Dynamic Flow Scheduling for Power-Efficient Data Center Networks

Zehua Guo*, Shufeng Hui†, Yang Xu†, H. Jonathan Chao†
*ChinaCache, †New York University
Email: *guolizihao@hotmail.com, †{shufeng.hui, yang, chao}@nyu.edu

***Abstract*—Power-efficient Data Center Networks (DCNs) have been proposed to save power of DCNs using OpenFlow. In these DCNs, the OpenFlow controller adaptively turns on and off links and OpenFlow switches to form a minimum-power subnet that satisfies traffic demand. As the subnet changes, flows are scheduled dynamically to routes composed of active switches and links. However, existing flow scheduling schemes could cause undesired results: (1) power inefficiency: due to unbalanced traffic allocation on active routes, extra switches and links may be activated to cater to bursty traffic surges on congested routes, and (2) Quality of Service (QoS) fluctuation: because of the limited flow entry processing ability, switches cannot timely install/delete/update flow entries to properly schedule flows.**

In this paper, we propose AggreFlow, a dynamic flow scheduling scheme that achieves power efficiency in DCNs and improved QoS using two techniques: *Flow-set Routing* and *Lazy Rerouting*. Flow-set Routing achieves load balancing and reduces the number of entry installment on switches by routing flows in a coarse-grained flow-set fashion. Lazy Rerouting maintains load balancing and spreads rerouting operations over a relatively long period of time, reducing the burstiness of entry installment/deletion/update on switches. We built a NS3 based fat-tree network simulation platform to evaluate AggreFlow's performance. The simulation results show AggreFlow reduces power consumption by about 18%, achieves load balancing and improved QoS (i.e., low packet loss rate and reducing the number of processing entries for flow scheduling by 98%), compared with baseline schemes.

***Index Terms*—Power-efficient data center networks; power saving; flow scheduling; OpenFlow**

I. Introduction

The popularity of cloud services accelerates the expanding of data centers. The high power consumption of data centers has become one of the most important concerns of their operators. Some recent studies present power-efficient DCNs, which enable network components (e.g., switches and links) to consume power proportionally to varying traffic demand [1][2]. With ElasticTree [1], a key enabler of power-efficient DCNs, traffic flows are consolidated on a subnet of the DCN called *minimum-power subnet*, which is composed of the minimum number of switches and links to sustain current network traffic demand. Thus, unused network components are turned off or put into sleep mode to save power [3]. When traffic demand exceeds the current subnet's capacity, more switches and links will be powered on to form a new subnet with sufficient capacity.

This work was conducted and completed while Dr. Zehua Guo was a visiting research scholar in New York University from 2013 to 2014.

These power-efficient DCNs usually employ Software-Defined Networking (SDN) (e.g., OpenFlow [4]) to consolidate and schedule flows. We argue that, to practically deploy power-efficient DCNs, an efficient flow scheduling scheme should achieve high power efficiency and improved Quality of Service (QoS). However, existing flow scheduling schemes cannot achieve the two aspects at the same time. ElasticTree proposes *balance-oblivious flow-level scheduling* schemes that consolidate flows in the DCN without load balancing consideration [1]. Thus, when bursty traffic surges on congested routes, extra switches and links may be activated to cater to the traffic increase.

To achieve load balancing on active routes, the SDN controller must take into account the load of each flow and conduct fine-grained flow-level scheduling. In particular, when some switches and links are about to be turned off to save power, the controller has to reroute many existing flows to maintain reachability and load balancing [1]. Since rerouting an existing flow requires the controller to generate multiple control messages to set up flow tables in the switches along this flow's old and new routes, a *control message storm* occurs if a large number of flows are rerouted simultaneously. The control message storm could impose a high processing burden on switches to install/delete/update flow entries used for flow scheduling. However, current OpenFlow switches suffer from traditional hardware design and have a limited processing ability (e.g., at most processing 200 entries per second) [5][6]. Since the minimum-power subnet must change with time-varying traffic demand [7][8][9], switches cannot timely update their flow tables to properly schedule a large number of flows, resulting in QoS fluctuation. The above problems will be detailed in Sections II-B and II-C.

In this paper, we propose a dynamic flow scheduling scheme named AggreFlow to achieve high power efficiency and load balancing in DCNs with improved QoS. AggreFlow mainly employs the two techniques listed below:

1) Flow-set routing. It aggregates flows into a small number of *flow-sets* based on flows' hash values, and achieves load balancing by conducting routing in a coarse-grained flow-set fashion, which reduces the number of control messages for routing flows.
2) Lazy rerouting. Each time the minimum-power subnet changes, a flow-set is not rerouted until a packet belonging to the flow-set enters the network. Lazy rerouting amortizes the rerouting operations on flow-sets over a

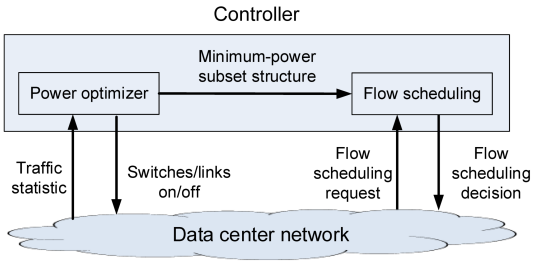


Fig. 1. Logical structure of a power-efficient DCN.

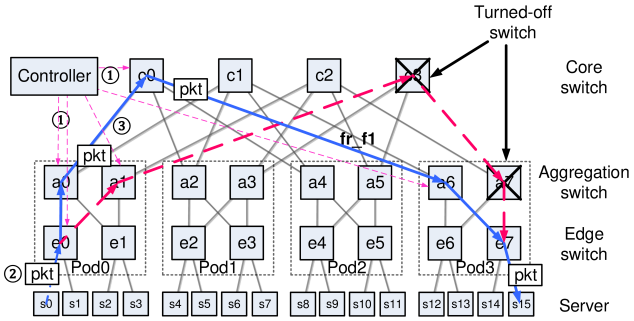


Fig. 2. Flow rerouting using the flow-level scheduling scheme in a 3-layer 4-pod fat-tree network. $f1$ is a flow with forwarding route fr_{f1}.

relatively long time, relieving switches from the control message storms. In addition, lazy rerouting reroutes a few flow-sets to maintain load balancing, and lets the majority of flows still be forwarded on their original routes. Hence, the amount of control messages for rerouting operations is significantly reduced.

We built a NS3 based fat-tree network simulation platform to evaluate AggreFlow's performance. The simulation results show AggreFlow reduces power consumption by about 18%, achieves load balancing and improved QoS (i.e., low packet loss rate and reducing control messages by 98%), compared with baseline schemes.

II. Background and Motivation

A. Power-efficient DCNs

Figure 1 shows the logical structure of a power-efficient DCN, which is composed of a DCN (including servers, switches and links) and a power consumption adapting system. The power consumption adapting system enables network components to consume power proportionally to traffic demand in the DCN and uses two components: power optimizer and flow scheduling [1][2]. Both components reside in an OpenFlow controller with global network information. The power optimizer component calculates the number of active network components based on current network traffic demand, configures power status of switches and links in the DCN, and notifies the current subnet structure to the flow-scheduling component [1][2]. Upon receiving the subnet structure, the flow scheduling component consolidates flows by adaptively routing and rerouting flows in the given subnet.

B. Imbalanced Loads on Active Routes

ElasticTree [1] proposes a simple balance-oblivious flow-level scheduling to consolidate flows. In a fat-tree network, the route of each flow is chosen in a deterministic left-to-right order. Only when the capacity of the leftmost route is insufficient for a flow, the second left route then will be evaluated for the flow, and so forth. Thus, the left routes could have more traffic loads than other routes, suffering from a higher chance of congestion. Under such an unbalanced traffic allocation, some links could be congested and request the controller to turn on more switches and links to accommodate bursty traffic surges while other links are under low utilization. Since the DCN traffic variation exhibits bursty [7][8][9], it could lead to unbalanced traffic allocation and impact power efficiency.

C. QoS Fluctuation

In a subnet, we can achieve load balancing by rerouting flows to the least loaded route [10]: the OpenFlow controller uses its global network view to conduct flow-level scheduling. We name this scheme balance-aware flow-level scheduling. However, flow-level scheduling schemes could impose a high burden on switches. First, flow-level scheduling schemes require multiple control messages to reroute an existing flow by setting up flow tables of the switches along this flow's old and new routes. Figure 2 shows an example to reroute an existing flow with flow-level scheduling schemes. Flow $f1$ is originally forwarded on route $e_0 \rightarrow a_1 \rightarrow c_3 \rightarrow a_7 \rightarrow e_7$ (red dash line). At time t_1, switches c_3 and a_7 are turned off to save power, and the controller immediately updates $f1$'s route to route fr_{f1}: $e_0 \rightarrow a_0 \rightarrow c_0 \rightarrow a_6 \rightarrow e_7$ (blue line). The rerouting operation consumes five control messages: one message to delete flow $f1$'s entry on switch a_1, one message to update flow $f1$'s entry on switch e_0, and three messages to install flow $f1$'s entries on switches a_0, c_0 and a_6[1]. At time t_2, the subsequent packets of flow $f1$ enter the DCN and are forwarded on route fr_{f1}. In the worst case (i.e., switch a_7 is not turned off in the above example), six control messages (i.e., control message to switches e_0, a_0, a_1, c_0, a_6 and a_7) are needed to reroute an existing flow.

Second, every time the minimum-power subnet changes, flow-level scheduling schemes would reroute many flows (i.e., all flows on the soon-to-be-closed routes and many flows on routes that will be still open) to maintain reachability and load balancing. Reports show that a commercial data center can consist of millions of flows [11][12]. To improve OoS, the rerouting operations require switches to install/delete/update entries in a very short period. We call this phenomenon the *control message storm*. The traffic variation in DCNs exhibits highly bursty [7][8][9][13], and the subnet reconfiguration may happen frequently to save power or accommodate traffic demand variation, leading to frequent control message storms. However, current OpenFlow switches suffer from hardware

[1]The controller does not send control messages to delete flow $f1$'s entries at switches c_3 and a_7 because they are closed and their flow tables are emptied.

design (e.g., flow entries must be organized in the TCAM in a priority order for correct and efficient matching; control messages must contend for limited bus bandwidth between a switch's CPU and ASIC [5]), and they have limited capacities to process entry update (e.g., Pica8 Pronto 3780 can only update at most 200 entries per second [6]). Therefore, the switches would not be able to timely update entries for all rerouting flows and thus degrade QoS.

D. Design Principles for Efficient Flow Scheduling Schemes

Based on the above analysis, we have the below considerations to design an efficient flow scheduling scheme for DCNs:

1) High power efficiency: as traffic varies, flows should be dynamically consolidated and rerouted to as few links as possible so that unused switches and links can be turned off or put into to sleep mode for power saving.
2) Good load balancing: in the minimum-power subnet, good load balancing among active routes can prevent activating extra switches and links to accommodate bursty traffic surges and save more power. Thus, we should take into account the traffic load of active routes to schedule flows.
3) Preventing control message storms: the main reason of the control message storm is that the flow-level scheduling scheme reroutes a large number of flows at the same time when the minimum-power subnet changes. To prevent control storms, we should (1) reduce the number of rerouted flows, (2) avoid conducting rerouting operations simultaneously, and (3) decrease the number of control messages for route configuration.

III. AggreFlow Overview

A. Term Definition

We first highlight some important terms used for AggreFlow and exemplify them in Figure 2.

Minimum-power subnet: a subnet of the DCN that is composed of the minimum number of switches and links to sustain the current network traffic demand.

Flow-set: a set of flows that are aggregated together based on their hash values.

Ingress edge switch is: an edge switch that connects to a flow's source server.

Egress edge switch es: an edge switch that connects to a flow's destination server.

Forwarding route fr: a route from a flow's ingress edge switch to its egress edge switch.

Turning-point switch ts: a switch that is at the turning point of a flow's (or flow-set's) forwarding route. In the fat-tree topology, a turning-point switch is either a core switch for inter-pod flows (which traverse different pods) or an aggregation switch for intra-pod flows (which only traverse aggregation switches in the same pod).

Upstream route ur: an upstream route that is directed from a flow-set's ingress edge switch to its turning-point switch.

Downstream route dr: a downstream route that is directed from a flow's turning-point switch to its egress edge switch.

In Figure 2, flow $f1$'s turning-point, source and destination switches are c_0, e_0, and e_7, respectively. Its uplink route is $e_0 \rightarrow a_0 \rightarrow c_0$; its downlink route is $c_0 \rightarrow a_6 \rightarrow e_7$; its forwarding route is $e_0 \rightarrow a_0 \rightarrow c_0 \rightarrow a_6 \rightarrow e_7$; the minimum-power subnet consists of all switches and links except the powered-off switches c_3 and a_7 and links related to the two switches.

B. AggreFlow Techniques

AggreFlow employs the three techniques to efficiently schedule flows: Flow-set Routing, Lazy Rerouting and Adaptive Rerouting. Flow-set Routing conducts a coarse-grained control on flows to reduce the number of control messages for route configuration. In DCNs, some network functions, such as traffic engineering, do not need each flow's specific information. For those functions, we can aggregate flows with the same hash value into a flow-set, and conduct routing in a coarse-grained flow-set fashion. Once we select a route for a flow-set, the following new flows that belong to the flow-set can be forwarded on the flow-set's route without querying the controller.

Lazy Rerouting avoids conducting rerouting operations simultaneously. The DCN traffic analysis shows a flow's packet arrivals exhibit an ON/OFF pattern [8][9][12][14]. For instance, in DCNs, the inter arrival time of a flow's two adjacent packets is longer than 100 ms [12][14]. Thus, every time the subnet changes, Lazy Rerouting updates the route of a ready-to-be-rerouted flow-set[2] only when a packet belonging to the flow-set enters the network. Such a rerouting spreads rerouting operations over a relatively long period of time, reducing the bursitness of flow entry installment/deletion/update on switches.

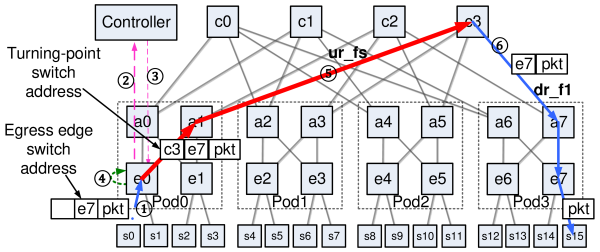
Adaptive Rerouting maintains load balancing on active routes in the subnet. As flows enter and exit the network, flow-sets' sizes may vary randomly. We monitor active routes' loads and adaptively reroute some flow-sets from high-loaded routes to low-loaded routes to maintain load balancing. For this technique, we can use many existing schemes.

C. Example

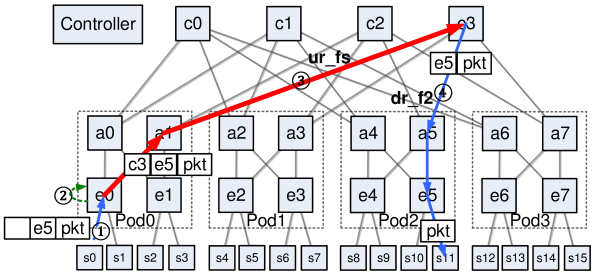
In Figure 3, we give an example that uses AggreFlow to schedule the same flow $f1$ as in Figure 2. For simplicity, we use a switch ID to represent the switch's address in the headers.

1) New Flow Routing: AggreFlow routes new flows as follows: In Figure 3(a), (1) the first packet of flow $f1$ enters the network from switch e_0. The packet's header is encapsulated with a blank header and address e_7, the address of its egress edge switch. (2) Switch e_0 cannot find flow $f1$'s flow-set, and then sends the routing request to the controller. (3) The controller informs switch e_0 flow $f1$'s flow-set fs, which is associated with hash value h and route $e_0 \rightarrow c_3$. (4) Switch e_0 inserts address c_3 into the packet's header. (5) Using address c_3, the packet is forwarded on its upstream

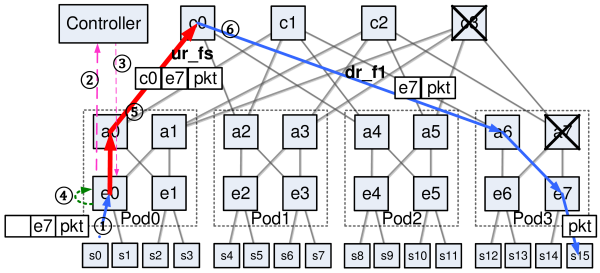
[2]In this paper, we use rerouting and route update interchangeably.



(a) AggreFlow routes a new flow $f1$ by creating a new flow-set fs.



(b) AggreFlow routes a new flow $f2$ by using an existing flow-set fs.



(c) AggreFlow reroutes an existing flow $f1$ by updating the route of the existing flow-set fs.

Fig. 3. Flow scheduling using AggreFlow in a 3-layer 4-pod fat-tree network F(3,4). ur_{fs} is flow-set fs's route; dr_{f1} and dr_{f2} are flows $f1$ and $f2$'s downstream routes, respectively.

route $ur_f : e_0 \rightarrow a_1 \rightarrow c_3$ to switch c_3. At switch c_3, the address c_3 is removed from the packet's header. (6) Using address e_7, the packet is forwarded on its downstream route $dr_{f1} : c_3 \rightarrow a_7 \rightarrow e_7$ to switch e_7. At switch e_7, the packet's header is removed. Finally, the packet is sent to its destination server s_{15}.

In Figure 3(b), new flow $f2$ with hash value h arrives at switch e_0. Switch e_0 finds that flow $f2$ belongs to flow-set fs, and then encapsulates c_3 into the packet's header without querying the controller. After the encapsulation, flow $f2$ is forwarded on flow-set fs's route ur_{fs} and its downstream route dr_{f2}.

In the routing procedure, the controller sends only one control message to switch e_0 to initialize flow-set fs by configuring fs's route ur_{fs}. After the initialization, no control messages are needed to configure routes for flows belonging to flow-set fs.

2) Existing Flow Rerouting: Figure 3(c) shows a process that uses AggreFlow to reroute flow $f1$ in the same minimum-power subnet as Figure 2. The process is explained below: At time t_1, switches c_3 and a_7 are turned off to save power, and the controller notifies all edge switches about the subnet

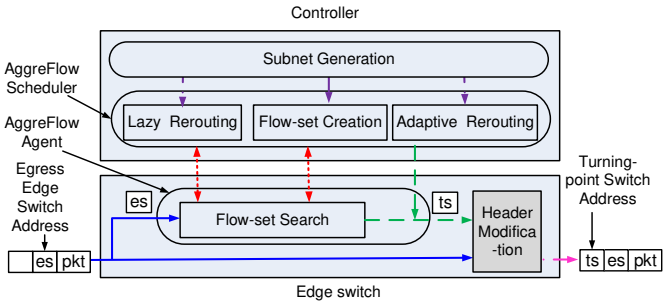


Fig. 4. AggreFlow processing procedure.

change. (1) At time t_2, a packet of flow $f1$ enters the network (assume it is the first packet that belongs to flow-set fs and enters the network at the subnet change). (2) Switch e_0 finds that flow-set fs's route is closed, and sends the rerouting request to the controller. (3) The controller informs switch e_0 to update flow-set fs's route to $e_0 \rightarrow c_0$. (4) Switch e_0 inserts address c_0 into the packet's header. (5)(6) Packet is forwarded via updated routes ur_{fs} and dr_{f1} to destination server s_{15}.

In the rerouting procedure, AggreFlow reroutes flow-set fs at time t_2. Compared with the flow-level scheduling that conducts rerouting operation at time t_1, AggreFlow postpones the operation a period of $t_2 - t_1$, and thus reduces switches' instant entry update overhead at t_1. Besides, the controller only sends one control message to switch e_0 to reroute flow-set fs. Assume flow-set fs contains N flows. For the worst case, flow-level scheduling consumes $N * 6$ control messages to reroute the N flows, while AggreFlow needs only one. Therefore, AggreFlow not only spreads rerouting operations over a relatively long time, but also reduces the number of control message for configuring rerouted flows.

IV. AggreFlow Design

In this section, we details AggreFlow's processing procedure and its modules.

A. AggreFlow Structure and Processing Procedure

Figure 4 shows AggreFlow's structure and processing procedure. AggreFlow consists of three components: subnet generation, AggreFlow scheduler and AggreFlow agents. Subnet generation considers the topology and traffic to determine the number and the location of active switches and links in the DCN. With the subnet's structure, AggreFlow scheduler and AggreFlow agents work together to efficiently schedule flows.

The input of an AggreFlow agent is the address of a packet's egress edge switch es, and the output is the address of the packet's turning-point switch ts[3]. The addresses of two switches indicate the packet's upstream and downstream routes. Switches use the two headers to forward the packet in the DCN. In the figure, each packet contains two headers when it enters a switch. In the DCN, each server is equipped with a system that stores the mapping relationship between servers and edge switches connected to the servers. Thus,

[3]In this paper, we use turning-point switch and route for flow-set interchangeably.

Algorithm 1 FlowsetSearch(p_f, T_{is}, R_{is})

Input:
p_f: flow f's packet;
T_{is}: flow-set routing table on ingress edge switch is;
R_{is}: the set of active routes connected to ingress edge switch is.
Output:
ts_{fs}: the turning-point switch address of flow-set fs that contains flow f.

1: **for** packet p_f arriving at switch is **do**
2: $h_f \leftarrow$ Hash(p_f);
3: use h_f to find (ts_{fs}, FLC_{fs}) in T_{is};
4: **if** $ts_{fs} == \varnothing$ **then**
5: $ts_{fs} \leftarrow$ FlowsetCreation(T_{is}, R_{is}, h_f);
6: **end if**
7: **if** $IsSubnetChanged == TRUE$ and $ShouldChangeRoute_{fs} == TRUE$ **then**
8: $ts_{fs} \leftarrow$ LazyRerouting(T_{is}, R_{is}, ts_{fs});
9: **end if**
10: **end for**

Algorithm 2 FlowsetCreation(T_{is}, R_{is}, h_f)

Input:
T_{is}: flow-set routing table on edge switch is;
R_{is}: the set of active routes connected to edge switch is;
h_f: packet p_f's hash value.
Output:
ts_{fs}: the turning-point switch address of flow-set fs that contains flow f.

1: $ts_{fs} \leftarrow$ LeastLoadRoute(R_{is});
2: $FLC_{fs} \leftarrow 0$;
3: map h_f to (ts_{fs}, FLC_{fs});
4: $T_{is} \leftarrow T_{is} \bigcup (ts_{fs}, FLC_{fs})$;
5: $ShouldChangeRoute_{fs} \leftarrow FALSE$;
6: **Return** ts_{fs};

before a packet leaves its source server, it is encapsulated with its header es. Considering MTU packets injected in the DCN cannot easily be expanded, we insert a blank header in advance and will change it to address ts after the processing. The header ts is selected by the modules detailed below.

B. Subset Generation

A fat-tree topology exhibits high regularity. Particularly, links have the same capacity, switches have the same size, and the topology is regular. We can take advantage of the regularity of a fat-tree network to determine whether to turn on/off switches or ports with much less computational burden. Specifically, in the fat-tree network, we activate all edge switches to accommodate traffic from servers. In each pod, the number of active aggregation switches equals the number of active links that support the aggregated uplink and downlink traffic. For example, in Figure 2, assume the rate of each link is 1 Gbps. If edge switch e_0 sends 1.5 Gbps of traffic up to the aggregation layer over two links, we must enable aggregation switches a_0 and a_1 to satisfy that demand. Similarly, the number of active core switches can be calculated based on the aggregated traffic between aggregation switch layer and core switch layer.

C. Flow-set Routing

Flow-set Routing is achieved by Flow-set Search and Flow-set Creation modules. Flow-set Search module routes a packet by searching the flow-set that has the same hash value with the packet. If a packet does not belong to any exiting flow-sets, Flow-set Creation module will create a new flow-set associated with the packet's hash value and select a route for the flow-set.

Algorithm 1 shows the pseudo code of Flow-set Search. In line 2, flow f's ingress edge switch is uses Hash function to compute flow f's hash value h_f. The computation includes two steps: (1) using CRC32 checksum algorithm to hash flow f's packet p_f's five tuples (i.e., source IP address, destination IP address, source port number, destination port number, and protocol field), (2) doing the mod operation on the result of the first step with $|T_{is}|$, the number of entries in switch is's flow-set routing table T_{is}[4]. A flow-set routing table T_{is} stores the route and load counter FLC of each flow-set that traverses switch is. The route of a flow-set is stored in the form of its turning-point switch address. An FLC records the number of packets that hit a flow-set entry in a period of time, and is used by Lazy Rerouting and Adaptive Rerouting modules to achieve load balancing. When the s idle timeout of an entry in the flow-set routing table expires, it will be removed.

In line 3, switch is uses hash value h_f to search flow f's flow-set fs's entry (ts_{fs},FLC_{fs}) in its flow-set routing table T_{is}. Lines 4 to 6 handles the case that flow f is a new flow that does not belong to any existing flow-sets. If flow-set fs's route does not exist, switch is will request the scheduler. The scheduler calls Flow-set Creation module to create a new flow-set fs for flow f and assigns a turning-point switch for flow-set fs based on the current network status, as addressed by Algorithm 2.

Lines 7 to 9 concern the case that the minimum-power subnet has changed and flow-set fs's route should be updated. $IsSubnetChanged$ is a boolean variable with default value $FALSE$, indicating an unchanged subnet. When the subnet changes, the scheduler sends the message $IsSubnetChanged = TRUE$ to each switch. Upon receiving the message, edge switches change boolean variable $ShouldChangeRoute$ of each flow-set to $TRUE$, which states a flow-set's route should be updated. If boolean variables $IsSubnetChanged$ and $ShouldChangeRoute_{fs}$ equal $TRUE$, switch is calls Lazy Rerouting module to update flow-set fs's turning-point switch ts_{fs} based on the current network status, as addressed by Algorithm 3.

Algorithm 2 describes the pseudo code of Flow-set Creation module. In line 1, the turning-point switch on the least loaded

[4]In this paper, we use entry and flow-set in the flow-set routing table interchangeably.

Algorithm 3 LazyRerouting(T_{is}, R_{is}, ts_{fs})

Input:
T_{is}: flow-set routing table on edge switch is;
R_{is}: the set of active routes connected to edge switch is;
ts_{fs}: the turning-point switch address of flow-set fs that contains flow f;
Output:
ts_{fs}^{new}: the address of the new turning-point switch of flow-set fs.

1: **if** ($ShouldChangeRoute_{fs} == TRUE$ *and* $route : is \rightarrow ts_{fs} \notin R_{is}$) **then**
2: $ts_{fs}^{new} \leftarrow$ LeastLoadRoute(R_{is});
3: $load_{is \rightarrow ts_{fs}^{new}} \leftarrow load_{is \rightarrow ts_{fs}^{new}} + FLC_{fs}$;
4: $ShouldChangeRoute_{fs} \leftarrow FALSE$;
5: **end if**
6: **if** ($ShouldChangeRoute_{fs} == TRUE$ *and* $load_{e \rightarrow ts_{fs} \rightarrow es_f} < load^{ave}$ *and* ReroutingDecision($route : e \rightarrow ts_{fs}$) $== TRUE$) **then**
7: $ts_{fs}^{new} \leftarrow$ LeastLoadRoute(R_{is});
8: $load_{is \rightarrow ts_{fs}^{new}} \leftarrow load_{is \rightarrow ts_{fs}^{new}} + FLC_{fs}$;
9: $load_{is \rightarrow ts_{fs}} \leftarrow load_{is \rightarrow ts_{fs}} - FLC_{fs}$;
10: $ShouldChangeRoute_{fs} \leftarrow FALSE$;
11: **end if**
12: **if** $ts_{fs}^{new} \neq \varnothing$ **then**
13: $ts_{fs} \leftarrow ts_{fs}^{new}$;
14: **end if**
15: **Return** ts_{fs}.

route in R_{is} (the set of active routes connected to ingress edge switch is) is selected as flow-set fs's the turning-point switch. In line 2, flow-set fs's load counter FLC_{fs} is initialized to 0. In lines 3 and 4, hash value h_f is mapped to flow-set fs's entry (ts_{fs}, FLC_{fs}), and this mapping is stored in flow-set routing table T_{is}. In line 5, $ShouldChangeRoute_{fs}$ changes to FALSE, indicating that flow-set fs's route is updated. In line 6, the turning-point switch of flow-set fs is returned to Flow-set Search module.

D. Lazy Rerouting

Algorithm 3 describes the pseudo code of Lazy Rerouting module. Lines 1 to 5 are concerned with the case that flow-set fs's old route is closed. When the minimum-power subnet changes to save power, flow-set fs's old route $is \rightarrow ts_{fs}$ does not exist in route-set R_{is} (the set of active routes that connect to ingress edge switch is). Thus, in route-set R_{is}, the turning-point switch on the least loaded route in R_{is} is selected as flow-set fs's new turning-point switch. FLC_{fs}, the traffic load of flow-set fs, is added to the traffic load of its new route $is \rightarrow ts_{fs}^{new}$. In line 4, after the route update, $ShouldChangeRoute_{fs}$ changes to $FALSE$ to state flow-set fs's route has been updated.

Lines 6 to 11 handle the case that one or more new routes are available for flow-set fs. When the minimum-power subnet changes to accommodate the increasing traffic demand, the turning-point switch of flow-set fs will be updated if it meets three requirements listed below: (1) flow-set fs's route has not been updated; (2) $load_{is \rightarrow ts_{fs}}$, the traffic load of route $is \rightarrow ts_{fs}$, does not reach the balancing counter threshold $load^{ave}$; (3) flow-set fs satisfies the rerouting probability:ReroutingDecision($route : is \rightarrow ts_{fs}$) $== TRUE$. Requirement (1) ensures that each flow-set is judged only once whether to be rerouted or not. This requirement prevents redundant rerouting judgments for the same flow-set. Requirement (2) ensures each active route with approximately equal traffic load. The balancing counter threshold $load^{ave}$ is the average traffic load of all active routes, and calculated by the sum of traffic loads of all active routes divided by the number of all active routes.

Requirement (3) prevents traffic starvation on existing-activated routes during the flow-set rerouting. If we only consider the first two requirements to reroute flow-sets, flow-sets are kept rerouting to the newly activated route(s) until the traffic loads of those newly activated routes reach the balancing counter threshold $load_{is}^{ave}$. Under such a situation, a few existing-activated routes could have no traffic for a transient time and experience traffic starvation. In order to prevent the undesired situation, RerouteDecision function sets the probabilities for rerouting flow-sets from their original existing-activated routes to newly activated routes. A flow-set is rerouted only when it is selected by RerouteDecision function.

If flow-set fs meets all the three requirements, ts_{fs}^{new}, FLC_{fs}, $load_{is \rightarrow ts_{fs}^{new}}$, $load_{is \rightarrow ts_{fs}}$ and $ShouldChangeRoute_{fs}$ are updated similar to lines 2 to 4. In lines 12 to 14, if a new turning-point switch ts_{fs}^{new} is selected for flow-set fs, ts_{fs} is updated to ts_{fs}^{new}. In line 16, the turning-point switch of flow-set fs is sent back to Flow-set Routing module.

E. Adaptive Rerouting

Adaptive Rerouting module monitors the traffic load of active routes and adaptively reroute some flow-sets to maintain load balancing. The traffic load of active route r, denoted $load\_r$, is periodically calculated and equals the total traffic load of flow-sets on route r. The traffic load of a flow-set is represented by the flow-set's load counter FLC. The AggreFlow scheduler can use OpenFlow meters to periodically pull $FLCs$ from edge switches [15]. Rerouting operations will be conducted on active route r if the absolute difference between $load\_r$ and $load^{ave}$ is larger than $load\_thd$, where $load^{ave}$ denotes the average traffic load of active routes, and $load\_thd$ is a preset threshold. If $load\_r > load^{ave}$ + $load\_thd$, a few flow-sets on route r are rerouted to other active routes with less loads so that $load\_r$ will approach to $load^{ave}$; if $load\_r < load^{ave} - load\_thd$, several flow-sets are rerouted from active routes with higher loads to route r until $load\_r$ close to $load^{ave}$.

V. Simulation

In this section, we evaluate the performance of AggreFlow in the fat-tree network simulation platform.

A. Comparison Schemes

We compare AggreFlow with two flow-level scheduling schemes.

Balance-oblivious flow-level scheduling [1]: The controller consolidates every flow to the leftmost route with sufficient capacity for the flow. For a single flow, a routing operation or a rerouting operation requires multiple control messages from the controller.

Balance-aware flow-level scheduling [4]: In a given minimum-power subnet, the controller routes each new flow to the least loaded route. As the subnet changes, the controller reroutes existing flows to the least loaded route one by one. The scheme's details are explained in Section II-C.

AggreFlow: AggreFlow employs Flow-set Routing, Lazy Rerouting and Adaptive Rerouting to efficiently schedule flows. The details are described in Section III-B. We let T(K) denote a flow-set routing table with K entries, and AggreFlow(K) denote AggreFlow scheme using T(K).

B. Simulation setup

We designed a packet-mode simulation platform on a NS-3 based fat-tree testbed. In the DCN, each link has the same rate, and each server sends a certain number of flows to all other servers. To emulate flow arrivals and terminations, each flow is given two states: ON and OFF. The ON status of a flow lasts a duration with an exponentially distributed random variable, which is determined when the flow is generated. The OFF status of a flow is the idle time of the flow and also lasts a duration of an exponentially distributed random variable, decided when the previous ON status finishes. The power of the DCN is the total power consumed by active switches and links/ports.

Generally speaking, DCNs usually incorporate some level of capacity safety margin to prepare for traffic surges [1]. In such cases, the network could allocate more capacity than essential for normal workload. To implement capacity safety margin ϕ, we monitor the utilization of each outgoing port/link of a switch. A new port on the same side of the switch will be enabled when the utilization exceeds 1-ϕ. Then, the corresponding port in another switch will be activated to establish the new link.

In our simulation, we use 3-layer 32-pod fat-tree network. Each link's rate is 1 Gbps, and the size of each packet is 1.5 KB. Each output port of a switch has a buffer space of 1,200 KB. The average ratio of ON period to OFF period is 5 [3]. Each flow is an inter-pod flow. The power status change of core switches causes flow rerouting. Traffic flows are generated in two separate slot intervals. In the first interval (0,60), each flow's arrival slot is a random variable in slot interval (0,40), and its termination slot is a random variable in slot interval (40,60). In the second slot interval (60,124), each flow's arrival slot is a random variable in interval (60,100), and its termination slot is a random variable in slot interval (100,124). No new flows are generated in slot intervals (40,60) and (100,124). Figure 5 shows the DCN utilization in our simulation. In slot intervals (0,40) and (60,100), the DCN utilization grows as the number of new flows increases. In slot intervals (40,60) and (100,124), the DCN utilization decreases as the existing flows terminate transmission. We take the power parameters of a switch from [1]. The capacity safety margin ϕ is set at 0.2.

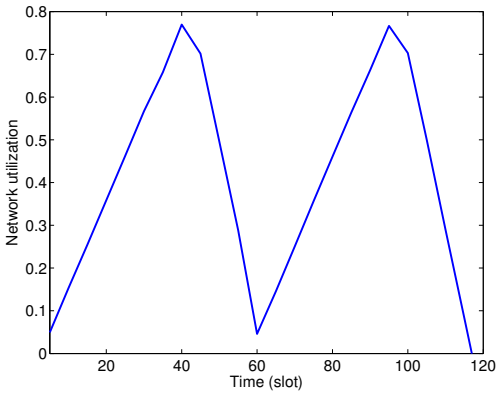


Fig. 5. Data center network utilization.

C. Simulation Results

In our simulation, we evaluate three aspects for each scheme: load-balancing performance, power consumption and QoS performance.

1) Load-balancing Performance: The load-balancing performance is evaluated in the form of Root Mean Squared Error (RMSE) of active routes in the DCN [10].

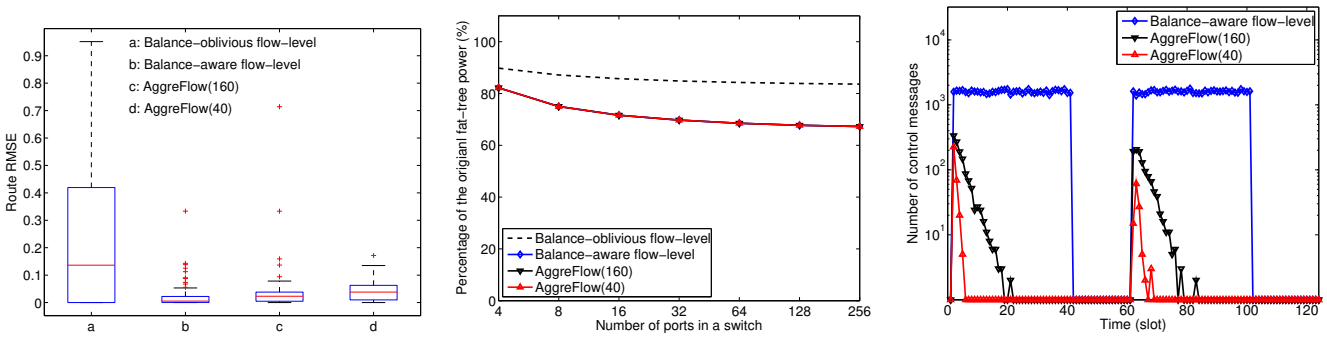
$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(load_l - load_{ave})^2}{N}} \quad (1)$$

where i denotes the i-th link in the network, the network consists of N links, $load_i$ denotes the i-th link's load, $load_ave$ denotes the average link load of N links in the network.

Figure 6(a) shows load-balancing performance of different flow scheduling schemes. A smaller RMSE, a better performance. If all active routes have the same load, RMSE is 0. In the figure, the box represents the center half of the data, and the red line represents the median data. The whiskers include 1-25-50-75-95-th percentiles of the data, and red crosses are 5% outliers.

Balance-oblivious flow-level performs worst since it greedily consolidates flows to the left routes in each switch layer. The unbalanced traffic allocation on active routes could degrade power consumption since extra switches and links may be activated to cater to bursty traffic surges on congested routes. Balance-aware flow-level represents the best performance because it conducts fine-grained flow-level routing and rerouting based on its global visibility.

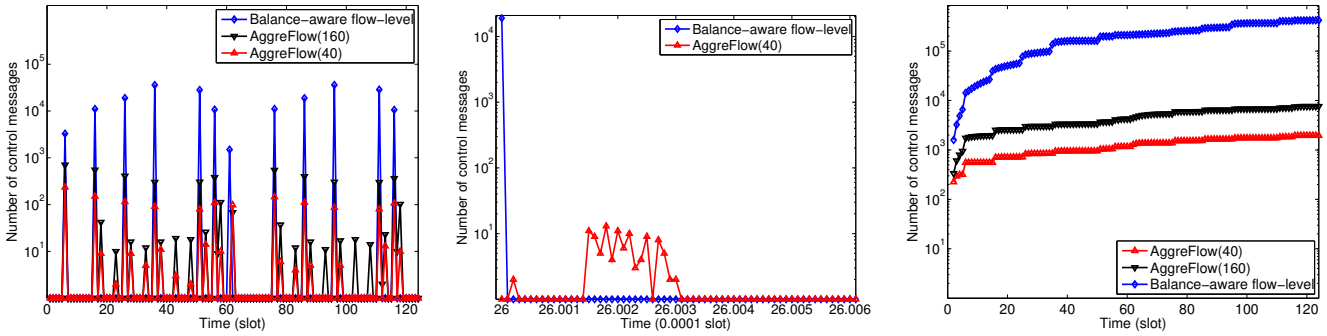
AggreFlow achieves the mean RMSE comparable to Balance-aware flow-level. AggreFlow's load-balancing performance can be further improved by using large flow-set routing tables. AggreFlow(160)'s load balancing performance is



(a) Load-balancing performance of different flow scheduling schemes.

(b) Power saving of the DCN using switches with different ports.

(c) Flow routing overhead of different flow scheduling schemes.



(d) Flow rerouting overhead of different flow scheduling schemes.

(e) Flow rerouting overhead in interval (26,26.006).

(f) Cumulative scheduling overhead of different flow scheduling schemes.

Fig. 6. Simulation results of different flow scheduling schemes. Balance-aware flow-level achieves the optimal power saving with varying traffic flows. In Figures 6(c)-6(f), we do not present the results of Balance-oblivious flow-level since it has the same result of Balance-aware flow-level.

more close to Balance-aware flow-level than AggreFlow(40). However, the better performance is at cost of higher control messages for rerouting more flow-sets.

2) Power Consumption: In our simulation, every 1 slot, the scheduler collects traffic statistics from switches and decides power status of switches and links. Each scheme selects active switches and links in a deterministic left-to-right order, so that unused switches and links are then turned off in a deterministic right-to-left order to save power. With such an active switch selection order, a specific number of active switches and links is coupled with only one minimum-power subnet.

For the metric of power consumption, we can divide flow scheduling schemes into two categories: the balance-oblivious scheme (i.e, Balance-oblivious flow-level) and the balance-aware scheme (i.e, all schemes except Balance-oblivious flow-level). For the entire simulation, both flow-level balance-aware schemes and Aggreflow consume the same power. Figure 6(b) shows the power efficiency of the schemes with the original fat-tree network using different switches. In the figure, the power saving increases as the number of switches' ports increase, and reaches the threshold 67% for the balance-aware schemes and 85% for the balance-oblivious scheme. Good load-balancing performance reduces power consumption by about 18% on average, while the unbalanced load allocation degrades power efficiency as the DCN's scale expands.

3) QoS Performance: We use four metrics to evaluate the QoS performace for each scheme: packet loss rate, routing overhead, rerouting overhead and cumulative scheduling overhead.

(1) Packet Loss Rate. Packet loss comes from the procedure of the minimum-power subnet reconfiguration. As traffic load increases, links have to accommodate more flows and become congested. When the load on the most congested link exceeds a pre-determined threshold (1-ϕ), a new minimum-power subnet is generated to relieve the current network congestion.

In our simulation, packet loss mainly comes from slots 15 and 75. As shown in Figure 5, traffic load increases equally at each slot interval. However, at the two slots, the subnet is small and thus performs worse than a large subnet to prevent packet loss when traffic surges occur. For the same subnet, the load-balancing performance impacts packet loss rate. Compared with the imbalanced load allocation, a balanced traffic allocation not only postpones the time that links reach the pre-determined thresholds, but also reduces the number of packets lost on the most congested links. Packet loss rate of Balance-aware flow-level, AggreFlow(40) and AggreFlow(160) are 0.19%, 0.193% and 0.193%, respectively. Although these AggreFlow schemes do not achieve the same load-balancing performance as Balance-aware flow-level does, their load-balancing performance is good enough to handle traffic surges on congested links.

(2) Flow Routing Overhead. We define the flow routing overhead as the number of control messages used for routing new flows at a single slot. Figure 6(c) shows flow routing

overhead of Balance-aware flow-level, AggreFlow(40) and AggreFlow(160). In the figure, Balance-aware flow-level consumes the highest overhead. In the two intervals (0,40) and (60,100), Balance-aware flow-level's overhead maintains about 1,500 control messages per slot since new flows arrive at the network in an approximate similar rate. In slot intervals (40,60) and (100,120), no new flows enter the network, and Balance-aware flow-level does not route any new flow.

AggreFlow(40)'s routing overhead is about 99% less than Balance-aware flow-level and only comes from the intervals (0,5) and (61,68). In intervals (0,5) and (61,68), AggreFlow creates new flow-sets as new flows enter the DCN. Specifically, in slot interval (0,5), the flow-set routing tables on different edge switches are initialized when the first flow of each flow-set enters the network. AggreFlow(40)'s the highest overhead comes from slot 1, the first slot to initialize flow-set routing tables, but it is still about 87.4% less than Balance-aware flow-level. As the number of initialized flow-sets increases, AggreFlow(40)'s overhead decreases. At the end of slot 5, the initialization completes. After the initialization, each flow-set is assigned with a route. Thus, in the slot interval (6,40), when an edge switch receives new flows, it finds a new flow's route from its flow-set routing table. In interval (40,60), no new flows enter the network, and some existing flows terminate transmission. As a result, a few flow-set entries are disabled from flow-set routing tables when their timeouts expire. Similarly, in slot interval (61,68), new flows enter the network and the DCN utilization increases. Under such a condition, some flow-set entries are initialized again in flow-set routing tables. AggreFlow(160)'s overhead is a little higher than that of AggreFlow(40) and comes from two longer slot intervals (0,18) and (61,83). AggreFlow(160) uses larger flow-set tables so that it requires more control messages and longer time to initialize its tables than AggreFlow(40) does.

(3) Flow Rerouting Overhead. The number of control messages used for rerouting existing flows at slot t is named flow rerouting overhead at slot t. Figure 6(d) shows flow rerouting overhead of Balance-aware flow-level, AggreFlow(40) and AggreFlow(160). In the figure, the overheads of all schemes vary as the DCN utilization changes shown in Figure 5. Balance-aware flow-level performs worst and consumes 36,067 messages at slot 96. This is because when each subnet changes, it reroutes a large number of flows, and each rerouting operation requires multiple control messages.

AggreFlow(40) reduces the overhead by about 99% compared with Balance-aware flow-level. Since the controller can reroute a set of flows by sending one control message to an edge switch, AggreFlow(40) requires much less control messages to reroute the same number of flows than Balance-ware flow-level does.

AggreFlow conducts rerouting operations in a relative long time. There are two reasons. First, using Lazy Rerouting, an AggreFlow agent reroutes a flow-set when it receives a packet belonging to the flow-set. Figure 6(e) shows the overhead consumed by rerouting operations in the interval (26,26.006). At slot 26, the minimum-power subnet changes, and Balance-aware low-level reroute flows immediately, whereas AggreFlow(40) spreads its rerouting operation over two separate slot intervals (26,26.0001) and (26.0011,26.0028). Second, using Adaptive Rerouting, AggreFlow dynamically reroutes some flow-sets to maintain load balancing when the scheduler detects imbalanced traffic loads on active routes.

In Figure 6(d), AggreFlow(160)'s overhead is larger than that of AggreFlow(40). AggreFlow(160) uses large flow-set routing tables and requires more control messages in each rerouting operation than AggreFlow(40) does.

(4) Cumulative Scheduling Overhead. Figure 6(f) shows cumulative overhead of different flow scheduling schemes. Compared with Balance-aware flow-level, AggreFlow(40) and AggreFlow(160) reduce cumulative overhead by about 99% and 98% on average, respectively.

VI. Discussion

In this section, we discuss some issues related to AggreFlow.

A. Prevent out-of-order Packets

In practice, we apply some actions to ensure a flow's packets arrive in-order after each rerouting operation. When one or more routes are about to be deactivated, flows will be forwarded via their new routes unless no packet of those flows is left on the soon-to-be-closed routes. When one or more new routes are activated, flows are not forwarded on their new routes until the left packets of the flows transmit termination on their original routes.

B. Header Encapsulation

Similarly to VL2 [8], AggreFlow can use an IP-in-IP encapsulation to insert the switch addresses. For a small DCN, we can give each switch a specific number to represent its address and reuse VLAN field to insert switch address [16]. Besides existing encapsulation techniques, we can also use state-of-the-art techniques (e.g., POF [17], PIF [18]) to design flexible packet headers.

C. AggreFlow Application Scenario

In DCNs, some network applications may pay attention to specific information contained in each packet of every flow. For example, in the applicaiton of network firewall, the packet is forwarded or dropped based on the rules that match the packet's source IP address, destination IP address and port numbers. AggreFlow can also be applied to such applications with small modification. For instance, we can conduct access control for each flow on edge switches (e.g., context-aware detection on packets of flows) before the flows are aggregated into flow-sets. We leave this issue for our future work.

VII. Related work

A. Data Center Cost Saving

In recent years, many studies have been conducted to optimize data center cost. Some researches introduce to save the power consumption of a data center by dynamically adjusting the efficiency of devices (e.g., servers [19], cooling system

[20][21]). Some other works propose to reduce the electricity cost of distributed data centers considering practical factors, such as time-of-use electricity rates [22][23][24][25][26][27], renewable energy availability [28][29][30].

Some recent studies consider the power consumption of network devices. ElasticTree [1] turns on and off switches and links based on the current traffic demand and consolidates traffic on a minimum-power subnet. CARPO [2] consolidates low-correlation flows together to further save power based on a observation that bandwidth demands of low-correlation flows usually do not peak at the same time in real DCNs. Widjaja et al. [3] explore the impacts of stage and switch size on power saving of a DCN. In [31], the authors further reduce power by considering the correlation between the DCN and servers.

Different from existing works that focus on saving power through adjusting power states of switches and links in DCNs, our work considers the impact of load balancing on power efficiency and solves the scalability problem to deploy a power-efficient DCN with time-varying traffic loads.

B. Flow Scheduling in DCNs

In Hedera [7], new flows are recognized as mice flows and routed by edge switches with oblivious static schemes. When a flow's transmission rate grows past a threshold rate, it is detected as an elephant flow and rerouted to a new route with less load. In DevoFlow [32], flows are classified into mice and elephant flows based on their transferred volumes. Mice flow routing are achieved by matching the exact-match flow entries, whereas DevoFlow controller reroutes elephant flows to the least congested path between the flows' end-hosts. Mahout [33] also focuses scheduling elephant flows, which are detected at end-hosts by looking at the TCP buffer of outgoing flows. The above schemes are designed for static DCNs, where all switches and links are always turned on. As the minimum-power subnet changes frequently, they may lead to imbalanced load on active routes or frequent control message storms.

VIII. Conclusion

In this paper, we identify two practical issues for deploying power-efficient DCNs: unbalanced traffic allocation of active routes could affect power efficiency; frequent control message storms would overwhelm OpenFlow switches. To achieve power saving and load balancing with low overhead, we propose a dynamic flow scheduling scheme named AggreFlow. AggreFlow schedules flows in a coarse-grained flow-set fashion, employs lazy rerouting to amortize the huge number of simultaneous rerouting operations over a relatively long period of time, and adaptively reroutes flow-sets to maintain load balancing on active routes. Simulation results show that AggreFlow achieves high power efficiency and good load-balancing performance with low overhead.

References

[1] B. Heller and et al., "Elastictree: saving energy in data center networks," in *USENIX NSDI'10*.

[2] X. Wang and et al., "Carpo: Correlation-aware power optimization in data center networks," in *IEEE INFOCOM'12*.

[3] I. Widjaja and et al., "Small versus large: switch sizing in topology design of energy-efficient data centers," in *IEEE/ACM IWQoS'13*.

[4] N. McKeown and et al., "Openflow: Enabling innovation in campus networks," 2008.

[5] K. He and et al., "Measuring control plane latency in sdn-enabled switches," in *ACM SOSR'15*.

[6] A. Wang and et al., "Scotch: Elastically scaling up sdn control-plane using vswitch based overlay," in *ACM CoNext'14*.

[7] M. Al-Fares and et al., "Hedera: Dynamic flow scheduling for data center networks." in *USENIX NSDI'10*.

[8] A. Greenberg and et al., "Vl2: a scalable and flexible data center network," in *ACM Computer Communication Review*, vol. 39, no. 4, 2009, pp. 51–62.

[9] T. Benson and et al., "Understanding data center traffic characteristics," *ACM Computer Communication Review*, vol. 40, no. 1, pp. 92–99, 2010.

[10] Z. Guo and et al., "Improving the performance of load balancing in software-defined networks through load variance-based synchronization," *Elsevier Computer Networks*, vol. 68, pp. 95–109, 2014.

[11] Extremetech, http://www.extremetech.com/extreme/161772-microsoft-now-has-one-million-servers-less-than-google-but-more-than-amazon-says-ballmer, 2013.

[12] T. Benson and et al., "Network traffic characteristics of data centers in the wild," in *ACM IMC'10*.

[13] B. Yan and et al, "Cab: A reactive wildcard rule caching system for software-defined networks," in *ACM HotSDN'14*.

[14] A. Vishnoi and et al., "Effective switch memory management in openflow networks," in *ACM DEBS'14*.

[15] OpenNetworkingFoundation, "Openflow switch specification v1.3.1," 2012.

[16] A. S. Iyer and et al., "Switchreduce: Reducing switch state and controller involvement in openflow networks," in *IFIP Networking'13*.

[17] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," in *ACM HotSDN'13*.

[18] P. Bosshart and et al., "P4: Programming protocol-independent packet processors," *ACM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[19] J. Heo and et al., "Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study," in *IEEE RTSS'07*.

[20] L. Barroso and et al., "The case for energy-proportional computing," *IEEE Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[21] C. Bash and et al., "Cool job allocation: measuring the power savings of placing jobs at cooling-efficient locations in the data center," in *USENIX ATC'07*.

[22] A. Qureshi and et al., "Cutting the electric bill for internet-scale systems," *ACM SIGCOMM'09*.

[23] L. Rao and et al., "Minimizing electricity cost: Optimization of distributed internet data centers in a multi-electricity-market environment," in *IEEE INFOCOM'10*.

[24] Z. Guo and et al., "Cutting the electricity cost of distributed datacenters through smart workload dispatching," *IEEE Communications Letters*, vol. 17, no. 12, pp. 2384–2387, 2013.

[25] Y. Zhang and et al., "Electricity bill capping for cloud-scale data centers that impact the power markets," in *IEEE ICPP'12*.

[26] J. Li and et al, "Towards optimal electric demand management for internet data centers," *IEEE Transactions on Smart Grid*, vol. 3, no. 1, pp. 183–192, 2012.

[27] Z. Guo and et al., "Jet: Electricity cost-aware dynamic workload management in geographically distributed datacenters," *Elsevier Computer Communications*, vol. 50, pp. 162–174, 2014.

[28] Y. Zhang and et al., "Greenware: greening cloud-scale data centers to maximize the use of renewable energy," in *Springer Middleware'11*.

[29] ——, "Testore: Exploiting thermal and energy storage to cut the electricity bill for datacenter cooling," in *IEEE CNSM'12*.

[30] Z. Liu and et al., "Greening geographical load balancing," in *ACM SIGMETRICS'11*.

[31] K. Zheng and et al., "Joint power optimization of data center network and servers with correlation analysis," in *INFOCOM'14*.

[32] A. R. Curtis and et al., "Devoflow: Scaling flow management for high-performance networks," in *ACM Computer Communication Review*, vol. 41, no. 4, 2011, pp. 254–265.

[33] ——, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *IEEE INFOCOM'11*.