Traffic Footprint Characterization of Workloads using BPF

Aditi Ghag

aghag@vmware.com

VMware

Outline

- Background
- Scheduling use case
- Characterization of workloads
- eBPF based framework
- Traffic footprint-aware container scheduling
- Discussion

Diversity of Workloads

Latency sensitive

Web search

Front-end

In-memory key-value store

Throughput intensive

Data analytics

Map reduce

live VMs migration

Short-lived

Functions

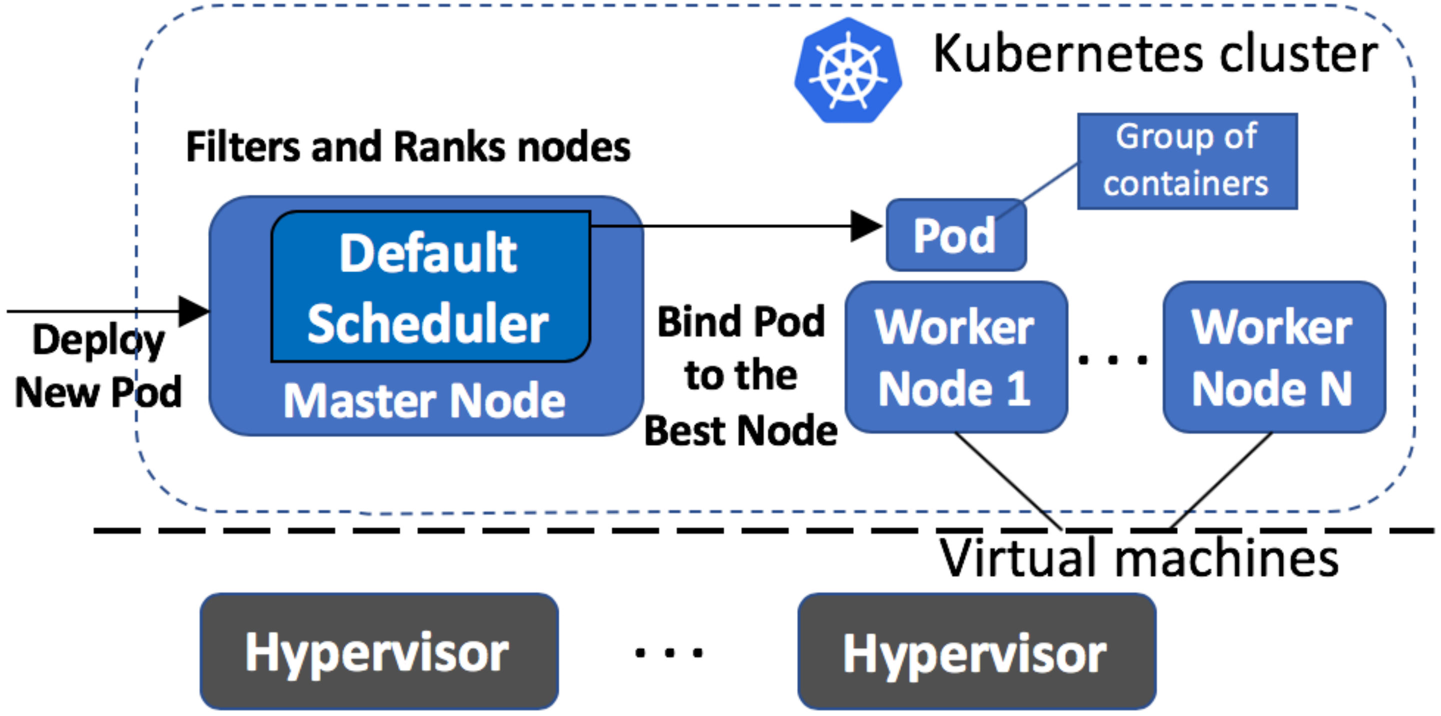
Distributed and Communication intensive

Microservices

Resources Scheduling use case

- Containerization
- Container Orchestration frameworks

Current Container Scheduling



- CPU
- Memory
- Policy

How do we add network awareness to the scheduler?

Characterization of Workloads

- Identify network characteristics of workloads

Traffic Footprint Characterization of Workloads

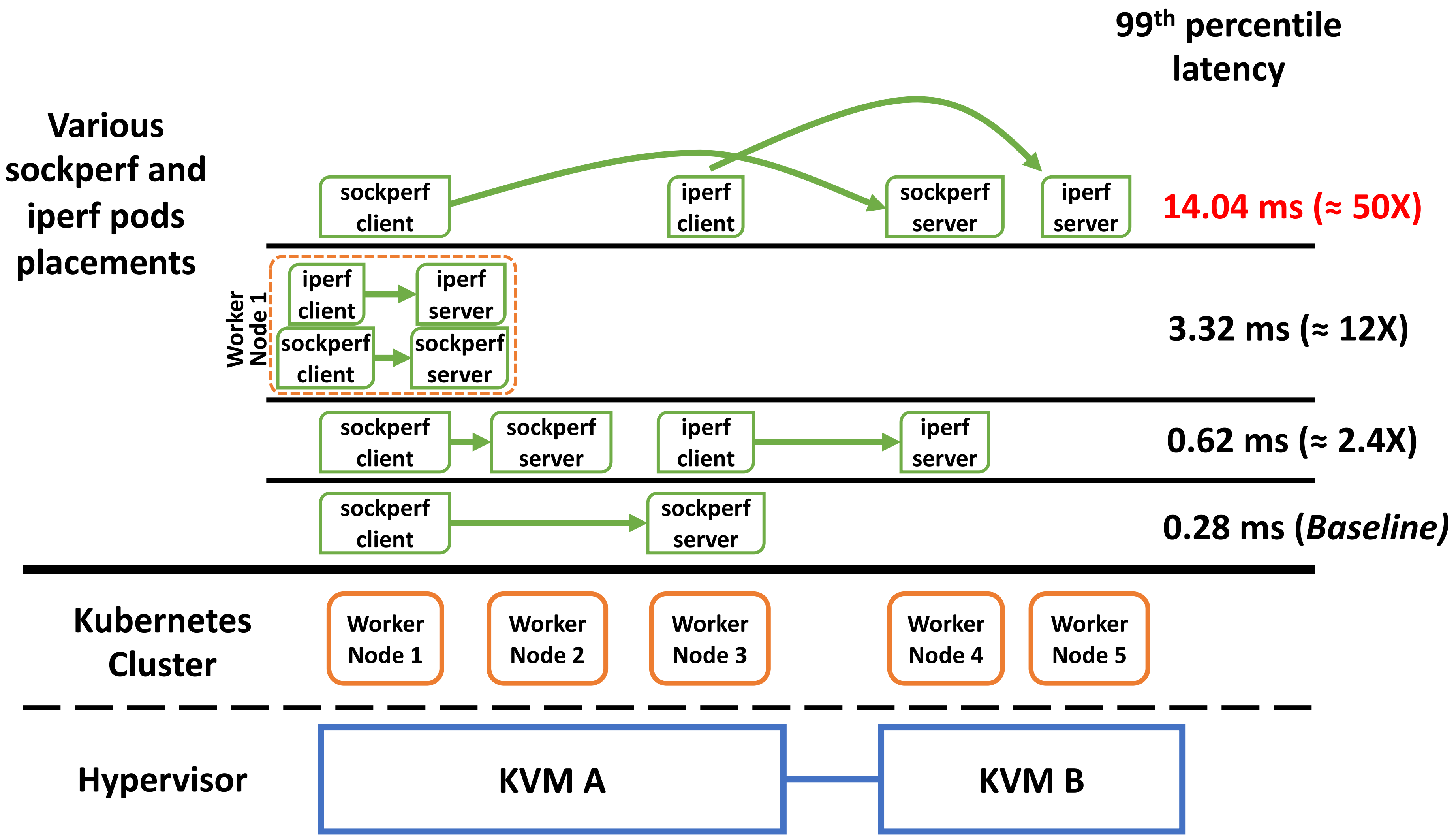
Elephants v/s Mice

- Elephant flows fill up network buffers
  - packet drops and queuing delays
  - Increased tail latency of mice flows

Containers (or VMs) that source or sink elephant flows: ***heavy*** network footprint

Effect of Elephant flows on Mice Flows

sockperf: Mice Flows
iperf: Elephant Flows



99th percentile latency

Various sockperf and iperf pods placements

sockperf client → **sockperf server**; **iperf client** → **iperf server** — **14.04 ms (≈ 50X)**

Worker Node 1: **iperf client** → **iperf server**; **sockperf client** → **sockperf server** — **3.32 ms (≈ 12X)**

sockperf client → **sockperf server**; **iperf client** → **iperf server** — **0.62 ms (≈ 2.4X)**

sockperf client → **sockperf server** — **0.28 ms (*Baseline)***

Kubernetes Cluster: **Worker Node 1**, **Worker Node 2**, **Worker Node 3**, **Worker Node 4**, **Worker Node 5**

Hypervisor: **KVM A**, **KVM B**

Detecting and Mapping Elephant Flows in End Hosts

Detecting Elephant Flows

Closer to application: has more context

Mapping Elephant flows to containers/VM(s)

- Learn workload network footprint
- Identify network state at infrastructure level

Traffic Footprint Characterizing Framework

eBPF based Elephant Flows Detection and Mapping

eBPF and Conntrack

- Conntrack tracks lifecycle of every flow

- eBPF enables to run user-supplied programs inside of kernel

- eBPF programs attached to Conntrack kernel events

eBPF Tracing with Conntrack

Data Structures

- BPF hash map
- Flow entry key
- Flow attributes value

Data Structures

```
struct flow_key
{
    u32 src_addr;
    u32 dst_addr;
    u16 src_port;
    u16 dst_port;
    u8  protocol;
};
```

```
struct flow_stats
{
    char iface_name[IFNAMSIZ];
    u64 tstamp;
    u16  zone_id;
    bool is_elephant_flow;
};
```

eBPF Tracing with Conntrack

Data Structures

- BPF hash map
- Flow Entry

Elephant Flows Detection and Mapping

- 1st hook point:
  Add flow
- 2nd hook point:
  Update flow counters
- 3rd hook point:
  Delete flow

Objective: Detect and map elephant flows to containers/VM(s)

Add Flow (1st Hook point)

Update Flow

Delete Flow

```
BPF_HASH(flows, struct flow_key, struct flow_stats);

int kprobe__nf_ct_deliver_cached_events(struct pt_regs
                                              *ctx,

                                           const struct
                                              nf_conn *ct)
{
       // Look for ‘ASSURED’ flows
       // Create flow entry in BPF hash map
}
```

Objective: Detect and map elephant flows to containers/VM(s)

Add Flow

Update Flow (2nd Hook point)

Delete Flow

```
// BPF table for sending 'add mapped elephant flows' event data to user space
BPF_PERF_OUTPUT(add_mapped_elephant_flows);


 int kprobe__nf_ct_refresh_acct(struct pt_regs *ctx,
                                 const struct nf_conn *ct,
                                 enum ip_conntrack_info ctinfo,
                                 const struct sk_buff *skb)
 {
     // Parse kernel data structures
    //  Identify elephant flows using number of bytes transferred
    //  Generate add elephant flow event
  add_mapped_elephant_flows.perf_submit(ctx, &flow_stats,
                                         sizeof(flow_stats));
 }
```

Objective: Detect and map elephant flows to containers/VM(s)

Add Flow

Update Flow

Delete Flow

(3rd Hook point)

```
// BPF table for sending 'delete mapped elephant flows' event data to user space
BPF_PERF_OUTPUT(del_mapped_elephant_flows);

  bool kprobe__nf_ct_delete(struct pt_regs *ctx,
                            const struct nf_conn *ct)
 {
     // If the given flow is marked as an elephant flow, generate delete elephant
     // flow event
    // Delete entry from BPF map

     del_mapped_elephant_flows.perf_submit(ctx, &flow_stats,
                                           sizeof(flow_stats));
 }
```

eBPF Tracing with Conntrack

Data Structures

- BPF hash map
- Flow Entry

Elephant Flows Detection and Mapping

- 1st hook point: Add flow
- 2nd hook point: Update flow counters
- 3rd hook point: Delete flow

Attributing Elephant Flows to Containers

- Conntrack Zones as identifiers

Traffic Footprint-aware Resource Scheduling

- Network-aware Kubernetes scheduler

Augmenting Container Scheduler (1)

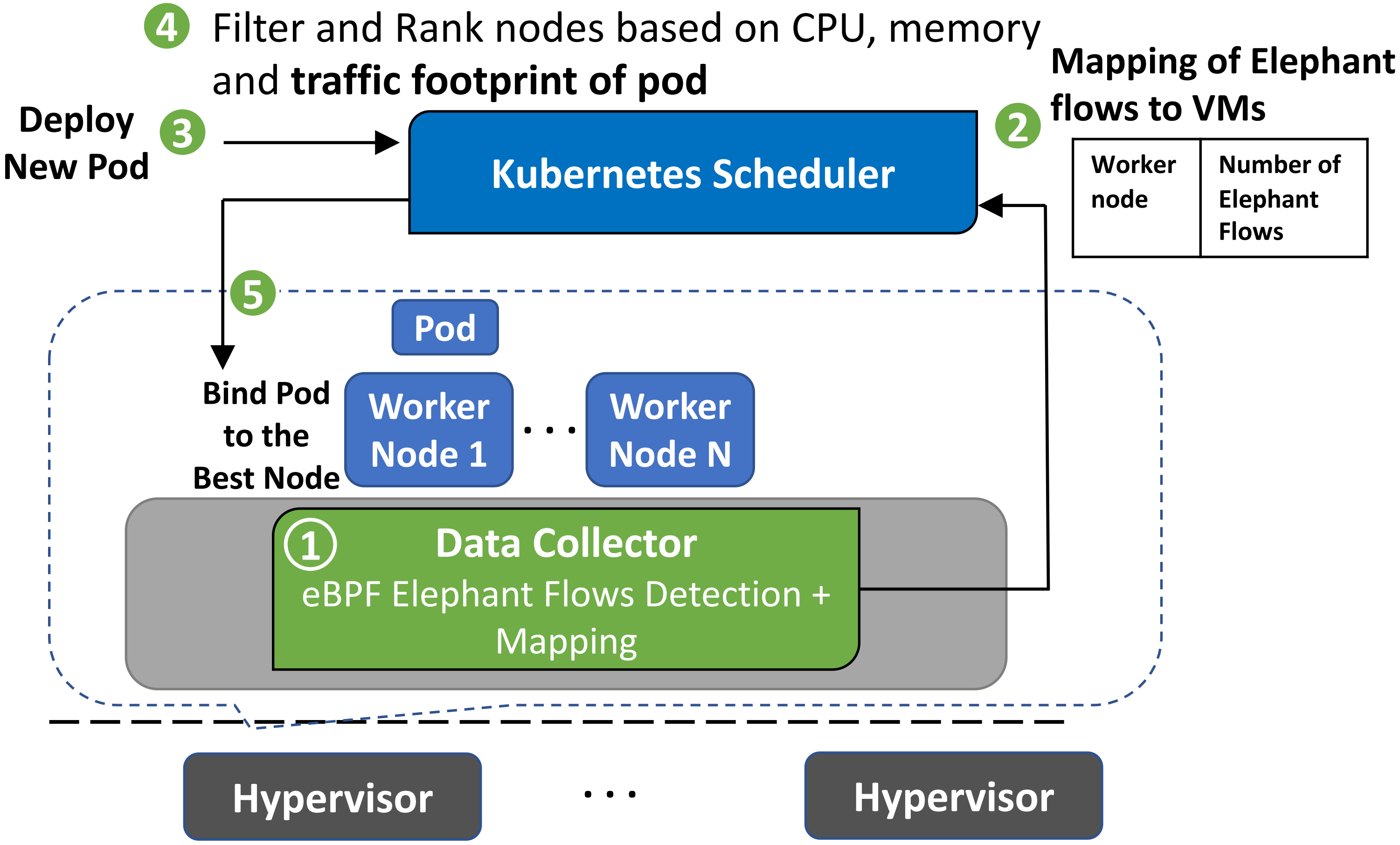
- Tag workloads with network footprint information

- Proactively isolate heavy and light footprint workloads

- Prefer hosts with less number of elephant flows

Augmenting Container Scheduler (2)

Goal: Intelligent placement of a pod in a VM (aka node)



4 Filter and Rank nodes based on CPU, memory and **traffic footprint of pod**

Deploy New Pod 3

Kubernetes Scheduler

2 **Mapping of Elephant flows to VMs**

Worker node	Number of Elephant Flows

5

Bind Pod to the Best Node

Pod

Worker Node 1 . . . **Worker Node N**

1 **Data Collector**

eBPF Elephant Flows Detection + Mapping

Hypervisor . . . **Hypervisor**

Other use cases

- Receive-side scaling
- Resources allocation
- Hardware offloading
- Flowlet generation

Future Work

- Use Conntrack metadata to store flow stats
- Deploy in high scale environments
- Explore other use cases
- Identify other network characteristics

Thank you

Questions?