A novel Internet access service with online traffic engineering of elephant flows

Sourav Maji, Malathi Veeraraghavan, Molly Buchanan, Fatma Alali, Jordi Ros-Giralt, and Alan Commike

Abstract—Elephant flows, which are high-rate large-sized flows, can cause increased packet delays and losses in other flows. Large enterprises often upgrade their access links even before the load reaches a high level to avoid service degradations caused by elephant flows. This work proposes and demonstrates an alternative solution to access-link upgrades. The solution consists of deploying a high-speed Elephant Flow Traffic Engineering System (EFTES) that monitors access-link traffic, identifies elephant flows in real time, and instructs the router to add a firewall filter to isolate identified elephant flows to a separate queue. The EFTES algorithms were implemented in an innovative high-speed, multicore system called R-Scope, which leveraged features of high-performance network interface cards such as packet coalescence and kernel bypass. This EFTES implementation was evaluated using real 10-Gbps Center for Applied Internet Data Analysis (CAIDA) traffic traces. EFTES could handle a 1-min traffic trace that contained 1.5 M flows with 8 cores. Experiments were conducted on the NSF GENI and Chameleon testbeds at 1 and 10 Gbps, respectively, to demonstrate the value offered by EFTES to delay-sensitive flows in the presence of artificial elephant flows that were added to a background replay of the CAIDA packet traces.

Index Terms—Elephant flows, Traffic engineering, High-speed networks.

I. INTRODUCTION

In the Internet, customer-provider relationships exist between enterprises and their ISPs, and between lower- and higher-tier ISPs. Today, if a customer wants a higher level of service from their ISPs, the most common option is to increase access link capacity. In this paper, we propose an alternative method in which, under certain conditions, a customer can obtain a higher level of service without upgrading their access links. We call this service Internet Access with Online Traffic Engineering (IAOTE).

Specifically, the IAOTE service is realized with a newly designed system called Elephant Flow Traffic Engineering System (EFTES), which identifies elephant flows (high-rate, large-sized flows) in real-time, and redirects these flows to a separate queue for isolation from other traffic. Given the long-tail nature of flow rates [1], only a few flows will have high enough rates to warrant redirection. Furthermore, if elephant flows do not occur frequently, it may be less expensive to deploy an EFTES to improve overall traffic performance (avoid the packet losses and delays caused by elephant flows) than it would be to increase access link capacity.

When a customer upgrades its access link to its provider, the customer network planners typically select a rate that is high enough to accommodate long-term growth in traffic volume. Fig. 1 illustrates a typical timeline. A customer has upgraded its link rate to $r_0$ at time $t_0$ expecting that this link can be used until time $t_1$, when the aggregate traffic volume is expected to grow to a level that warrants a link upgrade to rate $r_1$. However, at time $t_1$, computer system technologies, such as host I/O bus speeds, disk access rates, etc., improve to the point that a single flow or a set of parallel flows (i.e., elephant flows) from a cluster can use the entire access link for a dataset transfer. In the past 3 decades, $t_1$ has occurred before $t_2$ for some customers, e.g., in upgrades from T1/T3 to OC3 (when 100 Mbps Ethernet NICs became popular), OC3/OC12 to OC48 (when 1GE NICs became widespread), 1 GE to 10 GE (as 10GE NIC adoption increased), and now from 10 GE to 100 GE.

When elephant flows start appearing on the customer’s Internet access link, other traffic could be adversely affected due to packet losses and/or packet delays. Therefore, customers often upgrade their access link rates at time $t_2$, soon after $t_1$, instead of waiting until $t_3$ when the aggregate traffic volume justifies the upgrade as illustrated in Fig. 1. Based on the inter-arrival time between elephant flows, and the duration of these flows, there may be a period of time, e.g., between $t_2$ and $t_3$, during which the customer can benefit from our newly proposed IAOTE service.

EFTES can also be used within enterprise or provider networks to improve performance on links that occasionally carry elephant flows. Typically, the cost of upgrading such intra-domain links is lower than the cost of upgrading Internet access links since a customer is charged connection fees by a provider for the latter. Therefore, we anticipate EFTES being useful for a service such as IOATE, but EFTES could also be used within enterprises/provider networks.

In designing EFTES, the main challenge lies in identifying elephant flows in real-time on a high-speed link that carries millions of simultaneous flows. Also, a method is needed for determining an appropriate rate threshold and duration threshold for making the decision of whether or not a flow has the potential to be an elephant. Many of the prior algorithms [2]–[7] designed to identify high-rate flows do not determine absolute flow rates, but rather these algorithms seek to identify...
the flows with the highest rates. But if the absolute rate of a flow is not high enough to fill the headroom of a link, the flow is not a threat to other traffic, and hence should not be rerouted. Therefore, we designed new algorithms to measure per-flow rate.

The main contributions of this work are: (i) design of EFTEs and elephant-flow identification algorithms; (ii) EFTEs implementation in our high-performance, multi-core system called R-Scope @ [8]; (iii) evaluation of the EFTEs prototype with a 10-Gbps traffic trace; (iv) experiments with elephant flows; and (v) analysis of real 10-Gbps traffic traces.

The rest of the paper is organized as follows: Section II reviews prior work on elephant flow identification. The EFTEs architecture and algorithms are described in Section III. In an experimental evaluation of a high-speed implementation of EFTEs, we used real traffic traces that were collected on a 10-Gbps link at San Jose by the Center for Applied Internet Data Analysis (CAIDA) [9]. Therefore, before presenting the results of our evaluation of our high-speed EFTEs implementation, in Section IV, we describe three sets of analyses carried out on these traffic traces. These analyses informed our choice of parameters in the evaluation of our EFTEs implementation, which is described in Section V. Next, Section VI describes several experiments that we carried out to illustrate the basic value offered by EFTEs, and to study the impact of different TCP-layer and network-path parameters on the effects of elephant flows. Section VII concludes the paper.

II. RELATED WORK

Work on elephant-flow identification has been ongoing for more than a decade. In 2003, Estan and Varghese [10] proposed a method for online identification and measurement of heavy-hitter flows without having to track all flows. Our solution uses the approach proposed in this work that once a flow entry is created in the flow cache, the flow entry is updated for all packets of the flow. In other words, there is no pre-lookup packet sampling, but post-lookup packet sampling is included to limit the number of entries in the flow cache. Further, a shared goal in both our paper and this 2003 paper is to determine if a flow’s rate has exceeded a threshold during a set time interval. Unlike in the 2003 paper, which uses a multistage filter with multiple hashing functions, we use a more direct method of saving the timestamps of the first and last packets of flows to determine the actual rate. We show that the cost of processing these fields is not that high.

Solutions that aim to improve processing-power efficiency use pre-lookup packet sampling. Various papers [2]–[7], published from 2004 to 2010, propose the use of Bayesian and other such approaches for elephant-flow identification in the presence of packet sampling. These approaches compute the fraction of packets sent by a flow from a total of say D packets collected within an interval. Since there is no tracking of timestamps, the exact rate of a flow cannot be determined, which is required for our objective.

A 2015 paper [11] classifies elephant-flow identification algorithms into three categories: (i) per-flow maintenance and every-packet processing, (ii) partial-flows maintenance and packet sampling, and (iii) partial-flows maintenance and every-packet processing. Our algorithm falls in the last category since no flow-cache entry is created for flows with only small-sized packets.

Prior work with similar goals as ours, i.e., traffic engineering of elephant flows, include SIFT [12] and a proposal [13] to move large IP flows to lambda-connections, which are optical lightpaths established dynamically through an underlying network of optical WDM circuit switches that interconnect IP routers. In SIFT, flows whose total size exceeds a threshold are moved to a low-priority queue. There is no concept of measuring rate, and therefore a long-running but low-rate flow that potentially does no harm to other flows could also be moved over. Our solution aims to find only those flows that exceed a rate over a short duration that is comparable to the time taken to fill a switch buffer, as only these flows are likely to have adverse effects. The lambda-connections approach recommends redirecting aggregated IP flows, specifically subnet-to-subnet or AS-to-AS flows, while our solution redirects 5-tuple flows (application-to-application flows) to a separate queue. As flow durations may not be that long, our solution does not attempt to create a new circuit; instead our solution uses the DiffServ queuing approach.

From the 2010-2011 timeframe, we review three papers: Hedera [14], DevoFlow [15] and Mahout [16]. For datacenter networks, a dynamic flow scheduling algorithm called Hedera was proposed. As the goal of this solution is to efficiently utilize network resources, two steps are executed: a demand estimation process to find the maximum host-limited rate at which each source-destination pair of a flow can exchange data, and then a network path is found dynamically to accommodate all flows. This solution is not feasible for wide-area, multi-domain flows, which is our context. For software defined networks, DevoFlow proposes to (i) classify flows that have transferred 1-10 MB as elephant flows, (ii) find the least congested path for all elephant flows, and (iii) configure the paths using OpenFlow to set flow-specific table entries. As with other work, rate is not estimated with timestamps, and path-setup delay may be too large an overhead for our goal of fast redirection to minimize adverse effects. Mahout uses the approach of identifying elephant flows at the end hosts, but this solution requires software upgrades of end hosts while our approach is transparent to end hosts.

Newer papers from 2013 [17] and 2014 [18] propose a scheme with adaptive thresholds, and a machine-learning predictive scheme, respectively. Details of how the thresholds are modified are not provided. In our approach, we describe how measured background network traffic, packet drop rate and rate of identified elephant flows are combined in an adaptive approach for setting thresholds.

Finally, 2014-2016 papers [11], [19], [20] focus on the engineering challenge of keeping up with high packet rates. Streaming algorithms such as bloom filters and sketches, and LRU policies, are used to improve efficiency and execution speed. However, these algorithms are summary based and do not provide the rates of individual flows. Our algorithms are designed to determine the rates of flows using an O(1) operation for high-performance implementation.
In prior work [21], [22], we presented experimental studies for the effects of redirecting elephant flows to separate queues. In an analytical study [23] of NetFlow records collected from four ESnet routers over a 7-month period, we found that elephant flows are repetitive in nature. Therefore, an offline analysis can be conducted on NetFlow records to extract source and destination IP addresses of elephant flows, and these address pairs can be used to preset firewall filter rules in routers to redirect future elephant flows to separate queues or virtual circuits provisioned on paths that are different from the default IP-routed paths. This solution was effective for use on access links from major US Department of Energy (DOE) supercomputing facilities that are connected to ESnet, but less effective (67% vs 92%) on a link from a commercial peer into ESnet. The work presented in this paper is an online scheme for identifying elephant flows on these type of inter-domain links across which elephant flows are less predictable.

III. Elephant Flow Traffic Engineering System

Section III-A presents the overall architecture of the IAOTE service, and describes the two components of EFTES: Elephant Flow Identification Engine (EFIE) and an IAOTE Service Management System (SMS). Section III-B describes EFIE, and Section III-C describes IAOTE SMS.

A. IAOTE service architecture

![Fig. 2: IAOTE service architecture using EFTES](image)

Fig. 2 illustrates how the IAOTE service can be realized with one EFTES connected to a port in a provider router PR1 and one EFTES connected to a port in a customer router CR1. Packets sent on the access link are mirrored by routers PR1 and CR1 to the ports connected to the EFTES. The mirrored traffic is sent to the Elephant Flow Identification Engine (EFIE), and IAOTE Service Management System (SMS).

The EFIE classifies captured flows as elephants or monitored (details of the algorithms are provided in Section III-B), and when a flow is classified as an elephant, the EFIE sends the flow identifier directly or indirectly (through an SDN controller) to the router to set a firewall filter rule to send matched packets arriving on all input ports to a separate elephant-flow (EF) queue for transmission out on the access link as illustrated in Fig. 2. With this simple DiffServ [24] solution of having two queues with bandwidth borrowing, elephant-flow packets can be isolated thus preventing packet losses or delays in other flows. If this action is required in both directions of the access link, an EFTES would need to be deployed in both customer and provider networks.

The main functionality of the IOATE SMS is to compute and update the threshold parameters used by the EFIE. Specifically, the EFIE classifies a flow as an elephant if its rate, computed over a duration threshold, exceeds a rate threshold. These two thresholds, duration threshold and rate threshold, are computed and updated periodically by SMS. These thresholds do not need to be updated all the time; rather the SMS checks EFIE performance metrics periodically, and determines whether or not to update the thresholds. The algorithm executed by the SMS is described in Section III-C.

B. Elephant Flow Identification Engine (EFIE)

The EFIE maintains a flow cache with each entry storing the timestamps of the first-observed and last-observed packets of a flow, packet count, cumulative byte count, flow state, and location in a Least Recently Used (LRU) flow list (see Table 1). When a packet arrives, if its flow identifier (flowID) is not in the flow cache, a new flow entry is created. For each subsequent packet arrival, the last-packet timestamp, packet count, and byte count fields are updated. When a packet arrives for a flow that has been ongoing for longer than a duration threshold $T$, if its average rate up to that instant in time is greater than a high-rate threshold $R$, then the flow is classified as an elephant. For example, if $T = 1 \text{ sec}$, and $R = 1 \text{ Gbps}$, then the flow would have sent at least 1 Gbit (125 MB), which is large for many networks, by the time it is classified as an elephant. Therefore, in using both duration and rate thresholds, flows that are classified as elephants are both high-rate and large-sized.

If the EFIE decides that a flow is an elephant, the EFIE instructs the router to set a firewall filter to redirect packets of the elephant flow to a separate EF queue. The flow’s state variable is modified from monitored (which is the default initial state) to redirected. If the EFIE decides that a flow is not an elephant, the flow entry is removed from the flow cache. A flow-cache entry could be created and deleted multiple times within the lifetime of a flow. Every time a flow-cache entry is created, the flow rate is computed after a time period equal to the duration threshold, and compared with the rate threshold.

The last element of a flow-cache entry, location in LRU list, as shown in Table 1, supports eviction of flow entries. The LRU list is a linked list with a front variable and a back variable. When a packet arrives, the location in LRU list field is extracted from the corresponding flow cache entry, and placed in the front variable. The address stored in the back variable shows the location of the flow-cache entry that should be evicted when space is required for a new flow. This structure makes the process of evicting a flow entry an efficient $O(1)$ operation.

The flow cache is sized to be large enough to hold flows for a multiple of the duration threshold used to make the elephant-flow decision. Therefore, if a flowID at the back of the LRU
TABLE I: Notation

| Packet $P$ | {arrival time instant, flow identifier, and length} |
| Flow cache entry $FC[f]$ | {first packet timestamp, last packet timestamp, packet count, byte count, flow state, location in LRU list} |
| Algorithm (configurable) parameters: $(L, S, T, R, I)$ | packet length threshold, post-lookup sampling probability, duration threshold, rate threshold, inactivity threshold |
| $D$ and $R$ | duration and rate of a flow |

Algorithm 1: Handle incoming packets

Input: packet $P$

Initialization: $t \leftarrow$ arrival time instant $f \leftarrow$ flow ID of $P$

1. if $FC[f] == NULL$ then
   1.1 Algorithm 2
   1.2 New flow
2. else
   2.1 Algorithm 3
   2.2 Existing flow

Algorithm 2: is executed to create a new flow entry, i.e., when the received packet is the first packet of a flow. In order to limit the number of flows being monitored, three tests are applied to drop packets that are not likely to be part of elephant flows. First, packets that carry neither TCP segments nor UDP datagrams, e.g., ICMP packets, are dropped, as these flows are not likely to be elephants. Second, packets whose size (length) is shorter than a threshold $L$ are dropped. For TCP flows, the SYN segments are short packets, and hence will be dropped if $L$ is not 0. However, subsequent data packets are likely to have maximum-sized frames and therefore new flow entries will be created for TCP flows.

Algorithm 3: extracts the five tuples from the IP header of each received packet $P$ to form a flowID $f$. It then either creates a new flow entry (Algorithm 2) or updates an existing flow entry (Algorithm 3) in the flow cache $FC$.

For high-speed implementations, a hashing function is used for O(1) flow-cache lookup. Fig. 3a illustrates the hashing procedure. For flows that hash to the same bucket, a linear search is carried out in the list corresponding to that bucket. Each entry consists of the flowID key, and flow record value, which is a pointer to the location where the flow-entry fields, such as timestamps, byte counts, etc., are stored.

Missing in this algorithm is a packet sampling operation. In contrast, NetFlow/IPFIX implementations, whose function is to aggregate packet information into flow records, typically support packet sampling. Our algorithm for elephant flow identification is based on a comparison of the rate of a flow with an absolute threshold, and therefore, it is important to capture all packets of a monitored flow. To handle high packet arrival rates, other optimizations are built into the later steps of our algorithm to control the number of entries in the flow cache, but the system resources should be sized to allow the initial flow-cache lookup to occur at an unsampled rate.

Algorithm 2: is executed to create a new flow entry, i.e., when the received packet is the first packet of a flow. In order to limit the number of flows being monitored, three tests are applied to drop packets that are not likely to be part of elephant flows. First, packets that carry neither TCP segments nor UDP datagrams, e.g., ICMP packets, are dropped, as these flows are not likely to be elephants.

Second, packets whose size (length) is shorter than a threshold $L$ are dropped. For TCP flows, the SYN segments are short packets, and hence will be dropped if $L$ is not 0. However, subsequent data packets are likely to have maximum-sized frames and therefore new flow entries will be created for TCP flows. Traffic analysis, which is presented in a later section, shows that the number of flows to be managed in the flow cache decreases significantly when this operation is
Algorithm 2: Add new flow entry
1 if $P$ is not a TCP or UDP packet then
2 $\leftarrow$ drop $P$ and return;
3 $l \leftarrow$ length of packet $P$;
4 if $l < L$ then
5 $\leftarrow$ drop $P$ and return;
6 generate a random number $r$ from $X \sim U(0, 1)$;
7 if $r > S$ then
8 $\leftarrow$ drop $P$ and return;
9 if $FC$ is full then
10 $\leftarrow$ Evict a flow entry to create space
11 Initialize flow entry and add flowID to LRU list:
12 $FC[f].t \leftarrow t$; $FC[f].p \leftarrow 1$; $FC[f].b \leftarrow l$;
13 $FC[f].state \leftarrow$ monitored;
14 add flowID $f$ to LRU list; update LRU list front variable;
15 $FC[f].lru \leftarrow$ location in LRU list;

Algorithm 3: Update existing flow entry (check if elephant)
1 $\leftarrow$ Update fields of the flow entry for the arriving packet;
2 $FC[f].t \leftarrow t$; $FC[f].p \leftarrow FC[f].p + 1$; $FC[f].b \leftarrow FC[f].b + l$;
3 use $FC[f].lru$ location to update LRU list entry for $f$;
4 update LRU list front variable;
5 if $FC[f].state \leftarrow$ monitored then
6 $\leftarrow$ Determine flow duration and flow rate;
7 $\leftarrow FC[f].t - FC[f].t; R \leftarrow FC[f].b/D;
8 $\leftarrow$ Check if flow is an elephant;
9 if $D > T$ then
10 $\leftarrow$ Address of flow-cache entry;
11 delete LRU list entry that was stored in back variable;
12 update LRU list back variable;
13 if $A.state \leftarrow$ redirected then
14 send message to router to delete filter rule for flow $f$ in router;
15 message, with EF statistics, is also received by SMS;
16 delete flow entry at address $A$ of flow cache to free-up space;

Algorithm 4: Evict a flow entry
1 $A \leftarrow FC[f]$ from location stored in LRU-list back variable;
2 $\leftarrow$ Address of flow-cache entry;
3 delete LRU list entry that was stored in back variable;
4 update LRU list back variable;
5 if $A.state \leftarrow$ redirected then
6 send message to router to delete filter rule for flow $f$;
7 message, with EF statistics, is also received by SMS;
8 delete flow entry at address $A$ of flow cache to free-up space;

Algorithm 4: Evicts a flow entry from the flow cache thereby creating space to add a new flow entry. The LRU-list back variable has the location of the last node of the LRU list. Line 1 shows that from this location, the address $A$ of the flow-cache entry of the flow corresponding to the last element of the LRU list is determined. Lines 3-4 delete the last LRU-list node and update the LRU-list back variable. If the flow corresponding to the entry stored at $A$ in the flow cache was a redirected flow, a message is sent to delete the filter rule that was used for redirection, before the flow entry itself is deleted from the flow cache to free up space (lines 6 to 8). The message parameters include size (in bytes and packets), and

Algorithm 4: Evict a flow entry
1 $\leftarrow FC[f]$ from location stored in LRU-list back variable;
2 $\leftarrow$ Address of flow-cache entry;
3 delete LRU list entry that was stored in back variable;
4 update LRU list back variable;
5 if $A.state \leftarrow$ redirected then
6 send message to router to delete filter rule for flow $f$;
7 message, with EF statistics, is also received by SMS;
8 delete flow entry at address $A$ of flow cache to free-up space;

Algorithm 4: Evicts a flow entry from the flow cache thereby creating space to add a new flow entry. The LRU-list back variable has the location of the last node of the LRU list. Line 1 shows that from this location, the address $A$ of the flow-cache entry of the flow corresponding to the last element of the LRU list is determined. Lines 3-4 delete the last LRU-list node and update the LRU-list back variable. If the flow corresponding to the entry stored at $A$ in the flow cache was a redirected flow, a message is sent to delete the filter rule that was used for redirection, before the flow entry itself is deleted from the flow cache to free up space (lines 6 to 8). The message parameters include size (in bytes and packets), and router to set a firewall filter rule to redirect packets of this flow to the EF queue on the access link (lines 11-12). A filter rule can be set with the 5-tuple (source and destination IP addresses, source and destination port numbers, and protocol number) to only direct packets of a particular elephant flow to the EF queue. A router checks these rules for the special cases before looking up the default forwarding table.

If the flow has been monitored for a duration greater than the threshold $D$, but its rate is still below the rate threshold $R$, the flow is assumed to not be harmful, and hence removed from the cache (line 15). Lines 16-17 deal with the LRU-list updates required when a flow is removed from the flow cache. Implicit in Algorithm 3 is the fact that if the duration threshold corresponding to a flow has not been crossed when a packet arrives, the flow state is left unchanged as monitored.

Algorithm 4: Evicts a flow entry from the flow cache thereby creating space to add a new flow entry. The LRU-list back variable has the location of the last node of the LRU list. Line 1 shows that from this location, the address $A$ of the flow-cache entry of the flow corresponding to the last element of the LRU list is determined. Lines 3-4 delete the last LRU-list node and update the LRU-list back variable. If the flow corresponding to the entry stored at $A$ in the flow cache was a redirected flow, a message is sent to delete the filter rule that was used for redirection, before the flow entry itself is deleted from the flow cache to free up space (lines 6 to 8). The message parameters include size (in bytes and packets), and

Algorithm 4: Evicts a flow entry from the flow cache thereby creating space to add a new flow entry. The LRU-list back variable has the location of the last node of the LRU list. Line 1 shows that from this location, the address $A$ of the flow-cache entry of the flow corresponding to the last element of the LRU list is determined. Lines 3-4 delete the last LRU-list node and update the LRU-list back variable. If the flow corresponding to the entry stored at $A$ in the flow cache was a redirected flow, a message is sent to delete the filter rule that was used for redirection, before the flow entry itself is deleted from the flow cache to free up space (lines 6 to 8). The message parameters include size (in bytes and packets), and

Algorithm 4: Evicts a flow entry from the flow cache thereby creating space to add a new flow entry. The LRU-list back variable has the location of the last node of the LRU list. Line 1 shows that from this location, the address $A$ of the flow-cache entry of the flow corresponding to the last element of the LRU list is determined. Lines 3-4 delete the last LRU-list node and update the LRU-list back variable. If the flow corresponding to the entry stored at $A$ in the flow cache was a redirected flow, a message is sent to delete the filter rule that was used for redirection, before the flow entry itself is deleted from the flow cache to free up space (lines 6 to 8). The message parameters include size (in bytes and packets), and

Algorithm 4: Evicts a flow entry from the flow cache thereby creating space to add a new flow entry. The LRU-list back variable has the location of the last node of the LRU list. Line 1 shows that from this location, the address $A$ of the flow-cache entry of the flow corresponding to the last element of the LRU list is determined. Lines 3-4 delete the last LRU-list node and update the LRU-list back variable. If the flow corresponding to the entry stored at $A$ in the flow cache was a redirected flow, a message is sent to delete the filter rule that was used for redirection, before the flow entry itself is deleted from the flow cache to free up space (lines 6 to 8). The message parameters include size (in bytes and packets), and

Algorithm 4: Evicts a flow entry from the flow cache thereby creating space to add a new flow entry. The LRU-list back variable has the location of the last node of the LRU list. Line 1 shows that from this location, the address $A$ of the flow-cache entry of the flow corresponding to the last element of the LRU list is determined. Lines 3-4 delete the last LRU-list node and update the LRU-list back variable. If the flow corresponding to the entry stored at $A$ in the flow cache was a redirected flow, a message is sent to delete the filter rule that was used for redirection, before the flow entry itself is deleted from the flow cache to free up space (lines 6 to 8). The message parameters include size (in bytes and packets), and

Algorithm 4: Evicts a flow entry from the flow cache thereby creating space to add a new flow entry. The LRU-list back variable has the location of the last node of the LRU list. Line 1 shows that from this location, the address $A$ of the flow-cache entry of the flow corresponding to the last element of the LRU list is determined. Lines 3-4 delete the last LRU-list node and update the LRU-list back variable. If the flow corresponding to the entry stored at $A$ in the flow cache was a redirected flow, a message is sent to delete the filter rule that was used for redirection, before the flow entry itself is deleted from the flow cache to free up space (lines 6 to 8). The message parameters include size (in bytes and packets), and
first-packet and last-packet timestamps. As Fig. 2 illustrates, when the message is sent to the router (or SDN controller) for filter-rule deletion, it is also received by the IAOTE SMS for use in computation of the duration- and rate-thresholds required by Algorithm 3.

C. IOATE Service Management System (SMS)

The purpose of the SMS is to monitor the performance of the EFIE, and use the obtained information to determine appropriate values for the rate and duration thresholds used by EFIE. The SMS operation is based on the observation that if the rate of an elephant-flow exceeds the headroom on the link (capacity - background traffic rate) for a duration slightly longer than the time it takes to fill the packet buffer feeding the link, then packet losses are possible, i.e.,

\[
\int_{t_1}^{t_2} r_{bg}(t)dt + \int_{t_1}^{t_2} r_{e}(t)dt > C(t_2 - t_1) + B
\]  

(1)

where \(C\) is the link capacity, \(B\) is the buffer size, \(r_{bg}(t)\) is the background traffic rate, \(r_e(t)\) is an elephant-flow rate. The time interval \((t_1, t_2)\) is the time taken to fill the buffer. Datacenter switches often have small buffer sizes (e.g., 100 KB per 10 GigE port [26]). Our own experiments with GENI top-of-rack switches shows that buffer sizes are in the range (500 KB, 2MB) for 10 Gbps links.

Three types of input data are received by the SMS from the router and EFIE as illustrated in Fig. 2. These include: (i) Simple Network Management Protocol (SNMP) packet-discard statistics from the router, (ii) periodic short-duration packet captures from the port-mirrored trace sent from the router, and (iii) completed elephant-flow characteristics obtained from EFIE in the messages used to delete firewall filter rules.

The SMS algorithm starts by computing the background traffic rate per discrete time interval, which is set to \(B/C\). For example, with 100 KB buffer feeding a 10 Gb/s link, this time is 80 \(\mu\)s. It is impractical to read SNMP counters at this high rate. Typically, SNMP data collection is done every 10-30 sec [28]. Therefore, the background traffic rate is computed from the mirrored traffic sent to EFTE by the router. As this operation is compute-intensive, we propose to have the SMS execute this action only for short durations, e.g., 1 hour, at different times of day, and days of week, as needed. In other words, a sample of background traffic rate measurements is sufficient.

With an initial sample of background traffic rate measurements, the SMS can set a rate threshold. For example, \(R\) could be set to say 3 Gbps on a 10 Gbps link if the average background traffic rate is around 5 Gbps, which leaves 20% headroom for bursts. The duration threshold \(T\) is set to \(B/C\), i.e., the time taken to fill the access-link buffer.

Subsequent to this initial setting, the SMS can change both thresholds based on packet discard performance, and measurements of background traffic and completed elephant-flow statistics. If the packet-discard rates are high, then the rate threshold is dropped because even a lower-rate flow could cause packet losses when buffer sizes are small. But if such a reduction in rate threshold causes the number of elephant flows to increase significantly, then corrective actions are required. This is because there is a disadvantage to redirecting too many flows to the second queue since redirection can cause packets to arrive out-of-sequence. Out-of-sequence packet deliveries can trigger TCP fast retransmit/fast recovery procedure, which would result in a drop in TCP-flow throughput.

In summary, a dynamic method is used by the SMS to determine the rate and duration thresholds used by EFIE in order to ensure low packet-loss performance for all flows, while simultaneously limiting the number of flows subject to redirection. Control methods will be required to prevent instability.

IV. Traffic trace analysis

The Center for Applied Internet Data Analysis (CAIDA) [9] collects unsampled 1-hour packet traces once a quarter (every 3 months) on 10 Gbps links. Each 1-hour trace is saved in 61 files, with each file storing packet headers from roughly 1 Gbps worth of traffic (e.g., 100 Gbps worth of traffic per hour, per link). Each 1-minute file is approximately 2.4 GB.

We downloaded these traces and used them in experimental tests of our EFIE implementation as will be described in the next two sections. But in this section, we describe three types of analyses that we ran on these traces.

First, Section IV-A describes our analysis of a 1-min CAIDA trace to evaluate the efficiency of the hashing function used for flow-cache lookup. To determine an appropriate value for the rate threshold of the EFIE algorithms, we analyzed rates of flows in six 1-hour CAIDA traces. This analysis is presented in Section IV-B. Finally, Section IV-C describes an analysis of the same six 1-hour CAIDA traces to determine whether there is a performance benefit in using the length-based packet filter of Algorithm 2.

A. Evaluation of hashing algorithm

As described in Section III-B, for high-speed flow-cache lookups, a hashing function is used. Specifically, the Bernstein hash function [29] is used. An ideal hash function should map every key to a different bucket, and there should be no unused buckets. This is not always the case as illustrated in Fig. 3a, where multiple keys hash to the same bucket and there are unused buckets. It is therefore advisable to test a particular hash function with sample data. Specifically, a 1-min CAIDA packet trace was used (June 2014 trace collected at 13:00 hours UTC from the equinix-sanjose direction A monitor).

Fig. 4 shows the histogram of the number of flow entries per bucket after hashing 38M packets (1.8M flows). The number of buckets without a flow entry was 935593, and 772881 unused buckets. This is not always the case as illustrated in Fig. 3a, where multiple keys hash to the same bucket and there are unused buckets. It is therefore advisable to test a particular hash function with sample data. Specifically, a 1-min CAIDA packet trace was used (June 2014 trace collected at 13:00 hours UTC from the equinix-sanjose direction A monitor).

The Bernstein hash function performs well.
B. Flow-rate analysis

In this analysis, for each flow in the traffic trace, we computed the rate of the flow in each non-overlapping discrete time interval of duration $\mathcal{T}$ within its lifetime. This operation is effectively the same as running Algorithm 3 of EFIE without packet filtering or post-lookup packet sampling. For example, if a flow is 5-sec in duration, and $\mathcal{T}$ is set to 1 sec, then five rate values are computed for this flow. We use the term flow-window rate to describe this metric. The total number of time windows is $\sum_{f=1}^{F} \frac{D_f}{\mathcal{T}}$ where $F$ is the total number of flows in the trace, and $D_f$ is the duration of flow $f$ expressed as a multiple of $\mathcal{T}$.

Entries in Table II show the number of time windows in which the rate of some flow exceeded the rate threshold $\mathcal{R}$. A single flow could be counted multiple times in any of the entries in Table II since each flow has multiple flow-window rates, and more than one of these flow-window rates could exceed $\mathcal{R}$.

TABLE II: Number of times a flow-window rate $> \mathcal{R}$

<table>
<thead>
<tr>
<th>Rate threshold $\mathcal{R}$ (Mbps)</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 2014</td>
<td>4761</td>
<td>735</td>
<td>124</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>March 2014</td>
<td>1949</td>
<td>432</td>
<td>96</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>June 2013</td>
<td>1469</td>
<td>468</td>
<td>129</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>Feb 2013</td>
<td>2105</td>
<td>1099</td>
<td>88</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>July 2012</td>
<td>1639</td>
<td>395</td>
<td>97</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>March 2012</td>
<td>1276</td>
<td>324</td>
<td>97</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

Our analysis shows that the flow-window rate exceeded a rate threshold of 5% of the link capacity, i.e., 500 Mbps, a maximum of 21 times in the June 2013 trace, and there were no time windows in which any flow’s rate exceeded 10% of the link capacity (1 Gbps). We used 50 Mbps as the rate threshold $\mathcal{R}$ in our experimental testing of EFIE given the number of time windows observed with this setting.

C. Impact of packet-length based filtering

For EFTES performance optimization, Algorithm 2 includes three methods to drop packets as a way of limiting the number of entries in the flow cache. One of these methods was filtering out small packets, and initializing a flow-cache entry only for packets longer than the packet-length threshold, $\mathcal{L}$. This type of length-based packet filtering, if done without post-lookup packet sampling, will not affect flow-rate computation of potential elephant flows. This is because while the TCP SYN segment will be dropped, the first data packet (typically, maximum-sized) will be captured by EFIE, which will then create a flow-cache entry. Recall that once a flow-cache entry is created, all subsequent packets of the flow will be processed by EFIE. Thus, dropping the SYN segment will not hurt the flow-rate computation; to the contrary, the computed flow rate will be more accurate for high-RTT flows.
This section describes an analytical study of CAIDA packet traces to determine the potential savings in terms of computational effort required of EFIE when length-based packet filtering is executed. Two questions were asked: (i) what is the percentage reduction in the number of packets, and (ii) what is the percentage reduction in the number of flows? The question of whether the accuracy of elephant-flow identification is compromised with this filtering is addressed in Section V.

The analysis program implemented the relevant features of Algorithms 1 and 2, i.e., the program maintained a database with just flowIDs, and dropped a packet whose length was less than the threshold $L$ only if the corresponding flowID was not present in the database. However, this analysis program did not implement flow eviction, which means the reported percentage of packets dropped would be lower than in the real EFIE implementation. Inspite of this simplification, we found that significant savings in computational effort are possible with this type of filtering.

Specifically, six different one-hour packets traces collected by CAIDA were analyzed. All traces were collected by the same equinix-sanjose monitor on a 10 Gb/s link. Each trace consists of 61 one-minute packet capture files.

Fig. 5a shows the percentage of packets dropped for different values of the packet-length threshold $L$: 600B, 700B and 800B. The percentage of packets dropped was computed for each one-min file, and the statistics across the 61 one-min files collected in a quarter, e.g., March 2012, are plotted in the corresponding boxplot. With an 800B threshold, across all 366 (6 × 61) one-min files, we found that, in the worst case, the maximum percentage of packets left behind would have been 68.9% (this number was computed for a one-min trace from June 2013).

Fig. 5b shows the percentage of flows that would have been dropped for different values of the threshold $L$. With an 800B threshold, across all 366, one-min files, we found that, in the worst case, the percentage of flows left behind was 26.6% (this number was computed for a one-min trace from Feb 2013). This is a significant reduction in the number of flows handled.

In summary, packet length based filtering is a promising approach for EFIES performance optimization. The impact on accuracy of elephant-flow identification will be presented in the next section.

V. HIGH-SPEED EFIE IMPLEMENTATION AND EVALUATION

We implemented EFIE in a high-performance, multicore system called R-Scope [8]. The R-Scope system has two Intel Xeon E5-2670 processors running at 2.50 GHz for a total of 20 physical cores (without hyperthreading). It has four high-speed Solarflare SFC9100 optical network adapters, which support high-performance NIC features such as packet coalescence, receive-side scaling, kernel bypass, and polling-mode operation. A software layer called Dynamic Network Acceleration for many-Core (DNAC) interfaces with these high-speed NICs, and offers EFIE a set of library functions for efficient packet handling.

This EFIE implementation was tested with a 10-Gbps CAIDA packet trace as described below.

**Test setup:** A traffic-generator host and R-Scope were each connected with two 10-Gbps links to an Ethernet switch. Two instances of the Linux tcpreplay utility were executed concurrently on the traffic-generator host to play out a 1-min CAIDA packet trace (which had 38 M packets, 1.5 M flows, and an aggregate size of 28 GB) on each of the two interfaces. The outgoing traffic rate on each output interface of the traffic-generator host was around 2.5 Gbps. Therefore, to create a 10-Gbps incoming traffic trace at the R-Scope, we configured the switch to multicast all packets received from the traffic-generator host to both its ports that were connected to the R-Scope. This resulted in the R-Scope receiving 5 Gbps on each of its two ports, for a total of 10 Gbps.

The EFIE software was configured to use 4 cores to handle packets received on each port of R-Scope. Therefore, a total of 8 cores were used. The rate threshold $R$ was set to 50 Mbps, and the duration threshold $T$ was set to two different values: 100 ms and 1 sec. Four combinations of the packet filtering threshold, $L$, and sampling probability, $S$, were tested.

Two output metrics were collected: number of reported elephant flows, and percentage of dropped packets. The number of packets dropped is measured by a packet drop counter at each R-Scope interface, and reported out by DNAC. Packets are dropped when there is no space left in the DNAC queues that hold arriving packets while waiting to be read by the EFIE worker threads.

**Determine flow-cache size:** The cache size was varied from 1 K to 1 M flow entries in a series of experiments. For each setting, 20 runs were executed. In this experiment, the thresholds were set as follows: $T = 1s$, $L = 0$ B, and $S = 1$. In each run, the total number of packets received by R-Scope was about 138 M, and due to the high entropy of source and destination IP addresses in the trace, packets were evenly processed by the 8 cores.

Fig. 6 shows that the median number of dropped packets follows a U shaped curve with increasing flow-cache size. With a small flow cache, the rate of evictions can be high, and the extra compute cycles spent in flow-cache management lead to packet drops. A large flow cache may not fit in the...
processor cache, which then causes main-memory (RAM) operations. RAM access speeds are lower than compute and cache-access speeds, and therefore a larger percentage of packets are dropped if the flow cache is too large.

**Number of reported elephant flows:** From the set of sizes considered, the packet drop rate was smallest for a cache of size 50K. Interestingly, the variability (across the 20 runs) was also small for this size. We used this flow-cache size in the rest of the experiments. In general, processor-cache sizing is important when choosing EFTES hardware.

Fig. 7 shows statistics about the number of reported elephant flows across the 30 runs for each setting of the duration threshold, $T$, packet-filtering length threshold $L$, and post-lookup sampling probability $S$. The number of elephants reported for $T = 100$ ms was two orders of magnitude more than the number reported for $T = 1$ s. This is because for medium-to-high RTT-flows, there can be many silence periods within 1 sec, and therefore the observed rate in a 1-sec duration will not exceed the 50 Mbps rate threshold as often as in the 100-ms setting. The 100-ms duration was selected because WAN routers typically have buffers that can hold packets for 100-ms (e.g., 125 MB buffer serves a 10 Gbps link). This result shows that the duration threshold should ideally be set based on the switch buffer size and link capacity. The drawback of a small duration threshold will be considered in the percentage-of-dropped-packets metric.

But before moving to a discussion of the cost of using a small duration threshold, consider the impact of filtering and sampling. Fig. 7a shows that filtering without sampling ($L = 600B; S = 1$ setting) does not unduly affect the accuracy of elephant flow identification (the median is 2354 without filtering and 2116 with filtering alone). However, when sampling was introduced, $S = 0.001$, the median number of reported elephant flows dropped significantly to 919. This result appears to indicate that post-lookup packet sampling is not a good means to optimize EFIE performance. However, we caution that the rate threshold of 50 Mbps is fairly low, i.e., there are many flows that cross this threshold as will be seen in the analysis presented in the next section. With post-lookup sampling, such low-rate flows may not be captured. But if the
rate threshold was much higher, then post-lookup sampling is not likely to affect the number of reported elephant flows as much as in this low rate-threshold setting.

The average CAIDA traffic rate was about 3.5 Gbps, and the link capacity is 10 Gbps. Therefore, the rate threshold should have been set much higher since a 50-Mbps flow is not likely to cause packet drops. However, as our analysis in the next section will show, the CAIDA traffic trace did not have any real elephant flows, i.e., flows that could have caused buffer overflows. We therefore chose a low 50 Mbps rate threshold just to be able to have EFIE classify some flows as elephants.

**Percentage of dropped packets:** Fig. 8 shows the number of packets dropped by EFIE. The first key point to observe is that with 8 cores, the EFIE implementation is able to keep up with packets arriving at 10 Gbps. The maximum percentage of packets dropped is only 0.036% since the total number of packet is 38M. When the number of cores was increased to 10, in 6 out of 30 runs, there were no packet drops.

Next, we observe that the median number of packets dropped is only slightly higher (17K) in the 100-ms duration-threshold setting than in the 1-sec setting (13K). Since flow-eviction frequency is inversely proportional to $T$, EFIE needs to perform increased computation at the lower setting of $T$. However, it appears that with 8 cores, EFIE is able to keep up with 10 Gbps traffic even at the lower 100-ms duration-threshold setting.

Next, consider the effects of filtering and sampling. Fig. 8a shows that the median number of dropped packets falls by 83% with filtering. With filtering, there were no eviction events as most flow entries are removed at a frequency of 100 ms, which creates space in the flow cache. Without filtering, there were 2.13M evictions. Recall from Fig. 7a that the number of reported elephant flows was not significantly affected by filtering alone.

Therefore, we conclude that length-based packet filtering is a good solution to implement. Sampling offers a slight reduction in the number of packets dropped, but recall that its impact on the accuracy of the number of elephants reported was significant.

VI. Experimental studies of elephant flows

Two sets of experimental studies were conducted. First, we ran experiments to test the basic functionality of EFIE: identifying an elephant flow in real-time and redirecting it to an EF queue. This experiment demonstrates the value offered by EFTEs by comparing the impact of the elephant flow on a delay-sensitive flow, with and without redirection. These experiments are described in Section VI-A.

Next, Section VI-B describes a series of experiments that we undertook in two testbeds to study how various parameters of a potential elephant flow can determine whether or not the flow has adverse affects on other flows.

A. Illustration of the value offered by EFTEs

Fig. 9 shows the experimental setup used for testing the basic functionality and illustrating the value of EFTEs. The setup is a ProtoGENI [27] slice consisting of five bare-metal hosts (Host 1 to Host 5) located in the University of Kentucky testbed. All hosts have a single 4-core CPU (Intel Core 2 Quad CPU Q6600 @ 2.40GHz) and 8 GB of RAM. Each host has five 1 GigE Network Interface Cards (NICs). The eth0 NICs on hosts are used for remote login, while the other NICs are used in the data-plane of the experiment. Host 1 was used to send a ping flow, Host 2 was used to replay a real packet trace collected by the CAIDA to emulate background traffic, and Host 3 was used to generate an elephant flow with iperf3. All flows were destined to Host 4.

Host 5 was configured to serve as an IP router and forward packets between NICs 1 through 4 as illustrated in Fig. 9. In real usage, the EFIE would be executed in a host distinct from the router, in this experiment, the EFIE was executed on Host 5, as shown in Fig. 9. As part of EFTEs initialization, two output queues —a primary queue and an Elephant-Flow (EF) queue —were created for the output port NIC4 using the Linux traffic control (tc) utility. The tc configuration specifies rates for the two queues but allows for bandwidth borrowing between the queues, i.e., if one queue has no packets, the scheduler will send packets from the other queue. The default option was to send all packets to the primary queue.
TABLE III: Parameters used in different experimental runs, and observed packet losses

<table>
<thead>
<tr>
<th>Cases</th>
<th>TCP buffer</th>
<th>RTT_m (ms)</th>
<th>R_T (MB)</th>
<th>Packet loss</th>
<th>TCP buffer</th>
<th>RTT_m (ms)</th>
<th>R_T (MB)</th>
<th>Packet loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>160 KB</td>
<td>0.3</td>
<td>800 Mbps</td>
<td>no</td>
<td>1.2 MB</td>
<td>0.3</td>
<td>8 Gbps</td>
<td>yes</td>
</tr>
<tr>
<td>Case 2</td>
<td>2 MB</td>
<td>0.3</td>
<td>800 Mbps</td>
<td>yes</td>
<td>10 MB</td>
<td>0.3</td>
<td>8 Gbps</td>
<td>yes</td>
</tr>
<tr>
<td>Case 3</td>
<td>10 MB</td>
<td>0.3</td>
<td>100 Mbps</td>
<td>no</td>
<td>20 MB</td>
<td>0.3</td>
<td>1 Gbps</td>
<td>no</td>
</tr>
<tr>
<td>Case 4</td>
<td>40 MB</td>
<td>100</td>
<td>800 Mbps</td>
<td>yes</td>
<td>400 MB</td>
<td>100</td>
<td>8 Gbps</td>
<td>yes</td>
</tr>
<tr>
<td>Case 5</td>
<td>40 MB</td>
<td>100</td>
<td>100 Mbps</td>
<td>no</td>
<td>400 MB</td>
<td>100</td>
<td>1 Gbps</td>
<td>no</td>
</tr>
</tbody>
</table>

As shown in Fig. 9, libpcap is used to emulate the port mirroring operation described in Section III-A in which all packets sent out on the access link are copied to EFIE. In this experiment, NIC4 emulates the access link.

The EFIE rate threshold R is set to 100 Mbps and the duration threshold T is set to 1 sec. The length threshold L was set to 0, and post-lookup sampling probability S was set to 1 (which means no packets were dropped).

The CAIDA packet trace has only TCP and IP headers. The Ethernet header and payloads were stripped out before the packet trace was saved at CAIDA. We used a python packet manipulation module called scapy to add dummy payloads to each IP packet in the trace using the packet length specified in the IP header. In order to adapt the 10 Gbps CAIDA trace to our 1 Gbps experiment, packet inter-arrival times were increased by a factor of 10. We used tcprewrite to add source and destination MAC addresses to each packet, and the destination IP address of each packet was changed to the IP address of the eth1 interface of Host 4 so that the kernel-level IP packet-forwarding table of Host 5 (router emulator) required just one entry. This enabled the packet forwarding software to keep up with packets at 1 Gb/s speed. The tcpreplay tool was used to replay the modified CAIDA trace.

To emulate an elephant flow, a high-rate iperf3 HTCP flow was initiated on Host 3. Using tc, the sending rate on eth1 NIC was set to 800 Mbps.

With EFIE running, the ping flow was initiated, and the modified CAIDA packet trace was replayed. EFIE captured packets from all three flows and ran its algorithms. When the duration threshold (1 sec) was crossed by the iperf3 flow, the flow was identified by EFIE as an elephant flow. EFIE then invoked a script specifying the elephant flowID as an input parameter. The script used the tc utility to set a filter rule for subsequent packets of the elephant flow to be enqueued in the EF queue.

Fig. 10 shows the ping delay in two cases: with redirection of the elephant flow (in blue) and without redirection of the elephant flow (in red). Without redirection, the ping-flow packets experienced increased queuing delays (from 5 ms to over 13.8 ms) as the elephant flow and background traffic filled up the primary queue. The reason why ping-flow packets experienced lower delay (5 ms) soon after initially experiencing 13.8 ms is because the buffer filled up and the elephant flow experienced packet losses causing its sending rate to drop. But after the ping delay drops to 5 ms, it starts growing again as the elephant flow ramps up its sending rate.

In the run with elephant-flow redirection, the ping flow initially experienced an equally large delay (16.8 ms) because EFIE waits for 1 sec to determine whether a flow is an elephant before redirection. However, the effect of redirection can be seen in Fig. 10 shortly after 1 sec. As the elephant-flow packets were redirected to a separate EF queue, the ping flow delay dropped to an average of 2 ms. This method of online elephant-flow detection and traffic engineering prevents delay-sensitive flows (as exemplified by the ping flow) from experiencing high latency.

B. Impact of different parameters on elephant flow behavior

Two intra-rack experimental setups were used: (i) an InstaGENI setup, and (ii) Chameleon [30] setup. The topology was similar to the one shown in Fig. 9 except that a top-of-rack (TOR) Ethernet switch (instead of Host 5) connected the four hosts, Host 1 through Host 4. All links in the InstaGENI setup were 1 GigE, while in the Chameleon setup, all links were 10 GigE. Using UDP flows and a ping flow, we determined the switch buffer size to be about 2 MB in the InstaGENI TOR switch, and 5 MB in the Chameleon TOR switch.

A ping flow was sent from Host 1 to Host 4, the CAIDA trace, used as background traffic, was replayed from Host 2 to Host 4, and an iperf3-HTCP flow, emulating an elephant, was sent from Host 3 to Host 4. On the InstaGENI setup, the modified CAIDA 1 Gbps trace was replayed with the multiplier factor set to 1. On the Chameleon Cloud setup, the CAIDA trace was replayed without a rate change. The round-trip time (RTT) between Host 3 and Host 4 on InstaGENI and Chameleon Cloud were 0.348 ms and 0.237 ms, respectively. Two key parameters on the path of the HTCP flow, bottleneck link rate and RTT, were controlled using tc and netem, respectively. One other parameter, TCP buffer size (the same value was used for send- and receive-side buffers), was controlled for the HTCP flow. Output metrics, collected from iperf3 logs, include (i) per-sec HTCP-flow throughput, (ii) per-sec packet retransmissions, and (iii) per-sec congestion window size.

The five experimental cases that were executed are described in Table III. A 1-min CAIDA trace was used in each experiment, and the HTCP flow was started at a random time after the CAIDA-trace replay was initiated.

Low-RTT cases: In the first three cases, no additional delay was added to the HTCP-flow path, and therefore RTT was approximately 0.3 ms. Two values of bottleneck link rate were emulated using tc at Host 3.

Consider the difference between cases 1 and 2 in the InstaGENI experiments. The TCP buffer size in case 1 was smaller than the switch buffer size (which was 2 MB), while in case 2, the TCP buffer size was matched to the switch
(a) Case 1: Small TCP buffer; InstaGENI; EF rate: 800 Mbps

(b) Case 1: Small TCP buffer; Chameleon; EF rate: 8 Gbps

c) Case 2: Large TCP buffer; InstaGENI; EF rate: 800 Mbps

d) Case 2: Large TCP buffer; Chameleon; EF rate: 8 Gbps

Fig. 11: Plots from iperf3 logs for low-RTT path experiments (cases 1-2)
Fig. 12: Plots from iperf3 logs for high-RTT path experiments (case 4)

(a) InstaGENI; Large TCP buffer, EF rate: 800 Mbps

(b) Chameleon; Large TCP buffer; EF rate: 8 Gbps

Fig. 13: Case 5; InstaGENI; EF rate: 100 Mbps

when the TCP buffer size is larger than the switch buffer size, packet losses occur more easily since a whole cwnd worth of data could be sent in a burst by the HTCP sender. Results for the InstaGENI and Chameleon experiments are shown in Figs. 12a and 12b, respectively. In case 5, since the path is rate limited to a low value (100 Mbps in InstaGENI and 1 Gbps in Chameleon), even though the TCP buffer size is large, cwnd does not grow larger than the switch buffer size and there are no packet losses, as seen in Fig. 13.

In summary, SNMP-reported packet losses should be used to adjust the rate threshold because on high-RTT, high-rate paths, users would have likely set large TCP buffer sizes to achieve high throughput, and these buffer sizes are likely to exceed switch buffer sizes. Second, the background traffic rate should be measured, and the elephant-flow rate threshold should be correspondingly adjusted to leave sufficient headroom in link usage.

VII. SUMMARY AND CONCLUSIONS

Access links between customer and provider networks are typically under-provisioned. This paper proposed a new service architecture that uses online traffic engineering of elephant flows to enable the customer to receive improved service without having to purchase a higher-rate access link. This new access-link service is supported by a system called Elephant Flow Traffic Engineering System (EFTES). EFTES implements online elephant-flow identification algorithms, and a module for setting and deleting firewall filter rules in routers for redirected identified elephants flows to a separate queue. EFTES was implemented on a high-performance multi-core system called R-Scope. Using real traffic traces collected by CAIDA on a 10-Gb/s link, the performance of EFTES was evaluated. With 8 cores, EFTES could handle 10 Gb/s real packet traces. A performance optimization of EFTES, which consisted of dropping small-sized packets, was recommended, and shown through analysis to offer significant computational savings (e.g., an analysis of 366 1-min traces showed that in the worst case, only 26.6% of flows would remain if length-based packet filtering was executed) with only a 10% drop in the accuracy of reported elephant flows.

VIII. ACKNOWLEDGMENT

This work was supported by UVA NSF grants ACI-1340910, CNS-1405171, and CNS-1531065, and US DOE grant DE-SC0011358. We are grateful to CAIDA for providing us the high-speed traffic traces used extensively in this work. We also thank Xiaoyu Wang, University of Virginia, for her help with procuring and analyzing CAIDA datasets.

REFERENCES


Sourav Maji Sourav Maji is pursuing his PhD in Computer Engineering from University of Virginia since 2014. During this time, he has worked on the HNTES project with ESNet and rHNTES project with Reservoir Labs. Prior to that he worked as a Senior Software Developer in Ericsson Research and Development in Bangalore. Sourav has a MS in Communication Systems from National Institute of Technology Warangal, India.

Malathi Veeraraghavan Malathi Veeraraghavan is a Professor in the Charles L. Brown Department of Electrical and Computer Engineering, and an Associate Director in the Data Science Institute, at the University of Virginia. After a ten-year career at Bell Laboratories, she served on the faculty at Polytechnic University, Brooklyn, New York from 1999-2002. She served as Director of the Computer Engineering Program at UVa from 2003-2006. She holds thirty US patents, has over 130 publications, and has received six Best-paper awards. Her research projects are funded by grants from the National Science Foundation and the US Department of Energy. She has served on organizing and technical program committees of IEEE Communications Society and IEEE Computer Society conferences in various roles.

Molly Buchanan Molly Buchanan received her B.A, in Computer Science from the University of Virginia in 2016. Among her interests are computer science education as well as wireless computing and communication.

Fatma Alali Fatma Alali received her B.S. in Computer Engineering from Kuwait University in 2011, her M.S. in Computer Engineering from University of Virginia in 2015. Currently, she is a Ph.D student at University of Virginia. Among her research interest are Software Defined Networks, enabling Virtual Circuits (VC) services and InfiniBand networks.

Jordi Ros-Giralt Jordi Ros-Giralt received his B.S. in Electrical Engineering from BarcelonaTech, in Barcelona, his Ph.D. in Computer Science from the University of California, Irvine, and his MBA from the University of California Los Angeles. He was a recipient of the Balsells Fellowship and his research area includes Internet traffic engineering, high-performance computing and cyber security. Ros-Giralt is the author of more than 40 papers in communication journals and conferences and of several patents in the area of IP mobility, communications and high-performance computing.

Alan Commike Alan Commike, Managing Engineer, received his B.S. and M.S. in Computer Science from University at Buffalo in 1989 and 1991, respectively. Alans career has spanned roles from a company founder, to CTO, Director of Engineering, and PI. Alans early career concentrated in 3D graphics technologies at SGI. He later applied this background to his work at Quantum3D for the DARPA STAP-BOY program and his own company, QED Labs, under contract with AFRL. Alan leads the cyber security effort at Reservoir Labs.