Heavy Hitter Detection on Multi-Pipeline Switches

Fábio Luciano Verdi
verdi@ufscar.br
UFSCar and KTH Royal Institute of Technology
Stockholm, Sweden

Marco Chiesa
mchiesa@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

ABSTRACT

Recently, several applications have been designed and implemented to run entirely in the dataplane. However, most if not all the applications assume that network traffic traverses the same pipe, from ingress to egress inside the switch. While this seems to be a natural assumption, it does not hold for current programmable hardware that supports two to four pipes and network traffic is spread among the different pipes. As a consequence, several applications may not work properly in a multi-pipe architecture and need to be redesigned to fit into such architectural constraint. In this paper, we call the attention to this challenge and elaborate on an initial solution for counting heavy hitters (HH) in a multi-pipe hardware (MPHH). Our solution keeps the HH counter only in the egress pipeline while temporarily caching the hashes at the ingress pipeline. We then carry the hashes from ingress to egress by using data packets so that the HH are counted only in the egress pipeline. We present our design around this issue, the challenges observed so far and some initial results.

CCS CONCEPTS

• **Networks** → **Network measurement**; **Programmable networks**; **Network monitoring**.

KEYWORDS

Network monitoring, Programmable networks, Multi-pipelines, Heavy hitter detection.

ACM Reference Format:
Fábio Luciano Verdi and Marco Chiesa. 2021. Heavy Hitter Detection on Multi-Pipeline Switches. In *Symposium on Architectures for Networking and Communications Systems (ANCS '21), December 13–16, 2021, Layfette, IN, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3493425.3502760

1 INTRODUCTION

Several applications are now designed and implemented to run entirely (or partially) inside a switch so that monitoring and actuation can be performed at line-rate. These applications range from load balancers [7, 12, 23], congestion control [11, 16], heavy hitters detection [1, 14, 17, 19], and in-network caching [10, 15] to DDoS defense [13], fast rerouting [2, 6] and machine learning aggregation [18, 22].

However, when such applications are deployed into a programmable switch they need to face the constraints found in the hardware such as limited SRAM and TCAM, number of stages, number of registers, and more. Also and of great importance, programmable switches are designed to have more than one pipe to increase the processing capacity. Typically, current switches support two and four pipes and the number of physical ports in each pipe depends on the number of total physical ports in the switch. As an example, a 64-port switch with 4 pipes will assign 16 ports per pipe [9].

The multi-parallel pipes architecture is beneficial for high packet processing and to diminish race conditions when accessing the hardware resources. In a multi-pipe device, everything is local to the pipe including the registers, counters, and P4 tables. Typically, there is an instance of the same synthesized P4 program running in each pipe having pipe-local counters and registers [4]. A given register in one pipe is not seen by any other pipe which is exactly what the silicon designers needs to keep the design simple enough for supporting high throughput. However, this constraint may become cumbersome for some applications running in the switch.

Even information about single flows may be spread onto multiple pipes. This may be due to failures, load balancing, and traffic engineering [5, 21]. A given flow may start in one pipe and during its life-cycle it may move to others. Packet spraying [3, 8] and flowlets [20] have been used to load balance while fast-rerouting has been adopted for failure recovery. All of them are examples of scenarios where the network traffic may change from one pipe to another and affect dataplane monitoring applications running inside the switches.

In this paper, we elaborate around this multi-pipe constraint to design and implement a data structure that can support arbitrary Heavy Hitter (HH) detection applications on a multi-pipe hardware architecture. We call our data structure multi-pipe HH (MPHH). Our central idea is to choose exactly one pipe where counters (or sketches) for a specific flow will be stored. The HH sketch is then installed only at the egress pipeline while keeping a small cache in the ingress pipeline to temporarily store the incoming hashed packets. When a data packet enters in the ingress pipeline and leaves the switch through the pipe where the sketch is located, such a data packet carries hashed keys from the ingress to the egress, which in turn counts the packets in the sketch. The HH sketch in the egress pipeline can be any existing HH sketch.

2 MOTIVATION

Figure 1 depicts a simplified view regarding the spreading of network traffic among the pipes and how this can affect applications running in the switch.

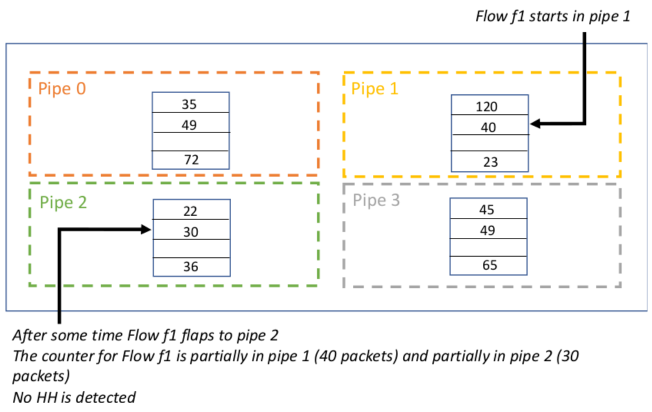

Flow f1 starts in pipe 1
Pipe 0
35
49
72
Pipe 1
120
40
23
Pipe 2
22
30
36
Pipe 3
45
49
65
After some time Flow f1 flaps to pipe 2
The counter for Flow f1 is partially in pipe 1 (40 packets) and partially in pipe 2 (30 packets)
No HH is detected

Figure 1: Heavy hitter application running in each pipe. In this case, no HH is detected.

The switch is running an HH detection application having an instance of the P4 synthesized program installed in every pipe. Now, suppose that flow *f1* starts in pipe 1 and due to a load balancing mechanism running on upstream switches in the path of *f1* (e.g. flowlet-based load balancing) or a failure that triggered fast-rerouting, *f1* arrives now in the switch through pipe 2. The HH counter at pipe 1 will have part of the packets counted while the remaining packets are counted at pipe 2. Depending on the HH threshold defined, the flow may not be counted as an HH or counted twice, once per pipe. Imagine that the HH threshold is 50 packets. If *f1* has 70 packets at all and flaps from pipe 1 to pipe 2 after counting 40 packets in pipe 1, then the HH will not be detected. If *f1* has 150 packets and flaps after have counted 50 packets in pipe 1, then the HH is counted twice.

3 MULTI-PIPE AWARE HEAVY HITTER DETECTION

The main idea behind our solution is to add a cache, that may be implemented using registers, in the ingress pipeline and move the HH application to the egress pipeline. In the egress, the application is running in only one pipe instead of being spread among all the pipes. We can also partition the data structures of the HH application among all egress pipes to achieve more uniform memory utilization, however for simplicity, we assume all data structures are in a single egress pipe.

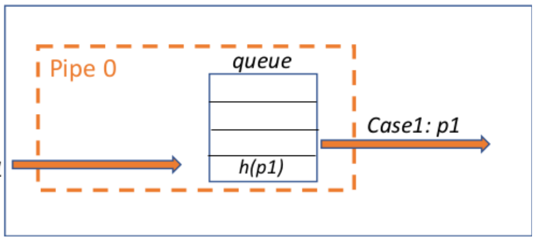
The cache is responsible for storing the hashes of the packet identifiers that can be used to update the data structures in the egress pipe. The hash of a packet identifier is stored into a cache when the output pipe of that packet is different from the pipe where the HH is running. If the data packet is instead forwarded to the pipe where the HH application is running, then no cache is needed. In this case, the data packet can carry the hashes (if any) as metadata from the cache to the HH counter in the egress pipeline.

Figure 2 shows an example of how our mechanism works in a 2-pipe switch. In the figure, the HH application is running in pipe 1. There are two cases to be considered:

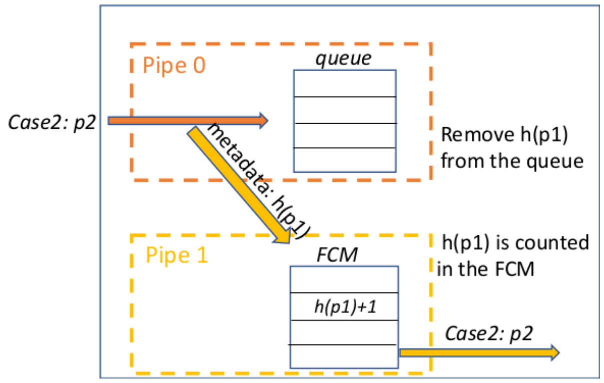
- Packet $p1$ arrives in pipe 0 and is forwarded to a port belonging to pipe 0 (Fig. 2a). In this case, it is necessary to add the hash into the cache at the ingress. A hash function is applied using a pre-configured "flow class" identifier, *e.g.,* the source IP address identifier or any other field combination. The data packet is then forwarded to its designated output port.
- Packet $p2$ arrives in pipe 0 and is forwarded to a port belonging to pipe 1 (Fig. 2b). In this case, it is not necessary to cache the hash in a queue at the ingress because the packet will be counted by the HH application in pipe 1. In this case, our mechanism carries the hashes stored in the cache (*i.e.,* the hash of p1) to the egress adding such hashes as metadata in the data packet as a way to drain the cache. The data structure in the egress pipeline updates both entries for $p1$ and $p2$. The data packet ($p2$) is then forwarded to the output port.

In the figure, for sake of demonstration, only one hash is being carried from ingress to egress. However, more hashes can be stored in the cache and moved as metadata depending on how many hashes are needed by the HH application as well as the switch capacity of updating multiple entries of a register (or updating multiple registers) in parallel.

The size and the quantity of queue data structures used in the ingress pipeline need also to be tuned with care. Queues are implemented as registers and such resource is scarce in programmable hardware. At the same time, the size of the queues affects the number of hashes that can be queued.



(a) Caching the hash.



(b) No caching. Carrying the hash from the queue to egress.

Figure 2: HH detection in a 2-pipe switch.

We carried out some preliminary experiments on a simulator which mimics a simplified programmable switch with 2 and 4 pipes. For instance, we simplify the switch architecture by assuming that at most 8 cached items can be moved to the egress pipeline. We also assume multiple updates can be performed on a register. We leave the problem of supporting multiple updates by partitioning the data structures into multiple registers as future work. For our evaluations, we adopted FCM [19] as the HH application. FCM is a three level sketch which consists of three hierarchical levels of registers that need to be updated for each single received packet.

We re-use the same parameters and traces from the FCM paper. The size of the FCM registers data structures is 2^{19} 8-bit entries for level 1, 2^{16} 16-bit entries for level 2, and 2^{13} 32-bit entries for level 3. Considering that FCM uses 2 hash functions, the memory requirements for the implementation are 1.31MB of SRAM in each pipe, which means 2.62MB for a 2-pipes switch and 5.24MB for a 4-pipes switch. We consider the task of detecting heavy hitters (classified with the source IP address) and use the f1-score as a metric of the system performance. F1-score is calculated as follows:

$$f1 - score = \frac{2 \times PR \times RR}{PR + RR}$$

where PR (Precision Rate) is the ratio of true instances reported including non HH and RR (Recall Rate) is the ratio of reported true instances.

We use the same data trace and the same hash function (BOB hash) utilized by the FCM paper. There are around 500K flows and 166 HHs in such a trace (with an HH threshold of 10K packets). We also collected the average queues size so that we can observe the total memory occupied by the cache in the ingress pipelines. Since we run FCM only on one egress pipe, the memory occupied by it is 1.31MB. We could spread FCM over all egress pipes and utilize 327KB per pipe. In both cases, this is a 4x reduction of memory utilization on a 4-pipe switch.

Table 1 presents a summary of the results.

#pipes	HH found	Non HH found	PR	RR	f1-score
2	166	4	0.9764	1	0,9880
4	166	5	0.9707	1	0,9851

Table 1: HH detection using FCM in a multi-pipe switch.

The "standard" FCM achieves an f1-score of 99.4% while our solution achieves 98.80% and 98.51% for 2 and 4 pipes, respectively, which is a very small difference.

We also analyzed the the maximum size of the queues used for caching. In our experiments, we choose to have 4 and 8 queues for 2 and 4 pipes, respectively. The results showed that, on average, assuming an evenly distribution of the traffic among the pipes, the maximum queue size is around 300 packets, respectively for 2 and 4 pipes. Considering a 32-bit hash, the memory usage is of 9.37KB (2 pipes) and 37.5KB (4 pipes), causing a very small impact on the memory occupancy.

4 CONCLUSIONS

In this paper, we touched upon the problem of deploying a standard-single pipe HH application (FCM) into a multi-pipe switch with 1/4th of memory usage keeping very similar accuracy in terms of f1-score. We also observed that the cache mechanism requires very small queues, at the least for the tested trace.

Future works include implementing MPHH on a Tofino architecture to tackle any architectural constraints such as limited number of updates per register. Moreover, we plan to evaluate our MPHH using other traces as well as other data-plane applications which are currently impacted by multi-pipes architectures such as load balancing, DDoS attacks and distributed machine learning training.

ACKNOWLEDGMENTS

We would like to thank CNPq and SAAB for funding this project.

REFERENCES

[1] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing Heavy-Hitter Detection Algorithms for Programmable Switches. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1172–1185. https://doi.org/10.1109/TNET.2020.2982739

[2] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. 2019. PURR: A Primitive for Reconfigurable Fast Reroute: Hope for the Best and Program for the Worst. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3359989.3365410

[3] Advait Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella. 2013. On the impact of packet spraying in data center networks. In *2013 Proceedings IEEE INFOCOM*. 2130–2138. https://doi.org/10.1109/INFCOM.2013.6567015

[4] P4 Spec Forum. 2021. *[PSA] Document effect of multiple 'pipelines' on Register extern?* Retrieved October 12, 2021 from https://github.com/p4lang/p4-spec/issues/353

[5] Hasanin Harkous, Chrysa Papagianni, Koen De Schepper, Michael Jarschel, Marinos Dimolianis, and Rastin Pries. 2021. Virtual Queues for P4: A Poor Man's Programmable Traffic Manager. *IEEE Transactions on Network and Service Management* 18, 3 (2021), 2860–2872. https://doi.org/10.1109/TNSM.2021.3077051

[6] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 161–176. https://www.usenix.org/conference/nsdi19/presentation/holterbach

[7] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A Programmable System for Performance-aware Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 701–721. https://www.usenix.org/conference/nsdi20/presentation/hsu

[8] Jiawei Huang, Wenjun Lyu, Weihe Li, Jianxin Wang, and Tian He. 2021. Mitigating Packet Reordering for Random Packet Spraying in Data Center Networks. *IEEE/ACM Transactions on Networking* 29, 3 (2021), 1183–1196. https://doi.org/10.1109/TNET.2021.3056601

[9] Intel. 2021. *P416 Intel Tofino Native Architecture - Public Version.* Retrieved October 12, 2021 from https://bit.ly/3Bydw4E

[10] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

[11] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. 2018. BurstRadar: Practical Real-Time Microburst Monitoring for Datacenter Networks. In *Proceedings of the 9th Asia-Pacific Workshop on Systems* (Jeju Island, Republic of Korea) *(APSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/3265723.3265731

[12] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. *SOSR '16: Proceedings of the Symposium on SDN Research* 10 (mar 2016), 1–12. https://doi.org/10.1145/2890955.2890968

[13] Xin Zhe Khooi, Levente Csikor, Dinil Mon Divakaran, and Min Suk Kang. 2020. DIDA: Distributed In-Network Defense Architecture Against Amplified Reflection DDoS Attacks. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. 277–281. https://doi.org/10.1109/NetSoft48620.2020.9165488

[14] Xin Zhe Khooi, Levente Csikor, Jialin Li, Min Suk Kang, and Dinil Mon Divakaran. 2021. Revisiting Heavy-Hitter Detection on Commodity Programmable Switches. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. 79–87. https://doi.org/10.1109/NetSoft51509.2021.9492531

[15] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 387–406. https://www.usenix.org/conference/osdi20/presentation/li-jialin

[16] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 44–58. https://doi.org/10.1145/3341302.3342085

[17] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/2934872.2934906

[18] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

[19] Cha Hwan Song, Pravein Govindan Kannan, Bryan Kian Hsiang Low, and Mun Choon Chan. 2020. FCM-Sketch: Generic Network Measurements with Data Plane Support. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 78–92. https://doi.org/10.1145/3386367.3432729

[20] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 407–420. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini

[21] Wei Wang, Yi Sun, Kai Zheng, Mohamed Ali Kaafar, Dan Li, and Zhongcheng Li. 2014. Freeway: Adaptively Isolating the Elephant and Mice Flows on Different Transmission Paths. In *2014 IEEE 22nd International Conference on Network Protocols*. 362–367. https://doi.org/10.1109/ICNP.2014.59

[22] Xiong Z and Zilberman N. 2019. Do switches dream of machine learning?: Toward in-network classification. 25–33.

[23] Tao Zhang, Yasi Lei, Qianqiang Zhang, Shaojun Zou, Juan Huang, and Fangmin Li. 2021. Fine-grained load balancing with traffic-aware rerouting in datacenter networks. *Journal of Cloud Computing: Advances, Systems and Applications* 37, 10 (jul 2021), 1–20. https://doi.org/10.1186/s13677-021-00252-8