DevoFlow: Cost-Effective Flow Management for High Performance Enterprise Networks

Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, Andrew R. Curtis*, Sujata Banerjee
HP Labs, Palo Alto, CA, US – {*FirstName.LastName*}*@hp.com*
*University of Waterloo School of Computer Science, Waterloo, Ontario, Canada – *a2curtis@uwaterloo.ca*

ABSTRACT

The OpenFlow framework enables flow-level control over Ethernet switching, as well as centralized visibility of the flows in the network. OpenFlow's coupling of these features comes with costs, however: the distributed-system costs of involving the OpenFlow controller on flow setups, and the switch-implementation costs of involving the switch's control plane too often.

In this paper, we analyze the overheads, and we propose *DevoFlow*, a modification of the OpenFlow model in which we try to gently break the coupling between centralized control and centralized visibility, in a way that maintains a useful amount of visibility without imposing unnecessary costs.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Performance, Measurement

Keywords

Network, OpenFlow, Traffic Engineering, Enterprise

1. INTRODUCTION

The OpenFlow [9] framework enables fine grained, flow-level control of Ethernet switching. OpenFlow has been deployed at various academic institutions and research laboratories, and has been the basis for many recent research papers [1, 13], as well as for hardware products from vendors such as NEC, Arista, and Toroki. (We will assume the reader is familiar with the basics of OpenFlow.)

Hotnets '10, October 20–21, 2010, Monterey, CA, USA.

Compared to traditional packet switching, OpenFlow provides several benefits: (1) a logically centralized controller can supervise all flow-level decisions, thereby avoiding the need to enforce global policies by carefully crafting switch-by-switch configurations; (2) the central controller can see all flows, potentially enabling globally-optimal management of network traffic; and (3) OpenFlow switches are relatively simple and future-proof, since policy is imposed by controller software, rather than by switch hardware or firmware.

While OpenFlow was originally proposed for campus and wide-area networks, others have made quantified arguments that OpenFlow is a viable approach to data-center networks [15]. The examples in this paper are taken from data-center environments, but our work should be applicable to other cases in which OpenFlow might be used.

However, OpenFlow is not perfect. In particular, we believe it excessively couples the first two features: central control and central visibility. If one wants the controller to have visibility over all flows, it must also be on the critical path of setting up all flows. Experience suggests that putting the controller on the critical path of every flow setup is not sufficiently scalable, as we discuss in section 3.1.

Perhaps, then, having full visibility over all flows is not quite the right goal. Instead, we propose devolving some of control back to the switches, in a way that preserves central control and visibility over all *significant* flows, while limiting the load on the central controller. Our design, called *DevoFlow* (for Devolved OpenFlow), modifies the OpenFlow model to redistribute as many decisions as possible to the switches, in ways amenable to simple and cost-effective hardware implementation.

In essence, DevoFlow is designed to allow aggressive use of wild-carded OpenFlow rules – thus reducing the number of switch-controller interactions, and the number of TCAM entries – by providing new mechanisms to detect QoS-significant flows efficiently, by waiting until they actually become significant. DevoFlow also introduces new mechanisms to allow switches to make local routing decisions when these do not actually require per-flow vetting by the controller.

The reader should note that in this paper we are not proposing any radical new designs. Rather, we are point-

ing out that a system like OpenFlow, when applied to high-performance enterprise or data-center networks, must account for *quantitative* real-world issues. Our arguments for DevoFlow are essentially an analysis of tradeoffs between centralization and its costs, especially with respect to real-world hardware limitations.

Therefore, we start in sec. 2 with a discussion of the benefits of centralized control and visibility, so as to understand how much devolution we can afford. Then, in sec. 3, we analyze the costs of flow setup and flow-statistics gathering, both for an idealized switch and under real-world implementation constraints. Finally, sec. 4 describes the DevoFlow design in some detail.

2. BENEFITS OF CENTRAL CONTROL

In this section, we discuss which benefits of OpenFlow's central-control model are worth preserving, and which could be tossed overboard to lighten the load.

A logically centralized controller can supervise all flow-level decisions, thereby avoiding the need to enforce global policies by carefully crafting switch-by-switch configurations: OpenFlow provides an advantage over traditional firewall-based security mechanisms, in that it avoids the complex and error prone process of creating a globally-consistent policy out of local accept/deny decisions [3, 13]. Similarly, OpenFlow can provide globally optimal admission control and flow-routing in support of QoS policies, in cases where a hop-by-hop QoS mechanism cannot always provide global optimality [8].

However, this does not mean that *all* flow setups should be mediated by a central controller. In particular, we argue that microflows (a *microflow* is equivalent to a specific end-to-end connection) can be divided into three broad categories: *security-sensitive flows*, which must be handled centrally to maintain security properties; *significant flows*, which should be handled centrally to maintain global QoS and congestion properties; and *normal flows*, whose setup can be devolved to individual switches.

Of course, all flows are potentially "security-sensitive," but some flows can be categorically, rather than individually, authorized by the central controller. Using standard OpenFlow, one can create wild-card rules that pre-authorize certain sets of flows (e.g.: "all MapReduce nodes within this subnet can freely intercommunicate") and install these rules into all switches. Similarly, the control can define flow categories that demand per-flow vetting (e.g., "all flows to or from the finance department subnet"). Thus, for the purposes of security, the controller need not be involved in every flow setup.

Central control of flow setup is also required for some kinds of QoS guarantees. However, in many settings, only those flows that require guarantees actually need to be approved individually at setup time. Other flows can be categorically treated as best-effort traffic. Kim *et al.* [8] describe an OpenFlow QoS framework that detects flows requiring QoS guarantees by matching against certain header fields (such as TCP port numbers) while wild-carding others. Flows that do not match one of these "flow spec" categories are treated as best-effort.

Some reviewers have questioned why we want to support QoS requirements within data-center networks. Two forces drive this requirement. First, the trend towards "fabric convergence" (using a single network fabric to carry normal networking, storage, HPC, video, etc.) aims to reduce cost and management complexity, but some of these services have strict QoS requirements. Second, many data center operators want to support multiple tenants. While compute and storage resources can be virtualized using well-understood techniques, the need to provide guaranteed and properly isolated network performance to multiple tenants creates interesting QoS requirements.

In summary, we believe that the central-control benefits of OpenFlow can be maintained by individually approving certain flows, but categorically approving others (by means of wildcard rules).

The central controller can see all flows, potentially enabling globally-optimal management of network traffic: Regardless of whether the controller should be involved in admission-control decisions at the start of certain flows, in order to properly manage the performance of a network, the controller needs to know about the current loads on network elements. (This assumes that we want to exploit statistical multiplexing gain, rather than strictly controlling flow admission to prevent oversubscription.) Further, the controller may need to know which flows are creating loads on certain links, so as to possibly re-route or throttle problematic flows, and to forecast future network loads. For example, NOX [15] "can utilize real-time information about network load ... to install flows on uncongested links."

More specifically, dynamic flow scheduling of data-center traffic in Hedera can deliver up to 113% more bisection bandwidth than static load balancing [1].

However, this requirement does not mean that the controller's flow-awareness should start with the initial setup of every flow. First, some flows ("mice") may be brief enough that, individually, they are of no concern, and are only interesting in the aggregate. Second, some QoS-significant best-effort flows might not be distinguishable as such at flow-setup time – that is, the controller cannot tell from the flow setup request whether a flow will become sufficiently intense (an "elephant") to be worth handling individually.

Instead, the controller should be able to efficiently detect elephant flows as they become significant, rather than paying the overhead of treating every new flow as a potential elephant. The controller can then re-route problematic elephants in mid-connection, if necessary. For example, Hedera's scheduler requires detection of "large" flows at the edge switches, where a large flow could be defined as taking 10% of the host-NIC bandwidth.

OpenFlow switches are relatively simple and future-

proof, since policy is imposed by controller software, rather than by switch hardware or firmware: Clearly, we would like to maintain this property. We believe that our DevoFlow proposal, while adding some complexity to the design, maintains a reasonable balance of switch simplicity vs. system performance, and may actually simplify the task of a switch designer who seeks a high-performance implementation.

3. COSTS OF OPENFLOW

In this section, we look at several costs of the original OpenFlow model – flow setups and statistics collection – from the perspectives both of an abstract distributed system design, and of a real-world switch implementation.

We ground our discussion of implementation costs in our experience implementing OpenFlow on the ProCurve 5406zl Ethernet switch, which uses an ASIC on each multi-port line card, and also has a CPU for management functions. This experimental implementation has been deployed both internally and in numerous academic institutions. The practical issues we describe are representative of issues facing any flow-management framework, and we believe that the 5406zl switch is representative of the current generation of Ethernet switches.

3.1 Flow setup costs

The costs of flow setup can be divided into controller load costs, network costs, and switch-internal costs.

Controller load: Involving the controller in all flows creates a potential scalability problem: any given controller instance can support only a limited number of flow setups per second; e.g., Tavakoli *et al.* [15] report that one NOX controller can handle "at least 30K new flow installs per second while maintaining a sub-10ms flow install time. [...] The controller's CPU is the bottleneck." Kandula *et al.* [7] found that 100K flows arrive every second on a 1500-server cluster, implying a need for multiple OpenFlow controllers.

Some researchers have worked on distributed implementations of the OpenFlow controller – also valuable for fault tolerance. For example, Tootoonchian and Ganjali [17] describe HyperFlow, a mechanism that distributes the controller, but their approach can only support global visibility of rare events such as link state changes, and not of frequent events such as flow arrivals. However, data-center traffic engineering requires central visibility of all large flows [1], which distributed OpenFlow controller approaches cannot yet achieve.

Network: Using the central controller for all flow setups imposes both network load and latency.

To set up a bi-directional flow on a N-switch path, OpenFlow generates $2N$ flow-entry installation packets, and at least one initial packet in each direction is diverted first to and then from the controller. This adds up to $2N + 4$ extra packets.[1] These exchanges also add latency – up to twice the controller-switch RTT. The average length of a flow in the Internet is very short, around 20 packets per flow [16], and datacenter traffic has similarly short flows [6]. Therefore, full flow-by-flow control using OpenFlow generates a lot of control traffic – on the order of one control packet for every two or three packets delivered if $N = 3$, which is a relatively short path, even within a datacenter.

In terms of network load, OpenFlow's one-way flow-setup overhead (assuming a minimum-length initial packet, and ignoring overheads for sending these messages via TCP) is about $94 + 144N$ bytes to or from the controller – e.g., about 526 bytes for a 3-switch path. Use of the optional flow-removed message adds $88N$ bytes. The two-way cost is almost double these amounts, regardless of whether the controller sets up both directions at once.

Switch-implementation costs: Real switches have finite bandwidths between their data and control planes, and finite compute capacity. These issues can limit the rate of flow setups; the best implementations we know of can set up only a few hundred flows per second. The current 5406zl implementation can do 146 setups per second.

First, on a flow-table miss, the data plane must invoke the switch's control plane, in order to encapsulate the packet for transmission to the controller.[2] Unfortunately, the management CPU on most switches is relatively wimpy, and was not intended to handle per-flow operations.

Second, even within a switch, control bandwidth may be limited, due to cost considerations. The control datapath within a linecard ASIC is very fast, so the switch can make forwarding decisions at line rate. On the other hand, the control datapath between the ASIC and the CPU is not frequently used in traditional switch operation, so this is typically a slow path. For example, the 5406zl has a raw bandwidth of 300 Gbit/sec, but we measured the loopback bandwidth between the ASIC and the management CPU at just 35 Mbit/sec. This four-order-of-magnitude difference is in line with observations made by others [4].

Finally, the slow switch CPU can limit the bandwidth between switch and central controller. Using the 5406zl we measured the bandwidth available for flow-setup payloads between the switch and the OpenFlow controller at just 10 Mbit/sec.

The DIFANE approach [18] avoids these costs by splitting pre-installed OpenFlow wildcard rules among multiple switches in a clever way that ensures all decisions can be made in the data plane. However, DIFANE does not address the issue of global visibility of flow states and statistics.

3.2 Switch state size

OpenFlow rules can have wildcards. While exact-match

[1]The controller could set up both directions at once, cutting the cost to $N + 2$ packets; NOX apparently has this optimization.

[2]While it might be possible to do a simple encapsulation entirely within the data plane, the OpenFlow specification requires the use of a secure channel, and it might not be feasible to implement the Transport Layer Security (TLS) processing and TCP connection state without using the switch's CPU.

rule lookups can be implemented with a hash table, wildcard rules must be stored in a TCAM. TCAM entries are an expensive resource, in terms of ASIC area and power consumption.

Also, OpenFlow rules require matching against at least 240 bits, and more for higher-radix switches. Since TCAM widths are typically multiples of 36 or 72 bits, generally this means about 288 bits are necessary for each rule. This can be compared to the 60-bit match for a traditional Ethernet address lookup (48 bits of MAC address + 12 bits of VLAN ID). Our switch hardware has enough SRAM for about 16K exact-match entries (although we do not currently use the SRAM), but has TCAM space for only about 1500 wildcard OpenFlow rules.

Finally, because OpenFlow rules are per-flow, rather than per-destination, each directly-connected host will typically require an order of magnitude more rules than in a traditional Ethernet lookup. (One study reports a 10:1 ratio of flows to hosts [6].) Use of wildcards could reduce this ratio, but this is often undesirable as it reduces the ability to implement flow-level policies (such as multipathing) and flow-level visibility.

The implication of state-size limits is that in a real-world OpenFlow deployment, there will be pressure to reduce the number of wild-card flow-table entries. (Note that DIFANE addresses this issue, at least in part.)

3.3 Hardware technology issues

A fair question to ask is whether our measurements are representative, especially since the 5406zl hardware was not designed to support OpenFlow. Hardware optimized for OpenFlow would clearly improve those numbers, but throwing hardware at the problem adds more cost and power consumption – especially when in the form of TCAMs. Moore's Law won't provide much relief; improvements in on-chip memory speed and density are offset by similar increases in Ethernet speeds and per-switch port density. (Off-chip memory for line-rate lookups becomes increasingly infeasible.)

Vendors of merchant-silicon Ethernet switch chips, such as Broadcom, Fulcrum, Fujitsu, and Marvell, are notably coy about the size of their TCAMs and other tables. However, we have been told that these chips operate under memory size and power constraints substantially similar to those for full-custom ASICs.

The NetFPGA implementation of OpenFlow [12] can store 32K–64K exact-match entries, but only has enough TCAM space to store 32 wildcard rules. While the limits of this experimental platform might not be typical of commercial switches, it conforms to the pattern of supporting many more exact-match rules than wildcard rules.

Although the state-size issues described in section 3.2 would not be as pressing in a software-based router implemented using commodity server hardware [5], we do not believe such systems will be cost-effective for most enterprise applications in the foreseeable future.

3.4 Flow statistics costs

Global bandwidth controllers, such as Hedera, need timely access to statistics. (Hedera's current implementation collects updates every 5 seconds [1], but this limit is imposed by their NetFPGA-based hardware; they would like to get their control loop down to a few tens of milliseconds [2].)

OpenFlow supports three per-flow counters (packets; bytes; flow duration) and provides two approaches for moving these statistics from switch to controller:

- **Push-based**: The controller learns of the start of a flow whenever it is involved in setting up a flow. Optionally, OpenFlow allows the controller to request an asynchronous notification when a switch removes a flow table entry, as the result of a controller-specified per-flow timeout. (OpenFlow supports both idle-entry timeouts and hard timeouts.) If flow-removed messages are used, this increases the per-flow message overhead from $2N + 2$ to $3N + 2$. Also, the existing push-based mechanism does not inform the controller about the behavior of a flow before the entry times out.
- **Pull-based**: the controller can send a Read-State message to retrieve the counters for a set of flows matching a wild-card flow specification. This returns $88F$ bytes for F flows. In the worst case, reading the stats for all 16K exact-match rules and 1500 wild-card rules theoretically supported by our switch would return 1.3MB; doing this once per second would require slightly more than the 10Mbit/sec bandwidth available between the switch CPU and the controller!
  Optionally, Read-State can request a report aggregated over all flows matching a wild-card specification; this can save switch-to-controller bandwidth but loses the ability to learn much about the behavior of specific flows.

In short, the existing statistics mechanisms are both relatively high overhead, and in particular they do not allow the controller to request statistics only for the small fraction of elephant flows that actually matter for performance.

4. DEVOFLOW

We have several design principles for DevoFlow:

- **Retain as much of OpenFlow as possible**: including simple implementations of fast switches.
- **Maintain central control over the important things**: but devolve other decisions to the switches as much as possible.
- **Maintain central visibility over QoS-significant flows**: but otherwise provide only aggregated information.
- **Reduce switch-controller network bandwidth.**
- **Avoid leaving the data plane**: since switch-internal bandwidth between the data and control planes, and switch CPU capacity, are both limited.

Our overall goal is to enable otherwise infeasible policies for high-performance enterprise and data-center networks.

DevoFlow attempts to resolve two dilemmas; a control dilemma:

1. Invoking the OpenFlow controller on every flow setup provides good start-of-flow visibility, but puts too much load on the control plane.
2. Aggressive use of OpenFlow flow-match wildcards reduces control-plane load, but prevents the controller from seeing the events that it wants to see.

and a statistics-gathering dilemma:

1. Collecting OpenFlow counters on lots of microflows, via the pull-based Read-State mechanism, can create too much control-plane load.
2. Aggregating counters over multiple microflows via the wild-card mechanism may undermine the controller's ability to manage specific elephant microflows.

Our resolution of these dilemmas is to support aggressive use of flow wildcards, by introducing a few new, simple mechanisms to improve visibility.

4.1 Mechanisms for devolving control

We introduce two new mechanisms for devolving control to a switch, *rule cloning* and *local actions*.

Rule cloning: Under the standard OpenFlow mechanism for wildcard rules, all packets matching a given rule are treated as one flow. This means that if we use a wildcard to avoid invoking the controller on each microflow arrival, we also are stuck with routing all matching microflows over the same path, and aggregating all statistics for these microflows into a single set of counters.

In DevoFlow, we augment the "action" part of a wildcard rule with a boolean CLONE flag. If the flag is clear, the switch follows the standard wildcard behavior. Otherwise, the switch locally "clones" the wildcard rule to create a new rule in which all of the wildcarded fields are replaced by values matching this microflow, and all other aspects of the original rule are inherited. Subsequent packets for the microflow match the microflow-specific rule, and thus contribute to microflow-specific counters. Also, this rule goes into the exact-match lookup table, and thus reduces the use of the TCAM, which avoids most of the TCAM power cost [10]. This approach is similar to the proposal by Casado *et al.* [4], but their approach does per-flow lookups in the control-plane software, which might not scale to high line rates.

Local actions: Certain flow-setup decisions might require decisions intermediate between the heavyweight "invoke the controller" and the lightweight "forward via this specific port" choices offered by standard OpenFlow. In DevoFlow, we envision rules augmented with a small set of possible "local routing actions" that a switch can take without paying the costs of invoking the controller. If a switch does not support an action, it defaults to invoking the controller, so as to preserve the desired semantics.

Examples of local routing actions include:

- **Multipath support**: where the switch is given a choice of several output ports for a clonable wildcard, not just one. The switch can then select, randomly or round-robin, between these ports on each microflow arrival; the microflow-specific rule then inherits the chosen port rather than the set of ports. (This prevents intra-flow re-ordering due to path changes.)
  OpenFlow does have support for Equal-Cost MultiPath (ECMP), but as Al-Fares *et al.* point out, ECMP "can cause substantial bandwidth losses due to long-term collisions." [1].
- **Rapid re-routing**: where a switch is given one or more fallback paths to use if the designated output port goes down. If the switch can execute this decision locally, it can recover from link failures almost immediately, rather than waiting several RTTs for the central controller to first discover the failure, and then to update the forwarding rules. OpenFlow *almost* supports this already, by allowing overlapping rules with different priorities, but it does not tell the switch *why* it would have multiple rules that could match a flow, and hence we need a small change to make indicate explicitly that one rule should replace another in the case of a specific port failure.

4.2 Mechanisms for efficient statistics collection

We offer two different ways to improve the efficiency of OpenFlow statistics collection.

Use sFlow: Instead of push-based or pull-based collection (see sec. 3.4), we can use sampling. In particular, the sFlow protocol [14] allows a switch to report the headers of randomly chosen packets to a monitoring node – which could be the OpenFlow controller. Samples are uniformly chosen and typically at a rate of 1/1000 packets, although this rate is adjustable. Because sFlow reports do not include the entire packet, the incremental load on the network is less than 0.1%, and since it is possible to implement sFlow entirely in the data plane, it does not add load to a switch's CPU. In fact, sFlow is already implemented in many switches, including the ProCurve 5406zl.

Triggers and reports: Alternatively, we propose another new push-based mechanism: extending OpenFlow rules to include threshold-based *triggers* on counters. When a trigger condition is met, for any kind of rule (wildcarded or not), the switch sends a *report*, similar to the Flow-Removal message, to the controller. (It can buffer these briefly, to pack several reports into one packet.)

The simplest trigger conditions are thresholds on the three per-flow counters (packets, bytes, and flow duration). These should be easy to implement within the data plane. One could also set thresholds on packet or byte rates, but to do

so would require more state (to define the right averaging interval) and more math, and might be harder to implement.

Is the triggers+reports mechanism superior to sFlow? At this stage in our research, we are not sure, so we are investigating both options. We suspect that it might be more efficient to use asynchronous reports triggered once per elephant flow, instead of adjusting the sFlow sampling rate so that elephants are detected quickly without excess overhead. Mori *et al.* [11] suggest that it might require 5–10 samples (at a sample rate of 1/1000 packets) to reliably detect 10K-packet elephants, whereas the trigger+reports mechanism can detect arbitrary-sized elephants while sending just one packet.

How quickly do we need to detect elephant flows? Kandula *et al.* [7] found that, in their data-center, almost all bytes were carried in flows lasting longer than 10 seconds – implying that the congestion they observed probably did not come from mice – but more than half of the bytes are in flows lasting under 25 seconds; they observe that this means that traffic engineering cannot depend on scheduling just a few long-running flows.

Hedera defines a "large" flow as one taking at least 1/10 of a host's NIC bandwidth [1]. At 1Gbps and 1500B/packet, this works out to about 8333 packets/sec, implying that sampling 1/1000 packets would take on the order of a second to identify a large flow. This is fast enough to support Hedera's existing 5-second control loop, but perhaps not fast enough to support their desired sub-100 msec. loop. Since the results in the Hedera paper were based on synthetic traffic, we suspect more research is necessary to address the detection-speed question.

4.3 Avoiding the control plane

One of our goals was to avoid leaving the data plane. Here we briefly discuss whether this is feasible for the new mechanisms that we have described.

Rule cloning: Ideally, we would like the data plane to directly insert entries into the exact-match table. This might require some redesign of existing data planes; some ASIC designers have assured us that this is not impossible. Otherwise, rule cloning would require a control plane operation once per flow.

Multipath support: This requires support for rule-cloning, a random-number generator, new bit on multi-action rules to say "multipath" rather than "multicast", and perhaps a small table to hold path-choice biasing weights.

Triggers: should be a modest extension to the mechanism that updates per-flow counters. Most modern switches support either NetFlow or IPFIX, which is about half of what we need – the other half is some additional per-entry storage for trigger threshold, and an additional comparator in the pipeline.

5. PRELIMINARY RESULTS

We are currently evaluating DevoFlow via simulations. Our very preliminary results show that, when re-routing elephant flows using DevoFlow triggers, we can increase throughput over ECMP routing by 16% and 24% for Clos and HyperX networks, respectively — within 6%–8% of optimal for our workload. With a trigger on flows that reach 1MB, DevoFlow sends 86% fewer packets to the controller, and uses 75% fewer flow table entries, compared to a Hedera-like approach that collects statistics once/second.

6. SUMMARY

OpenFlow's centralized decision-making yields important benefits based on the controller's global viewpoint, but the overheads required to set up and monitor each flow could be a bottleneck. *DevoFlow* preserves the benefits of OpenFlow, while devolving decisions (as appropriate) to the switches, thereby reducing these overheads.

7. REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. NSDI*, Apr. 2010.

[2] M. Al-Fares and A. Vahdat. Pers. communication, 2010.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proc. SIGCOMM*, pages 1–12, Aug. 2007.

[4] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking Packet Forwarding Hardware. In *Proc. HotNets*, Oct. 2008.

[5] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. SOSP*, pages 15–28, 2009.

[6] A. Greenberg et al. VL2: A Scalable and Flexible Data Center Network. In *Proc. SIGCOMM*, Aug. 2009.

[7] S. Kandula, S. Sengupta, A. Greenberg, and P. Patel. The Nature of Datacenter Traffic: Measurements & Analysis. In *Proc. IMC*, 2009.

[8] W. Kim, P. Sharma, J. Lee, S. Banerjee, J. Tourrilhes, S.-J. Lee, and P. Yalagandula. Automated and Scalable QoS Control for Network Convergence. In *Proc. INM/WREN*, San Jose, CA, Apr. 2010.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[10] N. Mohan and M. Sachdev. Low-Leakage Storage Cells for Ternary Content Addressable Memories. *IEEE Trans. VLSI Sys.*, 17(5):604–612, May 2009.

[11] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto. Identifying Elephant Flows Through Periodically Sampled Packets. In *Proc. IMC*, pages 115–120, Taormina, Oct. 2004.

[12] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proc. ANCS*, pages 1–9, 2008.

[13] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic Access Control for Enterprise Networks. In *Proc. WREN*, pages 11–18, Aug. 2009.

[14] sFlow. `http://sflow.org/about/index.php`.

[15] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. In *Proc. HotNets*, NY, NY, Oct. 2009.

[16] K. Thompson, G. Miller, and R. Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network*, 11(6):10–23, Nov. 1997.

[17] A. Tootoonchian and Y. Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proc. INM/WREN*, San Jose, CA, Apr. 2010.

[18] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-Based Networking with DIFANE. In *Proc. SIGCOMM*, Aug. 2010.