Networking 2013 1569702637

SwitchReduce : Reducing Switch State and Controller Involvement in OpenFlow Networks

Aakash S. Iyer * , Vijay Mann * Naga Rohit Samineni †,
* IBM Research - India
Email:{aakiyer1,vijamann}@in.ibm.com
†Indian Institute of Technology, Guwahati, India
Email:s.naga@iitg.ac.in

***Abstract*—OpenFlow is a popular network architecture where a logically centralized controller (the control plane) is physically decoupled from all forwarding switches (the data plane). Through this controller, the OpenFlow framework enables flow level granularity in switches thereby providing monitoring and control over each individual flow. Among other things, this architecture comes at the cost of placing significant stress on switch state size and overburdening the controller in various traffic engineering scenarios such as dynamic re-routing of flows. Storing a flow match rule and flow counter at every switch along a flow's path results in many thousands of entries per switch. Dynamic re-routing of a flow, either in an attempt to utilize less congested paths, or as a consequence of virtual machine migration, results in controller intervention at every switch along the old and new paths. In the absence of careful orchestration of flow storage and controller involvement, OpenFlow will be unable to scale to anticipated production data center sizes.**

In this context, we present SwitchReduce - a system to reduce switch state and controller involvement in OpenFlow networks. SwitchReduce is founded on the observation that the number of flow match rules at any switch should be no more than the set of unique processing actions it has to take on incoming flows. Additionally, the flow counters for every unique flow may be maintained at only one switch in the network. We have implemented SwitchReduce as a NOX controller application. Simulation results with real data center traffic traces reveal that SwitchReduce can reduce flow entries by up to approximately 49% on first hop switches, and up to 99.9% on interior switches, while reducing flow counters by 75% on average.

Keywords: Datacenter networks,OpenFlow, Routing and traffic engineering, New networking architectures

I. Introduction

The OpenFlow [1] architecture physically decouples the control plane from the data plane in a network. A logically centralized controller (the control plane) independently controls every single flow in the network by installing customized flow-rules in forwarding switches (the data plane). A flow-rule comprises of a Match field that matches the given flow, an Instructions field that details the actions to be taken on the flow, and Counters that maintain flow statistics. OpenFlow therefore, via Counters, also enables fine-grained monitoring of traffic from every individual flow. This architecture has the distinct advantage of providing central visibility into the network, thereby allowing various traffic engineering schemes [2] to be implemented, as well as centralized control, allowing security schemes [3] and networks policies to be microscopically enforced.

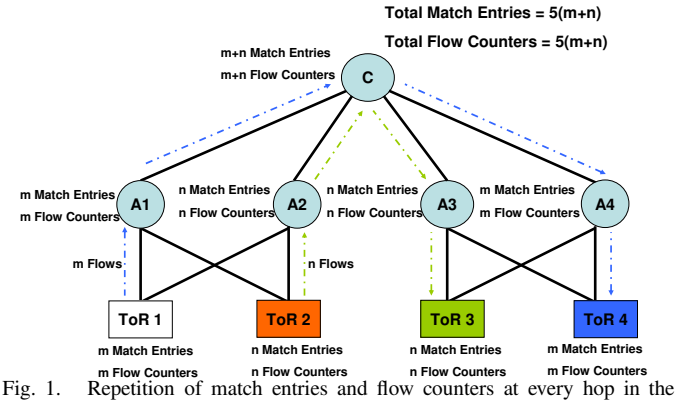

Fig. 1. Repetition of match entries and flow counters at every hop in the network

OpenFlow, with this centralized control and flow-level granularity, offers an attractive design option for data center networks. However, there are several important concerns such as increased switch memory requirement, controller bottleneck and high first packet latency which must be addressed before OpenFlow can be deployed at the scale of production data centers.

Increased Switch Memory requirements : Flow-level granularity places enormous stress on switch state size. Storing a flow rule for every flow that passes through a switch results in a linear increase in switch state with the number of flows through it. According to Curtis et al. [4], a Top-of-Rack (ToR) switch might have roughly 78,000 flow rules (if the rule timeout is 60 seconds). This means each ToR switch will store 78,000 Match Entries and 78,000 flow counters. This number is likely to be even higher on Aggregation and Core switches. With regards to flow counters especially, it is straightforward to observe that there is a lot of repetition throughout the network. As an example, if a flow passes through 4 hops, then there are 4 switches in the network where counters are provisioned for this flow. If there are 1,000,000 flows in the network, for an average of 4 hops per flow, there are 4,000,000 counters throughout the network. This increases RAM requirements of switches and contributes to bloated switch state.

Figure 1 demonstrates with a simple example how the total number of rules and counters in the network increase linearly with number of hops. Two sets of flows (m and n) traverse from the left side nodes to the right side nodes through the core of the network, with each flow requiring 5 hops. While the total number of flows are only (m+n), flow-level granularity results in a total of 5(m+n) match entries and 5(m+n) flow counters in the network.

Failure to compress switch state will render large scale OpenFlow data centers infeasible. According to Mysore et al. [5], with the emergence of 'mega data centers' with 100,000+ compute nodes housing a cloud environment with virtual hosts, the number of flows in the network will be in millions. Even with Layer 2 forwarding, MAC forwarding tables in such a scenario will need to hold hundreds of thousand to millions of entries - impractical with today's switch hardware. OpenFlow rules also use TCAMs (Ternary Content Addressable Memory), which are very costly and as a result most commercial switches do not support more than 5000 or 6000 TCAM based OpenFlow rules [4]. In an OpenFlow network, the number of entries in a switch is even higher than layer 2 since it is governed by the number of flows passing through it and not just the number of destination MACs being served by it. Also the size of each entry, 356 bits for OpenFlow v1.1, is larger than the corresponding 60-bit entry for layer-2 forwarding. All these facts suggest that today's switch hardware will not be able to sustain the demands of OpenFlow enabled mega data centers. Any optimization which compresses the number of Match Entries and Flow Counters in a switch will have a direct impact on switch state Size.

Controller Bottleneck : Enforcing the aforementioned flow-level granularity through a centralized control plane overburdens the controller and creates a processing bottleneck at the controller. In a dynamic network where various traffic engineering strategies may need to be deployed such as re-routing of flows, the controller becomes heavily overloaded with the task of updating flow entries on every switch belonging to the old and new paths of each re-routed flow. If the older route for a flow had five hops and the new route has four completely different hops, the controller has to send nine OpenFlow flow mod/add/del messages. This clearly limits the ability of the controller to dynamically alter routes at the scale of a production data center with millions of concurrent flows.

Involving the controller in the setup of every flow results in a large volume of control channel traffic. One *packet-in* message of at least 128-bytes must be sent to the controller for each new flow in the network. Additionally, one *flow-mod* message of 56-bytes must be sent by the controller to each switch lying along the new flow's path, as determined by the controller. For an average of 100,000 flows per second at a ToR [6], 100 ToRs in the network, and an average of 4 hops per flow, this results in over 10Gbps of control channel traffic from *packet-in* messages alone and over 4Gbps of control channel traffic from *flow-mod* messages alone.

High first packet latency - Packets that do not match a flow rule at a switch are redirected to the controller resulting in first packet latency of the order of several milliseconds (compared to microseconds for subsequent packets). The dominant component of this latency is the need to transition from the data plane to the control plane [4]. Pre-populating flow rules in switches with infinite timeouts (permanent rules) is the currently the only way possible to reduce latency. However, switch memory (mainly TCAM) becomes a constraint.

In this context, we present *SwitchReduce* - a system for reducing switch state and controller involvement in OpenFlow networks. SwitchReduce leverages the central visibility provided by OpenFlow to facilitate cooperation between switches which results in a division of labor and subsequently a reduction in switch state size as well as controller involvement. This co-operation between switches also enables pre-population of flow rules in the interior switches of the network which helps in reducing run-time control channel traffic, controller involvement and end-to-end latency as well.

SwitchReduce relies on two important observations. First, the number of match entries at any switch should be no more than the set of unique processing actions it has to take on incoming flows. Second, the flow counters for every unique flow may be maintained at only one switch in the network. SwitchReduce imposes no requirement on underlying switch hardware or software other than that the switches be OpenFlow v1.1+ enabled and implement the currently optional Push VLAN ID , Set VLAN ID and Pop VLAN ID actions.

We make the following contributions in this paper:

C1: We present a detailed design of SwitchReduce and describe its implementation as a NOX controller application.

C2: We present simulation results based on real data center traffic traces for two different topologies that demonstrate the potential savings that can be obtained through SwitchReduce.

C3: We present a case study that highlights the efficacy of SwitchReduce in reducing flow-setup latencies through proactive installation of OpenFlow rules, and easing the burden on the controller while dynamically rerouting flows.

The rest of this paper is organized as follows. Section II details the SwitchReduce system design. In section III, we describe our implementation of SwitchReduce as a NOX OpenFlow controller application. Section IV describes our simulation results. In section V, we present a case study profiling the applicability of SwitchReduce in real data center scenarios. Section VI presents an overview of related research. Finally, we conclude in section VII.

II. System Design

SwitchReduce is based on three founding principles:

1) Wildcard identical action flows
2) RouteHeaders : First-hop based routing to facilitate the aforementioned wildcarding
3) Division of labor : Collectively maintain traffic statistics across all switches

We explain each of these principles one by one and how we achieve them using current OpenFlow specifications.

A. Wildcard Identical Action Flows

The fundamental idea behind an OpenFlow rule is to be able to apply a customized action on every flow. Therefore, optimally, the number of flow rules should be bounded by the cardinality of the action space. This is the premise of our first founding principle.

At this point we would like to point out that, for ease of explanation, we limit our discussion in this section to forwarding rules only, although the principles described here can be extended to more diverse flow rules too as we discuss in Section V. For a forwarding flow rule, the Action field is basically a mapping of the flow to an outport.

We observe that the number of flows passing through a switch is several thousands. Also, the number of ports on

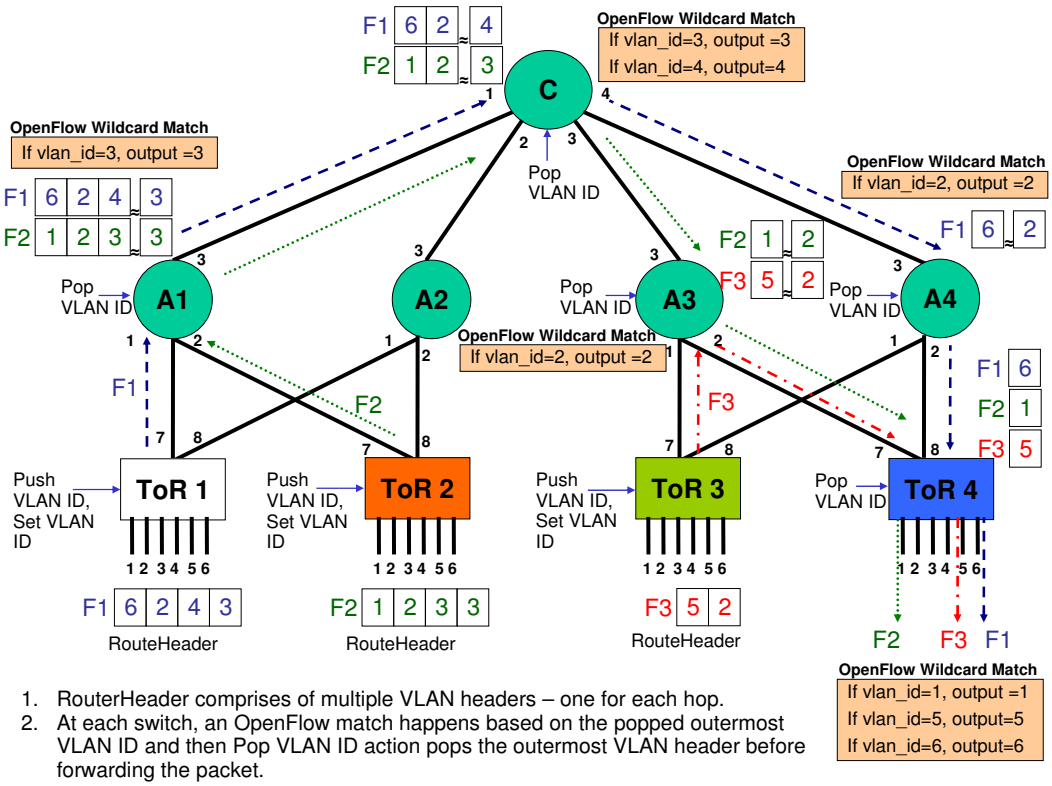

Fig. 2. RouteHeaders and use of Push VLAN ID and Pop VLAN ID operations in SwitchReduce

most commercial switches is less than 128. In fact most ToR switches typically have less than 64 ports, while aggregation and core switches have even fewer ports (albeit with much higher bandwidth). In some diverse topologies like FatTree [7], the switches used could have 52 ports for ToRs and 96 ports for Aggregation and Core [5]. Effectively, while there are several thousand flows through a switch, there are only a handful of ports that they can use. In other words, there are only a handful of actions that all flows must share. If we install one entry for every unique flow in a switch, we would end up with several thousand exact-match SRAM entries. In data centers that house several hundreds of thousands of servers [8] [9] [10], each possibly running several VMs, this would actually require millions of exact-match SRAM entries. This is not achievable with existing hardware. If on the other hand we install one entry for every unique action, we would end up with only a handful of wildcard TCAM entries.

Our first founding principle then is the following:

All flows in a switch that have identical actions associated with them, with the exception of flows at the first hop switch, can and should be compressed into one wildcard flow rule.

When paraphrased specifically for forwarding rules, the above statement becomes :

All flows in a switch that have the same outport, with the exception of flows at the first hop switch, can and should be compressed into one wildcard flow rule.

The first hop switch will be an OpenFlow enabled virtual switch (vswitch), like openvswitch [11]. This vswitch runs inside a hypervisor and connects all the VMs running inside the hypervisor to the network. In the absence of a virtual switch, the top-of-rack switch (ToR) will serve as the first hop.

At switches that are not first-hop switches, all flows with the same action get compressed into one wildcard flow entry. At the first hop, flows originating from directly connected VMs are not wildcarded but all other flows are. Note that a switch that serves as the first hop for some flows also serves as the last hop for some other flows. Flows for which this switch serves as the first hop switch are not wildcarded, but those for which it serves as the last hop switch are.

Flows at their first hop aren't wildcarded because of the following reason: We require one exact-match entry, a unique identity, for every flow at its first hop to both carry out the above mentioned wildcarding as well as to maintain flow level control and statistics. We discuss this in detail in sections II-B and II-C.

Optimally, the number of flow rules should be bounded by the cardinality of the Action space. Each wildcard flow rule should map all intended matching flows to the right action. Creation of these wildcards is a problem in itself that we address in section II-B. Maintenance of flow statistics in the absence of exact-match flow rules is another problem that we address in the section II-C.

B. RouteHeaders: First-hop based routing

To construct a wildcard rule, there must be some commonality (in terms of an OpenFlow header field) that is both exhaustive and exclusive to the flows being wildcarded. The wildcard can then be created on this common field. However, there is a high chance that such commonality may not exist between flows at a switch that share the same outport and therefore need to be wildcarded. In this section, we present an answer to the question: *How to create a wildcard when there is no inherent commonality between the flows being wildcarded?*

To achieve wildcarding, we exploit these underlying properties of OpenFlow:

- Centralized visibility makes the controller aware of the entire path from source to destination
- Centralized routing gives the controller complete freedom to choose any routing technique

The controller leverages these attributes to facilitate cooperation between switches such that each switch informs the next switch of the outport to use when forwarding the packet. The algorithm, detailed herein, divides a packet's path into 3 zones: The *First* Hop, The *Intermediate* Hops, The *Last* Hop. We explain this with the assistance of the example shown in Figure 2. In this example, we consider a three layer topology and assume that ToR switches form the lowermost layer. In a virtual environment, the ToR layer will be replaced by a vswitch layer, and Aggregation layer will be replaced by a ToR layer.

1) The First Hop: When a new flow originates at its first hop, the controller installs in it an exact match rule with Action:

- Forward packets belonging to this flow out on this designated port.(Port 7 for Flow F1 in our example on Figure 2)
- Use Push VLAN ID, Set VLAN ID to add a certain number of VLAN headers (collectively referred to hereafter as **RouteHeader**) onto packets belonging to this flow.

The number of added VLAN headers is equal to the number of remaining hops on the packet's path from the first hop to the destination. In Figure 2, the RouteHeader for Flow F1 is the 4-element array [6,2,4,3] corresponding to 4 newly added VLAN headers, one for each hop after ToR1. The RouteHeader is basically a concatenation of the appropriate number of 12-bit VLAN headers each uniquely identifying the action that has to be taken at every subsequent hop along the packet's path. Since we are specifically considering forwarding rules in this section, each VLAN ID in the RouteHeader uniquely identifies the outport for the packet at every subsequent hop along its path. For simplicity, we choose the value of the VLAN ID to be the same as the value of the outport on the corresponding hop. Thus, if the outport for a packet at a switch is X, the VLAN ID in the packet's RouteHeader for this hop will also have value X. In practice, this need not necessarily be the case. There just needs to be a one-to-one mapping between the space of VLAN IDs and the space of Actions. In other words, the VLAN ID would contain a unique Action ID. 12-bits can accommodate up to 4096 unique actions.

The RouteHeader, as the name suggests, carries the entire route that a flow will take based on a routing decision taken by the central controller. The outermost VLAN ID (right-most) contains the Action ID (simply the outport in our example) for the action taken by the second hop, the next VLAN ID contains the Action ID for the next hop and so on.

This is shown in Fig. 2. Since we consider only forwarding actions in this section, we use the outport itself as the Action ID. The outermost VLAN ID from the RouteHeader (shown as the rightmost 12-bit value '3' for Flow F1 in Figure 2) contains the outport for the second hop (Port 3 on Switch A1 - first hop after ToR1), the next VLAN ID '4' contains the outport for the third hop (Port 4 on Switch C) and so on.

2) The Intermediate Hops: When the controller installs an exact match rule for a flow in the first hop, it also installs a wildcard rule for it in the intermediate hops. In the match field of this wildcard rule, all fields except the VLAN ID are set to *Don't Care* while the VLAN ID is set to the Action ID (outport) for the flow. The corresponding action field simply contains the outport for the flow.

This wildcard rule can be either installed simultaneously while installing the first hop rule or, even better, proactively. Since there are only a handful of wildcard rules in SwitchReduce which are all known in advance, it is indeed possible and advisable to pre-populate the interior switches with these rules ("Interior" switches refer to switches in the core layers of the network, i.e. all switches except the edge switches). In a virtual environment (where the first hop is a vswitch), SwitchReduce makes it possible to proactively pre-populate all physical switches in the data center network with OpenFlow rules. SwitchReduce, therefore, presents a pragmatic way of pre-populating OpenFlow rules in switches (which is currently the only way to reduce flow-setup latencies in any OpenFlow network). OpenFlow literature [12] has recommended pre-populating rules as a means for reducing latencies but this has been hard to achieve in practice since rules are not known in advance.

According to the OpenFlow v1.1 specification [1], it is the outermost VLAN header of a packet that will be used by the switches to perform a VLAN-based match. Thus, when the packet arrives at the switch, it automatically matches the wildcard entry corresponding to the outermost VLAN ID of its RouteHeader.

The flow rule installed by the switch in the intermediate hop then is

- Set the outport for the packet to the specified value when the VLAN header contains the given value.(e.g. set outport to 3 when VLAN ID is 3 for flow F1 at Switch C in Figure 2.
- Use the Pop VLAN ID to pop the outermost VLAN ID.

The first action item above selects an outport based on the outermost VLAN header. The second action removes the outermost VLAN header from the RouteHeader before forwarding the packet to the outport. The new outermost VLAN header for the packet is therefore modified so that it corresponds to the Action ID for the next hop. Thus, in addition to forwarding the packet, each switch also prepares the packet for a match with the correct wildcard rule on the next switch. This cooperation between switches enables the wildcarding mechanism outlined in II-A. All packets that need to be forwarded out to a particular outport enter a switch with their outermost VLAN ID pre-set (by the previous hop) so that they all match the right wildcard entry.

3) The Last Hop: The last-hop also contains a wildcard rule for the given flow. The corresponding action is:

- Set the outport for the packet to the specified value when the VLAN header contains the given value.(e.g.

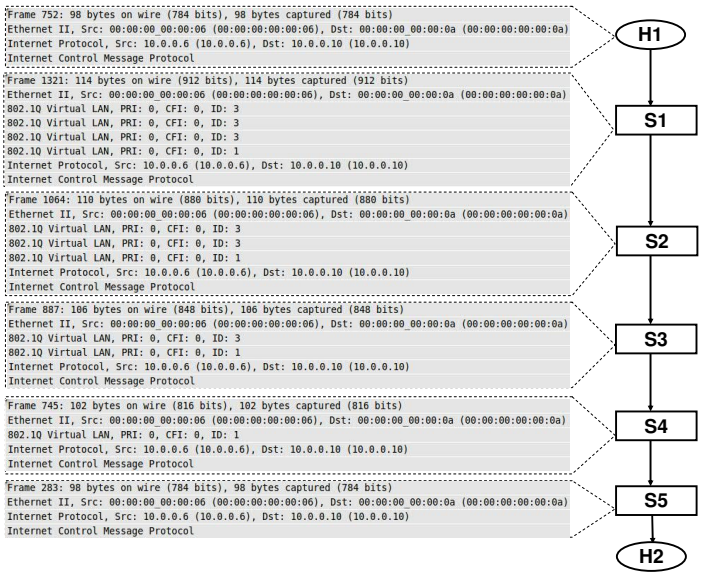

Fig. 3. Wireshark snapshot of a ping message from Host H1 to Host H2 - RouteHeader is [1,3,3,3]

set outport to 6 when VLAN ID is 6 for flow F1 at ToR4 in Figure 2.

- Use the Pop VLAN ID to pop the outermost VLAN ID.

In this particular context, where forwarding is the only action being performed by the switches, the last hop is identical to the intermediate hops. In fact, even these last hop rules can be pre-populated in a forwarding-only scenario. In Section V, we discuss a scenario where the last hop performs a role different than that of the intermediate hops and consider its implications.

So, in this fashion, RouteHeaders help realize the wildcarding idea that was introduced in Section II-A. Let us look at Figure 2 closely to understand how this wildcarding is being accomplished. Examine flows F2 and F3. F2 and F3 are between two separate source-destination pairs and have very diverse paths in the network. However, their paths cross at one link in the network, namely the link joining A3 to ToR4. Thus, A3 performs the same action on both F2 and F3. SwitchReduce seeks to install a wildcard rule in A3 such that both F2 and F3 match that wildcard. To do so, it creates a rule that says *If the VLAN ID is 2, output to port 2*. Now ToR2 installs in packets belonging to flow F2 the RouteHeader [1,2,3,3]. Switch A1 pops the outermost VLAN ID to send [1,2,3] to Switch C. Switch C then pops the outermost VLAN ID again to send [1,2] to Switch A3. Thus, when packets from flow F2 reach A3, the outermost VLAN ID is already set to '2' and hence a match with aforementioned wildcard rule happens. Similarly, ToR3 installs in packets belonging to flow F3 the RouteHeader [5,2]. Since A3 is the first hop (after ToR3) for flow F3, the outermost VLAN ID is already set to '2'. Thus, once again, a match with the aforementioned wildcard rule happens. In this manner, both F2 and F3 match the same wildcard at Switch A3.

C. Division of Labor

With the first two founding principles, we ensure that flow-level routing decisions can still be taken by the controller even though there are not as many flow-specific entries. To ensure that the sanctity of flow-level granularity, which is a salient property of OpenFlow, is not disturbed we must ensure that the controller is still able to gather flow level statistics.

This is automatically achieved by design. Recall that in our first founding principle, we chose not to wildcard flows at the first hop. This means that there is an exact-match rule, a unique identity, for every flow at its first hop - the switch where this flow first appears in the network. The controller can simply gather flow statistics from these first hop (vswitch or ToR) switches. The rest of the switches only need to collect port level statistics (which could be used to detect congestion in the network), and should not be involved in collection of flow level statistics. For any reliable delivery protocol like TCP, the end-to-end throughput is a constant therefore flow statistics would yield the same information regardless of which hop they are polled from.

As an example, suppose the link from core switch C (Port 3) to Aggregation switch A3 in figure 2 is congested. The controller simply needs to look at port statistics for port 3, which it still can, to detect congestion. Once it does that, it can look at all flows in the network that are being sent to that port. The controller has access to this information since it installed RouteHeaders for each flow. It can simply maintain a list of all flows that use any given port on any given switch. Then, it only needs to look at respective first-hops to figure out how much traffic is being contributed by each flow. This way it can control misbehaving hosts.

III. Implementation

We have implemented SwitchReduce as a NOX [13] controller application. We use an OpenFlow 1.1 User Switch Implementation [14] inside a Mininet [15] testbed. Mininet is a platform that enables creation of OpenFlow based software defined networks on a single PC using Linux processes in network namespaces.

SwitchReduce NOX application is a complete end-to-end implementation of SwitchReduce. Given any network, the application first learns its entire topology which includes the location of all switches in the network and their interconnections, as well as a mapping of hosts to ToR switches. The host to ToR mapping is obtained by tracking the first OFPT_PACKET_IN event at the controller from every ToR switch, which in the case of Mininet happens to be an ARP packet. This event also triggers a pro-active route computation algorithm within our controller which pre-computes the shortest available route between every possible pair of hosts. This is done to minimize controller processing time once actual traffic begins.

The controller also pre-populates all interior switches in the network as well as the last hops with all possible wildcard rules upon completion of the pro-active route computation algorithm. Thus, before traffic in the network begins, all switches except the first hop switches have wildcard rules installed in them. These wildcard rules have all bits except VLAN ID set to 'x'. The VLAN ID is set to one of the outport numbers. Thus, the number of wildcard rules is equal to the number of outports on each switch. The corresponding actions installed by the controller for these rules are OFPAT_POP_VLAN which pops the outermost header post a match, and OFPAT_OUTPUT which forwards the flow to its designated outports. At the

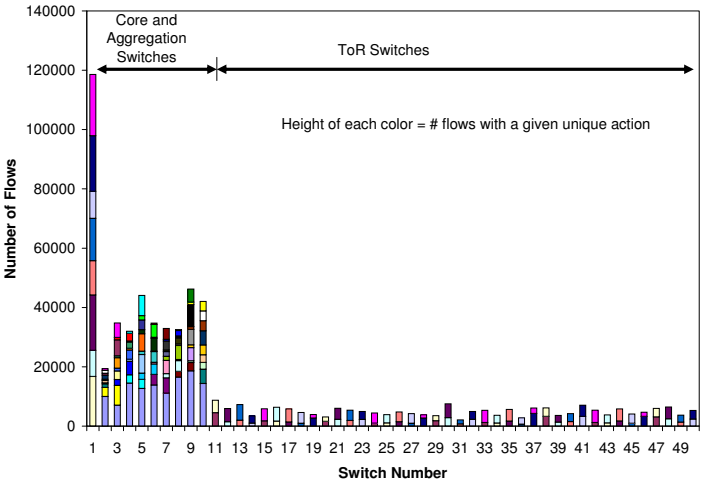

Fig. 4. Action overlap on Host ToRs and Aggregation/Core switches

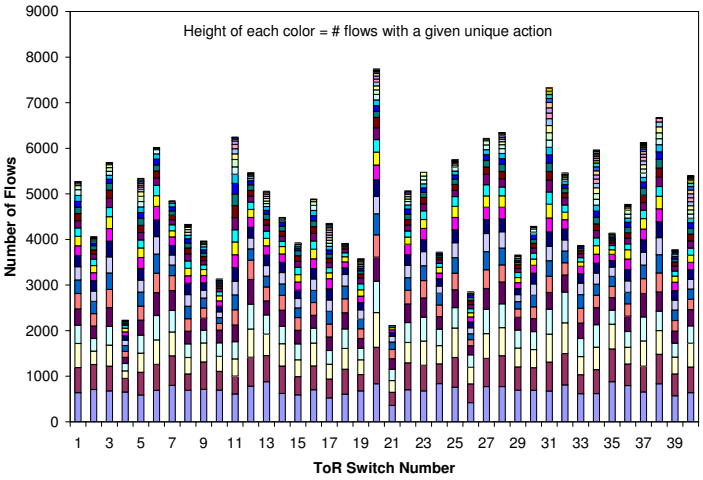

Fig. 5. Action overlap on destToRs

destination ToR, the last added VLAN header is removed ar the flow is delivered to its destination.

When a new flow arrives at the first hop switch, it sen an OFPT_PACKET_IN to the controller. The controller simp looks-up the pre-computed route for this flow and instal an exact match rule in the ToR with corresponding actio OFPAT_PUSH_VLAN and OFPAT_SET_VLAN_VID to pu and set the requisite number of VLAN headers onto the flo It also installs an OFPAT_OUTPUT action which forwards t packet out from the designated port.

We used ping to verify the functionality of our algorith... and used 'Wireshark' to trace the RouteHeader as a packet made its way through the network (refer Figure 3 for Wireshark snapshots of a ping packet as it is routed from Host H1 to H2 via Switches S1, S2, S3, S4 and S5 - RouteHeader is [1,3,3,3]).

IV. Simulation Results

Currently, none of the hardware switches support OpenFlow v1.1+. Hence, a full blown evaluation of SwitchReduce in a hardware testbed can not be done right now. Instead, we evaluate SwitchReduce using simulations to quantify the savings in switch state.

We developed a simulator that simulates a data center network. The simulator creates a network topology using as input the number of servers, servers per rack, switches and their connections, and type of topology. The simulator then maps a user specified number of VMs randomly onto the servers. It then takes a traffic matrix and stores all flows in the network as (Source VM, Destination VM) pairs. To route each flow, it uses Djikstra's shortest path algorithm. Finally, it polls the switches to examine the number of flows through each switch and groups them based on outport. This analysis reveals the overlap in flow rule actions and quantifies the compression that SwitchReduce accomplishes. To clearly demonstrate the switch state problem and a microscopic view of the gains achieved by SwitchReduce, we first present results for a modest tree topology comprising of 400 servers distributed uniformly over 40 racks. We use a real, dense traffic matrix for these results.

Subsequently, we present results for a large topology comprising of 11520 servers distributed uniformly over 288 racks.

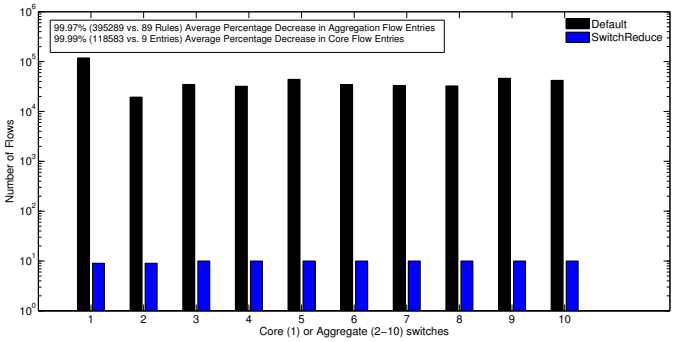

Fig. 6. Compression achieved by SwitchReduce on Aggregation/Core switches

This distribution of servers is adopted from [5] which in turn interprets this from [10] [8] [9]. We evaluate the performance of SwitchReduce here for both tree and fat-tree topologies. Also, here we use a traffic matrix derived from real data center traffic characteristics collected by Benson et al. [6].

In each of our simulations, we create a 3-layer network topology comprising of a ToR layer, an Aggregation layer and a Core layer. The ToR layer then comprises the first hop. We do not consider a virtual environment for these simulations so as to be able to demonstrate the effectiveness of SwitchReduce even in the absence of vswitches. Therefore, the results presented for ToR switches here are an underestimate. In a real data center network, the gains on ToR switches will also be as high as the gains on the Aggregation and Core switches of our simulation.

A. 400 server topology:

In this topology, there are 40 ToR switches, 9 Aggregation switches and 1 Core switch. Every ToR connects to two Aggregation switches, and Core switch connects to all Aggregation switches. We pack different number of VMs onto the servers in our experiment runs. We used real and highly dense traffic matrices from our earlier work [16] to drive the simulator.

The results in Figure 4-Figure 7 are for an input of 1000 VMs using a Real 1000x1000 dense traffic matrix. In Figure 4 - Figure 7, the X-axis represents switch number. The Core switch is number 1, Aggregation switches are 2 to 10 (in Figures 4 and 6), while ToR switches are 11 to 50 (in Figure

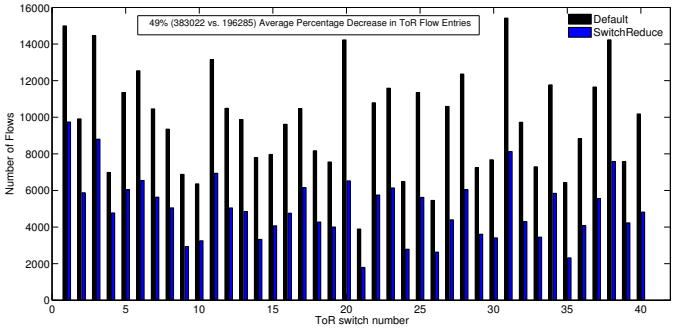

Fig. 7. Compression achieved by SwitchReduce on ToR switches

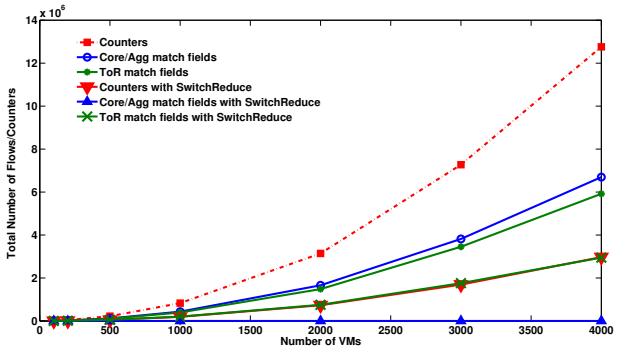

Fig. 8. Rate of increase of flow counters and match entries with VMs

4), and 1 to 40 (in Figures 5 and 7). For the traffic matrix used, total number of flows in the network were 193,478.

Figure 4 and Figure 5 capture overlap in flow-rule actions. Figure 4 captures the First Hop and Intermediate Hops. Figure 5 captures the last hop. Both Figure 4 and Figure 5 are stacked bar graphs demonstrating the number of flows passing through a switch and the number of unique actions taken on those flows. The number of different colors per bar are the number of unique actions, while the height of each color is the number of flows on which that action is taken.

While Figure 4 contains ToR as well as Agg/Core switches, Figure 5. contains only ToR switches. Note that the same ToR switch would serve as first hop (hostToR) for some flows and as last hop (destToR) for other flows. The total number of flows through ToR switches is the sum of heights of the corresponding bars in Figure 4(ToR switch numbers start from 11) and Figure 5.

Flows corresponding to ToR switches, numbers 11-50, on Figure 4 are not wildcarded. This is because these flows have their first hop at these corresponding ToRs so their unique identity will be preserved at this point in the network. All remaining flows on Figure 4 as well as flows on Figure 5 will be wildcarded. Figure 6 and Figure 7 show compression achieved on Agg/Core and ToR switches respectively by SwitchReduce. The Average compression on Agg/Core switches is more than 99%, while on ToR switches it is 49%.

Figure 8 shows the rate of increase of flow counters and match field entries maintained throughout the network (with and without SwitchReduce) as a function of number of VMs. Both Flow Counters and ToR match fields increase geometrically with increasing number of VMs. However, in the absence of SwitchReduce, the rate of increase is higher thereby causing the lines to diverge. Agg/Core match entries increase geometrically too in the absence of SwitchReduce but stay constant when SwitchReduce is used.

B. 11520 server topology

We first run SwitchReduce on a tree topology. This topology has 288 ToR switches, 24 Aggregation Switches and 1 Core Switch. Every ToR connects to two Aggregation switches, and Core switch connects to all Aggregation switches. We pack 10 VMs on each server.

Next, we run SwitchReduce on a fat-tree topology. This is identical to the topology interpreted by [5] from [10] [8] [9]. The racks are arranged in 12 rows with each row comprising of 24 racks. There are 288 ToR switches, 12 Aggregation Switches and 12 Core Switches. Each ToR connects to a single Aggregation switch and each Aggregation switch connects to all Core switches. We pack 10 VMs on each server.

In each case, we derive our traffic matrices from real data center traffic characteristics published by Benson et al. [6]. The data they collect from production data centers running map-reduce style tasks and from university data centers reveals that the median number of same-rack hosts that any host talks to is 2, while the median number of out-of-rack hosts that any host talks to is 4. Thus the traffic matrix that we use is created by randomly choosing 2 same-rack communicating hosts and 4 out-of-rack communicating hosts for every host. The communication is assumed to be bi-directional.

The results from these experiments are shown in Figure 9 which depicts the percentage reduction obtained by SwitchReduce in both the tree and fat-tree topologies. The results for both topologies are mostly overlapping with some small differences. The reduction percentage for the interior switches in both topologies is in the high 90s. The gains on Core switches in the fat-tree topology are slightly lesser than the tree topology. This is expected because in fat-tree the flows are balanced across 12 core switches, while they all go through just 1 core switch in the tree topology. The reduction percentage for ToRs in each topology is in the early 40s. This is about 8% smaller than the reduction obtained in our 400 server topology. The reason for this difference is that while there were only 10 servers-per-rack in the 400 server topology, there are 40 servers-per-rack in the 11520 server topology. Thus, each Switch has 40 downward facing ports in use compared to only 10 in the previous case. This results in a relatively bigger action space.

V. Case Study - Traffic Engineering

In this case study, we consider some typical flow characteristics within a data center. According to Kandula et al. [17] and Benson et al. [6], most data center flows (80%) last less than 10s and in general 100 new flows arrive every millisecond. They found that there are few long running flows (less than 0.1% last longer than 200s) contributing less than 20% of the total bytes while more than half the bytes are in flows that last no longer than 25s. Benson et. al. in [18] confirm that the implications of the aforementioned statistics for traffic engineering are that data center networks

1) Must scale to handle a large number of flows
2) Must accommodate both short and long lived flows without making assumptions on the size

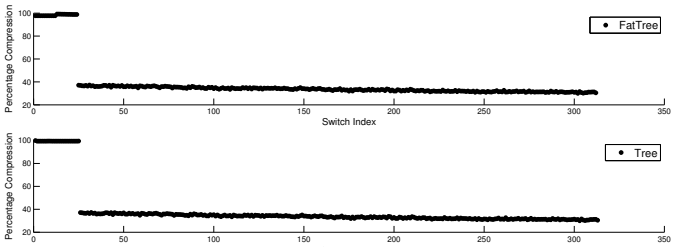

Fig. 9. Compression achieved by SwitchReduce for fat tree (Portland) and tree topologies

We have already discussed at length the explosion of switch state with number of flows in the network. This problem is further exacerbated by the fact that, as cited above, majority of the flows are short lived and arrive in bursts with very short inter-arrival times, followed by periods of dormancy. This means that at any time, a switch has flow entries belonging to flows that have recently expired (since they have not timed out yet) while it is also trying to accommodate new flows. This requires controller intervention to delete the old flow entries which would lead to excessive controller involvement and create processing bottlenecks at the controller threads as well as bandwidth bottlenecks in the control channels connecting the controller to switches. Alternately, it requires switches to just wait (if they happen to run out of storage space) for existing flow entries to time-out before they can add new flow entries, or just use slower look-up memory (like software tables) to store new flows.

SwitchReduce by design, as discussed in Section II, automatically addresses this problem. The number of flow rules in any switch is dictated by the number of unique actions taken on flows passing through it and does not grow linearly with number of flows. From the perspective of traffic engineering, where one of the primary responsibilities of the controller is to find optimal routes for new flows and hence the primary action taken by switches on flows is forwarding, a SwitchReduce enabled controller does not need to concern itself with switch state while making a routing decision. This is because with SwitchReduce, pushing a new flow through a switch will not necessarily require adding a new flow rule. Even with reactive flow installation instead of pro-active flow installation, once the network has reached a 'reasonably busy' state where some traffic is flowing in all directions and therefore traffic is being output to almost all ports of any switch, the entire set of possible wildcard rules would have already been installed in the switches. Thus routing a new flow through the network is unlikely to require a new flow installation at any switch other than its ingress switch.

In fact, like we mentioned in II-B, SwitchReduce is a very practical way of pro-actively installing rules in the interior switches of a network (or all physical switches of a network in case the first hop switch or ingress switch is a vswitch). This can be coupled with the topology discovery module of the controller (like we do in our implementation in Section III). When actual traffic arrives in the network, the controller only needs to figure out the route for each flow and install a rule at the ingress switch with the appropriate RouteHeader. In essence, SwitchReduce gives the controller complete freedom to choose the best possible path without worrying about increasing the volume of flows being served by switches lying along that path. Also, equally importantly, it limits the involvement of the controller for flow installation to only the

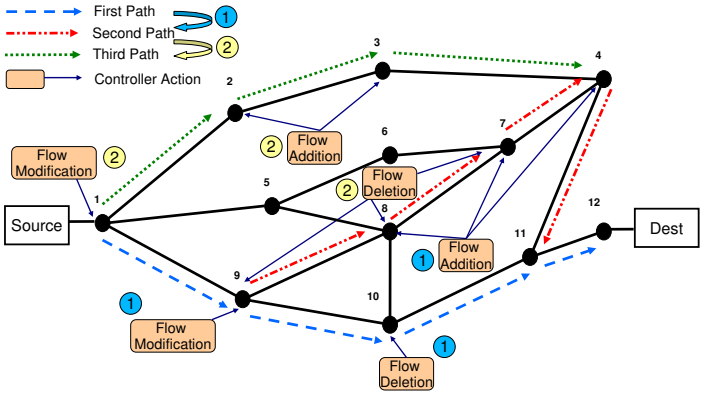

Fig. 10. Controller involvement in re-routing flows

ingress switch. This means that there is no flow-installation related control channel traffic in the interior switches which prevents bottlenecks in both control channel bandwidth as well as controller processing threads.

For long lived flows too, which contribute about 20% of the traffic, SwitchReduce does not hamper performance in any way. In fact, it gives the controller the freedom to choose multi-path routing to speed up this flow without having to worry about adding to switch state on each of the paths.

Lastly and most importantly, from a traffic engineering standpoint, the controller might wish to re-route some flows to constantly balance network load. It might need to examine network state periodically and dynamically change the paths that flows are taking [2]. This, however, is impractical in current architectures as it requires too much controller involvement. Not only does the controller need to figure out the new route, it also needs to delete flows at multiple switches along the original path and install flows at multiple switches along the new path. With SwitchReduce, though, the controller doesn't even need to touch any switch other than the ingress switch for dynamic re-routing of flows. It merely needs to re-write the RouteHeader in accordance with the new route and the intermediate switches will automatically forward this flow to the new path. Thus the controller needs to merely modify the flow rule at exactly one switch, and does not need to install or delete any flows on any of the interior switches. This is explained with the help of a figure 10

Consider a flow that is going from edge switch 1 to edge switch 12. Initially, the controller chooses path *1 - 9 - 10 - 11 - 12* for this flow. However, for some reason, it later decides to re-route the flow through *1 - 9 - 8 - 7 - 4 - 11 - 12* . In the absence of SwitchReduce, the controller would first have to delete the flow rules it has installed for this flow in switch 10 and modify the rule installed in switch 9. Further, it would have to install new rules in switches 8,7 and 4. Thus there are 5 controller interventions that need to be performed. Again, if the controller decides to re-route this flow through *1 - 2 - 3 - 4 - 11 - 12* , it would have to delete the rules in switches 9, 8, and 7, modify the rule in switch 1, and install new rules in switches 2 and 3. Thus 6 controller interventions need to be performed. If the controller was using SwitchReduce though, the controller needs to perform just a single intervention for each modification of flow path. It has to modify the flow rule at the ingress switch only.

VI. Related Work

Several concerns around the idea of centralized flow-level control [4] [19] in OpenFlow have been documented. Mogul et al. [20] state that since OpenFlow rules are per-flow instead of per-destination, each directly-connected host requires an order of magnitude more rules than in traditional Ethernet lookup. Greenberg et al. [17] report a 10:1 ratio of flows to hosts.

From a routing perspective alone, there are two works that are conceptually similar to SwitchReduce. These are Multi Protocol Label Switching (MPLS) [21] and SourceFlow [22].

MPLS provides a means to map IP addresses to simple, fixed-length labels, and creates label-switched paths that make high speed switching possible. A special label distribution protocol(LDP) needs to be employed to exchange labels between the MPLS-enabled switches to convey labels that need to be used. Packets are forwarded based on the label stacks pushed onto them by the MPLS edge router. SwitchReduce on the other hand leverages the centralized visibility and control offered by OpenFlow to embed RouteHeaders onto packets and avoids the need for any special distribution algorithms. This makes it feasible to dynamically change the meaning of RouteHeaders without disrupting traffic.

Chiba et al. [22] note in their work that OpenFlow action space is much smaller than the flow space and therefore it is advisable to not advisable to install per-flow rules in switches. SwitchReduce also relies on this observation. However their method reduces number of flow entries only on core switches and not edge switches. Also, they don't address the issue of repetition of flow counters. Additionally, while they claim their technique works with existing switches, switches need to perform the added function of incrementing an index counter and accessing the appropriate action from the action list. Lastly the action list, which is carried as a header, can grow indefinitely large resulting in potentially large overheads.

We provide an off-the-shelf OpenFlow solution and an algorithm for reducing both match rule and flow counters across the entire data center network, without requiring any modification to commodity hardware. We also provide a feasible way of pre-populating rules in core switches of the network with the aim of reducing controller involvement, control channel traffic and end-to-end latency. We also provide extensive experimental results on a simulator to measure switch state reduction.

VII. Conclusion

Flow-level granularity in OpenFlow comes at the cost of placing significant stress on switch state size and controller involvement. In this paper, we presented SwitchReduce - a system for reducing switch state and controller involvement in OpenFlow networks. SwitchReduce leverages central visibility of OpenFlow to facilitate cooperation between switches. It mandates that the number of flow rules be bounded by the cardinality of the Action space. Furthermore, flow counters for every flow may be maintained at only one switch in the network. Our implementation of SwitchReduce as a NOX controller application with software OpenFlow switches validates its realizability. Our evaluation on a simulator with real traffic traces demonstrated the potential reduction in switch state size that can be obtained through SwitchReduce. We are currently evaluating SwitchReduce on larger data center topologies with real hardware. We are also working on handling failures and topology changes.

References

[1] N. McKeon, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," in *ACM SIGCOMM Computer Communication Review*, April 2008.

[2] M. A. Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *USENIX NSDI*, 2010.

[3] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS Flooding Attack Detection Using NOX/OpenFlow," in *35th Annual IEEE Conference on Local Computer Networks*, 2010.

[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-Performance Networks," in *ACM SIGCOMM*, 2011.

[5] R. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric," in *ACM SIGCOMM*, August 2009.

[6] T.Benson, A. Akella, and D.Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *IMC*, 2010.

[7] C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," in *IEEE Transactions on Computers*, 1985.

[8] "Cisco Data Center Infrastructure 2.5 Design Guide." [Online]. Available: http://www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf

[9] "Configuring IP Unicast Layer 3 Switching on Supervisor Engine 2." [Online]. Available: http://www.cisco.com/en/US/docs/routers/7600/ios/12.1E/configuration/guide/cef.html

[10] "Inside Microsoft's $550 Million Mega Data Centers." [Online]. Available: http://www.informationweek.com/news/hardware/data_centers/showArticle.jhtml?articleID=208403723

[11] "Open vSwitch - An Open Virtual Switch." [Online]. Available: http://www.openvswitch.org

[12] R. Sherwood, "An experimenters guide to openflow," *GENI Engineering Workshop - http://www.rob-sherwood.net/GENI-Experimenters-Workshop.ppt*, June 2010.

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," in *ACM SIGCOMM Computer Communication Review*, July 2008.

[14] "OpenFlow 1.1 Software Switch." [Online]. Available: https://github.com/TrafficLab/of11softswitch

[15] "Mininet: rapid prototyping for software defined networks." [Online]. Available: http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet

[16] V. Mann, A. Kumar, P. Dutta, and S. Kalyanaraman, "VMFlow: Leveraging VM Mobility to Reduce Network Power Costs in Data Centers," in *IFIP Networking*, 2011.

[17] S.Kandula, S.Sengupta, A.Greenberg, and P.Patel, "The Nature of DataCenter Traffic: Measurement and Analysis," in *IMC*, 2009.

[18] T. Benson, A. Anand, A. Akkela, and M. Zhang, "MicroTE: Fine Grained Traffic Engineering for Data Centers," in *ACM CoNEXT*, December 2011.

[19] M.Yu, J.Rexford, M. Freedman, and J. Wang, "Flow-Based Networking with DIFANE," in *ACM SIGCOMM*, 2010.

[20] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. Curtis, and S. Banerjee, "DevoFlow: Cost-Effective Flow Management for High Performance Enterprise Networks," in *ACM Hotnets*, 2010.

[21] "Multiprotocol Label Switching (MPLS)." [Online]. Available: http://www.cisco.com/en/US/products/ps6557/products_ios_technology_home.html

[22] Y. Chiba, Y. Shinohara, and H. Shimonishi, "Source Flow: Handling Millions of Flows on Flow-based Nodes," in *ACM SIGCOMM*, 2010.