DevoFlow: Scaling Flow Management for High-Performance Networks[*]

Andrew R. Curtis
University of Waterloo

Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula
Puneet Sharma, Sujata Banerjee
HP Labs — Palo Alto

ABSTRACT

OpenFlow is a great concept, but its original design imposes excessive overheads. It can simplify network and traffic management in enterprise and data center environments, because it enables flow-level control over Ethernet switching and provides global visibility of the flows in the network. However, such fine-grained control and visibility comes with costs: the switch-implementation costs of involving the switch's control-plane too often and the distributed-system costs of involving the OpenFlow controller too frequently, both on flow setups and especially for statistics-gathering.

In this paper, we analyze these overheads, and show that OpenFlow's current design cannot meet the needs of high-performance networks. We design and evaluate *DevoFlow*, a modification of the OpenFlow model which gently breaks the coupling between control and global visibility, in a way that maintains a useful amount of visibility without imposing unnecessary costs. We evaluate DevoFlow through simulations, and find that it can load-balance data center traffic as well as fine-grained solutions, without as much overhead: DevoFlow uses 10–53 times fewer flow table entries at an average switch, and uses 10–42 times fewer control messages.

Categories and Subject Descriptors.
C.2 [**Internetworking**]: Network Architecture and Design
General Terms. Design, Measurement, Performance
Keywords. Data center, Flow-based networking

1. INTRODUCTION

Flow-based switches, such as those enabled by the OpenFlow [35] framework, support fine-grained, flow-level control of Ethernet switching. Such control is desirable because it enables (1) correct enforcement of flexible policies without carefully crafting switch-by-switch configurations, (2) visibility over all flows, allowing for near optimal management of network traffic, and (3) simple and future-proof switch design. OpenFlow has been deployed at various academic institutions and research laboratories, and has been the basis for many recent research papers (e.g., [5, 29, 33, 39, 43]), as well as for hardware implementations and research prototypes from vendors such as HP, NEC, Arista, and Toroki.

[*]The version of this paper that originally appeared in the SIGCOMM proceedings contains an error in the description of Algorithm 1. This version has corrected that error.

While OpenFlow was originally proposed for campus and wide-area networks, others have made quantified arguments that OpenFlow is a viable approach to high-performance networks, such as data center networks [45], and it has been used in proposals for traffic management in the data center [5,29]. The examples in this paper are taken from data center environments, but should be applicable to other cases where OpenFlow might be used.

OpenFlow is not perfect for all settings, however. In particular, we believe that it excessively couples central control and complete 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, and experience suggests that such centralized bottlenecks are difficult to scale. Scaling the central controller has been the topic of recent proposals [11, 33, 47]. More than the controller, however, we find that the switches themselves can be a bottleneck in flow setup. Experiments with our prototype OpenFlow implementation indicate that its ratio of data-plane to control-plane bandwidth is four orders of magnitude less than its aggregate forwarding rate. We find this slow control-data path adds unacceptable latency to flow setup, and cannot provide flow statistics timely enough for traffic management tasks such as load balancing. Maintaining complete visibility in a large OpenFlow network can also require hundreds of thousands of flow table entries at each switch. Commodity switches are not built with such large flow tables, making them inadequate for many high-performance OpenFlow networks.

Perhaps, then, full control and visibility over all flows is not the right goal. Instead, we argue and demonstrate that effective flow management can be achieved by devolving control of most flows back to the switches, while the controller maintains control over only targeted *significant flows* and has visibility over only these flows and packet samples. (For example, load balancing needs to manage long-lived, high-throughput flows, known as “elephant” flows.) Our framework to achieve this, *DevoFlow*, is designed for 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—through new mechanisms to detect significant flows efficiently, by waiting until they actually become significant. DevoFlow also introduces new mechanisms to allow switches to make local routing decisions, which forward flows that do not require vetting by the controller.

The reader should note that we are not proposing any radical new designs. Rather, we are pointing out that a system like OpenFlow, when applied to high-performance 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. (We focus on OpenFlow in this work, but any centralized flow controller will likely face similar tradeoffs.)

Our goal in designing DevoFlow is to enable cost-effective, scalable flow management. Our design principles are:

- *Keep flows in the data-plane* as much as possible. Involving the control-plane in all flow setups creates too many overheads in the controller, network, and switches.
- *Maintain enough visibility over network flows* for effective centralized flow management, but otherwise provide only aggregated flow statistics.
- *Simplify the design and implementation of fast switches* while retaining network programmability.

DevoFlow attempts to resolve two dilemmas — a control dilemma:

- Invoking the OpenFlow controller on every flow setup provides good start-of-flow visibility, but puts too much load on the control plane and adds too much setup delay to latency-sensitive traffic, and
- Aggressive use of OpenFlow flow-match wildcards or hash-based routing (such as ECMP) reduces control-plane load, but prevents the controller from effectively managing traffic.

and a statistics-gathering dilemma:

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

We resolve these two dilemmas by pushing responsibility over most flows to switches and adding efficient statistics collection mechanisms to identify significant flows, which are the only flows managed by the central controller. We discuss of the benefits of centralized control and visibility in §2, so as to understand how much devolution we can afford.

Our work here derives from a long line of related work that aims to allow operators to specify high-level policies at a logically centralized controller, which are then enforced across the network without the headache of manually crafting switch-by-switch configurations [10, 12, 25, 26]. This separation between forwarding rules and policy allows for innovative and promising network management solutions such as NOX [26, 45] and other proposals [29, 39, 49], but these solutions may not be realizable on many networks because the flow-based networking platform they are built on—OpenFlow—is not scalable. We are not the first to make this observation; however, others have focused on scaling the controller, e.g., Onix [33], Maestro [11], and a devolved controller design [44]. We find that the controller can present a scalability problem, but that switches may be a greater scalability bottleneck. Removing this bottleneck requires minimal changes: slightly more functionality in switch ASICs and more efficient statistics-collection mechanisms.

This paper builds on our earlier work [36], and makes the following major contributions: we measure the costs of OpenFlow on prototype hardware and provide a detailed analysis of its drawbacks in §3, we present the design and use of DevoFlow in §4, and we evaluate one use case of DevoFlow through simulations in §5.

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.

Avoids the need to construct global policies from 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 [12, 39]. 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 [31].

However, this does not mean that *all* flow setups should be mediated by a central controller. In particular, 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 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 controller 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.* [31] 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.

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

Near-optimal traffic management: To effectively manage the performance of a network, the controller needs to know about the current loads on most network elements. Maximizing some performance objectives may also require timely statistics on some flows in the network. (This assumes that we want to exploit statistical multiplexing gain, rather than strictly controlling flow admission to prevent oversubscription.)

We give two examples where the controller is needed to manage traffic: load balancing and energy-aware routing.

Example 1: Load balancing via a controller involves collecting flow statistics, possibly down to the specific flow-on-link level. This allows the controller to re-route or throttle problematic flows, and to forecast future network loads. For example, NOX [45] "can utilize real-time information about network load ... to install flows on uncongested links."

However, load-balancing does not require the controller to be aware of 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, Al Fares *et al.* proposed Hedera, a centralized flow scheduler for data-center networks [5]. Hedera requires detection of "large" flows at the edge switches; they define "large" as 10% of the host-NIC bandwidth. The controller schedules these elephant flows, while the switches route mice flows using equal-cost multipath (ECMP) to randomize their routes.

Example 2: Energy-aware routing, where routing minimizes the amount of energy used by the network, can significantly reduce the cost of powering a network by making the network power-proportional [8]; that is, its power use is directly proportional to utilization. Proposed approaches including shutting off switch and router components when they are idle, or adapting link rates to be as minimal as possible [3,7,27,28,40]. For some networks, these techniques can give significant energy savings: up to 22% for one enterprise workload [7] and close to 50% on another [40].

However, these techniques do not save much energy on high-performance networks. Mahadevan *et al.* [34] found that, for their Web 2.0 workload on a small cluster, link-rate adaption reduced energy use by 16%, while energy-aware routing reduced it by 58%. We are not aware of a similar comparison for port sleeping vs. energy-aware routing; however, it is unlikely that putting network components to sleep could save significant amounts of energy in such networks. This is because these networks typically have many aggregation and core switches that aggregate traffic from hundreds or thousands of servers. It is unlikely that ports can be transitioned from sleep state to wake state quickly enough to save significant amounts of energy on these switches.

We conclude that some use of a central controller is necessary to build a power-proportional high-performance network. The controller requires utilization statistics for links and at least some visibility of flows in the network. Heller *et al.* [29] route all flows with the controller to achieve energy-aware routing; however, it may be possible to perform energy-aware routing without full flow visibility. Here, the mice flows should be aggregated along a set of least-energy paths using wildcard rules, while the elephant flows should be detected and re-routed as necessary, to keep the congestion on powered-on links below some safety threshold.

OpenFlow switches are relatively simple and future-proof because policy is imposed by controller software, rather than by switch hardware or firmware. Clearly, we would like to maintain this property. We believe that DevoFlow, 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. OPENFLOW OVERHEADS

Flow-based networking involves the control-plane more frequently than traditional networking, and therefore has higher overheads. Its reliance on the control-plane has intrinsic overheads: the bandwidth and latency of communication between a switch and the central controller (§3.1). It also has implementation overheads, which can be broken down into implementation-imposed and implementation-specific overheads (§3.2). We also show that hardware changes alone cannot be a cost-effective way to reduce flow-based switching overheads in the near future (§3.3).

3.1 Intrinsic overheads

Flow-based networking intrinsically relies on a communication medium between switches and the central controller. This imposes both network load and latency.

To set up a bi-directional flow on an 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 [46], and datacenter traffic has similarly short flows, with the median flow carrying only 1 KB [9,24,30]. 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 highly connected network.

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.

3.2 Implementation overheads

In this section, we examine the overheads OpenFlow imposes on switch implementations. We ground our discussion in our experience implementing OpenFlow on the HP ProCurve 5406zl [1] 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 in numerous research institutions.

While we use the 5406zl switch as an example throughout this section, the overheads we discuss are a consequence of both basic physics and of realistic constraints on the hardware that a switch vendor can throw at its implementation. The practical issues we describe are representative of those facing any OpenFlow implementation, and we believe that the 5406zl is representative of the current generation of Ethernet switches. OpenFlow also creates implementation-imposed overheads at the controller, which we describe after our discussion of the overheads incurred at switches.

3.2.1 *Flow setup overheads*

Switches have finite bandwidths between their data- and control-planes, and finite compute capacity. These issues can

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

limit the rate of flow setups—the best implementations we know of can set up only a few hundred flows per second. To estimate the flow setup rate of the ProCurve 5406zl, we attached two servers to the switch and opened the next connection from one server to the other as soon as the previous connection was established. We found that the switch completes roughly 275 flow setups per second. This number is in line with what others have reported [43].

However, this rate is insufficient for flow setup in a high-performance network. The median inter-arrival time for flows at data center server is less than 30 ms [30], so we expect a rack of 40 servers to initiate approximately 1300 flows per second—far too many to send each flow to the controller.

The switch and controller are connected by a fast physical medium, so why is the switch capable of so few flow 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 data-plane 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. The line-card ASIC in the 5406zl switch has a raw bandwidth of 300 Gbit/sec, but we measured the loopback bandwidth between the ASIC and the management CPU at just 80 Mbit/sec. This four-order-of-magnitude difference is similar to observations made by others [13].

A switch's limited internal bandwidth and wimpy CPU limits the data rate between the switch and the central controller. Using the 5406zl, we measured the bandwidth available for flow-setup payloads between the switch and the OpenFlow controller at just 17 Mbit/sec.

We also measured the latency imposed. The ASIC can forward a packet within 5 μs, but we measured a round-trip time of 0.5 ms between the ASIC and the management CPU, and an RTT of 2 ms between that CPU and the OpenFlow controller. A new flow is delayed for at least 2 RTTs (forwarding the initial packet via the controller is delayed until the flow-setup RTT is over).

This flow-setup latency is far too high for high-performance networks, where most flows carry few bytes and latency is critical. Work from machines that miss their deadline in an interactive job is not included in the final results, lowering their quality and potentially reducing revenue. As a result, adding even 1ms delay to a latency-sensitive flow is "intolerable" [6]. Also, others have observed that the delay between arrival of a TCP flow's first packet and the controller's installation of new flow-table entries can create many out-of-order packets, leading to a collapse of the flow's initial throughput [50], especially if the switch-to-controller RTT is larger than that seen by the flow's packets.

Alternative approaches minimize these overheads but lose some of the benefits of OpenFlow. DIFANE [51] avoids these overheads 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. The types of management solutions we would like to enable (e.g., [5,29]) rely on global visibility and therefore it is unlikely they can be built on top of DIFANE. Another alternative, Mahout [17], performs elephant flow classifications at the end-hosts, by looking at the TCP buffer of outgoing flows, avoiding the need to invoke the controller for mice. Since this approach requires end-host modifications, it does not meet our goal of a drop-in replacement for OpenFlow.

[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, or even unencrypted TCP, without using the switch's CPU.

3.2.2 *Gathering flow statistics*

Global flow schedulers need timely access to statistics. If a few, long-lived flows constitute the majority of bytes transferred, then a scheduler can get by collecting flow statistics every several seconds; however, this is not the case in high-performance networks, where most of the longest-lived flows last only a few seconds [30].

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 $3N+2$. The existing push-based mechanism does not inform the controller about the behavior of a flow before the entry times out, as a result, push-based statistics are not currently useful for flow scheduling.
- **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. Under ideal settings, reading the statistics for 16K exact-match rules and the 1500 wild-card rules supported on the 5406zl would return 1.3 MB; doing this twice per second would require slightly more than the 17 Mbit/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.

Pull-based statistics can be used for flow scheduling if they can be collected frequently enough. The evaluation of one flow scheduler, Hedera [5], indicates that a 5 sec. control loop (the time to pull statistics from all access switches, compute a re-routing of elephant flows, and then update flow table entries where necessary) is fast enough for near-optimal load balancing on a fat-tree topology; however, their workload is based on flow lengths following a Pareto distribution. Recent measurement studies have shown data center flow sizes do not follow a Pareto distribution [9, 24]. Using a workload with flow lengths following the distribution of flow sizes measured in [24], we find that a 5 sec. statistics-gathering interval can improve utilization only 1–5% over

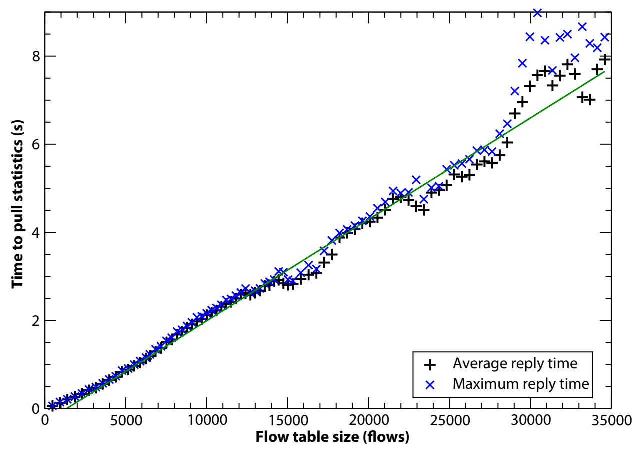

Figure 1: The latency of statistics gathering as the number of flow table entries is varied on the 5406zl Each point is the average of 10 runs, except for the maximum reply time points which are the maximum value seen out of those 10 runs. The switch was idle when the statistics were pulled.

randomized routing with ECMP (details are in §5). This is confirmed by Raiciu *et al.*, who found that the Hedera control loop needs to be less than 500ms to perform better than ECMP on their workload [41].

We measured how long it took to read statistics from the 5406zl as we varied the number of flow table entries. The results are shown in Figure 1. For this experiment (and all others in this section), we attached three servers to the switch: two servers were clients (A and B) and one was the OpenFlow controller. To get measurements with no other load on the switch, we configured the switch so that its flow table entries never expired. Then, both clients opened ten connections to each other. The controller waited 5 sec. to ensure that the switch was idle, and then pulled the statistics from the switch. This process was repeated until the flow table contained just over 32K entries.

From this experiment, we conclude that pull-based statistics cannot be collected frequently enough. We find that the statistic-gathering latency of the 5406zl is less than one second only when its flow table has fewer than 5600 entries and less than 500 ms when it has fewer than 3200 entries, and this is when there is no other load on the switch. Recall that a rack of 40 servers will initiate approximately 1300 flows per second. The default flow table entry timeout is 60 sec., so a rack's access switch can be expected to contain ∼78K entries, which, extrapolating from our measurements, could take well over 15 sec. to collect! (The 5406zl does not support such a large table.) This can be improved to 13K table entries by reducing the table entry timeout to 10 sec.; however, it takes about 2.5 sec. to pull statistics for 13K entries with no other load at the switch, which is still too long for flow schedulers like Hedera.

In short, the existing statistics mechanisms impose high overheads, and in particular they do not allow the controller to request statistics visibility only for the small fraction of significant flows that actually matter for performance.

3.2.3 *Impact on flow setup of statistics-gathering*

Statistics-gathering and flow setup compete for the limited switch-controller bandwidth—the more frequently statistics are gathered, the fewer flows the switch can set up.

We performed an experiment to measure this interference.

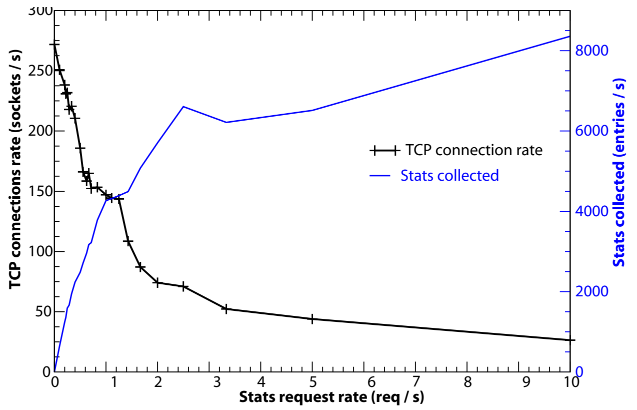

Figure 2: The flow setup rate between the clients and the number of statistics collected vs. the number of times statistics are requested per second. Each point is the average of 10 runs.

Here, each client has infinitely many flows to send to the other, and connections are established serially, that is, once one connection is established, another is opened. The client sends a single packet before closing each connection. We then vary the number of statistics-pulling requests between 0–10 requests per second.

The results are shown in Figure 2. We measured the number of connections achieved per second and the number of flow-entry statistics collected each second, as we varied the rate of requesting statistics for the entire flow table. It is clear that statistics-pulling interferes with flow setup. When statistics are never pulled, the clients can make 275 connections/sec.; when they are pulled once a second, collecting counters for just under 4500 entries, the clients can make fewer than 150 connections/sec.

3.2.4 *Switch state size*

A limited number of flow entries can be supported in hardware. The 5406zl switch hardware can support about 1500 OpenFlow rules, whereas the switch can support up to 64000 forwarding entries for standard Ethernet switching. One reason for this huge disparity is that OpenFlow rules are stored in a TCAM, necessary to support OpenFlow's wildcarding mechanism, and TCAM entries are an expensive resource, whereas Ethernet forwarding uses a simple hash lookup in a standard memory. It is possible to increase the number of TCAM entries, but TCAMs consume lots of ASIC space and power. Also, an OpenFlow rule is described by 10 header fields, which total 288 bits [35], whereas an Ethernet forwarding descriptor is 60 bits (48-bit MAC + 12-bit VLAN ID); so, even when fully optimized, OpenFlow entries will always use more state than Ethernet forwarding entries.

Finally, because OpenFlow rules are per-flow, rather than per-destination, each directly-connected host will typically require an order of magnitude more rules. (As we mentioned, an average ToR switch might have roughly 78,000 flow rules if the rule timeout is 60 sec.) 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. Table 1 summarizes these limits.

Forwarding scheme	Descriptor size	Possible entries on 5406zl	Entries per active host
Ethernet learning	60 bits	∼64000	1
OpenFlow	288 bits	∼1500	10 (typical)

Table 1: State sizes: OpenFlow vs. Ethernet learning

3.2.5 *Implementation-imposed controller overheads*

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. For example, Tavakoli *et al.* [45] report that one NOX controller can handle "at least 30K new flow installs per second while maintaining a sub-10 ms flow install time ... The controller's CPU is the bottleneck." Kandula *et al.* [30] found that 100K flows arrive every second on a 1500-server cluster, implying a need for multiple OpenFlow controllers.

Recently researchers have proposed more scalable OpenFlow controllers. Maestro [11] is a multi-threaded controller that can install about twice as many flows per second as NOX, without additional latency. Others have worked on distributed implementations of the OpenFlow controller (also valuable for fault tolerance) These include HyperFlow (Tootoonchian and Ganjali [47]) and Onix (Koponen *et al.* [33]). These distributed controllers can only support global visibility of rare events such as link-state changes, and not of frequent events such as flow arrivals. As such, they are not yet suitable for applications, such as Hedera [5], which need a global view of flow statistics.

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 these numbers, but throwing hardware at the problem adds more cost and power consumption. Also, Moore's law for hardware won't provide much relief, as Ethernet speeds have been increasing at least as fast as Moore's law over the long term. Adding a faster CPU to the switch may improve control-plane bandwidth, but is unlikely to provide significant improvements without a bigger datapath and reorganized memory hierarchy. We believe such changes are likely to be complicated and expensive, especially because, for high-performance workloads, OpenFlow needs significantly more bandwidth between the data-plane and the control-plane than switches normally support (see §5.3). We expect this bandwidth gap will shrink as ASIC designers pay more attention to OpenFlow, but we do not think they will let OpenFlow performance drive their chip-area budgets for several generations, at least. And while Moore's Law might ameliorate the pressure somewhat, the need to reduce both ASIC cost and energy consumption suggests that hardware resources will always be precious.

An alternative implementation path is to use software-based OpenFlow switch implementations on commodity server hardware [18]. Such switches may have more control-plane bandwidth, but we do not believe these systems will be cost-effective for most enterprise applications in the foreseeable future. Casado *et al.* [13] have also argued that "network processors" are not ideal, either.

4. DEVOFLOW

We now present the design of DevoFlow, which avoids the overheads described above by introducing mechanisms for efficient devolved control (§4.1) and statistics collection (§4.2). Then, we discuss how to implement these mechanisms (§4.4), and end with an example of using DevoFlow to reduce use of the control-plane (§4.4).

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, reducing the use of the TCAM, and so avoiding most of the TCAM power cost [37]. This resembles the proposal by Casado *et al.* [13], but their approach does per-flow lookups in 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 actions include multipath support and rapid re-routing:

- **Multipath support** gives the switch a choice of several output ports for a clonable wildcard, not just one. The switch can then select, randomly from a probability distribution 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.)

  This functionality is similar to equal-cost multipath (ECMP) routing; however, multipath wildcard rules provide more flexibility. ECMP (1) uniformly selects an output port uniformly at random and (2) requires that the cost of the multiple forwarding paths to be equal, so it load balances traffic poorly on irregular topologies. As an example, consider a topology with two equal-cost links between s and t, but the first link forwards at 1 Gbps whereas the second has 10 Gbps capacity. ECMP splits flows evenly across these paths, which is clearly not ideal since one path has 10 times more bandwidth than the other.

  DevoFlow solves this problem by allowing a clonable wildcard rule to select an output port for a microflow according to some probability distribution. This allows implementation of *oblivious routing* (see, e.g., [20, 32]), where a microflow follows any of the available end-to-end paths according to a probability distribution. Oblivious routing would be optimal for our previous example, where it would route 10/11th of the microflows for t on the 10Gbps link and 1/11th of them on the 1Gbps link.

- **Rapid re-routing** gives a switch 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 Efficient statistics collection

DevoFlow provides three different ways to improve the efficiency of OpenFlow statistics collection.

Sampling is an alternative to either push-based or pull-based collection (see §3.2.2). In particular, the sFlow protocol [42] 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, typically at a rate of 1/1000 packets, although this 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: extends OpenFlow with a new push-based mechanism: 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.

Approximate counters: can be maintained for all microflows that match a forwarding-table rule. Such counters maintain a view on the statistics for the top-k largest (as in, has transferred the most bytes) microflows in a space-efficient manner. Approximate counters can be implemented using streaming algorithms [21, 22, 23], which are generally simple, use very little memory, and identify the flows transferring the most bytes with high accuracy. For example, Golab *et al.*'s algorithm [23] correctly classifies 80–99% of the flows that transfer more than a threshold k of bytes. Implementing approximate counters in the ASIC is more difficult than DevoFlow's other mechanisms; however, they provide a more timely and accurate view of the network and can keep statistics on microflows without creating a table entry per microflow.

4.3 Implementation feasibility of DevoFlow

We have not implemented DevoFlow in hardware; however, our discussions with network hardware designers indicate that all of DevoFlow's mechanisms can be implemented cost-effectively in forthcoming production switches. (Sampling using sFlow, as we noted earlier, is already widely implemented in switch data-planes.)

Rule cloning: The data-plane needs to to directly insert entries into the exact-match table. This is similar to the existing MAC learning mechanism, the switch ASIC would need to be modified to take into account the formatting of entries in the flow table when learning new flows. If ASIC modification is not possible, rule cloning would require involving a CPU once per flow. Even so, this operation is considerably cheaper than invoking the centralized controller and should be orders of magnitude faster.

Multipath support: Can be implemented using specialized rule cloning or using a virtual port. Both methods require a small table to hold path-choice biasing weights for each multipath rule and a random number generator. The virtual port method is actually similar to link aggregation groups (LAG) and ECMP support, and could reuse the existing functional block with trivial modification.

Triggers: The mechanism needed to support triggers requires a counter and a comparator. It is similar to the one needed for rate limiters, and in some cases existing flexible rate limiters could be used to generate triggers. Most modern switches support a large number of flow counters, used for OpenFlow, NetFlow/IPFIX or ACLs. The ASIC would need to add a comparator to those counters to generate triggers; alternatively, the local CPU could poll periodically those counters and generate triggers itself.

Approximate counters: is the mechanism that would require the most extensive changes to current ASICs. It requires hashing on packet headers, which indirect to a set of counters, and then incrementing some of those counters. Switch ASICs have existing building blocks for most of these functions. It would also be non-trivial to support triggers on approximate counters; this might require using the local CPU.

4.4 Using DevoFlow

All OpenFlow solutions can be built on top of DevoFlow; however, DevoFlow enables scalable implementation of these solutions by reducing the number of flows that interact with the control-plane. Scalability relies on a finding a good definition of "significant flows" in a particular domain. These flows should represent a small fraction of the total flows, but should be sufficient to achieve the desired results.

As an example, we show how to load balance traffic with DevoFlow. First, we describe scalable flow scheduling with DevoFlow's mechanisms. Then, we describe how to use its multipath routing to statically load balance traffic without any use of the control-plane.

Flow scheduling: does not scale well if the scheduler relies on visibility over all flows, as is done in Hedera [5] because maintaining this visibility via the network is too costly, as our experiments in §3 showed.

Instead, we maintain visibility only over elephant flows, which is all that a system such as Hedera actually needs. While Hedera defines an elephant as a flow using at least

10% of a NIC's bandwidth, we define one as a flow that has transferred at least a threshold number of bytes X. A reasonable value for X is 1–10MB.

Our solution starts by initially routing incoming flows using DevoFlow's multipath wildcard rules; this avoids involving the control-plane in flow setup. We then detect elephant flows as they reach X bytes transferred. We can do this using using any combination of DevoFlow's statistics collection mechanisms. For example, we can place triggers on flow table entries, which generate a report for a flow after it has transferred X bytes; We could also use sampling or approximate counters; we evaluate each approach in §5.

Once a flow is classified as an elephant, the detecting switch or the sampling framework reports it to the DevoFlow controller. The controller finds the least congested path between the flow's endpoints, and re-routes the flow by inserting table entries for the flow at switches on this path.

The new route can be chosen, for example, by the decreasing best-fit bin packing algorithm of Correa and Goemans [16]. The algorithm's inputs are the network topology, link utilizations, and the rates and endpoints of the elephant flows. Its output is a routing of all elephant flows. Correa and Goemans proved that their algorithms finds routings with link utilizations at most 10% higher than the optimal routing, under a traffic model where all flows can be rearranged. We cannot guarantee this bound, because we only rearrange elephant flows; however, their theoretical results indicates their algorithm will perform as well as any other heuristic for flow scheduling.

Finally, we note that this architecture uses only edge switches to encapsulate new flows to send to the central controller. The controller programs core and aggregation switches reactively to flow setups from the edge switches. Therefore, the only overhead imposed is cost of installing flow table entries at the the core and aggregation switches—no overheads are imposed for statistics-gathering.

Static multipath routing: provides effective data-plane multipath load balancing with far greater flexibility than ECMP. By allowing clonable wildcard rules to select an output port for a microflow according to some probability distribution, we can implement *oblivious routing*, where an *s-t* microflow's path is randomly selected according to a precomputed probability distribution. This static routing scheme sets up these probability distributions so as to optimize routing any traffic matrix in a specified set; for example, in a data center one would generally like to optimize the routing of all "hose" traffic matrices [19], which is the set of all traffic matrices allowable as long as no end-host's ingress or egress rate exceeds a predefined rate.

Oblivious routing gives comparable throughput to the optimal dynamic routing scheme on many topologies. Kodialam *et al.* [32] found that packet-level oblivious routing achieves at least 94% of the throughput that dynamic routing does on the worst-case traffic matrix, for several wide-area network topologies.

However, these results assume that microflows can be split across multiple paths. While the flow-level multipath we implement with clonable wildcard rules does not conform to this assumption, we expect the theoretical results to be indicative of what to expect from flow-level oblivious routing on arbitrary topologies, just as it indicates the possible performance of oblivious routing on a Clos topology. Overall, the performance depends on the workload, as our results in §5 show. If oblivious routing does not achieve adequate performance on a particular topology and workload, it can be combined with DevoFlow's flow scheduler (described just above) to maximize utilization.

Finally, finding an oblivious routing is easy—one can be computed for any topology using linear programming [20]. Should the topology change, the forwarding probability distributions will need to be modified to retain optimality. Distributions for failure scenarios can be precomputed, and pushed to the switches once the central controller learns of a failure.

Algorithm 1 — Flow rate computation.

```
Input: set of flows $F$ and a set of ports $\mathcal{P}$
Output: a rate $r(f)$ of each flow $f \in F$

begin
Initialize: $F_a = \emptyset$; $\forall f, r(f) = 0$
Define: $P.\text{used}() = \sum_{f \in F_a \cap P} r(f)$
Define: $P.\text{unassigned\_flows}() = P - (P \cap F_a)$
while $\mathcal{P} \neq \emptyset$ do
    Sort $\mathcal{P}$ in ascending order, where the sort key
            for $P$ is $(P.\text{rate} - P.\text{used}())/|P.\text{unassigned\_flows}()|$
    $P = \mathcal{P}.\text{pop\_front}()$
    for each $f \in P.\text{unassigned\_flows}()$ do
        $r(f) = (P.\text{rate} - P.\text{used}())/|P.\text{unassigned\_flows}()|$
        $F_a = F_a \cup \{f\}$
end
```

5. EVALUATION

In this section, we present our simulated evaluation of DevoFlow. We use load balancing as an example of how it can achieve the same performance as fine-grained, OpenFlow-based flow scheduling without the overhead.

5.1 Simulation methodology

To evaluate how DevoFlow would work on a large-scale network, we implemented a flow-level data center network simulator. This fluid model captures the overheads generated by each flow and the coarse-grained behavior of flows in the network. The simulator is event-based, and whenever a flow is started, ended, or re-routed, the rate of all flows is recomputed using the algorithm shown in Algorithm 1. This algorithm works by assigning a rate to flows traversing the most-congested port, and then iterating to the next most-congested port until all flows have been assigned a rate.

We represent the network topology with a capacitated, directed graph. For these simulations, we used two topologies: a three-level Clos topology [15] and a two-dimensional HyperX topology [4]. In both topologies, all links were 1Gbps, and 20 servers were attached to each access switch. The Clos topology has 80 access switches (each with 8 uplinks), 80 aggregation switches, and 8 core switches. The HyperX topology is two-dimensional and forms a 9×9 grid, and so has 81 access switches, each attached to 16 other switches. Since the Clos network has 8 core switches, it is 1:2.5 oversubscribed; that is, its bisection bandwidth is 640 Gbps. bandwidth. The HyperX topology is 1:4 oversubscribed and thus has 405 Gbps of bisection bandwidth.

The Clos network has 1600 servers and the HyperX network has 1620. We sized our networks this way for two reasons: first, so that the Clos and HyperX networks would have nearly the same number of servers. Second, our workload is based on the measurements of Kandula *et al.* [30], which are from a cluster of 1500 servers. We are not sure how to scale their measurements up to much larger data centers, so we kept the number of servers close to the number measured in their study.

We simulate the behavior of OpenFlow at switches by modeling (1) switch flow tables, and (2) the limited data-plane to control-plane bandwidth. Switch flow tables can contain both exact-match and wildcard table entries. For all simulations, table entries expire after 10 seconds. When a flow arrives that does not much a table entry, the header of its first packet is placed in the switch's data-plane to control-plane queue. The service rate for this queue follows our measurements described in Section 3.2.1, so it services packets at 17Mbps. This queue has finite length, and when it is full, any arriving flow that does not match a table entry is dropped. We experimented with different lengths for this queue, and we found that when it holds 1000 packets, no flow setups were dropped. When we set its limit to 100, we found that fewer than 0.01% of flow setups were dropped in the worst case. For all results shown in this paper, we set the length of this queue to 100; we restart rejected flows after a simulated TCP timeout of 300 ms.

Finally, because we are interested in modeling switch overheads, we do not simulate a bottleneck at the OpenFlow controller; the simulated OpenFlow controller processes all flows instantly. Also, whenever the OpenFlow controller reroutes a flow, it installs the flow-table entries without any latency.

5.1.1 Workloads

We consider two workloads in our simulations: (1) a MapReduce job that has just gone into its shuffle stage, and (2) a workload based on measurements, by Kandula *et al.* at Microsoft Research (MSR) [30], of a 1500-server cluster.

The MapReduce-style traffic is modeled by randomly selecting n servers to be part of the reduce-phase shuffle. Each of these servers transfers 128 MB to each other server, by maintaining connections to k other servers at once. Each server randomizes the order it connects to the other servers, keeping k connections open until it has sent its payload. All measurements we present for this shuffle workload are for a one-minute period that starts 10 sec. after the shuffle begins.

In our MSR workload, we generated flows based on the distributions of flow inter-arrival times and flow sizes in [30]. We attempted to reverse-engineer their actual workload from only two distributions in their paper. In particular, we did not model dependence between sets of servers. We pick the destination of a flow by first determining whether the flow is to be an inter- or intra-rack flow, and then selecting a destination uniformly at random between the possible servers. For these simulations, we generated flows for four minutes, and present measurements from the last minute.

Additionally, we simulated a workload that combines the MSR and shuffle workloads, by generating flows according to both workloads simultaneously. We generated three minutes of MSR flows before starting the shuffle. We present measurements for the first minute after the shuffle began.

5.1.2 Schedulers

We compare static routing with ECMP to flow scheduling with several schedulers.

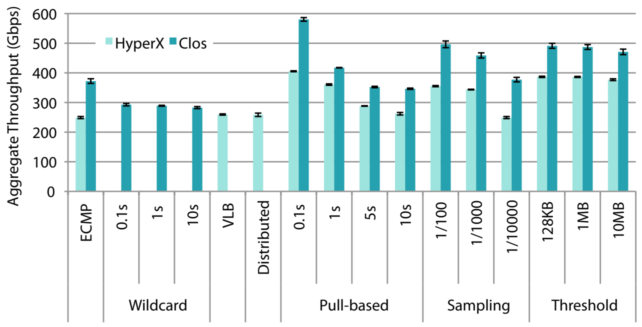

Figure 3: Throughput achieved by the schedulers for the shuffle workload with $n = 800$ and $k = 5$. OpenFlow-imposed overheads are not modeled in these simulations. All error bars in this paper show 95% confidence intervals for 10 runs.

The DevoFlow scheduler: behaves as described in Sec. 4, and collects statistics using either sampling or threshold triggers on multipath wildcard rules. The scheduler might re-reroute a flow after it has classified the flow as an elephant. New flows, before they become elephant flows, are routed using ECMP regardless of the mechanism to detect elephant flows. When the controller discovers an elephant flow, it installs flow-table entries at the switches on the least-congested path between the flow's endpoints. We model queueing of a flow between the data-plane and control-plane before it reaches the controller; however, we assume instantaneous computation at the controller and flow-table installations.

For elephant detection, we evaluate both sampling and triggers.

Our flow-level simulation does not simulate actual packets, which makes modeling of packet sampling non-trivial. In our approach:

1. We estimate the distribution of packets sent by a flow before it can be classified, with less than a 10% false-positive rate, as an elephant flow, using the approach described by Mori *et al.* [38].
2. Once a flow begins, we use that distribution to select how many packets it will transfer before being classified as an elephant; we assume that all packets are 1500 bytes. We then create an event to report the flow to the controller once it has transferred this number of packets.

Finally, we assume that the switch bundles 25 packet headers into a single report packet before sending the samples to the controller; this reduces the packet traffic without adding significant delay. Bundling packets this way adds latency to the arrival of samples at the controller. For our simulations, we did not impose a time-out this delay. We bundled samples from all ports on a switch, so when a 1 Gbps port is the only active port (and assuming it's fully loaded), this bundling could add up to 16 sec. of delay until a sample reaches the controller, when the sample rate is 1/1000 packets.

Fine-grained control using statistics pulling: simulates using OpenFlow in active mode. Every flow is set up at the central controller and the controller regularly pulls statistics, which it uses to schedule flows so as to maximize throughput. As with the DevoFlow scheduler, we route elephant flows using Correa and Goeman's bin-packing algorithm [16]. Here, we use Hedera's definition of an elephant flow: one with a demand is at least 10% of the NIC rate [5]. The rate of each flow is found using Algorithm 1 on an ideal network; that is, each access switch has an infinite-capacity uplink to a sin-

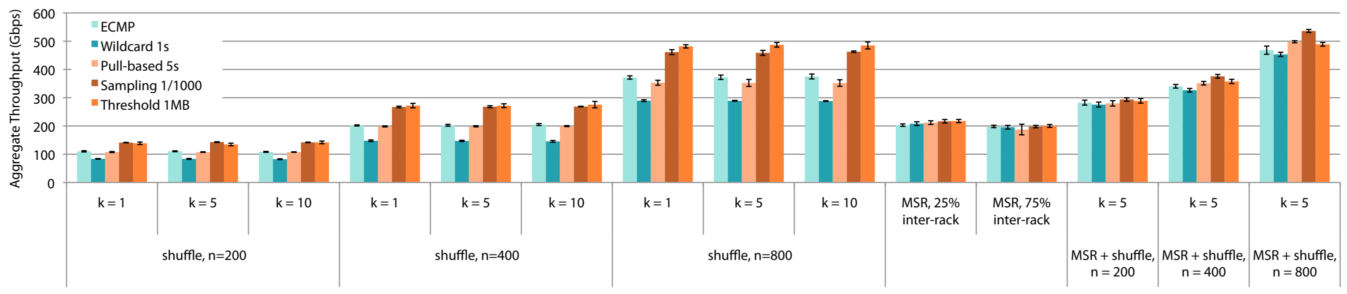

Figure 4: Aggregate throughput of the schedulers on the Clos network for different workloads. For the MSR plus shuffle workloads, 75% of the MSR workload-generated flows are inter-rack.

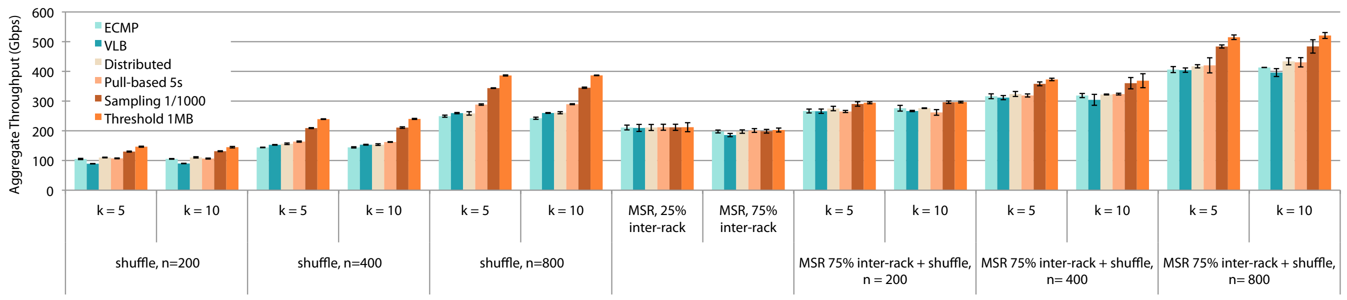

Figure 5: Aggregate throughput of the schedulers on the HyperX network for different workloads.

gle, non-blocking core switch. This allows us to estimate the demand of each flow when flow rates are constrained only by server NICs and not by the switching fabric.

Following the OpenFlow standard, each flow table entry provides 88 bytes of statistics [2]. We collect statistics only from the access switches. The ASIC transfers statistics to the controller at 17 Mbps, via 1500-byte packets. The controller applies the bin-packing algorithm immediately upon receiving a statistics report, and instantaneous installs a globally optimized routing for all flows.

Wildcard routing: performs multipath load balancing possible using only wildcard table entries. This controller reactively installs wildcard rules to create a unique spanning tree per destination: all flows destined to a server are routed along a spanning tree. When a flow is set up, the controller computes the least-congested path from the switch that registered the flow to the flow's destination's spanning tree, and installs the rules along this path. We simulated wildcard routing only on the Clos topology, because we are still developing the spanning tree algorithm for HyperX networks.

Valiant load balancing (VLB): balances traffic by routing each flow through an intermediate switch chosen uniformly at random; that switch then routes the flow on the shortest path to its destination [48]. On a Clos topology, ECMP implements VLB.

Distributed greedy routing: routes each flow by first greedily selecting the least-congested next-hop from the access switch, and then using shortest-path routing. We simulate this distributed routing scheme only on HyperX networks.

5.2 Performance

We begin by assessing the performance of the schedulers, using the aggregate throughput of all flows in the network as our metric. Figure 3 shows the performance of the schedulers under various settings, on a shuffle workload with $n = 800$ servers and $k = 5$ simultaneous connections/server. This simulation did *not* model the OpenFlow-imposed overheads; for example, the 100ms pull-based scheduler obtains all flow statistics every 100ms, regardless of the switch load.

We see that DevoFlow can improve throughput compared to ECMP by up to 32% on the Clos network and up to 55% on the HyperX network. The scheduler with the best performance on both networks is the pull-based scheduler when it re-routes flows every 100 ms. This is not entirely surprising, since this scheduler also has the highest overhead. Interestingly, VLB did not perform any better than ECMP on the HyperX network.

To study the effect of the workload on these results, we tried several values for n and k in the shuffle workload and we varied the fraction of traffic that remained within a rack on the MSR workload. These results are shown in Figure 4 for the Clos topology and Figure 5 for the HyperX network. Overall, we found that flow scheduling improves throughput for the shuffle workloads, even when the network has far more bisection bandwidth than the job demands.

For instance, with $n = 200$ servers, the maximum demand is 200 Gbps. Even though the Clos network has 640 Gbps of bisection bandwidth, we find that DevoFlow can increase performance of this shuffle by 29% over ECMP. We also observe that there was little difference in performance when we varied k.

Flow scheduling did not improve the throughput of the MSR workload. For this workload, regardless of the mix of inter- and intra-rack traffic, we found that ECMP achieves 90% of the optimal throughput[3] for this workload, so there is little room for improvement by scheduling flows. We suspect that a better model than our reverse-engineered distributions of the MSR workload would yield different results.

Because of this limitation, we simulated a combination of the MSR workload with a shuffle job. Here, we see improvements in throughput due to flow scheduling; however, the gains are less than when the shuffle job is ran in isolation.

[3]We found the optimal throughput by attaching all servers to a single non-blocking switch.

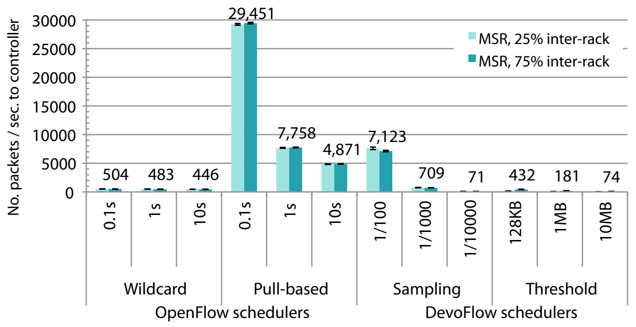

Figure 6: The number of packet arrivals per second at the controller using the different schedulers on the MSR workload.

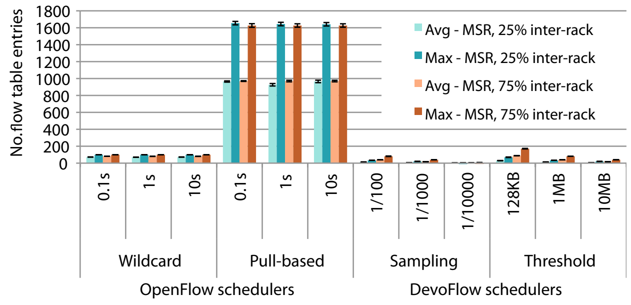

Figure 7: The average and maximum number of flow table entries at an access switch for the schedulers using the MSR workload.

5.3 Overheads

We used the MSR workload to evaluate the overhead of each approach because, even though we do not model the dependence between servers, we believe it gives a good indication of the rate of flow initiation. Figure 6 shows, for each scheduler, the rate of packets sent to the controller while simulating the MSR workload.

Load at the controller should scale proportionally to the number of servers in the data center. Therefore, when using an OpenFlow-style pull-based scheduler that collects stats every 100ms, in a large data center with 160K servers, we would expect a load of about 2.9M packets/sec., based on extrapolation from Figure 6. This would drop to 775K packets/sec. if stats are pulled once per second. We are not aware of any OpenFlow controller that can handle this message rate; for example, NOX can process 30K flow setups per second [45]. A distributed controller might be able to handle this load (which would require up to 98 NOX controllers, assuming they can be perfectly distributed and that statistics are pulled every 100 ms), but it might be difficult to coordinate so many controllers.

Figure 7 shows the number of flow table entries at any given access switch, for the MSR workload and various schedulers. For these simulations, we timed out the table entries after 10 sec. As expected, DevoFlow does not require many table entries, since it uses a single wildcard rule for all mice flows, and stores only exact-match entries for elephant flows. This does, however, assume support for the multipath routing wildcard rules of DevoFlow. If rule cloning were used instead, DevoFlow would use the same number of table entries as the pull-based OpenFlow scheduler because it would clone a rule for each flow. The pull-based scheduler uses an order of magnitude more table entries, on average, than DevoFlow.

We estimated the amount bandwidth required between a switch's data-plane and control-plane when statistics are collected with a pull-based mechanism. Figure 8 shows the bandwidth needed so that the 95th and 99th percentile flow setup latencies on the MSR workload are less than 2ms. Here, we assume that the only latency incurred is in the queue between the switch's data-plane and control-plane; we ignore any latency added by communication with the controller. That is, the figure shows the service rate needed for this queue, in order to maintain a waiting time of less than 2 ms in the 95th and 99th percentiles. The data to control-plane bandwidth sufficient for flow setup is directly proportional to this deadline, so a tighter deadline of 1 ms needs twice as much bandwidth to meet.

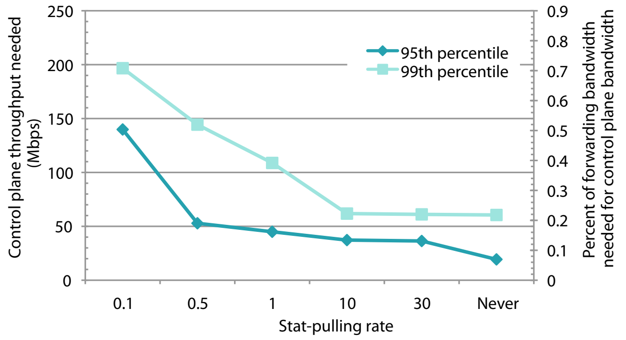

Figure 8: The control-plane bandwidth needed to pull statistics at various rates so that flow setup latency is less than 2ms in the 95th and 99th percentiles. Error bars are too small to be seen.

The scale on the right of the chart normalizes the required data-to-control-plane bandwidth to a switch's total forwarding rate (which in our case is 28 Gbps, because each ToR switch has 28 gigabit ports). For fine-grained (100 ms) flow management using OpenFlow, this bandwidth requirement would be up to 0.7% of its total forwarding rate. Assuming that the amount of control-plane bandwidth needed scales with the forwarding rate, a 144-port 10 Gbps switch needs just over 10 Gbps of control-plane bandwidth to support fine-grained flow management. We do not believe it is cost-effective to provide so much bandwidth, so DevoFlow's statistics-collection mechanisms are the better option because they are handled entirely within the data-plane.

6. CONCLUSIONS

Flow-based networking frameworks such as OpenFlow hold great promise—they separate policy specification from its realization, and therefore enable innovative network management solutions. However, we have shown that OpenFlow's current design does not meet the demands of high-performance networks. In particular, OpenFlow involves the controller in the handling of too many microflows, which creates excessive load on the controller and switches.

Our DevoFlow proposal allows operators to target only the flows that matter for their management problem. DevoFlow reduces the switch-internal communication between control- and data-planes by (a) reducing the need to transfer statistics for boring flows, and (b) potentially reducing the need to invoke the control-plane for most flow setups. It therefore reduces both the intrinsic and implementation overheads of flow-based networking, by reducing load on the network, the switch control-plane, and the central controller. DevoFlow handles most microflows in the data-plane, and therefore allows us to make the most out of switch resources. Our evaluation shows that DevoFlow performs as well as fine-grained flow management when load balancing traffic in the data center. Beyond this use case, we believe that DevoFlow can simplify the design of high-performance OpenFlow switches and enable scalable management architectures to be built on OpenFlow for data center QoS, multicast, routing-as-a-service [14], network virtualization [43], and energy-aware routing [29].

Acknowledgments

We would like to thank Joe Curcio, Charles Clark, Paul Congdon, Mark Gooch, and many others from HP Networking for helping us to understand how real switches are designed. Our shepherd, Paul Barford, and the anonymous reviewers gave us helpful advice on comments for the final version of this paper. S. Keshav, Alex López-Ortiz, and Earl Oliver gave us beneficial feedback on earlier drafts. We thank Brent Stephens for pointing out an error in our description of Algorithm 1 in a previous version of this paper.

7. REFERENCES

[1] HP ProCurve 5400 zl switch series. http://h17007.www1.hp.com/us/en/products/switches/HP_E5400_zl_Switch_Series/index.aspx.

[2] OpenFlow Switch Specification, Version 1.0.0. http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf.

[3] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy proportional datacenter networks. In *ISCA*, 2010.

[4] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: topology, routing, and packaging of efficient large-scale networks. In *Proc. Supercomputing*, 2009.

[5] 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.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. DCTCP: Efficient packet transport for the commoditized data center. In *SIGCOMM*, 2010.

[7] G. Ananthanarayanan and R. H. Katz. Greening the switch. In *USENIX Workshop on Power Aware Computing and Systems, (HotPower 2008)*, 2008.

[8] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[9] T. Benson, A. Akella, and D. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. IMC*, 2010.

[10] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *NSDI*, 2005.

[11] Z. Cai, A. L. Cox, and T. S. E. Ng. Maestro: A System for Scalable OpenFlow Control. Tech. Rep. TR10-08, Rice University, 2010.

[12] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *SIGCOMM*, pages 1–12, Aug. 2007.

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

[14] C.-C. Chen, L. Yuan, A. Greenberg, C.-N. Chuah, and P. Mohapatra. Routing-as-a-service (RaaS): A framework for tenant-directed route control in data center. In *INFOCOM*, 2011.

[15] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(5):406–424, 1953.

[16] J. R. Correa and M. X. Goemans. Improved bounds on nonblocking 3-stage clos networks. *SIAM J. Comput.*, 37(3):870–894, 2007.

[17] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM*, 2011.

[18] 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.

[19] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *SIGCOMM*, 1999.

[20] T. Erlebach and M. Rüegg. Optimal bandwidth reservation in hose-model VPNs with multi-path routing. In *IEEE INFOCOM*, 2004.

[21] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *SIGCOMM*, 2002.

[22] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*, 1998.

[23] L. Golab, D. DeHaan, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro. Identifying frequent items in sliding windows over on-line packet streams. In *IMC*, 2003.

[24] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. K. P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[25] A. Greenberg *et al.*. A clean slate 4D approach to network control and management. *SIGCOMM CCR*, 35:41–54, 2005.

[26] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. In *SIGCOMM CCR*, July 2008.

[27] M. Gupta, S. Grover, and S. Singh. A feasibility study for power management in LAN switches. In *ICNP*, 2004.

[28] M. Gupta and S. Singh. Using low-power modes for energy conservation in ethernet LANs. In *INFOCOM Mini-Conference*, 2007.

[29] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: saving energy in data center networks. In *NSDI*, 2010.

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

[31] 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*, 2010.

[32] M. Kodialam, T. V. Lakshman, and S. Sengupta. Maximum throughput routing of traffic in the hose model. In *Infocom*, 2006.

[33] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *OSDI*, 2010.

[34] P. Mahadevan, P. Sharma, S. Banerjee, and P. Ranganathan. Energy aware network operations. In *Proc. 12th IEEE Global Internet Symp.*, 2009.

[35] 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.

[36] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee. Devoflow: Cost-effective flow management for high performance enterprise networks. In *HotNets*, 2010.

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

[38] 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.

[39] 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.

[40] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *NSDI*, 2008.

[41] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, and M. Handley. Data center networking with multipath TCP. In *HotNets*, 2010.

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

[43] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *OSDI*, 2010.

[44] A. S.-W. Tam, K. Xi, and H. J. Chao. Use of Devolved Controllers in Data Center Networks. In *INFOCOM Workshop on Cloud Computing*, 2011.

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

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

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

[48] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *STOC*, 1981.

[49] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Hot-ICE*, 2011.

[50] C. Westphal. Personal communication, 2011.

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