Trumpet: Timely and Precise Triggers in Data Centers

Masoud Moshref (USC) Minlan Yu (USC) Ramesh Govindan (USC)
Amin Vahdat (Google, inc.)

Abstract

As data centers grow larger and strive to provide tight performance and availability SLAs, their monitoring infrastructure must move from passive systems that provide aggregated inputs to human operators, to active systems that enable programmed control. In this paper, we propose Trumpet, an event monitoring system that leverages CPU resources and end-host programmability, to monitor every packet and report events at millisecond timescales. Trumpet users can express many *network-wide events*, and the system efficiently detects these events using *triggers* at end-hosts. Using careful design, Trumpet can evaluate triggers by inspecting every packet at full line rate even on future generations of NICs, scale to thousands of triggers per end-host while bounding packet processing delay to a few microseconds, and report events to a controller within 10 milliseconds, even in the presence of attacks. We demonstrate these properties using an implementation of Trumpet, and also show that it allows operators to describe new network events such as detecting correlated bursts and loss, identifying the root cause of transient congestion, and detecting short-term anomalies at the scale of a data center tenant.

CCS Concepts

•Networks → End nodes; Network monitoring; Data center networks;

Keywords

Network Event Monitoring; End-host Monitoring

1. INTRODUCTION

Data center network management tasks range from fault diagnosis to traffic engineering, network planning, performance diagnosis, and attack prevention and require network monitoring. Commercial network monitoring tools (*e.g.,* SNMP, NetFlow) produce highly aggregated or sampled statistics (*e.g.,* per port count in SNMP, sampled NetFlow records) at relatively coarse time-scales (seconds to minutes), often provide input to dashboards designed to present aggregate network views to human operators, and require humans to interpret their output and initiate network control.

As data centers evolve to larger scales, higher speed, and higher link utilization, monitoring systems must detect *events* (such as transient congestion or server load imbalance) more precisely (inspecting every packet), at fine time-scales (on the order of milliseconds) and in a programmable fashion (so that the set of events can be dynamically determined). In this paper, we argue for a shift of monitoring infrastructure from *passive* to *active*: Instead of presenting aggregated network views to human operators, monitoring infrastructure should both allow for interactive drill down based on current conditions and for automated reaction to pre-defined, fine-grained packet events at the scope of entire fabrics rather than individual links or servers.

The challenge in achieving this goal is scale along many dimensions: the number of endpoints, the high aggregate traffic in the network, and, as data center networks move to support large numbers of tenants and services, the number of events of interest. To address this challenge, we leverage the relatively plentiful processing power at end-hosts in a data center. These end-hosts are programmable, already need to process every packet once, and can monitor all packets without sampling using their powerful CPUs.

Contributions. In this paper, we present the architecture and implementation of Trumpet: an event monitoring system in which users define network-wide events, a centralized controller installs *triggers* at end-hosts where triggers test for local conditions, and the controller aggregates these signals and tests for the presence of specified network-wide events. This architecture demonstrates the benefits of pushing, to end-hosts, the substantial scalability challenges faced by datacenter-scale per-packet monitoring. The key to Trumpet's efficacy is carefully optimized per-packet processing inline on packets being demultiplexed by a software switch.

We demonstrate the benefits of Trumpet with the following contributions. First, Trumpet introduces a *simple network-wide event definition language* which allows users to define network-wide events by specifying a *filter* to identify the set of packets over which a *predicate* is evaluated; an *event* occurs when the predicate evaluates to true. Users can set a predicate's time and flow granularity. Despite this simplicity, the language permits users to capture many interesting events: identifying if flows trigger congestion on other flows, when aggregate traffic to hosts of a service exceeds a threshold, or when connections experience a burst of packet loss.

Our second contribution is scaling event processing at end-hosts while still being able to detect events at timescales of

SIGCOMM '16, August 22-26, 2016, Florianopolis , Brazil
 ISBN 978-1-4503-4193-6/16/08...$15.00
DOI: http://dx.doi.org/10.1145/2934872.2934879

a few milliseconds. Upon receiving an event definition, the central controller installs *triggers* corresponding to events at end-hosts. A trigger determines whether the user-specified event has occurred at an end-host; if so, the end-host sends the trigger's output to the controller. The controller collects the trigger results from individual end-hosts to determine if the event definition has been satisfied across the network. The key challenge is to support thousands of triggers at full line rate without requiring extra CPU cores on a software switch, and without (a) dropping any packet, (b) missing any events, and (c) delaying any packet by more than a few μs.

To achieve these properties, Trumpet processes packets in two phases in the end-hosts: A match-and-scatter phase that matches each incoming packet and keeps per 5-tuple flow statistics; and a gather-test-and-report phase which runs at each trigger's time granularity, gathers per-trigger statistics, and reports when triggers are satisfied. We also design and implement a suite of algorithms and systems optimizations including caching to reduce computation, pre-fetching and data structure layout to increase memory access efficiency, careful use of virtual memory, and a queue-length adaptive scheduling technique to steal cycles from packet processing. Our design degrades gracefully under attack, and scales naturally to higher-speed NICs likely to be deployed in datacenters in the next few years.

Our third contribution is an *evaluation of Trumpet* across a variety of workloads and event descriptions. We evaluated Trumpet running on a single core with 4K triggers and (a) 64-byte packets on a 10G NIC at line rate and (b) 650-byte packets on a 40G NIC at line rate. Trumpet sustains this workload without packet loss, or missing a single event. More generally, we characterize the feasible set of parameters, or the *feasibility region*, in which Trumpet finishes every sweep, and never loses a packet. Each of our optimizations provides small benefits, but because at full line-rate we have almost no processing headroom, every optimization is crucial. Trumpet can report network-wide events within 1ms after the first trigger is matched at one end-host. Its matching complexity is independent of the number of triggers, a key reason we are able to scale the system to large numbers of triggers. Finally, Trumpet degrades gracefully under attack: with the appropriate choice of parameters (which can be determined by an accurate model), it can sustain a workload in which 96% of the packets are DoS packets, at the expense of not being able to monitor flows of size smaller than 128 bytes.

Fourth, we demonstrate the expressivity of Trumpet with three use cases: a) pacing traffic, using a tight control loop, when that traffic causes a burst of losses; b) automatically identifying flows responsible for transient congestion and c) detecting, using a network-wide event, services whose combined volume exceeds a threshold.

Getting precise and accurate visibility into data centers is a crucial problem, and Trumpet addresses an important part of this space. In practice, a spectrum of solutions is likely to be necessary, including approaches (Section 8) that use NICs and switches. NIC offloading saves end-host CPU and can monitor the flows that bypass the hypervisor (*e.g.,* using SR-IOV and RDMA). Other schemes that mirror packets or packet headers to a controller, where an operator or a script can examine headers or content, can help drill down on events triggered by Trumpet. As an aside, these schemes might need to process packets at high rates at the controller, and Trumpet's processing optimizations might be applicable to these.

2. THE CASE FOR TRUMPET

Today's monitoring infrastructure is designed for human monitoring, so coarse time scales and high levels of aggregation are sufficient. Modern data centers need real-time, fine-grained, and precise monitoring to feed a variety of control systems. In this section, we elaborate on this observation, which motivates the design of Trumpet.

Problems of today's monitoring systems. Network management systems in data centers rely on detecting network *events*. An event often indicates a network condition that may require a corrective action, mostly through a computer-assisted reaction. Events are often defined in terms of packets dropped, delayed, or delivered by the network, and can range in topological scope from a specific flow, to traffic aggregates (*e.g.,* all network traffic to a service), and in temporal scope from being extremely short-lived to long-lived events.

Traditional monitoring systems only provide coarse-grained views of the network at larger time scales, which are sufficient for humans to understand network state, but may be insufficient to capture events precisely or at fine time-scales. For example, SNMP provides per port counters every few minutes, too coarse-grained for traffic engineering or performance diagnosis. OpenFlow provides counters for aggregated flows (due to limited TCAM sizes [40]) and reports the updated counters every few seconds [13, 46], which cannot capture sub-second traffic events. sFlow [54] uses packet sampling which cannot capture transient events (*e.g.,* transient packet losses), track connection states (*e.g.,* congestion window), and correctly estimate link load [48]. These shortcomings in monitoring systems can lead to significant loss of network availability: a traffic surge in Google's Compute Engine was detected 3 minutes after causing 90% packet loss between two regions [22].

Moreover, today's monitoring systems do not scale well to larger networks with higher capacities and higher utilization. Higher link speed and larger scales mean more packets to monitor; higher network utilization requires more timely event reporting, because delayed reports of an outage can affect larger traffic volumes. Unfortunately, higher utilization also leaves fewer network resources for monitoring. For example, the reporting frequency of OpenFlow counters is inversely proportional to the number of connections managed (increasing new connections from 150 to 250 per second requires reducing reporting frequency from once per second to once per 5 seconds [13]). As a result, the precision of a network monitoring system can suffer at higher scales and utilization and therefore adversely impact the goal of achieving service level objectives and high utilization, especially in data center networks.

Data centers need precise, fine time-scale event monitoring. Data centers need novel kinds of event monitoring capabilities to capture a variety of network misbehaviors (*e.g.,* misconfiguration, transient looping, anomalies, drops) and as input for network management decisions (*e.g.,* traffic engineering, load balancing, VM migration). We describe a few examples here and list more in Section 3.

Identify losses caused by traffic bursts. Traffic bursts are common in data centers and can improve application performance [23, 58]. For example, NIC offloading [31] sends packets in batches to reduce processing overhead, and distributed file systems read and write in bulk to maximize disk throughput. However, bursty traffic may cause losses when they traverse shallow buffered switches [51], which significantly affects application performance. To improve performance, it may be necessary to detect lost packets (*e.g.,* by retransmissions) and packets in bursts (*e.g.,* by tracking packet timestamps), and identify their correlations (*e.g.,* by correlating packet sequence numbers). All these detections are not possible using packet sampling or flow level counters.

Identify root cause of congestion. Transient network congestion (*e.g.,* incast [8]) is hard to diagnose. For example, a MapReduce reducer may see significant performance degradation caused by network congestion, even though its aggregate demand may be well below switch capacity. This is because another application sending through the same switch can trigger transient incast losses, increasing the reducer's processing time and therefore the job finish time. Such congestion is often caused by short bursts of traffic at short timescales (10 ms). This may not be detectable on aggregate counters in today's switches (which have $>$ 1s granularity). To diagnose this behavior, it is important to identify TCP flows with high loss and correlate them with heavy hitters going through the bottleneck.

Monitor server load balance and load burst. Maintaining the service-level agreements (SLAs) for different applications requires careful provisioning and load balancing in cloud services [44] because imperfect load balance and short request bursts can lead to long tail latency [28, 26]. A good way to track service load is to monitor the network traffic [5]. If we can identify volume anomalies in short timescales, we can identify inefficient load balancing across servers, provisioning issues or DDoS attacks on some servers. For example, operators can query whether the long tail latency is because the service sees bursts of requests, or more than 50% of VMs of a service see a traffic surge as a result of a DDoS attack.

Trumpet. These kinds of event detection are beyond the capabilities of today's deployed systems. In this paper, we consider a qualitatively different point in the design space of monitoring systems. We ask: Does there exist a design for a monitoring system which can detect and report *thousands* of events within a few *milliseconds*, where event detections are *precise* because the system processes every packet (rather than, say, sampling), and event specifications can be *flexible* (permitting a range of spatial and temporal scopes of event definition)? Such a monitoring system would be especially useful for automatic diagnosis and control at millisecond timescales on timely reporting of traffic anomalies, fine-grained flow scheduling, pacing, traffic engineering, VM migration and network reconfigurations.

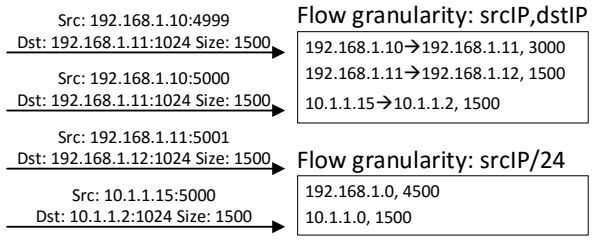


Figure 1: Flow granularity in Trumpet event definition

3. DEFINING EVENTS IN TRUMPET

Users of Trumpet define *events* using a variant of a match-action language [38], customized for expressing events rather than actions. An event in this language is defined by two elements: a *packet filter* and a *predicate*. A *packet filter* defines the set of packets of interest for the specific event, from among all packets entering, traversing, or leaving a data center (or part thereof) monitored by Trumpet. Filters are expressed using wildcards, ranges or prefix specifications on the appropriate packet header fields. If a packet matches multiple filters, it belongs to set of packet for each of the corresponding events. A *predicate* is simply a Boolean formula that checks for a condition defined on the set of packets defined by a packet filter; when the predicate evaluates to true, the event is said to have occurred. A predicate is usually defined over some aggregation function expressed on *per-packet variables*; users can specify a *spatio-temporal granularity* for the aggregation.

Elements of the event definition language. Each packet in Trumpet is associated with several variables. Table 1 shows the variables used for use cases discussed in this paper. It is easy to add new variables, and we have left an exploration of the expansion of Trumpet's vocabulary to future work.

The predicate can be defined in terms of mathematical and logical operations on these variables, and aggregation functions (`max`, `min`, `avg`, `count`, `sum`, `stddev`) of these variables. To specify the granularity at which the aggregation is performed, users can specify (a) a `time_interval` over which the predicate is to be evaluated, and (b) a `flow_granularity` that specifies how to bucket the universe of packets, so that the aggregation function is computed over the set of packets in each bucket within the last `time_interval`.

The `flow_granularity` can take values such as `5-tuple`[1] or any aggregation on packet fields (such as `srcIP`, `dstIP` and `srcIP/24`). Figure 1 shows two examples of counting the volume of four flows at two different flow granularities (`srcIP`, `dstIP` and `srcIP/24`). At `srcIP`, `dstIP` granularity, the first two flows are bucketed

[1] 5-tuple fields include source and destination IP, source and destination port, and protocol. Although all examples here use 5-tuple, Trumpet flow granularity specifications can include other packet header fields (*e.g.,* MAC, VLAN).

Variable	Description
volume	the size in bytes of the packet payload
ecn	if the packet is marked with ECN
rwnd	the actual receiver advertised window
ttl	the packet's time-to-live field
rtt	the packet round-trip time (as measured from the returning ACK)
is_lost	if the packet was retransmitted at least once
is_burst	if the packet occurred in a burst (packets from a flow with short inter-arrival time)
ack	latest ack
seq	maximum sequence number
dup	an estimate of the number of bytes sent because of duplicate acks

Table 1: Packet variables

Example	Event
Heavy flows to a rack with IP range 10.0.128.0/24	dstIP=10.0.128.0/24, sum(volume)>125KB, 5-tuples, 10ms
Large correlated burst & loss in any flow of a service on 10.0.128.0/24	srcIP=10.0.128.0/24, sum (is_lost & is_burst)>10%, 5-tuples, 10ms
Congestion of each TCP flow	Protocol=TCP, 1 - (ack - ack_lastepoch + dup) / (seq_lastepoch - ack_lastepoch) >0.5, 5- tuples, 10ms
Load spike on a service at 10.0.128.0/24 port:80	(dstIP=10.0.128.0/24 and dstPort=80), sum(volume)>100MB, dstIP/24, 10ms
Reachability loss between service A on 10.0.128.0/24 to B 10.20.93.0/24	(srcIP=10.0.128.0/24 and dstIP=10.20.93.0/24), sum(is_lost)>100, (srcIP and dstIP), 10ms
Popular service dependencies for a tenant on 10.0.128.0/20	srcIP=10.0.128.0/20, sum(volume)>10GB, (dstIP/24 and srcIP/24), 1s

Table 2: Example Event definitions (Filter, Predicate, Flow_granularity, Time_interval)

together, while the third flow is in a separate bucket. At srcIP/24 granularity, the first three flows are placed in the same bucket. The flow_granularity specification makes event descriptions much more concise. For example, to find chatty VM pairs in a service, the operator can define an event at the granularity of srcIP,dstIP without needing to explicitly identify the actual IP addresses of the communicating pairs. In some cases, these IP addresses may not be known in advance: *e.g.,* all the users of a datacenter internal service. Additional examples of event definitions with different flow_granularity are given below.

Example event definitions. Table 2 gives examples of some event definitions, based on the discussion in Section 2.

The first example is to find heavy hitter flows that send a burst of packets to a rack (say 10.0.128.0/24). The user of Trumpet would define the IP block as the filter, define a predicate that collects the total number of bytes and checks if the sum exceeds a threshold (say 125 KB as 10% of a 1G link capacity) in a *10 ms time interval* and *5-tuple flow granularity*. Notice that this event definition is expressed at the scale of a rack (say 10.0.128.0/24), and can flag *any* 5-tuple flow whose behavior matches the predicate.

The second example detects a large correlated burst of losses in any flow of a service whose servers use a given IP block (say 10.0.128.0/24). The user defines a predicate that counts the number of packets for which is_lost[2] and is_burst [31] is simultaneously true and checks when the count passes the threshold (the predicate is shown in Table 2). Events with flow_granularity definitions over multiple flows can be used for coflow scheduling [9], guiding SDN rule placement [41] and estimating the demand of FlowGroups for rate limiting [33].

The next event in the table detects when a TCP connection experiences congestion. Specifically, the event predicate checks if a connection does not receive most of the acks for which it was waiting from the beginning of a measurement *epoch* whose duration is the time_interval of the predicate. This event tracks seq, ack and dup[3] variables for *all TCP flows in both directions* for the current and previous epochs. It computes the size of acked bytes in the current epoch using the ack and dup of the other side of connection and compares it against outstanding bytes in the last epoch based on the ack and seq. Similar events can be used to detect the violation of bandwidth allocation policies [4] at short timescales, debug new variants of TCP congestion control algorithms, and detect unresponsive VMs to ECN marks and RTT delays.

A slightly different example (the fourth in Table 2) detects if there is a load spike on a distributed service (as defined by an IP address block 10.0.128.0/24). In this example, the predicate evaluates to true if the total volume of traffic to all destination servers within this IP block over a 10 ms time interval exceeds this threshold. For this event, the flow_granularity is dstIP/24: the whole service. [4]

Beyond the above examples, Trumpet can also be used on other management tasks such as: (a) diagnosing reachability problems between the VMs of two services by counting the total packet losses among any source and destination pair (similar queries can find black holes [23, 58] and transient connectivity problems in middleboxes [47]) and (b) finding popular service dependencies of a tenant by checking if any set of servers in a specific source IP/24 collectively send more than 10GB per second of traffic to a set of servers in a destination IP/24 (a service IP range) (useful, for example, to migrate their VMs to the same rack [27] or put load balancing rules on ToRs with chatty services [19]).

[2]The number of retransmissions over-estimates the number of lost packets. A more accurate solution is more complex and needs careful RTO estimation [3]

[3]Duplicate acks show that a packet is received although it is not the one expected. dup increases by 1460 bytes for each dup-ack and decreases based on acked bytes for each regular ack.

[4]To IP addresses of a service cannot be expressed in a single range, Trumpet allows event specifications with multiple filters.

4. AN OVERVIEW OF TRUMPET

Trumpet is designed to support thousands of dynamically instantiated concurrent events which can result in thousands of triggers at each host. For example, a host may run 50 VMs of different tenants each communicating with 10 services. To support different management tasks (*e.g.,* accounting, anomaly detection, debugging, *etc.*), we may need to define triggers on 10 different per-packet variables (*e.g.,* in Table 1) and over different time-intervals and predicates.

Trumpet consists of two components (Figure 2): the Trumpet Event Manager (TEM) at the controller and a Trumpet Packet Monitor (TPM) at each end-host.

Users submit event descriptions (Section 3) to the TEM, which analyzes these descriptions statically and performs the following sequence of actions. First, based on the filter description and on network topology, it determines which end-hosts should monitor the event. Second, it generates *triggers* for each end-host and installs the triggers at the TPM at each host. A trigger is a customized version of the event description that includes filters and predicates in flow and time granularity. However, the predicate in a trigger can be slightly different from the predicate in the corresponding event definition because the event description may describe a network-wide event while a trigger captures a host local event. Third, TEM collects trigger *satisfaction* reports from host TPMs. TPMs generate satisfaction reports when a trigger predicate evaluates to true. For each satisfied trigger, TEM may poll other TPMs that run triggers for the same event in order to determine if the network-wide predicate is satisfied.

An important architectural question in the design of Trumpet is where to place the monitoring functionality (the TPMs). Trumpet chooses end-hosts for various reasons. Other architectural choices may not be as expressive or timely. In-network packet inspection using measurement capabilities at switches lack the flexibility to describe events in terms of per-packet variables; for example, it is hard to get visibility into variables like loss, burst, or RTT at the granularity of individual packets since switches see higher aggregate traffic than end-hosts. Alternatively, sending all packets to the controller to evaluate event descriptions would result in significantly high overhead and might not enable timely detection.

Trumpet's TPM monitors packets in the hypervisor where a software switch passes packets from NICs to VMs and vice versa. This choice leverages the programmability of end-hosts, the visibility of the hypervisor into traffic entering and leaving hosted VMs, and the ability of new CPUs to quickly (*e.g.,* using Direct Data I/O [30]) inspect all packets at fine time-scales. Finally, Trumpet piggybacks on the trend towards dedicating CPU resources to packet switching in software switches [24]: a software switch often runs all processing for each packet in a single core before sending the packet to VMs because it is expensive to mirror packets across cores due to data copy and synchronization overhead [14]. For exactly the same reason, and because it needs complete visibility into all traffic entering or leaving a host, TPM is co-located with the software switch on this core. This trend is predicated on increasing core counts in modern servers.

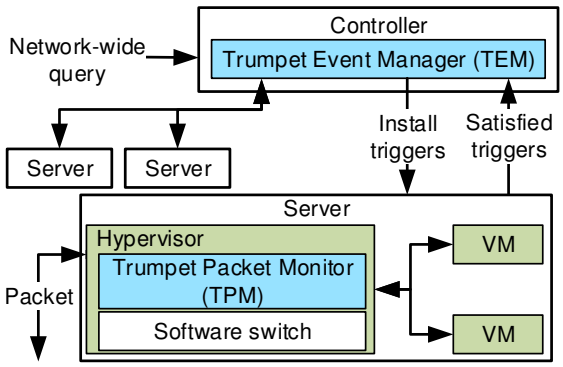


Figure 2: Trumpet system overview

Trumpet is designed for data centers that use software packet demultiplexing, leveraging its programmability for supporting complex event descriptions, extensibility to new requirements, and resource elasticity to handle load peaks. However, Trumpet can also be used in two ways in data centers where the NIC directly transfers traffic to different VMs (*e.g.,* using kernel bypass, SR-IOV, or receive-side scaling). The first is mirroring traffic to the hypervisor. New NICs allow mirroring traffic to a separate queue that is readable by the hypervisor, using which Trumpet can evaluate trigger predicates. Although this has CPU overhead of processing packets twice (in the hypervisor and VMs), this still preserves the goal of reducing packet latency at VMs. Moreover, because trigger evaluation is not on the packet processing path, Trumpet is not constrained by bounds on packet delay, so it may not need a dedicated core. We have evaluated Trumpet with the mirroring capability in Section 7.2. The second is NIC offloading. With the advent of FPGA (*e.g.,* SmartNIC[17]) and network processors at NICs [6], Trumpet can offload some event processing to NICs. For example, it can offload trigger filters in order to selectively send packet headers to the hypervisor. As NIC capabilities evolve, Trumpet may be able to evaluate simpler predicates within the NIC, leaving CPU cores free to perform even more complex processing tasks (*e.g.,* understanding correlations across multiple flows) for which some of our techniques will continue to be useful.

Trumpet also depends upon being able to inspect packet headers, both IP and TCP, so header encryption could reduce the expressivity of Trumpet.

The design of both TPM and TEM present significant challenges. In the next two sections, we present the design of these components, with a greater emphasis on the systems and scaling challenges posed by precise and timely measurement at the TPM. We also discuss the design of TEM, but a complete design of a *highly scalable* TEM using, for example, techniques from [45, 58], is beyond the scope of this paper.

5. TRUMPET PACKET MONITOR

The primary challenge in Trumpet is the design of the Trumpet Packet Monitor (TPM). TPM must, at line rate: (a) determine which trigger's filter a packet matches, (b) update statistics of per-packet variables associated with the trigger's predicate and (c) evaluate the predicate at the specified time granularity, which can be as low as 10 milliseconds. For a

10G NIC, in the worst-case (small packet) rate of 14.8 Mpps, these computations must fit within a budget of less than 70ns per packet, on average.

5.1 Design Challenges

The need for two-phase processing. Our event definitions constrain the space of possible designs. We cannot perform all of the three steps outlined above when a packet is received. Specifically, the step that checks the predicate must be performed at the specified time-granularity. For instance, a predicate of the form (#packets for a flow during 10ms is *below* 10), can only be evaluated at the end of the 10ms interval. Thus, TPM must consist of two phases: a computation that occurs per-packet (*phase one*), and another that occurs at the predicate evaluation granularity (*phase two*). Now, if we target fine time-scales of a few milliseconds, we cannot rely on the OS CPU scheduler to manage the processing of the two phases because scheduling delays can cause a phase to exceed the per-packet budget discussed above. Hence, Trumpet piggybacks on the core dedicated to software switching, carefully managing the two phases as described later.

Strawman Approaches. To illustrate the challenges underlying the TPM design, consider two different strawman designs: (a) *match-later*, in which phase one simply records a history of packet headers and phase two matches packets to triggers, computes statistics of per-packet variables and evaluates the predicate, and (b) *match-first*, in which phase one matches each incoming packet to its trigger and updates statistics, and phase two simply evaluates the predicate.

Neither of these extremes performs well. With 4096 triggers each of which simply counts the number of packets from an IP address prefix at a time-granularity of 10ms, both options drop 20% of the packets at full 10G packet rate of 14.8 Mpps. Such a high packet rate at a server is common in NFV applications [16], at higher bandwidth links and in certain datacenter applications [36]. A monitoring system cannot induce loss of packets destined to services and applications, and packet loss also degrades the efficacy of the monitoring system itself, so we consider these solutions unacceptable. Moreover, these strawman solutions do not achieve the *full expressivity of our event definitions*: for example, they do not track losses and bursts that require keeping per flow state (Section 3). Modifying them to do so would add complexity, which might result in much higher losses at line rates.

Design Requirements. These results and the strawman designs help identify several requirements for TPM design. First, both match-first and match-later, even with a state-of-the-art matching implementation, induce loss because the overhead of matching the packet to a filter often exceeds the 70ns budget available to process a single packet at the rate of 14.8 Mpps 64-byte packets on a 10G NIC. This implies that more *efficient per-packet* processing (phase one) is necessary. We also impose the requirement that the system should *degrade gracefully under DoS attacks*, since an attack can disrupt monitoring and attacks on cloud providers are not uncommon [39]. Moreover, with match-first, any delay in processing a packet will add queueing delay to processing subsequent packets: a monitoring system should, ideally, impose *small and bounded delay*.

Second, we observed that match-first scales worse than match-later because it incurs 3× higher TLB and 30% higher cache misses (match-later exhibits much more locality because it performs all packet related actions at once). Thus, *cache and TLB efficiency* is a crucial design requirement for being able to scale trigger processing to full line rate.

5.2 Our Approach

In Trumpet, TPM splits the monitoring functions into the following two phases: (a) *Match and scatter* in which incoming packets are matched to a 5-tuple flow[5] (the finest `flow_granularity` specification allowed), which stores per packet counts/other statistics per flow (this scatters statistics across flows), and (b) *Gather-test-and-report*, which runs at the specified trigger time granularity, gathers the statistics to the right `flow_granularity`, evaluates the predicate and reports to TEM when the predicate evaluates to true.

Partitioning the processing in this manner enables Trumpet to satisfy the requirements discussed in the previous subsection:

- As we discuss below, this partitioning permits the design of data structures that promote cache and TLB efficiency, which, as we discuss above, is crucial for performance. Furthermore, the balance of CPU overhead between the two phases permits efficient packet processing, without compromising expressivity: in the first phase, we can minimize matching overhead by caching lookups.
- Small and bounded delays can be achieved by cooperatively scheduling these phases (which avoids synchronization overhead): the second phase is *queue-adaptive* and runs only when the NIC queue is empty.
- Match and scatter per 5-tuple flows allows us to track per flow statistics such as loss and burst and separating that from the gather-test-and-report phase let us compute the statistics once and share these among multiple triggers matching the same flow, but at different flow granularities.
- This partitioning also localizes the two processing bottlenecks in the system: packet processing and gathering statistics. As we discuss later, this allows us to design safeguards for the system to degrade gracefully under attacks, avoiding packet loss completely while maintaining the fidelity of statistics gathering.

5.3 Data Structures in Trumpet

Trumpet uses four data structures (Figure 3): a *flow table* to store statistics of per-packet variables for the flow, a *trigger repository* contains all the triggers, a *trigger index* for fast matching, and a *filter table* for DoS-resilience. We describe the latter two data structures in detail later.

The *flow table* is a hash table, keyed on the flow's 5-tuple, that keeps, for each flow, only the statistics required for the triggers that match the flow (Figure 3). Thus, for example, if

[5] Unlike match-first, which matches against triggers defined at multiple granularities.

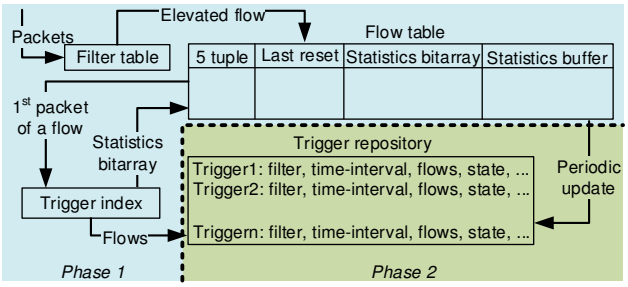


Figure 3: Two-stage approach in Trumpet

a flow only matches a single trigger whose predicate is expressed in terms of volume (payload size), the flow table does not track other per-packet variables (loss and burst, round-trip times, congestion windows, *etc.*). The variables can be shared among triggers, and the TEM tells the TPM which variables to track based on static analysis of event descriptions at trigger installation. The flow table maintains enough memory to store most per-packet statistics, but some, like loss and burst indicators are stored in dynamically allocated data structures. Finally, the TPM maintains a statically allocated overflow pool to deal with hash collisions. The *trigger repository* not only contains the definition and state of triggers, but also tracks a list of 5-tuple flows that match each trigger. In a later section, we discuss how we optimize the layout of these data structures to increase cache efficiency.

Algorithm 1: Processing packets in Trumpet

```
1  Function processPackets(packetBatch)
2      foreach Packet p do
3          prefetch(p)
4      foreach Packet p do
5          if p.flow != lastpacket.flow then
6              p.hash = calculateHash()
7              prefetchFlowEntry(p)
8      foreach Packet p do
9          if p.flow != lastpacket.flow then
10             e = flowTable.find(p)
11             if e == NULL then
12                 e = flowTable.add(p)
13                 triggers = triggerMatcher.match(p)
14                 e.summaryBitarray =
                       bitarray(triggers.summaries.id)
15         if e.lastUpdate < epoch then
16             e.resetSummaries()
17         e.updateSummaries(p)
18         forwardPacket(p)
```

5.4 Phase 1: Match and Scatter

In this phase, Algorithm 1 runs over every packet. It looks up the packet's 5-tuple in the flow-table and updates the per-packet statistics. If the lookup fails, we use a *matching* algorithm (lines 11-14) to match the packet to one or more triggers (a flow may, in general, match more than one trigger).

Fast trigger matching using tuple search. Matching a packet header against trigger filters is an instance of multi-dimensional matching with wildcard rules. For this, we build a *trigger index* based on the tuple search algorithm [52], which is also used in Open vSwitch [46]. The tuple search algorithm uses the observation that there are only a limited number of patterns in these wildcard filters (*e.g.,* only 32 prefix lengths for the IP prefix). The trigger index consists of multiple hash tables, one for each pattern, each of which stores the filters and the corresponding triggers for each filter. Searching in each hash table involves masking packet header fields to match the hash table's pattern (*e.g.,* for a table defined on a /24 prefix, we mask out the lower 8 bits of the IP address), then hashing the result to obtain the matched triggers. Tuple search memory usage is linear to the number of triggers, and its update time is constant.

Performance Optimizations. Since packet processing imposes a limited time budget, we use several optimizations to reduce computation and increase cache efficiency. For performance, software switches often read a *batch* of packets from the NIC. When we process this batch, we use two forms of *cache prefetching* to reduce packet processing delay: (1) prefetching packet headers to L1 cache (lines 2-3) and (2) prefetching flow table entries (lines 4-7). Data center applications have been observed to generate a burst of packets on the same flow [31], so we cache the result of the last flow table lookup (lines 5, 9). To minimize the impact of TLB misses, we store the flow table in huge pages. In Section 7, we demonstrate that each optimization is critical for Trumpet's performance.

5.5 Phase 2: Gather, Test, and Report

This phase gathers all statistics from the flow table entries into the `flow_granularity` specified for each trigger (recall that the flow-table stores statistics at 5-tuple granularity, but a trigger may be defined on coarser flow granularities, like `dstIP/24`). Then, it evaluates the predicate, and reports all the predicates that evaluate to true to the TEM.

The simplest implementation, which runs this entire phase in one sweep of triggers, can result in large packet delays or even packet loss, since packets might be queued or dropped in the NIC while this phase is ongoing. Scheduling this phase is one of the trickier aspects of Trumpet's design.

At a high-level, our implementation works as follows. Time is divided into *epochs* of size T, which is the greatest common factor of the time granularities of all the triggers. Trumpet supports a T as small as 10 milliseconds. In each flow table entry, we double-buffer the statistics (like volumes, loss, burst, *etc.*): one buffer collects statistics for the odd-numbered epochs, another for even-numbered epochs. In the i-th epoch, we gather statistics from the $i-1$-th epoch. Thus, double-buffering gives us the flexibility to interleave trigger sweeps with packet processing.

We schedule this gathering sweep in a *queue-adaptive* fashion (Algorithm 2). When the NIC queue is empty, we run a sweep step for a bounded time (Algorithm 3). Because of Trumpet's careful overall design, it is always able to stay ahead of incoming packets *so that these sweeps are never starved* (Section 7). This bound determines the delay imposed by the measurement system, and can be configured. In our experiments, we could bound delay to less than 10 μs (Figure 4). Each invocation of this algorithm processes some number

Algorithm 2: Trumpet main loop

```
1  Function mainLoop(timeBudget)
2      if time to sweep then
3          startSweep()
4      if last sweep is not finished then
5          sweep(timeBudget)
6      while Packets in NIC queue do
7          processPackets(batches of 16 packets)
```

of triggers from the trigger repository (lines 4-5 in Algorithm 3). This processing essentially gathers all the statistics from the flow entries (recall that each trigger entry has a list of matched flow entries).

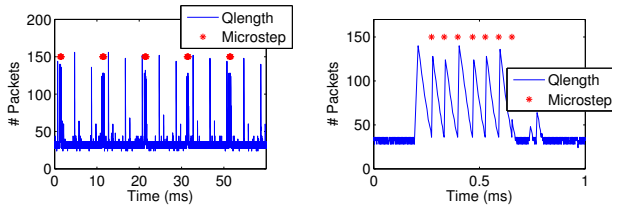
Algorithm 3: Periodic triggers sweeping

```
1  Function sweep(timeBudget)
2      b = timeBudget / 10
3      foreach t = nextTrigger() do
4          foreach t.FlowList do
5              flowNum = processFlowList(t)
6              t.flowNum += flowNum
7              b = b - (t.avgUpdateTime × flowNum)
8              if b ≤0 then
9                  if passedTime ≥ timeBudget then
10                     saveSweepState()
11                     updateAvgUpdateTime(passedTime)
12                     return
13                 b = timeBudget / 10
14         if epoch - t.lastUpdate ≥ t.timeInterval then
15             if t.condition(t) then
16                 report(t)
17             reset(t)
18             t.lastUpdate = epoch
19     updateAvgUpdateTimes(passedTime)
```

Once all the flows for a trigger have been processed, the algorithm tests the predicate and reports to the TEM if the predicate evaluates to true (lines 14-18 in Algorithm 3). However, while processing a trigger, the processing bound may be exceeded: in this case, we save the sweep state (line 10), and resume the sweep at that point when the NIC queue is next empty. For each trigger, during a sweep step, instead of computing elapsed time after processing each flow entry, which can be expensive (∼100 cycles), we only compute the actual elapsed time when the estimated elapsed time exceeds a small fraction of the budget (lines 7-13). The estimate is based on the number of flows processed for this trigger so far.

Our approach assumes we can complete sweeping all of the triggers within one epoch (10ms): in Section 7.2, we demonstrate that we can achieve this for 4K triggers on one core for a 10G NIC with small packets at full line rate even under DoS attack, and show that removing some of our optimizations (discussed below) can result in *unfinished sweeps*. We also demonstrate that, in many cases, shorter epoch durations cannot be sustained in at least one current CPU.

Performance Optimizations. We lazily reset counters and remove old flows. In each flow table entry, we store the epoch number when the entry was reset. Then, when we update an entry or read its data during trigger sweep, if the stored epoch number does not match the current, we reset the statistics (lines 15-16 in Algorithm 1).



(a) Multiple sweeps (b) Zoomed in View

Figure 4: Queue-adaptive sweep scheduling strategy is effective at keeping queue sizes below 150 packets or 10 μs.

Trumpet also incorporates several memory optimizations for this phase. Trigger entries are stored contiguously in memory to facilitate hardware cache prefetching. We store the trigger repository in a huge page to reduce TLB misses and store the list of flow entries that match a trigger in a chunked linked list [43] to improve access locality. Each chunk contains 64 flow entries, and these are allocated from a pre-allocated huge page, again for TLB efficiency. For triggers with many flows, chunking reduces the cost of pointer lookups comparing to a linked list of flows at the cost of slightly higher memory usage.

Our final optimization involves efficiently gathering statistics at the required flow granularity. For example, to support a trigger that reports if the volume of traffic to a host (A) from any source IP (at flow-granularity of source IP) is larger than a threshold, we must track the volume per source IP across flows. We can track this with a hash table per trigger, but this adds memory and lookup overhead. Instead, we dynamically instantiate a new instance of the trigger: when a packet from source IP X matches the trigger with a filter dstIP=A, we create a new trigger with filter srcIP=X and dstIP=A.

5.6 Degrading Gracefully Under DoS Attacks

It is important for Trumpet to be robust to DoS attacks that aim at exhausting resources in the monitoring system and either cause packet loss, or prevent trigger sweep completion (which would prevent accurate detection of events). The expensive steps in Trumpet are matching new flows against triggers in the match-and-scatter phase and updating a trigger based on its flow list in the gather-test-and-report phase.

We assume the attacker knows the design of the system. To attack the system by triggering the largest possible number of expensive match operations, the attacker should send one minimal-sized packet per flow. With this, sweeps might not complete, so trigger reports might not be correctly reported. To mitigate this, when we detect unfinished sweeps, we impose a *DoS threshold*: matching is invoked only on flows whose size in bytes exceeds this threshold in the filter table, and flows will be removed from the flow table if they send fewer bytes than the threshold as new flows collide with them in the flow table or triggers read them at sweep. The DoS threshold can be predicted by profiling the cost of each packet processing operation and each matching operation (Section 6). This design comes at the cost of not monitoring very small flows: we quantify this tradeoff in Section 7.

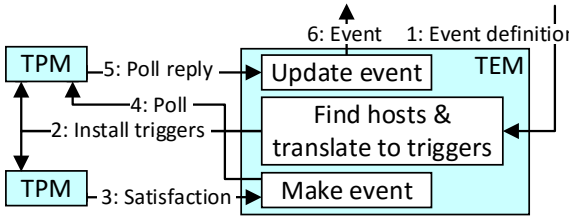


Figure 5: TEM interactions with TPMs

5.7 Summary

Trumpet balances several conflicting goals: event expressivity, tight processing and delay budgets, and efficient core usage. It achieves this by carefully partitioning trigger processing over two phases (match-and-scatter and gather-test-and-report) to keep data access locality per packet processing, keep per flow statistics efficiently, and access trigger information once per epoch. To make match-and-scatter phase efficient, Trumpet uses tuple search matching, NIC polling (DPDK), batching packets, cache prefetching, huge pages (fewer TLB misses) and caching last flow entry. To minimize processing delay during gather-test-and-report, we proposed a queue-adaptive multi-step sweep. We optimized the sweep to finish within the query time interval using lazy reset (bringing less data into cache), accessing data in a linear fashion (leveraging cache prefetching), checking the time less often, chunking flow entries (fewer pointer jumps) and using huge pages (fewer TLB misses). While some of these are well-known, others such as the adaptive sweep, our two-phase partitioning, and our approach to DDoS resiliency are novel. Moreover, the combination of these techniques is crucial to achieving the conflicting goals. Finally, our experiences with cache and TLB effects and data structure design can inform future efforts in fast packet processing.

6. TRUMPET EVENT MANAGER

Trumpet Event Manager (TEM) translates network-wide events to triggers at hosts, collects satisfied triggers and statistics from hosts, and reports network-wide events. Figure 5 shows the detailed process using an example: Consider an event expressed as a predicate over the total traffic volume received by a service on two hosts where the predicate is true if that quantity exceeds a threshold of 10Mbps. In step 1, TEM statically analyzes the event description to determine the trigger predicate, finds the hosts to install the triggers on based on the event filter defined by the service IP addresses and the mapping of IP addresses to hosts. Then, it divides the threshold among the triggers and installs them in step 2. For the example, we set the threshold for each host as half of the event threshold (5Mbps). If neither of the host triggers exceed 5Mbps, their sum cannot be larger than 10Mbps. In step 3, TPMs at hosts measure traffic and send trigger satisfaction messages to TEM specifying the event and epoch when evaluating triggers in the gather-test-and-report phase.

In step 4, upon receiving the first satisfaction report for the event, TEM polls the other hosts for their local value of the quantity at that epoch. For our example, a TPM may have sent a satisfaction with value 7Mbps, thus TEM asks for the value at the other TPM to check if its value is above 3Mbps or not. In step 5, TPMs respond to the controller polls after finishing phase 2 when they can steal cycles from packet processing; TPM allows triggers to keep the history of a few last epochs to answer polls. Finally in step 6, TEM evaluates the event after receiving all poll responses. For this to work, TEM relies on time synchronized measurement epochs; the synchronization accuracy of PTP [29] should be sufficient for TEM.

Network wide queries. The approach of dividing a network-wide threshold for an aggregate function can accommodate many kinds of aggregate functions. For example, it is possible to design thresholds for triggers to bound the error on any convex function on the average of quantities distributed among hosts [50] or their standard deviation [18]. Beyond these, Trumpet can also support other kinds of network-wide queries. For example: a) Gather statistics from many events: To find if 50% of VMs (or a certain number of VMs) of a tenant receive traffic exceeding a threshold, we add events for each VM in Trumpet, gather their trigger satisfaction every epoch and report if more than 50% of the related events happened. b) Drill down based on events: TEM can install events conditionally, for example, install a heavy hitter detection event only when another event (*e.g.,* loss) happens. c) Estimating network-wide statistics: We can monitor standard deviation of traffic volume to a set of VMs with a bounded error, ε by defining two events over the standard deviation ($std < std_{old} - \varepsilon$ and $std > std_{old} + \varepsilon$) and updating them accordingly if one of them is satisfied. d) Relative predicates: By feeding the estimate of average and standard deviation to another event, we can detect outlier VMs that receive more than $k\times$ standard deviation above the average.

40G and beyond. Although Trumpet needs only one core to process 14.8Mpps small packets on 10G links, or full line rate 650 byte packets on 40G links, at 100G and/or with smaller packet sizes, multiple cores might be necessary. To avoid inter-core synchronization, TEM runs independent TPMs on each core and treats them as independent entities. Some synchronization overhead is encountered at the TEM, but that is small as it is incurred only when one of triggers is satisfied. Assuming all packets from a flow are usually handled by the same core, this ensures TPMs can keep the state of a flow correctly. Our network-wide use-case in Section 7.1 demonstrates this capability. It is also possible to design, at each host with multiple TPMs, a *local-TEM* which performs similar aggregation at the host itself with minimal synchronization overhead: this can reduce polling overhead and speed up event detection at the TEM.

DoS resiliency. TEM allows operators to set a DoS threshold such that flows whose size are below that threshold are not processed. This reduces matching overhead, allowing Trumpet to degrade gracefully under DoS attacks. TEM calculates the threshold based on a model that profiles offline, using the set of defined triggers (Eq. 1), the TPM processing costs in the end-host system configuration. Trumpet processing costs include: a) packet processing and checking the filter table, T_P b) matching, T_M c) updating the flow table, T_U and d) sweeping, T_S. The goal is to find the maximum threshold

value that keeps total Trumpet processing time, T, below 1s in each second. We can find $T - T_P$ by profiling the free time of the CPU in an experiment that only forwards the traffic with smallest packets that do not pass DoS threshold. Matching overhead per flow (*Match(#patterns)*) is a linear function of # filter patterns with coefficients that can be calculated offline. Similarly, we compute maximum time to update per flow statistics (*Update*) and the time to update a trigger based on a flow (*Sweep*) offline.[6] The factor 1.5 in Eq. 4 is because the sweep can run in multiple steps while new flows may arrive. As a result, triggers may keep flow entries for both flows that came last epoch and current epoch. Thus, the sweep process may process 50% more entries per trigger on average. We evaluate the accuracy of our model in Section 7.2.

$$T = T_P + T_M + T_U + T_S \quad (1)$$

$$T_M = Match(\#patterns) \times \left(\frac{rate_{dos}}{threshold} + \frac{rate_{good}}{\text{avg pkt per flow}}\right) \quad (2)$$

$$T_U = Update \times rate_{good} \quad (3)$$

$$T_S = 1.5 \times Sweep \times \text{max trigger per flow} \times \frac{rate_{good}}{\text{avg pkt per flow}} \quad (4)$$

TEM Scalability and TPM performance. As we show in Section 7, TEM can process nearly 16M trigger satisfaction reports per second per core. If necessary, we can scale TEM even more by sharding events on different servers or by reducing the frequency of polling if TEM divides the threshold unequally based on the history of trigger satisfaction at each host [10]. To avoid delaying packets at end-hosts, TPM uses non-blocking sockets. To reduce poll response latency, TPM batches socket reads and writes and poll processing. TEM can, if necessary (left to future work), reduce TPM overhead by slicing event filters to minimize the number of trigger patterns for matching, and by time multiplexing triggers on hosts by (un)installing them over time.

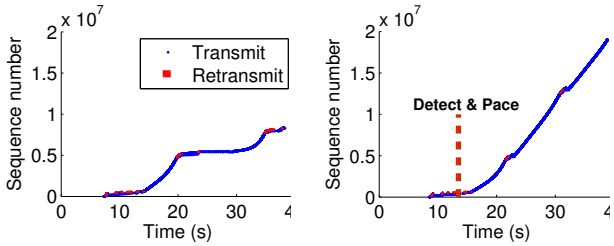
7. EVALUATION

In this section, we evaluate Trumpet's expressivity and performance using an implementation of Trumpet's TPM and TEM (10,000 lines of C code).[7] Our implementation uses DPDK 2.2 [15] to bypass the network stack. Our experiments in this section are conducted on 3 Xeon E5-2650 v3 2.30GHz with two 10-core CPUs 25MB L3 and 256KB L2 cache. Our machine has an Intel 82599 10G NIC.

7.1 Expressivity

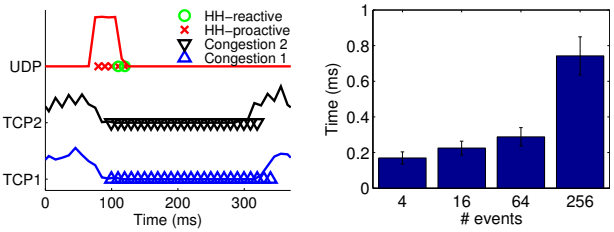
Trumpet is expressive enough to track and control fine time-scale flow dynamics, correlate or drill-down on interfering flows, or detect service-scale anomalies. To demonstrate this, we implemented in Trumpet the three use-cases discussed in Section 2 whose event definition is presented in Section 3.

Identifying losses caused by traffic bursts. Trumpet can detect losses caused by traffic bursts and, in a tight control loop, activate pacing to smooth bursts; this can ameliorate the overhead of pacing [1, 21]. Such an approach can be used in data centers with bottlenecks in the wide-area: bursts can be preferentially paced to reduce loss. As an aside, VM stacks, which may often run different transport protocols, can use Trumpet instead of implementing burst (and other kinds of event) detection within their protocol stacks. In our experiment, a user connects to a server in the cloud through an edge router with shallow queues. The connection between the Internet user and the edge router is 10Mbps and incurs 100ms delay. The link from the server to the edge router is 1 Gbps. Figure 6a shows that there are packet losses every time there is a traffic burst. Trumpet quickly detects the burst and installs a trigger that informs the controller whenever there are more than 8 lost packets within a burst. Figure 6b shows that the trigger is matched after 3 seconds from the starting point of the experiment when the bursts become large enough. Based on the triggers, the controller quickly enables TCP pacing to eliminate the losses and achieves much higher throughput (from 1.94Mbps to 5.3Mbps).



(a) Original (b) With Trumpet+Pacing

Figure 6: Losses caused by bursts



(a) Congestion diagnosis (b) Network-wide scenario

Figure 7: Network-wide and congestion usecases

Identifying the root cause of congestion. Despite advances in transport protocols, transient packet losses can be triggered during sudden onset of flows. In data centers, these losses can slow distributed computations and affect job completion times. To diagnose this transient congestion, it is necessary to identify competing flows that might be root causes for the congestion. Trumpet can be used for this task in two ways. In the *reactive* drill-down approach, the operator defines a TCP congestion detection event and, when it is triggered for many flows, programmatically installs another event to find heavy hitters at the common bottleneck of those flows. This may miss short flows because of the delay in detecting congestion. In the *proactive* approach, the operator installs both events on all servers in advance and correlates their occurrence.

To demonstrate both approaches, we use three senders and one receiver with a bottleneck bandwidth of 100Mbps. Using iperf, two senders send TCP traffic for 20s, and after 15s, the

[6] The model assumes that the attacker is not aware of the installed triggers, thus triggers only need to process "good" flows. The model can be extended to the case where the attacker knows the installed triggers; we have left this to future work.

[7] Available at https://github.com/USC-NSL/Trumpet

third sender sends a short UDP flow (512KB in 40ms) and creates congestion. We define a congestion detection event to report if, at the end of an epoch, any 5-tuple flow cannot receive acks for more than 50% of outstanding data at the beginning of the epoch (Table 2). We define the heavy hitter detection event to report if the volume of any 5-tuple flow is larger than 12KB in every 10ms (10% of link bandwidth).

Figure 7a shows the normalized throughput of the three connections measured at the receiver as lines and the detected events at the controller as dots. The congestion event could correctly detect throughput loss in TCP connections three epochs after the start of the UDP traffic. The delay is partly because it takes time for the UDP traffic to affect TCP connection throughput. After detecting congestion, TEM installs the `HH_Reactive` event to detect heavy hitters with 200μs delay (order of ping delay). `HH_Reactive` can correctly detect the UDP connection in the fourth epoch after its initiation. Figure 7a shows that the proactive approach (`HH_proactive`) could also correctly detect the UDP flow over its lifetime.

Network-wide event to detect transient service-scale anomaly. This use-case shows how Trumpet can detect if the total volume of traffic of a service exceeds a given threshold within a 10ms window. In our experiment, two end-hosts each monitor two 10G ports on two cores. The controller treats each core as a separate end-host (this demonstrates how Trumpet can handle higher NIC speeds). The RTT between the controller and end-hosts is 170 μs. We generate different number of triggers that are satisfied every measurement epoch. When the predicate of a trigger is satisfied at an end-host, the end-host sends a satisfaction message to the controller and the controller polls other servers and collects replies from them.

Figure 7b shows the delay between receiving the first satisfaction and the last poll reply at the controller for different numbers of events. Even with 256 satisfied events, the average delay is only about 0.75ms. The average delay for processing satisfactions and generating polls at the controller is 60ns, showing that Trumpet controller can potentially scale to more than 16M messages from TPMs per second per core.

7.2 Performance

Methodology. In our experiments, our test machine receives traffic from a sender. The traffic generation pattern we test mirrors published flow arrival statistics for data centers [49]: exponential flow inter-arrival time *at each host* with 1ms average with 300 active flows. The TPM monitors packets and forwards them based on their destination IP. It has 4k triggers and each packet matches 8 triggers. Packets may match multiple triggers when a flow is monitored using multiple queries with different predicates (variable, aggregate function, threshold), time intervals or flow granularities. Triggers track one of 3 statistics: packet count, traffic volume or the number of lost packets. In particular, tracking packet losses requires tracking retransmissions across multiple epochs, which can consume memory and additional CPU cycles. However, the triggers are designed such that the 8 triggers matched by each packet cover the 3 types of statistics. Also, every trigger is evaluated at the granularity of 10ms. Unless otherwise specified, the default burst size is 1 packet (*i.e.,* every subsequent packet changes its flow tuples), which foils lookup caching. These settings were chosen to saturate the packet processing core on our server. We also explore the parameter space (number of triggers, flow arrival rates, time interval, *etc.*) to characterize the *feasibility region* — the set of parameters for which Trumpet can monitor events without losing a packet or missing a sweep.

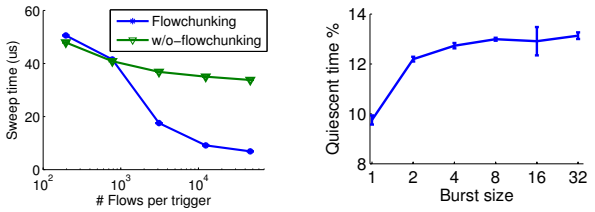
We use several metrics to evaluate our schemes, including the fraction of time the CPU was *quiescent* (*i.e.,* not in either of the two phases; this quantity is obtained by aggregating the time between two queue polls that returned no traffic), the time for the sweep phase, and whether a particular design point incurred loss or not. We ran each experiment for 5 times with different flow arrival patterns for 50 seconds. The variance across runs is very small, and we present the average.

Baseline experiment. We ran our synthetic traffic at the maximum packet rate of 14.8Mpps (64B per packet) on a 10G link for the trigger configuration discussed above and found that (no associated figure) TPM (a) *never loses a packet*, (b) is able to *correctly compute* every statistic, and (c) is able to *complete sweeps* within an epoch so triggers are correctly evaluated. We also *got the same results on a 40G (4x10G) NIC* with 650 byte packets at full line rate. For a 40G NIC, we expected to be able to support 256 byte packets (since for a 10G NIC we can support 64 byte packets at line rate). However, in our evaluation setting, TPM has to poll four ports, and this polling introduces overhead, requiring a larger packet size for Trumpet to be feasible at 40G.

The time granularity in this experiment is 10ms, and we earlier showed that other simpler strategies (Section 5.1) incur 10-20% packet loss in this regime. This experiment validates our design and suggests that it may be possible to monitor a variety of events precisely and at fine-timescales by leveraging the computing capabilities of end-hosts in data centers. More important, in this baseline experiment, the *CPU is never quiescent*. Finally, we have designed TPM for the worst-case: servers in data centers are unlikely to see sustained 14.88Mpps rate (we quantify below how our system performs at lower rates).

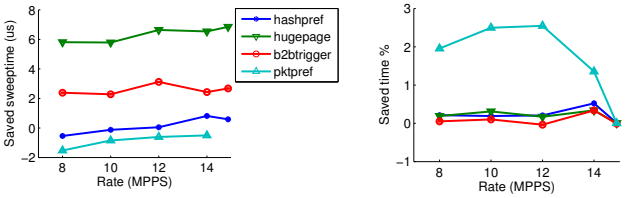
Match-and-scatter optimizations. We described several optimizations for the match-and-scatter phase in Section 5.4. Here we quantify the benefits of each optimization (Figure 9). Packet prefetching saves about 2% of CPU time over different packet rates, a significant improvement because we save about 200μs in a 10ms interval. This is more than enough for two sweeps (each sweep takes $< 100\mu s$). More important, recall that at full packet rate, the CPU is never quiescent. Thus, any small increase in CPU time will cause the monitoring system to lose packets, which is completely unacceptable. Indeed, when we turn off packet prefetching, at 14.88Mpps, we experience 4.5% packet loss, and TPM cannot finish the sweep of all the triggers.

Similarly, although other optimizations contribute small benefits, each of these benefits is critical: without these, TPM would either lose packets or not be able to finish sweeps in



(a) Flow Block (b) Burst size

Figure 8: Optimizations saving



(a) Sweep time (b) Quiescent time

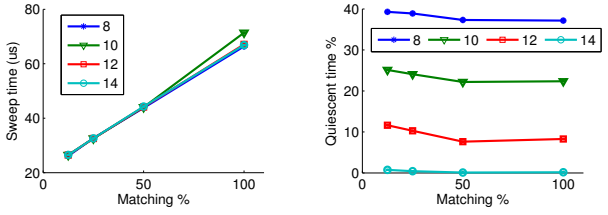
Figure 9: Optimizations saving of flow tables and trigger tables

time (which can lead to missed events). Using huge pages for storing the flow table and triggers saved 6μs of sweep time. Moreover, huge pages and prefetching flow table entries saves 0.2% of CPU time. To understand the effectiveness of caching the result of the last flow table lookup for a burst of packets, Figure 8b shows the impact on quiescent time with different burst sizes for 12Mpps: with the growth of the burst size from 1 to 8, the CPU quiescent time increases from 10% to 13%.

Gather-test-and-report phase optimizations. For a similar reason, even small benefits in this phase can be critical to the viability of TPM. Storing trigger entries contiguously in the trigger repository reduces the sweep time by 2.5μs (Figure 9) because it enables hardware prefetching. To evaluate the benefit of using chunked lists, we keep the same total number of active flows as 300, but change the number of flows per trigger from 200 to 45k (recall that, in our traffic, 1000 flows arrive at each host every second). Figure 8a shows that this can save up to 27μs (80%) in sweep time when there are many flows per trigger, and its overhead is small when there are few flows per trigger.

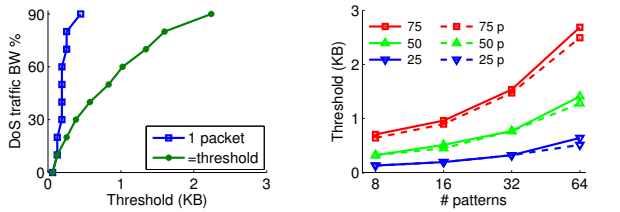
Resource-proportional design. Trumpet's careful partitioning of functionality results in resource-usage that scales well both with traffic rate, and with the level of *monitored* traffic. This efficient design is the key reason we are able to increase the expressivity of Trumpet's language, and track fairly complex per-packet properties like loss rates and RTTs.

Figure 10b shows that CPU quiescent time decreases steadily as a function of the packet rate: at 8Mpps, the



(a) Sweep time (b) Quiescent time

Figure 10: Proportional resource usage on % flows matched (legend shows rate in Mpps)



(a) DoS BW per threshold (b) DoS threshold prediction

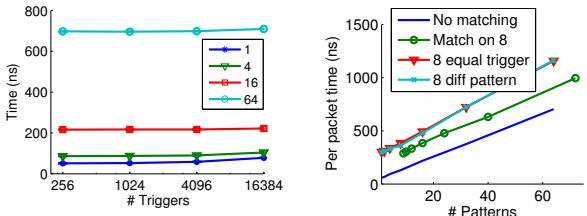
Figure 11: DoS resiliency

system is quiescent 40% of the time, while at 14Mpps it is always fully utilized. In this experiment, the quiescent time is independent of the fraction of matching traffic because the dominant matching cost is incurred for every new flow, regardless of whether the flow matches triggers filters or not. However, the gather phase scales well with the level of *monitored* traffic: as Figure 10a shows, the sweep time is proportional only to the number of flows that match a trigger. Thus if trigger filters match fewer flows, the sweep will take shorter, and the TPM can support smaller time intervals.

DoS-resilience. We perform an experiment in which the TPM is subjected to the full line rate. Legitimate flows send 10 packets of maximum size (1.5KB). We consider two attack models: a) Syn attack where the attacker sends only one packet per flow to maximize packet processing overhead b) Threshold attack where the attacker knows the threshold and sends flows with size equal to the threshold to maximize matching overhead. Also all flows come in a burst of one (no subsequent packets are from the same flow). Our experiment quantifies the percentage of DoS traffic that can be tolerated at each DoS threshold. Thus, it uses two independent knobs: the fraction of DoS traffic in the full packet rate, and the DoS threshold. Each point in Figure 11a plots the DoS fraction and the threshold at which the system is functional: even a small increase in one of these can cause the system to fail (either lose packets or not finish sweeps).

As Figure 11a shows, for any threshold more than 440B (on the right of the one packet line), the TPM can sustain a SYN attack. This form of attack is easy for the TPM to handle, since if the traffic is below the threshold, then matching is not incurred. The threshold is higher than 1 packet because there are false positives in the fast counter array implementation of the filter table [2]. At lower thresholds, smaller fractions of attack traffic can be sustained. For the threshold attack, a threshold of 384B (832B) ensures immunity to more than 30% (50%) DoS traffic; this level of bandwidth usage by DoS traffic means that 90% (96%) of packets are from the attacker. The DoS threshold can be decreased even further, to 128 bytes, by checking against a few filters just before matching triggers, to see if a flow would likely match *any* trigger (this is cheaper than checking which trigger a flow matches). At this threshold, a very small fraction of Web server flows in a large content provider would go unmonitored [49].

As discussed in Section 6, the DoS threshold can be calculated by a profiling-based model. Figure 11b shows that the predicted threshold is close to the experimental threshold over different number of trigger patterns (series show DoS traffic % and suffix "p" means prediction).



(a) Scale to many triggers (the curves are different #patterns)

(b) Impact of #patterns

Figure 12: Performance of trigger matching

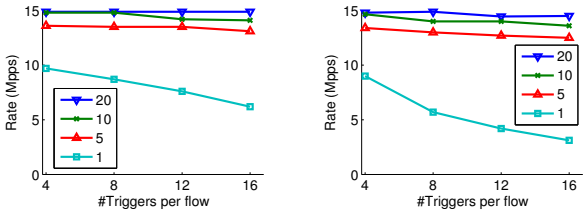
Performance of matching triggers. A key bottleneck in Trumpet is the cost of matching a packet to a trigger filter. Even with aggressive caching of lookups, because our event descriptions involve a multi-dimensional filter, this cost can be prohibitive. Figure 12a depicts the impact of matching complexity on Trumpet. Recall that we use the tuple search algorithm to match across multiple dimensions.

The first important result is that this cost is independent of the number of triggers. This is key to our scaling performance: we are able to support nearly 4K concurrent triggers precisely because the tuple search algorithm has this nice scaling property. Where this algorithm scales less well is in the direction of the number of "patterns". Recall that a pattern is a type of filter specification: a filter expressed on only `srcIP` or `dstIP` is a different pattern than one expressed on the conjunction of those two fields. Our matching cost increases with the number of patterns. It may be possible for the TEM to analyze trigger descriptions and avoid installing triggers with too many patterns at a TPM, a form of admission control[8]. To further reduce the impact of the total number of patterns, we can adopt trie indexes [52] to reduce the number of hash table lookups. We have left this to future work.

Finally, in Figure 12b we test how matching cost depends on increasingly complex matching scenarios. We consider four such scenarios: *no matching* (no flows match any triggers), *same 8 triggers* (all the flows match the same 8 triggers, the other triggers are not matched), *diff 8 triggers* (each flow matches 8 different triggers, but these triggers have the same filter), and *8 trigger patterns* (each flow matches 8 different trigger patterns). We observe that the per packet processing time increases from *no matching* to *same 8 triggers* and to *diff 8 triggers*. However, the processing time does not further grow with *8 trigger patterns* because our performance does not depend on whether a flow matches different trigger pattern or not, but only depends on the number of patterns.

TPM's feasibility region. We ran experiments with different traffic properties (packet rate, number of active flows, flow arrival rate) and TPM parameters (number of triggers, number of triggers per flow, time interval) to explore the *feasibility region* for the TPM on our server. We call a set of parameters feasible if, for those parameters, TPM does not drop packets and completes all sweeps. The feasibility region is the maximal boundary of feasible parameter sets. We run each parameter set 5 times and show the feasibility region in Figure 13 for 300 and 600 active flows per time

[8]More generally, TEM might perform other forms of admission control, such as rejecting events whose filters span a large part of the address space.



(a) 300 active flows

(b) 600 active flows

Figure 13: Feasibility region over the TPM parameters

interval (*e.g.,* 10ms). The settings have different number of triggers per flow (x axis) and different time intervals (series). As the number of triggers per flow increases, the number of triggers also increases accordingly. For example, the setting with 8 triggers per flow has 4k triggers, and the setting with 16 triggers per flow has 8k triggers.

As we decrease the time interval, there will be fewer cycles in an epoch to use for sweeps, and there is a higher chance that the TPM cannot finish the sweep. For example in Figure 13a, if the time interval is 5 ms and there are 8 triggers per flow, the TPM cannot finish all sweeps for 14.88Mpps. However, reducing the rate to 13.5Mpps (73B packets in 10G), gives more free cycles between packets for sweeping, which makes the setting feasible. Moreover, as we increase the triggers per flow, the gather phase must process more flow entries, thus can handle lower rates. For example, if the time interval is 10 ms and there are 16 triggers per flow, the packet rate must be below 14.1Mpps. Increasing the number of active flows to 600 in Figure 13b also increases the number of flow entries to be processed in the gather phase and reduces the feasible packet rate. We have also tried with different flow arrival rates and noticed that the rate does not affect the feasible boundary until it changes the number of active flows in an interval significantly. Cloud operators can characterize the feasibility region for their servers, and then parameterize TPM based on their workload.

TPM and packet mirroring. We have verified the mirroring capability by configuring the NIC to mirror every packet to a reserved queue for TPM while other cores read from the other queues. The TPM could handle 7.25Mpps without any packet loss or unfinished sweeps. This rate is almost half of the 10G maximum line rate as each packet is fetched twice from the NIC.

8. RELATED WORK

In Section 2, we described why commercial packet monitoring solutions are insufficient for our purposes. Recent research has proposed a variety of solutions towards more active forms of monitoring, mostly by leveraging switch functionality. Planck [48] mirrors sampled packets at switches. NetSight [25] and EverFlow [58] use switches inside the network to send packet headers to a controller. NetSight incurs bandwidth overhead and CPU overhead by processing packet postcards twice (at hosts/switches and controller). EverFlow [58] tries to address the bandwidth overhead by letting the operator select packets for mirroring. OpenSketch [56], FlowRadar [35] and UnivMon [37] use sketches inside switches to support many per-flow queries with low band-

width overhead, but require changes in switches. Relative to these systems, Trumpet sits at a different point in the design space: it can monitor every packet by using compute resources in servers, and incurs less bandwidth overhead by sending trigger satisfaction reports. Ultimately, we see these classes of solutions co-existing in a data center: Trumpet can give quick and early warning of impending problems and has more visibility into host-level artifacts (*e.g.,* NIC drops), and other switch-based monitoring systems can be used for deep drill-down when packet or header inspection inside the network is required.

At end-hosts, existing monitoring tools often focus on a specific type of information, require access to VMs or use too much of a resource. For example, SNAP [57] and HONE [53] monitor TCP-level statistics for performance diagnosis. PerfSight [55] instruments hypervisor and VM middleboxes to diagnose packet losses and resource contention among VMs in 100ms timescale. RINC [20] infers internal TCP state of VMs from packets. Pingmesh [23] leverages active probes to diagnose connectivity problems. In contrast, Trumpet shows it is possible to inspect every packet at end-hosts at line speed for a wide set of use-cases. Trumpet runs in the hypervisor and assumes no access to the VM's networking stack. Trumpet can possibly be extended to non-virtualized systems (*e.g.*, containers, by interposing on the network stack), but we have left this to future work. n2disk [42] allows capturing all packets to disk in 10G links using multiple cores. Trumpet aggregates packet-level information on the fly and requires no storage. Packet monitoring functions can be offloaded to programmable NICs [24, 6]. However, these devices currently have limited resources, are hard to program [34], and have limited programmability compared to CPUs. Trumpet can leverage NIC offloading if the hypervisor cannot see packets (*e.g.,* SR-IOV and RDMA). As NICs evolve, it may be possible to save CPU by offloading some parts of Trumpet processing, such as matching, to NICs.

At the controller, there are many systems that can potentially integrate with Trumpet. Kinetic [32] allows controlling networks based on events that can be generated using Trumpet. Felix [7] generates matching filters for end-host measurement from high-level user queries and routing configuration.

Gigascope [12] introduces a stream database of traffic mirrored from the network, and supports a variety of traffic queries on the database. It also introduces a two stage approach of low-level queries and high-level queries to improve streaming efficiency [11]. Instead of collecting traffic into a central database, Trumpet distributes the monitoring and stream processing at all the end-hosts and aggregates the trigger information to capture network-wide events. Trumpet also introduces a new trigger repository that captures events in 10 ms intervals at line speed.

9. CONCLUSIONS

In this paper, we discuss the design of a new capability in data-center network: active fine-timescale and precise event monitoring. Our event description language is expressive enough to permit novel events that are particularly relevant in data centers. Our algorithms and systems optimizations ensure a design that can process every packet at line-rate, is DoS-resilient, scales to future network technologies, and permits programmed tight-loop control and drill-down.

In future work, Trumpet can benefit from NIC capabilities such as rule matching to detect events in links with higher packet rates with less CPU overhead. Moreover, in collaboration with monitoring at other vantage points (*e.g.,* switches), Trumpet may achieve the most cost-effective and comprehensive solution for network telemetry and root cause analysis.

Acknowledgement. We thank our shepherd Hitesh Ballani and anonymous SIGCOMM reviewers for their insightful feedback. This paper is partially supported by NSF grants CNS-1453662 and CNS-1423505.

References

[1] A. Aggarwal, S. Savage, and T. Anderson. "Understanding the Performance of TCP Pacing". In: *INFOCOM*. Vol. 3. 2000.

[2] O. Alipourfard, M. Moshref, and M. Yu. "Re-evaluating Measurement Algorithms in Software". In: *HotNets*. 2015.

[3] M. Allman, W. M. Eddy, and S. Ostermann. "Estimating Loss Rates with TCP". In: *SIGMETRICS Performance Evaluation Review* 31.3 (2003), pp. 12–24.

[4] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. "End-to-End Performance Isolation Through Virtual Datacenters". In: *OSDI*. 2014.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. "Workload Analysis of a Large-scale Key-value Store". In: *SIGMETRICS*. 2012.

[6] H. Ballani et al. "Enabling End-host Network Functions". In: *SIGCOMM*. 2015.

[7] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang. "Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis". In: *SOSR*. 2016.

[8] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. "Understanding TCP Incast Throughput Collapse in Datacenter Networks". In: *WREN*. 2009.

[9] M. Chowdhury and I. Stoica. "Efficient Coflow Scheduling Without Prior Knowledge". In: *SIGCOMM*. 2015.

[10] G. Cormode, R. Keralapura, and J. Ramimirtham. "Communication-Efficient Distributed Monitoring of Thresholded Counts". In: *SIGMOD*. 2006.

[11] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. "Holistic UDAFs at Streaming Speeds". In: *SIGMOD*. 2004.

[12] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. "Gigascope: a Stream Database for Network Applications". In: *SIGMOD*. 2003.

[13] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. "DevoFlow: Scaling Flow Management for High-Performance Networks". In: *SIGCOMM*. 2011.

[14] M. Dobrescu, K. Argyraki, G. Iannaccone, M. Manesh, and S. Ratnasamy. "Controlling Parallelism in a Multicore Software Router". In: *PRESTO*. 2010.

[15] *DPDK*. http://dpdk.org.

[16] D. E. Eisenbud et al. "Maglev: A Fast and Reliable Software Network Load Balancer". In: *NSDI*. 2016.

[17] D. Firestone. "SmartNIC: FPGA Innovation in OCS Servers for Microsoft Azure". In: *OCP U.S. Summit*. 2016.

[18] M. Gabel, A. Schuster, and D. Keren. "Communication-Efficient Distributed Variance Monitoring and Outlier Detection for Multivariate Time Series". In: *IPDPS*. 2014.

[19] R. Gandhi, Y. C. Hu, C.-k. Koh, H. H. Liu, and M. Zhang. "Rubik: Unlocking the Power of Locality and End-Point Flexibility in Cloud Scale Load Balancing". In: *ATC*. 2015.

[20] M. Ghasemi, T. Benson, and J. Rexford. *RINC: Real-Time Inference-based Network Diagnosis in the Cloud*. Tech. rep. Technical Report TR-975-14, Princeton University, 2015.

[21] M. Ghobadi and Y. Ganjali. "TCP Pacing in Data Center Networks". In: *High-Performance Interconnects (HOTI)*. 2013.

[22] *Google Compute Engine Incident 15041*. https://status.cloud.google.com/incident/compute/15041. 2015.

[23] C. Guo et al. "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis". In: *SIGCOMM*. 2015.

[24] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. *SoftNIC: A Software NIC to Augment Hardware*. Tech. rep. UCB/EECS-2015-155. http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html. EECS Department, University of California, Berkeley, 2015.

[25] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks". In: *NSDI*. 2014.

[26] Y.-J. Hong and M. Thottethodi. "Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier". In: *SOCC*. 2013.

[27] L. Hu, K. Schwan, A. Gulati, J. Zhang, and C. Wang. "Netcohort: Detecting and Managing VM Ensembles in Virtualized Data Centers". In: *ICAC*. 2012.

[28] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. "Characterizing Load Imbalance in Real-World Networked Caches". In: *HotNets*. 2014.

[29] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". In: *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)* (2008), pp. 1–269.

[30] *Intel Data Direct I/O Technology*. http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[31] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. "Bullet Trains: A Study of Nic Burst Behavior at Microsecond Timescales". In: *CoNEXT*. 2013.

[32] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. "Kinetic: Verifiable Dynamic Network Control". In: *NSDI*. 2015.

[33] A. Kumar et al. "BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing". In: *SIGCOMM*. 2015.

[34] B. Li et al. "ClickNP: Highly Flexible and High-performance Network Processing with Reconfigurable Hardware". In: *SIGCOMM*. 2016.

[35] Y. Li, R. Miao, C. Kim, and M. Yu. "FlowRadar: A Better NetFlow for Data Centers". In: *NSDI*. 2016.

[36] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. "MICA: A Holistic Approach to Fast In-memory Key-value Storage". In: *NSDI*. 2014.

[37] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon". In: *SIGCOMM*. 2016.

[38] N. McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Computer Communication Review* 38.2 (2008).

[39] R. Miao, R. Potharaju, M. Yu, and N. Jain. "The Dark Menace: Characterizing Network-based Attacks in the Cloud". In: *IMC*. 2015.

[40] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. "DREAM: Dynamic Resource Allocation for Software-defined Measurement". In: *SIGCOMM*. 2014.

[41] M. Moshref, M. Yu, A. Sharma, and R. Govindan. "Scalable Rule Management for Data Centers". In: *NSDI*. 2013.

[42] *n2disk: A Multi-Gigabit Network Traffic Recorder with Indexing Capabilities*. http://www.ntop.org/products/traffic-recording-replay/n2disk/.

[43] N. Parlante. *Linked List Basics*. http://cslibrary.stanford.edu/103/LinkedListBasics.pdf. 2001.

[44] P. Patel et al. "Ananta: Cloud Scale Load Balancing". In: *SIGCOMM*. 2013.

[45] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. "Fastpass: A Centralized "Zero-queue" Datacenter Network". In: *SIGCOMM*. 2014.

[46] B. Pfaff et al. "The Design and Implementation of Open vSwitch". In: *NSDI*. 2015.

[47] R. Potharaju and N. Jain. "Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters". In: *IMC*. 2013.

[48] J. Rasley et al. "Planck: Millisecond-scale Monitoring and Control for Commodity Networks". In: *SIGCOMM*. 2014.

[49] A. Roy, H. Zeng, J. Bagga, G. M. Porter, and A. C. Snoeren. "Inside the Social Network's (Datacenter) Network". In: *SIGCOMM*. 2015.

[50] I. Sharfman, A. Schuster, and D. Keren. "A Geometric Approach to Monitoring Threshold Functions over Distributed Data Streams". In: *Transaction on Database Systems* 32.4 (Nov. 2007).

[51] A. Singh et al. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In: *SIGCOMM*. 2015.

[52] V. Srinivasan, S. Suri, and G. Varghese. "Packet Classification Using Tuple Space Search". In: *SIGCOMM*. 1999.

[53] P. Sun, M. Yu, M. J. Freedman, J. Rexford, and D. Walker. "HONE: Joint Host-Network Traffic Management in Software-Defined Networks". In: *Journal of Network and Systems Management* 23.2 (2015), pp. 374–399.

[54] M. Wang, B. Li, and Z. Li. "sFlow: Towards Resource-efficient and Agile Service Federation in Service Overlay Networks". In: *International Conference on Distributed Computing Systems*. 2004.

[55] W. Wu, K. He, and A. Akella. "PerfSight: Performance Diagnosis for Software Dataplanes". In: *IMC*. 2015.

[56] M. Yu, L. Jose, and R. Miao. "Software Defined Traffic Measurement with OpenSketch". In: *NSDI*. 2013.

[57] M. Yu et al. "Profiling Network Performance for Multi-tier Data Center Applications". In: *NSDI*. 2011.

[58] Y. Zhu et al. "Packet-Level Telemetry in Large Datacenter Networks". In: *SIGCOMM*. 2015.