Virtualizing Traffic Shapers for Practical Resource Allocation

Gautam Kumar[‡∨], *Srikanth Kandula*[‡], *Peter Bodik*[‡], *Ishai Menache*[‡]
Microsoft Research[‡] *and University of California, Berkeley*[∨]

1 Introduction

Many network resource allocation scenarios would benefit from the use of traffic shapers, such as weighted fair queues (WFQs) [2], priority queues [10] and rate limiters [11]. However, the number of such shapers implemented in hardware in switches, routers, and network interface cards (NICs) is very low; typically less than ten (see Table 1). Instead of simple use of hardware traffic shapers, network operators thus have to resort to more complex solutions.

For example, public clouds such as Amazon Web Services and Windows Azure want to limit the network bandwidth that is allocated to each VM along each path through their network. However, NICs only support a small number of rate limiters in hardware [11]. So, the cloud providers implement rate limits in software, forcing all server traffic to go through the hypervisor. This results in both worse latency and lower throughput because it requires more copying (between guest VM and hypervisor) and it disallows multi-core optimizations such as SRIOV and Direct-IO which let VMs directly read and write from NIC buffers. Dedicating more cores to the hypervisor improves throughput but the cloud provider now has fewer to allocate to customers.

As another example, major companies have announced centralized traffic engineering using SDN switches [6]. To drive networks at high utilization, these schemes solve a global optimization problem to map traffic across all the network paths and allocate available network bandwidth among hosts based on business requirements. Here too, there is a need to ensure that hosts adhere to their allocated rates along each path. Further, there is a need to prioritize among traffic classes; e.g., customer-facing traffic where extra latency means lost revenue should be prioritized over bulk data transfers that only care about finishing within a deadline. However, the number of traffic classes is several orders of magnitude larger than the number of available shapers in modern switches (see Table 1 and [24]). And so, designers rely on coarse prioritization and distributed rate limiters where the (changing) rate limits are communicated by the central scheduler to each of the end-hosts to be enforced in software [6].

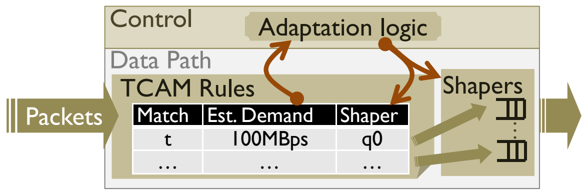

Figure 1: vShaper uses rules on the forwarding path to estimate class' demands and maps them to the shapers available at the switches. Periodically, an adaptation logic changes the mapping from classes to shapers.

A potential solution to this problem is to build hardware support for many more shapers. Maintaining a shaper requires some memory management (e.g., a separate queue for its packets) and some scheduling overhead (e.g., to implement WFQ or priority queuing). To support eight shapers per port, a switch with 64 I/O ports (e.g., an Arista 7050) today uses 2^{15} shapers internally–per shaper, per output port, per input port, the last because of virtual-output queuing [22]. A few concerns exist with trying to increase the number of shapers per port. First the turn-around time to build new hardware is large and application requirements can change quickly. For example, Cisco and Broadcom refresh chassis hardware once every two years whereas 25% of the users of Amazon web services report using 3-70X more instances within the last year [7], likely resulting in more traffic classes. Second, more shapers in hardware means more ASIC and more complex scheduling algorithms which add costs, especially to verify correctness [13].

Here, we propose to virtualize traffic shapers. That is, we present a technique called vShaper that uses few physical shapers to mimic the traffic shaping behavior of many more shapers. Given k physical shapers, n classes of traffic ($n \gg k$), and an error function E that captures the quality of traffic shaping, our intent is to map the traffic classes to the available shapers so as to minimize the error. An example error function could measure how close the bandwidth allocation across classes comes to the ideal allocation achieved when n weighted fair queues are available. At first blush, there are $\frac{k^n}{k!}$ potential groupings of classes to shapers and hence evaluating every one of these is computationally infeasible.

The intuition behind our approach is that traffic shaping becomes substantially simpler if we could (roughly) estimate the traffic demands of each class. For example, priority queuing shapes traffic such that bandwidth is allocated first to the highest priority class, then to the second priority class and so on. Supposing that the demands of each class is known, let x be the highest priority class such that the sum of the traffic demands of classes with priority higher than x is smaller than the link capacity C, and the sum when including the demand of class x exceeds C. Here, the rate allocation is simple – classes with priority higher than x get rate equal to their demand, those with lower priority get zero bandwidth and class x gets some amount no more than its demand. Hence, regardless of however many classes there may be, this effective rate allocation can be enforced by simply mapping the classes into three priority queues.

In the general case, i.e., for other shapers and error functions, the mapping is not as simple even when demands can be estimated. For weighted fair queues (see §2), we identify similar *lossless* groups, i.e., traffic classes that can be mapped into the same shaper without incurring any error. By formulating the mapping of classes to shapers as an optimization problem, we present both an (optimal) dynamic program and a faster greedy solution. Our experiments and analysis reveals that the expected error in rate allocation reduces very quickly with the number of available shapers and is not affected much by the number of traffic classes, thereby hinting at why virtualizing shapers is an effective idea.

How to estimate per-class traffic demands? And, since predicting future demands accurately is impossible, so how to be robust to inaccuracies in demand estimates? Here, we make three observations. First, we show that demands can be measured on the dataplane with commodity (software-defined) switches. vShaper installs ACL filters or OpenFlow rules at the ingress interfaces to estimate traffic demands. Dataplane measurement lets vShaper track changing traffic patterns quickly and without additional software infrastructure. OpenFlow rules and switch ACLs support fairly rich filters. And, switches and NICs support a few orders of magnitude more filters than shapers (Table 1). Optionally,

Switch	**Num. Shapers**		**Num**
	WFQs	Priority Qs	**Rules**
A 7050S[1]	7, interchangeable		512
I G8264[8]	8, interchangeable		256
C Nexus3K[4, 3]	8	1	1000+
J EX3300[9]	8	5	7000
D S4810[5]	4 (3 usable as Priority Qs)		1024

C = Cisco, A = Arista, I = IBM, J = Juniper, D = Dell

Table 1: A survey of the support for shapers in popular commodity top-of-rack switches. The numbers are per interface.

shims at the end-hosts or edge switches can tag packets, e.g., using the DSCP field, to identify which traffic belongs to which class. Second, we note that most of the inaccuracies in demand estimates have only a small impact on shaping error. In the above priority queuing example, the critical priority class x remains the same even if the demands of individual classes change as long as the sum of demands continues to satisfy the above property. Further, vShaper periodically adapts the mapping from classes to shapers based on recent demand estimates. We show that such adaptation can be done often (e.g., once every 50ms on today's switches). Figure 1 depicts a potential implementation where the adaptation logic is built in software on the control processor at a switch. The adaptation reduces the effects of error in demand estimation. Finally, we note that there are multiple switches along a path and multiple paths between racks due to the current *bushy* nature of the datacenter network cores (e.g., fat-trees, VL2). We leave exploiting multiple paths to future work but observe that employing vShaper at every switch along a path leads to lower error than using vShaper at just one (bottleneck) switch.

vShaper was motivated by the observation that recent work in full bisection networks and software defined switches has simplified management of the networks; servers can be placed anywhere in the datacenter [14, 21], routing and traffic engineering have become easier [6]. However, extracting good performance from the dataplane especially with multiple tenants still requires complicated software [19, 24] or new hardware support [25]. We build upon the observation by Lam et. al. [20] that using the shapers already available in the dataplane at switches and NICs could be useful but take the next step towards making the available shapers support many more traffic classes by virtualizing them. It is possible that these ideas could help towards building cost-effective middleboxes that offer dedicated traffic shaping functionality.

Early experiments from a prototype on a merchant silicon switch (an Arista 7048) and simulations that replay traces from a few datacenter workloads demonstrate the

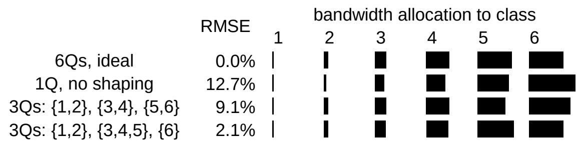

Figure 2: Example bandwidth allocation for 6 classes (horizontal bars) under 4 different groupings and the corresponding root mean square error. The classes have demands= 0.002, 0.04, 0.18, 0.36, 0.6, and 0.9 (as fraction of capacity), weights= 0.02, 0.08, 0.1, 0.2, 0.3, and 0.3, and have been sorted in decreasing order on $\frac{w}{d}$. Using three queues is better that one. But, it is even more important to use the right grouping.

feasibility of this approach. We show that we can change the mapping for 40 shapers in 5.6 ms. In a simulation of a network traffic from a production, data-parallel cluster with hundreds of classes, vShaper achieves 2-3% error with just 8 weighted-fair queues, which is much smaller than existing approaches (§3).

2 Design Principles of vShaper

In this section, we outline how vShaper virtualizes shapers using the case of weighted fair queues to allocate network bandwidth as an example. We discuss other shapers briefly in §2.2.

The goal of **fair queuing** is: given traffic classes, each with a demand rate d_i and a weight w_i, allocate bandwidth in a weighted max-min fair manner [19, 20, 23, 24]. That is, each class receives a rate proportional to its weight and the unused bandwidth is allocated to classes that can use them. A class can correspond to a TCP flow, a network service, a VM or all the VMs of a tenant. Such bandwidth shaping is important to restrict misbehaving tenants and to offer premium services [19, 23, 24].

If switches had enough weighted fair queues (WFQs) to map each class onto a WFQ, then we could achieve weighted max-min allocation with just a little more support (e.g., with end-to-end flow control [20]). The chief obstacle however is that there can be many fewer WFQs per interface (see Table 1) than classes (100s-1000s [24]).

Suppose that there are k WFQs and n classes with $n \gg k$. How to map classes to shapers such that each class achieves a rate close to what it would have in the ideal case where there each class is mapped to a WFQ? To build such mapping, note first that the rates achieved by an *ideal* WFQ scheme is given by

$$r_i = \min(d_i, w_i\alpha), \text{ where } \alpha = \arg\max_{\sum r_i \le C} \sum r_i. \qquad (1)$$

Here, α is the max-min share normalized by weight, and C is the capacity of the outgoing link. Intuitively, classes that demand less than their weighted share get rate equal to their demand whereas the others get no more than their share. We call the former classes *underweight* and the latter *overweight*.

Simple observations. Suppose that the weight of each queue is set as the sum of the weights of the classes that are allocated to it. Observe first that if all underweight classes are mapped into a single queue (say q), then $\left(\sum_{i\in q} d_i\right) \le \left(\sum_{i\in q} w_i\right)\alpha$, meaning that each class gets its demand rate, exactly as in (1). Next, consider a subset of overweight classes that have the same $\frac{d}{w}$ ratio. Assume that we map all these classes to a single queue q. Because the cumulative rate of traffic $\sum_{i\in q} w_i\alpha$ is divided proportionally to demand, each class obtains rate $\frac{d_i}{\sum_{j\in q} d_j}\sum_{j\in q} w_j\alpha = d_i\frac{w_i}{d_i}\alpha = w_i\alpha$ which is, again, identical to the ideal rate (1).

vShaper**'s aggregation rules.** The mappings described above lead to "lossless" aggregation compared to ideal WFQ. Inspired by these, vShaper applies three basic rules for class aggregation: (i) The weight of each queue is set as the sum of the weights of the classes that are allocated to it; (ii) All underweight classes are aggregated into a single queue; and (iii) Classes are grouped contiguously based on their $\frac{d_i}{w_i}$ ratio. We note that rule (iii) still requires an *algorithm* to determine the $\frac{d}{w}$ boundaries for each queue. The specific algorithms that we propose are in §2.1; here we reason about their properties.

Measuring quality of aggregation. Based on the above rules, the rates obtained by vShaper (denoted $\hat{r}_i$) are

$$\hat{r}_i = \begin{cases} d_i & \text{if i is underweight} \\ \hat{\alpha}\frac{d_i}{\sum_{j\in s} d_j}\sum_{j\in q(i)} w_j & \text{if i is overweight,} \end{cases} \qquad (2)$$

where $q(i)$ denotes the queue to which i is assigned. It is easy to show that $\hat{\alpha}$ is equal to α in (1). Note that (ii) immediately implies that $\hat{r}_i = r_i$ for all underweight classes. However, $\hat{r}_i$ could be different than r_i for overweight classes.

vShaper aims to minimize the sum of squares difference between the fair-queue rate r_i and the rate $\hat{r}_i$ achieved due to virtualization. Namely our measure for the quality of aggregation is $\sum_i \beta_i(r_i - \hat{r}_i)^2$. The β_i's are class-specific weights, which can be set by the network manager. For example, β_i can be set inversely proportional to the class' demand to eliminate the bias towards the elephant classes (high d_i). Using (1) and (2), the sum of squares cost function (divided by α) is given by

$$\sum_i \beta_i \left(w_i - d_i \frac{\sum_{j\in q(i)} w_j}{\sum_{j\in q(i)} d_j} \right)^2. \qquad (3)$$

Fig. 2 shows a few example groupings and the corresponding error for $\beta_i = 1$.

Why *contiguous* grouping is reasonable? Recall that there are up to $\frac{k^n}{k!}$ possible groupings of n (overweight) classes into k queues. Here, we show that vShaper's choice to group based on $\frac{d}{w}$ (rule (iii)) is optimal when class-specific weights are such that $\beta_i = \frac{1}{d_i}$. For general weight assignments, the error is small and upper-bounded.

Theorem 1. *Consider the error* (3) *with $\beta_i = \frac{1}{d_i}$. There exists a minimizer of this error which assigns classes to queues contiguously based on the $\frac{w_i}{d_i}$ ratio. That is, this assignment has the property that for any queue q and class i, if $\operatorname{argmin}_{j\in q}\frac{w_j}{d_j} \le \frac{w_i}{d_i} \le \operatorname{argmax}_{j\in q}\frac{w_j}{d_j}$, then i is assigned to q.*

The proof, which we outline below, follows through a reduction to a k-means problem in single dimension.

Proof. For $\beta_i = \frac{1}{d_i}$, (3) can be equivalently written as $\sum_i d_i \left(\frac{w_i}{d_i} - \frac{\sum_{j\in q(i)} w_j}{\sum_{j\in q(i)} d_j}\right)^2$. Consider the following transformation: for each class (w_i, d_i) create d_i virtual points having a scalar value of $\frac{w_i}{d_i}$ each. Consider the k-means clustering problem [17] over these virtual points, where the number of desired clusters is the number of queues, k. Suppose that all virtual points originating from a class i are forced to be assigned to the same cluster, denoted $q(i)$. Then, the k-means cost function can be written as $\sum_i d_i \left(\frac{w_i}{d_i} - \frac{\sum_{j\in q(i)} d_j \frac{w_j}{d_j}}{\sum_{j\in q(i)} d_j}\right)^2$, which is identical to (3). Therefore, our original minimization problem can be viewed as a k-means problem in single dimension, with the constraint that all of the virtual points originating from a class lie in the same cluster.

Obviously, any solution of the unconstrained 1-d k-means problem is contiguous w.r.t. the point values (i.e., $\frac{w_i}{d_i}$). We claim that imposing the additional constraint that all virtual points belong to the same cluster does not matter. Indeed, initiating from any solution to the unconstrained 1-d k-means problem, we perform a "k-means iteration", consisting of two steps: (i) assign all virtual points of a class to their closest center; in case of a tie, all such points are sent to the same cluster; (ii) update the cluster centers based on the new assignment. Both steps do not increase the cost; the former because points are associated to a closest center, and the latter because the overall error of a set of points is smallest when the center is chosen as their average. □

2.1 Algorithms for contiguous grouping

As discussed above, grouping classes based on their $\frac{d_i}{w_i}$ ratio can lead to low errors. Here, we design algorithms that minimize (3) under rules (i)-(iii). Since rule (iii) is enforced, any such algorithm first orders the classes according to their $\frac{d_i}{w_i}$ (ties broken arbitrarily), and then has to determine the "boundary points" in terms of $\frac{d}{w}$ value, which determine the allocation to the different queues. We implemented several algorithms. We describe below two algorithms which have led to notably good performance in our experiments. Assume without loss of generality that $\frac{w_1}{d_1} \le \frac{w_2}{d_2} \le \cdots \le \frac{w_n}{d_n}$.

1) DP-based algorithm. This Dynamic Programming (DP)-based algorithm obtains the *optimal* error under rules (i)-(iii). The DP-based algorithm updates an $n \times k$ matrix A, where each entry $A(i,k')$ stands for the optimal cost for dividing classes $i, i+1, \dots, n$ to the remaining k' queues (assuming that the previous points have been assigned to the first $k'-1$ queues). Entries are updated recursively via the equation $A(i,k') = \min_{i\le j\le n}\left\{B_{i,j} + A(j+1, k'-1)\right\}$, where $B_{i,j}$ is the total squared error (in the sense of (3)) of the cluster consisting of points $\{i, i+1, \dots, j\}$. It can be shown that the running complexity of the algorithm is $O(n^2k)$, while the space complexity can be reduced to $O(nk)$ by optimizing the order of updates. Importantly, the complexity values do not depend on the spread of the d values.

2) Greedy algorithm. The greedy algorithm is an iterative algorithm with lower complexity: $O(n\log n)$ time and $O(n)$ space. In each iteration, greedy identifies a contiguous pair of classes which have the smallest error (in the sense of (3)) if combined, and constructs a new combination class with demands (and weights) set to the sum of demands (and weights) of the constituent classes. Greedy then places this newly constructed class back into the sorted list. Due to the contiguity constraint, the sorted list begins with $n-1$ entries, one for each pair of classes $\{i, i+1\}$ for $i = 1, \dots, n$. Each step, by combining two entries, reduces the size of the list by one and greedy terminates when there are k entries. Combining a pair of classes is constant time. Inserting and removing from the sorted list can be done in $O(\log n)$ time. As we show in Section 3, the greedy algorithm exhibits close to optimal performance.

2.2 Other shapers

As outlined in the introduction, when using priority queues for bandwidth shaping, we only need three queues; one for classes that are completely satisfied, one for the partially satisfied class, and one for the unsatisfied classes. We could further split these queues to protect against inaccurate demand estimates. We leave grouping of priority queues to achieve low latency for future work.

Approximating many *rate limiters* with few is similar, but simpler than weighted fair queuing. Rate limiters, unlike WFQs, are not work conserving; they hold each

Operation	**Latency**	**Freq**
Update shaper	2.2ms	every adapt. intrvl
Update 40 shapers	5.6ms	every adapt. intrvl
Point rule to diff. shaper	2.1ms	every adapt. intrvl
Insert a new ACL	12.05ms	once
Add new rule to an ACL	13.43ms	once, classes arrive/ leave
Updating a rule	7.13ms	once, classes arrive/ leave
Apply ACL to an i/f	4.72ms	once

Table 2: Changing switch parameters on an Arista7048S using a Python script on the EOS shell

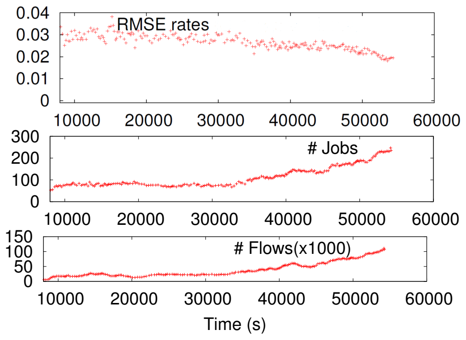

Figure 3: vShaper's ability to keep apart per-job traffic in a large map-reduce cluster

class to a specified rate value whereas WFQs distribute excess bandwidth among the classes that can use it. Assume each class has a rate limit R_i and demand d_i. Here too, mapping under-rate classes ($d_i \leq R_i$) into a single rate limiter with limit $\sum R_i$ does not impact the rates obtained by these classes. Similarly, for over-rate classes, mapping classes with the same $\frac{d_i}{R_i}$ into the same rate limiter would result in the correct rate limits for all grouped classes. With this intuition, we can use a similar greedy algorithm to cluster many classes into a few rate limiters.

3 Evaluation

To verify the feasibility of vShaper, we microbenchmarked various aspects of vShaper on an Arista 7048S 48x10G switch. We used access-control lists (ACLs) to specify the classes and to measure their demands. The ACL counters were very accurate– less than 1% error over the entire range of demands [0, 10Gbps]. We report the latency to modify different aspects of switch state in Table 2. The numbers are overestimates and could improve by optimizing the Python script that runs on the switch (for e.g., by batching updates, see time to update 1 and 40 shapers). We conclude that it is possible to adapt switch state at least an order of magnitude more often than end-to-end controllers.

We further evaluate vShaper in the context of large data-parallel clusters to allocate bandwidth across hundreds of concurrent jobs. We perform a fluid model simulation that replays traces from a several thousand node production data-parallel cluster. Figures 3 (middle, bottom) show the number of jobs and their flows that pass through a typical link in the examined cluster. We see that on average there are over 100 jobs and over 50K flows on that link. Suppose that the goal of shaping is to provide equal share across jobs independent of their number of flows. Fig. 3 (top) shows the RMSE between rates achieved by vShaper when mapping jobs to 8 queues compared to those achieved by an "ideal" scheme that maps each job to a separate queue. Whereas the ideal scheme would require hundreds of WFQs, vShaper achieves an error of about 3% with 8 queues, indicating that vShaper's mapping can be effective in practice.

To compare vShaper with several variants, we present results on a simulated dataset that has thousands of traffic samples. Each sample contains a number of classes n sampled from an exponential distribution with mean 10. The demands and weights of each class are sampled from a pareto distribution with shape parameter of 0.4. Figure 4 compares vShaper with several alternatives – no shaping (just 1 queue), a scheme that randomly assigns classes to queues and sets the weight of the queue to be the sum of the weights of classes assigned to that queue, two schemes proposed by NetShare [20], the dynamic programming based algorithm described in §2.1 and another contiguous grouping scheme that partitions the overweight classes *geometrically* on $\frac{w}{d}$. In other words, given n classes and k queues, let $\gamma = \left(\frac{\max_i w_i/d_i}{\min_i w_i/d_i}\right)^{\frac{1}{k}}$, this scheme assigns class i to the queue j for the smallest value j that satisfies $\frac{w_i}{d_i} \leq \gamma^j \min_i \frac{w_i}{d_i}$. For details about NetShare's schemes, we refer the reader to [20]. We see that using 8 queues, vShaper's error, at the 90th percentile (across different traffic samples) is approximately 1%, compared to 15% of algorithms proposed by NetShare (see Fig. 4). Neither of the schemes proposed by NetShare perform much better than random. Also, vShaper's greedy algorithm has error that is comparable to that of the dynamic program. Recall that the DP scheme has $O(n^2k)$ time complexity whereas the greedy algorithm has $O(n \log n)$ time complexity. On the other end, vShaper has better error than the geometric partitioning algorithm which has $O(n)$ time complexity. Finally, note that, as expected, the error reduces quickly with the number of available shapers. The marginal reduction in error when going from 2 to 4 queues is roughly 4X larger than when going from 6 to 8 queues. This hints that a small number of shapers may suffice.

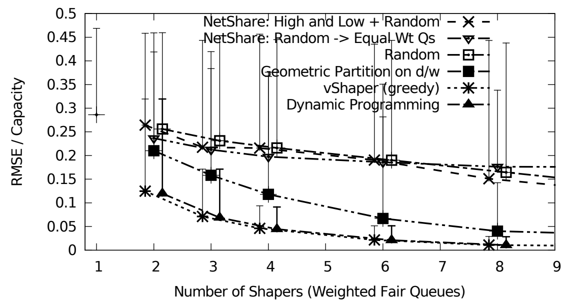

Figure 4: On simulated data, we find that vShaper's shaping leads to much less error than alternatives. The lines are at the 90th percentile, whereas the upper error bar is at the maximum. We see that vShaper, with just eight queues, yields no worse than a couple percentage error whereas the best previously known scheme has more than 15% error. In fact, random mapping requires about 100 queues before it achieves error that is similar to vShaper with eight queues.

4 Related Work and Final Remarks

There is a large body of work on network resource allocation that vShaper could apply to. On sharing network bandwidth, Seawall [24] provides a per-VM max-min weighted fair share using TCP-style end-to-end congestion feedback and rate adaptation; SecondNet [18] and Oktopus [16] carve network-slices for tenants and enforce them using rate limits per VM; FairCloud [23] designs better policies for sharing bandwidth (e.g., those that are strategy proof) and eyeQ [19] provides per-VM max-min weighted fair shares in the context of a full bisection bandwidth datacenter topology where congestion is limited to the first and the last hops. pFabric [15] argues for a switch with very small buffers, extremely granular priorities (e.g., based on remaining bytes in that flow) and strict priority based packet deliver, i.e., switch chooses packets to send (and drop) based on their priorities; this achieves near-optimal flow completion times, but could starve long flows with low priorities. By virtualizing WFQs, rate limiters and priority queues, vShaper can significantly simplify each of these solutions. In that respect, vShaper is closest to NetShare [20] which also observes that using WFQs on the data path simplifies bandwidth sharing. However, vShaper uniquely offers techniques to dynamically map many more classes onto the few available shapers based on their estimated demands. Finally, vShaper builds on the recent availability of open switch software stacks (e.g., Arista EOS) and OpenFlow [12] which allowed us to run Python scripts on the switch and programmatically alter aspects along the data plane (viz: demand estimation, mapping of classes to WFQs).

In conclusion, vShaper shows that by dynamically mapping traffic classes on to hardware shapers, based on demand estimates, it can mimic the behavior of a hypothetical entity that supports many more shapers. We showed that such dynamic mapping can be implemented along the data plane on today's commodity switches.

References

[1] Arista 7050S Switch. http://goo.gl/JHwgW.
[2] Cisco: Configuring Weighted Fair Queuing. http://bit.ly/g0PU2r.
[3] Cisco Nexus ACL Guide. http://goo.gl/tz7jh.
[4] Cisco Nexus QoS Guide. http://goo.gl/oicTM.
[5] Dell Force10 S4810 Switch. http://goo.gl/V1fq2.
[6] Google's Secret Switch to the Next Wave of Networking. http://bit.ly/Iup9s8.
[7] High Levels of AWS Growth. http://bit.ly/PRN7Fa.
[8] IBM RackSwitch G8264 Application Guide. http://goo.gl/eZpRS.
[9] Juniper EX3300 Switch. http://goo.gl/vBC24.
[10] Juniper: Understanding CoS Priority Group Shaping and Queue Shaping. http://juni.pr/Yd928A.
[11] Mellanox MLX4 Driver for VMware. http://bit.ly/XQLhWB.
[12] OpenFlow. http://www.openflow.org/.
[13] Personal Communication with Nick McKeown.
[14] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.
[15] M. Alizadeh et al. Deconstructing datacenter packet transport. In *HotNets*, pages 133–138, 2012.
[16] H. Ballani et al. Towards Predictable Datacenter Networks. In *ACM SIGCOMM*, 2011.
[17] R. O. Duda, P. E. Hart, and D. G. Stork. Pattern classification and scene analysis 2nd ed. 1995.
[18] C. Guo et al. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNEXT*, 2010.
[19] V. Jeyakumar et al. EyeQ: Practical Network Performance Isolation for the Multi-tenant Cloud. In *Usenix HotCloud*, 2012.
[20] V. T. Lam, S. Radhakrishnan, A. Vahdat, G. Varghese, and R. Pan. Netshare and stochastic netshare: Predictable bandwidth allocation for data centers. *SIGCOMM CCR*, 2012.
[21] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.
[22] N. McKeown et al. Tiny tera: A packet switch core. In *Hot Interconnect*, 1996.
[23] L. Popa et al. FairCloud: Sharing the Network in Cloud Computing. In *ACM SIGCOMM*, 2012.
[24] A. Shieh et al. Sharing the Data Center Network. In *Usenix NSDI*, 2011.
[25] C. Wilson et al. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *ACM SIGCOMM*, 2011.