PerfSight: Performance Diagnosis for Software Dataplanes

Wenfei Wu, Keqiang He, Aditya Akella
University of Wisconsin-Madison

ABSTRACT

The advent of network functions virtualization (NFV) means that data planes are no longer simply composed of routers and switches. Instead they are very complex and involve a variety of sophisticated packet processing elements that reside on the OSes and software running on compute servers where network functions (NFs) are hosted. In this paper, we argue that these new "software data planes" are susceptible to at least three new classes of performance problems. To diagnose such problems, we design, implement and evaluate, PerfSight, a ground-up system that works by extracting comprehensive low-level information regarding packet processing and I/O performance of the various elements in the software data plane. PerfSight then analyzes the information gathered in various dimensions (e.g., across all VMs on a machine, or all VMs deployed by a tenant). By looking across aggregates, we show that it becomes possible to detect and diagnose key performance problems. Experimental results show that our framework can result in accurate detection of the root causes of key performance problems in software data planes, and it imposes very little overhead.

Categories and Subject Descriptors

C.2 COMPUTER-COMMUNICATION NETWORKS [**C.2.3 Network Operations**]: Network management

Keywords

data center networks, software data plane, performance, troubleshooting

1. INTRODUCTION

Data plane diagnosis tools are invaluable toward managing and troubleshooting networks. Tools such as ping and traceroute, and frameworks such as NetFlow [4] and sFlow [8], are routinely used by network operators both to understand whether the network is functioning as expected, and, if not, understand what may be causing the underlying problem.

However, in recent years, network data planes have changed in fundamental ways. In addition to simple L2 and L3 devices, they are increasingly composed of a wide range of network functions, or middleboxes, that perform custom packet processing to aid in satisfying various network-wide objectives pertaining to performance, security, and compliance; examples include firewalls, load balancers, application gateways, accelerators, etc. With the advent of software switching, and more importantly, network functions virtualization (NFV), traditional hardware switching elements and middleboxes are being realized using software running on generic compute platforms (e.g., a virtual machine, or VM).

Thus, the "data plane" that packets traverse on end-to-end paths now includes—in addition to hardware L2/L3 devices and links—a variety of software components that reside within compute servers' virtualization stacks and within the VMs running various middlebox software. Examples include physical and virtual NICs and their drivers, various packet processing routines in hypervisors and within middlebox logic, virtual switches, hypervisor I/O handlers, host and guest network stacks, etc. We refer to this new portion of the data plane as the *software data plane*.

Our community has developed a variety of innovative tools and frameworks for diagnosing problems in hardware dataplanes. Examples include traceroute, path MTU discovery [5], available capacity detection [34], tomography [13], etc. Unfortunately, we don't have similar tools for software data planes.

In fact, software data planes present new challenges to diagnosis. Because they span a variety of software components running on shared compute resources, where each component can perform fairly sophisticated actions, software data planes are much more susceptible to a range of *subtle performance problems*. We argue that there are at least three classes of performance problems—those arising due to mis-allocation of resources to software data plane elements, contention amongst elements for shared resources, and buggy design/implementation. Such problems either don't arise frequently in traditional hardware data planes (e.g., implementations with performance bugs are rare), or they are simpler to diagnose because only a handful of resources are allocated (e.g., bandwidth and router buffering, vs. CPU, disk, memory, network etc. in software data planes) and contention observed manifests at a small number of locations (e.g., buffers building up or link utilization growing vs. drops/buffering at a multitude of possible locations in the virtualization stack—See Section 2.1). Furthermore, because of stateful packet processing, problems arising in one middlebox may quickly propagate up- or down-stream to other middleboxes on an end-to-end path, which complicates accurate diagnosis of root causes (See Section 2.2).

We design a general system, called PerfSight, for accurate and quick diagnosis of a broad variety of performance problems that may arise in current and future software data planes. Our approach is rooted in viewing the software data plane as a *pipeline of ele-*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
IMC'15, October 28–30, 2015, Tokyo, Japan.
© 2015 ACM. ISBN 978-1-4503-3848-6/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2815675.2815698.

ments, where an element is a logical unit that reads traffic from or writes traffic to another by buffers or function calls. This abstraction captures a variety of entities on the software data path, including middlebox logic, routines in the hypervisor, and routines in a VM's network stack. We decouple the problem of statistics collection from diagnosis. We identify elements and their input/output methods, and statically analyze their code paths to determine where packets can be buffered or dropped; we then instrument all such locations to collect a suite of statistics in a light-weight fashion. When queried by a controller at run-time, these statistics are returned in a generic format, and then consumed by diagnostic applications to perform interesting analytics.

We develop two novel applications that aid in diagnosing key software data plane problems, and we believe our approach covers a large variety of potential problems that may arise. At a high level, these diagnostic applications analyze the gathered element statistics in different "dimensions": e.g., across all VMs on a single physical machine, or across a collection of middlebox VMs that are chained together. We show that by identifying the exact locations in the software data plane where the performance problems are arising—i.e., which particular buffer or element in a VM, hypervisor or the kernel is facing a problem—we can detect contention across VMs vs. problems arising due to resource limitations within a single VM. Similarly, by studying the nature of the performance problem faced by a middlebox—e.g., whether it is stalled for read vs. writes—and in what status its neighboring middleboxes are (i.e., whether they are also stalled or not), we can quickly narrow down the root cause middlebox(es) whose performance problems have propagated throughout a chain.

We have built a prototype of PerfSight in an environment running a Linux 3.2 kernel, Open vSwitch [7], and QEMU /KVM [12]. We conduct several experiments with this prototype on a small scale experimental testbed, using topologies representing chains of middleboxes, and a variety of workloads. We study the effectiveness of PerfSight and find that in all cases PerfSight's low level instrumentation provides accurate information about the exact location in the software data plane of an observed performance problem. We also find that PerfSight can identify contention for a variety of different shared resources (e.g., memory or network bandwidth, CPU, or NIC capacity), it can accurately detect bottleneck middleboxes irrespective of the specific resource that was under-provisioned, and it can accurately pin-point the root cause of problems in complex multi-chain settings where problems can arbitrarily propagate. We also illustrate how an operator can use PerfSight to implement cross-tenant workload management and elastic scaling in a multi-tenant set-up. Finally, we show that PerfSight's instrumentation imposes an insignificant overhead ($< 1\%$).

The remainder of this paper is organized as follows. In Section 2, we describe the performance problems of software dataplanes and motivate the need for a better diagnosis framework. In Section 3, we sketch the high level architecture of PerfSight. Then, in Section 4 and Section 5, we present the design of PerfSight's two major tasks— statistics gathering and diagnostic applications. Section 6 describes the implementation details. We present experimental results showing the accuracy and overhead of PerfSight in Section 7. We present related work in Section 8 and conclude the paper in Section 9.

2. BACKGROUND AND MOTIVATION

In this section, we first present software dataplane definition and the performance problems associated with it. Then we talk about the challenges in accurate software dataplane diagnosis and the need for a better diagnosis approach.

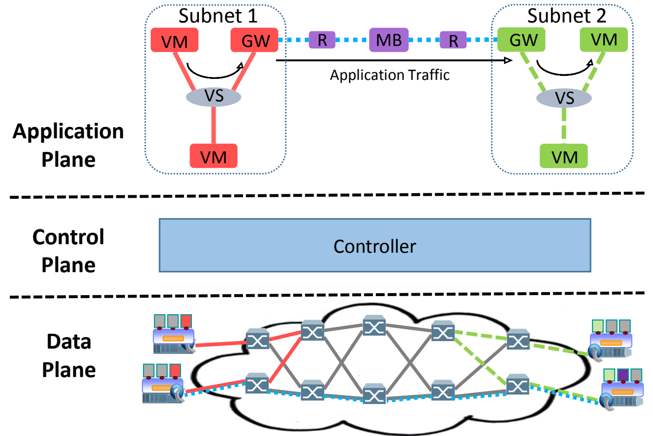


Figure 1: 3 Planes in the Cloud Architecture

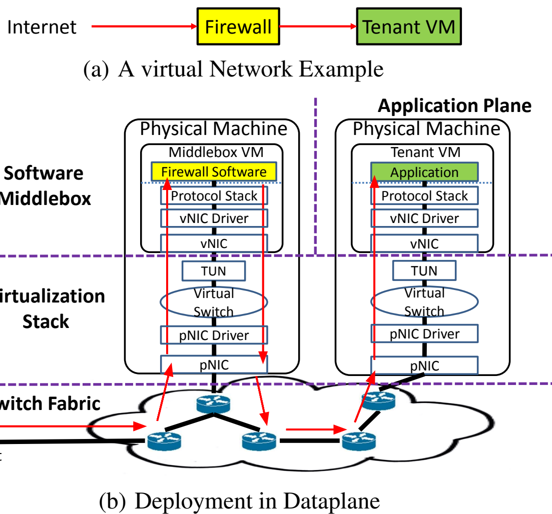
2.1 Software Dataplane

While our work applies to general networking settings, for the purposes of this paper, we focus on multi-tenant cloud data centers. In this setting, tenants deploy *virtual private clusters* composed of application end-points (this could be service software in the case of private data centers, or VMs in the case of public clouds), network function services (or middeboxes), and logical links between subsets of them. Network functions improve network performance or security, and tenants increasingly desire to deploy sophisticated sets of middlebox functionality within their clusters [30].

This setting can be viewed logically as being composed of three planes (Figure 1): *control plane*, *application plane* and *data plane*. Tenants interact with the application plane, requesting (re)deployment of virtual private clusters. The control plane, which the cloud operator runs, responds to such requests by computing suitable deployment policies, e.g., determining where VMs and middleboxes ought to be placed, instantiating virtual links between VMs and middleboxes (using tunneling schemes or encapsulation policies [30]), and computing the forwarding state configuration to determine how traffic traverses VMs/middleboxes and virtual links. The dataplane for the tenant's virtual cluster, where fast path actions are performed on the tenant's traffic, then follows the configurations provided by the control plane to deliver network traffic between the appropriate end-points in each virtual cluster.

In this paper, we focus on middleboxes that are implemented as software and deployed in VMs attached to virtual switches, an increasingly popular trend also known as *network functions virtualization (NFV)* [15]. In NFV, the middlebox VMs, similar to application VMs, are allocated fixed resources (e.g. CPU, memory, network bandwidth); the controller deploys all VMs to physical machines that have sufficient resources.

Figure 2(a) shows a tenant with a simple virtual cluster, consisting of an application VM and a firewall; furthermore, the tenant requires all Internet traffic to traverse through the firewall. While this is a simple example, virtual clusters can have much more complex sequences of middleboxes, where a given sequence may only apply to a specific traffic substream. The physical deployment computed by the controller is shown in Figure 2(b). Consider the path that Internet-originated traffic would traverse to arrive at the tenant VM (and vice versa for the outgoing traffic). Each such packet is forwarded by the cloud gateway to the physical NIC (pNIC) of a physical server hosting the middlebox VM first; then it traverses



(a) A virtual Network Example

(b) Deployment in Dataplane

Figure 2: Data Plane Organization

the pNIC driver, the virtual switch, the hypervisor I/O handler, the virtual NIC (vNIC), the vNIC driver and the VM guest OS network stack, and finally arrives at the software firewall. After being processed by the software firewall, the packet traverses all the layers back down to the pNIC. Then it is delivered by the physical network to the next hop in the virtual network (tenant VM or another middlebox).

Whereas traditional dataplanes consisted just of hardware switching elements and network links connecting network end points, the advent of NFV means that we need to rethink what constitutes the data plane. In particular, it now also includes the software components shown above that are traversed within middlebox VMs. We refer to this portion of the data plane as the *software data plane*.

The software data plane is composed of a variety of *elements*, each of which performs a certain logical function. In Figure 2, each blue rectangle or oval is an element. The software dataplane can thus be viewed as a pipeline of elements with data traversing from one to the next. The output of one element is the input to its successor. Neighboring elements exchange messages via buffers or function calls.

Elements can be further divided into two categories: (a) Those belonging to the *virtualization stack*: such elements are *shared* by multiple VMs, and examples include the pNIC driver, the packet processing routine, virtual switches and the hypervisor I/O handler; (b) those belonging to the *software middlebox*: such elements are confined to one middlebox VM, and examples include the vNIC, vNIC driver, VM guest OS network stack and middlebox software.

In a multi-tenant setting, software data planes belonging to different tenants may "overlap" on one or more physical machines.

2.2 Performance Problems

In what follows, we argue that software data plane performance problems can arise due to at least three underlying reasons. Furthermore, addressing the problem requires a different approach in each case. A diagnostic approach therefore needs to carefully delineate and accurately identify the root cause.

When the offered load on a middlebox exceeds the capacity allocated to it (along some resource dimensions), the traversing flows' throughput/latency will be **bottlenecked**. Because most middleboxes perform complex actions, such bottlenecks may arise not just due to increased traffic volume but also due to sudden, unexpected changes in the traffic profile. Addressing bottlenecks is up to the tenant (e.g., the tenant can redeploy the middlebox in a "larger" VM).

If multiple elements contend for a shared resource, and their requirements exceed the available resource capacity, all involved elements cannot achieve their expected performance. Such **contention** usually happens in the virtualization stack, because all tenant VMs and middlebox VMs in one physical machine share the same datapath in the virtualization stack. What make things worse is that the shared resource may not be explicitly allocated. For example, it is hard to allocate memory bandwidth to individual VMs; shared buffers in the virtualization stack are similarly not allocated either. To address this, impacted middlebox VMs may have to be migrated to locations with less contention. Often, this requires the cloud operator's involvement.

Design and implementation defects that lead to inefficient computation widely exist in software, and thus software dataplanes are naturally impacted by such **performance bugs**. Indeed, it has been shown previously [33] that middlebox software bugs can result in "soft failures" (e.g., a significant drop in throughput when middlebox software is "upgraded"). To address such performance bugs, a tenant must reload their VMs with a suitable version of software.

2.3 Accurate Diagnosis is Challenging

After a tenant experiences performance problems and submits trouble tickets, we assume that she submits a ticket to her data center operator. Because there are many different kinds of middleboxes and a multitude of elements in software data planes, multiple middleboxes in a virtual cluster can often be operated in a sequence (or "chained"), and multiple tenants' clusters can overlap at arbitrary physical machines; accurately diagnosing data plane performance problems and identifying root causes is not easy for the operator.

A common approach to detect bottlenecks is to monitor the resource utilization on VMs [3]. While this may work in some cases, there are a variety of middleboxes for which resource utilization does not reflect workload intensity. For example, a video stream transcoder [19] may employ non-blocking I/O instead of blocking I/O to avoid context switching. For this middlebox, CPU utilization is always 100%, but we lack a way of distinguishing the portion of CPU cycles spent on processing vs. busy waiting. Another alternative—monitoring traffic volume changes—is also insufficient as bottlenecks may arise due to changes in traffic profiles (e.g., when a middlebox encounters a traffic profile that is different from the one it was optimized for).

Contention is similarly hard to diagnose. Some OS statistics such as packet drops at the pNIC can determine certain forms of contention (e.g., for network bandwidth), but such statistics are not always available today for other key resources. For instance, when multiple VMs on a machine are performing heavy memory copies, they contend with each other over the shared memory bus, which is hard to diagnose due to lack of suitable statistics. Another reason why contention is hard to diagnose is that its effects may only be indirectly felt. For instance, a VM trying to saturate a certain amount of network capacity may be unable to do so in the presence of competing memory intensive workloads, because the two sets of workloads contend for the memory bus (an empirical example is shown in Figure 3). Simply monitoring memory utilization or network packet drops does not reveal the root cause of contention.

The final factor that complicates diagnosis is that performance problems in a middlebox—arising due to, e.g., bugs—may *propagate* across a virtual cluster due to the chaining of middleboxes. The upshot is that the wrong middlebox may be identified as being the root cause of poor performance in a chain or cluster, and incor-

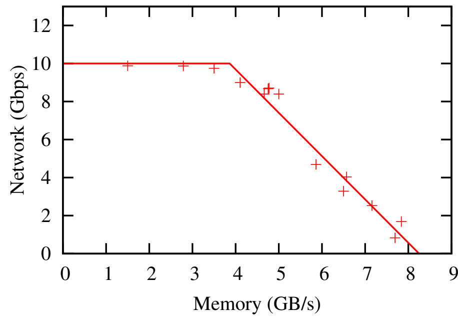


Figure 3: There are 8 VMs in a 8-core hypervisor with a 10Gbps NIC. Some of the VMs perform intensive memory copy operations, and the others send traffic to another machine by best effort. We vary the memory copy workload and measure the total throughput of the memory and the network respectively in each case. When memory throughput is low, the NIC capacity is fully saturated (10Gbps). However, when the memory throughput exceeds a threshold, every 1 GB/s increase of memory throughput causes 439 Mbps decrease of network throughput.

rect/inefficient counter-measures may be adopted. In Section 7 we will show such an example—a load balancer, a content filter and an HTTP server form a chain and the content filter writes logs to a shared NFS server (see Figure 12(d)). When the NFS server has a bug, performance degrades in the whole chain. In this case, it is challenging to find out the root cause middlebox.

2.4 Need for a New Approach

Existing troubleshooting techniques such as Anteater, HSA, NICE, Libra, etc. [31, 24, 14, 37] focus on the correctness of the network; thus, they cannot be used to detect performance problems. Tools such as ping and traceroute provide end-to-end tenant-level information which cannot pin-point root causes (e.g., those due to contention) accurately. Other diagnostic frameworks, such as NDB and VND [20, 35], collect traces on the datapath, either at network switches (NDB), or at vNICs or pNICs (VND). This can impact performance (of multiple tenants' virtual clusters) significantly. But, more importantly, such traces don't provide enough information to perform accurate root cause diagnosis; e.g., they cannot determine that packets were dropped due to memory contention.

Thus, we argue for a ground-up comprehensive framework for software dataplane diagnosis. We posit that given the complexity of the software data plane, a framework that performs *low-level, broad instrumentation* of various elements in the software dataplane, coupled with *suitable analytics*, can provide accurate and useful diagnoses.

3. PERFSIGHT OVERVIEW

We design and implement a system for software data plane diagnosis called PerfSight. Figure 4 shows the architecture of PerfSight. It has 3 high level components: an agent running on each physical server, a central controller, and a set of diagnostic applications atop the controller.

Diagnostic applications are developed using the controller-operator interfaces. These applications query the software data plane for various statistics; the controller then translates these to queries issued to agents running on the corresponding physical servers. The

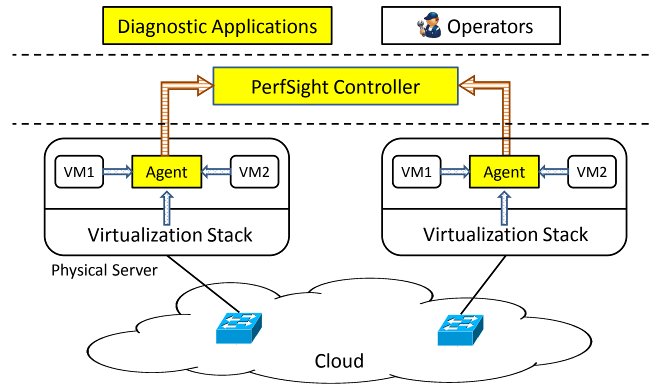


Figure 4: PerfSight Architecture

agents interrogate the relevant elements for statistics, and report the gathered statistics back to the application.

PerfSight uses a simple unified low-level interface for recording/retrieving statistics of the software data plane elements. This abstracts the complexity arising from the diversity of the data plane elements, and simplifies the task of designing analytics engines. The interface can be extended so that operators can easily enrich the set of statistics gathered.

PerfSight decouples data collection from analytics so that each can be improved or replaced by more advanced techniques. PerfSight's analytics applications work by performing *aggregate* diagnostics on the gathered statistics; more specifically, these applications jointly analyze data gathered across multiple instances of a middlebox, multiple middlebox VMs deployed on a single server, or multiple middleboxes deployed in one or more virtual clusters, to accurately pin-point the root cause of the observed performance problem.

In what follows, we first describe what statistics we collect and how they are collected. Then, we describe how these statistics are analyzed to obtain insights into software data plane performance problems.

4. STATISTICS GATHERING

The key insight we leverage to perform software dataplane diagnosis is to view the dataplane as a pipeline of low-level *elements*. By instrumenting at the element-level, we can obtain fine-grained statistics at key points in the software data plane. These then form the basis for interesting and accurate diagnostic applications. In what follows, we describe how we obtain element-level statistics on throughput, packet drops, and packet size. The key challenge lies in understanding where and how to instrument elements such that the data collection overhead is minimal.

4.1 Element Abstraction and Statistics

Figure 5 shows the elements and buffers in a virtualization environment that uses a Linux kernel, QEMU/KVM, and Open vSwitch (OVS). Each element receives packets from its predecessor, processes them based on internal logic, and delivers them to its successor. The exchange of packets between elements is often implemented by a buffer between them or by function calls. For example, in Figure 5, the pNIC driver and the packet processing routine (e.g., the NAPI routine in Linux) use the physical CPU (pCPU) backlog (a buffer) to exchange messages, whereas the NAPI routine and virtual switch use function calls.

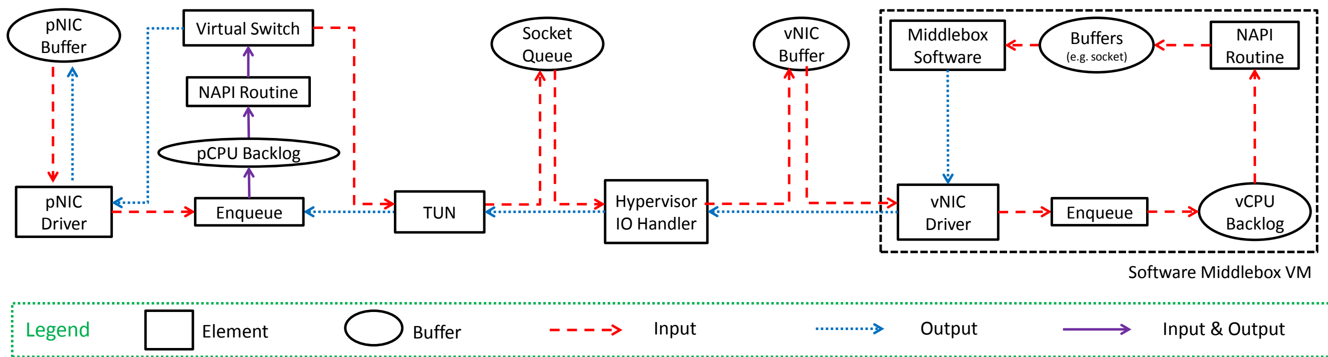


Figure 5: Elements in a Software Dataplane (QEMU/KVM, Open vSwitch, Linux)

Each element has *input methods* and *output methods*. For example, in the pNIC driver, the interrupt handler reads from the pNIC (or DMA mapped memory) and enqueues packets to the CPU backlog queue; the NAPI routine dequeues packets from the CPU backlog and calls the virtual switch frame handling function; the virtual switch frame handling function writes packets to a TAP's socket which is associated with a VM; and, the hypervisor I/O handler reads packets from the TAP and writes packets to the associated vNIC.

In each element, by analyzing its code, we can determine the code path that a packet traverses from input to output and possible code branches that might drop it. Statistics—e.g., how many bytes/packets were received/sent/dropped—of the traffic on the datapath between an element's input and output methods can be then recorded by instrumenting the code path and relevant branches. Another key statistic we gather is I/O time. This records the time spent on an I/O method (read and write). This can also be obtained in an easy and light-weight fashion by comparing the timestamps before and after the read/write function in an element. I/O time can reveal whether an element is facing starvation, and it plays a particularly crucial role in the diagnosis of propagation issues, as we show in Section 5.

In our current implementation, we perform the instrumentation task manually and exhaustively, but we believe it can be automated using program analysis. Interestingly, we find that many useful statistics actually don't require special instrumentation as they can be readily gathered, and hence statistics gathering imposes low overhead in this case. This is particularly true for elements residing in the host OS kernel. For example, packet count and byte count are basic statistics that already exist in most kernel elements. For other elements, e.g., those in the hypervisor and those inside middlebox software, we perform the instrumentation as indicated above; our extensive evaluation (Section 7) shows that the overhead is very low.

Our current prototype of PerfSight extracts/implements the 3 types of counters in each element: a packet counter, a byte counter, and an I/O time counter. The packet and byte counters are used by all types of elements while I/O time counter is only used for elements that interact with buffers. The counters accumulate as packets are processed.

Aggregate statistics such the instantaneous/average packet drop rates, throughput, and packet size can then be easily derived per element from these statistics. Operators can implement more complicated statistics at an element such as packet size distribution tracking if they can accept the resulting performance impact.

Our manual inspection of the code of many middleboxes, virtual switch implementations, and hypervisors has shown that several key entities in software data planes map to this abstraction. Thus, this abstraction is reasonable, and Section 5 shows that it is useful for diagnosis purpose. We show that our proposed statistics suffice to successfully diagnose the performance problems we discuss. In general, though, the set of statistics collected may fall short. That said, PerfSight itself is a framework, and, if necessary, the operator can add more statistics to suit her diagnostic needs.

4.2 Agent

An agent in each physical server gathers element statistics. To reduce overhead, the agent pulls counter values from elements only when required. Due to the diversity of elements, especially across the kernel, hypervisor and the middlebox software, the element/agent interfaces are tailor-designed for each type of element. In particular, we use custom APIs for elements in the kernel: e.g., in the NIC driver and the TAP, the counters are in net_device, and they can be accessed by reading the corresponding device file in the file system. When the NAPI routine processes packets in CPU backlogs, statistics are stored in the data structure softnet_data, and they can be accessed via /proc in the file system. In virtual switches, each switch rule has its own statistics and can be fetched by virtual switch control channels. For elements in the hypervisor and middleboxes, we design a common generic element-agent API.

After fetching statistics from elements, the agent provides statistics to the controller in a simple, unified format, which is as follows:

```
<TimeStamp, Element, (attr1, value1),
(attr2, value2), (attr2, value3)...>
```

That is, the response of a query to an element returns the timestamp, element ID, and a list of <attribute, value> pairs, where each pair describes a counter and its value at that timestamp. For example, a NIC driver's statistics can be described as

```
<t1, eth0, ("Rx bytes", v1),
("Tx bytes", v2), ... >
```

This interface is generic and simple for infrastructure operators to extend the statistics. When an operator needs to add a customized counter, she would need to add the counter into related elements, add logic in the agent to fetch this counter and add the result to the message that is returned to the controller.

4.3 Controller

The PerfSight controller delivers statistics requests and responses between operators and agents. When the operator requests a virtual

cluster's information, the PerfSight controller obtains the physical location of the related virtual elements and sends requests to the agents of those elements.

```
function GETATTR(tenantID,elementID,attributes)
    return vNet[tenantID].elem[elementID].attr[attributes]

function GETTHROUGHPUT(tid, eid)
    attr ← ["time", "bytes"]
    <t1, b1> ← GETATTR(tid, eid, attr)
    sleep(T)
    <t2, b2> ← GETATTR(tid, eid, attr)
    return (b2 − b1) / (t2 − t1)

function GETPKTLOSS(tid, eid)
    attr ← ["inPkts", "outPkts"]
    < b1_i, b1_o > ←GETATTR(tid, eid, attr)
    sleep(T)
    < b2_i, b2_o > ←GETATTR(tid, eid, attr)
    return (b2_i − b2_o) − (b1_i − b1_o)

function GETAVGPKTSIZE(tid, eid)
    attr ← ["bytes", "pktCount"]
    <b1, c1> ← GETATTR(tid, eid, attr)
    sleep(T)
    <b2, c2> ← GETATTR(tid, eid, attr)
    return (b2 − b1) / (c2 − c1)
```

Figure 6: Basic Utility Routines

Through controller/operator interfaces, the operator can query the attributes of each element. To retrieve the attribute of an element in a virtual cluster, the PerfSight controller first finds the physical location of the element, i.e., vNet[tenantID].elem[elementID], and then sends a request for attributes to the element's agent, and finally receives the requested attributes' values.

This interface can be used by an operator to perform basic monitoring of the performance of specific portions of her infrastructure. Figure 6 provides examples of basic utility routines for monitoring element states. As we argue next, interesting diagnostic applications can be built on these statistics.

5. DIAGNOSIS

The most interesting aspect of PerfSight is the set of diagnostic applications it enables. In what follows, we show how the statistics gathered in the aforementioned fashion can be used to develop algorithms that help resolve the performance problems central to software dataplanes that we mentioned in Section 2.

5.1 Detecting Contention and Bottleneck Middleboxes

Before we describe our algorithms, we give a few comments on how contention and bottlenecks manifest, and what makes detection hard.

Contention: VMs on the same physical server can contend for hardware or software resources in the virtualization stack. Returning to the example situation outlined in Sections 2.2 and 2.3, memory bandwidth was the shared hardware resource in contention; in yet other situations, hardware resources such as CPU and the NIC may be in contention. Likewise, in Figure 5, the shared datapath of input traffic and output traffic (purple/solid arrows) can be the software resource in contention; similar contention can arise for other software resources such as shared buffers/queues.

Algorithm 1 Detect Contention and Bottleneck

```
1: function FINDCONTENTIONANDMIDDLEBOX()
2:     elements ← { e | e ∈ virtualization stack }
3:     elemLoss← ∅
4:     for e ∈ elements do
5:         loss = GETPACKETLOSS(e)
6:         elemLoss.add(<e, loss>)
7:     return SORTBYLOSS(elemLoss)
```

However, not all contentions display explicit symptoms. Among hardware resources, high CPU, memory and network utilization are explicit symptoms of contention, but memory bandwidth contention does not have an explicit manifestation. Whether symptoms of contention for software resources can be observed depends on the software implementation. For example, OVS provides QoS and statistics by which contention can be directly identified, but not all elements in the virtualization stack have appropriate instrumentation to enable such direct identification. Thus, on the whole, resource contention in the virtualization stack is difficult to detect.

Bottleneck Middlebox: A bottleneck middlebox is constrained by the amount of resources allocated to it, but, as discussed in Sections 2.2 and 2.3, whether a middlebox is a bottleneck cannot be judged solely based on its resource utilization.

Accurate Detection: The main idea underlying our diagnostic application here is as follows: Elements in the virtualization stack deliver packets to each other via intermediate buffers or function calls, and they typically use *nonblocking I/O* in doing so. That is, if an element cannot write to its successor, or its target buffer is full, packets get dropped. Thus, we obtain the *packet loss of each element* in the software data plane, and use it to locate where VMs are contending for resources or whether a VM is under-provisioned along some resource.

The application is designed as shown in Algorithm 1. For each element in the virtualization stack, we use the utility routing GetPktLoss() to obtain its packet loss; we sort all elements by their packet loss. Finally, the element with most packet loss is returned. This application monitors all related elements in the virtual network, so its cost is proportional to the size of the virtual network.

Crucially, the location of packet loss reveals the possible resources that VMs are contending for or are in shortage of. To aid in this, we build a simple offline *rule book* that maps packet loss locations to specific resources that may be running low or facing contention.

We construct the rule book as follows: we set up a variety of experiments where VMs contend for different resources, and we exhaustively track possible packet loss locations (details are in Section 7.1). The result is summarized in Table 1. Some symptoms reveal the resource in contention directly; for example, if incoming traffic exceeds pNIC capacity, packets are dropped at the pNIC.

In some cases, different kinds of contentions can have the same symptoms making detection hard: e.g., contention on CPU and memory bandwidth both lead to VMs being unable to fetch packets from TUN to vNIC, causing TUNs to drop packets which is the only externally visible symptom. In such cases, the operator can combine this with other symptoms such as CPU utilization and NIC throughput to distinguish the specific root cause.

Bottleneck middlebox detection is similar: when a tenant's deployment is facing end-to-end performance problems (and the tenant complains about it), the operator first selects middleboxes with

Table 1: Resource in Shortage and Symptom Rule Book

Resource in Shortage	Packet Drop Location
CPU	TUN (aggregated)
Memory Space	pNIC Driver
Memory Bandwidth	TUN (aggregated)
Incoming Bandwidth	pNIC
Outgoing Bandwidth	Backlog Enqueue
pCPU Backlog	Backlog Enqueue
VM Bottleneck (CPU or Bandwidth)	TUN (individual)

high resource utilization and includes them in a "suspicious" set; in the degenerate case (e.g., when no high utilization is apparent) all of the tenants middleboxes could be included in this set. Then, we use our light-weight statistics to distinguish those middleboxes that are facing legitimate issues, such as packet drops, against those whose resources naturally run at a high utilization but are otherwise not bottlenecks (e.g., a video encoder). Thus, we can more accurately pin-point bottleneck middleboxes.

Contention and bottleneck can be distinguished based on whether loss is spread across multiple VMs (contention) or confined to on VM's software data path (bottleneck).

5.2 Combating Propagation

Performance problems, e.g., due to implementation bugs, in one middlebox in a chain may propagate to others, causing them to appear to perform poorly as well. In particular, as we show below, the other middleboxes may appear to have stalled reading/writing from/to the network or I/O devices. Given this, however, determining the root cause middlebox can be challenging.

In what follows, we show how the statistics gathered by PerfSight can be used to infer the specific *states* that different middleboxes are operating under—e.g., under/over-loaded and read/write-blocked—which can then be used to identify the root cause.

Analysis and key insights Before describing our algorithm for identifying the root cause middlebox, we first present the underlying insights in identifying the states of different middleboxes in a chain.

In virtual settings, middlebox software exchanges data with the guest OS. Consider some time interval t_{total}; from the middlebox's perspective, this time interval can be split across the middlebox performing input, processing, and output. Thus, we have

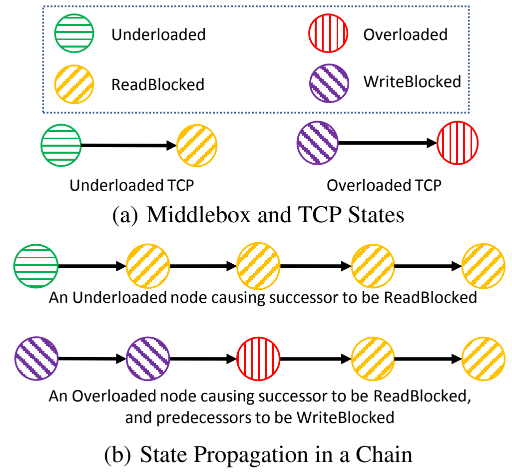
$$t_{total} = t_{input} + t_{process} + t_{output}.$$

The input/output time is constituted by block time, t_{block}, and memory copy time, t_{memcpy}; i.e.:

$$t_{input/output} = t_{block} + t_{memcpy}.$$

Input block time is the time spent waiting for new data, and output block time is spent waiting for buffers in kernel to be ready (e.g., waiting for TCP sending window to open up). Memory copy time is the time spent on copying data between user space and kernel space. When the middlebox software is blocked or processing, the kernel is waiting for packet arrival or transmiting data from/to vNIC.

The input function of middlebox software (e.g., recv() in TCP, FromDevice() in Click [27]) fetches data from kernel space to user space. If the buffer in kernel is ready with data, the input function is not blocked ($t_{block} = 0$), so the input method is as fast as a memory copy (which is at least 2 orders of magnitude faster



(a) Middlebox and TCP States

(b) State Propagation in a Chain

Figure 7: Middlebox States and Propagation

than network transmission). Assume that during the input function reading b bytes data from the kernel, the vNIC capacity is C and the memory copy speed is C_{mem}; in general, we have $C_{mem} \gg C$.

Thus, when there is no input blocking, we have:

$$t_{input} = t_{memcpy} = \frac{b}{C_{mem}} \ll \frac{b}{C}.$$

Thus, we can define a middlebox is *ReadBlocked* if it satisfies

$$\frac{b_{input}}{t_{input}} < C,$$

where b_{input} is the bytes read by the input function.

Similarly, when the output function (e.g., send() in TCP, ToDevice() in Click) sends data, the data (b bytes) would be copied from the user space to kernel space. As before, if the output function is not blocked,

$$t_{output} \ll \frac{b}{C}.$$

Thus, we can define a middlebox is *WriteBlocked* if it satisfies

$$\frac{b_{output}}{t_{output}} < C,$$

where b_{output} is the bytes written by the output function.

If two neighboring (in virtual topology) middleboxes use non-blocking I/O methods (e.g., packet-level processing) to exchange messages, their states do not impact each other. If they use TCP, TCP's congestion control makes the middlebox states influence each other, causing propagation. In particular, there are two possibilities: (1) the TCP sender does not send fast enough, causing the receiver to be ReadBlocked, (2) the receiver cannot receive or process data quick enough, causing the sender to be WriteBlocked. We define the sender in (1) as *Underloaded* and the receiver in (2) as *Overloaded* (Figure 7(a)). In a chain of TCP connections, an Underloaded source causes all its successors to be ReadBlocked, and an Overloaded middlebox causes all its predecessors to be WriteBlocked and successors to be ReadBlocked (Figure 7(b)).

Accurate Detection Of Root Cause(s) Based on the above analysis and insights, we develop the following algorithm (see Algorithm 2) underlying diagnostic application for determining the root cause middlebox(es): each middlebox's statistics are fetched (including input/output bytes/time and the middlebox vNIC capacity) (line 7). Then their respective states—under/overloaded, and

Algorithm 2 Locate Root Cause Middlebox

```
1:  function GETSTAT(tid, mb)
2:      attr←[“inBytes",“inTime",“outBytes",“outTime"]
3:      return GETATTR(tid, mb, attr)
4:  function GETROOTCAUSE(tid)
5:      midboxes←{mb|GetAttr(tid,mb,“type")=“middlebox"}
6:      cand ← midboxes
7:      for mb ∈ midboxes do
8:          C ← GETATTR(tid, mb, “Capacity")
9:          < b1_i, t1_i, b1_o, t1_o > ← GETSTAT(tid, mb)
10:         sleep(T)
11:         < b2_i, t2_i, b2_o, t2_o > ← GETSTAT(tid, mb)
12:         if (t2_i − t1_i > (b2_i − b1_i)/C) then
13:             mb.state = ReadBlocked
14:             cand←cand−GETSUCCESSOR(mb)−mb
15:         else if (t2_o − t1_o > (b2_o − b1_o)/C) then
16:             mb.state = WriteBlocked
17:             cand←cand−GETPREDECESS(mb) −mb
18:     return cand
```

read/write blocked—are computed based on the statistics (line 12-15). If a middlebox is ReadBlocked, all its successors are waiting for its data and are also ReadBlocked; we filter out the entire such chain of ReadBlocked middleboxes from the suspicious candidates (line 14). Similarly, a WriteBlocked middlebox and all of its predecessors can be filtered out (line 17). The remaining middleboxes in the candidate set are returned (line 18) as the plausible root cause middleboxes. In Figure 7(b), this algorithm would identify the green middlebox in the first chain and the red middlebox in the second chain as the root cause. This application monitors all middleboxes in a virtual network, so its cost is linear to the size of the virtual network.

6. IMPLEMENTATION

Most elements (e.g., NIC driver and virtual switch) already have their own counters and logs implemented—in these cases, PerfSight simply makes use of the existing ones; if some counters are missing in an element, we instrument the elements and add augmented counters. The communication channel between an element and the corresponding agent on the physical server is implemented specifically according to the element. In the following paragraphs, we will describe the implementation details at each key element that a packet goes through on the software dataplane. The environment we use to simulate the cloud is Linux kernel v3.20, Open vSwitch and QEMU [1].

For the *physical NIC*, the Linux kernel maintains a data structure called net_device which records received/sent bytes/packets. net_device can be accessed via the file system interface from userspace (e.g., ifconfig). The PerfSight agent uses this. In the *NAPI routine* which processes packets in CPU backlog queues, a data structure called softnet_data is maintained for each queue. softnet_data records packets dropped between the CPU backlog queue and the target callback function (i.e., virtual switch packet handler). This record is accessable from the /proc file system. In *virtual switches*, each rule has statistics for packets processed and dropped. The statistics are accessable via the OpenFlow control channel. A *TAP*'s transmit function enqueues a packet into a socket queue. The TAP has a net_device record to keep track of Rx/Tx byte/packet counts, which are accessible via file system. QEMU, which delivers packets from TAP socket to a vNIC data structure, does not have intrinsic statistics. Thus, we instrument it to obtain statistics in QEMU. We write these counters into logs and PerfSight fetches the counters' values from the logs. Inside the *middlebox VM* (in the guest OS kernel), the difference from the virtualization stack is that the NAPI routine delivers a packet from vCPU backlog queues to another buffer in the kernel (e.g., socket). The statistics records are then similar to the virtualization stack. Middlebox software reads packets/messages from the kernel via system calls. After a middlebox performs its own processing, it sends packets/messages out to the guest OS kernel via system calls. We instrument middlebox software to record Rx/Tx byte/packet counts and the time spent on reading from/writing to the kernel. In PerfSight, we use sockets between middlebox software and the agent corresponding to the physical server on which the middlebox is located to fetch these counters.

[1]Currently our solution is limited to this platform, but we believe PerfSight is a general solution to all platforms.

When a middlebox sends a packet out, the packet is delivered across several elements and finally put into the pCPU backlog. The write function to the kernel calls the vNIC transmit function, which causes an interrupt in QEMU. The interrupt handler in QEMU calls the transmit function in the TAP. The TAP transmit function enqueues the packets into the pCPU backlog queue. When dequeued, the packet is processed by the virtual switch and sent to another device's (another TAP or pNIC according to the packet's destination) transmit function. In each layer that the packet goes through, the instrumentation and communication with the agent is similar to the packet receiving datapath.

Discussion. We assume that the cloud provider offers the virtual network services and the NFV service, so they have the source code. PerfSight is a framework, and it defines interfaces between components. Even if some services (e.g. middlebox) are provided by third party, they can use the interface to provide statistics so that PerfSight can perform diagnosis.

7. EVALUATION

To evaluate PerfSight, we set up a cluster with Dell T5500 servers, each of which has 8 cores, 16GB memory and a 10Gbps NIC. Each server runs Ubuntu with Linux kernel 3.2. We use Open vSwitch, QEMU to set up the virtualization stack. We install middlebox software in VMs and route tenant traffic to traverse the VM to simulate NFV-like settings.

In what follows, we first validate the basic functionalities of PerfSight, and then show the effectiveness of using PerfSight by illustrating how it can diagnose performance problems, reflecting the examples outlined in Section 2. Finally we show that it imposes negligible overhead.

7.1 Functional Validation

In what follows, we experimentally illustrate that PerfSight works as expected, using the example of the packet loss function (Section 4.3).

In this experiment, we start 8 VMs on a physical machine. Two of these VMs function as middleboxes and the rest as tenant VMs. The middlebox software we use is that of a load balancer [1]. We start a handful of long-lived TCP flows that traverse the middlebox VMs, and we monitor their throughput. Over time, we inject various performance problems. All the while, we run PerfSight's packet loss function at various software data plane elements in the 8 VMs to identify if, when and where packet loss happens.

The results are shown in Figure 8. We plot the average throughput of the middlebox flows on the left y axis, and the packet drop counts on the right y axis. During the interval 10-20 seconds, we flood a large amount of packets into the physical machine (these are

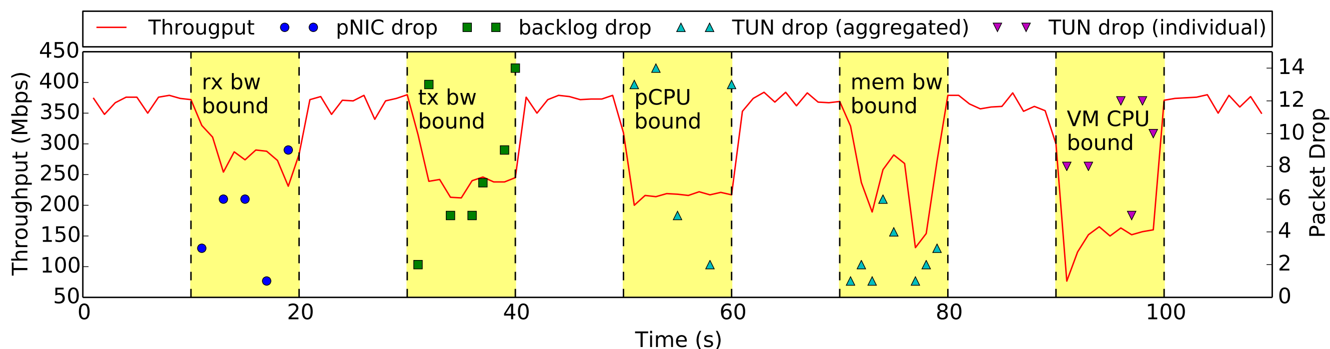


Figure 8: Throughput and Packet Drop in Virtualization Stack During Performance Problems

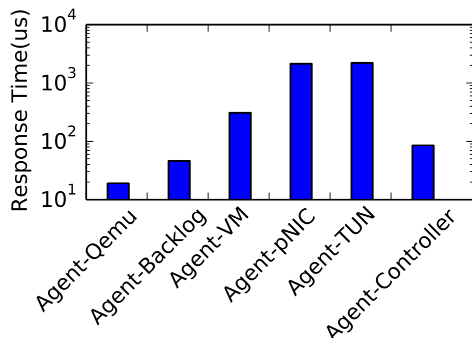
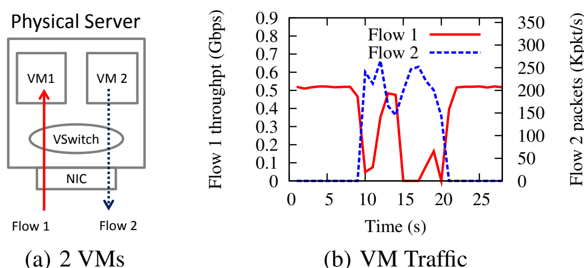


Figure 9: Response Time between Agent and other components



(a) 2 VMs (b) VM Traffic

Figure 10: CPU Backlog Queue Contention Detection Example

received by the non middlebox VMs). As such, the virtualization stack cannot clear the pNIC DMA buffer quickly enough, leading to lower TCP throughput for the middlebox VMs. What we noticed from PerfSight was that packets were dropped in the pNIC, which is as we expected (Table 1). During 30-40 seconds, the tenant VMs (non middlebox) flood a large amount of outgoing packets impacting middlebox traffic throughput. The flooded packets quickly fill up the CPU backlog first, leading to drops there for middlebox traffic, which is again as expected (Table 1). During 50-60 and 70-80 seconds, we make tenant VMs perform CPU intensive and memory access intensive workloads. As a result, the middlebox VMs do not acquire enough resources to process packets. This causes their packets to be accumulated and dropped at the TUN's socket buffer (which is the last buffer before entering VMs). When VMs are contending for resources, all VM's performance are impacted, so all VMs are dropping packets. While the above showed various forms of contention and interference, we now illustrate a single VM becoming a bottleneck. During 90-100 seconds, we start a CPU intensive workload inside one middlebox VM. From PerfSight, we note that only that VM drops packets at its associated TUN.

Response Time. The agent on each physical machine is the pivot of data collection and delivery. We measure how quickly the agent can exchange data with other components. As is shown in Figure 9, fetching statistics from network devices (e.g. TUN, pNIC) costs about 2ms, and all other components' statistics collection can be completed in 500us. This time granularity is fine enough to closely monitor the network states.

7.2 Diagnosis Accuracy

We now establish PerfSight's accuracy. We first conduct two experiments for detecting contention.

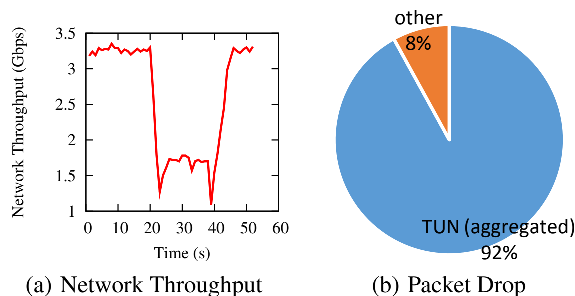
Detecting contention in virtualization stack (case 1). In Figure 5, we can observe that the pCPU backlog queue is exercised by multiple datapaths, which leads to the risk of it becoming a location where significant contention arises.

We set up two VMs – VM1 and VM2 – in a physical machine, and then make VM1 receive network traffic with a rate limit of 500Mbps. Roughly 10s into the measurement, we make VM2 send small packets as fast as it can. At this time, VM2's throughput decreases and oscillates; see Figure 10.

To diagnose this contention problem, PerfSight first checks if the VMs are overwhelming the NIC. VM2's peak sending rate (obtained using the GetThroughput() routine) is 250K packets per second (80Mbps); thus, the sum of the sending and receiving rates is well below the NIC capacity (1Gbps). PerfSight then checks packet drop counters at various elements in the virtualization stack. We found that the enqueue element in the virtualization stack (Figure 5) saw significant drops, and because outgoing bandwidth is not the problem (Table 1), the resource under contention ought to be the pCPU backlog queues (Table 1).

In a virtualized setup, both the incoming and outgoing packets are put into pCPU backlog queues first, and then forwarded during backlog processing. However, each CPU core's backlog queue length is limited to 300 packets. In our example, VM2 overwhelms the pCPU backlogs by flooding a large amount of small packets, and only a handful of VM1's packets can ever be enqueued into the backlog queue. This manifests as many more packets of VM1 arriving into the enqueue element and far fewer making it out.

Without PerfSight helping us identify where exactly packets are getting dropped and tying that back to the resource under con-



(a) Network Throughput (b) Packet Drop

Figure 11: Memory Bandwidth Contention Detection Example

tention (Table 1), it is difficult to determine the reason for VM1 and VM2 interfering with each other's performance.

Detecting contention in virtualization stack (case 2). Along the lines of the example Section 2, we simulated a machine with significant oversubscription in the virtualization stack. We start by running on a machine some number of VMs performing network transfers initially; their total network throughput is about 3.25Gbps as shown in Figure 11(a). At time 20s, another set of VMs start to access memory intensively. The total network throughput decreases to 1.7Gbps, because the two sets of VMs are contending for the memory bandwidth. We assume that the tenant running the former set of VMs wishes to diagnose this problem.

We observe that the physical machine is dropping packets at the network-intensive VMs' TUNs (Figure 11(b)). Thus, we infer that the machine's memory or outgoing bandwidth are overloaded (based on Table 1); at this time, we cannot distinguish which specific resource is experiencing contention. In response, wearing the operator's hat, we migrate some of the network-intensive VMs, and the network throughput recovers to the original 3.2Gbps.

Combating propagation. We validate our approach for identifying poorly-performing middleboxes in the face of propagation of problems where root cause cannot be obviously identified. The scenario we use is a multi-chain setting shown Figure 12(a): we deploy a load balancer (Balance [1]) and two content filter proxies (CherryProxy [2]) between clients and HTTP servers. We make the two content filter proxies output logs to a shared file system (NFS). All VMs' vNIC capacity are set to be 100Mbps.

In this virtual network, we simulate different cases and perform diagnosis; we find that our application always determines the root cause accurately. We conducted a variety of experiments where different points in this multi-chain are problematic and the issues can propagate arbitrarily through the chains, but for simplicity we focus on those exercising the VMs shown within the box shown by the dashed red line. For each experiment (figures b, c, and d) we show the performance metrics we derived for each middlebox (e.g., b/t_{in}, b/t_{out}), and the corresponding state we inferred for the middlebox.

In our first experiment, we make the client perform HTTP POSTs as fast as possible (with the idea of creating a bottleneck at the server within the dashed red box); we observe the state of middleboxes and determine that the load balancer and the content filter are WriteBlocked and the NFS server is Readblocked (Figure 12(b)). From this, our algorithm infers that server 1 is overloaded, identifying the true bottleneck.

In a second run, we have the client make HTTP POST requests at a slow rate. We monitor middlebox states and determine that other middleboxes are ReadBlocked, leading our algorithm to conclude that the client is Underloaded, which is indeed the case (Figure 12(c)).

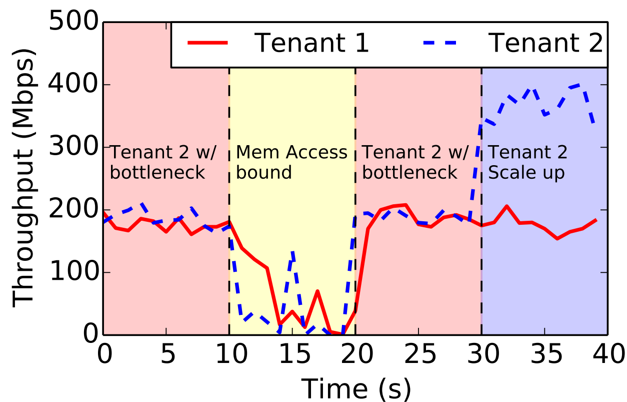


Figure 13: Throughput in Multi-tenant Experiment

As a final experiment, we inject an internal error (memory leak) into the NFS server,[2] causing it to become overloaded. The problem propagates through the chain middleboxes within the dashed red box, causing the load balancer and the content filter to be WriteBlocked. Using our algorithm, we exclude the ReadBlocked and WriteBlocked middleboxes and correctly determine that the NFS server is the bottleneck (Figure 12(d)).

7.3 Using PerfSight

We now consider a richer setting where we illustrate how we envision an operator using PerfSight. We show how an operator can detect and respond to contention and bottlenecks. We then describe propagation.

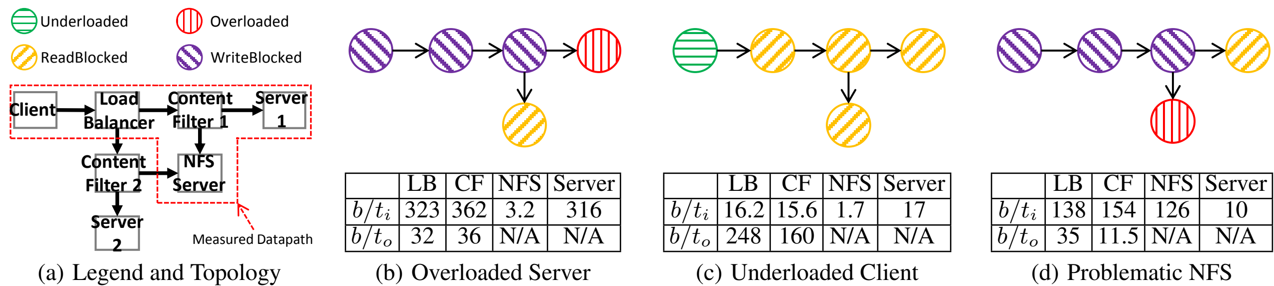
We use a setup with two tenants each with their own virtual network. Each network has a server, a load balancer proxy and a client with the client sending traffic to the server (Figure 14). We assume that the operator places the two load balancers in the same physical machine. We measure both tenants' throughput and show it in Figure 13.

Initially (0-10s), tenant 1 sends traffic at 180Mbps, while tenant 2 intends to send two flows at 360Mbps in total. However, tenant 2's load balancer can only process 200Mbps traffic, so tenant 2's total throughput is constrained by its load balancer. At this point, using PerfSight the operator identifies that the TUN of load balancer 2 is dropping packets and it is in an Overloaded state. Thus, the operator has identified tenant 2's bottleneck. Between 10 and 20s, the operator introduces a management task that happens to be memory access intensive into the physical machine. The operator finds that both tenants' load balancer VMs are impacted, they're dropping packets at their TUNs, and they're in ReadBlocked state. The operator identifies memory bandwidth over-subscription and responds by migrating the memory intensive task elsewhere. Thus, we see that the throughput immediately reverts to the original value (20-30s). However, this still does not address the bottleneck that tenant 2 is facing at its load balancer. After determining that tenant 2 is bottlenecked at its load balancer, the operator scales it out and reroutes half of tenant 2's traffic to the new instance. The total throughput increases to 360Mbps and no packets are dropped at the load balancers.

7.4 Overhead and Scalability

Overhead. Among the counters we implemented, packet counts/bytes are simple/low overhead because each time they are incremented by

[2] CentOS bug track 7267.



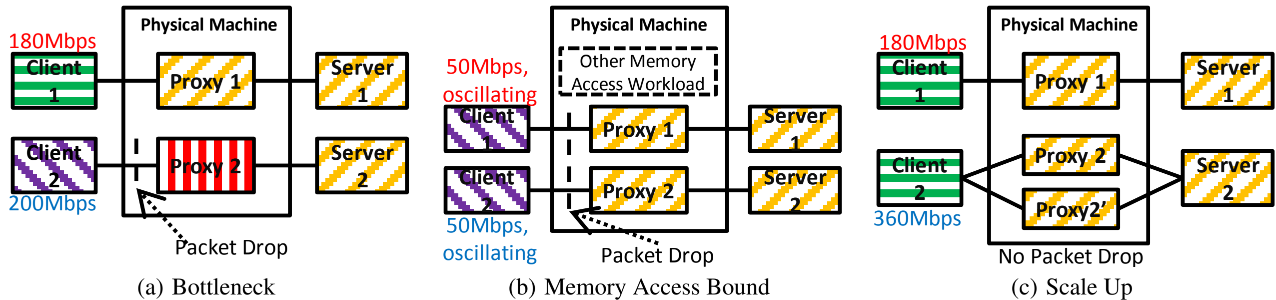
	LB	CF	NFS	Server
b/t_i	323	362	3.2	316
b/t_o	32	36	N/A	N/A

	LB	CF	NFS	Server
b/t_i	16.2	15.6	1.7	17
b/t_o	248	160	N/A	N/A

	LB	CF	NFS	Server
b/t_i	138	154	126	10
b/t_o	35	11.5	N/A	N/A

(a) Legend and Topology (b) Overloaded Server (c) Underloaded Client (d) Problematic NFS

Figure 12: Root cause detection in the face of propagation (Unit: Mbps, b: bytes, t_i: t_{input}, t_o: t_{output})



(a) Bottleneck (b) Memory Access Bound (c) Scale Up

Figure 14: A multi-tenant experiment illustrating how an operator can use PerfSight to aid her tenants

Table 2: Throughput with/without Time Counters
1: Blocked, without Counters, 2: Blocked, with Counters
3: Overloaded, without Counter, 4: Overloaded, with Counters.
Each experiment is repeated for 100 times

Experiment	1	2	3	4
Mean μ (Mbps)	42.02	41.79	499	490.2
Variance σ^2	4.57	4.92	1554	1281

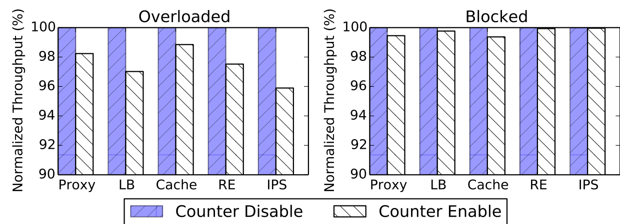


Figure 15: Time Counter Overhead within Middleboxes

a value; time counters are a bit more complex because they need to get time twice, and accumulate the difference. We first measure the time spent to update a counter. We find that simple counters consume 3ns per update, while a timer counter consumes 0.29us per update in our testbed. Even with the maximum throughput of 10Gbps and 1500 bytes MTU, each packet takes 1.2us to traverse an element. Thus, the simple counters impose 10^{-3} smaller overhead, which is negligible.

However, the time counters do have an impact on performance. If an element is overloaded and CPU bound, time counters would cost extra CPU cycles, which degrades the element's performance. To delve into this, we build a virtual network with an HTTP server, an HTTP client and a proxy. The client uploads data to the server via the proxy. In this setting, if we limit the client's sending rate (on its vNIC), the proxy will be ReadBlocked; if not, TCP would saturate the virtual link and cause the proxy to become Overloaded. We compare the throughput between the cases where the proxy is/not instrumented with time counters. The result is shown in Table 2. We can conclude that the impact of the time counters is very small (under 2% in terms of throughput; we also found the average latency impact to be under 1.5% (not shown for brevity)).

We repeat the similar experiments on different kinds of middleboxes [10, 1, 6], and show the results in Figure 15. In all kinds of middleboxes, the impact is less than 5%.

Another source of overhead is from polling these counters. The overhead is shown in Figure 16, and it still very small. Even if we poll them every 100ms (which is generally sufficient for the kind of diagnostics we wish to run), the CPU utilization is less than 0.5%, which is negligible.

Scalability. Our experimental set up is indeed small, but we believe that our system's response time and overhead will stay small even at larger set ups due to the following reasons: (1) The statistics gathering is distributed in each element, so it's not likely to become a bottleneck. (2) The diagnostic applications' complexity is O(n) where n is the number of involved elements. (3) Cloud operators can aggregate tenants' tickets to diagnose if they have elements overlapping with each other.

8. RELATED WORK

The systems and networking communities have developed a variety of diagnostic tools and frameworks. In this section, we place PerfSight in the context of these prior frameworks.

sFlow, NetFlow, ndb, tcpdump, X-Trace and SNAP [20, 9, 17, 36] provide various ways to collect key information from the network (and from network attached systems). Unfortunately, these techniques don't provide fine-grained information about which spe-

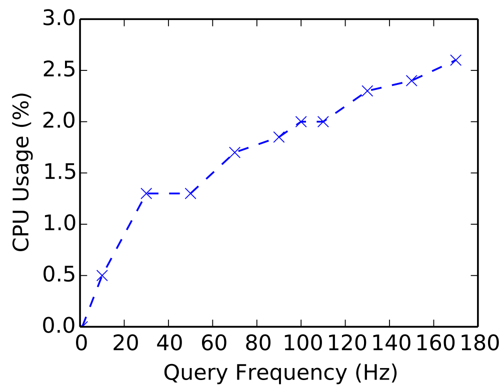


Figure 16: Query Frequency and CPU Usage

cific parts of the software data plane are facing/imposing problems. As a result, they are far less useful in diagnosing and fixing subtle software data plane issues. For example, compared with SNMP which only defines the format of management messages, PerfSight has statistics gathering, framework (interfaces) and diagnostics applications, so PerfSight is a complete diagnostic solution. Furthermore, some of these techniques (tcpdump and SNAP) can impose a high overhead.

Recent solutions such as header space analysis (HSA), NICE, Libra, Anteater, NetPlumber, VeriFlow [25, 24, 14, 37, 31, 23] are also useful for diagnosing virtual networks. However, they are not very useful toward diagnosing performance problems. In contrast with general network reachability issues that the above tools are very capable at helping with, performance problems are much more tricky: they are often ephemeral; they can manifest unexpectedly, e.g., under specific corner-case workloads; and, they can arise due to a wide variety of different issues (mainly because of the complexity of the software data plane), as a result of which doing root cause detection, and fixing the observed problems can both be very difficult.

For traditional networks, NeST [32] was proposed as a mechanism for instrumenting the network stack toward diagnosis. NeST uses a dependency graph to find stalled components. However, the virtualized network is more complex than the traditional network. Not only does it have more components, but also its components are much more heterogeneous (i.e., blocking/nonblocking middleboxes). Simple dependency analysis also does not suffice for some key performance problems (e.g., it does not aid in bottleneck middlebox detection).

There are some other works in the general distributed systems world that try to correlate symptoms of performance problems and their root causes. Some of them use end-to-end information to infer the internal components' states (i.e., Sherlock, netMedic, netDiagnoser [11, 22, 16, 18]). Unfortunately, these techniques are not as accurate as our direct instrumentation-based approach.

Some of the aforementioned prior systems (SCORE, MaxCoverage, shrink and codebook [28, 29, 21, 26]) are based on bipartite graph models (with symptoms on one side and possible causes on the other, and weight assignment used to determine the most likely root causes of problems observed). However, such models cannot be used when the problem propagates or in situations where components interfere with each other, both of which can arise very frequently in the context of software data planes.

On the whole, we believe that PerfSight is a novel, first-of-a-kind and important contribution to the space of diagnosis frameworks. Its particular focus on software data planes makes it invaluable for future network function virtualization set ups.

9. CONCLUSION

With the advent of NFV, data planes are becoming increasingly complex, involving a variety of software packet processing elements. In this paper, we argued that these new "software data planes" are susceptible to subtle performance problems that don't occur (or are infrequent) in traditional hardware-based data planes. We argued that diagnosing these problems is difficult because no existing tool or system provides the right level of information to tease apart various potential causes of the observed degradation. To this end, we present a system, PerfSight, a ground-up approach for extracting comprehensive low-level information regarding packet processing performance of the various elements in the data plane and for conducting rich analysis on the information gathered. Through careful experiments, we show that our framework can result in accurate detection of the root causes of performance problems in software data planes, and it imposes very little overhead.

Acknowledgement

We would like to thank Anja Feldmann (our shepherd) and the anonymous reviewers for their valuable feedback. We also thank Vyas Sekar for his comments in the early stage of this project. This work is supported in part by National Science Foundation (grants CNS-1345249, CNS-1302041 and CNS-1330308) and the Wisconsin Institute on Software-Defined Datacenters of Madison.

10. REFERENCES

[1] Balance: the open source load-balancer and tcp proxy. `http://www.inlab.de/balance.html`.
[2] Cherryproxy: a filtering http proxy extensible in python. `http://www.decalage.info/python/cherryproxy`.
[3] http://rightscale.com.
[4] http://tools.ietf.org/html/rfc3954.html.
[5] Path mtu discovery. `http://tools.ietf.org/html/rfc1191`.
[6] Snort: Open source network intrusion prevention. `http://www.snort.org`.
[7] www.openvswitch.org.
[8] www.sflow.org.
[9] www.tcpdump.org.
[10] A. Anand, V. Sekar, and A. Akella. Smartre: an architecture for coordinated network-wide redundancy elimination. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 87–98. ACM, 2009.
[11] P. Bahl, R. Chandra, A. Greenberg, and S. Kandula. Towards highly reliable enterprise network services via interence of multi-level dependencies. In *SIGCOMM*, 2007.
[12] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
[13] T. Bu, N. Duffield, F. L. Presti, and D. Towsley. Network tomography on general topologies. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 21–30, New York, NY, USA, 2002. ACM.

[14] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A nice way to test openflow applications. In *NSDI*, 2012.

[15] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng, et al. Network functions virtualisation–introductory white paper. In *SDN and OpenFlow World Congress*, 2012.

[16] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *CoNEXT*, 2007.

[17] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stocia. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.

[18] A. Gember, A. Akella, T. Benson, and R. Grandl. Stratos: Virtual middleboxes as first-class entities. In *UW-Madison, Technical Report*, 2012.

[19] A. Goel, C. Krasic, K. Li, and J. Walpole. Supporting low latency tcp-based media streams. In *Quality of Service, 2002. Tenth IEEE International Workshop on*, pages 193–203. IEEE, 2002.

[20] N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.

[21] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: A tool for failure diagnosis in ip networks. In *MineNet workshop*, 2005.

[22] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *SIGCOMM*, 2009.

[23] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2014.

[24] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[25] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *HotSDN*, 2012.

[26] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A coding approach to event correlation. In *IM*, 1995.

[27] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 2000.

[28] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Ip fault localization via risk modeling. In *NSDI*, 2005.

[29] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *INFOCOM*, 2007.

[30] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[31] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[32] J. N. McCann. Automating performance diagnosis in networked systems. In *PHD thesis*, 2012.

[33] R. Potharaju and N. Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *IMC*, 2013.

[34] C. D. P. Ramanathan and D. Moore. Packet dispersion techniques and capacity estimation.

[35] W. Wu, G. Wang, A. Akella, and A. Shaikh. Virtual network diagnosis as a service. In *SoCC*, 2013.

[36] M. Yu, A. Greenberg, D. Maltz, J. Rexford, and L. Yuan. Profiling network performance for multi-tier data center applications. In *NSDI*, 2011.

[37] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *NSDI*, 2014.