Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

2012

vSlicer: Latency-Aware Virtual Machine Scheduling via Differentiated-Frequency CPU Slicing

Cong Xu
Purdue University, xu172@cs.purdue.edu

Sahan Gamage
Purdue University, sgamage@cs.purdue.edu

Oawab B, Rao
Purdue University, phosakot@cs.purdue.edu

Ardalan Kangarlou
Purdue University, ardalan@cs.purdue.edu

Ramana Rao Kompella
Purdue University, kompella@cs.purdue.edu

See next page for additional authors

Report Number:

12-001

Xu, Cong; Gamage, Sahan; Rao, Oawab B,; Kangarlou, Ardalan; Kompella, Ramana Rao; and Xu, Dongyan, "vSlicer: Latency-Aware Virtual Machine Scheduling via Differentiated-Frequency CPU Slicing" (2012). *Department of Computer Science Technical Reports.* Paper 1757.
https://docs.lib.purdue.edu/cstech/1757

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Authors
Cong Xu; Sahan Gamage; Oawab B, Rao; Ardalan Kangarlou; Ramana Rao Kompella; and Dongyan Xu

This article is available at Purdue e-Pubs: https://docs.lib.purdue.edu/cstech/1757

vSlicer: Latency-Aware Virtual Machine Scheduling via Differentiated-Frequency CPU Slicing

Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou,
Ramana Rao Kompella, Dongyan Xu

Department of Computer Science
Purdue University
West Lafayette, Indiana 47907, USA
{xu172,sgamage,phosakot,ardalan,kompella,dxu}@cs.purdue.edu

ABSTRACT

Recent advances in virtualization technologies have made it feasible to host multiple virtual machines (VMs) in the same physical host and even the same CPU core, with fair share of the physical resources among the VMs. However, as more VMs share the same core/CPU, the CPU access latency experienced by each VM increases substantially, which translates into longer I/O processing latency perceived by I/O-bound applications. To mitigate such impact while retaining the benefit of CPU sharing, we introduce a new class of VMs called *latency-sensitive VMs (LSVMs)*, which achieve better performance for I/O-bound applications while maintaining the same resource share (and thus cost) as other CPU-sharing VMs. LSVMs are enabled by *vSlicer*, a hypervisor-level technique that schedules each LSVM more frequently but with a smaller micro time slice. vSlicer enables more timely processing of I/O events by LSVMs, without violating the CPU share fairness among all sharing VMs. Our evaluation of a vSlicer prototype in Xen shows that vSlicer substantially reduces network packet round-trip times and jitter and improves application-level performance. For example, vSlicer *doubles* both the connection rate and request processing throughput of an Apache web server; reduces a VoIP server's upstream jitter *by 62%*; and shortens the execution times of Intel MPI benchmark programs *by half or more*.

General Terms

Design, Performance, Measurement

Keywords

Virtualization, Scheduler, Cloud Computing

1. INTRODUCTION

The advent of the cloud computing paradigm has allowed enterprises and users to reduce their capital and operational expenditures significantly, because they can simply lease cloud resources to host their applications with a simple pay-as-you-go charging model. A key approach that powers cloud-based hosting is virtual machine (VM) consolidation, where a single physical machine is "sliced" into multiple VMs each assigned virtual core(s) for their execution. While each VM is typically assigned at least one virtual core (e.g., vCPU in Xen [13] parlance), the mapping between virtual and physical cores is not always one-to-one. For example, in commercial cloud offerings such as Amazon EC2 [1], the compute instances (VMs) are allocated in the units of EC2 compute units (ECU), each of which is roughly equivalent of a 1GHz machine, with the smallest EC2 instance allocated 1 ECU. In a 4 GHz physical machine, there may be four VMs sharing a physical CPU. In such cases, the CPU scheduler in the underlying hypervisor (e.g., Xen's default credit scheduler) schedules the runnable VMs in a round-robin fashion, with each VM given access to the physical CPU for the same amount of time, ensuring fairness among the CPU-sharing VMs.

Unfortunately, recent research [35, 18, 24, 34, 33] has discovered a serious downside of CPU sharing among multiple VMs: It leads to significant negative impact on I/O-bound applications running in those VMs. In this paper, we especially address a key aspect of the impact: *I/O processing latency perceived by applications.* More specifically, a VM with a pending I/O event will have to wait for its turn to access the CPU before processing the I/O event. Because of the multiple sharing VMs, the CPU access latency tends to be a *multiple* of the default CPU time slice for each VM (e.g., 30ms in Xen); and such latency cannot be hidden from the corresponding application. This impact is particularly harmful to I/O-bound applications, which in this paper refer to applications *involving both I/O and computation, with I/O dominating computation.* For example, consider a simple VoIP gateway server which basically establishes and maintains connections between clients. For fast call setup and traffic relay, the gateway's network I/O dominates its computation (e.g., audio transcoding). With default CPU slices for the sharing VMs, the VM that hosts the gateway may not be able to access the CPU in time to process requests for new calls or traffic from ongoing calls. Another example is a low-volume web server that needs to quickly respond to client requests, yet its overall CPU usage is relatively lower.

To avoid the impact on I/O processing latency, one could choose to request a non-sharing VM that exclusively occupies a physical CPU. However, that would incur higher cost which may not be desirable for cost-sensitive customers. In this paper, we propose to mitigate such impact *with the presence of CPU-sharing VMs* (e.g., small- or micro-instances of EC2). More specifically, we introduce a new (sub)class of VM instances called *latency-sensitive VMs* (LSVMs), which will achieve better performance for I/O-bound applications.

Contrary to LSVMs, we also define non-latency-sensitive VMs (NLSVMs) for the execution of *CPU-bound applications that do not have stringent timing/latency requirement*. LSVMs and NLSVMs will share the same CPU with fair share and similar cost; whereas the LSVMs will achieve lower I/O processing latency.

One way to enable LSVMs, as advocated by existing work [20, 25, 28, 21], is to modify the hypervisor's CPU scheduler to prioritize certain I/O-bound VMs over the CPU-bound ones. For example, [20] preferentially schedules communication oriented applications over their CPU-intensive counterparts. Unfortunately, it introduces short-term unfairness in CPU shares. Similarly, partial boost is used in [25] to help I/O-bound tasks to preempt a running vCPU in response to an incoming event. However, such a system is hard to configure for preserving fairness among the sharing VMs, which is undesirable for a VM-hosting cloud. The credit scheduler is extended in [28] to support soft real-time applications. But it may give more CPU time to latency-sensitive VMs thus breaking the fairness among VMs. Moreover, all these approaches introduce heavy-weight and intrusive modifications to the scheduler, which involve tracking the VMs' CPU usage and I/O access patterns.

In this paper, we propose our solution named *vSlicer* to realize LSVMs. vSlicer is based on a simple idea which we call *differentiated-frequency microslicing*. Traditional VM schedulers such as Xen's credit scheduler "slice up" a CPU in relatively large time slices. Under vSlicer, we further divide a CPU slice (e.g., 30ms) of a given LSVM into several *microslices* (e.g., 5ms) and schedule the LSVM at a higher frequency (e.g., 6 times) compared to an NLSVM (one time) in each scheduling round. Therefore, both the LSVMs and NLSVMs sharing a physical core will still obtain the same amount of CPU time thus ensuring fairness; but an LSVM will be scheduled more frequently albeit with a smaller time slice, resulting in shorter CPU access latency for the LSVM. Consequently, for an I/O-bound application, vSlicer gives the corresponding LSVM more frequent CPU accesses – each for a shorter duration – to process its pending I/O activities, resulting in better application-level performance. vSlicer is application-agnostic and does not require profiling/inferring individual VMs' workloads at runtime, making it a simple and generic solution for VM-hosting clouds.

Since the overall CPU share is the same for both LSVMs and NLSVMs, their charging model does not need any change and can be priced the same. At first glance, it may appear that every cost-sensitive customer (namely, one who is unwilling to upgrade to VMs with exclusive CPUs) would request only LSVMs. This is not true for the simple reason that LSVMs may not help all applications across the board. In particular, running a CPU-bound application in an LSVM may actually be worse than running it in an NLSVM, because of the more frequent context switches and subsequently more frequent cache flushes. Therefore, customers running CPU-bound applications will be motivated to choose NLSVMs over LSVMs. Consequently, we are likely to see a mix of LSVMs and NLSVMs sharing the physical machines.

The main contributions of this paper are as follows:

- We propose a new class of CPU-sharing VMs called LSVMs to mitigate the impact of VM consolidation on I/O processing latency in VM-hosting clouds. LSVMs achieve much better performance for I/O-bound applications while maintaining the same cost benefit and CPU-share fairness across all sharing VMs.
- We develop a simple, effective technique called vSlicer to realize LSVMs. Based on the idea of differentiated-frequency microslicing, vSlicer enhances the CPU scheduler of the hypervisor by scheduling LSVMs with smaller microslices but with higher frequency while scheduling NLSVMs with regular (larger) slices, giving I/O-bound VMs more timely access to the CPU for I/O processing without penalizing the NLSVMs' CPU shares.
- We have implemented a prototype of vSlicer in the Xen hypervisor and conducted extensive evaluation with both micro-benchmarks and application benchmarks. Our micro-benchmark evaluation shows that vSlicer significantly reduces network packet round-trip times (RTTs) and packet jitter (by 70% compared to the vanilla Xen scheduler). Our evaluation with application benchmarks shows substantial improvement in application-specific performance metrics. For example, in our experiments, vSlicer *doubles* both the connection rate and request processing throughput of an Apache web server; reduces a VoIP server's upstream jitter *by 62%*; and shortens the execution times of Intel MPI benchmark programs *by half or more*.

The rest of the paper is organized as follows. We explain our motivation in detail in Section 2 followed by the design of vSlicer in Section 3. Section 4 describes the Xen based prototype of vSlicer. Then we present our evaluation results in Section 5. We discuss some possible extensions in Section 6 followed by related work and conclusions in Section 7 and Section 8.

2. MOTIVATION

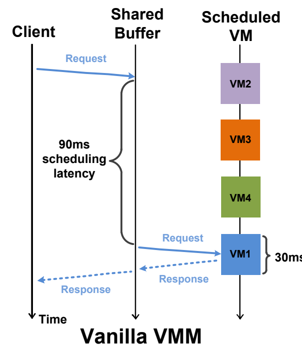


Figure 1: Application responsiveness with credit scheduler

In this section, we motivate the problem by demonstrating the impact of VMs' CPU sharing on I/O processing latency. We then discuss the inadequacy of simple solutions that readily exist today.

2.1 The Impact of CPU Sharing

To understand the negative impact of VM CPU sharing on the latency of I/O processing, consider the example shown in Figure 1. In this example, 4 VMs are sharing a physical CPU. VM1 is hosting an I/O-bound application while VM2-VM4 are hosting CPU-bound applications. The application in VM1 waits for client requests and then responds to the requests with data or control messages. This simple communication pattern can be found in many appli-

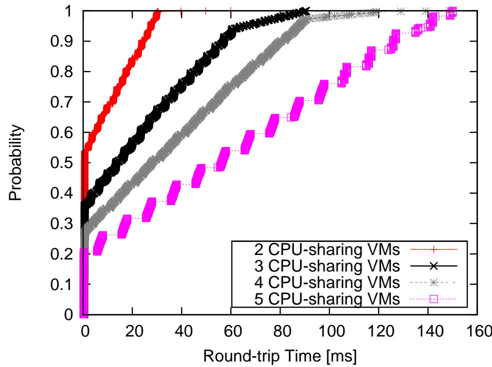


Figure 2: CDF of *ping* round-trip time

cations such as web servers, VoIP proxies, and MPI jobs. We assume that the VM scheduler in the hypervisor uses a proportional-share scheduling policy adopted by many commercial VM platforms (e.g., Xen, that is used in Amazon EC2 [1], RackSpace [10] and GoGrid [2] commercial clouds). Since each VM has a runnable task in it, it occupies the entire CPU slice allotted to it. As shown in the figure, when a request for VM1 arrives at the physical host, it needs to be buffered outside VM1 (e.g., in the VMM or in the privileged driver domain not shown in the figure), until VM1 is scheduled to run. When VM1 gets scheduled, it will process the request and generate a response. Assuming a CPU slice of 30ms, the request response latency can be as high as 90ms (i.e. *(Number of sharing VMs -1)* × *Time Slice*). Such a high latency hampers the responsiveness (and consequently, request processing rate) of the application in VM1.

We perform a simple experiment to demonstrate this increase in latency empirically. Figure 2 shows the CDF of the round-trip time (RTT) by "pinging" VM1. In our measurement experiments, we vary the number of non-idle, CPU-sharing VMs from 2 to 5 (including VM1). Our results clearly show that the ping RTT increases with the number of CPU-sharing VMs; and the worst-case RTT is proportional to *(Number of sharing VMs -1)* × *Time Slice*.

2.2 Problems with Alternative Solutions

We now examine several alternative solutions and argue why they do not work well in our setting.

Prioritize I/O-Bound VMs The first option to reduce the above I/O processing latency is to prioritize the VMs running the I/O-intensive applications. In fact, Xen's credit scheduler uses *BOOST* mechanism to shorten the I/O response time by temporarily boosting (i.e. assigning a higher priority to) the I/O bound VMs. This mechanism works quite well for pure I/O bound VMs. However, in the presence of heterogeneous workloads, once the VM gets scheduled to process the I/O request by the BOOST mechanism, it will consume its CPU share (i.e. credits in Xen terms) due to the CPU bound segment of the workload. This will effectively disable the BOOST mechanism for the rest of this scheduling cycle resulting higher I/O latencies. In other words, while BOOST can temporarily cede the CPU to I/O-bound VMs, it can often lead to exhausting the VM's credits early enough, and then, it may starve for the rest of the scheduling round (since the credit scheduler is CPU-fair across VMs).

A naive workaround to this would be aggressively boosting the I/O bound VMs without considering its CPU share. Unfortunately, this prioritization will break the overall CPU

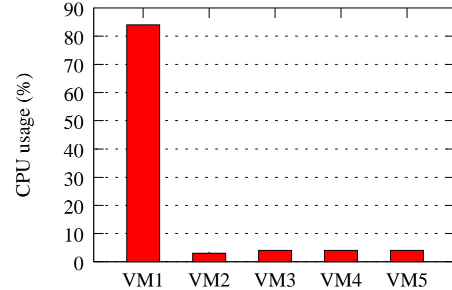


Figure 3: Unfair CPU allocation under aggressive boost

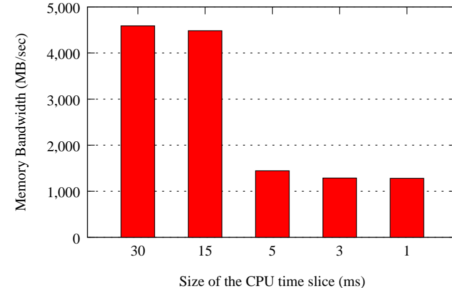


Figure 4: STREAM benchmark performance under credit scheduler with various time slice sizes

fairness in the system. We demonstrate such an unfairness in Figure 3, where VM1 is hosting an I/O-bound application with a network-intensive task and a computation task whereas other VMs are hosting computation-intensive applications. The incoming packets to VM1 trigger the boosting of VM1 so that it can process the packets. However, since the hypervisor does not preempt a scheduled VM as long as the VM has runnable tasks, the computation portion of the application in VM1 will consume the rest of the time slice after the packet processing is done. This causes CPU-time deprivation of other VMs, as long as packets destined to VM1 keep arriving, compromising the CPU fairness of the overall system.

Soft Real-Time Scheduler The second option is to adopt a soft real-time scheduler such as Xen's former scheduler – Simple Earliest-Deadline First (SEDF) scheduler [17]. SEDF is based on a preemptive, deadline-driven real-time scheduling algorithm to achieve latency guarantees. However, such a scheduler requires complex configuration and careful parameter tuning and selection – per-VM – to achieve the latency guarantees desired, which may not be possible in a cloud environment with dynamic placement and migration of VMs. In addition, and perhaps more importantly, extending SEDF to perform global load-balancing on multicore systems is non-trivial, making it not attractive on multicore platforms. Because of these reasons, SEDF has been replaced by the credit scheduler as Xen's default scheduler.

Reducing Slice Size for all VMs The third option is to uniformly reduce the time slice size of the credit scheduler so that all the sharing VMs will get scheduled in and out more frequently, resulting in shorter CPU access latency. However, such an option would increase the number of context switches (and cache flushes) in the system, degrading the performance of CPU-bound applications running in the NLSVMs. To demonstrate the problem with this option, we measure the *memory bandwidth* of VMs running the STREAM benchmarks [6], scheduled by the credit scheduler under various time slice sizes (30ms to 1ms). The STREAM

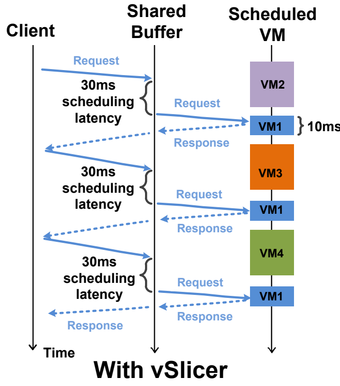


Figure 5: Application responsiveness with vSlicer

benchmarks measure memory bandwidth for large array operations such as copy, addition, scalar multiplication, and triad. Here we only present the "STREAM-copy" results in Figure 4. (We obtain similar results from the other 3 benchmarks.) The results indicate that reducing the time slice size uniformly is clearly not desirable as it degrades the memory access efficiency and consequently application performance of the VMs.

3. DESIGN

The previous section suggests that, if the CPU-sharing VMs are scheduled in a strictly round-robin fashion, it will be difficult to reduce the I/O processing latency without hurting the performance of CPU-bound NLSVMs. On the other hand, prioritizing the LSVMs may violate the CPU share fairness among all VMs. To address this dilemma, we come up with the following key idea behind vSlicer: Within one scheduling round, the CPU time for an LSVM does not have to be allocated in one single time slice. Instead, it can be allocated "in installment" as long as the sum of the installments (i.e., microslices) is equal to a standard CPU time slice. Such a high-frequency microslicing will give more opportunities to the LSVM to process pending I/O events; yet it does not affect/preempt the regular time slices allocated to the NLSVMs. This ensures timely processing of I/O events while maintaining fair share of the CPU among all VMs. We illustrate this idea in Figure 5 for the same application scenario as in Figure 1. In one scheduling round, the LSVM (VM1) will be scheduled three times (instead of once), each for a microslice of 10ms (instead of 30ms). As a result, it can process three requests (instead of one) in the same time period, improving the application's responsiveness.

For the purely CPU-bound applications, as demonstrated in Section 2, there is a strong incentive *not* to run them in LSVMs because the higher-frequency microslicing will cause more frequent cache flushes which will hurt application performance. Fortunately, the NLSVMs under vSlicer will give these applications the same performance as if running them in round-robin-scheduled VMs with the default time slice.

3.1 vSlicer Scheduling Model

The idea of CPU microslicing itself is quite general; one could pick any size for the microslice and simply derive the scheduling frequency. There are two main concerns one needs to keep in mind though. First, setting the microslice too small will excessively increase the context switch overhead; so it is important to keep it to a reasonable duration (e.g., at least 5ms). Second, the best schedule one can come up with, in terms of latency for LSVMs, depends on the number of LSVMs and NLSVMs sharing a core. In practice, we expect only a small number (≤ 5) that share a core, and even among these, the number of LSVMs is going to be very small (≤ 2).

We use the following approach to determine the scheduling order in one scheduling round. Assume m LSVMs and n NLSVMs are sharing a single CPU core. We denote the scheduling period (i.e., scheduling round) by T_P and the total time an LSVM executes during a scheduling period as T_{LSVM}. Similarly, the total time an NLSVM executes during a scheduling period is T_{NLSVM}. We want T_{NLSVM} to be a fairly large value to allow each CPU-bound VM to execute sufficiently long. (In our implementation we use Xen credit scheduler's default time slice 30ms as T_{NLSVM}.) Since we aim to fairly allocate the CPU among all the VMs (both LSVMs and NLSVMs), we want the following to hold:

$$T_{LSVM} = T_{NLSVM} \tag{1}$$

Let us denote the time period where one (micro-)round of LSVMs are scheduled after scheduling an NLSVM as T_S. vSlicer runs all the LSVMs during T_S in round robin fashion. We want to further divide T_S into micro time slices T_m (refer to Figure 6b for illustration of T_S and T_m). The selection of T_m depends on the scheduling latency we intend to achieve. We will further discuss the scheduling latency achieved by the vSlicer later in this section. Depending on the selection of T_m, an LSVM can run one or more times during a single time slice T_S. Let us denote the total time the ith LSVM runs during T_S as T_{n_i}.

$$\sum_{i=1}^{m} T_{n_i} = T_S \tag{2}$$

Suppose the ith LSVM can get scheduled r_i times during T_S. We have:

$$T_{n_i} = r_i \times T_m \text{ where } r_i \geq 1 \tag{3}$$

In this paper, we assume all the LSVMs have the same latency requirement and hence, for any $i, j \in \{1, m\}$ we have $T_{n_i} = T_{n_j} = T_n$ and $r_i = r_j = r$. Equation 3 becomes

$$T_n = r \times T_m \text{ where } r \geq 1 \tag{4}$$

and

$$T_S = m \times T_n \tag{5}$$

Given vSlicer's alternating scheduling of LSVMs and NLSVMs (i.e., it schedules a round of all LSVMs followed by one of the NLSVMs), the total time that an LSVM executes during a scheduling period T_P is equal to the number of NLSVMs multiplied by the time an LSVM executes during a time slice T_S (i.e. T_n). That is:

$$T_{LSVM} = n \times T_n \tag{6}$$

A scheduling period consists of running times of all LSVMs and NLSVMs and therefore we get:

$$T_P = mT_{LSVM} + nT_{NLSVM} \tag{7}$$
$$= m \times (n \times T_n) + nT_{NLSVM} \tag{8}$$

Rearranging the first term of RHS of Equation (8) and substituting from Equation (5) gives us:

$$T_P = nT_S + nT_{NLSVM} \tag{9}$$

Also substituting for T_{LSVM} from (1) to (7) we get :

$$\begin{aligned} T_P &= mT_{NLSVM} + nT_{NLSVM} \\ &= (m+n)T_{NLSVM} \end{aligned} \tag{10}$$

Combining Equations (9) and (10) gives us an important invariant we maintain in the system:

$$nT_S + nT_{NLSVM} = (m+n)T_{NLSVM} \tag{11}$$

That is, maintaining this invariant ensures that we are not violating CPU share fairness while scheduling LSVMs more frequently. Moreover, Equation (11) allows us to define T_S, T_n in terms of T_{NLSVM}. That is :

$$\begin{aligned} T_S &= \frac{mT_{NLSVM}}{n} \\ T_n &= \frac{T_{NLSVM}}{n} \\ T_m r &= \frac{T_{NLSVM}}{n} \end{aligned} \tag{12}$$

As mentioned earlier, the selection of T_m depends on the desired scheduling latency of the LSVM. Equation (12) defines the product of T_m and r in terms of T_{NLSVM} and n. The only restriction for the selection of T_m is, it should be a whole divisor of $\frac{T_{NLSVM}}{n}$. However, selecting a too small value for T_m will increase the number of context switches during T_S, affecting the performance of the all LSVMs.

Let us denote the required latency for an LSVM during T_S as T_l. To achieve this scheduling latency we should schedule the ith VM within T_l. Since we schedule all the LSVMs in a round-robin order, all the other $(m-1)$ LSVMs should be executed in less than T_l. That is:

$$(m-1)T_m \leq T_l$$

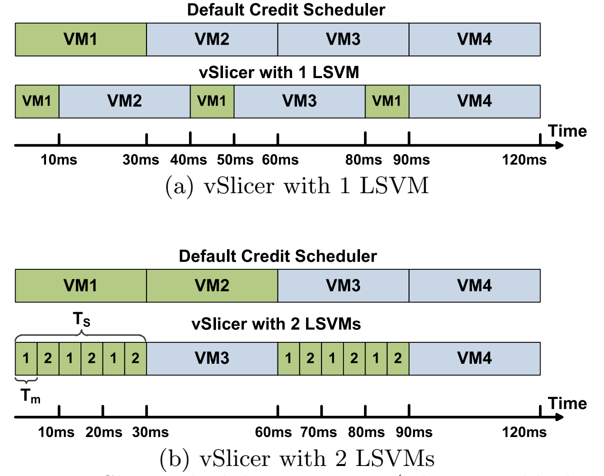
which gives us the upper bound for T_m:

$$T_m \leq \lfloor \frac{T_l}{(m-1)} \rfloor$$

If we consider the influence of NLSVMs, the scheduling latency curve for a specific LSVM looks like a *continuous wavy line*. The wave crest is $T_{NLSVM} + (m-1)T_m$.

Examples We now show two examples of scheduling sequence under two different settings. Figure 6 illustrates two scheduling sequences for a system running four VMs. In Figure 6a, we have one LSVM and three NLSVMs. If all these VMs were scheduled by the default credit scheduler, any of them would experience a 90ms scheduling latency. Under vSlicer, by dividing the time slice of the LSVM (i.e. VM1) to multiple microslices and scheduling it three times during the scheduling round, the latency drops to 30ms. In Figure 6b, there are two LSVMs and two NLSVMs in the system. By dividing the time slice into 5ms microslices, vSlicer can achieve a best-case latency of 5ms and a worst-case latency of 35ms.

In our discussion towards the end of Section 2, we emphasized that reducing the time slice uniformly for all sharing VMs is not a desirable option, primarily due to the increased context switches between the VMs. Now that we have discussed the details of vSlicer, let us quantitatively compare the credit scheduler – with uniformly reduced time slice – with vSlicer using a system with two LSVMs and two NLSVMs. With the credit scheduler having the default time slice, the CPU access latency of each LSVM is (m+n-1)T_{NLSVM}. Here it is (4-1)$T_{NLSVM} = 3 \times 30 = 90ms$. In order to reduce the latency to 15ms, we need to reduce the time slice from 30ms to 5ms, which will make the context switch rate increase by 6 ×. With vSlicer, however, setting $T_m = 5ms$, to achieve 15ms average latency, would increase the number of context switches only by 3 ×.



(a) vSlicer with 1 LSVM

(b) vSlicer with 2 LSVMs

Figure 6: vSlicer scheduling sequence (The green block indicates LSVM)

4. IMPLEMENTATION

vSlicer only requires a simple modification to the VM scheduler in the hypervisor. The VMs in the physical host are grouped at two levels. First, vSlicer maintains a list of VMs that are executing in a physical CPU. Second, within this group vSlicer divides these VMs into LSVMs and NLSVMs. Decision on whether a particular VM is LSVM or NLSVM is left to the user (or the cloud administrator) and vSlicer provides an interface to the administrative tools (such as *xm tools* in Xen) to configure that. However, the grouping of VMs per physical CPU is done by the global load balancing algorithm of the VM scheduler.

While the design of vSlicer is generic and hence applicable to many VMMs (e.g., Xen, VMware [12]), we implement a prototype of vSlicer in Xen 3.4.2. In our implementation, we add a new scheduler type in Xen, called sched_vSlicer by extending the credit scheduler. The vSlicer code is in the critical path of the scheduler code which is frequently executed. Therefore we keep the modifications to the critical path of the credit scheduler to a minimum, with only 250 lines of additional code. The user-level utilities add another 400 lines of code which is executed only when the user configures the system using the Xen management tools.

Since vSlicer is based on the credit scheduler, vSlicer inherits its proportional fairness policy. We maintain the credit scheduler's existing set of controls, *weight* and *cap*, that decide the proportional share of the VM, and the maximum amount of CPU a domain will be able to consume even if the host system has idle CPU cycles respectively. We add a new control in addition to these two to specify the micro time slice. Initially vSlicer treats all the VMs as

NLSVMs, which have their micoslices set to zero. When a user configures a particular VM to be LSVM, the microslice of that VM will be set to the specified value. This action will trigger vSlicer configuration functions, which will in turn re-calculate the global parameters such as T_S. Starting from the next scheduling interrupt, vSlicer will schedule that VM as an LSVM.

Algorithm 1 Scheduling Algorithm for vSlicer

Require: $num_nlsvm \geq 1$
Require: $num_lsvm + nlsvm \geq 3$
Ensure: $schedule_time = now$
Ensure: $time_slice = T_{NLSVM}$
Ensure: $micro_slice = T_m$

```
 1: burn_credit(curr_vm.schedule_time, now)
 2: if curr_vm is nlsvm then
 3:   insert_tail(curr_vm, runq);
 4: else {curr_vm is lsvm}
 5:   burn_micro(curr_vm.micro_credits, micro_slice)
 6:   if curr_vm.credits > 0 then
 7:     if curr_vm.micro_credits > 0 then
 8:       insert_before_nlsvm(curr_vm, runq);
 9:     else {curr_vm.micro_credits ≤ 0}
10:       insert_after_nlsvm(curr_vm, runq);
11:     end if
12:   else {curr_vm.credits ≤ 0}
13:     insert_tail(curr_vm, runq);
14:   end if
15: end if
16: next_vm ⇐ get_first_elem(runq);
17: if next_vm is nlsvm then
18:   next_vm.runtime ⇐ time_slice;
19: else {next_vm is lsvm}
20:   next_vm.runtime ⇐ micro_slice;
21: end if
22: run(next_vm);
```

Scheduling Algorithm The most important function that we modify is *do_schedule*, which is executed in the critical path and responsible for selecting the next vCPU for pCPU from the run queue. We show the pseudo-code of the algorithm in Algorithm 1.

We assign *micro credits* to each LSVM in addition to the credits assigned by the original algorithm of Xen credit scheduler. vSlicer algorithm uses the micro credits to schedule LSVMs during T_S in a round-robin order. We initialize the algorithm by initializing T_{NLSVM}, T_S, and T_m. T_{NLSVM} is defined by the implementation (in our implementation we used Xen's default 30ms). T_S and T_m can be calculated using T_{NLSVM}, m, n, and equations in Section 3.1. This initialization has to be done in the event of: a vCPU migration, a VM initialization, a VM shutting down or any other event that changes the number of VMs running in the particular CPU core.

The vSlicer algorithm is executed whenever the time slice of the currently running VM expires. First the algorithm checks the VM type. If it is an NLSVM, the time slice of it has expired and hence the VM is inserted to the back of the *run queue*. In vSlicer both NLSVMs and LSVMs share a single run queue. If the the current VM is an LSVM, depending on how much credits and *micro credits* the VM has, it will be scheduled to run in the same T_S, in the same T_P, or in the next scheduling period. Then the algorithm picks the next VM to run from the head of the run queue. If it is an NLSVM, it will be assigned a regular time slice (T_{NLSVM}). If it is an LSVM, it will be assigned a microslice.

5. EVALUATION

In this section, we present our detailed evaluation of vSlicer using the Xen-based prototype. We use both micro-benchmarks and application-level benchmarks to evaluate the effectiveness of vSlicer. Our experiments evaluate three key aspects: (a) transport-level latency reduction achieved by vSlicer; (b) overall CPU-sharing fairness with vSlicer; and (c) application-level performance improvement by vSlicer.

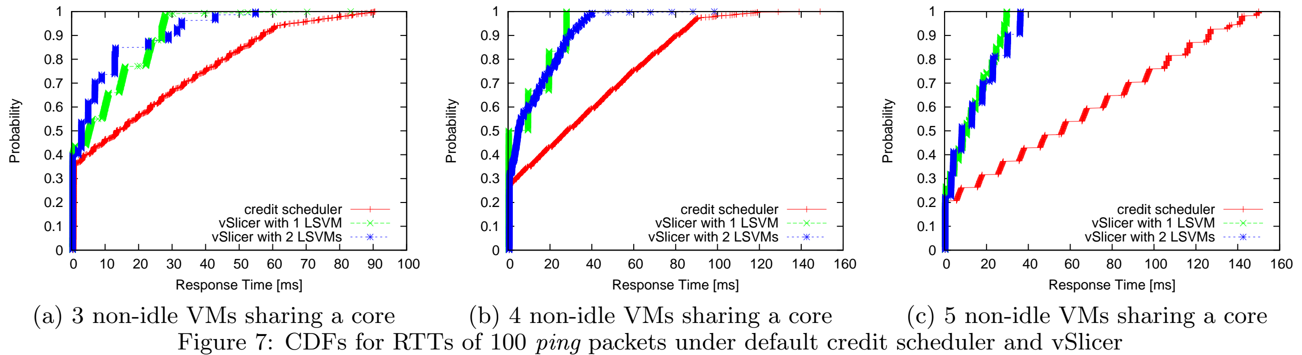
Experimental Setup Our experiments involve physical machines (desktops as clients and servers as VM hosts) connected by a Gigabit Ethernet network. Each physical server hosts multiple VMs and has a dual-core 3GHz Intel Xeon CPU with 4GB of RAM and a Broadcom NetXtreme 5752 Gigabit Ethernet card. These hosts run Xen 3.4.2 with Linux 2.6.18 running in the driver domain (dom0). The VMs share one core of the host, whereas the driver domain is pinned to the other core. Each VM in this host is allocated 512MB of RAM and a single VCPU, except the VM that hosts the MyConnection media server (Section 5.2) which is allocated 1GB RAM following the requirement of the MyConnection benchmarks. The physical client machine has a 2.4GHz Intel Core 2 Duo CPU with 4GB of RAM and an Intel Pro Gigabit network card and runs Linux 2.6.35.

5.1 Evaluation with Micro-benchmarks

This section presents improvement of network I/O performance achieved by vSlicer using micro-benchmarks. In each experiment we vary the number of VMs sharing the same core from 3 to 5 and measure the same transport-level metrics under vSlicer and Xen's default credit scheduler, respectively. We keep the CPU utilization of each VM to 40% using the *lookbusy* tool [7].

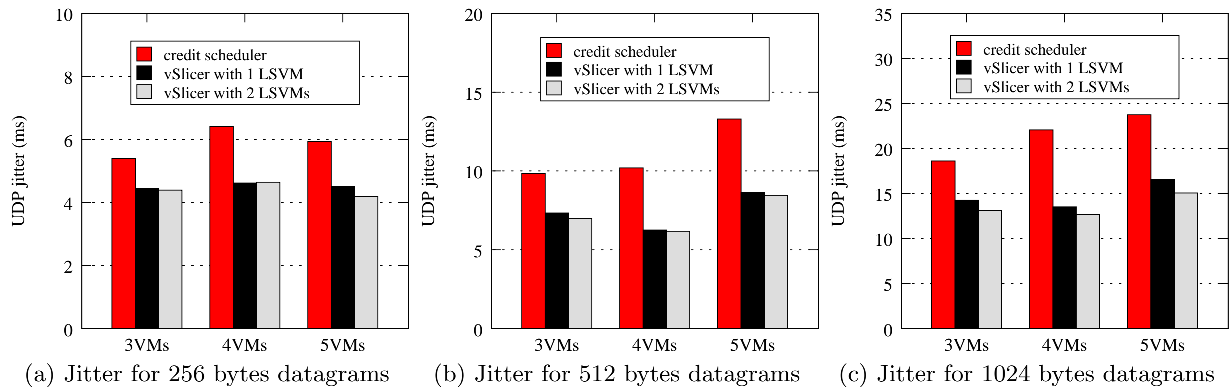
Ping RTT Recall the experiment presented in Section 2 that measures the RTTs of ping packets to a non-idle VM from another physical machine in the same LAN. We repeat the same experiment, but use vSlicer as the VM scheduler and compare the results with those achieved by the default scheduler. Figure 7 shows the CDFs of RTTs of 100 *ping* packets, with 3, 4, and 5 CPU-sharing VMs, respectively. For each setup, we show the CDFs under the credit scheduler, vSlicer with 1 LSVM (the ping receiver), and vSlicer with 2 LSVMs (one being the ping receiver), respectively. These results show that vSlicer consistently reduces the ping RTTs in all setups. For example, in the 4-VMs scenario (Figure 7b), vSlicer reduces the average RTT from 35ms to 10ms with 1 LSVM (the other three are NLSVMs), a 71% reduction. With 5 CPU-sharing VMs (Figure 7b), the average ping RTT is shortened by about 80% under vSlicer. More importantly, we find that, under vSlicer, the RTT towards an LSVM *does not increase linearly with the number of sharing VMs*. With vSlicer, the average RTT we observe across our experiments remains about 12ms with 1 LSVM; and 14ms with 2 LSVMs; whereas a near-linear increase in average RTT is observed under the default scheduler.

UDP Jitter UDP is a simpler transport protocol with no reliable, in-order packet deliver guarantee. Yet UDP is



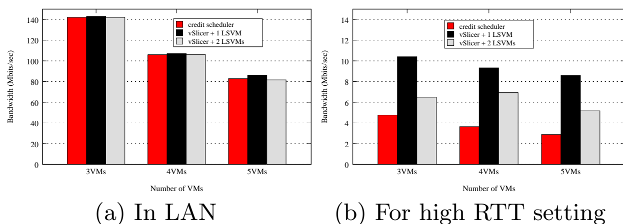
(a) 3 non-idle VMs sharing a core (b) 4 non-idle VMs sharing a core (c) 5 non-idle VMs sharing a core

Figure 7: CDFs for RTTs of 100 *ping* packets under default credit scheduler and vSlicer



(a) Jitter for 256 bytes datagrams (b) Jitter for 512 bytes datagrams (c) Jitter for 1024 bytes datagrams

Figure 8: Effect of vSlicer on UDP jitter



(a) In LAN (b) For high RTT setting

Figure 9: Effect of vSlicer on TCP throughput

popular in audio/video streaming, online gaming and other latency-sensitive applications. We measure the jitter of UDP datagrams, which will translate into user-level QoS of the aforementioned applications. We use Iperf [5] to generate a stream of UDP datagrams and vary the datagram size in each setup. The UDP receiver runs in a non-idle VM (an LSVM when running on vSlicer) and the UDP sender is a different physical machine in the same LAN. We also vary the number of CPU-sharing VMs from 3 to 5. The average UDP jitter observed on the receiver side is shown in Figure 8. The results under different datagram sizes all show UDP jitter reduction. The reason for the jitter reduction is that an LSVM has multiple opportunities to run during one scheduling round under vSlicer (vs. only one under the default scheduler), leading to more timely and more evenly timed processing of UDP datagrams.

TCP Throughput Our measurement of TCP throughput generates some interesting (and somewhat surprising) results. Since vSlicer reduces a VM's CPU access latency and benefits latency-sensitive applications, we first thought that vSlicer would also improve TCP throughput to/from a VM. We use Iperf to measure the TCP bandwidth between a physical machine and a VM in the same LAN. The Iperf server runs in a non-idle (40% CPU load) VM sharing the CPU core with 2-4 other non-idle VMs. Interestingly, as shown in Figure 9a, vSlicer does not improve TCP throughput within a LAN. The reason, after a closer examination, is the following: First, even with vSlicer, LSVMs experience longer latencies periodically when the NLSVMs are getting scheduled. This delay would be less compared to the delay with the default credit scheduler (30ms compared to the 60ms in 3 VM scenario). However, this is still high compared to the sub-millisecond latencies in the LAN environments. Second, when we microslice the time slice of the LSVM (in this case from 30ms to 15ms) , we also reduce the amount of packets that can be processed during a single micro time slice by some fraction (by 50% in this case), which means that the rest of the packets have to wait one full NLSVM execution time slice until they get processed, which makes throughput of the connection similar with credit scheduler and vSlicer.

However, the results are different when we look at a WAN environment. We simulate higher RTTs in a WAN by adding 30ms of network delay between the TCP sender and receiver using Linux *netem* module. This time we observe that vSlicer improves TCP throughput by up to 3×, as shown in Figure 9b. When we add 30ms network delay, this delay will effectively mask the execution period of the NLSVM. Recall that our VM scheduling pattern from Section 3 – an execution of an NLSVM is always followed by an execution period of all the LSVMs. So if we consider 3 VM case with one LSVM, once LSVM acknowledges a set of TCP packets and schedule out, it will take another 30ms time for the arrival of another batch of TCP data segments due to the

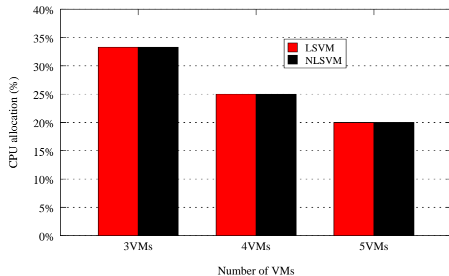


Figure 10: CPU utilization for different VMs with credit scheduler and vSlicer

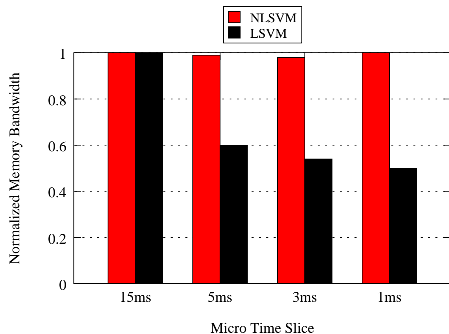


Figure 11: STREAM benchmark performance under different configurations

added network delay. Now, during this 30ms in the receiving host, one NLSVM will be executed and the LSVM will be scheduled by the time of the arrival of TCP data packets, which can be immediately processed. On the other hand if we consider the default credit scheduler, adding 30ms network delay will mask the execution time of just one VM. However since credit scheduler schedules VMs in the round-robin fashion, in the 3 VM scenario, packets still have to wait one more time slice until the receiving VM gets scheduled.

Fairness of CPU Sharing After evaluating vSlicer's improvement of network I/O, we now evaluate the fairness of CPU sharing among all sharing VMs (LSVMs and NLSVMs). We use *xentop* to monitor the CPU utilization of each VM while running lookbusy and sysbench benchmark in each VM. We observe that, regardless of its type (LSVM or NLSVM), each VM has an equal share of the CPU as the other VMs. Figure 10 shows the average CPU utilization (reported by *xentop* over a period of 30 seconds) of one LSVM and one NLSVM (out of a total of 3, 4, or 5 VMs) under the credit scheduler and vSlicer, respectively. The results show that vSlicer maintains the same CPU sharing fairness as the credit scheduler.

We then measure the performance of a CPU/memory-bound application running in an NLSVM under vSlicer. We use the STREAM benchmarks as in Section 2 and run 4 VMs – two LSVMs and two NLSVMs in a physical host. We run the STREAM benchmark in one of the NLSVMs, each getting one regular 30ms time slice in a scheduling round, while we vary the microslice size (from 15ms to 1ms) of the sharing LSVMs. Figure 11 shows the results in terms of memory bandwidth achieved by the benchmark. For comparison, we normalize the memory bandwidth relative to the one achieved by the default credit scheduler with the same 4 VMs and same workloads. The results show that the performance of STREAM running in the NLSVM (the red bars) is not affected by the more frequent scheduling of the LSVMs under vSlicer, maintaining (almost) the same performance as under the credit scheduler. To demonstrate the unsuitability of LSVMs for CPU-bound applications, we also run the STREAM benchmark in an LSVM and the results are shown by the black bars in Figure 11. This time the STREAM performance degrades with the decrease of microslice size (i.e., with the increase of LSVM scheduling frequency).

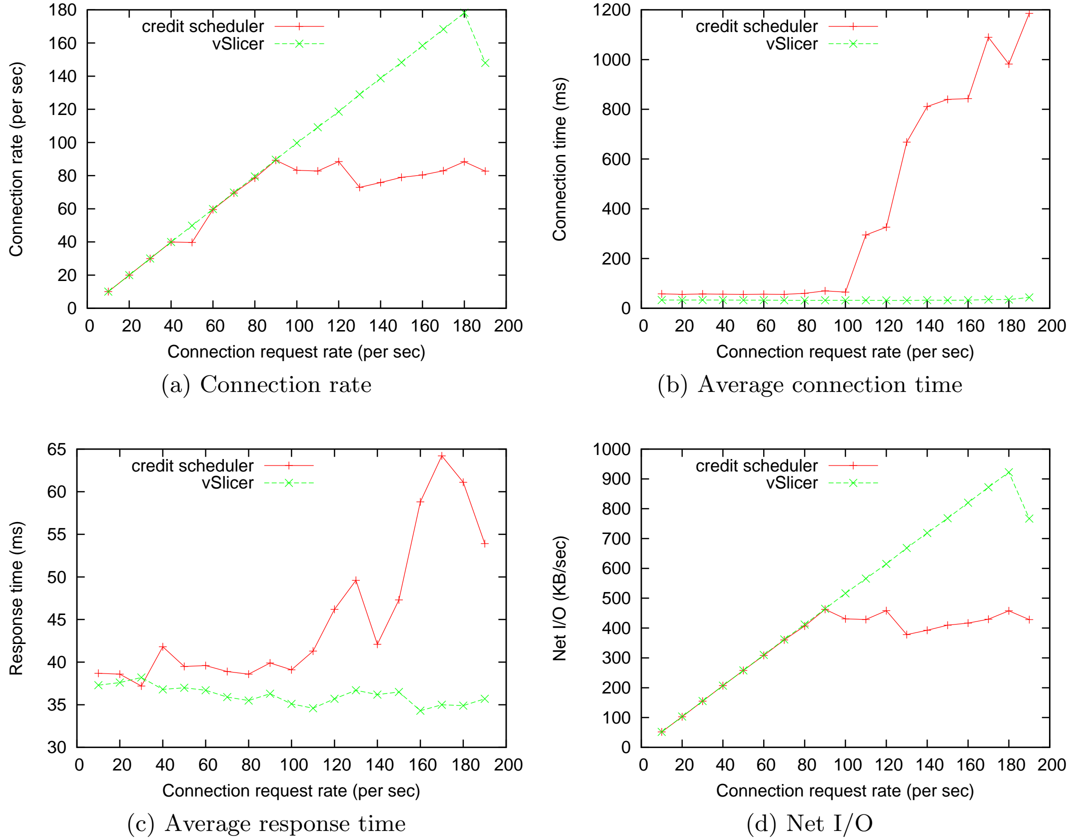
5.2 Evaluation of Application Performance

Experiment with Apache Web Server We first use the Apache web server along with httperf [3] to evaluate the effectiveness of vSlicer for I/O-bound applications. While not a soft-real-time application, the Apache web server is sensitive to (network and disk) I/O processing latency, which will cause delay in both connection establishment and data transmission stages and thus affect the web server's response time and request handling throughput.

In this experiment the physical server hosts four core-sharing VMs. Two of the VMs are LSVMs, with one of them running the Apache web server. A physical client machine generates requests for a 5KB web page with httperf to measure the web server's performance. For comparison, we perform the experiment under the default credit scheduler and under vSlicer. We measure the following metrics: (a) connection rate, (b) connection time, (c) response time, and (d) net I/O (average network throughput), with the corresponding results shown in the four sub-figures of Figure 12. Under the credit scheduler, the connection rate saturates at 90 connections/sec and the net I/O throughput saturates at 450 KB/s. Under vSlicer, Apache can sustain up to a 180 connections/sec connection rate and achieve up to 900 KB/s throughput. Moreover, the connection time and response time are much shorter and more stable under vSlicer; whereas under the credit scheduler, these two metrics increase rapidly once the request rate goes beyond 100 requests/sec.

To understand the root cause for the saturated connection rate of 90 connections/sec under the credit scheduler, we first traced packets using *tcpdump* at multiple points: (1) in the client host, (2) in the driver domain of the physical server, and (3) in the LSVM where the Apache server runs. We make two interesting observations: First, when the connection rate goes beyond 90 connections/sec, packet retransmissions start to appear in the trace. Second, our further analysis of flows with packet retransmissions shows that almost all of the retransmissions happen due to the packets dropped at the driver domain (by comparing the traces from the driver domain and from the VM).

To identify the main culprit of the dropped packets inside the driver domain, we inserted tracing points along the path taken by the packets inside the driver domain from physical NIC (*peth*) to the VMs virtual interface (*vif*). We found out that the I/O ring buffer, which connects the driver domain and the VM, gets full when the request rate exceeds 90 connections/sec while the VM is waiting in the run queue. This in turn back-pressures the packet processing *tasklets* in the driver domain causing packet drops. On the other hand, with vSlicer, the LSVM running the Apache server gets scheduled more frequently and hence, it empties the ring buffer more often hence eliminating the back-pressure. Compared with the maximum CPU access latency (90ms) under the credit scheduler, the maximum latency for the



(a) Connection rate (b) Average connection time

(c) Average response time (d) Net I/O

Figure 12: Apache web server experiment results

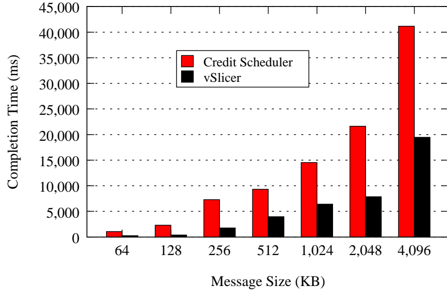


Figure 13: Performance of Intel MPI benchmark: *Alltoall*

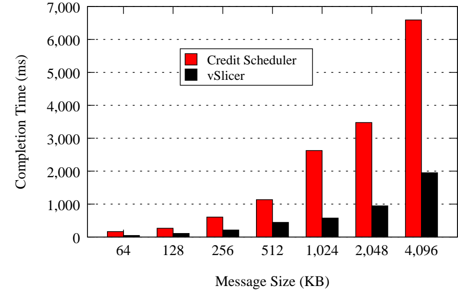


Figure 14: Performance of Intel MPI benchmark: *Sendrecv*

LSVM is 40ms under vSlicer. This translates into a higher connection rate (up to 180 connections/sec) of the web server without packet drops and retransmissions in the driver domain.

Experiments with MPI Benchmarks We next evaluate the effectiveness of vSlicer for reducing the execution time of MPI communication primitives using the Intel MPI Benchmark (IMB) [4]. Our setup consists of 4 VMs each with MPICH2 [8] libraries installed. We host these 4 VMs in two physical hosts with 2 VMs sharing a single CPU core. We also run 2 other VMs per core with CPU-bound tasks. When experimenting with vSlicer, we mark the VMs running the IMB as LSVMs and the VMs running CPU-bound tasks as NLSVMs. We measure the execution time of two MPI communications primitives from IMB suite: *Sendrecv* and *Alltoall*.

In the IMB *Alltoall* benchmark, each MPI process sends a distinct message to each process in the system. A process executing this communication pattern usually sends messages to all other processes using non-blocking *sends* and waits for the receipt messages from all other processes. When vSlicer is used, each LSVM gets scheduled frequently for a micro time slice period (of 5ms), leading to more timely processing of send/receive messages to/from other processes and hence faster process of the entire MPI job. Figure 13 shows that, under various message sizes, vSlicer reduces the execution time by half or more, compared with the credit scheduler.

In the IMB *Sendrecv* benchmark, the MPI processes form a periodic communication chain. Each process sends a message to its right neighbor in the chain and receives a message from its left neighbor. Figure 14 shows the results for this benchmark. vSlicer leads to significant reduction in the execution time (up to 4.5 $\times$ improvement when the message size is 1024KB). The reduction is even higher than in the *Alltoall* case. The main reason lies in the chain of dependencies imposed by this particular communication pattern.

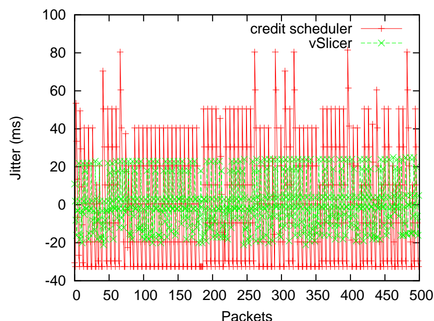


Figure 15: Single line VoIP upstream jitter

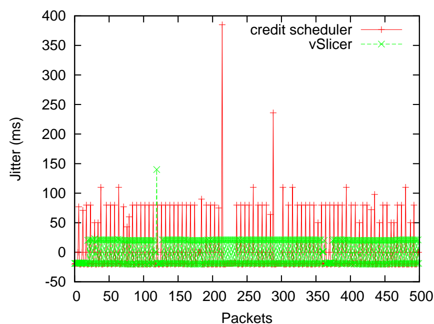


Figure 16: Single line VoIP downstream jitter

Each process depends on its left neighbor to receive and acknowledge the message being sent; and each process depends on its right neighbor to send a complete message. The longer message processing delays incurred by the credit scheduler causes the entire messaging chain to take longer time in a *cascading* way. When vSlicer is used, each LSVM has multiple opportunities in one scheduling round to process those incoming/outgoing messages, leading to faster progress of messaging chain.

Experiments with MyConnection Server Finally, we evaluate the effectiveness of vSlicer with latency-sensitive, soft real-time applications such as streaming media servers and VoIP gateways. We use MyConnection Server (MCS) [9] as our benchmark application. MCS is a suite of benchmarks for assessing the performance and quality of networking and computing infrastructures for hosting soft real-time applications such as VoIP, video streaming, IPTV, and video conferencing. We use the *VoIP* test and the *streaming video* test of MCS for our evaluation of vSlicer. We run MCS in a VM which shares the same CPU core with 3 other non-idle VMs. Two of these VMs are LSVMs, including the VM that runs the MCS tests. The VoIP/media streaming clients run in another physical machine in the same LAN, but we

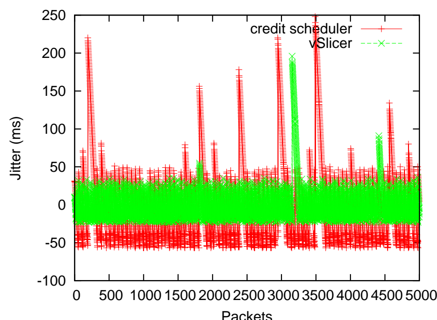


Figure 17: Multi-line VoIP upstream jitter

simulate remote clients in the real world by introducing a random delay between 10ms to 30ms using the Linux *netem* module.

The VoIP test generates voice traffic of one or more active VoIP sessions with a selected audio compression algorithm. In this test, a VoIP client connects to the MCS via the SIP protocol, emulates one or more voice conversations using G.711 codec, and measures QoS metrics such as jitter, packet loss, and the discarded packet percentage. Figure 15 and Figure 16 show the upstream and downstream jitter for the *single line* VoIP test (i.e. when only one VoIP session is active), respectively. Figure 17 shows the upstream jitter for the *multi-line* VoIP test (i.e. when multiple VoIP sessions are active simultaneously). Table 1 and Table 2 summarize the results of the VoIP test. Compared with the credit scheduler, vSlicer achieves a 16.6ms (62%) reduction in upstream jitter and 11ms (43%) reduction in downstream jitter in the single line VoIP test. In the case of multi-line VoIP test, vSlicer achieves a 23.7ms (65%) reduction in downstream jitter and 29.2% reduction in downstream packet loss. Under the credit scheduler, we could not even obtain meaningful downstream jitter results for the multi-line VoIP test, due to the heavy packet loss.

Scheduler	Upstream Jitter	Downstream Jitter	Packets Discard
Credit scheduler	26.7ms	25.8ms	1.2%
vSlicer	10.1ms	14.8ms	0%

Table 1: Single line VoIP test results under credit scheduler and vSlicer

Scheduler	Upstream Jitter	Downstream Packet Loss	Packets Discarded
Credit scheduler	36.7ms	44.5%	6.0%
vSlicer	13.0ms	15.3%	1.5%

Table 2: Multi-line VoIP test results under credit scheduler and vSlicer

Scheduler	Video Jitter (ms)	Audio Jitter (ms)	Trip Time (ms)	SETUP Time (ms)	DESCRIBE Time (ms)	PLAY Time (ms)
Credit	46.2	41.2	110	361	480	509
vSlicer	16.6	15.8	51	176	262	243

Table 3: Streaming video test results under credit scheduler and vSlicer

The *streaming video* test involves video streaming sessions from the MCS to the clients via TCP based on the Real Time Streaming Protocol (RTSP) [11]. The streaming video server sends a series of audio and video packets at a fixed rate to the client. The client will measure the packet jitter and the server will measure the trip time, which is the application-level round-trip time. The test also measures the time to perform different RTSP commands such as SETUP, DESCRIBE, and PLAY. In this experiment, the payload of each audio packet is 32 bytes and the payload of each video packet is 160 bytes. The media transmission rate is 20 packets per second (for both audio and video packets). Table 3 shows the results of the test. Compared with the credit scheduler, vSlicer reduces the video jitter by 29.6ms (64%)

and reduces the audio jitter by 25.4ms (62%). Furthermore, vSlicer achieves significant improvements (time reduction) for all the other streaming video metrics measured.

6. DISCUSSION

Non-uniform Microslice Size In this paper, we assume that each LSVM is assigned the same micro time slice. In fact, vSlicer can be fairly easily generalized to support different microslice sizes for LSVMs in the same system. The change to the scheduling model (Section 3.1) is that, instead of scheduling LSVMs in a round-robin order, different microslices will need to be accommodated within T_S. One possible approach would be to use the earliest-deadline-first [29] policy within T_S. Another possibility would be to devise a frequency-based policy where the number of times an LSVM gets scheduled within T_S is inversely proportional to its microslice (T_m).

Determining VM Type As we pointed out in Section 4, the decision on whether a particular VM is an LSVM or an NLSVM is left up to the user/administrator. However, it may be possible to infer the type of a VM by dynamically monitoring its I/O and computation behavior. Moreover, since an application or an entire VM may change its behavior (e.g., from I/O-bound to CPU-bound) at runtime, it is desirable to allow a VM to *switch* between LSVM and NLSVM dynamically. We leave such capability as future work.

Effectiveness towards Other Types of I/O In our evaluation we mainly focus on the effectiveness of vSlicer in reducing network I/O processing latency. However, we point out that, by design, vSlicer is *not* specific to network I/O and can effectively reduce the processing latency of other types of I/O (e.g., disk I/O). For example, in Xen, disk I/O is handled similarly to network I/O: When an application requests a disk I/O, the request will go through the driver domain. However, if the requesting VM is not running when the disk I/O is completed, the VM may have to wait for multiple time slices before getting the CPU to process the I/O. vSlicer will give the VM more timely access to the CPU for processing the disk I/O.

7. RELATED WORK

Existing work that is most related to vSlicer has been discussed in Section 1. We now discuss other related efforts that belong to the general area of I/O performance improvement for virtualized environments. They fall into two broad categories: (1) reducing/analyzing virtualization overheads along the I/O path and (2) VM scheduling.

Reducing/Analyzing Virtualization Overheads In recent years, researchers have proposed various solutions to measure and alleviate virtualization-induced overheads along the I/O path. For instance, Chadha *et al.* present an execution-driven, simulation-based analysis methodology with symbol annotation to evaluate the performance of I/O virtualization [16]. Their methodology provides detailed architecture-level profiling information which will allow designers to evaluate the effectiveness of hardware-level enhancement for more efficient virtualized I/O. Menon *et al.* have proposed several optimizations to improve network device virtualization using techniques such as packet coalescing [32], scatter/gather I/O, checksum offload, segmentation offload [30], and offloading device driver functionality [31]. vSlicer is complementary to these techniques. The offloading technique improves network *throughput* but does not reduce the *application-perceived latency*. Packet coalescing may even increase the response time of a VM when sending one interrupt for several arriving packets. vSlicer alleviates this problem by differentiating the VMs as LSVMs and NLSVMs and satisfying their corresponding requirements for CPU time. XenSocket [38], XenLoop [36], Fido [15], and Xway [26] specialize in improving inter-VM communication within the same physical host. vFlood [18] and vSnoop [24] improve TCP *throughput* – but *not* application-perceived latency – between VMs in a datacenter. IVC [22] is another effort in this direction that targets high-performance computing platforms and applications. Lange *et al.* address how to minimize the overhead of virtualization for HPC via passthrough I/O [27], which enables direct guest/application access to a machine's specialized communication hardware. This in turn achieves both high bandwidth and low latency properties of that hardware.

VM Scheduling MRG [23] is a VM scheduler to improve the I/O performance of MapReduce on cloud servers. By exploiting the homogeneity of VM behaviors in MapReduce, MRG sorts the VMs in the CPU run queue based on their priorities as well as the pending I/O operations and batches the I/O operations from several VMs, hence reducing the context switch overhead. A two-level scheduling policy is proposed to achieve proportional fair sharing across both MapReduce clusters and individual VMs. MRG works well only when the guest VMs and the driver domain share the same CPU/core; and it is specific to MapReduce. On the other hand, vSlicer assumes that the driver domain is pinned to a separate core and it does not depend on specific applications' behaviors. Virtuoso's vSched [14] is a soft real-time scheduler based on a periodic real-time scheduling model. Like Xen SEDF, it is based on the earliest-deadline-first (EDF) policy; whereas vSlicer is adapted from Xen's credit scheduler based on the round-robin policy. vSched is a user-level program that runs on Linux and schedules type-II VMs [19] (e.g., VMware GSX) running as processes; whereas vSlicer is a hypervisor-level scheduler. The hybrid scheduling framework [37] combines two scheduling policies to meet the different requirements of high-throughput workload and concurrent processing workload. Its co-scheduling strategy may schedule all related vCPUs of a VM simultaneously to reduce the synchronization overhead and response time of multi-threaded applications. However, it requires that the number of vCPUs be no more than the number of physical CPUs.

8. CONCLUSION

We have presented vSlicer as a technique to support a new class of CPU-sharing VMs called LSVMs. LSVMs improve the performance of I/O-bound applications by reducing the I/O processing latency; yet they do not violate the CPU share fairness among all VMs sharing the same CPU. vSlicer is based on the idea of differentiated-frequency CPU microslicing, where the regular time slice for an LSVM is further divided into smaller microslices for scheduling the LSVM multiple times within each scheduling round. Therefore, the LSVM is given more frequent accesses to the CPU for timely processing of I/O events. vSlicer is simple and generic for

implementation in various hypervisors. Our evaluation of a Xen-based vSlicer prototype demonstrates significant improvement at both network I/O and application levels over Xen's credit scheduler.

9. REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[2] Gogrid Cloud. http://www.gogrid.com.

[3] Httperf. http://www.hpl.hp.com/research/linux/httperf/.

[4] Intel MPI benchmark. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[5] The Iperf Benchmark. http://www.noc.ucf.edu/Tools/Iperf/.

[6] J. McCalpin. The STREAM benchmark. http://www.cs.virginia.edu/stream/.

[7] Lookbusy-a synthetic load generator. http://www.devin.com/lookbusy/.

[8] MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.

[9] Myconnection Server. http://www.myconnectionserver.com/.

[10] Rackspace Cloud. http://www.rackspace.com.

[11] RFC 2326:Real Time Streaming Protocol (RTSP). http://rfc-ref.org/RFC-TEXTS/2326/chapter10.html.

[12] VMware ESX. http://www.vmware.com/products/esx/.

[13] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *ACM SOSP* (2003).

[14] Bin Li, P. A. D. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *ACM SC'05* (2005).

[15] Burtsev, A., Srinivasan, K., Radhakrishnan, P., Bairavasundaram, L. N., Voruganti, K., and Goodson, G. R. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *USENIX ATC* (2009).

[16] Chadha, V., Illikkal, R., Iyer, R., Moses, J., Newell, D., and Figueiredo, R. I/O Processing in a Virtualized Platform: A Simulation-Driven Approach. In *ACM VEE* (2007).

[17] Cherkasova, L., Gupta, D., and Vahdat, A. Comparison of the three cpu schedulers in xen. *SIGMETRICS Performormance Evaluation Review 35*, 2 (2007), 42–51.

[18] Gamage, S., Kangarlou, A., Kompella, R. R., and Xu, D. Opportunistic flooding to improve TCP transmit performance in virtualized clouds. In *ACM SOCC* (2011).

[19] Goldberg, R. Survey of virtual machine research. *IEEE Computer* (1974), 34–35.

[20] Govindan, S., Nath, A. R., Das, A., Urgaonkar, B., and Sivasubramaniam, A. Xen and Co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In *ACM VEE* (2007).

[21] Hu, Y., Long, X., Zhang, J., He, J., and Xia, L. I/O Scheduling Model of Virtual Machine Based on. In *ACM HPDC* (2010).

[22] Huang, Wei, Koop, J., M., Gao, Qi, Panda, and K., D. Virtual machine aware communication libraries for high performance computing. In *ACM/IEEE SC* (2007).

[23] Kang, H., Chen, Y., Wong, J. L., Sion, R., and Wu, J. Enhancement of Xen's scheduler for MapReduce workloads. In *ACM HPDC'11* (2011).

[24] Kangarlou, A., Gamage, S., Kompella, R. R., and Xu, D. vSnoop: Improving TCP throughput in virtualized environments via acknowledgement offload. In *ACM/IEEE SC* (2010).

[25] Kim, H., Lim, H., Jeong, J., Jo, H., and Lee, J. Task-aware virtual machine scheduling for i/o performance. In *ACM VEE* (2009).

[26] Kim, K., Kim, C., Jung, S.-I., Shin, H.-S., and Kim, J.-S. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In *ACM VEE* (2008).

[27] Lange, J. R., Pedretti, K., Dinda, P., Bridges, P. G., Bae, C., Soltero, P., and Merritt, A. Minimal-overhead virtualization of a large scale supercomputer. In *ACM VEE* (2011).

[28] Lee, M., Krishnakumar, A. S., Krishnan, P., Singh, N., and Yajnik, S. Supporting soft real-time tasks in the Xen hypervisor. In *ACM VEE* (2010).

[29] Leslie, I. M., D. Mcauley, R. B., T. Roscoe, P. T. B., D. Evers, R. F., and Hyden, E. The design and implementation of an operating system to support distributed multimedia applications. In *IEEE Journal of Selected Areas in Communications* (1996).

[30] Menon, A., Cox, A. L., and Zwaenepoel, W. Optimizing network virtualization in Xen. In *USENIX ATC* (2006).

[31] Menon, A., Schubert, S., and Zwaenepoel, W. TwinDrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers. In *ACM ASPLOS* (2009).

[32] Menon, A., and Zwaenepoel, W. Optimizing TCP receive performance. In *USENIX ATC* (2008).

[33] Nishiguchi, N. Evaluation and consideration of the credit scheduler for client virtualization. In *Xen Summit Asia 2008* (2008).

[34] Patnaik, D., Krishnakumar, A., Krishnan, P., Singh, N., and Yajnik, S. Performance implications of hosting enterprise telephony applications on virtualized multi-core platforms. Tech. rep., IPTComm, 2009.

[35] Waldspurger, C., and Rosenblum, M. I/O virtulization. In *Communications of the ACM* (2012).

[36] Wang, J., Wright, K.-L., and Gopalan, K. XenLoop: A transparent high performance inter-vm network loopback. In *ACM HPDC* (2008).

[37] Weng, C., Wang, Z., Li, M., and Lu, X. The hybrid scheduling framework for virtual machine systems. In *ACM VEE* (2009).

[38] Zhang, X., McIntosh, S., Rohatgi, P., and Griffin, J. L. XenSocket: A high-throughput interdomain transport for virtual machines. In *ACM/IFIP/USENIX Middleware* (2007).