Latency Slicing in Software-Defined Networks

Ramakrishnan Durairajan
University of Wisconsin, Madison

Srinivas Govindan
University of Wisconsin, Madison

Aaron Gember
University of Wisconsin, Madison

Aditya Akella
University of Wisconsin, Madison

Abstract

Recent advances in Software-Defined Networks have made it feasible to carve out multiple networks (slices) in the same physical infrastructure and even the same switch, with fair share of the physical resources among the slices. However, as more flows share the same switch, the switch latency experienced by each slice increases substantially, which translates into longer latency perceived by latency-sensitive applications. To mitigate such impact while retaining the benefit of switch sharing, we introduce a new class of flows called latency-sensitive flows (LSFs), which achieve better performance for latency-sensitive applications while maintaining the same resource share (and thus cost) as other flows sharing the switch. LSFs are enabled by microslotting, a technique that schedules each LSF more frequently but with a smaller micro time slot. Microslotting enables more timely processing of latency-sensitive flows by LSFs, without violating the switch share fairness among all sharing slices. Our evaluation of a Microslotting prototype in Open vSwitch [5] shows that microslotting substantially reduces network packet round-trip times and jitter and improves application-level performance.

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. Virtualization is a key technology that enables sharing of the underlying physical infrastructure and resources and gives each tenant in a data center (a slice) with its own network topology and control over the traffic [7]. While virtual machines are a standard abstraction for this, the right abstraction for the network is not yet defined [3]. While each slice is typically assigned at least one virtual switch, the mapping between virtual and physical switches is not always one-to-one. For example, in commercial cloud offerings such as Amazon EC2 [1], the network is designed to be a sharable entity (in the form of slices) for effective communication and the operation in such slices depends entirely on isolation at various levels. Existing isolation mechanisms provide isolation of slices at various levels (like physical, traffic and control), but latency is not guaranteed. For instance, consider a company K that rents resources from a cloud provider C. C must restrict access to Cs servers that manage intellectual information from non-employees of K; provide access to employees of K with certain privileges; provide access to a subset of customers (of K) with a specific latency guarantee and to another subset of customers (of K) with a different latency guarantee. All these should be isolated from other tenants of C (which has different latency requirements). We intend to solve this problem of guaranteeing latency along with isolation.

Unfortunately, a serious downside of switch sharing among multiple slices leads to significant negative impact on latency-sensitive applications running in those slices. In this paper, we especially address a key aspect of the impact: latency perceived by applications. More specifically, a slice with latency-sensitive flows will have to wait for its turn to access the switch. Because of the multiple sharing slices, the switch access latency tends to be a multiple of the default switch time slot for each slice (e.g., 30ms in Open vSwitch); and such latency cannot be hidden from the corresponding application. This impact is particularly harmful to latency-sensitive applications. For example, consider a simple VoIP gateway server which basically establishes and maintains connections between clients. For fast call setup and traffic relay, the gateways network latency dominates its computation (e.g., audio transcoding). With default switch time slots for the sharing slices, the slice that hosts the gateway may not be able to access the switch in time to process requests for new calls or traffic from ongoing calls. An-

other example is a low-volume web server that needs to quickly respond to client requests, yet its overall switch usage is relatively lower.

To avoid the impact on switch processing latency, one could choose to request a non-sharing slice that exclusively occupies a physical switch. 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 switch-sharing slices. More specifically, we introduce a new (sub)class of flow instances called latency-sensitive flows (LSFs), which will achieve better performance for latency-sensitive applications. Contrary to LSFs, we also define non-latency-sensitive slices (NSFs) for the execution of non-latency-sensitive applications that do not have stringent timing/latency requirement. LSFs and NSFs will share the same switch with fair share and similar cost; whereas the LSFs will achieve lower switch processing latency. One way to enable LSFs is to modify the switch's scheduler to prioritize certain latency-sensitive slices over the other slices. Unfortunately, it introduces short-term unfairness in switch shares. Moreover, this approach introduces heavy-weight and intrusive modifications to the scheduler, which involve tracking the slices switch usage and flow patterns.

In this paper, to mitigate such impacts while retaining the benefit of switch sharing, we introduce a new class of flows called latency-sensitive flows (LSFs), which achieve better performance for latency-sensitive applications while maintaining the same resource share (and thus cost) as other flows sharing the switch. LSFs are enabled by microslotting, a technique that schedules each LSF more frequently but with a smaller micro time slot. Microslotting enables more timely processing of latency-sensitive flows by LSFs, without violating the switch share fairness among all sharing slices. Our evaluation of a Microslotting prototype in Open vSwitch shows that microslotting substantially reduces network packet round-trip times and jitter and improves application-level performance.

The rest of the paper is organized as follows. We explain our problem statement in Section 2 followed by the motivation in Section 3. In section 4, we describe the design of Microslotting. Then we present our implementation in section 5 followed by the evaluation in section 6. We discuss the future work and availability of the source in section 7 followed by conclusion in section 8.

2 Problem Statement

SDN allows for slicing the network to provide isolated control over different physical and traffic subsets. Given that, how do we integrate low-latency guarantees along with physical, control and traffic isolations?

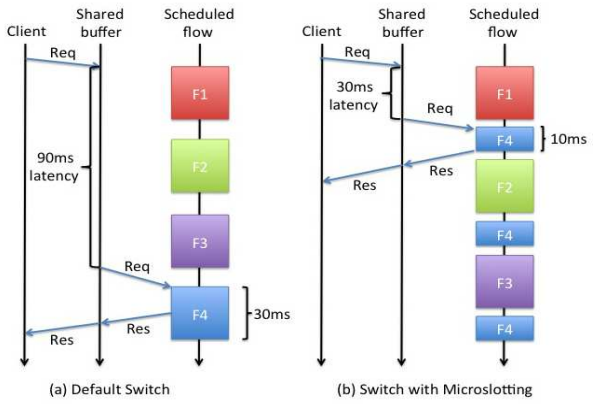

Figure 1: Comparison of default switch and switch with Microslotting

3 Motivation

Traditional state of the art networks provides isolation by various mechanisms. 1) VLANs [2] provide a way to separate the processing of different classes of packets in the network. VLANs provide traffic isolation, but add overhead to the already difficult task of writing network configurations and hence does not guarantee performance, 2) firewalls prevent certain packets from flowing onto certain parts of the network and provides physical isolation but requires deploying them at appropriate locations, 3) mechanism shown in splendid isolation [4] argues that instead of relying on low level mechanisms (as described above), isolation must be provided at the level of programming slices. Though this approach provides all the isolation levels, it does not give a global view of the underlying network as each tenant runs their own controller and hence latency cannot be guaranteed. So we need a centralized virtualization layer like FlowVisor [8] that operate on slices with isolation levels. FlowVisor provides bandwidth guarantee at a coarse-grained manner by setting the PCP bit of each VLAN corresponding to each slice but it doesnt provide latency guarantee.

3.1 Impact of switch sharing

To understand the negative impact of slice sharing a switch on the latency, consider the example shown in Figure 1. Assume four slices are sharing a physical switch. Slice4 is hosting a latency-sensitive application while slice1-slice3 are hosting bandwidth-sensitive applications. The application in slice4 waits for client requests and then responds to the requests with data or control messages. This simple communication pattern can be found in many applications such as web servers, VoIP

proxies, and MPI jobs. We assume that the slice scheduler in the flowvisor uses a proportional-share scheduling policy. Since each slice has a runnable task in it, it occupies the entire switch time slot allotted to it. When a request for slice4 arrives at the physical host, it needs to be buffered outside slice4 (e.g., in the sliceM or in the privileged driver domain), until slice4 is scheduled to run. When slice4 gets scheduled, it will process the request and generate a response. Assuming a switch time slot of 30ms, the request response latency can be as high as 90ms (i.e. (Number of sharing slices -1) * Time Slice). Such a high latency hampers the responsiveness (and consequently, request processing rate) of the application in slice4.

3.2 Problems with Alternative Solutions

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

Prioritize latency-sensitive slices: The first option to reduce the above switch processing latency is to prioritize the slices running the latency-sensitive applications. In fact, this mechanism works quite well for pure latency-sensitive slices. However, in the presence of heterogeneous workloads, once the slice gets scheduled to process the flow request by this mechanism, it will consume its switch share due to the switch bound segment of the workload. This will effectively disable the priority for the rest of this scheduling cycle resulting higher switch latencies. In other words, while priority can temporarily cede the switch to latency-sensitive slices, it can often lead to exhausting the slices credits early enough, and then, it may starve for the rest of the scheduling round (since the credit scheduler is switch-fair across slices).

Workaround: A naive workaround to this would be aggressively boosting the latency-sensitive slices without considering its switch share. Unfortunately, this prioritization will break the overall switch fairness in the system. For instance, consider a switch where slice1 is hosting a latency-sensitive application with a network-intensive task and a computation task whereas other slices are hosting computation-intensive applications. The incoming packets to slice1 trigger the boosting of slice1 so that it can process the packets. However, since the switch does not preempt a scheduled slice as long as the slice has runnable tasks, the computation portion of the application in slice1 will consume the rest of the time slice after the packet processing is done. This causes switch-time deprivation of other slices, as long as packets destined to slice1 keep arriving, compromising the switch fairness of the overall system.

Reduce time slot for all slices: The third option is to uniformly reduce the time slice size of the scheduler so that all the sharing slices will get scheduled in and out more frequently, resulting in shorter switch access latency. However, such an option would increase the number of context switches (and cache flushes) in the system, degrading the performance of other applications running in the switch.

4 Design

The previous section suggests that, if the switch-sharing slices are scheduled in a strictly round-robin fashion, it will be difficult to reduce the latency without hurting the performance of NSFs. On the other hand, prioritizing the LSFs may violate the switch share fairness among all slices. To address this dilemma, we borrow the following key idea from vSlicer [9] and apply it for switches. We call it Microslotting: Within one scheduling round, the switch time for an LSF 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., microslots) is equal to a standard switch time slot. Such a high-frequency microslotting will give more opportunities to the LSFs to process pending packets; yet it does not affect/preempt the regular time slots allocated to the NSFs. This ensures timely processing of packets while maintaining fair share of the switch among all slices. Right-side of Figure 1 explains the Microslotting for the same example in section 3.1.

For the purely latency-sensitive applications, as demonstrated in Section 2, there is a strong incentive not to run them in LSFs because the higher-frequency microslotting will cause more frequent cache flushes which will hurt application performance. Fortunately, the NSFs under our mechanism will give these applications the same performance as if running them in round-robin-scheduled slices with the default time slice.

4.1 Microslotting scheduling model

The idea of microslotting itself is quite general; one could pick any size for the microslot and simply derive the scheduling frequency. There are two main concerns one needs to keep in mind though. First, setting the microslot 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 LSFs, depends on the number of LSFs and NSFs sharing a switch. In practice, we expect only a small number (5) that share a switch, and even among these, the number of LSFs is going to be very small (≤ 2).

We use the following approach to determine the scheduling order in one scheduling round. Assume m LSFs and n NSFs are sharing a single switch. We denote

the scheduling period (i.e., scheduling round) by $T_{default}$ and the total time an LSF executes during a scheduling period as T_{LSF}. Similarly, the total time an NSF executes during a scheduling period is T_{NSF} . We want T_{NSF} to be a fairly large value to allow each non-latency sensitive slice to execute sufficiently long. Since we aim to fairly allocate the CPU among all the slices (both LSFs and NSFs), we want the following to hold:

$$T_{NSF} = T_{LSF} \tag{1}$$

Let us denote the time period where one (micro-)round of LSFs are scheduled after scheduling an NSF as T_S . Microslotting runs all the LSFs during T_S in round robin fashion. We want to further divide T_S into micro time slots 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 microslotting later in this section. Depending on the selection of T_m, an LSF can run one or more times during a single time slot T_S. Let us denote the total time the i^{th} LSF runs during T_S as T_{n_i}.

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

Suppose the i^{th} LSF can get scheduled r_i times during T_S. We have:

$$T_{n_i} = r_i * T_m \; where \; r_i \geq 1 \tag{3}$$

In this project, we assume all the LSFs 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

$$T_n = r * T_m \; where \; r \geq 1 \tag{4}$$

and

$$T_s = m * T_n \tag{5}$$

Given Microslottings alternating scheduling of LSFs and NSFs (i.e., it schedules a round of all LSFs followed by one of the NSFs), the total time that an LSF executes during a scheduling period $T_{default}$ is equal to the number of NSFs multiplied by the time an LSF executes during a time slice T_S (i.e. T_n). That is:

$$T_{LSF} = n * T_n \tag{6}$$

A scheduling period consists of running times of all LSFs and NSFs and therefore we get:

$$T_P = m * T_{LSF} + n * T_{NSF} \tag{7}$$

Using (6) in (7), we get:

$$T_P = m * (n * T_n) + n * T_{NSF} \tag{8}$$

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

$$T_P = n * T_S + n * T_{NSF} \tag{9}$$

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

$$TP = m * T_{NSF} + n * T_{NSF} = (m+n) * T_{NSF} \tag{10}$$

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

$$n * T_S + n * T_{NSF} = (m+n) * T_{NSF} \tag{11}$$

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

$$T_S = \frac{m * T_{NSF}}{n} \tag{12}$$

$$T_n = \frac{T_{NSF}}{n} \tag{13}$$

$$T_m r = \frac{T_{NSF}}{n} \tag{14}$$

As mentioned earlier, the selection of T_m depends on the desired scheduling latency of the LSF. Equation (14) defines the product of T_m and r in terms of T_{NSF} and n. The only restriction for the selection of T_m is, it should be a whole divisor of T_{NSF} . However, selecting a too small n value for T_m will increase the number of context switches during T_S, affecting the performance of the all LSFs. Let us denote the required latency for an LSF during T_S as T_l. To achieve this scheduling latency we should schedule the i^{th} slice within T_l. Since we schedule all the LSFs in a round-robin order, all the other (m ? 1) LSFs should be executed in less than T_l. That is:

$$(m-1) * T_m \leq T_l \tag{15}$$

Equation (15) gives us the upper bound for T_m.

Examples We now show two examples of scheduling sequence under two different settings. Figure 2 illustrates two scheduling sequences for a system running four slices. In Figure X, we have one LSF and three NSFs. If all these slices were scheduled by the default scheduler, any of them would experience a 90ms scheduling latency. Under microslotting, by dividing the time slice of the LSF (i.e. VM1) to multiple microslots and scheduling it three times during the scheduling round, the latency drops to 30ms.

In our discussion towards the end of Section 3, we emphasized that reducing the time slots uniformly for all

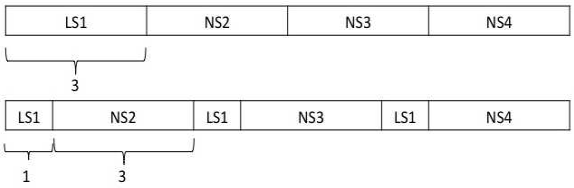

Figure 2: Comparison of scheduling in default and microslotting switches

sharing slices is not a desirable option, primarily due to the increased context switches between the slices. Now that we have discussed the details of microslotting, let us quantitatively compare the default scheduler with uniformly reduced time slice with microslotting using a system with two LSVMs and two NLSVMs. With the default scheduler having the default time slice, the switch access latency of each LSF is (m+n-1)*T_{NSF}. Here it is (4-1)*T_{NSF} = 3 * 30 = 90ms. In order to reduce the latency to 15ms, we need to reduce the time slot from 30ms to 5ms, which will make the context switch rate increase by 6 x. With microslotting, however, setting T_m = 5ms, to achieve 15ms average latency, would increase the number of context switches only by 3 x.

5 Implementation

In Open vSwitch, a virtual switch moves layer-2 packets between two or more network ports. In the software, it is represented by an object called *datapath*, which contains a list of ports used by the virtual switch, and a *flow table* that associates a specific action to each flow. Each packet flowing through the switch is compared against the entries in the flow table, and in case of a match the corresponding action is executed.

The software architecture of Open vSwitch puts a clear separation between the controller, in charge of managing ports and computing the content of the flow tables, and the datapath, which does the actual packet switching. The two components can run on different systems, and, as a consequence, the datapath can be implemented with various technology, from software to hardware.

The software architecture of OpenvSwitch is represented in Figure 3, which includes two options for the datapath. One, on the left side, indicates a pure user space implementation. On the right side, a kernel implementation.

The original Open vSwitch code is written for Linux in portable C code, though there are obvious system dependencies when it comes to access specific kernel modules, or even APIs to send and receive packets.

The upper layers of the user space code compile clearly on FreeBSD and are:

Figure 3: Open vSwitch architecture

- the *ofproto library*, implementing the core of the openflow protocol [6];
- clients using the *ofproto* library (ovs-vswitchd, ovs-openflowd).

The lower layers handle the communication with the network devices provided by the kernel, so they are more platform dependent.

The user space implementation is based on a device abstraction, called *netdev*, that implements the API functions to communicates with the network devices provided by the kernel. The kernel space implementation implements the flow table directly into the kernel, allowing a faster processing of traffic compared to the userspace version.

We modify the following components in Open vSwitch:

User space changes: At *Ofproto* inform the latency to the Open vSwitch's kernel before adding the flow to the flow table. These latencies where added using "/sys" from user space.

Kernel space changes: At *Datapath* a separate worker thread is used for microslotting. This accesses "/sys" file that was written in the user space. At VPort, the micro LSFs and NSFs are interleaved.

6 Evaluation

In this section, we present our detailed evaluation of Microslotting using the Open vSwitch prototype. Our experiments evaluate two key aspects: (a) transport-level latency reduction achieved by microslotting; (b) overall switch-sharing fairness with Microslotting; and (c) application-level performance improvement by microslotting.

Experimental Setup: Our experiments involve evaluating our modified version of Open vSwitch along with

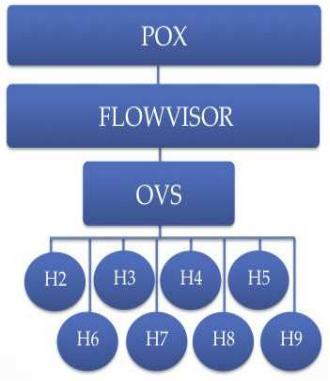

Figure 4: Tesbed used to evaluate Microslotting

unmodified version to quantify the difference in latencies and bandwidth. The testbed used in our evaluation is shown in Figure 4. We plan to run a number of experiments by varying the number of slices per switch. The difference in latency per flow should increase with increase in number of slices per switch especially for small flows. The amount of bandwidth over-provisioning will determine the extent to which the bandwidth guarantee is satisfied.

7 Future work and Availability

Both the modified kernel of Open vSwitch as well the complete report base can be downloaded from:

http://pages.cs.wisc.edu/~govindan/openvswitch-impl-final.tar.gz

Future work:

- Since a switch is just the first level of abstraction, we have modified things at the switch level. However we wanted to investigate if we can achieve end-to-end latency guarantee? We wanted to leverage the circuit-pusher explained in the next point to achieve this.

- Circuit-pusher (Floodlight) - Circuit Pusher app utilizes floodlight rest APIs to create a bidirectional circuit, i.e., permanent flow entry, on all switches in route between two devices based on IP addresses with specified priority. We think combining Microslotting with circuit pusher will result in low end-to-end latency.

- Dynamic route selection - Can we combine Microslotting, circuit-pusher and create a mechanism to dynamically select routes.

- We have implemented our mechanism and set latency using pad fields in the open flow. We hope there is a field in openflow protocol to set these values. This could avoid things like access from "/sys" file and further reduce latency.

- It is very clear that what we have is a very base version. This could be enhanced in terms of implementation, protocol modification, and testbed evaluation.

8 Conclusion

We have presented Microslotting as a technique to support a new class of switch-sharing flows called LSFs. LSFs improve the performance of latency-sensitive applications by reducing the packet processing latency; yet they do not violate the switch share fairness among all slices (and indeed flows) sharing the same switch. Microslotting is based on the idea of differentiated-frequency switch micro-slotting, where the regular time slot for an LSF is further divided into smaller microslots for scheduling the LSF multiple times within each scheduling round. Therefore, the LSF is given more frequent accesses to the switch for timely processing of latency-sensitive packets. Microslotting is simple and generic for implementation in various switches. Our evaluation of a Open vSwitch based Microslotting prototype demonstrates significant improvement at both network I/O and application levels over default switches.

9 Acknowledgments

We would like to thank Prof. Aditya Akella and Aaron Gember for the guidance and support throughout the project.

References

[1] AMAZON ELASTIC COMPUTE CLOUD. http://aws.amazon.com/ec2/.

[2] COMMITTEE, L. S. Ieee802.1q - ieee standard for local and metropolitan area networks virtual bridged local area networks. In *IEEE Computer Society* (2005).

[3] DRUTSKOY, D., KELLER, E., AND REXFORD, J. Scalable network virtualization in software-defined networks. In *In Proceedings of IEEE Internet Computing* (2012).

[4] GUTZ, S., STORY, A., SCHLESINGER, C., AND FOSTER, N. Splendid isolation: a slice abstraction for software-defined networks. In *In Proceedings of the first workshop on Hot topics in software defined networks* (2012).

[5] OPEN VSWITCH: AN OPEN VIRTUAL SWITCH. http://openvswitch.org.

[6] OPENFLOW. http://www.openflow.org.

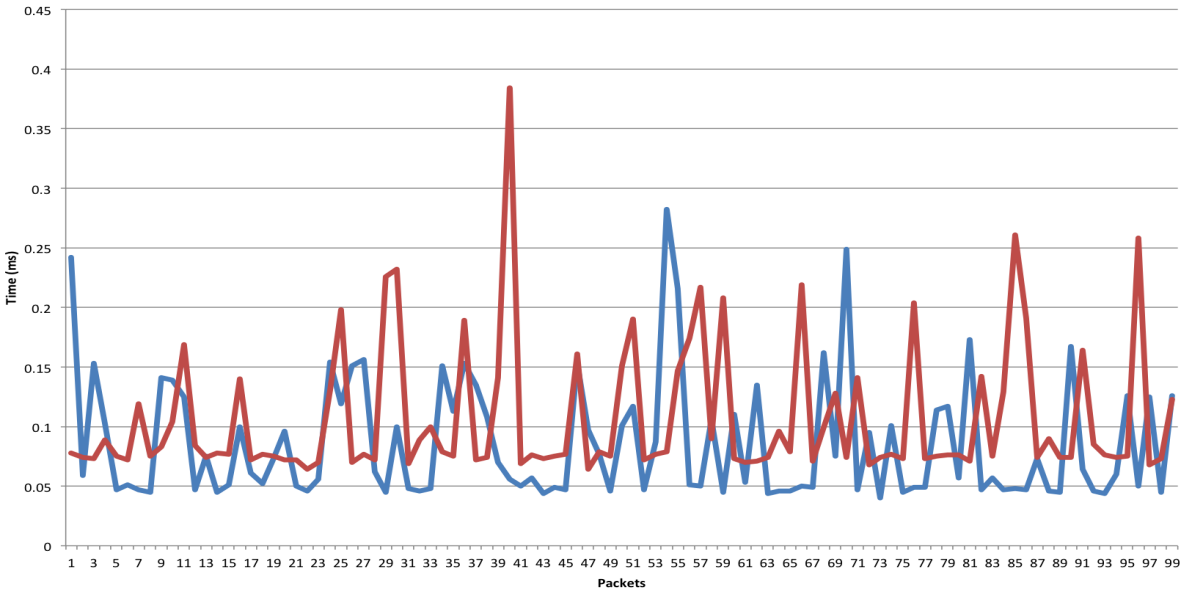

Figure 5: Initial results

[7] SHERWOOD, R., GIBB, G., YAP, K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the production network be the testbed? In *In Proceedings of the 9th USENIX conference on Operating systems design and implementation, 2010* (2010).

[8] SHERWOOD, R., GIBB, G., YAP, K., APPENZELLER, G., MCKEOWN, N., AND PARULKAR, G. Flowvisor: A network virtualization layer. In *Technical Report* (2009).

[9] XU, C., GAMAGE, S., RAO, P., KANGARLOU, A., KOMPELLA, R. R., AND XU, D. vslicer: Latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Technical Report* (2012).