Scalable Software-Defined Networking through Hybrid Switching

Hongli Xu[1] He Huang[2] Shigang Chen[3] Gongming Zhao[1]
Email: xuhongli@ustc.edu.cn, huangh@suda.edu.cn, sgchen@cise.ufl.edu,zgm1993@mail.ustc.edu.cn
[1]School of Computer Science and Technology, University of Science and Technology of China, China
[2]School of Computer Science and Technology, Schoow University, China
[3]Department of Computer & Information of Science & Engineering, University of Florida, USA

***Abstract*—Traditional networks rely on aggregate routing and decentralized control to achieve scalability. On the contrary, software-defined networks achieve near optimal network performance and policy-based management through per-flow routing and centralized control, which however face scalability challenge due to (1) limited TCAM and on-die memory for storing the forwarding table and (2) per-flow communication/computation overhead at the controller. This paper presents a novel hybrid switching design, which integrates traditional switching and SDN switching for the purpose of achieving both scalability and optimal performance. We show that the integration also leads to unexpected benefits of making both types of switching more efficient under the hybrid design. Numerical evaluation demonstrates the superior performance of hybrid switching when comparing with the state-of-the-art SDN design.**

I. Introduction

Scalability has been a core issue in the history of large network development. The conventional wisdom holds two design principles: aggregate routing paths and distributed control. Modern switches/routers forward packets from incoming ports to outgoing ports via switching fabric. The data plane uses ASIC hardware and on-die memory (such as SRAM) to process packets in real time at high speed. The on-die memory is typically a few megabytes. Increasing on-die memory is technically feasible, but it comes with a much higher price tag and access time is longer. There is a huge incentive to keep on-die memory small because smaller memory can be made faster and cheaper. To make the matter more challenging, limited on-die memory may have to be shared among routing/performance/measurement/security functions that are implemented on the same chip. The amount of on-die memory allocated for storing the forwarding (or routing) table will be limited, which makes per-flow routing (*i.e.*, one table entry for each flow) unscalable to a large network with millions of concurrent flows. To address this scalability problem, the classical design principle is to perform destination-based aggregate routing (instead of per-flow routing), where all flows with the same destination address or address prefix will share the same path. The second design principle is to decentralize the routing control function in order to avoid a single point of failure or performance bottleneck.

The emergence of software-defined networking (SDN) [1] [2] [3] [4] [5] [6] [7] has shattered both principles. It uses a centralized controller to determine per-flow paths and deploy these paths to the switches' forwarding tables. With the network-wide information at one place, the centralized control makes it far easier to enforce global policies and achieve optimal traffic management. These benefits outweigh the scalability concern in the compromise made by the SDN design. But the scalability problem will not go away under per-flow routing and centralized control. Today's SDN switches typically have a few thousands of entries in their on-die flow tables. Even the high-end Broadcom Trident2 chipset supports only 16K forwarding rules [5]. When there are too many flows to fit in the flow table, we will have to reject some flows [5], replace existing flows in the table with new flows (which causes churns and increases the load of the controller to repetitively deploy paths for the same flows), or bring aggregate routing back.

Forwarding rules with wildcards were proposed for aggregate routing [4]. But wildcard rules can only be implemented through TCAM (Ternary Content Addressable Memory), which is small, costly and energy-hungry. The small number of wildcard rules may result in aggressive aggregation when facing a large number of excess flows, whereas the benefits of SDN rest upon its ability of differentiating arbitrary individual flows. Moreover, there lacks systematic studies on how to construct and manage optimal wildcard rules in a dynamic, heavily loaded network, which is a challenging problem. Therefore, alternative or complementary solutions to the scalability problem are under call.

If traditional aggregate routing and decentralized control help scalability while SDN helps performance, our idea is to integrate them for hybrid switching, which achieves the benefits of both worlds and is not subject to the restriction of TCAM. This paper presents a novel hybrid switching design. On the one hand, it leverages the mature methods of traditional switching to achieve scalability by avoiding per-flow communication/computation overhead to the controller and reducing the number of forwarding rules needed to support a large number of flows. On the other hand, it exploits the flexibility of SDN switching to achieve near optimal network performance without overflowing the forwarding table. More interestingly, we show that the integration of traditional switching and SDN switching brings unexpected benefits to each other. The SDN's centralized control will help implement traditional switching much more efficiently. In the meanwhile, with a hybrid deployment design, we show that traditional aggregate routing will help greatly reduce the overhead of deploying SDN paths. We also discuss how to perform per-flow traffic measurement

Authorized licensed use limited to: University at Buffalo Libraries. Downloaded on December 03,2020 at 10:59:16 UTC from IEEE Xplore. Restrictions apply.

without using the OpenFlow counters in the forwarding table, which is important in our hybrid switching design where many (or even most) flows are not in the table. Finally, we give a case study on how to perform global optimization at the controller for flow re-routing under hybrid switching. Our numerical evaluation shows that the proposed hybrid switching design outperforms wildcard-based DevoFlow [4] by significantly lowering the number of forwarding rules under the same traffic conditions or achieves much better network performance under the same forwarding-table size.

It should be pointed out that our hybrid switching study is fundamentally different from the prior work on partial or mixed SDN networks where SDN switches and traditional devices co-exist [3] [8] [9] [10]. The concern there is how to make these two types of switches operate together. Their SDN switches face the same scalability problem as described earlier. This paper does not consider partial SDN networks. We study full SDN networks where traditional switching is integrated with SDN switching at all devices to improve scalability.

The rest of this paper is organized as follows. Section II discusses the related work. Section III introduces the switching, routing, and hybrid schemes. We present the design of the hybrid switching in Section IV. The simulation results are presented in Section V. Section VI draws the conclusion.

II. Related Work

SDN was introduced to improve network performance through centralized control [1]. For example, the centralized traffic engineering service of B4 [2] was able to drive links to near 100% utilization, with load balancing among alternative paths. However, the limited size of the forwarding table and the communication/computation/management load at the controller place constraint on the scalability of such a centralized design.

To address these problems, the prior art mainly uses wildcards. The Plug-n-Serve system [11] and the Aster*x system [6] use wildcards to aggregate multiple flows into a single rule (occupying one table entry). Their use of wildcards is limited to the suffixes of source address. But as each wildcard rule bundles many flows together based on their address adjacency, it partially compromises the SDN's flexibility of differentiating arbitrary individual flows in traffic engineering.

Hedera [12] was designed to re-route large flows in a datacenter network that has a structured topology such as fat tree [13] to provide probabilistic default path selection among multiple alternative choices. DevoFlow [4] is more general for arbitrary network topologies. It combines pre-deployed wildcard rules and dynamically-established exact rules, *with a design goal of reducing the need to involve the controller in setting up paths for new flows*. Limited by the small size of TCAM, the number of wildcard rules is small, and each wildcard rule may have to match numerous flows. In order to differentiate individual flows (so as to measure their individual sizes and re-route them as needed for load balancing), when a new flow matches a wildcard rule, an exact rule specifically for that flow will be created based on the template of the wildcard rule, without involving the controller. DevoFlow successfully solves the controller's load problem, but it does not address the problem of limited table size, particularly if we want to retain the flexibility of differentiating arbitrary individual flows. It still uses per-flow rules, each costing hundreds of bits.

Cohen *et al.* [5] studied the effect of forwarding-table size on network utilization. They formulated this problem as an NP-hard optimization problem and presented approximate algorithms. They assumed that when a switch's forwarding table is full, new flows will simply be dropped. Huang *et al.* [14] considered splittable flows, each of which is allowed to follow multiple paths to improve network utilization. They studied joint optimization of rule placement and traffic engineering for QoS provisioning. Our paper considers unsplittable flows, such as TCP flows whose window adaptation may be adversely affected if packets of the same flow follow different paths.

III. Switching, Routing, and Hybrid

A. Traditional Switching

A traditional Ethernet switch uses a *switch table* to learn reachability information from the packets (or data frames in layer-2 terminology) that it receives. When a switch receives a frame from a port, it learns that the source MAC address in the frame header can be reached from that port. This information is stored in the switch table where each entry contains an MAC address and a port number.

If a switch receives a frame whose destination MAC address is in the switch table, the switch will forward the packet to the corresponding port. Otherwise, it will forward the frame to all ports except for the port from which the frame is received, generating a broadcast. For two-way communication between two hosts, broadcast will happen only once because the first exchange between the hosts will let all switches along the communication path learn how to reach them. To achieve high throughput, the switch table is often implemented as a hash table in SRAM [15].

The above design has the advantage of high switching efficiency, low processing overhead, and low hardware/software cost. It can scale to very large networks. But it does not provide path selection and traffic measurement in support of network-wide traffic engineering. Packets are forwarded along fixed paths determined by the network topology without the input of dynamic load conditions or user-specified traffic policies.

B. Traditional Routing

Routers or layer-3 switches are able to perform distributed path selection through routing protocols such as OSPF [16] for intra-domain routing or BGP [17] for inter-domain routing. The modern router architecture consists of a control plane, where the routing protocols and management functions are implemented, and a data plane, which handles packet forwarding at high speed. To achieve high throughput, the *routing table* can be cached in on-die SRAM at the arrival network interface. Each routing-table entry consists of a destination address prefix, an output port, and other fields.

The path selection is coarse-grained. It is destination-based, not flow-based, which helps reduce the table size. There may be many flows from a source network to a destination network. They will all follow the same path because they share the

Authorized licensed use limited to: University at Buffalo Libraries. Downloaded on December 03,2020 at 10:59:16 UTC from IEEE Xplore. Restrictions apply.

same destination address prefix. This limits the flexibility in offering quality of service, balancing load, utilizing the under-used alternative paths, or performing flow-level management policies.

C. Software-Defined Switching

Comparing with traditional switching/routing, a fundamental difference of the SDN architecture is its centralized control. An SDN network consists of three types of devices: a central controller, SDN switches that are inter-connected to form a network, and end hosts that are connected to the switches. An SDN switch has a *forwarding table* specifying per-flow paths. Each table entry contains a forwarding rule, which has 10 fields defined in OpenFlow [18], specifying source MAC address, destination MAC address, source IP address, destination IP address, protocol, source port, destination port, output port, instruction, and stat counters, totaling 288 bits [1]. The forwarding table is typically implemented in TCAM to support wildcard fields and parallel lookup of all table entries. For exact rules without wildcards, they may be implemented in TCAM or SRAM. Although we view the forwarding table logically as a single table, it may be implemented by multi-tiered tables [19].

When a switch receives a packet, it matches the fields in the packet's headers against the table. If there is a matching entry, the packet is handled according to the instruction field, which may drop, log, or forward the packet to the output port. When there is no matching entry, the switch sends a request carrying the packet header to the controller which selects a path for the flow and installs proper rules on the switches along the path. The controller knows the network topology and collects traffic statistics from the switches. With this information and user-defined policies, the controller makes path selection.

The centralized control architecture is simple. It takes the control plane out of the devices and pushes most of the complexity to the controller, leaving the switches only with its data plane. Not only does this simplify the data-forwarding devices, but the centralized control makes it easier to enforce complex traffic management policies.

However, the shortcomings of the centralized control are also obvious. The controller can become a performance bottle-neck. It has to set up the paths of all flows, incurring per-flow computation overhead for path selection and per-flow communication overhead of transiting a packet header to the controller and the forwarding rules back to the switches along the selected path, together with acknowledgement packets, as well as flow statistics collection. Such per-flow overhead becomes significant when most flows are short with only a small number of packets, which is unfortunately the common case [15].

Wildcard rules help relieve the problem by aggregating multiple individual flows into a single entry. However, the aggregation is based on the adjacency of the addresses/ports, not based on the individual flows' policy/performance require-ments. Because TCAM is costly and uses a lot of energy, the number of wildcard rules will be small, which means that aggressive aggregation will be necessary for a large network with numerous flows. This certainly limits how individual flows can be routed across the network for best overall performance.

D. Idea of Hybrid Switching

Both traditional switching/routing and SDN switching have pros and cons. The former adopts coarse-grained, destination-based path selection for space saving. With careful use of available table space, today's switches and routers are able to scale to very large networks. The latter provides fine-grained path selection and management functions at the flow level. On the one hand, fine-grained per-flow paths require more forwarding rules. But on the other hand, each forwarding rule takes much more space than an entry in the traditional switch table or routing table. That means the number of available forwarding rules will be smaller, given the same memory space. This is even more true if TCAM is used for implementing the forwarding table. Fewer forwarding rules in availability and more rules in demand contradict each other in system design.

To solve this problem and relieve the computation/commu nication bottleneck at the controller, we propose *hybrid switch-ing* that combines traditional switching/routing and SDN switching for the benefits of both worlds. We refer to the forwarding paths used in traditional switching/routing as *tradi-tional paths*, and the path in SDN switching as *SDN paths*. A device that performs hybrid switching is called a *hybrid switch*, and its benefits are summarized below.

First, studies on real network traffic showed that most flows were short-lived with light traffic [20]. Routing them via SDN's forwarding tables has little lasting impact on network performance and does not justify the associated overhead. Moreover, it causes additional delay due to communication with the controller, path selection and deployment, which hurts flow performance. With hybrid switching, we will direct these flows through the traditional paths, avoiding per-flow overhead to the controller and reducing the number of forwarding rules needed.

Second, studies also showed that the elephant flows domi-nate in traffic volume although their number may be relatively small [20]. From the traffic engineering's point of view, it is economic to focus on these flows and route them via optimal SDN paths for desired network performance. SDN switching allows us to directly control a selected number of flows that have the most impact.

Third, with the help of centralized control from SDN, we will be able to implement traditional switching/routing much more efficiently. With the help of traditional paths, we will be able to implement SDN paths much more efficiently. Hybrid switching is not a simple combination of SDN switching and traditional switching/routing, but instead a full integration, in which the two are inter-twined and help each other to make both perform better.

Note that in the rest of the paper, whenever we refer to *forwarding table*, we imply SDN switching. Similarly, *switch table* implies traditional switching, and *routing table* implies traditional routing.

Authorized licensed use limited to: University at Buffalo Libraries. Downloaded on December 03,2020 at 10:59:16 UTC from IEEE Xplore. Restrictions apply.

E. Dual Switch v.s. Hybrid Switch

A dual switch is a simple combination of an SDN switch and a traditional switch. In practice, it can be used in either role. When it is configured for SDN switching, it does not use its switch table; when it is configured for traditional switching, it does not use its forwarding table. Hence, a dual switch is different than a hybrid switch, which integrates SDN switching and traditional switching into one. As we will see shortly, the implementation of traditional switching in a hybrid switch is different from the old way (Section III-B; it involves the controller!). As far as we know, there are dual switches available today, but not hybrid switches.

IV. Design of Hybrid Switching

This section goes step by step in explaining our design of hybrid switching. First, we discuss how to implement traditional switching and traditional routing more efficiently with the help of a centralized controller. Second, we integrate SDN switching with traditional switching/routing. Third, we discuss how to identify large flows by per-flow statistics measurement in a compact space without using the counters in the forwarding tables. Fourth, we design hybrid path deployment that exploits traditional paths to reduce the number of forwarding rules needed for SDN paths. Fifth, we combine all the pieces into an overall design of hybrid switching.

A. Traditional Switching with A Controller

We consider an SDN network where each switch also implements a traditional switch table, in addition to the forwarding table for SDN switching. We explain how to integrate switch tables into the SDN architecture and implement them more efficiently. Before we proceed, we stress that this work is different from research on a *partial SDN* network [8], which is a mix of SDN switches and traditional switches, where SDN switches are dual switches (Section III-E), allowing them to operate with neighboring traditional switches. In this work, we assume a *full SDN* network where all switches are SDN switches. More precisely, they are hybrid switches, implementing the traditional switch tables *with the help of a centralized controller*.

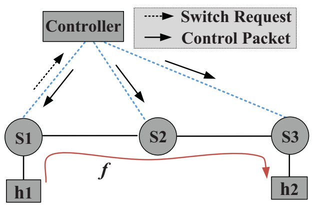

Fig. 1: Implementing switch tables with the help of a centralized controller

When a switch receives a data frame and does not find a matching entry in its switch table, it sends a request, carrying the destination MAC address, to the controller. The controller has the full knowledge of the switches, the hosts, and the network topology. It finds a path to the destination. When there are multiple paths, it selects one based on certain criterion such as shortest distance. It will then send a control packet, with the proper switch-table entry, to every switch on the path, including the requesting switch, as shown in Figure 1. The data frame, as well as all subsequent frames in the flow, will be forwarded along this path unimpeded towards its destination. With the help of the controller, we avoid the broadcast — which may reach all end hosts of the whole network — when a matching entry cannot be found.

B. Traditional Routing with A Controller

Next we consider an SDN network where each switch is a layer-3 switch, which implements a traditional routing protocol, in addition to its forwarding table. We use OSPF [16] as example. In the classical implementation of OSPF, every router periodically sends the state of its adjacent links to all other routers, and receives such information from other routers as well. Therefore, all routers have the same view of the whole network, based on which they compute their routing tables. Now, with a centralized controller, all routers only need to periodically send their link states to the controller, which collects a global view of the network, computes the routing tables of all routers, and updates the routers with the changes in their routing tables.

In the original OSPF, each router receives $O(|E|)$ link states, where E is the set of links in the network. The overall communication complexity for all routers is $O(|N|\ |E|)$, where N is the number of routers. With the controller's help, routers do not receive link states. Only the controller does, with a communication complexity $O(|E|)$ for the whole network. For the routing-table update, only the difference will be transmitted and the controller is able to contain such difference to a small amount by adopting a route computation algorithm that keeps the stability of the routes. In fact, even in traditional OSPF networks, the protocol is often configured to compute routes based on hop counts to avoid route churns [16]. In an SDN network, the controller has more reasons to adopt such a strategy because it has another tool, SDN switching, for dealing with traffic engineering and load balancing on different paths.

C. Integration of SDN Switching with Traditional Switching/Routing

We first consider the integration of SDN switching with traditional routing. When a switch receives a packet, it matches the packet against both the SDN forwarding table and the traditional routing table. As long as the forwarding table has a matching entry, it takes the precedence and the packet will be forwarded accordingly. If the packet belongs to a new flow and the forwarding table does not have a matching entry, there are two path selection strategies.

1. Traditional Path First (TPF): New flows will take the traditional paths by default without causing any immediate overhead to the controller. For a packet from a new flow, without a match in the forwarding table, the switch will handle the packet according to the routing table, which will always give a matching entry, meaning that it can scale to an arbitrary number of flows. New flows will not automatically generate requests to the controller for path selection, in contrast to

Authorized licensed use limited to: University at Buffalo Libraries. Downloaded on December 03,2020 at 10:59:16 UTC from IEEE Xplore. Restrictions apply.

what today's SDN switches do. This property helps reduce the controller's communication/computation burden and avoid a potential performance bottleneck in the system. While all new flows follow the traditional paths by default, the switches will monitor their flows, identify the large ones, and estimate their sizes. Periodically they will send the information of the identified large flows to the controller, which performs global optimization to improve network performance by re-routing some or all of the large flows via optimal SDN paths, subject to the size constraint of the forwarding tables at the switches. The formulation of the optimization is dependent on the user-specified performance and management requirements, which vary in practice; we will provide a case study in the next section. The controller will then update the switches' forwarding tables by installing the new paths; see [21] [7] for update schemes that ensure packet-level routing consistency.

2. SDN Path First (SPF): New flows will take the SDN paths by default. For a packet from a new flow, without a match in the forwarding table, as long as the switch's forwarding table is not overflown, it will forward the packet header to the controller for installing an SDN path. If the forwarding table is full, the switch forwards the packet based on the routing table.

Although SPF solves the overflow problem of forwarding tables, it still faces other problems of SDN switching as explained in Section III-C: per-flow communication/computation overhead to the controller (even for small flows that contain a few packets themselves) and extra delay to a flow's first packet due to the setup of SDN path. We advocate TPF not only because it avoids these problems but also because batch setup of forwarding paths for a set of flows together tend to produce better global optimization than setup of the paths one at a time sequentially.

Next, we consider the integration of SDN switching with traditional switching under TPF. When a switch receives a data frame, it matches the frame against both the SDN forwarding table and the traditional switch table. As long as the forwarding table has a matching entry, the frame will be processed based on that entry; otherwise, if the switch table has a matching entry, the frame will be forwarded to the specified output port. If there is no matching entry in either table, the switch will send a request, carrying the frame's destination MAC address, to the controller, which will establish a traditional path towards the destination and install proper entries in the forwarding tables along the path. Again, all switches will monitor their flows and send the information of large flows to the controller, which will perform global optimization periodically by re-routing large flows to optimal SDN paths.

D. Hybrid Path Deployment

When the controller decides to re-route a flow from its traditional path to an SDN path p, under the traditional wisdom, the controller must deploy one forwarding rule at every switch on p, occupying an entry in the switch's forwarding table [7] [5]. However, the proposed hybrid switching offers a new opportunity to save forwarding-table space. Consider an arbitrary switch s on p. Let t be the output port specified in the forwarding rule that the controller intends to deploy at s. If t is also what the traditional path from s to the destination specifies, there is no need to actually deploy the forwarding rule because without this rule, switch s will use the traditional path automatically, which will forward the packet to port t.

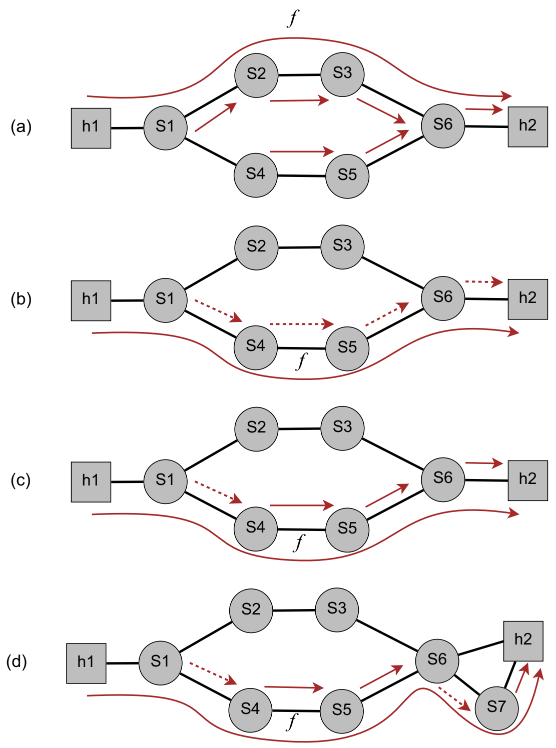

Fig. 2: Illustration of hybrid path deployment. (a) Flow f follows the traditional path, which is specified by solid arrows. (b) Flow f is re-routed to an SDN path, $s_1 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6$, where the forwarding rules are shown as dashed arrows. (c) Due to hybrid deployment, only one forwarding rule at s_1 is actually deployed. (d) In a slightly different example, we show that more than one forwarding rule may be used to deviate the SDN path from the traditional path for more than once.

Figure 2 shows an example of hybrid path deployment, where the top plot shows a flow f passing through a traditional path from host h_1 to host h_2. Assume all switches already have proper matching entries for h_2 in their switch tables (or routing tables), as shown by solid arrows in the top plot. Now the controller wants to re-route f to a different path, $s_1 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow h_2$, requiring four forwarding rules to be deployed at four switches, as shown by dashed arrows in the second plot. The forwarding rules at s_4, s_5 and s_6 specify the same outports as the traditional paths do. With hybrid deployment, the controller only needs to deploy one forwarding rule at s_1, where the rest of the SDN path follows the traditional path, as the third plot shows. In a more complicated case of the fourth plot, two forwarding rules are deployed at s_1 and s_6, respectively, allowing the SDN path to deviate from the traditional path for a second time at s_6.

As a side benefit, hybrid path deployment can help reduce the dependency in the order of forward-rule deployment, which in turn helps reduce the deployment time. According to [21] [7], in order to ensure packet-level routing consistency, we

Authorized licensed use limited to: University at Buffalo Libraries. Downloaded on December 03,2020 at 10:59:16 UTC from IEEE Xplore. Restrictions apply.

should perform two-phase deployment when re-routing f, with the first phase installing the forwarding rules at s_4, s_5 and s_6 (the third plot), and then the second phase installing the forwarding rule at s_1; see the original paper [7] for reasons. Under hybrid path deployment, we need only one phase of installing the rule at s_1 in this example, which cuts down deployment time.

E. Per-flow Statistics and Large-flow Identification

According to the OpenFlow specification [22], each entry in the forwarding table has statistic counters for per-flow traffic measurement. However, this is insufficient for our design because many or even most flows will follow the traditional paths specified in the switch table (or routing table) where there is no per-flow entry. More importantly, under TPF in Section III-D, all new flows follow the traditional paths by default. When we want to find the large ones among them for re-routing, the counters in the forwarding table will not help. We need to adopt a new mechanism to collect per-flow statistics without incurring too much space overhead as the size of SRAM is limited. The idea is for each switch to produce a traffic synopsis, a compact data structure that summarizes the traffic of all flows passing the switch and supports queries on individual flows.

There is a rich literature on per-flow size measurement under tight memory. For a few examples, the *Multiresolution Space-Code Bloom Filter* (MSCBF) [23] employs multiple Bloom filters to encode packets with different sampling probabilities; the *Counter Braids* (CB) [24], [25] is a counting architecture that maps flows to counters in multiple tiers for space reduction; the method of *randomized counter sharing* (RCS) [26] was proposed to further reduce memory requirement and processing time by using virtual storage vectors. We adopt RCS because its per-packet overhead is very small (updating a single counter) and it can support an arbitrary number of flows with a pre-allocated memory space. For completeness, we adapt RCS in the context of this paper below.

To cover all flows, it is sufficient for only the edge switches to perform traffic measurement, which relieves the core switches from this overhead. During each measurement period, every edge switch uses RCS to build a traffic synopsis, which is an array C of counters. Let c be the size of the allocated array. The ith counter in the array is denoted as $C[i]$, $0 \leq i \leq c-1$. Each flow is mapped to v counters that are pseudo-randomly selected from C based on the hash output of the flow identifier, which typically contains addresses/ports from the packet's headers. These v counters logically form a *storage vector* of the flow, denoted as C_f, where f is the identifier of the flow. The ith counter of the vector, denoted as $C_f[i]$, $0 \leq i \leq v-1$, is selected from C as follows: $C_f[i] \equiv C[H_i(f)]$, where $H_i(...)$ is a hash function whose range is $[0, c)$. The operation of online data encoding is very simple: During each measurement period, when the switch receives a packet, it extracts the flow identifier f from the packet's headers, randomly selects a counter from C_f, and increases the counter. It should be stressed that the array C can support a virtually arbitrary number x of flows, each of which simply takes v counters from C to record its information. When $x \gg c$, the counters will be heavily shared by different flows. In a shared counter, one flow's information is other flows' noise.

Each edge switch also samples the arrival packets to record a number of flow identifiers. Large flows have proportionally higher probabilities to be sampled. At the end of every measurement period, it estimates the size of each sampled flow f by summing up the counters in C_f after subtracting away a noise term, which is simply the average counter value across the whole synopsis. According to the analysis in [26], RCS gives very accurate estimates for large flows whose sizes are much larger than the average flow size. This fact is confirmed in our experiments. Knowing the sizes of the sampled flows, each edge switch reports its largest k flows and their sizes to the controller, where k is a system parameter set by the controller. As we will show in the case study, the computation overhead of the controller is dependent on the number of flows to be re-routed. By adjusting this value, the controller has a way to prevent the switches from overloading itself.

F. Overall Design

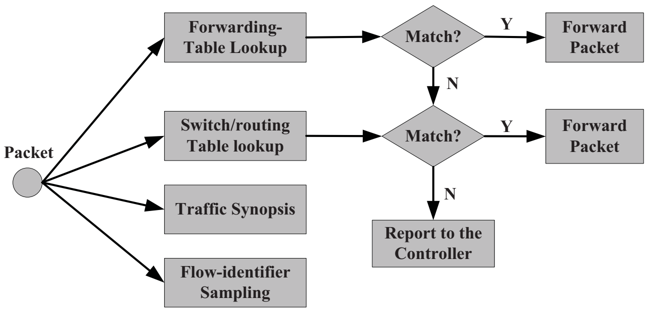

Fig. 3: Illustration of Real-Time Packet Processing

The overall design is illustrated through a packet-processing flow chart in Figure 3. When a packet arrives at an input port of a switch, it is processed with forwarding-table lookup and switch/routing table lookup, which may be carried out in parallel by ASIC hardware on chip. We adopt the TPF strategy: The packet will be handled by the forwarding table if a matching entry exists. If not, the packet will be forwarded based on the switch/routing table if a matching entry exists. In case of switch table, there may not exist a matching entry. When this happens, the switch will report the destination MAC address to the controller for path selection. In the meantime, for an edge switch, the packet will also be processed for flow sampling and synopsis-based traffic measurement.

Besides the above real-time packet processing, at the end of each measurement period, every edge switch will estimate the sizes of the sampled flows based on the information recorded in the synopsis. It reports the largest k flows and their estimated rates to the controller for possible re-routing, where a flow's estimated rate is the estimated flow size divided by the measurement period.

G. A Case Study on Re-routing via Optimal SDN Paths

The component missing so far is the controller's global optimization in re-routing large flows via SDN paths. However,

Authorized licensed use limited to: University at Buffalo Libraries. Downloaded on December 03,2020 at 10:59:16 UTC from IEEE Xplore. Restrictions apply.

the implementation of this component is directly related to the performance goal and the management policies, which are set by the system admin and vary greatly in practice. Nevertheless, the hybrid switching architecture in the previous section has an open design that can accommodate any implementation of path selection. For the purpose of completeness and numerical evaluation, we provide a case study below with one implementation; its properties should not be viewed as limitation of our hybrid switching design because this specific implementation can be replaced with other implementations in practice without affecting the rest of the overall design as discussed in Section IV-F.

We first give the notations: Denote the set of n switches as $S = \{s_1, ..., s_n\}$, and the set of m hosts (terminals) as $H = \{h_1, ..., h_m\}$. The network topology is modeled as a graph $G = (S \cup H, E)$, where E is the set of links. Let $c(e)$ be the capacity of a link $e \in E$ and $l(e)$ be the load of the link. The switches measure traffic loads on all their ports (*i.e.*, adjacent links) and make the information available to the controller through Openflow [1].

From the large flows reported by the switches, the controller selects a subset Π of the largest ones for re-routing. The size of Π may be constrained by the budget of execution time for solving the optimization problem; the relationship between the size of Π and the execution time can be roughly estimated based on the past executions. Let $r(f)$ be the estimated rate of flow $f \in \Pi$, which is reported by the edge switch of the flow. Let $\mathcal{P}(f)$ be the set of candidate paths for flow f. $\mathcal{P}(f)$ is determined based on the management policies and performance objectives. For example, if there are too many possible paths that satisfy the management policies, we may include only a certain number of the best ones under a certain performance criterion, such as having the shortest number of hops or having the large capacities. $\mathcal{P}(f)$ also contains the path $p^*(f)$ that the flow is currently routed through.

Let $y_f^p \in \{0, 1\}$ be an indicator variable for whether flow f will be routed on a path $p \in \mathcal{P}(f)$. Let $T(s)$ be the number of residual entries in the forwarding table at switch s. Let $I(f, p, s)$ be a binary value for hybrid path deployment (Section IV-D): if path p assigned to flow f overlaps with the flow's traditional path at switch s, then there is no need to deploy an entry on the forwarding table of switch s, *i.e.*, $I(f, p, s) = 0$; otherwise $I(f, p, s) = 1$. We formalize the global load-balancing optimization as a non-linear program.

$$\min \quad \lambda$$
$$s.t. \begin{cases} b(e) = l(e) - \sum_{f \in \Pi : e \in p^*(f)} r(f), & \forall e \in E \\ \sum_{p \in \mathcal{P}(f)} y_f^p = 1, & \forall f \in \Pi \\ b(e) + \sum_{f \in \Pi} \sum_{p \in \mathcal{P}(f): e \in p} y_f^p r(f) \leq \lambda \cdot c(e), & \forall e \in E \\ \sum_{f \in \Pi} \sum_{p \in \mathcal{P}(f): s \in p} y_f^p \cdot I(f, p, s) \leq T(s), & \forall s \in S \\ y_f^p \in \{0, 1\}, & \forall p, f \\ \lambda \leq 1. \end{cases} \tag{1}$$

The first set of equations computes the background traffic load $b(e)$, $\forall e \in E$, when the flows in Π are taken out. The second set of equations requires that flow $f \in \Pi$ is not splittable; it will be forwarded through a single path from $\mathcal{P}(f)$. The third set of inequalities states that the traffic load on each link e, which is the sum of background traffic load and traffic from the flows scheduled on the path, should not exceed $\lambda \cdot c(e)$, where λ is the load-balance factor (less than 1) that will be minimized. The fourth set of inequalities describes the size constraints of the forwarding tables. The objective is to minimize λ. Globally reducing the link loads and achieving load balance have many benefits. It helps prevent large queuing delays that happen when λ approaches towards 1. It leaves room for new flows or allows the existing flows to increase their rates for better network throughput. It indirectly helps balance the occupation of the forwarding tables as flows are spread among alternative paths. We point out that the above non-linear program will always have at least one feasible solution — all flows $f \in \Pi$ take their current paths $p^*(f)$, which means no re-routing and all flows keep the status quo.

The integer programming problem can be solved numerically for a small network topology with the size of Π determined by the controller. There are also approximate methods that convert integer programming into linear programming for which efficient numerical solutions exist. An example is relaxation and random rounding [27]. It relaxes Eq. (1) by replacing the fifth line of integer constraints with $y_f^p \geq 0$, turning the problem into linear programming. After solving the linear programming problem, we round y_f^p to zero or one probabilistically based on its value. We use this method in the numerical evaluation of the proposed work and it produces very good results.

V. Numerical Evaluation

In this section, we will present the results from large-scale simulation (since one focus of this paper is scalability).

A. *Performance Metrics and Evaluation Setting*

We evaluate the proposed hybrid switching (HS) through simulations and compare it with the most-related, state-of-the-art work of DevoFlow [4]. Both of them try to improve the scalability of a large SDN system, but their design goals are somewhat different: The main goal of HS is to address the problem of limited forwarding-table size and to reduce the overhead of the controller. The main goal of DevoFlow is to reduce the overhead of the controller, while achieving high network performance.

We use the following performance metrics in our numerical evaluation: (1) the maximum number of forwarding rules (or flow entries needed) on any switch at any time during the simulation; (2) the average number of forwarding rules per switch at any time during the simulation; (3) the maximum network throughput; (4) the maximum load factor of any link; (5) the communication overhead to/from the controller. During a simulation run, at each time instance, we measure the maximum number of forwarding rules per switch and the average number of forwarding rules on any switch. We use their largest values over time during the simulation as the first two metrics. As we continuously increase the number of flows, we measure the maximum throughput that the network can support. The load factor of a link is the traffic load divided by the link capacity. The load balancing metric is the maximum load factor

Authorized licensed use limited to: University at Buffalo Libraries. Downloaded on December 03,2020 at 10:59:16 UTC from IEEE Xplore. Restrictions apply.

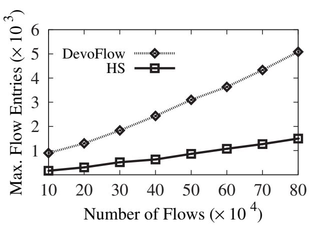
Fig. 4: Maximum number of flow entries without FTS constraint

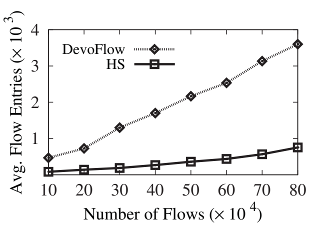
Fig. 5: Average number of flow entries without FTS constraint

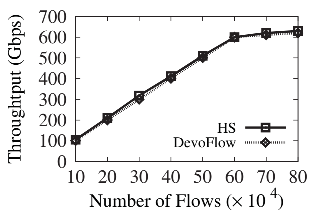
Fig. 6: Network throughput without FTS constraint

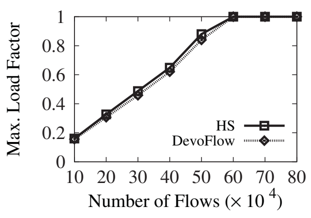
Fig. 7: Maximum load factor without FTS constraint

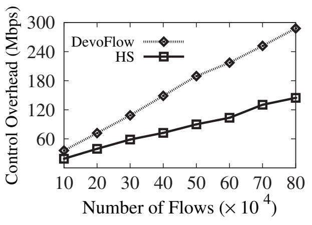
Fig. 8: Communication overhead without FTS constraint

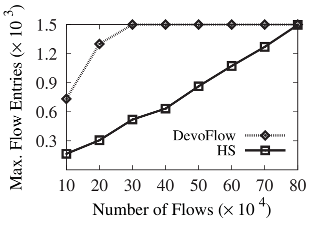
Fig. 9: Maximum number of flow entries with FTS constraint

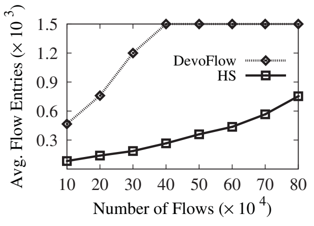
Fig. 10: Average number of flow entries with FTS constraint

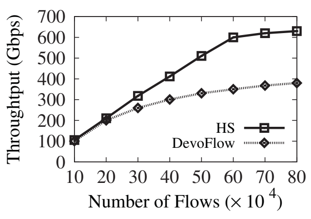
Fig. 11: Network throughput with FTS constraint

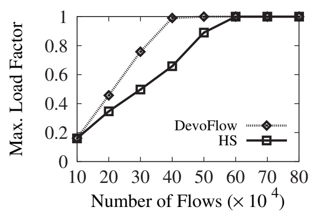
Fig. 12: Maximum load factor with FTS constraint

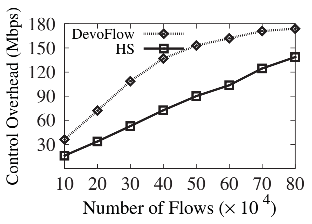
Fig. 13: Communication overhead with FTS constraint

among all links. We also measure the total communication traffic to/from the controller, divided by the time period of simulation, which gives the fifth metric. By default, each switch will report the traffic estimation information of the largest 1,500 (elephant) flows at most to the controller.

Our simulations adopt the two-dimensional HyperX topology [28], which has 81 access switches, each attached to 16 other switches. All links are 1Gbps, and 20 servers will be attached with each access switch. So, the HyperX topology has 1620 servers. We use power law for the flow-size distribution, where 20% of all flows account for 80% of traffic volume [20]. We generate three types of flows: (1) random flows, whose sources and destinations are randomly picked; (2) server flows, which simulate the traffic between random hosts and a number of designated servers, *e.g.*, mail servers and web servers; (3) associative flows, which simulate the traffic between a subnet and a server, *e.g.*, communications between the finance department and the finance database or between a hospital and a datacenter that houses the patient data. Each type of flows accounts for one third of total traffic.

The simulations are performed under two scenarios. The first scenario has no forwarding-table size (FTS) constraint, assuming that the switches have sufficient space to handle all flows. This hypothetical scenario tests the performance of HS (DevoFlow) when the table size is not a limiting issue. The second scenario has an FTS constraint and tests how well HS (DevoFlow) performs when the table size becomes a problem.

B. Performance Comparison without FTS Constraint

Our first set of simulations compares HS and DevoFlow in the scenario without FTS constraint. The results are shown in Figures 4-8, where the horizontal axis is the number of flows in the network, ranging from 10×10^4 to 80×10^4. In Figures 4 and 5, as the number of flows increases, there are more elephant flows as well. As a result, the maximum/average numbers of flow entries (or forwarding rules) increase in both HS and DevoFlow. In comparison, the proposed HS solution uses much fewer flow entries than DevoFlow. For example, when there are 50×10^4 flows (about 300 flows per server), HS uses a maximum number of 600 flow entries on a switch, while DevoFlow uses 3,100; HS uses 400 flow entries per switch on average, while DevoFlow needs 2,200 on average. More specifically, HS reduces the required flow entries by about 80.5% compared with DevoFlow. That's because our hybrid path deployment mechanism helps to reduce the required flow entries by combining traditional routing and SDN routing.

Figures 6-7 show that HS and DevoFlow achieve similar network performance, including throughput and load balancing. The reason is that without FTS constraint, both designs can dynamically schedule the elephant flows for efficient routing. Figure 8 shows that HS has much smaller communication overhead at the controller than DevoFlow. As the number of flows increases, DevoFlow deploys more forwarding rules than HS (see Figure 5), which results in higher control overhead. For example, when there are 80×10^4 flows (about 500 flows per server), the control overheads of HS and DevoFlow are about 140Mbps and 290Mbps, respectively.

C. Performance Comparison with FTS Constraint

The second set of simulations compares HS and DevoFlow with FTS constraint, where the forwarding table size is set to 1,500 entries [4]. The results are shown in Figures 9-13, where the horizontal axis is the number of flows in the

Authorized licensed use limited to: University at Buffalo Libraries. Downloaded on December 03,2020 at 10:59:16 UTC from IEEE Xplore. Restrictions apply.

network. In Figures 9-10, HS needs much fewer flow entries than DevoFlow, whose table is saturated much earlier. As an example, when there are 50×10^4 flows in the network, HS uses a maximum number of 800 flow entries in any switch, while DevoFlow uses 1,500 (with the forwarding table becoming full); HS uses an average of 400 flow entries per switch, while DevoFlow uses 1,500 on average, meaning that all forwarding tables are saturated. This saturation has a performance impact, as shown below.

When the number of flows is less than 30×10^4, the forwarding table is not made full by DevoFlow in Figure 10. In this case, HS and DevoFlow achieve similar network performance, including throughput and load balancing, as shown in Figures 11-12. On the contrary, when the number of flows is more than 30×10^4 and the forwarding table is made full by DevoFlow, HS outperforms DevoFlow because the routing flexibility of the latter is constrained. For example, when there are 80×10^4 flows in the network, HS improves network throughput by 63% when comparing with DevoFlow. Note that, From Figure 12, when the number of flows exceeds 60×10^4, the maximum load factor is close to 1 for both DevoFlow and HS. However, HS can forward more flows than DevoFlow by Fig. 11. Since HS deploys a fewer number of forwarding rules than DevoFlow, it incurs smaller control overhead, as shown in Figure 13.

VI. Conclusion

In this paper, we have designed a novel hybrid switching mechanism, which integrates traditional switching and SDN switching for the purpose of achieving both scalability and optimal performance. Moreover, a hybrid path deployment method has been presented to reduce the required forwarding rules. Numerical evaluation demonstrates the superior performance of hybrid switching when comparing with the DevoFlow solution. In the future, we will study efficient flow prediction mechanism, which may help to reduce the number of forwarding rules and to enhance the network route performance.

Acknowledgement

The corresponding author of this paper is He Huang. This search of Xu and Zhao is supported by NSFC under No. 61472383, U1301256 and 61472385, and NSF of Jiangsu in China under No. BK20161257. The research of Huang is partially supported by NSFC under Grant No. 61572342 and 61672369, NSF of Jiangsu Province under Grant No. BK20151240 and BK20161258. The research of Chen is supported by NSF under No. STC-1562485.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[3] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic engineering in software defined networks," in *IEEE INFOCOM*, 2013, pp. 2211–2219.

[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 Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 254–265.

[5] R. Cohen, L. Lewin-Eytan, J. S. Naor *et al.*, "On the effect of forwarding table size on sdn network utilization," in *Proc. IEEE INFOCOM*. IEEE, 2014, pp. 1734–1742.

[6] N. Handigol, S. Seetharaman, M. Flajslik, A. Gember, N. McKeown, G. Parulkar, A. Akella, N. Feamster, R. Clark, A. Krishnamurthy *et al.*, "Aster* x: Load-balancing web traffic over wide-area networks," 2011.

[7] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 539–550.

[8] S. Vissicchio, L. Vanbever, and O. Bonaventure, "Opportunities and research challenges of hybrid software defined networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 70–75, 2014.

[9] Y. Guo, Z. Wang, X. Yin, X. Shi, and J. Wu, "Traffic engineering in sdn/ospf hybrid network," in *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*. IEEE, 2014, pp. 563–568.

[10] J. He and W. Song, "Achieving near-optimal traffic engineering in hybrid software defined networks," in *IFIP Networking Conference (IFIP Networking), 2015*. IEEE, 2015, pp. 1–9.

[11] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *ACM SIGCOMM Demo*, 2009.

[12] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *NSDI*, vol. 10, 2010, pp. 19–19.

[13] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[14] H. Huang, S. Guo, P. Li, B. Ye, and I. Stojmenovic, "Joint optimization of rule placement and traffic engineering for qos provisioning in software defined network," *IEEE Transactions on Computers*, no. 1, 2015.

[15] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 181–192, 2005.

[16] J. Moy, "Ospf version 2," 1997.

[17] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (bgp-4)," Tech. Rep., 2005.

[18] "The openflow switch," openflowswitch.org.

[19] K. Kannan and S. Banerjee, "Compact tcam: Flow entry compaction in tcam for power aware sdn," in *International Conference on Distributed Computing and Networking*. Springer, 2013, pp. 439–444.

[20] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009, pp. 202–208.

[21] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 323–334.

[22] O. S. Specification-Version, "1.4.0," 2013.

[23] A. Kumar, J. Xu, and J. Wang, "Space-code bloom filter for efficient per-flow traffic measurement," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2327–2339, 2006.

[24] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter Braids: A Novel Counter Architecture for Per-Flow Measurement," *Proc. of ACM SIGMETRICS*, June 2008.

[25] Y. Lu and B. Prabhakar, "Robust Counting Via Counter Braids: An Error-Resilient Network Measurement Architecture," *Proc. of IEEE INFOCOM*, April 2009.

[26] T. Li, S. Chen, and Y. Ling, "Fast and compact per-flow traffic measurement through randomized counter sharing," *Proc. of IEEE INFOCOM*, pp. 1799–1807, April 2011.

[27] T. Friedrich and T. Sauerwald, "Near-perfect load balancing by randomized rounding," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*. ACM, 2009, pp. 121–130.

[28] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "Hyperx: topology, routing, and packaging of efficient large-scale networks," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 41.

Authorized licensed use limited to: University at Buffalo Libraries. Downloaded on December 03,2020 at 10:59:16 UTC from IEEE Xplore. Restrictions apply.