no synchronization cost.

C. Dynamic Time Slice (DTS)

As discussed, in Credit scheduler, the fixed time slice of 30ms is used, independent of the states of VCPU, BOOST or UNDER. The slice for the BOOST VCPU is divided into two phases. In the first phase, the VCPU will run under the BOOST priority for 10ms and then switch down to the UNDER priority, entering the second phase. The time length of VCPU in the BOOST state is essential to the scheduling. If this length is too long, other VCPUs with the same priority (i.e., BOOST) could be over delayed, increasing the I/O response time. Otherwise, if it is too short, the BOOST VCPU could not get sufficient time to execute before being preempted by other BOOST VCPUs, balancing out the benefits of BOOST state. With these factors in mind, to strike a balance between the average turnaround time of the system and the mean response time of each individual I/O-sensitive application, we propose dynamic time slice that would be adjusted according to the number of VCPUs in the corresponding run queues. Depending on the state, the time slice after this optimization is defined on the selected VCPU by the conducting the following steps:

- Changing the length of its first BOOST phase as 2ms, approximately amounting to the time slice for the I/O processes running on BOOST VCPUs, inserting it into the run queue, and then changing its priority into UNDER. As a consequence, the selected VCPU would be located before all the UNDER VCPUs and after all the BOOST VCPU in the queue. Clearly, if there is no more BOOST VCPUs in the queue, the just inserted VCPU can be continued to schedule for execution.
- Calculating the time slice for the UNDER VCPU based on the number of UNDER VCPUs in the queue according to the following formula:

$$time\_slice = \begin{cases} 30ms & \text{if } qlen > avg\_length \\ 60ms & \text{if } qlen \leq avg\_lento, \end{cases} \quad (1)$$

  here, avg_length is the default length of the run queue, which is typically set to 4 after system boot-ups. $qlen$ refers to the number of UNDER VCPUs in the queue when the time slice is calculated.

The rationale behind this definition is twofold. First, for the BOOST phase, I/O processes in general could be frequently blocked, yielding the VCPUs. As a result, the VCPUs usually do not need longer time slices. On the other hand, if there are other applications other than the blocked I/O processes running on the same BOOST VCPU, those applications will continue to run at BOOST level till the end of the time slice. As a result, it would delay the scheduling of other BOOST VCPUs, which is not desired. This also requires a small time slice for the BOOST VCPUs. Second, for the UNDER phase, we distinguish two cases. On one hand, due to the VCPU affinity, the number of the UNDER VCPUs in the queue remains small and largely unchanged. If there are a quite few UNDER VCPUs, the time slice can be increased accordingly to minimize the scheduling overhead. As discussed previously,

in reality the number of UNDER VCPUs associated with a PCPU is usually small, and the improved algorithm should optimize this common case. On the other hand, the increased time slice should not be varied in a wide range as otherwise, it could compromise the fairness, the goal of the Credit scheduler. Therefore, in our definition, we use two fixed values (30ms and 60ms) to optimize the scheduler for the short run queues while maintaining the fairness goal.

Given this definition, the algorithm for dynamic time slice is straightforward. First, a color attribute is added to the VCPU's data structure, which is set to red when the VCPU is inserted into the run queue, and to black when it is fetched out for recalculating the time slice. Based on this attribute, the algorithm then performs the following:

1) Checking the selected VCPU state. If it is in BOOST state then the time slice is set to 30ms directly.
2) Otherwise, if the selected VCPU is not in BOOST state, then checking the color of the VCPU. If it is black, the time slice will be set to a value calculated according to Eq. (1). Otherwise, if it is red, indicating the time slice calculated previously has expired, and the algorithm needs to recalculate it.
3) Locking the entire run queue, counting the VCPUs and recalculating the time slice according to Eq. (1), then setting all VCPUs to black and storing the time slices into each PCPU's local structures.
4) Setting the time slice of the forthcoming VCPU to the just calculated value and switching off it.

D. Algorithm Combinations

The three proposed algorithms are independent of each other. They are designed for different problems and invoked at different time during the execution. Currently, the LB algorithm for load balancing the BOOST VCPUs is invoked by the hypervisor after a BOOST VCPU is inserted into a queue and before the scheduler is tickled. This is reasonable under the assumption that each BOOST VCPU has an equal opportunity to preempt the running VCPU, implying that the driver domain can be scheduled out at any time. However, this assumption is not always true when considering the effects of the prevention algorithm (PPP) since the driver domain cannot be preempted arbitrarily. In this situation, the prevention algorithm should be designed in conjunction with the load balancing algorithm for further performance improvements. More specifically, according to the original scheduler, after an VCPU is boosted, it likely preempts the running driver domain even though it has pending packets. However, in our implementation, this premature preemption is temporarily forbidden, and a new PCPU has to be selected for this new woke-up BOOST VCPU with the stated load balancing in mind. Therefore the performance can be improved. In contrast to LB and PPP, the DTS algorithm is relatively isolated to be completely implemented in the Credit scheduler, and invoked when a VCPU is scheduled.

E. Remarks

Although our algorithm to SMB is effective to improve the latency-sensitive applications as we will show in the empirical studies in the next section, there is still some room left unexplored for further improvements. For example, we did not consider the temporal locality and queue features other than the number of BOOST VCPUs in the current PCPU selection criteria.

Like the proposed LB algorithm, the prevention algorithm can also be improved as well. First, we can use a *counter* vector instead of a bit vector to track the number of pending packets in the driver domain. The advantage of using a counter vector is that the values of the elements can be increased or decreased on the fly with respects to the packet arriving at or leaving from the driver domain, saving the work each time to check the event channels. However, a problem with this improvement is the size of the counter, the *probabilistic counting* could be a potential solution to this problem for further study. Second, as a side effect, the prevention algorithm could pin down the driver domain to the PCPU if it has a large number of pending packets, we therefore need new optimizations on both the scheduling algorithms and the network I/O mechanisms running in due course to address this problem; the current dynamic time slice may not suffice, but a large time slice may also have negative effects on the fairness of the Credit algorithm, especially for compute-intensive applications on single-processor platforms.

Since the time slice for each selected VCPU, depending its state, is different, the organization of the run queues may have impact on the performance of the DTS algorithm. We can organize all the logic CPUs in a red-black tree rather than the linked list queue, which could be more efficient. It is not necessary to assign a run queue of VCPU for each PCPU, rather, we can assign a red-black tree for each group to organize and maintain the VCPUs.

V. Performance Evaluations

In this section, we evaluate our algorithms via intensive empirical studies. After introducing the experimental setup, we assess each proposed optimization by measuring the response times of a latency-sensitive application. Our numerical results show that the proposed optimizations can remarkably improve the performance of the latency-sensitive tasks.

A. Experimental setup

Our experimental setup includes two physical machines and 32 virtual machines (excluding the driver domain). The two physical machines are connected by a 100 Mbps Ethernet and each of them is configured according to Table I. All the virtual machines are installed on one physical machine and managed by Xen-3.4.2 while the other one as a remote machine communicates with those virtual machines.

We configure each domain based on Table II and use CentOS 5.2 as the guest OS in the driver domain (i.e., Dom0 in our experiments). As the major goal of this research is to optimize the latency-sensitive applications by reducing the response

TABLE I: Hardware configuration

CPU	Intel Xeon E5462 2.80GHz
RAM	16GB RAM
Core	Yorkfield (45 nm) / Stepping: C0 number of cores: 4
Logic processor	4
Socket **frequency**	Socket 771 (FC-LGA6) 2.80 GHz (400 MHz ×7.0) FSB: 1600 MHz
L1 data cache	4 × 32 KB, 8-Way, 64 byte lines
L1 code cache	4 × 32 KB, 8-Way, 64 byte lines
L2 cache	2 × 6 MB, 24-Way, 64 byte lines rate: 2800 MHz
HardDisk	320G IDE
NIC	100Mbps

TABLE II: Domain (VM) configuration

Kernel	"/boot/vmlinuz-2.6.18.8-xen"
Ramdisk	"/boot/initrd-2.6.18.8-xen.img"
Memory	256
Name	"DomU"
VCPUs	2
Disk	['file:/dom1.img, hda1, w']
Root	"/dev/hda1 ro"

time, we deliberately select the `ping` and a basic search (BS) programs as our benchmarks since these programs are usually adopted for this purpose in the literature. Concretely, a remote system is used to send pings to the guest who only acknowledges the receipt of the ping packets without doing any other computations. We use this benchmark to measure the network latency whereby to evaluate our proposed algorithms.

B. Evaluations on the Load-Balancing of BOOST VCPUs

To evaluate the LB optimization on the BOOST domains, we first create 20 guest domains in this experiment which are evenly divided into two groups, each with 10 virtual machines. We then allow each machine in the second group to ping a machine in the first group every second in order to generate a large amount of BOOST VCPUs. We then measure each ping response time. The comparison results before and after the improvement are shown in Fig. 3. It is clear that the packet response time is not only dramatically reduced after the optimization but also relatively stable between each ping operation. This demonstrates the benefits of the load balancing among the BOOST domains.

This phenomenon is not difficult to understand as during the scheduling, the BOOST VCPUs are organized to wait in the run queue, which would cause the VCPUs in the front of the queue to have more opportunities to get scheduled quickly and thus shorten the ping response time; but for the VCPUs in the rear of the queue, they would get scheduled slowly, lengthening the ping response time and resulting in the unstableness. In contrast, after the improvements with the load balancing, BOOST VCPUs are distributed evenly into each PCPU's run queue so that in average the number of BOOST VCPUs in each run queue is decreased accordingly. As a result, even a BOOST VCPU at the backend of the run queue would not be waiting too long. Therefore, the ping response time is fluctuated little, approximately around 0.08ms, which is relatively stable compared with the before improvement.

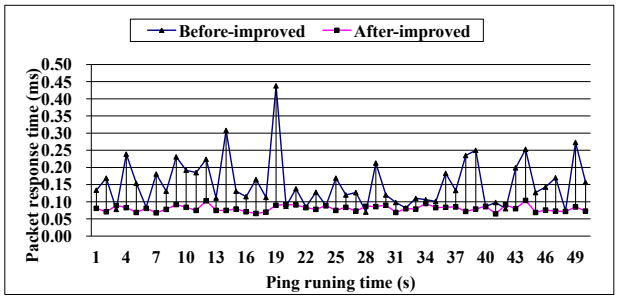


Fig. 3: The runtime of ping program before and after being improved.

C. Evaluations on Prevention of the Premature Preemption

As the system performance will be degraded if the back-end driver is frequently switched off in the process of dispatching network packets, the number of domains involved in this experiment should be increased accordingly so that the advantages of this optimization would be prominently displayed. To this end, we still create 20 virtual machines and another machine to ping these 20 virtual machines. In this way, a large number of the ping packets have to be dispatched by the backend dispatcher to multiple blocked VCPUs. We measure the distribution of the response time of the first 1000 ping requests as the metric to evaluate the optimization. Fig. 4 shows the comparison results on the distribution before and after the improvements. From this figure, we can clearly see that a large percentage of the response time before the improvement falls into the interval greater than 0.10ms whereas for the improved results, it is less than 0.10ms, demonstrating the effectiveness of our improvement to the latency-sensitive applications.

The results are straightforward: before the improvement, due to the extensive communications between the number of guest domains, it is highly likely that the VCPUs allocated to the back-end driver domain would be preempted by the VCPUs that just woke up in other domains. Consequently, it would delay to wake up some other VCPUs to dispatch the ping packets. This is evidenced by the hash bars in Fig. 4 where the ping response time that is greater than 0.1ms occupies a large percentage (close to 47%) of the total results. With the PPP optimization, the VCPUs in the back-end driver domain are prevented from being switched off during the packet dispatching. As a result, the ping response time after improved is less than 0.1ms in a large part (about 75%) as shown by the dotted bars in Fig. 4.

D. Evaluations on Dynamic Time Slices

Although the DTS algorithm is designed to optimize the response time of each individual latency-sensitive task when the run queue length is short, it may also effect the overall performance of other tasks as a whole, especially the compute-intensive tasks. To evaluate the algorithm in a comprehensive

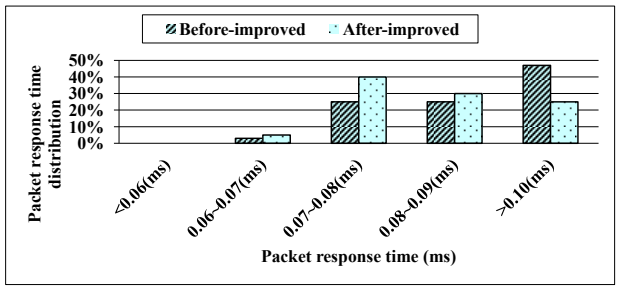


Fig. 4: The ping time distribution before and after the improvement.

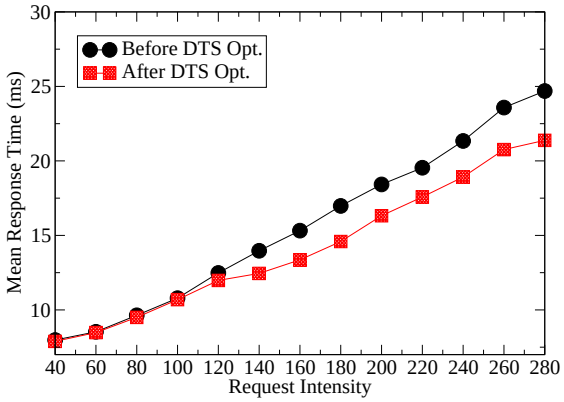


Fig. 5: Mean response time before and after DTS improvement

way, in the following set of experiments, we focus on the evaluations of its impact on both the mean response time of the latency-sensitive applications and the average turnaround time of the system as well. Our results show that with the dynamic time slice, both the mean response time and the average turnaround time can be more or less improved.

1) Mean Response Time of VCPUS: In this experiment, we measure the mean response time of VCPUS by adopting the BS benchmark. To this end, we first configure the measured machine as a search server, which includes four guest OSes, and observe the responses to the search requests in different intensities from these guest OSes. We then use the average of the response time over the four guest OSes as a metric to evaluate the DTS algorithm.

As shown in Fig. 5, the mean response time after the DTS improvement is gradually reduced compared to when the improvement is not employed. This observation is not surprisingly. As the search service is deployed in the virtual machines, it could happen that there are multiple processes running on the same VCPU. When the VCPU is boosted, in addition to the I/O processes, other compute-intensive processes would enable the BOOST VCPU to execute a long time, which is not necessary. After the improvements, the time slice is adjusted to 2ms, only allowing the I/O processes to have the BOOST VCPUs while others having the UNDER VCPUs. In the figure, when the request intensities are low, the mean response time before and after the improvements is almost the same. This is because in this case, there are not that many BOOST VCPUs waiting in the queue. However, with the intensity growth, the results after the improvements is clearly better than those before improved because a large number of BOOST VCPU could be produced as a result of the ever-increasing requests.

2) Average Turnaround Time: In Credit scheduler, the time slice for the UNDER VCPUs is set to 30ms whereas in the improved algorithm, we adjust this value for the UNDER VCPUs based on their queue lengths and optimize the case when such a queue length is less than a default value. In the measurements, to avoid the impact of the multicore architectures on the scheduling algorithm, we deliberately conduct the experiment on a single PCPU by creating different a number of guest OSes, which could further result in a different number of VCPUs for the same PCPU. Additionally, we deploy *SysBench* [7] on each gust OS and adopt its prime-sum[3] as the benchmark to measure the CPU performance.

Table III shows the compared average turnaround times between the after-improved and the before-improved when the number of VCPUs in the run queue is varied from 1 to 8. Given the default queue length of 4, we can observe that the turnaround times after the improvements are obviously lower than those before the improvements, and the difference between them is gradually increased with respects to the execution time as shown in Fig. 6. The reason is that when the number of UNDER VCPUs is less than the default value, the algorithm sets the time slice by 60ms in order to minimize the scheduling overhead. However, when the queue length of the UNDER VCPUs is greater than the default value, both the improved and unimproved algorithms have the same time slice of 30ms. As a result, they exhibit approximately the same turnaround time. These results demonstrate that when the number of UNDER VCPUs in the queue is small, increasing the tim e.

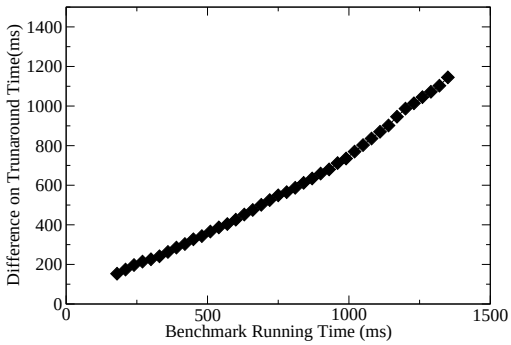


Fig. 6: Difference of the average turnaround time between before-improved and after-improved (the default VCPU queue length is 4).

VI. Related Work

Improving the Credit scheduler to assure that I/O domains will get timely response has long been an active research topic.

[3]The testing command line is "sysbench –test=cpu –cpus-max-prime=100000 run".

TABLE III: Compared turnaround times for the SysBench Prime-Sum benchmark. The default VCPU queue length is 4.

#VCPU	1	2	3	4	5	6	7	8
Before-improved (s)	361.823	724.438	1086.978	1452.598	1807.154	2215.224	2533.542	2895.687
After-improved (s)	361.758	722.989	1084.652	1447.488	1807.065	2215.439	2533.453	2895.406

Although various methods have been proposed [4], [5], [8]–[11], in practice I/O latency is still an obstacle, leaving much room unexplored for further improvements. Govindan *et al.* [8] proposed a communication-aware scheduling to preferentially scheduling communication-oriented domains over compute-intensive counterparts by scarifying a short-term fairness. Similarly, Ongaro *et al.* [5] advocated a *boost/tickle* mechanism to favor the scheduling of I/O domains as we discussed. To supplement this mechanism with knowledge about the characteristics of guest-level tasks for further achieving both low I/O latency and fair CPU allocation, a task-aware VM scheduling was proposed in [9]. Some related work pertain to network virtualization techniques to fully exploit the benefits of virtualization is summarized in [12]. Other research with the same goal in the multicore environments includes [4], [10], [11]. However, no one in these studies considers all the problems we delved into in this paper.

SMB was first identified by Ongaro *et al.* [5] in their original paper of introducing BOOST, and later studied in [6] for real-time Credit scheduling. However, the unbalanced BOOST VCPUs caused by this problem on multiprocessor platforms has not yet aroused much attention in the literature. To our best knowledge, we are the first to investigate this special load balancing problem. The PP problem was also first discussed in Ongaro *et al.* [5] but has not been well studied since then. Yoo *et al.* [6] investigated this problem from a real-time prospective by proposing three strategies. However, their methods are incomparable with ours presented in this paper since our goal is not for real-time computing.

Ongaro *et al.* introduced the ideal of disabling scheduler tickle to prevent the PP problem in [5] but left unexplored. Our method can be viewed as an extension of their idea to the multiprocessor platforms. The subtle, but important extension is that we allow the scheduler to be tickled, yet forbid when scheduling the PCPU, on which the driver domain is running.

Chen *et al.* [4] discussed dynamic time slice problem based on their analysis on the ineffective holding time in the communication-intensive multiprocessing programs. Similar to ours, they also suggested a set of formulae to compute the variable time slice in order to dynamically scale the context switching frequency. However, their goal is to reduce the actual execution time of the program whereas our goal is to improve the response time of each scheduled BOOST VCPU.

VII. Conclusions

In this paper, we presented three improvements to the Credit scheduler in Xen for latency-sensitive applications on multicore systems. These improvements are motivated by the observations on the performance flaws in the current Xen default VM scheduler implementations. The first improvement is the load balancing of BOOST domains to to mitigate the adverse impact of the identified SMB issue. The second is the prevention of the premature preemption to improve the Credit scheduler to address the PP problem for the latency-sensitive applications. The last one is the dynamic time slice which can be adjusted in the runtime as a reaction to the changes of the number of VCUPs in the run queue. Our empirical results show that the proposed improvement can remarkably boost the performance of latency-sensitive applications with minimized the mean response time of VCPUs without have adverse impact on the average turnaround time of the compute-intensive applications.

Acknowledgments

This research is sponsored in part by the National 973 Program of China under Grant No. 2011CB302301. China National Funds for Distinguished Young Scientists under Grant No. 61025008. Special thanks go to Canqun Zhang and Jiaxiang Li for fruitful test on our work. We would also like to thank anonymous reviewers for their constructive comments and helpful advice.

References

[1] 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 *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, New York, NY, USA, 2003, pp. 164–177.

[2] Credit scheduler, http://wiki.xensource.com/xenwiki/creditscheduler. [Online]. Available: http://wiki.xensource.com/xenwiki/CreditScheduler

[3] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three CPU schedulers in Xen," *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 2, pp. 42–51, 2007.

[4] H. Chen, H. Jin, K. Hu, and J. Huang, "Dynamic switching-frequency scaling: Scheduling pinned domain in Xen VMM," in *Proc. of the 39th International Conference on Parallel Processing (ICPP)*, San Diego, California, USA, Sep. 2010.

[5] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling I/O in virtual machine monitors," in *Proc. of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Seattle, WA, USA, March 2008.

[6] B. Yoo, Y. Won, J. Choi, S. Yoon, S. Cho, and S. Kang, "SSD characterization: from energy consumption's perspective," in *Proc. of the 3rd USENIX conference on Hot topics in storage and file systems (HotStorage)*, 2011, pp. 3–3.

[7] A system performance benchmark. [Online]. Available: http://sysbench.sourceforge.net

[8] 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 *Proc. of the 3rd Intemational Conference on Virtual Execution Environments (VEE)*, San Diego, Califomia,USA, 2007, pp. 126–136.

[9] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware virtual machine scheduling for I/O performance," in *Proc. of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Washington, DC, USA, March 2009.

[10] G. Liao, D. Guo, L. N. Bhuyan, and S. R. King, "Software techniques to improve virtualized I/O on multi-core platforms," in *Proc. of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, San Jose, USA, 2008.

[11] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia, "I/O scheduling model of virtual machine based on multi-core dynamic partitioning," in *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, New York, NY, USA, 2010, pp. 142–154.

[12] S. Rixner, "Network virtualization: Breaking the performance barrier," *Queue*, vol. 6, no. 1, pp. 37:36–37:ff, Jan. 2008.