Predictive Queue Management in SDNs

Riccardo Mutschlechner, Stephen Sturdevant, Kausik Subramanian

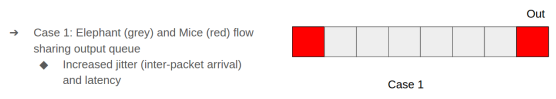
1 Abstract

Datacenters see diverse network workloads with different objectives. Primarily, traffic can be classified into two kinds of flows: high-throughput flows (elephant) and low-throughput, latency-sensitive (mouse) flows. Many applications have a multi-layer partition/aggregate pattern workflow where a task at each layer is partitioned into multiple tasks and provided to the workers at the next level. To provide some guarantees, workers will typically be assigned tight deadlines. Network delays can lead to missed deadlines, causing deteriorated performance. Thus, one of the key factors to application performance in datacenters is to provide low latency guarantees to the latency-sensitive flows. In this project, we try to address these issues in a software-defined datacenter network by building a predictive models of how switch queues build up. This is aided by OpenFlow's support for querying switches for flow statistics to accurate predict flow characteristics. With a predictive model in place, we develop a greedy earliest-deadline-first scheduling for finding new paths for flows to prevent queue buildups affecting mice flows. We present a best-case performance evaluation of our scheduling, which demonstrates the usefulness of managing queues effectively in datacenters.

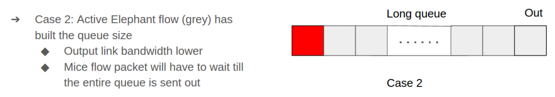
2 Motivation

To frame our problem more specifically, we define two notions of flows: mice flows, and elephant flows. Mice flows are low-throughput, latency-sensitive flows which are generally used for distributed applications and are tightly coupled with application performance. These types of flows have very low tolerance to loss and latency. Secondly, we have elephant flows, which are high-throughput, latency-tolerant flows. These flows are generally used for data replication and MapReduce style jobs.

When elephant and mice flows traverse the same queue, the major reason for increased latency is queue buildups. This is very problematic for the latency-intolerant mice flows. There are two primary cases shown in Figure 1 in which queue buildup severely increases the latency of mice flows. In the first case, the mice flows experience increased latency as they are intermixed with the elephant flows in queue, getting an unfair share of transmission time as well as high latency and jitter. In the second case, the mice flows are being enqueued at the tail of a queue which is already completely backed up by existing elephant flows. This is also very problematic, as the mice flow packets will either be very highly delayed or dropped altogether. Thus, there is a need to classify flows and monitor switch queues to detect queue buildups in real-time. When we detect queue buildups, we need to dynamically schedule flows on different routes based on their classification to ensure low latencies.


Case 1: Elephant (grey) and Mice (red) flow sharing output queue
Increased jitter (inter-packet arrival) and latency
Out
Case 1

(a) Case 1


Case 2: Active Elephant flow (grey) has built the queue size
Output link bandwidth lower
Mice flow packet will have to wait till the entire queue is sent out
Long queue
Out
Case 2

(b) Case 2

Fig. 1: Cases of Queue Buildup

The prominence of software-defined networks has increased in recent times, and offers a centralized control of the network. The SDN controller has a global view of the network (such as topology and routing information) and can collect any relevant statistics (such as link utilization and traffic matrix). Using the resources available in a typical SDN setup, we believe there exist the means to schedule flows in real-time to avoid the problem of mice and elephant flow conflicts. Thus, we ask the following question: *How can we perform dynamic flow scheduling to minimize latency of the mice flows in a software-defined datacenter?*

3 Related Work

Alizadeh et. al [3] show that long-lived, greedy TCP flows cause the length of the bottleneck queue to grow until packets are dropped, resulting in the familiar sawtooth pattern in buffer-lengths. Queue buildup is also a major factor for increasing latencies observed in datacenters. To minimize queue buildup, they develop a modified TCP, DCTCP which uses the queue statistics at switches for congestion control, nearly achieving zero queue buildup.

Flow scheduling in datacenters has been an active area of research in recent times. Hedera [2] performs scheduling of flows by estimating TCP bandwidth requirements and dynamic flow scheduling using different placement algorithms (most importantly simulated annealing which is a probabilistic search technique for paths) to achieve full bisection bandwidth requirements. On the other hand, we focus on the problem of dynamically scheduling flows in terms of latency-sensitivity (mice flows, unlike elephant flows, are highly sensitive to latency), which also requires monitoring queue sizes at switches in real-time and a dynamic path placement with the objective of decreasing latency. Hedera does not focus on the aspect of latency in its flow-scheduling algorithm. This also reduces the timescales of scheduling decisions, and requires the need to devise real-time flow-placement algorithms which need not be optimal.

Fastpass [7] presents a datacenter network architecture where the time of packet transmissions and the path traversed by packets are determined by a centralized arbiter. This approach uses a timeslot allocation algorithm to determine when the endpoints packets should be sent and a path assignment algorithm to determine paths to eliminate congestion at switches. The centralized arbiter ensures zero queue buildup in the switches by globally scheduling each packet, and thus flows do not suffer queueing delays. However, there is a throughput penalty for achieving very low queueing delays, as some links would have to be idle.

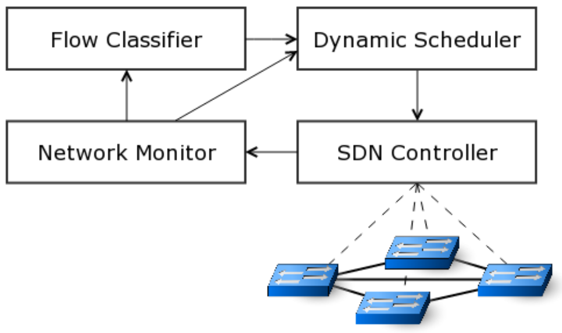


Fig. 2: System architecture

Crovella et. al [5] examined a commonly used service, Apache; specifically, its routing policies. The paper points out that Apache does not prioritize shorter requests versus longer requests. The authors then examine the potential benefits of different queueing algorithms that prefer shortest-connection first, whose principles can be used in the context of flow-scheduling. The paper mentions that web server workloads are primarily tail-heavy, which may work to our benefit in the sense that elephant flows will not pay much of a penalty relative to the gains that mice flows will see.

4 System Architecture

Figure 2 displays the system architecture of the scheduler used for predictive queue management in software-defined networks. The four components of the system are as follows:

- **SDN Controller**: The centralized point of control of the network. The advantages of using

SDNs for predictive queue management are twofold: (1) OpenFlow has support for centralized monitoring of switches for different statistics, thus eliminating the need for specialized boxes for measurement and developing clever monitoring algorithms to minimize load on switches due to querying statistics (2) The controller can add/modify forwarding rules at switches, thus providing a centralized point to decide routing of flows, which is received as output from the dynamic scheduler to prevent queue buildups.

- **Network Monitor**: Based on the monitoring algorithm used at the controller, statistics from the entire network topology is collected and passed to the network monitor, which store the necessary statistics needed to predict queue buildups in switches.

- **Flow Classifier**: Using the characteristics gathered by the network monitor, the flow classifier will classify the flow as mice/elephant based on some threshold (can be user-defined)

- **Dynamic Scheduler**: The scheduler predicts queue buildups using the characteristics from the network monitor, and dynamically schedules paths for flows to prevent queue buildups affecting latency-sensitive mice flows using an earliest-deadline-first greedy scheduling algorithm to find new paths, which are sent to the controller to be deployed to the network.

We describe the components in detail in the following sections.

5 Flow Characteristics

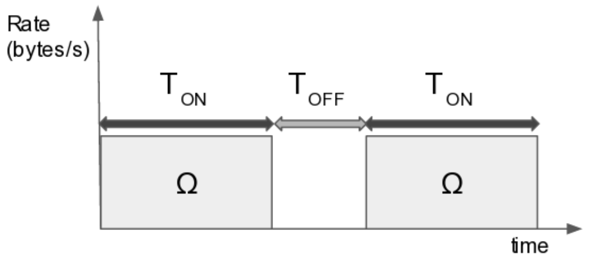


Fig. 3: Flow Characteristic Model

To predict how queue builds up, we need to develop a model of a flow such that we can predict the data it sends in any duration of time. We characterize each flow as following an repeating ON/OFF model [4] as shown in Figure 3. The three main parameters are:

1. **Active time(T_{ON})**: The duration of time for which the flow is actively transmitting data

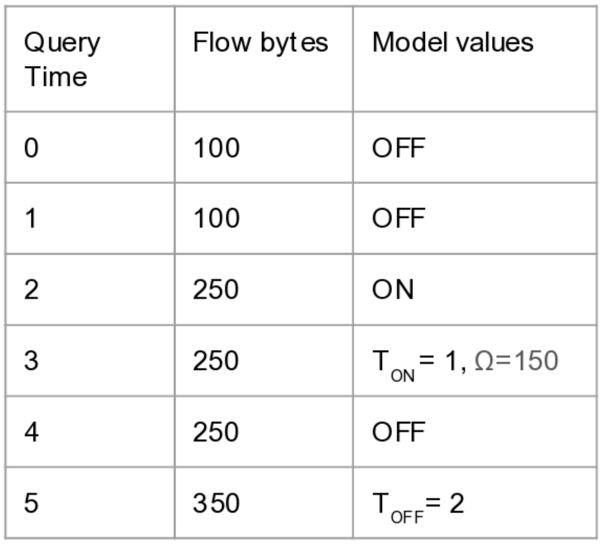
2. **Inactive time(T_{OFF})**: The duration of time for which the flow is inactive between two transmission cycles

3. **Transmission Bytes(Ω)**: The total number of bytes sent in a single transmission period (the rate is assumed to be constant in the period)

By maintaining a model for a flow, we can use this to easily estimate the number of bytes sent by this flow in a time interval. The queue buildup model can be built by using this information from all flows going through the queue. Development of sophisticated flow characteristic models based on specific distributions is a direction of future work.

5.1 Measurement

We use OpenFlow's support for obtaining flow statistics from switches to estimate the flow characteristic parameters. An example measurement trace for a flow is shown in Figure 4. In every t_{meas}, the edge switch where the flow enters the network, is queried



Query Time	Flow bytes	Model values
0	100	OFF
1	100	OFF
2	250	ON
3	250	$T_{ON} = 1$, Ω=150
4	250	OFF
5	350	$T_{OFF} = 2$

Fig. 4: Example measurement trace for a flow. An interval where the total bytes change is marked as an ON period, and conversely, an OFF period means the reading is constant.

for the statistics of the flow, which returns the cumulative bytes received at the switch. We maintain a expotential moving average(EWMA) for each parameter, and update the average from each reading.

The main advantages of this approach is that by measuring flow characteristics, we do not need cooperation from the tenants to provide information about flows (which may be difficult in public clouds and can be misused by malicious tenants). The major shortcoming of this approach is that the least count of T_{ON} and T_{OFF} is the query time interval t_{meas} (we cannot say anything about a time inside a query interval). Thus, it is a crucial trade-off between accuracy of the flow characteristics and the load on switches due to frequent querying.

5.2 Path Characteristic

While we measure the characteristic of the flow at the source edge switch, the flow characteristics would change as the flow traverses through the network. Let us consider a single switch. If a flow sent data at a rate greater than the output bandwidth, the rate of the output flow would be truncated by the output capacity. If there is no losses at switch, then the output Ω would remain constant. Using this, we can find the output T_{ON} and T_{OFF}, which would be different from the input characteristic. However, instead of measuring this at each switch (which would lead to increased load of querying), we predict the path characteristics of the flow as it flows through the network.

For estimating the path characteristics of the flow, we assume two things: (1) No losses at the switch- our predictions would be more conservative than if there were losses (2) No fair queuing at switches- the output bandwidth is alloted in proportion to the input rates of the flows, so if a flow sends a higher rate, then it would get a larger share of the output bandwidth.

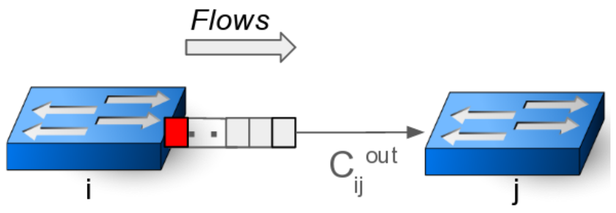


Fig. 5: Illustration of a switch queue model

Consider two switch sw_i and sw_j(Figure 5) and let the out link capacity connecting $sw_i \rightarrow sw_j$ to be C_{ij}^{out}. Let $Flows$ denote the set of flows traversing $sw_i \rightarrow sw_j$. Consider a flow f at sw_i with characteristics Ω, T_{ON}, T_{OFF}. The output characteristics $\Omega', T'_{ON}, T'_{OFF}$ are described as follows:

$$\Omega' = \Omega \tag{1}$$

The output rate is proportional to the proportion of input data sent by f. The data sent by f in a active period is Ω. Let $\Omega(t)$ denote the bytes sent by the flow in time t. We decide the proportion by the amount of data sent by other flows sent in T_{ON} and divide the output capacity using this proportion. Thus, the estimated output rate R' is defined as follows:

$$R' = min(\frac{\Omega}{T_{ON}}, \quad C_{ij}^{out} \times \frac{\Omega}{\sum_{f \in Flows} \Omega_f(T_{ON})}) \tag{2}$$

The *min* operation is to ensure that the output rate does not increase the input rate, the flow cannot accelerate at any point in the path. With the estimated output rate, we can find the other parameters of the characteristic. As the rate will decrease, the active time will increase to sent the Ω bytes.

$$T'_{ON} = \frac{\Omega}{R'} \tag{3}$$

The inactive times will reduce because of the increase in active times.

$$T'_{OFF} = T_{OFF} - (T'_{ON} - T_{ON}) \tag{4}$$

Thus, by measuring flow statistics and using these equations, we have a comprehensive picture of characteristics of flows on all switches of the topology. In the next section, we explain how we use these to predict queue buildups.

5.3 Flow Classifier

In our current iteration, we have defined two set of classes (elephant and mice), and defined a configurable threshold on Ω and T_{ON} to classify a flow. However, there can exist multiple classes of flows with different priorities, and the scheduling can be modified to take into account the priorities.

6 Queue Buildup Model

As discussed earlier, latency-sensitive mice flows are affected when a switch queue increases beyond some threshold. Instead of reactively responding to these events, which may be difficult, we predictively try to ensure that mice flows are not affected by these queue buildups. To perform this, we need a model of queue builds up. Using the same example in Figure 5, we define $Q_{ij}(t)$ as the function of queue size of $sw_i \rightarrow sw_j$ over time t as:

$$Q_{ij}(t) = Q_{ij}(t=0) - C^{out}_{ij} \times t + \sum_{f \in Flows} \Omega_f(t) \tag{5}$$

The intuition behind the above equation is that increase is queue size from initial queue size ($Q_{ij}(t = 0)$) is the difference of bytes received in the queue ($\sum_{f \in Flows} \Omega_f(t)$) and the bytes send out the queue ($C^{out}_{ij} \times t$).

OpenFlow has no support for querying current queue size in the switch, thus there is no direct way to measure the queue size at any point of time. So, to predict initial size, we query flow statistics for $Flows$ at sw_i and sw_j, and the difference between the statistics is a good estimate of current queue size.

Finally, we define the critical time t_c for a switch when $Q_{ij}(t_c) > threshold$ such that there are mice flows affected by the queue buildup. This threshold can be defined based on latency limits or can be set to total queue size to denote congestion events. The critical threshold can also depend on the number of mice flows being affected (can ignore small number of mice flows being affected in large networks).

7 Scheduling Algorithm

By building a predictive model of queues, we can manage the flows effectively and ahead of time, instead of reacting to failures or higher latencies. Another advantage of this approach is that we can run the scheduler at larger epochs, thus reducing the overload of scheduling.

Let us formally define the objectives of scheduling. The input to the scheduler is the set of flows and their paths along the network, with the estimated characteristics for each flow. Also, using the queue buildup model of each switch queue and certain user-defined objectives of critical threshold, we can obtain the critical times for each of switches, if the flows maintain their current paths.

The scheduler is invoked in every t_{sched} epoch, and uses a earliest-deadline-first algorithm, which basically means we only consider switches with critical times $< t_{sched}$ within the current time. The rationale behind this is that these switches will be critical before the scheduler is invoked again, so require attention in this iteration.

Once we have identified the switches in critical mode this epoch, the next step of the scheduling algorithm is to reroute certain flows traversing these switches along paths such that the switches will not be critical in this epoch. For this, we use a greedy depth-first search in the network to find suitable reroute paths for flows. For choosing the flows to reroute, we choose elephant flows over mice flows, as rerouting a single elephant can dramatically change the critical time at the switch. Also to prevent route-flapping, we do not choose flows which was rerouted in the previous two iterations of the scheduler.

Once we choose a flow, we now decide a new route for this flow. We explore the network from the source switch in a depth-first manner such that we recompute the critical time for the switch if this flow was also added along the switch. If a switch is or becomes critical within this epoch, we do not explore this switch further (*unfeasible* path). This search is greedy because we will terminate once we find a path from source to destination (or leave the flow's path as it is). This way, we always do better than a naive scheduling algorithm which does not take into consideration queue buildups.

8 Evaluation

Initially, we intended to use the POX SDN controller and Mininet to emulate a datacenter network. Unfortunately, this does not provide a good way to measure latency due to high varience in measurements. Instead, we simulated a workload with a tail heavy distribution similar to the one described in [3] and estimated how many critical events our scheduling algorithm could prevent in comparison to doing no scheduling at all.

8.1 Simulation Framework

Thus, to evaluate the performance improvement of predictive queue management scheduling, we decided to pursue a simulation approach. One of the available simulators for software-defined networks *fs-sdn* [6] had the limitations that it did not simulate queueing delays, and thus could not used to evaluate our system.

Thus, to provide a best-case analysis of our algorithm,we built a flow-based simulator which simulates flows and its paths in the network, instead of individual packets. The simulator can be configured for network topology, total number of flows, and the composition of elephant and mice flows. For these experiments, we provide exact flow characteristics to the scheduler for the edge switches.

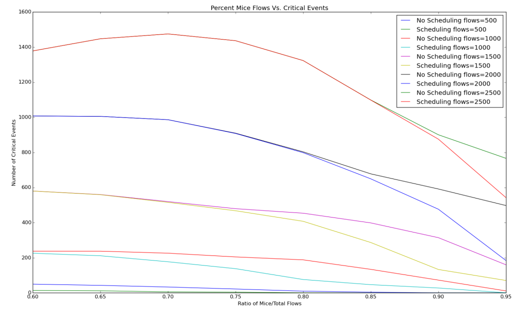


Fig. 6: Plot of number of critical events observed in scheduling and no scheduling for varying numbers of flows and ratios between mice and total flows. For each flow and ratio value, scheduling consistently results in either an equal or smaller number of critical events.

The flows are randomly distributed across endpoints, and the flows are sent through the shortest path between their endpoints. Since we know exactly the characteristics of each flow, the simulator can accurately count the number of mice flows affected by queue buildups(critical events) at some point of time if the shortest path allocation was followed. After this, we run a single iteration of our scheduler from this configuration, and count the number of flows affected after the new paths are deployed, decide by the greedy EDF search.

8.2 Results

We used the Fat-Tree topology [1] as a representative datacenter network with the hosts connected to the edges. In our simulations, we looked at two different Fat-Tree topologies. The first one has 20 nodes, and would represent a small datacenter. The other has 45 nodes, representing a medium-sized datacenter. In 6 we look at how the number of critical events

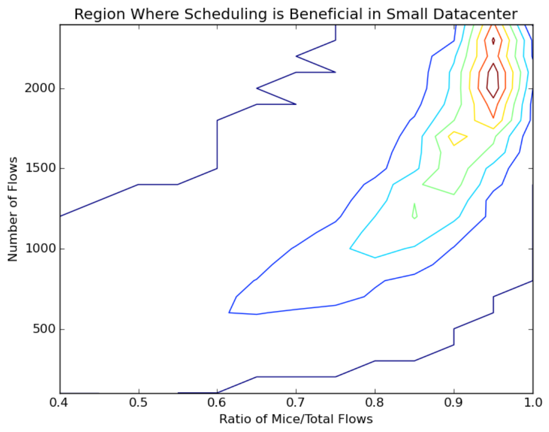


Fig. 7: Contour plot for small-sized datacenter. Blue indicates small difference between the number of critical events observed from no scheduling and with scheduling, red indicates a large difference.

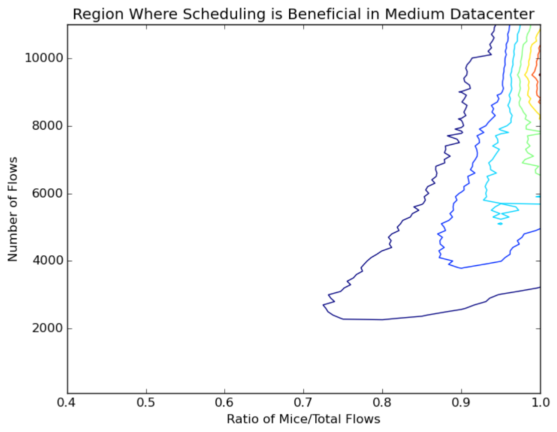


Fig. 8: Contour plot for medium-sized datacenter. Blue indicates small difference between the number of critical events observed from no scheduling and with scheduling, red indicates a large difference.

varies by the number of total flows and by the ratio of mice to total flows for both scheduling and without scheduling. These are results we observed in the small datacenter simulation, although the results for the medium datacenter are similar. As expected, the number of critical events increases with the number of flows in both cases, but with scheduling, we consistently perform at least as good if not better for each value of flow and ratio. Unexpectedly, the number of critical events does not seem to strictly decrease as the ratio of mice flows increases. For example, in the case of 2500 flows, the maximum occurs at a ratio of 0.7. We suspect that the reason for this comes from the fact that at lower ratios of mice flows, there are fewer total mice flows, and thus, fewer which could possibly be affected by a critical event. Although, it is still possible that this result is an artifact of our simulator, and would require further investigation to rule out entirely.

We then wanted to see if there was a particular range of flow and ratio values in each topology for which our scheduling might perform the best. Figure 7 shows a contour plot with number of flows along the y-axis, ratio of mice to total along the x-axis, and the difference in number of critical events between no scheduling and scheduling in the z-axis. In the small data center, the largest improvement is seen in the region with flows between 2-2.5 thousand, and ratios between .9 and .95. Within this region, we observed a 45 percent improvement over no scheduling. Figure 8 shows a similar plot for the medium datacenter. In the medium data center, the best region when flows range between 8 and 10 thousand, and the ratio is between .9 and 1. In this region, we observed a 58 percent improvement over no scheduling.

9 Conclusion and Future Work

In this project, we present a predictive queue management system for software-defined networks, and demonstrate the best-case performance improvement of the approach.

One of the concerns with this approach is its scalability. Since the measurement and scheduling is independent from each other, distributed controllers can be used for measuring slices of the network, and can be aggregated to a centralized server for making the scheduling decisions. Similarily, the new rules to be installed can be distributed across the controllers for fast installation. However, the scheduler can be a bottleneck as number of flows increase, and the scheduling epoch has to be decided in conjuction of the scale of the network. Also, another direction of research is to evaluate the optimal scheduling (instead of the greedy approach proposed in this paper).

By building predictive models of flows, there are interesting applications to this approach. This can be used for predicting good rule timeout values for switches, and perform efficient rule management in SDNs. Also, the predictive models can be use to proactively install rules in the network, thus reducing the latency suffered by flows due to forwarding rules installation.

Finally, this project helped us understand and build models characterizing different phenomena in the networks, and the complexity of dependent phenomena. The experience of building a complete system from scratch was fulfilling, and made us realize the challenges involved in them. Finally, software-defined networks are still in a nascent phase, and our failed experiments with Mininet and non-availability of good simulation frameworks like *ns-2* emphasise the requirement of good prototyping software for SDN applications to run simulations and evaluate different parameters.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[4] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[5] M. E. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proceedings of the 2Nd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 2*, USITS'99, pages 22–22, Berkeley, CA, USA, 1999. USENIX Association.

[6] M. Gupta, J. Sommers, and P. Barford. Fast, accurate simulation for sdn prototyping. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 31–36, New York, NY, USA, 2013. ACM.

[7] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 307–318, New York, NY, USA, 2014. ACM.