Traffic Footprint Characterization of Workloads using BPF

Aditi Ghag, Yaniv Ben-Itzhak, Justin Pettit, Ben Pfaff
VMware
aghag@vmware.com

ABSTRACT

Application workloads are becoming increasingly diverse in terms of their network resource requirements and performance characteristics. As opposed to long running monoliths deployed in virtual machines, containerized workloads can be as short lived as few seconds or minutes or in case of microservices, highly distributed and communication intensive. Quantifying network resource requirements of application workloads is a challenging problem since network is a highly dynamic and distributed resource. Getting visibility into network characteristics of diverse workloads will enable their intelligent placement in container or VMs on a common infrastructure. Today, container orchestrators that schedule these workloads primarily consider their CPU and memory resource requirements since they can easily be quantified. However, network resources characterization isn't as straight forward. Ineffective scheduling of containerized workloads, which could be throughput intensive or latency sensitive, can lead to adverse network performance. Hence, this work proposes an eBPF (extended Berkeley Packet Filter) based framework that characterizes and learns network footprints of applications running in a cluster, and thereby making network a first class citizen in areas like container scheduling.

1 KEYWORDS

BPF, Linux, networking, Conntrack, resource scheduling

2 INTRODUCTION

Data centers and cloud environments host workloads that are quite diverse in terms of their network resource requirements and performance characteristics. There are latency-sensitive applications like web search queries or front-end applications, or in-memory key-value stores such as Redis that are used as caching layers. These applications tend to have stringent tail latency requirements, that are usually defined by SLA terms like the 99^{th} percentile response latency. On the other hand, there are applications like data analytics, file transfers, Hadoop that are throughout intensive. Their large throughput requirements are usually satisfied by the large available capacity in 10GB+ in virtualized environments. On the other side of the spectrum, there are workloads like Functions that can be as short-lived as few seconds or minutes or microservices, which are highly distributed and communication intensive. In case of microservice applications, services are deployed in containers and they make calls over the network to execute end-to-end business logic. Inter-services network calls, thus, cumulatively contribute to end to end application response time.

There has been a recent trend, where these applications are being containerized, and they are deployed by a container orchestrations platform like Kubernetes. An effective resource scheduler needs to account for the performance requirements of diverse workloads while scheduling them on a common infrastructure. Container orchestrators, such as Swarm [9], Mesos [1], and Kubernetes [6] are the leading orchestration platforms. In this paper, we focus on Kubernetes, which is one of the most popular orchestrators used in the industry. However, the ideas proposed in this paper are generally applicable to other orchestrators as well. A Kubernetes cluster is formed from a master node, and a pool of worker nodes. In most deployments, these nodes are virtual machines. The master node runs all the Kubernetes control plane components, including the scheduler. The scheduler manages groups of containers, referred to as *pods*, and schedules them by deciding which worker node should host each pod [1]. Currently, the scheduling decisions [5] are made based on availability of resources like CPU, memory, disk, etc and some policy constraints. Given that applications §2 rely on network, we propose to augment Kubernetes scheduler with network awareness about the network characteristics of application workloads. In this paper, we take an approach to build an eBPF based framework in order to learn and characterize network footprint characteristics of application workloads. We further show how this enhanced information can be used to augment current container schedulers.

The next section describes how workloads can be characterized based on their network footprint. This is followed by implementation details of the eBPF based framework, traffic footprint-aware container scheduling and future work.

[1] In this paper, we use pods and containers interchangeably.

3 CHARACTERIZATION OF WORKLOADS

This section describes how workloads can be characterized based on their network footprint.

3.1 Latency v/s Throughput

We focus on a well known network issue, which is achieving low latency for mice flows (those that send relatively little amounts of data) by separating them from the elephant flows (those that send a lot of data). Research shows that these elephant flows tend to fill network buffers [12, 15, 21], penalizing the throughput and tail latency of delay-sensitive mice flows and *coflows*, that is, collections of flows with a shared completion time [13, 14, 19, 22]. Most microservices traffic consist of mice flows from short API call requests and responses, which can get drowned out by other microservices running elephant flows. Automatically identifying the mice and elephant microservices flows becomes really important. We distinguish throughput and latency considerations, since the former can be satisfied by the large available capacity (10+ Gbps) in virtualized environments, even when multiple workloads are sharing the network bandwidth. We distinguish throughput and latency considerations, since the former can be satisfied by the large available capacity (10+ Gbps) in virtualized environments, even when multiple microservices are sharing the network bandwidth.

3.1.1 Co-located Elephant and Mice flows. To learn how heavy network footprint microservices affect light network footprint microservices, we conducted several tests by creating different combinations of these deployed in a Kubernetes cluster. We deployed 2 KVM hypervisors with worker nodes, as depicted in Figure 1. We then deployed pairs of pods to generate flows of the desired kinds, using `iperf` pairs to generate elephants and `sockperf` pairs for mice. For different placements of client-server pod pairs, we measured the latency experienced by the mice flows.

Figure 1 shows the results. The baseline 99th percentile latency of `sockperf` was 0.28 ms when its client and server pods are deployed in the same hypervisor. Deployed across two different hypervisors, the latency increased 4× to 1.16 ms. Co-locating `sockperf` pods with `iperf` pods on a single worker node increased mice latency by 12× to 3.32 ms. We measured the worst mice latency when paired `iperf` and `sockperf` pods were both separated across different hypervisors, at 14.04 ms or 50× baseline. We observed a similar spike in latency if we co-located the `sockperf` client or server onto the same worker node as the `iperf` client. This demonstrates how detecting nodes running containers that

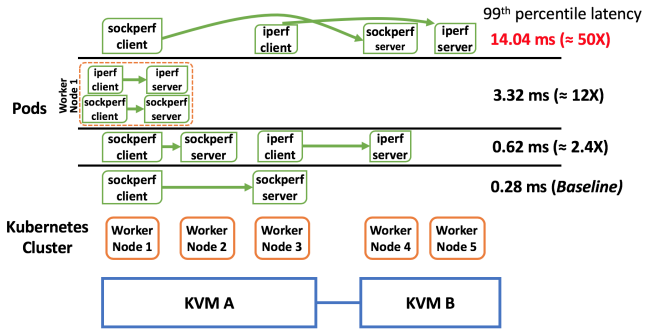

Figure 1: Sockperf client 99th percentile latency for different placements of sockperf (mice) and iperf (elephant) pods

are contributing to elephant flows, in addition to identifying the hypervisors that source and sink these flows, can aid scheduling light network footprint microservices.

3.2 Characterizing the Network Footprint

We say that workloads that source or sink elephant flows have a *heavy* network footprint and that others have a *light* footprint. A naive scheduling algorithm can potentially harm performance by scheduling heavy and light footprint containerized workloads into the same node. Hence, we built a framework to learn workloads' network footprints and explicitly tag those with heavy footprints. When new instances of these tagged containers are deployed later, the scheduler tries to keep them away from light footprint containers.

3.3 Detecting and Mapping Elephant Flows in End Hosts

Past research proposing sampling and in-network elephant detection solutions [11, 15, 20] focus mainly on detecting elephant flows, without focusing on locating their sources or sinks. To identify the flows' endpoints VMs or containers, one would have to track the IP addresses associated with these VMs and containers, which might not scale well for environments running containers at a huge scale.

Moreover, in overlay network-based virtualized environments, containers or VMs traffic get encapsulated, and multiple tenant networks can have overlapping subnets. In such cases, more information such as virtual network identifier (VNI) [4, 10] needs to be extracted from encapsulation headers in order to identify VM and container endpoints from flow level data. For these reasons, we leverage eBPF to map elephant flows to their corresponding container (see § 4.1 and 4.2 for details).

4 IMPLEMENTATION

This section describes the eBPF based traffic footprint characterization framework in detail.

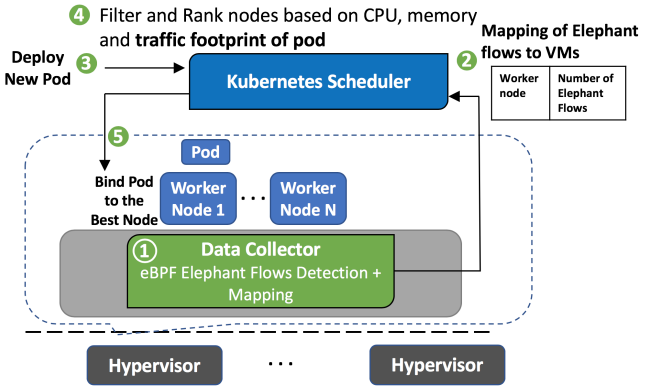

Figure 2: Traffic footprint-aware Container Scheduling

4.1 eBPF

To detect runtime behavior in the hypervisor, we leveraged extensions to the BPF subsystem in recent versions of the Linux kernel. BPF [2] has been used in networking for decades, primarily for filtering network packets. For example, when a `tcpdump` user specifies a packet filter, `tcpdump` translates the filter into a small BPF program and passes it to the kernel. The kernel checks the program for safety and then runs it on every received packet.

BPF was originally a simple stack-based virtual machine. Recent Linux kernels have enhanced BPF to look like a modern CPU, which makes it able to run larger programs more efficiently. These *extended BPF*, or eBPF [3], programs can be configured to run at various *hook points* in the kernel. Hook points now exist in nearly every subsystem within the Linux kernel [16]. Using appropriate hook points, the eBPF program can safely and efficiently retrieve runtime information from the hypervisor kernel.

4.2 Detecting and Mapping Elephant Flows Using eBPF

We used Linux kernel connection tracking functionality, called *conntrack*, which tracks the lifecycle of every flow and maintains flow-level statistics. We wrote an eBPF program in C, which traces three conntrack kernel module functions to get flow-level information.

4.2.1 Data Structures. The eBPF program creates a BPF map to store flow-level attributes. The BPF map key and value structures are listed below. The key is a struct containing 5-tuple using which a flow is identified. The value structure comprises of flow attributes like interface, where the flow originated from, timestamp to keep track of flow duration, conntrack zone identifier (usage is explained in section §4.3), and a flag that indicates whether a flow is an Elephant flow.

```
struct flow_key
{
    u32 src_addr;
    u32 dst_addr;
    u16 src_port;
    u16 dst_port;
    u8  protocol;
};
struct flow_attributes
{
    char iface_name[IFNAMSIZ];
    u64  tstamp;
    u16  zone_id;
    bool is_elephant_flow;
};
```

4.2.2 Tracing Conntrack kernel events. The first hook point in the eBPF program executes whenever a new flow is added. The hook computes a key for the flow from its 5-tuple and inserts an entry for it in a BPF map. We only track flows that are marked by conntrack as `ASSURED`, meaning that traffic has been seen in both directions. This conserves memory by filtering stray traffic.

The second hook point is executed when a packet updates statistics for a tracked flow. If the byte count and timestamp for the flow indicate that this flow now qualifies as an elephant, the hook sets a flag in the entry to that effect and then it generates an add-flow event with information about the flow and sends it to the data collector discussed in the next section.

Finally, the third hook point is executed when a flow is deleted from conntrack. The program looks up this flow in the map, and if it's marked as an elephant flow, it sends a delete-flow event to the data collector.

4.2.3 Overhead. We measured throughput and latency of iperf and sockperf applications, respectively, with and without running the eBPF program. The penalty of additional processing done by eBPF programs was found to be negligible. To identify an elephant flow, we maintain flow-level additional attributes in a map of 100,000 entries, which takes up about 3 MB of memory. Memory overhead could be reduced by storing the extended flow statistics in conntrack module's per-flow extension metadata. Since BPF allows running code directly in the kernel from user space, this new functionality can be added with minimal CPU overhead.

4.3 Learning Containerized Workload Network Footprint

All the containers in a VM share the VM's virtual network interface. To learn a containerized workload's footprint, we need to attribute a given elephant flow to a particular container. For this benefit, we used *conntrack zone* [8], an identifier that partitions conntrack entries for namespacing and fairness.

Each container's traffic is mapped to a different conntrack zone. Our eBPF hooks store the conntrack zone id in the map described earlier. When the eBPF program detects an elephant flow, it identifies its container using its zone id. The corresponding workload running inside the container will then be identified and tagged as one with a heavy network footprint. This information is fed to a container scheduler such that when the instances of this workload are later redeployed, the tagged network information can be proactively used by the scheduler, described in §4.4.

4.4 Traffic footprint-aware Container Scheduling

We run a distributed data collector, written in Python, on every hypervisor in its Kubernetes cluster. The collector loads the eBPF program described in the previous section. It then listens for add and delete elephant flow events from the BPF program. The collector also translates the hypervisor's VM interface to the VM's node label in the Kubernetes cluster so that the Kubernetes scheduler can identify the correct worker node. It then creates an update message, for every node that includes the number of elephant flows running in containers deployed inside the node, and sends it to the scheduler, as shown in Figure 2. The scheduler will try to schedule containers running light network footprint workloads away from worker nodes that are running heavy network footprint containers.

5 OTHER USER CASES

There can be other potential use cases that can use the network footprint characterizing eBPF based framework.

Resource Allocation: Containers or VMs running workloads with heavy network footprint can be allocated high bandwidth, and dynamically assigned to separate receive-side scaling (RSS) [7] queues.

Hardware offloading: The eBPF framework can be extended to tag Elephant flows or flows that can benefit from hardware offloading, depending on what kind of flow processing is being offloaded to hardware.

Flowlet generation Flowlet switching [18] is a popular idea, proposed in the past, that divides flows into smaller units for network load balancing purposes, with limiting reordering. But studies [17] show that dividing small flows for load balancing is unnecessary, and need not be exposed to reordering. Hence, the eBPF framework when identifies Elephant flows can mark these flows such that flowlet generation logic can only operate on such long-lived flows.

6 FUTURE WORK

Storing flow attributes Currently, we use a BPF map to store flow attributes, which can be stored in conntrack extension metadata.

High scale environments We need to deploy the framework in high scale environments to further validate the overheads of the eBPF program.

Explore other use cases It's worthwhile to explore the other use cases.

Other network characteristics The current framework can be extended to characterize other network characteristics of workloads such as in-cast scenarios, applications dependencies, packet rate, etc.

7 CONCLUSION

We presented a framework that uses BPF to characterize network footprint of application workloads. The enhanced information is then used to augment container schedulers in order to optimize performance of latency sensitive workloads.

8 ACKNOWLEDGEMENTS

The author would like to thank Yi-Hung Wei for discussions, on Conntrack, Cheng-Chun Tu and Brenden Blanco for help with debugging eBPF program.

REFERENCES

[1] Apache Mesos. http://mesos.apache.org/.
[2] BPF. https://www.kernel.org/doc/Documentation/networking/filter.txt.
[3] eBPF reference guide. https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md.
[4] Geneve: Generic Network Virtualization Encapsulation. https://datatracker.ietf.org/doc/draft-ietf-nvo3-geneve/.
[5] Kubernetes scheduler. https://github.com/eBay/Kubernetes/blob/master/docs/devel/scheduler_algorithm.md.
[6] Production-Grade Container Orchestration - Kubernetes. https://kubernetes.io.
[7] Receive Side Scaling (RSS). https://www.kernel.org/doc/Documentation/networking/scaling.txt.
[8] RFC netfilter conntrack. https://lwn.net/Articles/370152/.
[9] Swarm: a Docker-native clustering system. https://github.com/docker/swarm.
[10] Virtual eXtensible Local Area Network (VXLAN). https://datatracker.ietf.org/doc/rfc7348/.
[11] 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*, SIGCOMM '15, 2015.
[12] Y. Ben-Itzhak, C. Caba, L. Schour, and S. Vargaftik. C-share: Optical circuits sharing for software-defined data-centers. *arXiv preprint arXiv:1609.04521*, 2016.
[13] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.
[14] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 443–454. ACM, 2014.

[15] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM, 2011 Proceedings IEEE*, pages 1629–1637. IEEE, 2011.

[16] B. Gregg. Performance superpowers with enhanced BPF. USENIX ATC, 2017.

[17] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based load balancing for fast datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 45(4):465–478, Aug. 2015.

[18] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2):51–62, Mar. 2007.

[19] Z. Qiu, C. Stein, and Y. Zhong. Minimizing the total weighted completion time of coflows in datacenter networks. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 294–303. ACM, 2015.

[20] 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*, SOSR '17, 2017.

[21] R. Trestian, G.-M. Muntean, and K. Katrinis. Micetrap: Scalable traffic engineering of datacenter mice flows using openflow. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 904–907. IEEE, 2013.

[22] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 424–432. IEEE, 2015.