Real-time Detection, Isolation and Monitoring of Elephant Flows using Commodity SDN System

Sharat Chandra Madanapalli*, Minzhao Lyu†, Himal Kumar†, Hassan Habibi Gharakheili†, and Vijay Sivaraman†
*BITS Pilani, India, †UNSW Sydney, Australia
Emails: f2014108@pilani.bits-pilani.ac.in, {m.lyu, himal.kumar}@student.unsw.edu.au, {h.habibi, vijay}@unsw.edu.au

***Abstract*—Operators of enterprise and carrier networks increasingly require real-time visibility into traffic patterns in their network, so they can do better resource management (congestion detection, dynamic routing, capacity scheduling) and security protection (detection of intrusions and volumetric attacks). Of particular interest are elephant flows that transfer large volumes, since they demand most resources and can inflict most damage. Today's techniques for detecting and monitoring elephant flows are based on software-based packet analysis or hardware-based inspection, which are either unscalable or expensive. In this paper we design, implement, and evaluate an SDN-based solution that is scalable (to tens of Gigabits-per-second) and inexpensive (built using commodity OpenFlow switches). We first develop a system architecture that judiciously combines software packet inspection with hardware flow-table counters to identify and monitor heavy flows. We then use real traffic traces taken from a campus network to tune our algorithm parameters for desired trade-off between software load and hardware table size. Finally, we prototype our solution on a commodity OpenFlow hardware switch together with open-source controller and packet inspection software, and demonstrate operation at 10Gbps in a real campus network.**

I. Introduction

Elephant flows typically constitute only a small fraction (5-10%) of flows by number, but account for a vast majority (60-80%) of traffic by volume [1]. Detecting and isolating elephant flows can therefore help an operator in a variety of ways: they can bandwidth-limit elephant flows, or dynamically route them along alternate paths, in order to protect performance for short flows; they can classify traffic by application (e.g. video versus peer-to-peer) for better capacity planning and provisioning; they can bypass elephant transfers across middle-boxes (like firewalls) so as to reduce the load on these expensive appliances; and they can log such transfers for forensic purposes in case security breaches or copyright violations need to be investigated. Indeed, several carriers and enterprises we have talked to have expressed a strong interest in some or all of the capabilities mentioned above.

Existing solutions for detecting and isolating elephant flows are software-based, and hence unscalable to high rates, or hardware-based, and hence prohibitively expensive. Some operators use NetFlow to periodically obtain aggregated IP flow information from switches – this not only requires hardware capability to decode, collate and export flow entries, which entails a cost-premium, but also imposes a penalty in switch CPU utilization in the range of 7-22% [2]. Statistical sampling of traffic using sFlow reduces this overhead, but inevitably leads to reduced accuracy. Moreover, NetFlow and sFlow can only passively monitor elephant flows, they cannot actively isolate or control them. Special-purpose hardware appliances, often marketed as "deep packet inspection" (DPI) engines (e.g. from Sandvine and Procera) are able to inspect, isolate, and control elephant flows, but come at prohibitive cost, often running into the hundreds of thousands of dollars for 10 Gbps operation.

978-1-5386-3416-5/18/$31.00 © 2018 IEEE

In this paper we explore the use of the Software Defined Networking (SDN) paradigm for identifying, isolating, and monitoring elephant flows. The use of Openflow, which allows match-action rules at the flow-level, seems by its nature ideally suited to provide flow-level visibility and control in a low-cost and scalable manner. Indeed, prior works such as [3]–[6] have used SDN (OpenFlow and/or P4) for elephant flow monitoring, and as we will discuss in §II, they differ from our work in their approach and trade-offs decisions. While the use of per-flow rules in OpenFlow may seem conceptually trivial, there are significant challenges to be overcome when trying to implement these at high rates: flow-tables in switch hardware are limited in size, and the switch agents cannot handle a large number of interactions (additions or stats collections) with flow-table entries. It therefore becomes necessary to limit the number of hardware flow-table entries and the flow-mod rate, by absorbing some of the load in software. Further, this has to be done without overloading the controller, ensuring soft resilience in the case of controller failures.

In this paper we design, implement, and evaluate an OpenFlow-based method to detect, isolate, and monitor elephant flows. Our method dynamically balances the load between software and hardware so that desired performance is achieved within the constraints of hardware table size and update rate. Our first contribution is to develop a system architecture that judiciously partitions load between software-based packet inspection and hardware-based flow-forwarding, without imposing load on the controller. For our second contribution, we conduct an offline evaluation of our scheme using trace data from a campus network, and show how the critical parameters can be tuned to achieve desired performance trade-off. For our final contribution we implement our scheme using a commodity OpenFlow switch and open-source software, and demonstrate its ability to identify, isolate, and monitor elephant flows at 10 Gbps in a live campus network.

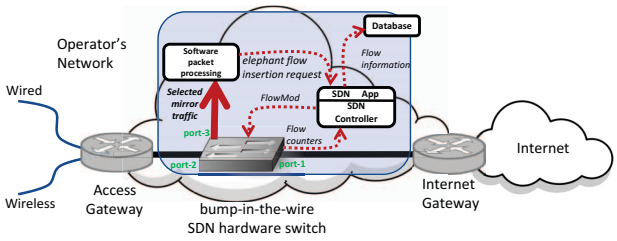


Fig. 1. Architecture.

II. Related Work

Detection and monitoring of elephant flows have been investigated by many researchers in legacy networks, SDNs and the upcoming custom dataplane based solutions. In [1], authors proposed a Bayes' theorem based framework to find the most efficient threshold towards number of per flow packets in elephant flow detection. Empowered by machine learning classification models, network traffic carrying elephant flows can be identified with about 90% accuracy using sampled packets [7]. NetFlow and sFlow are popular choices which collect IP-based flow-level statistics using packet sampling on legacy network devices. Approaches using NetFlow introduce significant overheads towards CPU utilization on hardware switch and may compromise network performance [8]. Also, adopting sFlow-based solutions affects accuracy of elephant flow detection due to low sampling rate resulting in delayed detection and/or missing an elephant flow.

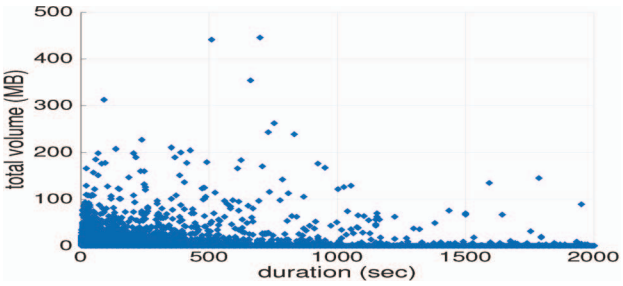
There are several solutions which leverage the flexibility offered by SDN. However, Openflow agent in commodity SDN suffers when a large number of flow-level statistics are queried from the switch [9]. Many approaches have been taken to address the issue of heavy periodic polling for elephant flow detection ranging from dynamic change of thresholds [4] to hierarchical approach [10]. Customizing the data plane seems to help alleviate the scalability issues in Openflow agent. DevoFlow [3] employs packet sampling, threshold-based trigger and approximate counter to reduce polling overhead when monitoring elephant flows. HashPipe [5] uses P4 to implement and maintain heavy-hitters' flow-level information entirely on data plane with a low rate of false detection when enough switch memory is available. However, modifying data plane is not completely feasible in commodity SDN-enabled networks. Also, the system's capability is constrained by limited memory size of hardware switches. Therfore, we propose a solution to detect and isolate elephant flows on commodity SDN switch without the need for customizing data plane, and minimize polling overhead by only monitoring isolated heavy-hitters.

III. System Design and Architecture

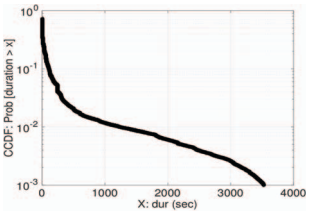
In this section we describe our SDN-based architecture, including the major architectural decisions, the functional blocks and our Openflow table structure.

A. Architectural Decisions and Functional Blocks

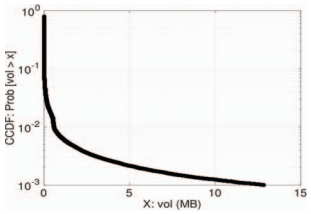
Our solution is designed to be a "bump-in-the-wire" on the link at which active monitoring of elephant flows is desired. Our system is therefore transparent to the network, and does not modify packets in any way. Further, our SDN switch does not send any data packets to the controller; instead, any packets that need to be inspected in software are sent as copies to a separate interface of the switch, to which a software inspection engine is attached. This protects the controller from overload from the data-plane, allowing it to service other SDN applications. Fig. 1 shows the functional blocks in our architecture applied to a typical carrier or enterprise network. The operational flow of events is as follows: assume that traffic enters (from the content provider) on port-1 and exits (towards the users) on port-2; the switch is initially configured to mirror all traffic to the packet processing software on port-3.



(a) Duration versus volume of flows.



(b) CCDF of flow duration.



(c) CCDF of flow volume.

Fig. 2. Distribution of flows during peak hour.

Our second architectural decision is in the judicious combination of flexible software for packet-header processing and scalable hardware for flow-counter monitoring. The software tracks all the flows uniquely identified by 5-tuple. It maintains its internal data structures that contain the duration and volume of individual flows. Using a built-in event mechanism, the software is able to detect an elephant flow, and makes a request to the controller (through a REST API exposed by an SDN application running on top of the controller) inserting a reactive rule to isolate the elephant flow. The elephant flow isolation serves two purposes. Firstly, the elephant flow is no longer mirrored to the software and hence reduces the load on the software. Secondly, the specific flow-entry enables fine-grained monitoring of the isolated elephant flow.

Upon insertion of a reactive flow-entry, the SDN App starts polling its counter periodically. The application dynamically adjusts the polling frequency based on the number of existing entries on the switch as well as flow-mod rate on the OpenFlow agent. The collected counters are written into a time-series database from which any other application can read and perform desired operations such as traffic classification, rate control, anomaly detection, etc.

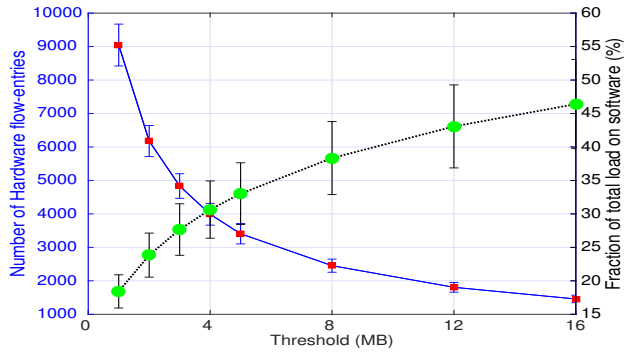
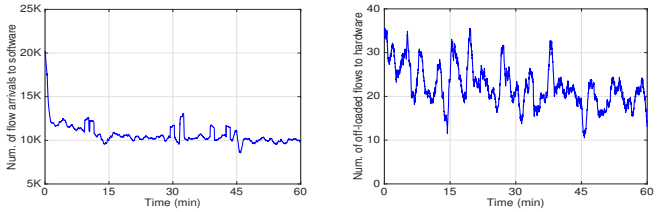


Fig. 3. Evaluation of elephant flow detection using various volume threshold.



(a) Flow arrival (software). (b) Elephant-flow arrival (hardware).

Fig. 4. Flow arrival

B. OpenFlow Table Pipeline

The match and action paradigm of SDN is used to dynamically isolate elephant flows once they are detected by the packet processing software. The Openflow pipeline on the SDN switch consists of two tables. Table-0 consists of the reactive rules which correspond to the detected elephant flows. Reactive rules have an inactivity timeout of 60 sec to minimize the active number of entries on the hardware. The action corresponding to these rules is to forward the packets without mirroring them. These reactive flow rules also enable flow level telemetry to be carried out by the use of OpenFlowStatsRequest messages on the OF-Channel. Table-0 is initialized with only one flow rule – packets which miss this table are automatically processed by rules in Table-1.

Our Table-1 contains only two rules that forward and mirror bidirectional traffic. Note that proactive rules ensure that no `PACKET_IN` message is generated, reducing load on the Openflow agent of the switch as well as the SDN controller. The flow insertion happens when events occur in software.

IV. Elephant-Flow Detection Method

We aim to develop an efficient method which is able to detect elephant flows quickly and accurately after their commencement while keeping the load on the software tractable. To perform such a detection, the software packet analyzer can employ thresholds on flow volume and/or flow duration, which need to be judiciously tuned to achieve our objective. We use several packet traces obtained from a fraction of our University campus network during peak hours (i.e. 2-5pm), each one-hour trace comprising of over 500 million packets from over 4 million flows. We believe that these uniformly sampled short traces still represent the composition of flows in our campus traffic. We describe the characteristics of trace data, then quantify the benefits from our solution.

A. Trace Data

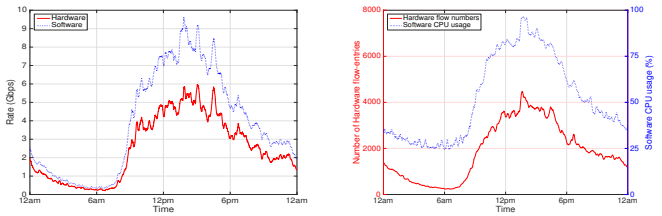
We collected a `pcap` trace data during peak hours from the university wired network serving thousands of students and staff – an ethics clearance was obtained (UNSW Human Research Ethics Advisory Panel approval number HC16712) in order to conduct this trial. We then log the time-stamp, header (i.e 5-tuple), and length of individual packets in a file, and use the log file to compute flow-level statistics.

We start examining the properties of flows in campus traffic traces. Fig. 2 depicts the scatter plot of flow duration versus flow volume, followed by distributions of flow duration and flow volume. In Fig. 2(a), it can be seen that a large fraction of flows are short in duration and small in volume – centered around the origin. We also observe that the duration of mice flows (i.e. those with small volume) spans the entire range of [0, 2000] seconds in Fig. 2(a) – data points are scattered across and close to the x-axis suggesting that the flow duration can not be indicative of the flow type either elephant or mice. Note that 99% of flows have duration less than 1000 seconds as shown in Fig. 2(b). However, the CCDF plot of flow volume in Fig. 2(c) shows that only 0.26% of all flows have volume greater than 4 MB, suggesting potential elephant flows. This small fraction of flows collectively contribute to 72% of total volume of all flows seen in our dataset.

B. Simulation Methodology and Results

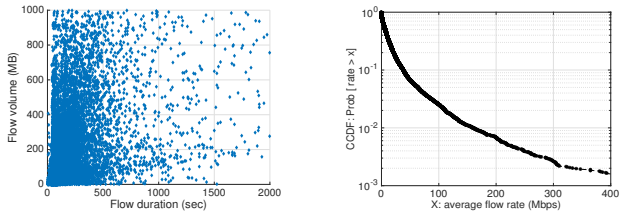
We wrote a native simulation that takes packet arrivals, header, and size from the trace as input, and performs software service (maintaining flow-level states, detecting elephant flows, and offloading to hardware) and hardware service (maintaining flow-table entries, updating per-flow counters and aging-out inactive flows). Both software and hardware modules maintain their internal states using separate data structures. For each run, a threshold of flow volume is passed to the simulation which is used to offload elephant flows from the software module to the hardware module.

The number of flow-mod, hardware flow-entries, and the load on the software is tracked in our simulation of each one-hour trace data. Fig. 3 shows the evaluation results of our simulation. Each data point shows the average value of corresponding metric computed from a number of runs with an error bar representing the standard deviation value. Unsurprisingly, as volume threshold increases it results in less flows being pushed into the hardware (as shown by solid blue lines and left y-axis) and more load is offered to the software (as shown by dotted black lines and right y-axis). For example, increasing the threshold from 1 MB to 4 MB, the average number of flows on the hardware is reduced by 55% (i.e. from 9000 to 4000), in exchange, the fraction of software load is increased by 66% (i.e. from 18% to 30%). Since monitoring of reactive flow-entries (i.e. counter collection request) causes the SDN switch to return a multi-part reply (e.g. our Noviflow switch returns usage statistics of 25 flows per each reply), this imposes a significant delay to our SDN App to collect the per-flow counter of all reactive entries. We therefore choose to operate our system that maintains on average 4000 flow-



(a) Hardware load vs. Software load. (b) Hardware flow-entries vs. Software CPU usage.

Fig. 5. Evaluation results of our prototype.



(a) Duration versus volume of flows (b) CCDF of flow rate.

Fig. 6. Distribution of isolated flows.

entries on hardware which ensures a sustained polling period of 5 seconds – this means that a 4 MB threshold would suit our traffic and network setup. Further, since the average size of a web-page as of 2017 is about 2.5 MB [11] the chosen 4 MB threshold would avoid isolating mice flows corresponding to HTTP-based web traffic. Fig. 4 depicts the rate of new flow arrival to software and hardware over one-hour period in our simulation. We observe that all traffic flows are captured by software at bootstrap thus a large number of flows per second (more than 18000) are detected as new arrival (including ongoing flows), this is stabilized to 12000 flow arrivals per second in 90 seconds, and reaches to 10000 arrivals per second on average after 15 minutes. Of this large number of new flow arrivals to software, only 25 flows per second on average arrive to hardware (i.e. detected elephant flows).

Note that among reactive flow-entires inserted into the hardware, we find a fraction of flows that do not exhibit elephant behavior. We deem hardware flow-entries with average rate of greater than 300 Kbps (equivalent of rate required for the lowest resolution video streaming [12]) as "actual" elephant flows. We found that that the fraction of actual elephant flows rises as the threshold increases and is stabilized at 5 MB threshold. Therefore, we propose to consider a threshold of flow duration in addition to primary threshold of flow volume, enhancing the accuracy of elephant flow detection. We explored the impact of the duration threshold for given thresholds of 4 MB on our metrics of software load and number of hardware flow-entries (the plot is omitted due to space constraints). We therefore perform a duration check in the software module once the flow volume hits the chosen threshold. Our results show that offloading flows with the volume of 4 MB and the duration of at most 300 seconds would significantly reduce the number of hardware flow-entries (i.e. 10%) while incurring a negligible increase in software load (i.e. 2.5%). We emphasize that these threshold values should be chosen by the network operator based on profiles of traffic, constraints of hardware, and resources available to the software engine.

V. Prototype Implementation

We have implemented a fully functional prototype of our system that uses our proposed solution. Our system includes NoviSwitch 2116 hardware, Ryu SDN controller, NetBricks [13] as packet processing software, and InfluxDB. Ryu controller and InfluxDB are run in separate virtual machines configured by 4-core CPU and 32 GB RAM on an enterprise standard server. A physical server with 32GB RAM and 16-core CPU that runs NetBricks framework over Data Plane Development Kit (DPDK) [14] to perform software packet processing of the mirrored campus traffic.

Fig. 5 depicts the evaluation results of our system prototype for a period of 24 hours. The hardware load is shown by solid red line in Fig. 5(a) and the software load is stacked on top of it, shown by dotted blue line. As we can see the total load (sum of software and hardware) of our campus traffic peaks during afternoon hours exceeding 9 Gbps. We note that only one third of total load on average is handled by the software which slightly grows during peak hours reaching to 43%. Considering performance constraints, we track the number of hardware flow-entries and profile the average CPU usage of software in Fig. 5(b) – the average CPU usage is always below 100% and only exceeds 90% during the peak hour (a negligible packet loss). It is also seen that the number of flow-entries is capped to 4000 except for few minutes – this results in dynamic reduction of the frequency of polling per-flow counters during that period from our SDN App.

We now look at the elephant flows of our campus traffic that are detected by software and isolated by hardware table entries within 24 hours. Our SDN App periodically monitors counters of individual reactive flow-entries from the hardware and writes into a table of InfluxDB. Fig. 6 depicts the analysis of reactive flow-entries present on the hardware. Fig. 6(a) is visually insightful as data points in the scatter plot of duration versus volume are grouped along the y-axis which is completely orthogonal to the observation we had earlier in Fig. 2(a) from all traffic flows – thus representing elephant flows. Considering the flow average rate in Fig. 6(b), more than 90% of reactive hardware entries indeed represent elephant flows, those that operate at the rate of more than 300 Kbps – this corroborates with our simulation results with enhancement due to additional duration check.

VI. Conclusion

Real-time visibility into elephant flows is increasingly becoming of high interest for network operators due to their resource consumption. Existing tools of monitoring of elephant flows are either costly hardware-based inspectors, or unscalable software-based analyzer. In this paper, we have proposed our solution that combines the scalability of commodity SDN hardware with the flexibility of packet processing software. We have developed our method to detect elephant flows, tuned our algorithm parameters using real traffic traces obtained from our campus network, and implemented a prototype and evaluated our solution operating at 10 Gbps in a real campus network.

REFERENCES

[1] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying elephant flows through periodically sampled packets," in *Proc. ACM IMC*, Taormina, Sicily, Italy, 2004.

[2] "Network performance analysis," Cisco Systems, Tech. Rep., 2005.

[3] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM*, Toronto, Ontario, Canada, 2011.

[4] Z. Liu, D. Gao, Y. Liu, H. Zhang, and C. H. Foh, "An adaptive approach for elephant flow detection with the rapidly changing traffic in data center network," *International Journal of Network Management*, pp. e1987–n/a, e1987 nem.1987. [Online]. Available: http://dx.doi.org/10.1002/nem.1987

[5] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 164–176.

[6] Y. Afek, A. Bremler-Barr, S. Landau Feibish, and L. Schiff, "Sampling and large flow detection in sdn," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 345–346.

[7] D. Rossi and S. Valenti, "Fine-grained traffic classification with netflow data," in *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, ser. IWCMC '10. New York, NY, USA: ACM, 2010, pp. 479–483.

[8] "Network performance analysis," Cisco Systems, Tech. Rep., 2005.

[9] K. Giotis, C. Argyropoulos, G. Androulidakis, D. Kalogeras, and V. Maglaris, "Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on sdn environments," *Comput. Netw.*, vol. 62, pp. 122–136, Apr. 2014.

[10] L. Yang, B. Ng, and W. K. Seah, "Heavy hitter detection and identification in software defined networking," in *Computer Communication and Networks (ICCCN), 2016 25th International Conference on*. IEEE, 2016, pp. 1–10.

[11] T. Howe. (2017) Is web page bloat costing your site? https://www.hallme.com/blog/the-cost-of-site-bandwidth/.

[12] Youtube, "Live encoder settings, bitrates, and resolutions," https://support.google.com/youtube/answer/2853702?hl=en, 2017.

[13] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of NFV," in *Proc. USENIX OSDI*, Savannah, GA, USA, 2016.

[14] D. Intel, "Data plane development kit," http://dpdk.org/, 2015.