Packet-Level Network Telemetry and Analytics

by

Oliver Michel

B.Sc., University of Vienna, 2013

M.S., University of Colorado Boulder, 2015

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2019

This thesis entitled:
Packet-Level Network Telemetry and Analytics
written by Oliver Michel
has been approved for the Department of Computer Science

_____
Professor Eric Keller

_____
Professor Dirk Grunwald

_____
Professor Sangtae Ha

_____
Professor Eric Rozner

_____
Professor Eric Wustrow

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Michel, Oliver (Ph.D., Computer Science)

Packet-Level Network Telemetry and Analytics

Thesis directed by Professor Eric Keller

Continuous monitoring is an essential part of the operation of computer networks. High-fidelity monitoring data can be used to detect security issues, misconfigurations, equipment failure, or to perform traffic engineering. With networks growing in complexity, traffic volume, and facing more complex attacks, the need for continuous and precise monitoring is greater than ever before. Existing SNMP or NetFlow based approaches are not suited for these new challenges as they compromise on flexibility, fidelity, and performance. These compromises are a result of the assumption that analytics software cannot scale to high traffic rates.

In this work, we look holistically at the requirements and challenges in network monitoring and present an architecture consisting of integrated telemetry, analytics, and record persistence components. By finding the right balance between responsibilities of hardware and software, we demonstrate that flexible and high-fidelity network analytics at high rates is indeed possible.

Our system includes a packet-level, analytics-aware telemetry component in the data plane that runs at line-rates of several Terabits per second and tightly integrates with a flexible software network analytics platform. Operators can interact with this system through a time series database interface that also provides record persistence. We implement a full prototype of our system called Jetstream which can process approximately 80 million packets per 16-core commodity server for a wide variety of monitoring applications and scales linearly with server count.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Today's networks are larger and more complex than ever before [65]; they carry more traffic, run more advanced applications [70, 50, 97, 149] and are continuing to grow [55]. In particular, wide-area, enterprise, and datacenter networks commonly operate at traffic rates of hundreds of millions of packets per second per switch [193]. This makes continuous network monitoring essential to the operation of cloud-scale networks. Network monitoring enables operators to detect security issues, misconfigurations, equipment failure and to perform traffic engineering [162, 154, 21, 36, 80, 103].

Network Monitoring

Network monitoring and analytics describe the process of extracting information from network devices in the form of statistics or traffic records and transforming this data into usable information (see figure 1.1). Usable information refers to alerts, summarizing statistics, or other metrics that can be directly used to make network management decisions either by the operator or autonomously. In practice, either polling-based approaches or passive monitoring are used to gather data from the network. Polling entails querying network devices continuously for specific metrics. A widely used protocol enabling this functionality is the Simple Network Management Protocol (SNMP). The primary alternative is to export information about the traffic that traverses the network in the form of mirrored packets, packet records, or flow records to an analysis system that then calculates metrics of interest. This approach, which we focus on in this thesis, is commonly referred to as *passive monitoring* or *streaming telemetry*. To mine useful information from network data (i.e., performing analytics), big data processing technologies, such as *streaming analytics*, are commonly

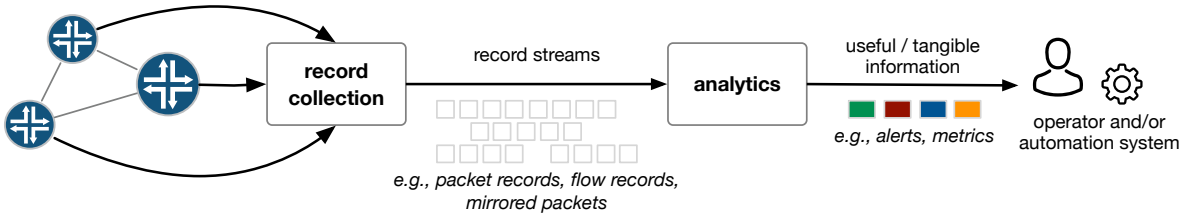


Figure 1.1: Overview of a Telemetry-Based Network Monitoring and Analytics System

used.

While the overall objective of network monitoring and analytics, that is *gathering useful information from network traffic data or device statistics*, seems straight forward, deploying such functionality at the scale and complexity of modern computer networks is extremely challenging and is an area of ongoing research [154]. Previous work has aimed at providing solutions for a variety of sub-problems including telemetry [155, 193], query abstractions [122, 79], or hardware offload for analytics [185, 107].

Existing Solutions make Compromises

In this thesis, we take a step back and have a holistic view at the field of network monitoring. Network monitoring platforms that support a wide range of applications are complex systems that must balance competing requirements. Existing systems generally compromise along three axes: flexibility, fidelity, and performance.

Flexibility describes how adaptable a network monitoring and analytics platform is to evolving user requirements. Many existing solutions offload analytics tasks to programmable hardware [122, 185, 107]. Programmable switches are rigid, resource-constrained devices that can only carry out basic operations (i.e., no loops, no floating point arithmetic, or divisions) and also prohibit the deployment of multiple parallel applications due to limited RAM and ALU resources [38, 39, 151]. This hinders practical deployment requirements and the capability of applications.

Fidelity refers to the detail of data provided by the telemetry system. Filtering, aggregation, and sampling have been the de-facto standards for decades [83, 154]. These approaches, of course, compromise on data fidelity. Early filtering and sampling must be carried out extremely carefully

as the risk of missing crucial packets or introducing sampling bias is high. Early aggregation has the result of only having access to byte totals, overall packet counts, and flow durations, and again, limits the set of possible applications [56]. A growing set of monitoring applications today requires per-packet information, such as individual time stamps, to apply complex machine learning models for traffic classification or intrusion detection [108, 126, 118].

Performance and scalability describe how much network traffic a monitoring system can sustain as a function of the deployed resources, and whether the system can easily scale to absorb more traffic or run computationally more expensive applications. Existing analytics solutions (often based on the stream processing paradigm) provide high flexibility and programmability, but compromise on performance. They commonly report processing rates of only one to two million events per second per core [79, 187, 10, 193]. This is in contrast to traffic rates of several hundreds of millions of packets per second per switch in wide-area or data center networks. Even with linear scalability, racks full of servers for monitoring and analytics alone would be required.

Evolving Requirements

To get an insight into why these compromises are made, we start with our vision of an *ideal* network monitoring and analytics system: Each switch in the network streams a record for every single packet that it sees to a cluster of servers running different analytics tasks in software. This architecture would allow for virtually unlimited monitoring applications and queries as the analytics applications can be written in general purpose languages and run on general purpose CPUs.

Unfortunately, the current physical reality is more complex than that. There is a mismatch between the number of packets that can be streamed from the network and the amount of packets that can be efficiently analyzed using ("off-the-shelf") streaming analytics software [79, 122]. This is the main cause for compromises. Existing systems use strategies to reduce the load of the network analytics platform (e.g., by hardware offload or sampling) hurting flexibility and capability, or, alternatively, provide high flexibility through programmable software analytics and then compromise on performance.

A holistic Approach to Network Telemetry & Analytics

In this thesis, we show that these compromises are not inevitable and that network monitoring and analytics can be performed at high flexibility, with high fidelity, and at cloud-scale rates. We make the argument that compromises of previous work are merely a result of how researchers have approached the problem. This is based on two insights: First, software is not inherently incapable of providing fast packet-level analytics and scalability. Low processing rates are rather a result of design. Existing stream processing systems are general-purpose systems to be used across many industries and workload types. Packet-level network analytics represents a very special case and allows for many optimizations. Second, the majority of existing work treats network telemetry (i.e., exporting information from network devices) and network analytics (i.e. the process of mining meaningful information from the exported data) as two separate problem spaces.

As a result, telemetry and analytics have been kept entirely separate, or on the other extreme, weaved into a single component. In this thesis we show that by looking at both subproblems holistically and carefully integrating the different monitoring components, we can provide high performance without compromising on applications, operational flexibility, or data fidelity.
This approach led to the following contributions of this thesis:

(1) We propose a novel in-network data structure that can generate feature-rich grouped packet records at line rates of several Terabits per second.

(2) We increase processing parallelism by offloading load balancing of packet records to the network and letting switches and network interface cards steer packet records to individual processing pipelines in software.

(3) We increase processing performance of these pipelines through domain-specific optimizations to rates of up to 30 million packets per second per core.

(4) We motivate the need for network record persistence and show how retrospective network queries can help investigating a variety of network performance and security issues.

Introducing Toccoa

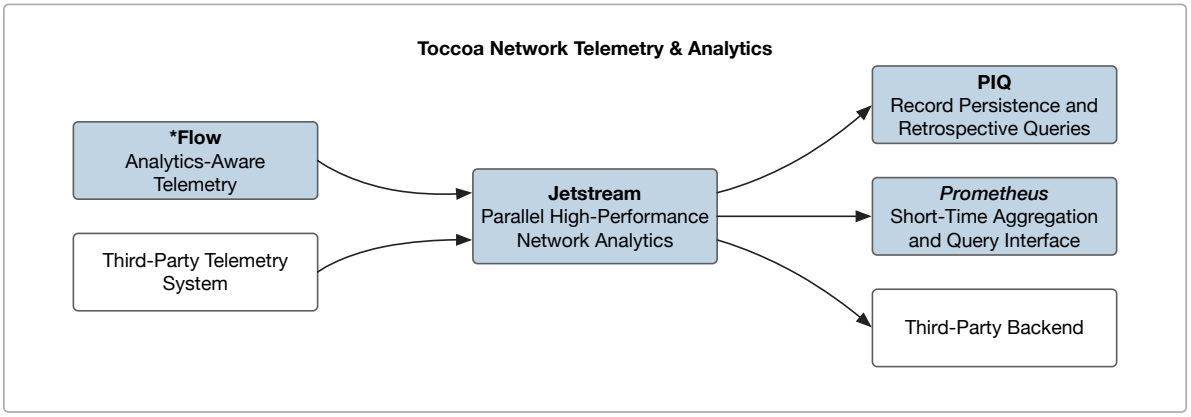


Figure 1.2: Toccoa Network Telemetry & Analytics Components

We built a comprehensive network monitoring and analytics system encompassing these contributions, called Toccoa[1] . Toccoa is a novel, carefully re-architected system for high-performance network telemetry and analytics. Toccoa's overall architecture is depicted in figure 1.2 and consists of three components:

(1) **Flow*, an analytics-aware telemetry system that leverages a specialized in-network data structure to generate grouped packet-level records that greatly improve analytics performance.

(2) *Jetstream*, a high-performance network analytics platform that tightly integrates with *Flow and leverages parallelism in modern multi-core CPU architectures to achieve high-throughput for long-running queries and monitoring applications.

(3) *PIQ*, a network record persistence system that allows for detailed, retrospective queries on network records.

Toccoa's modular design also allows for different telemetry systems or data sources or a variety of backends besides PIQ. To demonstrate this, we implemented a backend for short-term aggregation, queries, and visualization using the Prometheus [11] time-series database and the Grafana [7] visualization framework.

[1] Named after the Toccoa river, known for its fast water flow and whitewater rapids, to contrast with the slow moving nature of a stream. [19]

In the remainder of this thesis, we first provide background and elaborate on the changing network monitoring landscape by discussing new challenges, as well as new opportunities, and motivate the decisions we have made in the design of Toccoa (chapter 2). Chapters 3, 4, and 5 in depth describe Toccoa's three main components and evaluate each of them. Chapter 6 gives direction for future research and concludes this work.

Chapter 2

Network Monitoring and Analytics

2.1 Background

The need for network monitoring is as old as the operation of computer networks itself. Even in the early days of networks, the systems were complex, heterogenous infrastructures with countless possibilities for misconfiguration or equipment failure [81]. In this chapter, we give an overview of the field of network monitoring in the broader context of network management.

2.1.1 Network Management

Network management describes the overall operation of a computer network. Network monitoring is an essential part of the operation of computer networks as it provides insight into the current state of the network. These insights often contain indications for the need for adjustments, such as provisioning more resources or changing configurations.

Network Management is often characterized by a control loop consisting of three distinct phases that are repeated throughout the network's lifecycle [102]:

- In the *Design Phase*, operators map desired behavior, and new or changed requirements to configuration changes.
- In the *Deployment Phase*, configuration changes are safely delivered and rolled back in case they are not satisfactory.

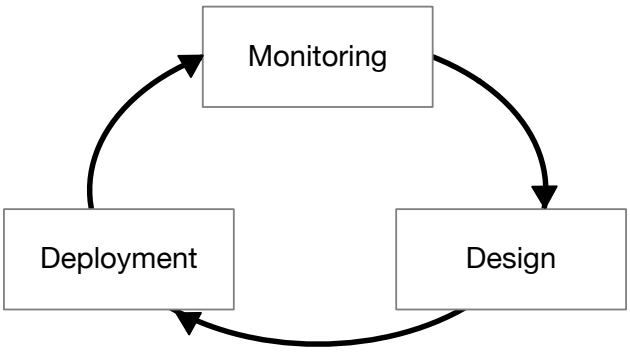


Figure 2.1: Network Management and Operation Lifecycle

- In the *Monitoring Phase*, the network's behavior is continuously measured, usage patterns are being identified, and previously rolled out changes are verified.

In this model, the output of each phase serves as the input for the next phase. We can see that network monitoring plays an essential role for two reasons. First, network monitoring data can be used to verify that the previous set of changes to the network were correct and are working as intended. Second, network monitoring aids the operator in understanding the current behavior of the network. In particular, network monitoring data can be used to anticipate future requirements. These requirements may include the need for better support for a type of traffic that is growing, or the provisioning of additional equipment in a part of the network where the utilization is rapidly increasing [77, 102, 165, 184].

In the context of network automation and intent-driven networking, this lifecycle dependency graph does not change. Rather, instead of a (human) operator analyzing monitoring data, drawing conclusions and making decisions about required changes, a network automation platform would analyze the data, identify optimization opportunities, and change configurations or provision resources within its capabilities accordingly [182, 145, 86].

2.1.2 Sensors

Network monitoring data is typically gathered through sensors. A sensor is a piece of software or physical device that collects information within an environment (in our case the network). This

information can then be used for reporting and/or reconfiguration, or just be archived for later analysis. Given this definition, a sensor can be almost any kind of device or software: a network tap, an IDS log, or a server access log to name a few. In the context of network security, sensors are typically divided into two types: *Host-Based Sensors* and *Network-Based Sensors* [60].

While host-based sensors usually provide more accurate, detailed information about a certain subsystem they are monitoring, in this work, we primarily focus on network-based sensors. Network-based sensors collect data directly from the network without an intermediate processing entity that filters data or generates events. Network-based sensors typically come in two different types: polling-based sensors and telemetry-based sensors. The following two sections (section 2.1.2.1 and section 2.1.2.2) elaborate on these two approaches in detail.

2.1.2.1 Polling-based Network Monitoring

Protocols such as SNMP can be used to query devices for statistics, such as the total number of packets or bytes that entered the device through a particular port. This data is heavily compressed and does not provide any insight into what type of traffic actually caused for example unusually high packet counts on a specific link. This type of data collection is extremely fast and does incur very little overhead, however, it is also inflexible as the set of statistical features exported is fixed.

2.1.2.2 Telemetry-based Network Monitoring

Telemetry systems export information about the actual traffic that traversed a network device. This data can be raw or aggregated at different levels of granularity. While data exported in this format does not provide fixed statistics, the generated records usually have all the meta data (e.g., byte counts) such that any type of statistics can be calculated from the data. This makes this type of data gathering extremely flexible but also significantly increases the volume of data to be exported in contrast to statistics-based data.

A *flow record* is a series of packets sharing the same five-tuple of source and destination IP address, IP protocol type, and layer four source and destination ports. Flow records generally

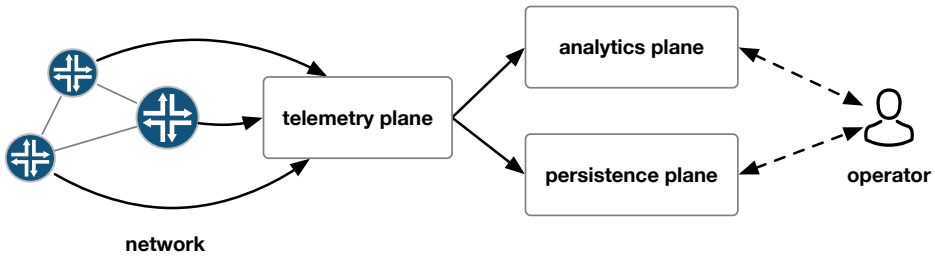


Figure 2.2: Telemetry-based Network Monitoring Architecture

contain summarized information (e.g., total packet count, flow duration, or total number of bytes) about the packets within a respective flow. Over the past years, several flow export standards have been proposed. The most widely used protocols are NetFlow developed by Cisco Systems [56] together with sFlow [146]. As NetFlow is the basis for the IP Flow Information Export Protocol (IPFIX) [31], which is being pushed forward by the IETF, NetFlow has become the de–facto industry standard.

Going even further, *packet records* provide insight into every single packet that traverses the network. As such, packet records contain, for example, timestamps, byte counts and TCP flags for every single packet. Consequently, packet records, while providing the highest level of granularity and detail, are expensive to generate. Also, traditionally, the shear amount of packets in today's networks makes this approach without any optimizations often infeasible.

2.1.3 Telemetry-based Network Monitoring Architecture

Given the new demands and challenges in network monitoring, as well as the emergence of new telemetry and analytics technologies, we believe packet-level monitoring with high coverage is indeed possible. While solutions for some of the required components for a full packet-level network monitoring system have been proposed over the past years, it still remains unclear how these components should be orchestrated in order to form a full end-to-end, packet-level network monitoring solution.

While we elaborate in detail on the individual components, their design and implementation

(chapters 3, 4, and 5), we also study possible ways of integrating the separate components. Figure 2.2 illustrates the architecture of such a system, which would leverage the components described in this proposal along with other recent work. At a high level, the system can be conceptualized as three planes:

(1) A *Telemetry Plane* that collects network records at high rates in the network and sends them to a software platform.

(2) A real time *Analytics Plane* for long-running network queries, that leverages scale out stream processing engines to scan network records and detect problems. Suspicious packets are marked for later analysis.

(3) A *Persistence Plane* and query subsystem in which network records can be saved over longer times at different granularities and retrospectively queried to further investigate issues in the network either by a human or by secondary analysis systems.

We now describe the three planes and their requirements in detail.

2.1.3.1 Telemetry Plane

In contrast to polling-based data gathering, a telemetry plane pushes records collected in the network to the analytics plane. These records are usually directly related to data plane traffic that was handled by the respective device and can be exported at different levels of detail. For instance, the telemetry plane could mirror entire packets including their payload, export truncated packets, generate pre-formatted packet records, or generate flow records at different levels of granularity.

Still, as a telemetry plane implementation typically runs directly on a networking device, the implementation and computational requirements for the network device need to remain practical for deployment. This means that the telemetry functionality cannot require all of the switch's resources. This could have the result that the switch is not able to properly forward network packets anymore, which should always remain the device's primary occupation. Also, performing

too many logical tasks, such as heavy aggregation, filtering, sampling, or record preprocessing on the network device, hinders flexibility and the ability to use exported records for a wide range of applications.

As a consequence of these trade-offs, ideally, the telemetry plane is runtime-configurable and selective to the degree that not every packet passing through the device is handled in the same way by the telemetry system. We imagine that based on filters that are configurable at device runtime, certain flows can e.g., be exported in a raw format (potentially including the packet's payload), while others are exported in a more aggregated format. Such functionality can be implemented on advanced programmable forwarding engines, e.g., P4 [38] hardware. Using these technologies, would also provide access to additional data about switch state, such as queue depths.

2.1.3.2 Analytics Plane

In our architecture, the analytics plane performs long-running analytics and runs queries on data received from the telemetry plane. Possible applications range from performance monitoring applications over failure detection to intrusion detection systems. As the requirements for the analytics plane may be extremely application- and deployment-specific, this component should rather be a framework than a fixed function application.

We believe that the stream processing paradigm is a suitable fit for this workload. A usual stream processing application consists of a series of processors (sometimes called operators or kernels) connected through edges in the form of an acyclic directed graph. Data elements (often called tuples) are passed from one processor to the next through the graph. Each processor that a tuple traverses, applies some function to the data and effectively transforms the tuple.

This design has two main advantages. First, new applications can easily be composed from reusable operators. The stream processing framework could include a standard library of commonly used operators (such as map, reduce, filter). Application logic that is not implementable using the standard library, can still be implemented in custom operators. Second, the stream processing paradigm allows for natural parallelization. As each processor can run in its own thread, the

processing graph conceptually spans across several CPU cores with thread-safe queues (the graph's edges) connecting the processors. Additionally, using load-balancing techniques, one single type of operator can be parallelized by distributing tuples across multiple instances of it.

For high packet rates, the telemetry plane can already load balance packet records to different analytics plane entry points, where behind each entry point a processing graph runs across machines in parallel. Processors in the processing graph should be runtime-configurable such that for example the filter mask of a filter element can be changed. Complete changes to the topology of the graph are not usually required at runtime. We leave this functionality to the persistence and interactive query component described in the next section.

2.1.3.3 Persistence Plane

We envision a general persistence plane with adapters to consume data from both the telemetry and analytics planes. In the simplest case, input could be truncated packet headers cloned from network switches. Most commodity switches support these features, which are sufficient to enable many of the examples described in Chapter 5.

More advanced still, the persistence plane could consume input from the real-time analytics plane. There are at least two use cases for such integration. First, alerts from the real-time analytics system could trigger more advanced retrospective queries. For example, an alert indicating high queue depths or a dropped packet could automatically invoke a more detailed diagnostics query. Additionally, the real-time analytics plane can serve as a preprocessor, normalizing record formats and pre-filtering out data that does not need to be stored.

2.2 Applications

Network Monitoring technologies can be used for a variety of monitoring tasks and objectives. Here, we are outline four popular use-cases of network monitoring, which are important for the operation of computer networks.

2.2.1 Performance Monitoring and Debugging

Computer networks are complex, often heterogenous systems, that require careful configuration and tuning in order to provide best performance and reliability. Especially in the context of large wide-area, data center, or enterprise networks, it is imperative to continuously monitor the operation of the network to maintain and comply with operational objectives and service level agreements (SLA).

As a result of the complexity and dimensions of such network systems, mistakes and failures are inevitable [65]. Failures often happen seemingly randomly, and often affect only a small portion of the traffic (i.e., single packets within a flow). However, even single packet losses can have significant impact on the overall operation of the network and user experience [193, 78, 121]. As a result, researchers have proposed several monitoring solutions aimed at performance monitoring and network debugging. These solutions often leverage packet-level measurements [193, 133], use programmable forwarding engines [124, 122] or in-band telemetry technologies [140, 93, 167].

2.2.2 Traffic Engineering

Traffic Engineering (TE) is the practice of optimizing the overall performance of computer networks for different objectives such as maximizing throughput, minimizing latency, minimizing congestion, or, in general, enforcing service level agreements [29]. Inflated latency, suboptimal throughput, or congestion typically occur when either network resources are insufficient to accommodate the required load, or when traffic is inefficiently mapped and distributed to available resources. The former case generally requires expanding the available capacities through adding links or increasing switching capacity, whereas the latter case can often be quickly mitigated by adapting the network configuration [150, 36].

In either case, network monitoring data is the primary source to detect such mapping inefficiencies or to detect that the network is operating at capacity. Additionally, network monitoring data (such as utilization metrics) can be used to predict future demand. A series of prior work

focuses on the estimation of future traffic demands using historical and limited measurements [160, 71].

2.2.3 Traffic Classification

Network traffic classification is an important network monitoring task for several applications and can be an invaluable tool for network operators that need to know what traffic (or classes of traffic) are flowing over their network [126]. This information is often used for purposes of intrusion detection, traffic engineering, accounting, or to identify network usage that is not in accordance with the provider's terms of service (ToS). Recently, more Internet service providers (ISP) are often subject to government lawful interception (LI) requests about network usage by individuals or organizations. Traffic classification plays a major role in such LI solutions [30].

Traditionally, traffic classification solutions rely either on transport-layer well-known port numbers [62] or on network packet payloads. Using port numbers is dependent on the assumption that services actually consistently use well-known TCP or UDP port numbers. In practice, applications increasingly use non-standard port numbers [91]. Performing deep-packet inspection (DPI) on the packet's payload is not only computationally expensive, it also requires the packets to be unencrypted. Today, an increasing number of services pervasively use encryption [98]. Modern, often machine-learning based approaches can alleviate these issues by using more sophisticated behavior-based traffic classification techniques. These systems, however, generally rely on packet-level telemetry data. We further elaborate on these techniques and the need for packet-level network telemetry in section 2.3.2.

2.2.4 Intrusion Detection

Another task that traditionally relies on the analysis of packet payloads is intrusion detection. With the number of security breaches and attacks to networks and IT infrastructure continuously growing, intrusion detection has become one of the most important tasks within the operation of computer networks [100, 162, 118].

Most intrusion detection systems (IDS) can be classified as either signature-based or anomaly-based. Signature-based IDS provide no protection against zero-day attacks as they compare packet features or byte sequences with a database of known attack signatures. This naturally requires the attack to be known ahead of time. While IT security companies are relatively quick in releasing updated signatures for new attacks, there can still be a considerable window during which a new attack cannot be detected by such systems [162].

Anomaly-based intrusion detection often leverages learning-based approaches to establish a model of trustworthy network traffic patterns and characteristics. Deviations from this model are then considered anomalies and are candidates for intrusions. Due to the rapid development of new attacks, malware, and viruses, anomaly-based approaches are a promising alternative to traditional, signature-based systems. However, their operation is significantly more complex and often require packet-level records. Anomaly-based intrusion detection is an active area of ongoing research [118, 109, 132, 47].

2.3 Emerging Requirements and Challenges

The field of network monitoring is currently undergoing a rapid evolution. Today's networks are operating at unprecedented packet and byte rates. Network traffic rates continue to grow [55] and, as a result, today's network monitoring solutions compromise in different ways in an attempt to keep up with the speeds of modern network deployments. Furthermore, through increased complexity in modern computer networks and the raise of security threats, monitoring systems underlie new requirements. In particular in regard to the need for packet-level monitoring data, these new requirements even exacerbate the effect of previously introduced compromises like flow-level data aggregation.

2.3.1 Traffic Volume

Modern data center and wide-area networks operate at traffic rates of several Terabits per second or hundreds of millions of packets per second, posing significant challenges for network telemetry

Cluster	Database	Web	Hadoop
Number of servers	4309	13108	6898
Number of racks	155	324	365
Number of pods	4	8	9
Average traffic [M packets/s]	16.56	706.61	90.85
Peak traffic rate [M packets/s]	21.66	961.26	168.42

Table 2.1: Facebook 24 hour datacenter trace [144, 69] statistics.

and analytics. Apart from the computing requirements, it is common for modern data-intensive applications that data sets exceed the storage resources (both persistent and memory) of a single compute node by several orders of magnitude making distributed data-heavy computation inevitable [27, 129]. As a consequence, significant amounts of data constantly need to be transferred over the network. Emerging technologies such as *RDMA over converged Ethernet* [168] or *NVMe over Fabrics* [169] enable and accelerate this trend.

Data Center and WAN Packet Rates

To give concrete numbers, we analyzed a data set from Facebook [69] described in [144], as well as packet traces from a 10 Gb Ethernet Internet backbone link between Seattle and Chicago collected by the Center for Applied Internet Data Analysis (CAIDA) [4] in 2015 [44]. We use these traces for various performance and scalability evaluations throughout this work (see sections 3.8 and 4.7). As we do not focus on technologies relying on *deep packet inspection* (DPI), packet rates are generally the more important metric compared to byte rates for our purposes.

For the first data set, table 2.1 summarizes the traffic rates within the entire cluster for three different clusters in a Facebook datacenter averaged over a 24 hour time period. A cluster here is a collection of servers spanning multiple racks that run the same class of applications. The traffic within the Hadoop cluster mostly consists of intra-cluster traffic while servers in database and web clusters mostly communicate across cluster boundaries or communicate with hosts on the public Internet (e.g., in the case of customer-facing services).

Our second dataset of network traces contains anonymized packet-level data collected by

Month	Direction	Average Pkt./s	Average Bits/s
Feb.	A	328.43 K	2.11 G
Feb.	B	613.07 K	4.36 G
May	A	301.64 K	1.88 G
May	B	383.20 K	2.59 G
Sep.	A	313.59 K	1.78 G
Sep.	B	403.90 K	2.93 G
Dec.	A	438.00 K	2.52 G
Dec.	B	420.75 K	3.14 G

Table 2.2: Statistics for CAIDA 2015 Passive Internet Core Traces [44]

CAIDA in 2015 [44]. Each trace covers the same one hour timeframe during different months of the year. The data was collected at a Equinix point of presence (PoP) in Chicago. Packet rates range from around 300K per second to around 600K per second. While these traffic rates are significantly lower compared to those in the Facebook traces, it is important to note that these traces only cover a single 10GbE link as opposed to the aggregate traffic across a cluster.

Compromising on Flexibility and Fidelity

In order to deal with such traffic rates, existing monitoring systems compromise along three competing axes: flexibility, fidelity, and performance. Flexibility describes how adaptable a network analytics platform is to evolving user requirements. Fidelity refers to the detail of data provided by the telemetry system. Performance and scalability describes how much network traffic a monitoring system can sustain as a function of the deployed resources and whether the system can easily scale to absorb more traffic or run more computationally expensive applications.

First, prior work that offloads analytics to the data plane sacrifices flexibility due to inherent limitations of programmable line-rate hardware. Such programmable forwarding engines (PFE) have constrained hardware resources in terms of Random Access Memory (RAM) and Arithmetic Logic Units (ALU) restricting the complexity of applications and limiting the possible number of parallel monitoring applications [39, 38]. As a result of this rigid hardware design, programming models for these devices are also restrictive and only facilitate a subset of analytical tasks [122, 185, 107]. Lastly, changing the program running on a PFE (and potentially even compiling it first), can

result in data plane downtimes of several seconds.

Second, filtering, aggregation, and sampling, have been the de-facto standards for decades. These approaches, of course, compromise on data fidelity. Aggregation reduces the load of the analytics system at the cost of granularity, as per-packet data is reduced to groups of packets in the form of sums or counts [31, 83]. Sampling and filtering reduces the number of packets or flows to be analyzed. Sampling, however, increases the chance of missing critical information while filtering restricts the set of possible applications [164, 146].

Third, existing analytics solutions (often based on the stream processing paradigm) provide high flexibility and programmability, but compromise on performance. For example, Cisco Tetration [136], OpenSOC [10] and NetQRE [186] report processing rates in the order of a couple of million network packets per second per server. General purpose stream processing systems, such as Spark [187], report higher processing rates around two to three million events per second per core. Today's cloud-scale computer networks, however, run switches that process several hundreds of millions of packets per second. Scaling these existing systems to these traffic rates would literally require racks full of servers.

In conclusion, existing systems either use strategies to reduce the load of the network analytics platform (e.g., by hardware offload or event rate reduction) hurting flexibility and capability, or, alternatively, provide high flexibility through programmable software analytics but then compromise on performance. We further elaborate on these trade-offs in section 4.2.1.

2.3.2 Packet-Level Data

Networks are not only rapidly growing in size and traffic volume, but also in complexity. While network virtualization and SDN make network operations more flexible, they also introduce complexity by enabling multi-layer network topologies and extensive use of network virtualization which can make troubleshooting more difficult. Early, but promising, advances in intent-based networking remedy these issues but require more fine-grained, close-to-real-time insight into network operations posing new challenges by requiring high-fidelity measurements.

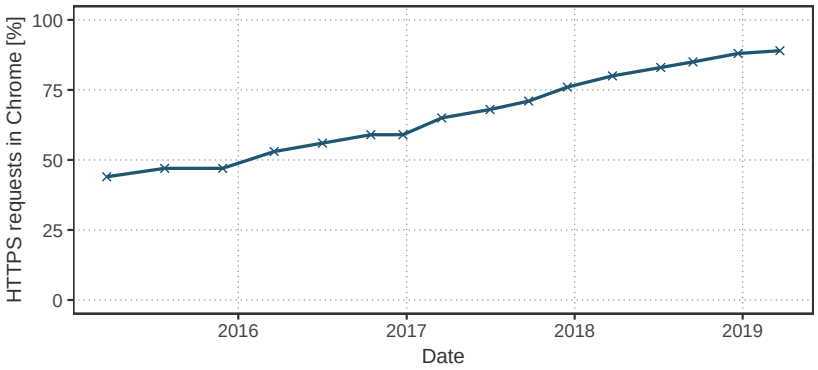


Figure 2.3: Percentage of Pages loaded over HTTPS in Google Chrome in the United States [76]

Moreover, while traditional intrusion detection systems leveraging deep packet inspection were never able to scale to very high traffic rates, DPI also becomes decreasingly effective as more and more traffic is encrypted [76]. Researchers propose behavioral analysis of network traffic leveraging machine learning as a solution to this problem [109, 126]. While this approach generally does not require insight into the packet payload, it requires detailed features of individual packets to work, such as packet inter-arrival times and individual byte counts. Current solutions do not provide such fine-grained, packet-level insight at high rates.

Traffic Classification and Intrusion Detection

In sections 2.2.3 and 2.2.4, we introduced both traffic classification and intrusion detection as popular applications for network monitoring technologies. As previously mentioned, both applications traditionally rely on application-layer analysis and deep packet inspection [72].

These approaches were based on a series of related assumptions [126]. The first assumption is, that a third party inspecting the traffic which is unaffiliated with either the source or recipient is able to inspect each packet's payload. The second assumption is that the classified or detection system maintains a database of all application-layer signatures of applications or attacks, respectively. Naive approaches for traffic classification furthermore rely on the fact that traffic uses well-known port numbers [62].

With the pervasive deployment of various encryption technologies [98], the first assumption is becoming increasingly invalid. For example, the fraction of TLS traffic handled by Google's

Chrome web browser doubled between March 2015 and March 2019 [76]. This trend is depicted in figure 2.3. DPI technologies depend on the fact that the packet's payload and L4 header fields are not obfuscated. Additionally, governments are increasingly imposing regulations constraining the ability for third parties to inspect the payload of network packets at all [177].

With the growing number of network applications and threats, the second assumption is also increasingly hard to keep valid. It is becoming harder to maintain comprehensive databases of attack signatures and keeping them current, causing increasing vulnerability to zero-day attacks [127]. These requirements only exacerbate the main problem with payload-based analytics strategies, which is the high computational complexity and therefore high cost [106, 72].

As a result, researchers have proposed directions toward detecting anomalies and classifying traffic based on statistical features of flow or packet data without the need for packet payload inspection [126, 108, 109, 188, 63]. The majority of these approaches leverage machine-learning (ML) technologies. Examples for such features include packet inter-arrival times, individual packet sizes, flow durations, or TCP flags. Machine-learning systems are then trained to associate known conditions, such as traffic classes or anomalies, with a particular set or distribution of features. Alternatively, such fine-grained, packet-level features can also be used to iteratively construct a baseline model of the network traffic. If the inspected traffic deviates from this model significantly, an anomaly and, therefore, a potential intrusion is detected.

Packet-level features that are commonly used for intrusion detection or traffic classification include:

- packet inter-arrival times [113, 188, 120, 134, 166, 125, 63, 28, 179]
- packet lengths statistics [113, 188, 143, 120, 134, 166, 125, 63, 28, 179]
- TCP advertised window bytes [134, 28]
- TCP flags [134, 28]
- inter-packet byte counts [166]
- packet arrival order [63]
- various metrics derived from the above (e.g., Fourier transforms of inter-arrival times, flow idle times, mean packet sizes, flow duration, number of TCP data packets) [113, 120, 143,

134, 166, 125, 63, 28, 179]

We can see that the most commonly used packet-level features used for anomaly detection or traffic classification are packet inter-arrival times, as well as individual byte counts. Additionally, TCP-specific features such as the set of TCP flags set in a packet, or TCP window advertisements are commonly used. Flow-level telemetry systems generally do not include these features. At the flow level, packet lengths are aggregated to the total number of bytes in the flow and time stamps are only provided for the first and last packet of the flow. Some versions of IPFIX can include all TCP flags seen in a flow aggregated by the bitwise, logical `AND` operation [57].

Network Performance Monitoring

In section 2.2.1, we explained why performance monitoring and network debugging is a crucial task in the operation of modern high-performance and high-availability networks. Many data center networks guarantee availability of 99.999% or higher over a year. 99.999% availability, for example, equates to a maximum allowable downtime of 5.26 minutes per year, or 864 milliseconds per day. Even with redundant systems, operators must monitor the health of their networks very closely to avoid catastrophic failures that may result in an outage under all circumstances [193, 121, 78].

Fine-grained performance monitoring therefore is an increasingly important part of network operation. The main challenge here is that many failures only happen intermittently or only affect single packets. These failures, however, can significantly impact end user experience through inflated latency, which can have notable business impact over time [42, 159]. Few intermittent packet drops or other faults can also be indicators for the network reaching capacity or faulty equipment. Given the above mentioned quality of service and availability guarantees cloud and service providers offer their customers, it is imperative to detect even smaller issues early, analyze the situation, and take steps in order to prevent the situation from escalating to more catastrophic failures.

Using flow-level measurements or switch counters for this use-case is insufficient. For example, so called silent packet drops are not visible through packet drop counters on switches [193].

Packet-level telemetry data (e.g., including TCP sequence numbers), however, can be used to detect such silent drops. In general, congestion within the network is hard to analyze in detail without switch-level information, such as detailed data plane queueing information. Developers of modern programmable switching platforms have access to fine-grained performance data, such as nanosecond enqueue and dequeue timestamps and ingress and egress queue occupancy counters [140, 93, 122, 61]. Nevertheless, given the short-lived character of many performance-related anomalies, it is not sufficient to expose switch-level information in an aggregated representation. Flow records, for example, can include hundreds of packets and span time frames of several minutes, such that an aggregate statistic (e.g., sum or mean) of this performance data is not granular enough to properly investigate the problem [133].

2.4 Enabling Technologies and Opportunities

Through changing requirements, network monitoring and analytics systems face new challenges as explained in section 2.3. New technologies in the field of high-performance packet processing, both in data plane devices and in general-purpose computers, alleviate some of these challenges by enabling packet processing at unprecedented rates with a high degree of programmability. Using these technologies in an intelligent and efficient way as we demonstrate it in this work, paves the way for cost-efficient, high-throughput monitoring solutions. In this section we introduce these technologies in detail.

2.4.1 Programmable Data Planes

Traditional SDN technologies are designed around match and action patterns in the data plane based on packet header fields [77, 70, 20]. The most prominent implementation of this architecture is OpenFlow [114]. The current version of OpenFlow (1.5) supports 44 different match fields [74]. Still, most hardware implementations of the OpenFlow specification support anything between 12 and approximately 20 header fields [101]. To our knowledge there is no hardware implementation of the full OpenFlow 1.5 specification today. With this trend of slow adoption, there is (on top

of technical challenges) a demand for a paradigm shift. Instead of continuously extending the OpenFlow protocol, it should be possible to match packets on arbitrary bit patterns. As this is a relatively trivial task in software, it is complicated at high packet rates in hardware and at scale.

2.4.1.1 Programmable Switches

Recent research exploring programmable hardware has shown that this vision of flexible packet forwarding can actually be achieved at Terabit speeds using custom and specialized ASIC designs [40, 99, 124, 3]. As these hardware designs continue emerging, work on programming abstractions and a variety of use cases has been presented by academia and industry [122, 89, 90, 155, 148, 94]. Apart from [158], which proposes this concept in the domain of network processors, P4 [38] is the most prominent example of work in this area as it aims to cover the entire processing range from ASICs, over FPGAs, NPUs to CPUs using a common language interface.

Using the P4 language, an abstract forwarding model can be described which defines how a packet is traversing a data plane device, which bit ranges need to be extracted and matched and how a packet is initially parsed and further processed. This abstract forwarding model can then be compiled using a switch-vendor specific compiler to a target-dependent version which then provides platform-compatible description of the processing flow. In order to describe a foundation of compatibility requirements between data plane programs and target hardware, the *protocol-indepenent switch architecture (PISA)* has been proposed [124]. PISA describes a standard programmable pipeline architecture as implemented in Barefoot Network's Tofino programmable switching chip [124].

Figure 2.4 depicts the pipeline layout described in PISA. In this architecture, upon receiving a packet, the packet is parsed using a programmable parser abstracted as a finite state machine (FSM). Individual packet headers (e.g., for IP, TCP, or entirely user-defined protocols) can then be matched by a series of match-action tables. These tables are divided into an ingress and an egress pipeline. Each pipeline has multiple stages that each consist of a combination of SRAM and TCAM for the match part and a ALUs for realizing basic actions (e.g., performing arithmetic or

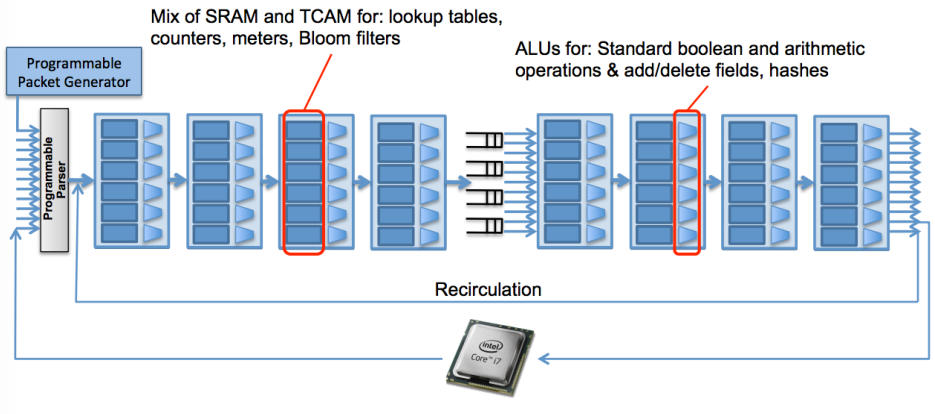


Figure 2.4: Protocol Independent Switch Architecture

changing or inserting header fields) on the packet. The different stages and pipelines run in parallel and guarantee line-rate processing of packets. A recirculation port can be use to pass a packet through the pipelines multiple times. Additionally, a CPU port can be used to send a packet to the control plane for software processing [156]. In all implementations, PISA match-action tables are programmable at switch runtime through different vendor-specific wire protocols or RPC interfaces in a manner comparable to the OpenFlow model.

2.4.1.2 Programmable Network Interface Cards

Similar to programmable switching platforms, also network interface cards (NIC) become increasingly programmable. These devices generally expose a less rigid architecture (compared to PISA) and can be programmed using more flexible tools, such as constrained versions of the C programming language.

Network Processing Units (NPU)

Network processing units (sometimes also called network flow processors (NFP)) are specialized integrated circuits that typically have a high amount of logical cores to enable a high degree of parallel processing. The Netronome NFP-4000 NPU, for example has 48 cores dedicated to per-packet processing and 60 cores for flow processing with around 19 MB of on-chip memory. This NPU can process up to 100Gb per second or 148M packets per second of traffic and allows for

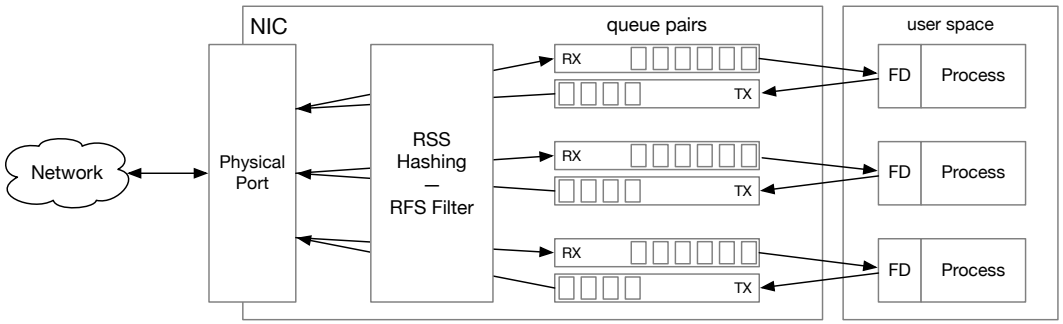


Figure 2.5: NIC Receive-Side Scaling (RSS) and Receive Flow Steering (RFS) Architecture

flexible implementation of security, routing, or monitoring applications using a subset of the C programming language [123]. Our network telemetry system (described in chapter 3) provides an implementation for both the Barefoot Tofino PFE and the Netronome NFP-4000 NPU.

Receive-Side Scaling (RSS) and Receive Flow Steering (RFS)

Modern network interface cards have multiple ingress and egress packet buffers that can be addressed individually through kernel-bypass networking technologies such as DPDK [5] or Netmap [141]. Receive-Side Scaling (RSS) and the related technology of Receive Flow Steering (RFS) are NIC driver features that allow distributing received packets across different physical memory buffers on the card. Using plain RSS, incoming packets can be distributed across these buffers based on a hash function over some part of the packet's header space [110]. RFS allows for slightly more advanced logic by specifying match rules over packet header fields for the individual queues [85]. Applications leveraging RSS or RFS can request file descriptors that only cover a specified subset of the NICs queues. Consequently, these different processes can also bind to separate CPUs, such that the NIC can effectively steer packets to a specific CPU core for processing. The resulting architecture is depicted in figure 2.5. We use this feature heavily in our network analytics system (described in chapter 4).

2.4.2 Parallel Streaming Analytics

While stream processing systems have been used for years to process large amounts of data in real-time, e.g., in the field of processing data gathered from sensor networks [171, 139], general

stream-processing frameworks have gained immense popularity in the past couple of years driven primarily by developments in big web properties, such as Twitter, Facebook, and LinkedIn.

Real-time data is typically ill-suited for many of the analytics systems that are designed to process batched data. Existing systems such as Hadoop [1], make use of data parallelism by splitting data sets and distributing them across a cluster of nodes. In contrast, real-time analytics systems distribute data to computation as it is generated, i.e., it is processed as a stream. Today, there are a wide range of stream processing frameworks that can be used to implement real-time analytics systems, such as Storm [16] (from Twitter), Samza [15] (from LinkedIn), S4 [14] (from Yahoo), Flink [24], or Spark Streaming [26].

The prevalent abstraction provided by these systems is a directed graph of processing elements, where users specify the processing elements, the connections between them, and data flows along the edges. Each node is a processing element that can execute arbitrary code on the data (e.g., compute statistics, transform it, generate new data, or store it).

As an example of the internal network within one of these systems, we'll focus on the Storm real-time processing framework. When a topology (a graph of processing elements) is created, Storm distributes the different processing elements throughout the cluster, with different numbers of spouts (processing element that serves as a source of data) and bolts (executes arbitrary code) being run on each server in the cluster, depending on various settings in the topology. Each server then establishes ring buffers [170] that queue data that is to be passed between different spouts and bolts on the same server. When data is generated by a processing element, the Storm process running on the server where the data was generated must determine if the destination is a local element or is being run on a remote server. Once this determination is made, the data is either placed on the input ring buffer or transmitted over the network, where it is forwarded locally to the correct element. There is no interaction with the network, though custom scheduling or placement can be implemented, e.g., to optimize many potential metrics, such as bandwidth and latency [22].

The topology abstraction provided by Storm allows users to process real-time data in a processing graph, and provides simple mechanisms that allow for a program to be split into separate

parts that can be distributed over a cluster. In Storm and other systems, however, topologies are created with a static internal forwarding configuration when they are launched and are very difficult to modify without restarting the entire topology – causing a loss of data, loss of state, and delay of real-time analysis. These limitations limit the flexibility of the systems, as there are very few mechanisms for the topologies to react to changes, such as changes in load or network conditions, or desires to introduce new processing to act on live data without loss of information or state.

Chapter 3

Packet-Level Network Telemetry

Measurement plays a key role in network operation and management. An important but unaddressed practical requirement in high speed networks is supporting concurrent applications with diverse and potentially dynamic measurement objectives. In this chapter, we introduce *Flow, a hardware-accelerated, analytics-aware telemetry system for efficient, concurrent, and dynamic measurement. The design insight is to carefully partition processing between switch ASICs and application software. In *Flow, the switch ASIC implements a pipeline that exports telemetry data in a flexible format that allows applications to efficiently compute many different statistics. Applications can operate concurrently and dynamically on configurable packet streams without impacting each other. We implement *Flow as a line rate P4 program for a 3.2 Tb/s commodity switch and evaluate it with four example monitoring applications.

3.1 Introduction

For high speed networks, which have 100 Gb/s links and multi-Tb/s switches, it is challenging to design measurement systems that support a variety of applications without compromising on important practical requirements. Traditional switch hardware is inflexible and can only measure coarse grained statistics [56, 135], while measurement in software is very expensive to scale [155].

Advances in switch hardware, however, are presenting new opportunities. As the chip space and power cost of programmability drops [152, 37], switches are quickly moving towards reconfigurable ASICs [124, 130] that are capable of custom packet processing at high line rates. Recent

telemetry systems [155, 122] have shown that these programmable forwarding engines (PFEs) can implement custom streaming measurement queries for fine-grained traffic and network performance statistics.

A remaining open question though is whether telemetry systems can harness the flexibility and performance of PFEs while also meeting requirements for practical deployment. Current PFE accelerated telemetry systems [155, 122] focus on efficiency, compiling queries to minimize workload on servers in the telemetry infrastructure. Efficiency matters, but compiled queries do not address two other practical requirements that are equally important: concurrent measurement and dynamic queries.

First, support for concurrent measurement. In practice, there are likely to be multiple applications measuring the network concurrently, with queries for different statistics. A practical telemetry system needs to multiplex the PFE across all the simultaneously active queries. This is a challenge with compiled queries. Each query requires different computation that, given the line-rate processing model of a PFE [152], must map to dedicated computational resources, which are limited in PFEs.

Equally important for practical deployment is support for dynamic querying. As network conditions change, applications and operators will introduce or modify queries. A practical telemetry system needs to support these dynamics at runtime without disrupting the network. This is challenging with compiled PFE queries because recompiling and reloading the PFE is highly disruptive. Adding or removing a query pauses not only measurement, but also forwarding for multiple seconds.

Introducing *Flow

We introduce *Flow, a practical PFE-accelerated telemetry system that is not only flexible and efficient, but also supports concurrent measurement and dynamic queries. Our core insight is that concurrency and disruption challenges are caused by compiling too much of the measurement query to the PFE, and can be resolved without significant impact to performance by carefully lifting parts of it up to software.

At a high level, a query can be decomposed into three logical operations: a *select* operation that determines which packet header and metadata features to capture; a *grouping* operation that describes how to map packets to flows; and a *aggregation function* that defines how to compute statistics over the streams of grouped packet features. The primary benefit of using the PFE lies in its capability to implement the select and grouping operations efficiently because it has direct access to packet headers and low latency SRAM [122]. The challenge is implementing aggregation functions in the PFE, which are computationally complex and query dependent.

*Flow is based on the observation that for servers, the situation is exactly reversed. A server cannot efficiently access the headers of every packet in a network, and high memory latency makes it expensive to group packets. However, once the packet features are extracted and grouped, a server can perform more coarse-grained grouping and mathematical computation very efficiently.

*Flow's design, depicted in figure 3.3, plays to the strengths of both PFEs and servers. Instead of compiling entire queries to the PFE, *Flow places parts of the select and grouping logic that are common to all queries into a match+action pipeline in the PFE. The pipeline operates at line rate and exports a stream of records that software can compute a diverse range of custom streaming statistics from without needing to group per-packet records. This design maintains the efficiency benefits of using a PFE while eliminating the root causes of concurrency and disruption issues. Further, it increases flexibility by enabling more complex aggregation functions than a PFE can support.

Grouped Packet Vectors

To lift the aggregation function off of the PFE, *Flow introduces a new record format for telemetry data. In *Flow, PFEs export a stream of *grouped packet vectors* (GPVs) to software processors. A GPV contains a flow key, e.g., IP 5-tuple, and a variable-length list of packet feature tuples, e.g., timestamps and sizes, from a sequence of packets in that flow.

Each application can efficiently measure different aggregate statistics from the packet feature tuples in the same GPV stream. Applications can also dynamically change measurement without impacting the network, similar to what a stream of raw packet headers [80] would allow, but without

the cost of cloning each packet to a server or grouping in software.

Dynamic in-PFE Cache

Switches generate GPVs at line rate by compiling the *Flow cache to their PFEs, alongside other forwarding logic. The cache is an append only data structure that maps packets to GPVs and evicts them to software as needed.

To utilize limited PFE memory, e.g., around 10MB as efficiently as possible, we introduce a key-value cache that supports dynamic memory allocation and can be implemented as a sequence of match+action tables for PFEs. It builds on recent match+action implementations of fixed width key-value caches [122, 90, 155] by introducing a line rate memory pool to support variable sized entries. Ultimately, dynamic memory allocation increases the average number of packet feature tuples that accumulate in a GPV before it needs to be evicted, which lowers the rate of processing that software must support.

We implemented the *Flow cache for a 100BF-32X switch, a 3.2 Tb/s switch with a Barefoot Tofino [124] PFE that is programmable with P4 [38]. The cache is compiler-guaranteed to run at line rate and uses a fixed amount of hardware resources regardless of the number or form of measurement queries.

Contributions

This chapter has four main contributions. First, the idea of using grouped packet vectors (GPVs) to lift the aggregation functions of traffic queries out of data plane hardware. Second, the design of a novel PFE cache data structure with dynamic memory allocation for efficient GPV generation. Third, the evaluation of a prototype of *Flow implemented on a readily available commodity P4 switch. Finally, an interface that allows for practical load balancing and record distribution to parallel analytics pipelines.

3.2 Background

In this section, we describe design goals that are important for practical deployment of a monitoring system for PFE supported traffic statistic queries. We focus supporting queries that

	Efficient	Flexible	Concurrent	Dynamic
Netflow	✓		✓	✓
Software		✓	✓	✓
PFE Queries	✓	✓		
***Flow**	✓	✓	✓	✓

Table 3.1: Practical requirements for PFE supported network queries.

are *efficient*, *expressive*, *concurrent*, and *dynamic*.

3.2.1 Design Goals

*Flow is designed to meet four design goals that are important for a practical PFE accelerated telemetry system.

Efficiency

We focus on efficient usage of processing servers in the telemetry and monitoring infrastructure of a network. Efficiency is important because telemetry and monitoring systems need to scale to high throughputs [155] and network coverage [103]. An inefficient telemetry system deployed at scale can significantly increase the total cost of a network, in terms of dollars and power consumption.

Flexibility

A flexible telemetry system lets applications define the aggregation functions that compute traffic and data plane performance statistics. There are a wide range of statistics that are useful in different scenarios and for different applications. Customizable aggregation functions allow a telemetry system to offer the broadest support.

We break flexibility into three dimensions: *selection flexibility*, *grouping flexibility*, and *aggregation flexibility*. Selection flexibility is the quantity and variety of packet header and processing metadata fields, e.g., queue lengths, that can be measured. Grouping flexibility is the capability for applications to specify the classes that packets are grouped into, e.g., TCP flows based on IP 5-tuple, and the timescales over which the metrics are computed, e.g., for complete flows, flowlets, or individual packets. Aggregation flexibility is the capability to support metrics with custom and potentially complex aggregation functions that perform advanced mathematical operations or

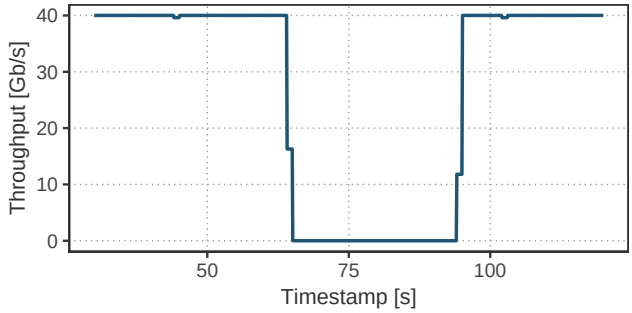


Figure 3.1: Network disruption from recompiling a PFE (Example)

iterating over features from each packet multiple times.

Flexibility is also important for supporting future applications that may identify new useful metrics and systems that apply machine learning algorithms to analyze the network in many dimensions [126].

Concurrency

Concurrency is the capability to support many measurement queries at the same time. Concurrency is important because different applications require different statistics and, in a real network, there are likely to be many types of applications in use.

Consider a scenario where an operator is debugging an incast situation [48] and a network-wide security system is auditing for compromised hosts [109]. These applications would ideally run concurrently and have the need to measure different statistics. Debugging, for example, may benefit from measuring the number of simultaneously active TCP flows in a switch queue over small epochs, while a security application may require per-flow packet counters and timing statistics.

Dynamic Queries

Support for dynamic queries is the capability to introduce or modify new queries at run time. It is important for monitoring applications, which may need to adapt as network conditions change, or themselves be launched at network run-time. Dynamic queries also enable interactive measurement [122] that can help network operators diagnose performance issues, e.g., which queue is dropping packets between a pair of hosts?

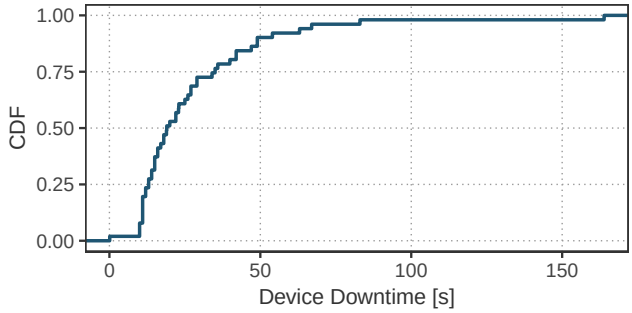


Figure 3.2: Network disruption from recompiling a PFE (CDF)

3.2.2 Prior Telemetry Systems

Prior telemetry systems meet some, but not all, of the above design goals, as summarized in Table 3.1. We group prior systems based on where they implement the logic to select packet features, group packets into flows, and apply aggregation functions.

NetFlow Hardware

Many switches integrate hardware to generate NetFlow records [52] that summarize flows at the granularity of IP 5-tuple. NetFlow records are compact because they contain fully-computed aggregate statistics. ASICs [194, 54] in the switch data path do all the work of generating the records, so the overhead for monitoring and measurement applications is low. NetFlow is also dynamic. The ASICs are not embedded into the forwarding path, so a user can select different NetFlow features without pausing forwarding.

However, NetFlow sacrifices flexibility. Flow records have a fixed granularity and users choose statistics from a fixed list. Newer NetFlow ASICs [54] offer more statistics, but cannot support custom user-defined statistics or different granularities.

Software Processing

A more flexible approach is mirroring packets, or packet headers, to commodity servers that compute traffic statistics [53, 68, 64, 111]. Servers can also support concurrent and dynamic telemetry, as they are not in-line with data plane forwarding.

The drawback of software is efficiency. Two of the largest overheads for measurement in

software are I/O [141], to get each packet or header to the measurement process, and hash table operations, to group packets by flow [155, 122, 104]. To demonstrate, we implemented a simple C++ application that reads packets from a PCAP, using libpcap [18], and computes the average packet length for each TCP flow. The application spent an average of 1535 cycles per packet on hash operations alone, using the relatively efficient C++ `std::unordered_map` [34]. In another application, which computed average packet length over pre-grouped vectors of packet lengths, the computation only took an average of 45 cycles per packet.

The benchmarks illustrate that mathematical operations for computing aggregate statistics are not a significant bottleneck for measurement in software. Modern CPUs with vector instructions can perform upwards of 1 trillion floating point operations per second [117].

PFE Compiled Queries

Programmable forwarding engines (PFEs), the forwarding ASICs in next generation commodity switches [124, 46, 130, 123, 51, 178], are appealing for telemetry because they can perform stateful line-rate computation on packets. Several recent systems have shown that traffic measurement queries can compile to PFE configurations [155, 122]. These systems allow applications (or users) to define custom statistics computation functions and export records that include the aggregate statistics. Compiled queries provide efficiency and flexibility. However, they are not well suited for concurrent or dynamic measurement.

Concurrency is a challenge because of the processing models and computational resources available in a PFE. Each measurement query compiles to its own dedicated computational and memory resources in the PFE, to run in parallel at line rate. Computational resources are extremely limited, particularly those for stateful computation [152], making it challenging to fit more than a few queries concurrently.

Dynamic queries are a challenge because PFEs programs are statically compiled into configurations for the ALUs in the PFE. Adding a compiled query requires reloading the entire PFE program, which pauses all forwarding for multiple seconds, as figure 3.2 shows. While it is possible to change forwarding rules at run-time to direct traffic through different pre-compiled functions,

the actual computation can only be changed at compile time.

3.3 PFE Accelerated Telemetry

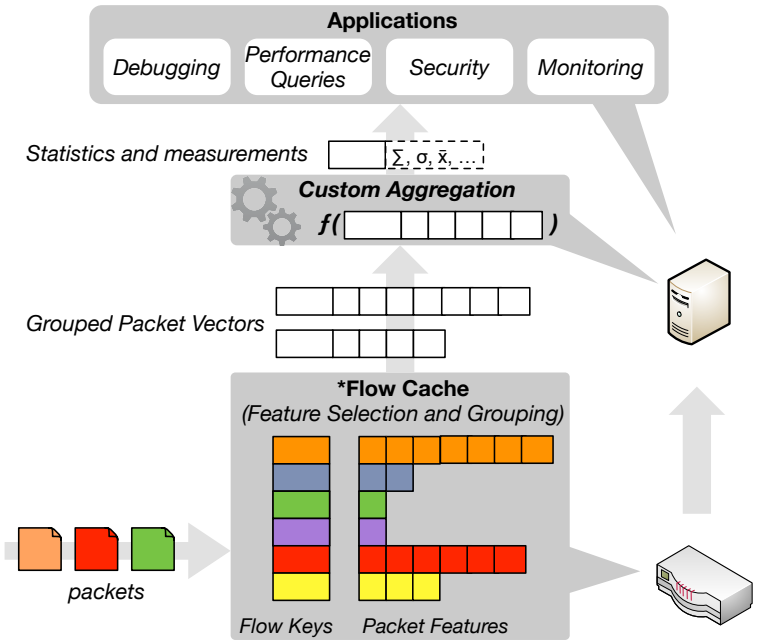


Figure 3.3: Overview of *Flow.

*Flow is a PFE accelerated telemetry system that supports *efficient*, *flexible*, *concurrent*, and *dynamic* network measurement. It gains efficiency and flexibility by leveraging the PFE to select features from packet headers and group them by flow. However, unlike prior systems, *Flow lifts the complex and measurement-specific statistic computation, which are difficult to support in the PFE without limiting concurrent and dynamic measurement, up into software. Although part of the measurement is now in software, the feature selection and grouping done by the PFE reduces the I/O and hash table overheads significantly, allowing it to efficiently compute statistics and scale to terabit rate traffic using a small number of cores.

In this section, we overview the architecture of *Flow, depicted in figure 3.3.

Grouped Packet Vectors

The key to decoupling feature selection and grouping from statistics computation is the *grouped packet vector* (GPV), a flexible and efficient format for telemetry data streamed from switches. A GPV stream is flexible because it contains per-packet features. Each application can measure different statistics from the same GPV stream and dynamically change measurement as needed, without impacting other applications or the PFE. GPVs are also efficient. Since the packet features are already extracted from packets and grouped in flowlets by IP 5-tuple, applications can compute statistics with minimal I/O or hash table overheads.

***Flow Telemetry Switches**

Switches with programmable forwarding engines [124, 152] (PFEs) compile the *Flow cache to their PFEs to generates GPVs. The cache is implemented as a sequence of match+action tables that applies to packets at line rate and in parallel with other data plane logic. The tables extract feature tuples from packets, insert them into per-flow GPVs, and stream the GPVs to monitoring servers using application-specific load balancing and filtering.

GPV Processing

*Flow integrates with analytics software through a thin *Flow client that configures the mapping of per-application NIC queues to analytics processes. Each analytics application defines its own statistics to compute over the packet tuples in GPVs and can dynamically change them as needed. Since the packet tuples are pre-grouped, the computation is extremely efficient because the bottleneck of mapping packets to flows is removed. Further, if fine granularity is needed, the applications can analyze the individual packet feature themselves, e.g., to identify the root cause of short lived congestion events.

GPV Serialization

We implemented adapters to serialize GPVs to two formats: a custom protocol buffer [75] and a simple binary format consisting of a flat array of C structs. The protocol buffer format is well suited to long term storage in scale out databases, e.g., Redis [13]. The binary format is used for buffering and microbenchmarks.

DFR Trace	NetFlow Trace	Packet Header Trace	
Record Count	49 M	49 M	1098 M
File Size	9.4 GB	2.3 GB	72 GB

Table 3.2: Record count and sizes for a 1 hour 10 Gbit/s core Internet router trace [44]. DFRs contain IP 5 tuples, packet sizes, timestamps, TCP flags, ToS flag, and TTLs

.

Legacy Applications

The *Flow processor can also push GPVs to any target, including legacy monitoring applications. It can export GPVs directly, or use adapters to convert them into another flow or packet record format. We also implemented an adapter to convert GPVs into NetFlow records with custom features.

3.4 Grouped Packet Vectors

*Flow exports telemetry data from the switch in the grouped packet vector (GPV) format, illustrated in Figure 3.4, a new record format designed to support the decoupling of packet feature selection and grouping from aggregate statistics computation. A grouped packet vector contains an IP 5-tuple flow key and a variable length vector of feature tuples from sequential packets in the respective flow. As Figure 3.4 shows, a GPV is a hybrid between a packet record and a flow record. It inherits some of the best attributes of both formats and also has unique benefits that are critical for the overall Toccoa architecture.

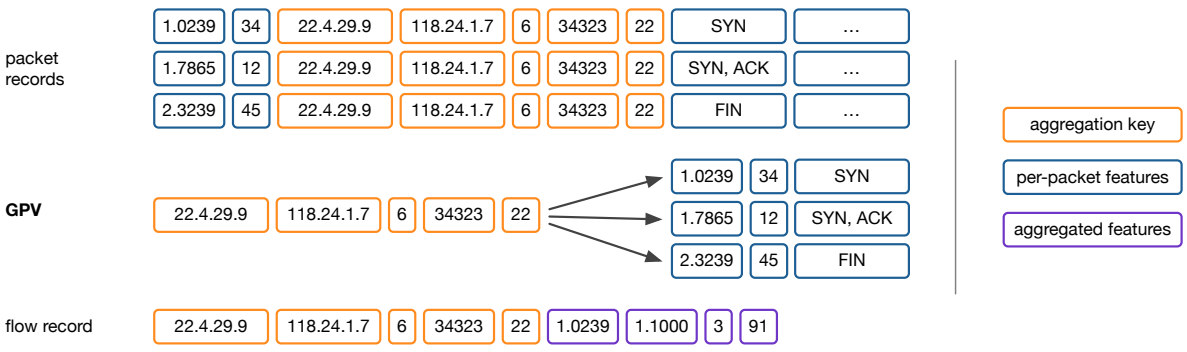


Figure 3.4: Comparison of grouped packet vectors, flow records, and packet records.

Similar to packet records, a stream of GPVs contains features from each individual packet. Unlike packet records, however, GPVs get the features to software in a format that is well suited for efficient statistics computation. An application can compute aggregate statistics directly on a GPV, without paying the overhead of receiving each packet, extracting features from it, or mapping it to a flow.

Similar to flow records, each GPV represents multiple packets and deduplicates the IP 5-tuple. They are approximately an order of magnitude smaller than packet header records and do not require software to perform expensive per-packet key value operations to map packet features to flows. Flow records are also compact and can be processed by software without grouping but, unlike flow records, GPVs do not lock the software into specific statistics. Instead, they allow the software to compute any statistics, efficiently, from the per-packet features. This works well in practice because many useful statistics derive from small, common subsets of packet features. For example, the statistics required by the 3 monitoring applications and 6 Marple [122] queries we describe in Section 3.7 can all be computed from IP 5-tuples, packet lengths, arrival timestamps, queue depths, and TCP sequence numbers.

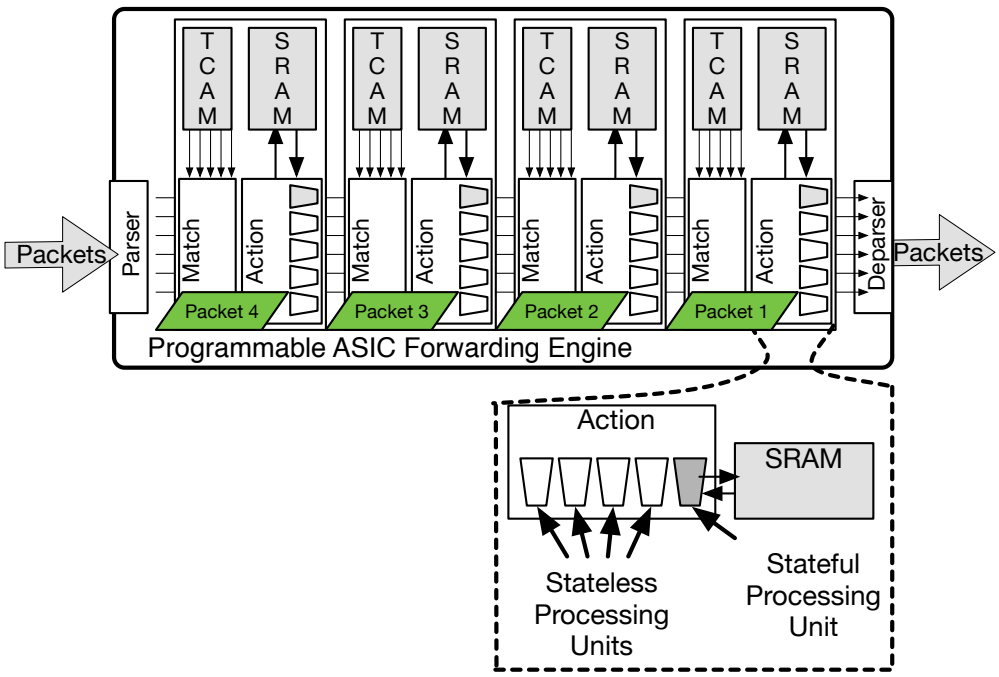


Figure 3.5: PFE architecture

3.5 Generating GPVs

The core of *Flow is a cache that maps packets to GPVs and runs at line rate in a switch programmable forwarding engine (PFE). A GPV cache would be simple to implement in software. However, the target platforms for *Flow are the hardware data planes of next-generation networks; PFE ASICs that process packets at guaranteed line rates exceeding 1 billion packets per second [152, 39, 49]. To meet chip space and timing requirements, PFEs significantly restrict stateful operations, which makes it challenging to implement cache eviction and dynamic memory allocation.

In this section, we describe the architecture and limitations of PFEs, cache eviction and memory allocation policies that can be implemented in a PFE, and our P4 implementation of the *Flow cache for the Tofino.

3.5.1 PFE Architecture

We already gave an overview of PISA and programmable forwarding engines in section 2.4.1. Figure 3.5 illustrates the architecture of a PFE ASIC in more detail. It receives packets from multiple network interfaces, parses their headers, processes them with a pipeline of match tables and action processors, and finally deparses the packets and sends them to an output buffer. PFEs are designed specifically to implement match+action forwarding applications, e.g., P4 [38] programs, at guaranteed line rates that are orders of magnitude higher than other programmable platforms, such as CPUs, network processors [178], or FPGAs, assuming the same chip space and power budgets. They meet this goal with highly specialized architectures that exploit pipelining and instruction level parallelism [152, 39]. PFEs make it straightforward to implement custom terabit rate data planes, so long as they are limited to functionality that maps naturally to the match+action model, e.g., forwarding, access control, encapsulation, or address translation.

It can be challenging to take advantage of PFEs for more complex applications, especially those that require state persisting across packets, e.g., a cache. Persistent arrays, called "register arrays" in P4 programs, are stored in SRAM banks local to each action processor. They are limited in three important ways. First, a program can only access a register array from tables and actions implemented in the same stage. Second, each register array can only be accessed once per packet, using a stateful ALU that can implement simple programs for simultaneous reads and writes, conditional updates, and basic mathematical operations. Finally, the sequential dependencies between register arrays in the same stage are limited. In currently available PFEs [124], there can be no sequential dependencies; all of the registers in a stage must be accessed in parallel. Recent work, however, has demonstrated that future PFEs can ease this restriction to support pairwise dependencies, at the cost of slightly increased chip space [152] or lower line rates [49].

3.5.2 Design

To implement the *Flow cache as a pipeline of match+action tables that can compile to PFEs with the restrictions described above, we simplified the algorithms used for cache eviction and memory allocation. We do not claim that these are the best possible heuristics for eviction and allocation, only that they are intuitive and empirically effective starting points for a variable width flow cache that operates at multi-terabit line rates on currently available PFEs.

Cache Eviction

The *Flow cache uses a simple evict on collision policy. Whenever a packet from an untracked flow arrives and the cache needs to make room for a new entry, it simply evicts the entry of a currently tracked flow with the same hash value. This policy is surprisingly effective in practice, as prior work has shown [105, 155, 122], because it approximates a least recently used (LRU) policy.

Memory Allocation

The *Flow cache allocates a narrow ring buffer for each flow, which stores GPVs. Whenever the ring buffer fills up, the cache flushes its contents to software. When an active flow fills its narrow buffer for the first time, the cache attempts to allocate a wider buffer for it, drawn from a pool with fewer entries than there are cache slots. If the allocation succeeds, the entry keeps the buffer until the flow is evicted; otherwise, the entry uses the narrow buffer until it is evicted.

This simple memory allocation policy is effective for *Flow because it leverages the long-tailed nature of packet inter-arrival time distributions [35]. In any given time interval, most of the packets arriving will be from a few highly active flows. A flow that fills up its narrow buffer in the short period of time before it is evicted is more likely to be one of the highly active flows. Allocating a wide buffer to such a flow will reduce the overall rate of messages to software, and thus its workload, by allowing the cache to accumulate more packet tuples in the ring buffer before needing to flush its contents to software.

This allocation policy also frees memory quickly once a flow's activity level drops, since frees happen automatically with evictions.

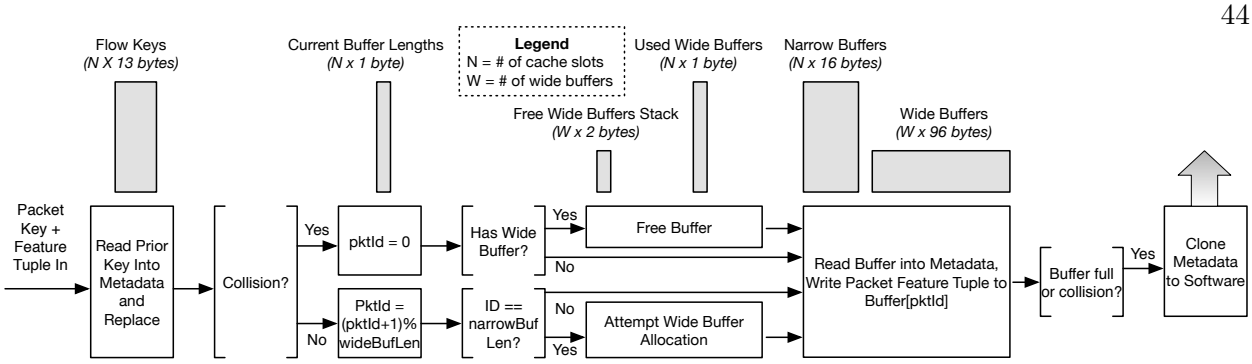


Figure 3.6: The *Flow cache as a match+action pipeline. White boxes represent sequences of actions, brackets represent conditions implemented as match rules, and gray boxes represent register arrays.

3.5.3 Implementation

Using the above heuristics for cache eviction and memory allocation, we implemented the *Flow cache as a pipeline of P4 match+action tables for the Tofino [124]. The implementation consists of approximately 2000 lines of P4 code that implements the tables, 900 lines of Python code that implements a minimal control program to install rules into the tables at runtime, and a large library that is autogenerated by the Tofino's compiler toolchain. The source code has been tested on both the Tofino's cycle-accurate simulator and a Wedge 100BF-32X.

Figure 3.6 depicts the control flow of the pipeline. It extracts a tuple of features from each packet, maps the tuple to a GPV using a hash of the packet's key, and then either appends the tuple to a dynamically sized ring buffer (if the packet's flow is currently tracked), or evicts the GPV of a prior flow, frees memory, and replaces it with a new entry (if the packet's flow is not currently tracked).

We implemented the evict on collision heuristic using simultaneous read / write operations when updating the register arrays that store flow keys. The update action writes the current packet's key to the array, using its hash value as an index, and reads the data at that position into metadata in the packet. If there was a collision, which the subsequent stage can determine by

comparing the packet's key with the loaded key, the remaining tables will evict and reset the GPV. Otherwise, the remaining tables will append the packet's features to the GPV.

We implemented the memory allocation using a stack. When a cache slot fills its narrow buffer for the first time, the PFE checks a stack of pointers to free extension blocks. If the stack is not empty, the PFE pops the top pointer from the stack. It stores the pointer in a register array that tracks which, if any, extension block each flow owns. For subsequent packets, the PFE loads the pointer from the array before updating its buffers. When the flow is evicted, the PFE removes the pointer from the array and pushes it back onto the free stack.

This design requires the cache to move pointers between the free stack and the allocated pointer array in both directions. We implemented it by placing the stack before the allocated pointer array, and resubmitting the packet to complete the free operation by pushing its pointer back onto the stack. The resubmission is necessary on the Tofino because sequentially dependent register arrays must be placed in different stages and there is no way to move "backwards" in the pipeline.

3.5.4 Configuration

Compile-time

The current implementation of the *Flow cache has three compile-time parameters: the number of cache slots; the number of entries in the dynamic memory pool; the width of the narrow and wide vectors; and the width of each packet feature tuple.

Feature tuple width depends on application requirements. For the other parameters, we implemented an OpenTuner [23] script that operates on a trace of packet arrival timestamps and a software model of the *Flow cache. The benchmarks in section 3.8 show that performance under specific parameters is stable for long periods of time.

Run-time

The *Flow cache also allows operators to configure the following parameters at run-time by installing rules into P4 match+action tables. Immediately proceeding the *Flow cache, a filtering

table lets operators install rules that determine which flows *Flow applies to, and which packet header and metadata fields go into packet feature tuples.

3.6 Analytics-aware Network Telemetry

The *Flow analytics plane interface connects the line-rate telemetry system with analytics processing servers.

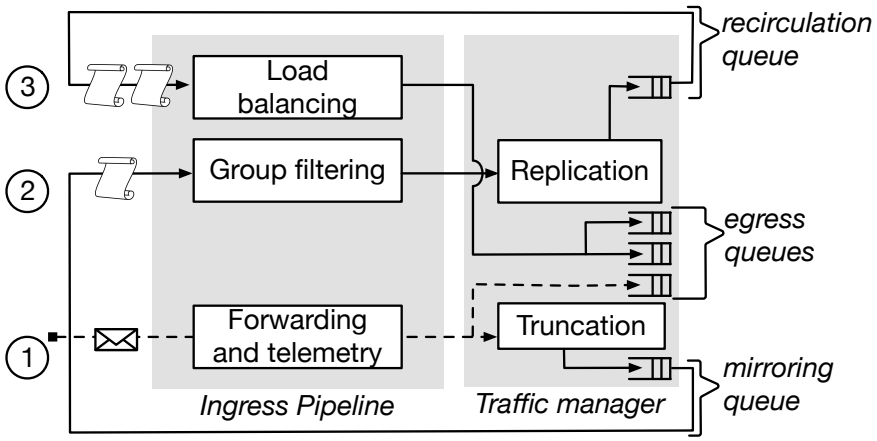


Figure 3.7: Architecture and integration of *Flows analytics plane interface to support telemetry record filtering and load balancing for multiple parallel applications

*Flows's analytics plane interface distributes and load balances GPV streams to application pipeline instances, solving the problem of getting the right telemetry streams to the right analytics servers efficiently. This, in turn, eliminates the initial choke points of software stream processing (i.e., software load balancing across processors) and allows the application pipeline instances to operate entirely in parallel.

We again leverages the hardware of programmable switches (e.g., the Tofino) to support three important functions at high line rates (e.g., $>$ 100M telemetry records per second per switch): replicating streams of GPV records to multiple concurrent applications, filtering each application's

stream to only contain relevant packet flows, and load balancing each application's stream across an arbitrary number of stream processing pipeline instances.

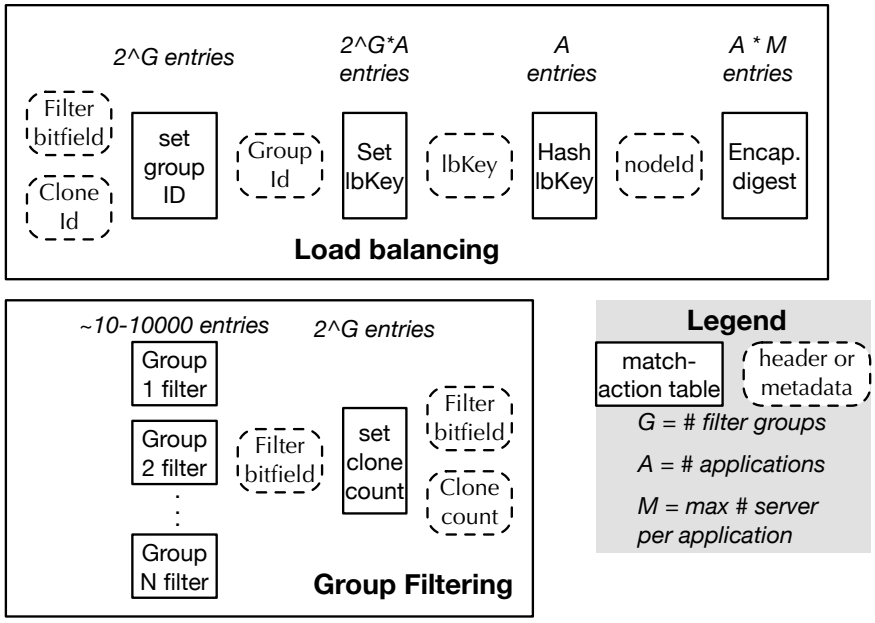


Figure 3.8: Ingress pipeline components of Fig. 3.7

Abstraction

The abstraction for the *Flow analytics plane interface is simple and application-centric. An application registers the IP addresses of its assigned Jetstream processing servers, defines the header fields to be used for load balancing flows across the servers, and specifies which packet flows it needs to monitor with a set of ternary filtering rules over packet flow key fields (e.g., IP 5-tuple).

Implementation

As illustrated by Figures 3.7, the *Flow analytics plane interface leverages three features of programmable switch ASICs: a programmable ingress pipeline, a traffic manager with configurable multicast and packet truncation, and internal packet recirculation ports. All of these features are supported by today's P4 switches, e.g., the Barefoot Tofino.

Switch processing begins with the underlying telemetry system that process data plane pack-

ets (1 in figure 3.7). *Flow exports GPVs that are encapsulated and copied to a mirroring engine in the switch's traffic manager. The mirroring engine recirculates the GPV packets for processing by Jetstream (2 in figure 3.7).

*Flow first applies tables that determine which groups of applications need to receive copies of the digest. Group filtering tables populate a bitfield in metadata, where `bitfield[a] = 1` indicates that application group `a` needs to receive a copy of the GPV. A subsequent table matches on the bitfield to calculate how many times the GPV needs to be cloned and populate metadata for the multicast engine. The multicast engine clones the digest the appropriate number of times and sends all clones to a recirculation port.

When the clones arrive back at ingress (3 in Fig. 3.7), Jetstream applies load-balancing tables that select a processing server for each digest clone, update its destination address and port, and select the appropriate physical egress port.

Configuration

An analytics framework (e.g., Jetstream) configures the distribution and load balancing for each stream processing application by adding rules to the tables in figure 3.8. First, to register a new application, it installs rules into the encapsulation table that map server IDs to the IP address and port on which the application's stream processing pipeline is running. Second, to configure per-application load balancing, rules are added to a load balancing key table, which copies a customizable subset of packet header fields to metadata that is hashed to select a server ID.

Finally, to configure distribution, *Flow assigns the application to a filtering group. All the applications in a filtering group will receive copies of GPVs that match the filtering group's table. The filtering group tables store ternary rules over flow key fields, e.g., IP 5-tuple. When adding an application to a filtering group g, it also updates all tables that match on the filter bitfield by adding rules to account for scenarios where `bitfield[g] = 1` (the group needs to process a GPV).

3.7 Processing GPVs

The *Flow analytics plane interface streams GPVs to processing servers. There, measurement and monitoring applications (potentially running concurrently) can compute a wealth of traffic statistics from the GPVs and dynamically change their analysis without impacting the network.

In this section, we describe the *Flow agent that receives GPVs from the *Flow-enabled switch.

3.7.1 The *Flow Agent

The *Flow agent reads GPV packets from queues filled by NIC drivers and pushes them to application queues. While applications can process GPVs directly, the *Flow agent implements three performance and housekeeping functions that are generally useful.

Load Balancing

As an alternative to hardware-supported load balancing (e.g., for smaller deployments), the *Flow agent supports load balancing in two directions. First, a single *Flow agent can load balance a GPV stream across multiple queues to support applications that require multiple per-core instances to support the rate of the GPV stream. Second, multiple *Flow agents can push GPVs to the same queue, to support applications that operate at higher rates than a single *Flow agent can support.

GPV Reassembly

GPVs from a *Flow cache typically group packets from short intervals, e.g., under 1 second on average, due to the limited amount of memory available for caching in PFEs. To reduce the workload of applications, the *Flow agent can re-assemble the GPVs into a lower-rate stream of records that each represent a longer interval.

Cache Flushing

The *Flow agent can also flush the *Flow cache if timely updates are a priority. The *Flow agent tracks the last eviction time of each slot based on the GPVs it receives. It scans the table periodically and, for any slot that has not been evicted within a threshold period of time, sends a

control packet back to the *Flow cache that forces an eviction.

3.7.2 *Flow Monitoring Applications

In the initial design, to demonstrate the practicality of *Flow, we implemented three monitoring applications that require concurrent measurement of traffic in multiple dimensions or packet-level visibility into flows. We ported these applications to the Jetstream framework at a later point in time. These requirements go beyond what prior PFE accelerated systems could support with compiled queries. With *Flow, however, they can operate efficiently, concurrently, and dynamically.

The GPV format for the monitoring applications was a 192 bit fixed width header followed by a variable length vector of 32 bit packet feature tuples. The fixed width header includes IP 5-tuple (104 bits), ingress port ID (8 bits), packet count (16 bits), and start timestamp (64 bits). The packet feature tuples include a 20 bit timestamp delta (e.g., arrival time - GPV start time), an 11 bit packet size, and a 1 bit flag indicating a high queue length during packet forwarding. Chapter 4 elaborates on monitoring applications in more detail and presents the preferred architecture for network streaming analytics.

Host Timing Profiler

The host timing profiler generates vectors that each contain the arrival times of all packets from a specific host within a time interval. Such timing profiles are used for protocol optimizers [180], simulators [41], and experiments [175].

Prior to *Flow, an application would build these vectors by processing per- packet records in software, performing an expensive hash table operation to determine which host transmitted each packet.

With *Flow, however, the application only performs 1 hash operation per GPV, and simply copies timestamps from the feature tuples of the GPV to the end of the respective host timing vector. The reduction in hash table operations lets the application scale more efficiently.

Traffic Classifier

The traffic classifier uses machine learning models to predict which type of application generated

a traffic flow. Many systems use flow classification, such as for QoS aware routing [67, 82], security [108, 163], or identifying applications using random port numbers or share ports. To maximize accuracy, these applications typically rely on feature vectors that contain dozens or even hundreds of different flow statistics [108]. The high cardinality is an obstacle to using PFEs for accelerating traffic classifiers, because it requires concurrent measurement in many dimensions.

*Flow is an ideal solution, since it allows an application to efficiently compute many features from the GPV stream generated by the *Flow cache. Our example classifier, based on prior work [126], measures the packet sizes of up to the first 8 packets, the means of packet sizes and inter-arrival times, and the standard deviations of packet size and inter-arrival times.

We implemented both training and classification applications, which use the same shared measurement and feature extraction code. The training application reads labeled "ground truth" GPVs from a binary file and builds a model using Dlib [95]; the classifier reads GPVs and predicts application classes using the model.

Micro-burst Diagnostics

This application detects micro-bursts [88, 147, 48], short lived congestion events in the network, and identifies the network hosts with packets in the congested queue at the point in time when the micro-burst occurred. This knowledge can help an operator or control application to diagnose the root cause of periodic micro-bursts, e.g., TCP incasts [48], and also understand which hosts are affected by them.

Micro-bursts are difficult to debug because they occur at extremely small timescales, *e.g.,*on the order of 10 microseconds [190]. At these timescales, visibility into host behavior at the granularity of individual packets is essential. Prior to *Flow, the only way for a monitoring system to have such visibility was to process a record from each packet in software [193, 80, 192, 183] and pay the overhead of frequent hash table operations.

With *Flow, however, a monitoring system can diagnose micro-bursts efficiently by processing a GPV stream, making it possible to monitor much more of the network without requiring additional servers.

The *Flow micro-burst debugger keeps a cache of GPVs from the most recent flows. When each GPV first arrives, it checks if the high queue length flag is set in any packet tuple. If so, the debugger uses the cached GPVs to build a globally ordered list of packet tuples, based on arrival timestamp. It scans the list backwards from the packet tuple with the high queue length flag to identify packet tuples that arrived immediately before it. Finally, the debugger determines the IP source addresses from the GPVs corresponding with the tuples and outputs the set of unique addresses.

3.7.3 Interactive Measurement Framework

An important motivation for network measurement, besides monitoring applications, is operator-driven performance measurement. Marple [122] is a recent system that lets PFEs accelerate this task. It presents a high level language for queries based around simple primitives (filter, map, group, and zip) and statistics computation functions. These queries, which can express a rich variety of measurement objectives, compile directly to the PFE, where they operate at high rates.

As discussed in section 3.2, compiled queries make it challenging to support concurrent or dynamic measurement. Using *Flow, a measurement framework can gain the throughput benefits of PFE acceleration without sacrificing concurrency or dynamic queries, by implementing measurement queries in software, over a stream of GPVs, instead of in hardware, over a stream of packets.

To demonstrate, we extended the RaftLib [32] C++ stream processing framework with kernels that implement each of Marple's query primitives on a GPV stream. A user can define any Marple query by connecting the primitive kernels together in a connected graph defined in a short configuration file, similar to a Click [96] configuration file, but written in C++. The configuration compiles to a compact Linux application that operates on a stream of GPVs from the *Flow agent.

We re-wrote 6 example Marple queries from the original publication [122] as RaftLib configurations, listed in table 3.5. The queries are functionally equivalent to the originals, but can all run concurrently and dynamically, without impacting each other or the network. These applica-

tions operate on GPVs with features used by the *Flow monitoring application, plus a 32 bit TCP sequence number in each packet feature tuple.

3.8 Evaluation

In this section, we evaluate our implementations of the *Flow cache and the *Flow agent. First, we analyze the PFE resource requirements and eviction rates of the *Flow cache to show that it is practical on real hardware. Next, we benchmark the *Flow agent and monitoring applications to quantify the scalability and flexibility benefits of GPVs. Finally, we compare the *Flow measurement query framework with Marple, to showcase *Flow's support for concurrent and dynamic measurement.

All benchmarks were done with 8 unsampled traces from 10 Gbit/s core Internet routers taken in 2015 [44]. Each trace contained around 1.5 billion packets.

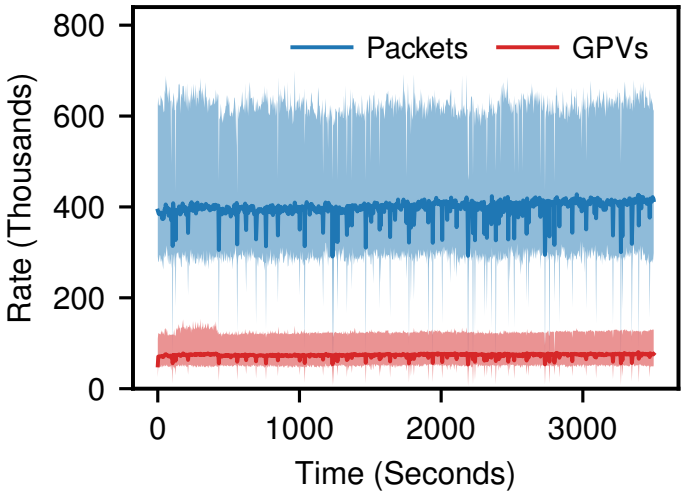
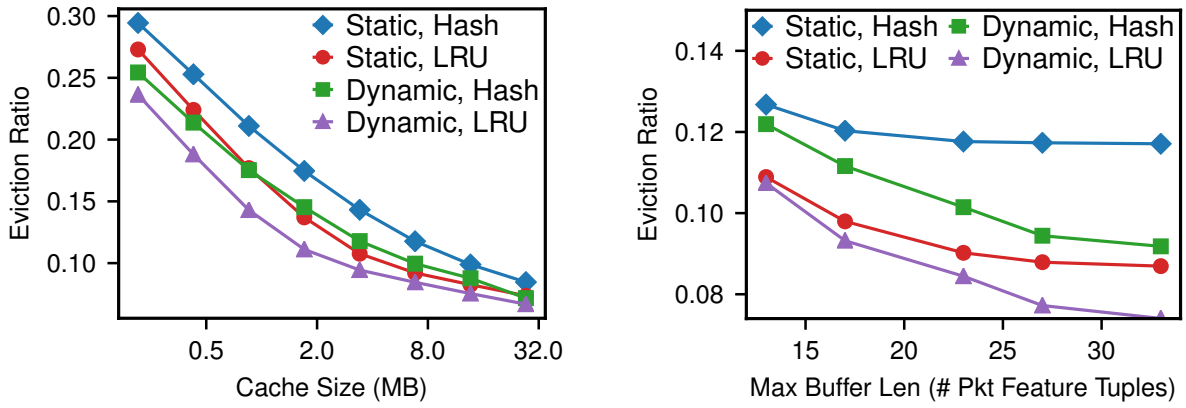


Figure 3.9: Min/avg./max of packet and GPV rates with *Flow for Tofino

	Key Update	Memory Management	Pkt. Feature Update	**Total**
Computational				
Tables	3.8%	3.2%	17.9%	**25%**
sALUs	10.4%	6.3%	58.3%	**75%**
VLIWs	1.6%	1.1%	9.3%	**13%**
Stages	8.3%	12.5%	29.1%	**50%**
Memory				
SRAM	4.3%	1.0%	10.9%	**16.3%**
TCAM	1.1%	1.1%	10.3%	**12.5%**

Table 3.3: Resource requirements for *Flow on the Tofino, configured with 16384 cache slots, 16384 16-byte short buffers, and 4096 96-byte wide buffers.



(a) PFE memory vs eviction ratio

(b) GPV buffer length vs eviction ratio

Figure 3.10: Eviction ratio for different cache configurations

3.8.1 The *Flow Cache

We analyzed the resource requirements of the *Flow cache to understand whether it is practical to deploy and how much it can reduce the workload of software.

PFE Resource Usage

We analyzed the resource requirements of the *Flow cache configured with a tuple size of 32-bits, to

support the *Flow monitoring applications, and a maximum GPV buffer length of 28, the maximum length possible while still fitting entirely into an ingress or egress pipeline of the Tofino. We used the tuning script, described in Section 3.8.1, to choose the remaining parameters using a 60 second trace from the 12/2015 dataset [45] and a limit of 1 MB of PFE memory.

Table 3.3 shows the computational and memory resource requirements for the *Flow cache on the Tofino, broken down by function. Utilization was low for most resources, besides stateful ALUs and stages. The cache used stateful ALUs heavily because it striped flow keys and packet feature vectors across the Tofino's 32 bit register arrays, and each register array requires a separate sALU. It required 12 stages because many of the stateful operations were sequential: it had to access the key and packet count before attempting a memory allocation or free; and it had to perform the memory operation before updating the feature tuple buffer.

Despite the high sALU and stage utilization, it is still practical to deploy the *Flow cache alongside other common data plane functions. Forwarding, access control, multicast, rate limiting, encapsulation, and many other common functions do not require stateful operations, and so do not need sALUs. Instead, they need tables and SRAM, for exact match+action tables; TCAM, for longest prefix matching tables; and VLIWs, for modifying packet headers. These are precisely the resources that *Flow leaves free.

Further, the stage requirements of *Flow do not impact other applications. Tables for functions that are independent of *Flow can be placed in the same stages as the *Flow cache tables. The Tofino has high instruction parallelism and applies multiple tables in parallel, as long as there are enough computational and memory resources available to implement them.

PFE Resources vs. Eviction Rate

Figure 3.9 shows the average packet and GPV rates for the Internet router traces, using the *Flow cache with the Tofino pipeline configuration described above. Shaded areas represent the range of values observed. An application operating on GPVs from the *Flow cache instead of packet headers needed to process under 18% as many records, on average, while still having access to the features of individual packets. The cache tracked GPVs for an average of 640MS and a maximum of 131

seconds. 14% of GPVs were cached for longer than 1 second and 1.3% were cached for longer than 5 seconds.

To analyze workload reduction with other configurations, we measured eviction ratio: the ratio of evicted GPVs to packets. Eviction ratio depends on the configuration of the cache: the amount of memory it has available; the maximum possible buffer length; whether it uses the dynamic memory allocator; and its eviction policy. We measured eviction ratio as these parameters varied using a software model of the *Flow cache. The software model allowed us to evaluate how *Flow performs on not only today's PFEs, but also on future architectures. We analyzed configurations that use up to 32 MB of memory, pipelines long enough to store buffers for 32 packet feature tuples, and hardware support for an 8-way LRU eviction policy. Larger memories, longer pipelines, and more advanced eviction policies are all proposed features that are practical to include in next generation PFEs [39, 49, 122].

Figure 3.10a plots eviction ratio as cache memory size varies, for four configurations of caches: with or without dynamic memory allocation; and with either a hash on collision eviction policy or an 8-way LRU. Division of memory between the narrow and wide buffers was selected by the AutoTuner script. With dynamic memory allocation, the eviction ratio was between 0.25 and 0.071. This corresponds to an event rate reduction of between 4× and 14× for software, compared to processing packet headers directly.

On average, dynamic memory allocation reduced the amount of SRAM required to achieve a target eviction ratio by a factor of 2. It provided as much benefit as an 8-way LRU, but without requiring new hardware.

Figure 3.10b shows eviction rates as the maximum buffer length varied. Longer buffers required more pipeline stages, but significantly reduced eviction ratio when dynamic memory allocation was enabled.

# Cores	Agent	Profiler	Classifier	Debugger
1	0.60M	1.51M	1.18M	0.16M
2	1.12M	3.02M	2.27M	0.29M
4	1.85M	5.12M	4.62M	0.55M
8	3.07M	8.64M	7.98M	1.06M
16	3.95M	10.06M	11.43M	1.37M

Table 3.4: Average throughput, in GPVs per second, for *Flow agent and applications.

3.8.2 *Flow Agent and Applications

We benchmarked the *Flow agent and monitoring applications, described in Section 3.7.3, to measure their throughput and quantify the flexibility of GPVs.

Experimental Setup

Our test server contained an Intel Xeon E5-2683 v4 CPU (16 cores) and 128 GB of RAM. We benchmarked maximum throughput by pre-populating buffers with GPVs generated by the *Flow cache. We configured the *Flow agent to read from these buffers and measured its throughput for reassembling the GPVs and writing them to a placeholder application queue. We then measured the throughput of each application individually, driven by a process that filled its input queue from a pre-populated buffer of reassembled GPVs. To benchmark multiple cores, we divided the GPVs across multiple buffers, one per core, that was each serviced by separate instances of the applications.

Throughput

Table 3.4 shows the average throughput of the *Flow agent and monitoring applications, in units of reassembled GPVs processed per second. For perspective, the average reassembled GPV rates for the 2015 10 Gbit/s Internet router traces, which are equal to their flow rates, are under 20K per second [44]. The high throughput makes it practical for a single server to scale to terabit rate monitoring. A server using 10 cores, for example, can scale to cover over 100 such 10 Gb/s links by dedicating 8 cores to the *Flow agent and 2 cores to the profiler or classifier.

Throughput was highest for the profiler and classifier. Both applications scaled to over 10M

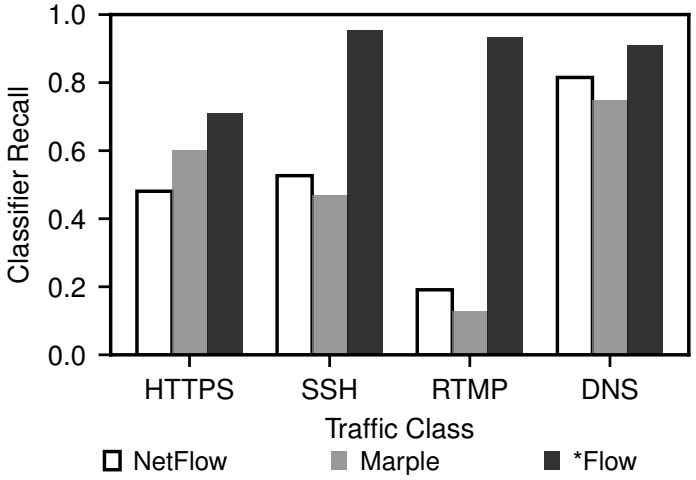


Figure 3.11: Recall of *Flow and baseline classifiers.

reassembled GPVs per second, each of which contained an average of 33 packet feature tuples. This corresponds to a processing rate of over 300 M packet tuples per second, around 750× the average packet rate of an individual 10 Gb/s Internet router link.

Throughput for the *Flow agent and debugging application was lower, bottlenecked by associative operations. The bottleneck in the *Flow agent was the C++ `std::unordered_map` that it used to map each GPV to a reassembled GPV. The reassembly was expensive, but allowed the profiler and classifier to operate without similar bottlenecks, contributing to their high throughput.

In the debugger, the bottleneck was the C++ `std::map` it used to globally order packet tuples. In our benchmarks, we intentionally stressed the debugger by setting the high queue length flag in every packet feature tuple, forcing it to apply the global ordering function frequently. In practice, throughput would be much higher because high queue lengths only occur when there are problems in the network.

Classifier Accuracy

To quantify the flexibility benefits of GPVs, we compared the *Flow traffic classifier to traffic classifiers that only use features that prior, less flexible, telemetry systems can measure. The NetFlow classifier uses metrics available from a traditional NetFlow switch: duration, byte count, and packet count. The Marple classifier also includes the average and maximum packet sizes

Configuration	# Stages	# Atoms	Max Width
***Flow cache**	**11**	**33**	**5**
Marple Queries			
Concurrent Connections	4	10	3
EWMA Latencies	6	11	4
Flowlet Size Histogram	11	31	6
Packet Counts per Source	5	7	2
TCP Non-Monotonic	5	6	2
TCP Out of Sequence	7	14	4

Table 3.5: Banzai pipeline usage for the *Flow cache and compiled Marple queries.

as features, representing a query that compiles to use approximately the same amount of PFE resources as the *Flow cache.

Figure 3.11 shows the recall of the traffic classifiers on the 12/2015 Internet router trace. The *Flow classifier performed best because it had access to additional features from the GPVs. This demonstrates the inherent benefit of *Flow, and flexible GPV records, for monitoring applications that rely on machine learning and data mining.

3.8.3 Comparison with Marple

Finally, to showcase *Flow's support for concurrent and dynamic measurement, we compare the resource requirements for operator driven measurements using compiled Marple queries against the requirements using *Flow.

PFE Resources

For comparison, we implemented the *Flow cache for the same platform that Marple queries compile to: Banzai [152], a configurable machine model of PFE ASICs. In Banzai, the computational resources of a PFE are abstracted as atoms, similar to sALUs, that are spread across a configurable number of stages. The pipeline has a fixed width, which defines the number of atoms in each stage.

Table 3.5 summarizes the resource usage for the Banzai implementation. The requirements for *Flow were similar to those of a single statically compiled Marple query. Implementing all 6

queries, which represent only a small fraction of the possible queries, would require 79 atoms, over 2X more than the *Flow cache. A GPV stream contains the information necessary to support all the queries concurrently, and software can dynamically change them as needed without interrupting the network.

Server Resources

The throughput of the *Flow analytics framework was between 40 to 45K GPVs/s per core. This corresponded to a per-core monitoring capacity of 15 - 50 Gb/s, depending on trace. Analysis suggested that the bottleneck in our current prototype is message passing overheads in the underlying stream processing library that can be significantly optimized [116].

Even without optimization, the server resource requirements of the *Flow analytics framework are similar to Marple, which required around one 8 core server per 640 Gb/s switch [122] to support measurement of flows that were evicted from the PFE early.

3.8.4 Analytics Plane Interface

The *Flow analytics plane interface is designed for low resource requirements and high throughput in the current generation of P4 programmable switches. On the Barefoot Tofino, it requires under 10% of the ingress pipeline's memory and compute resources (e.g., SRAM, TCAM, SALUs, and VLIW units) when compiled to support a maximum of 8 filtering groups, 32 applications, and up to 128 servers per application.

The ingress pipeline components of the interface are compiler guaranteed to run at line rate, but throughput can be limited by recirculation bandwidth. For example, the Tofino has 400 Gb/s of internal recirculation port bandwidth. *Flow utilizes the limited bandwidth efficiently because it only recirculates telemetry digests which are, by design, orders of magnitude smaller than the packet or flows that they summarize. Assuming 80 byte GPVs, the Tofino's 400 Gb/s is sufficient for exporting 625M GPVs per second.

3.9 Conclusion

Measurement is important for both network monitoring applications and operators alike, especially in large and high speed networks. Programmable forwarding engines (PFEs) can enable flexible telemetry systems that scale to the demands of such environments. Prior systems have focused on leveraging PFEs to scale efficiently with respect to throughput, but have not addressed the equally important requirement of scaling to support many concurrent applications with dynamic measurement needs. As a solution, we introduced *Flow, a PFE-accelerated telemetry system that supports dynamic measurement from many concurrent applications without sacrificing efficiency or flexibility. The core idea is to intelligently partition the query processing between a PFE and software. In support of this, we introduced GPVs, or grouped packet vectors, a flexible format for network telemetry data that is efficient for processing in software. *Flow can dynamically distribute GPVs across monitoring applications through a load balancing component. This enables highly-parallel and optimized GPV processing in software on commodity servers. We designed and implemented a *Flow cache that generates GPVs and operates at line rate on the Barefoot Tofino, a commodity 3.2 Tb/s P4 forwarding engine. To make the most of limited PFE memory, the *Flow cache features the first implementation of a dynamic memory allocator in a line rate P4 program. Evaluation showed that *Flow was practical in the switch hardware and enabled powerful GPV based applications that scaled efficiently to terabit rates with the capability for flexible, dynamic, and concurrent measurement.

Chapter 4

Scalable Network Streaming Analytics

Traditionally, network monitoring and analytics systems rely on aggregation (e.g., flow records) or sampling to cope with high packet rates. This has the downside that, in doing so, we lose data granularity and accuracy, and, in general, limit the possible network analytics we can perform. Recent proposals leveraging software-defined networking or programmable hardware provide more fine-grained, per-packet monitoring but are still based on the fundamental principle of data reduction in the network, before analytics.

In this chapter we present a cloud-scale, packet-level monitoring and analytics system based on stream processing entirely in software. Software provides virtually unlimited programmability and makes modern (e.g., machine-learning) network analytics applications possible. We identify unique features of network analytics applications and workloads. Based on these insights we built a customized network analytics solution. Our implementation shows that we can scale up to over 70M packets per second per 16-core stream processing server.

4.1 Introduction

Network management is a critical task in the overall operation of a network and impacts the availability, performance, and security. Effective network management relies on the ability of operators to perform analytics on network traffic. Analytics is the act of finding meaningful patterns in data which can trigger actions. In the case of networks, the data consists of all packets flowing through a network, and the actions include traffic isolation, device reconfiguration, or alerts to

the operator. Historically, we largely relied on humans in a network operation center to watch some transformed version of the data (often in the form of graphs) and interpret that data to take action. As networks grow in size, complexity, and cost, automated and sophisticated analytics are becoming more commonplace. This enables the network to become an essential part of detecting security issues [162], misconfiguration [80, 103], equipment failure [154], as well as performing traffic engineering [21, 36].

Network analytics has largely revolved around network devices (switches) summarizing flow information (e.g., through NetFlow records[57]) and passing records to custom software for analysis (whether by computers or humans). Due to the limits of the information captured, there are only a number of things we can determine from this data and therefore a limited number of things we can do. The introduction of programmable switches, such as those that support OpenFlow [114] or P4 [38] programming paradigms, has opened up new opportunities to revisit network analytics and explore ways to increase the rates at which records can be generated and provide more choice in what information is analyzed.

This new opportunity has sparked a wave of work in this space recently. OpenSketch [185] demonstrated the ability to realize sketching algorithms in programmable switches, thus allowing the efficient monitoring for some application (such as heavy hitter detection). UnivMon [107] extended that work to provide the ability to perform universal sketches in the programmable switches, allowing multiple applications (with different information needs) to be able to simultaneously analyze the traffic. Marple [122], went to the extreme and demonstrated the ability to compile entire queries into a programmable forwarding engine.

Common across this work is that they each focused on ways to make software have to do less work. But, in leveraging the hardware to not just collect the data, but perform some data reduction or analysis, they sacrifice the flexibility that software brings.

In this work, we focus on the complementary approach of enabling software to do more, and in doing so, enable network analytics that can support:

- multiple analytics applications operating simultaneously,
- network wide and datacenter scale analysis,
- applications which require high-fidelity information, *and*
- efficient compute resource usage.

The issue isn't that streaming analytics is inherently incapable of these goals, but to achieve them, we need better strategies for leveraging and extending these systems.

First, existing system are designed for a general purpose, and in turn, incapable of being able to handle high rates of network traffic. This motivated the need for Sonata [79], which used Spark [187], to introduce new filtering mechanisms. We outline domain-specific optimizations which overcome these performance limitations for the specific task of network analytics.

Second, existing network analytics designs systems focus on one aspect of the system. This becomes evident when supporting multiple applications and multiple instances, using existing stream processing systems as an independent part of the network analytics system (as is done in the recent work on programmable switches), we end up with choke points such as for directing streams to one or more instances and for aggregating results across instances. To overcome this, we maintain independent pipelines by recognizing that we are not building a stream processing system but a network analytics system, so we push the responsibility of directing streams to one or more cores (across multiple servers) into the telemetry system, and push the aggregation into the operator interactive system, thus enabling the stream processing to be highly parallel.

We present Jetstream, a network analytics system that leverages optimized pipeline-based stream processing, and a Prometheus [11] and Grafana [7] based query backend. We implemented five applications, including a heavy-hitter detection which, by design has 100% accuracy, traffic accounting, DDoS detection, a TCP out-of-order segment detector, and a complex auto-encoder based intrusion detection system modeled after Kitsune [118]. Jetstream scales linearly with core count across machines and can process over 200 million packet records per second leveraging only

3 commodity servers. The optimizations we introduce allow our stream processing component to outperform Spark by roughly 3x when streaming from memory (this was to isolate the benefits of the optimizations from the benefits of Jetstream including kernel bypass and Spark not).

4.2 Motivation

In this section, we motivate Jetstream by exploring the roles and limitations of data plane and software components in current monitoring systems.

4.2.1 Compromising on Flexibility

Recent work has focused on taking advantage of the programmability of switches to offload analytics from software stream processors to the network data plane, under the assumption that the software is not suitable for high-traffic environments. While the approach can improve performance, it also has significant drawbacks related to flexibility.

Flexibility describes how adaptable a network analytics platform is to evolving user requirements. For this chapter we split up flexibility slightly differently than before (compare section 2.3): For the context of network analytics, there are several axes: *application flexibility*, gauged by the classes of applications and metrics the system can support efficiently; *programming flexibility*, or the capability to implement functionality with high level programming languages and standard algorithms; and *scalability*, the ability to increase network coverage or the number of concurrent monitoring applications without decreasing overall system efficiency.

All prior work that offloads analytics to the data plane sacrifices multiple axes of flexibility due to inherent limitations of line-rate hardware. One line of work, sketching, offloads the computation of a sub-class of streaming statistics to data plane hardware using memory-efficient probabilistic data structures. Due to their memory-efficiency, sketches can scale to high network coverage [185] and many concurrent applications [107] using the limited amounts of line rate memory in data plane hardware. However, sketches sacrifice application and programming flexibility – they only support a certain class of streaming statistics (an already specific subclass) and require system developers

to implement their metric calculation functions as sketching algorithms, which can be challenging to design.

Another line of work [122] compiles more general flow statistic calculation functions to programs that are split between servers and data plane hardware. Compared to sketches, these compiled packet queries offer improved programming flexibility because they allow users to implement statistical calculation with traditional algorithms and higher level languages. However, they sacrifice scalability. Without the probabilistic data structures of sketches, memory requirements in the data plane are significantly higher. As flow rates or the number of applications grow, the proportion of work done by the backing servers increases, drastically reducing overall system performance. Further, although compiled queries support a broader class of metrics than sketches, they are still limited by the computational capabilities of the underlying switch hardware, which can only implement simple stateful functions [152] and have limited support for mathematical calculation [148].

A different, bolder approach was taken with Sonata [79], which introduces the idea of using the data plane for filtering in addition to aggregation. The idea is to forward packet measurements of relevant flows to a scalable stream processing cluster that calculates metrics implemented in high level languages. Still, again based on the assumption that software cannot scale to high traffic rates, the system relies on heavy filtering and partial analytics offload to hardware which limits possible applications.

4.2.2 Software Network Analytics Strawman

Why are stream processing systems unable to process at high enough rates, causing systems such as Sonata [79], which leveraged Spark as its processing engine, to need filtering to keep up? To help with this question, lets start with a strawman ideal system (as illustrated in Figure 4.1) – where the network switches capture some record about every packet, pass that to a cluster running software to process in real-time, and then stores the information in a database for administrators and monitoring systems. With this, there are two key observations that highlight opportunities to challenge this basic assumption that software isn't fast enough.

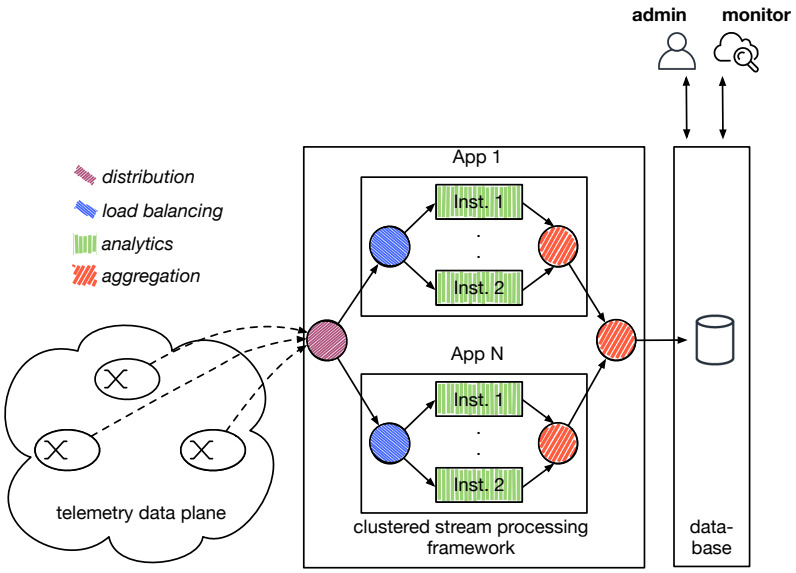


Figure 4.1: Strawman network analytics system.

General purpose means less efficient

Stream processors run on general-purpose hardware with general compute capabilities, so all the application metrics can be implemented. But, the systems are themselves designed for general use and aim to provide acceptable performance in diverse workloads, e.g., business, science, finance, and for a wide range of compute graphs and physical topologies. The generality comes at the cost of extra overhead that applies to every record the system processes [115]. This overhead greatly reduces performance with network analytics workloads, which are characterized by high rates of small records.

As a simple example, consider packet processing systems such as netmap [141] and DPDK [84] which regularly report the ability to process in excess of 10M packets per second per core. In contrast, a simple analytics program implemented in Spark [187], which receives packet headers as input and counts the total number of packets, has a total throughput of around 1M network packet records per second per core, or $> 10\times$ lower. But it's not just kernel bypass I/O, as we detail in Section 4.4.2. There are a number of optimizations, which collectively, can greatly improve the performance of stream processing, just by narrowing the scope to network analytics.

Choke points and bottlenecks

One reason to use software is scalability. That is, we want to run multiple analytics applications simultaneously, and dynamically change them over time. To support this in a stream processor used for network analytics, the software needs to distribute the stream of telemetry data to the applications. A naive solution would broadcast to all applications, and a slightly better one would send just the subset of records a given application needs to that application. This is one choke point – simply analyzing a high rate stream to decide where each record should go requires processing that, in a general-purpose stream processor, is expensive.

Once within the application, a second choke point is the software component that needs to load balance the traffic to one of potentially many worker instances that can process the stream. Finally, a typical stream processing network analytics application would aggregate results across the instances to output the metric(s) of interest. This requires each worker to send data to a single aggregator – a third choke point.

4.3 Introducing Jetstream

Jetstream is a high-performance network analytics system that makes no compromises on flexibility. It lets applications perform packet analytics, including the calculation of arbitrary metrics, entirely in software and scales linearly with server resources. To overcome the issues observed in section 4.2, we co-designed the telemetry, analytics processing, and analysis interface to keep the stream processing units entirely parallel (as pipeline units). As figure 4.2 illustrates, the design moves functionality out of the stream processor to eliminate choke points: we move distribution and load balancing into the network switches (for analytics-aware network telemetry – section 3.6), and push aggregation to the database and analysis system (for user analysis with on-demand aggregation). We are then left with highly parallel stream processing, to which we then apply domain-specific optimizations for high-performance network analytics.

Using Jetstream

Jetstream is designed to support two common network analytics tasks. First, stream processing

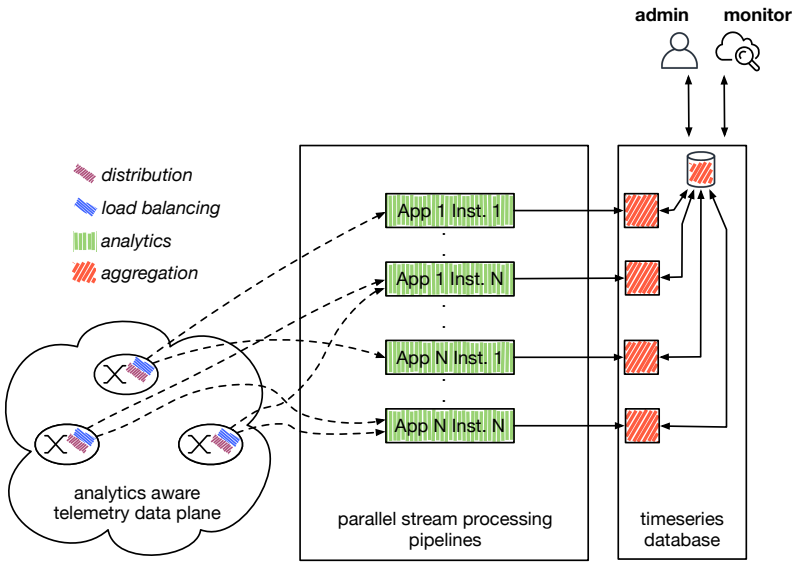


Figure 4.2: Jetstream network analytics system

applications that operate on records of every packet. A stream processing application could be a header-based intrusion detector [118], a queue depth monitor that alerts network controllers of congestion and its root cause [157], or a metric calculator that generates records for higher level network monitoring applications. These applications are built by composing stream processing kernels that are either drawn from Jetstream's standard library, or implemented as custom C++ classes. Second, Jetstream also supports analytics queries that poll a timeseries database storing flow metric time series, e.g., the average rate of every TCP flow at 1 second intervals. The queries can be written by the operator, e.g., for interactive debugging [122], or by other systems, e.g., a network controller [36, 73].

4.3.1 Analytics-aware Network Telemetry

The telemetry plane is the component of a monitoring system that operates at line rate and is integrated into the packet processing pipelines of reconfigurable switch hardware, e.g., P4 engines [38, 124] in order to efficiently export packet records to the analytics tier. It cannot implement an entire monitoring application because of the restrictions that switch hardware enforces to support line-rate operation [152]. It can also perform lightweight preprocessing to reduce the

analytics workload: removing packet payloads [80], filtering out unmonitored flows [79], calculating simple aggregate statistics [122], or sampling [185].

Jetstream leverages this technology for telemetry and offloading distribution and load balancing of telemetry data streams by tightly integrating with the *Flow analytics interface described in section 3.6.

4.3.2 Highly-parallel Streaming Analytics

The streaming analytics component analyzes traffic at the packet level, touching on every single exported packet in software. It is the core component of the overall Toccoa system, supporting custom applications implemented as stream processing programs.

The benefit of stream processing, as a paradigm, is compartmentalization of state and logic. Processing functions only share data via streams. This allows each processing element to run in parallel, on a separate processing core or even a separate server. In this model, each processor (or *kernel*) usually transforms data (or *tuples*) from one or multiple input streams into one or multiple output streams by performing some sort of stateful or stateless computation on the tuples and including the results in the output tuple. A program is a processing graph (or *pipeline*) that is organized in several *stages*. Each stage performs one computational task that transforms the stream of tuples (e.g., map, filter, or reduce). In traditional stream processing, applications scale at the stage level by adding or removing kernels. Each kernel typically runs in a separate thread and maps to a physical processor core. This model requires load balancing in software and introduces choke points reducing overall performance (see Section 4.2.2).

To avoid such choke points and achieve overall high throughput, we scale at the granularity of full pipelines. Multiple such pipelines can run in parallel and map to a subset of the machine's cores. *Flow's analytics plane interface component assigns packet records to these pipelines by setting the UDP destination port number. On the network interface cards of the analytics machines, packets are placed into individual hardware queues based on the destination port number. Individual Jetstream pipelines then read from their assigned queue. In order to scale to entire large-scale

networks, multiple of such pipelines can run in parallel by making use of the partitionability of packet records. We elaborate on the streaming analytics component in more detail in Section 4.4.

4.3.3 User Analysis and Monitoring with On-Demand Aggregation

The results of the stream processing pipeline, which will generally consist of high level information at lower rates, can be fed into security systems as alerts [142] or stored in a time series database for visualization, auditing, and offline analysis [172]. We designed a monitoring system that allows users to define, query, and analyze their own application-specific flow metrics, calculated by the stream processing pipeline. The monitoring system consists of an application for metric collection, a proxy that facilitates the transfer of data between the stream processor and a database, and a database that allows users to query collected metrics at any level of flow aggregation, i.e., over any subset of IP 5-tuple fields.

Collecting metrics for all flows in the database can require aggregation across parallel instances of the metric calculation pipelines. To maintain pipeline independent processing, we push cross-pipeline data aggregation into the database itself, which is already optimized to aggregate data from many sources. Each metric calculation pipeline streams data directly into its own database proxy, which exposes per-instance flow metrics through an interface that the database scrapes.

The database handles data aggregation across application instances by matching and accumulating metrics with the same name and type. Users can then query the database to extract their desired network metrics. We dive into each phase of the user analysis and monitoring system in Section 4.5

4.4 High-Performance Stream Processing for Network Records

The *Flow analytics plane interface sends telemetry records directly to the load balanced stream processing pipelines of one or more Jetstream applications. This allows the pipelines to avoid interaction (e.g., the first two choke points in section 4.2) and enables us to focus entirely on optimizations for the workload. In this section, we explore some of the distinct characteristics of

packet analytics workloads and how we can optimize stream processors for them.

4.4.1 Packet Analytics Workloads

We identify six key differences between packet analytics workloads and typical stream processing tasks.

High Record Rates

One of the most striking differences between packet analytics workloads and typical stream processing workloads are higher record rates. For example, Twitter reports [173, 174] that their stream processing cluster handles up to 46 M events per second. For comparison, the aggregate rate of packets leaving their cache network is over 320 M per second, and this only represents approximately 3% of their total network.

Small Records

Although record rates are higher for packet analytics, the sizes of individual records are smaller, which makes the overall bit-rate of the processing manageable. Network analytics applications are predominately interested in statistics derived from packet headers and processing metadata, which are only a small portion of each packet. A 40 B packet record, for example, can contain the headers required for most packet analytics tasks. In contrast, records in typical stream processing workloads are much larger.

Event Rate Reduction

Packet analytics applications often aggregate data significantly before applying heavyweight data mining or visualization algorithms, e.g., by TCP connection. This is not true for general stream processing workloads, where the back-end algorithm may operate on features derived from each record.

Simple, Well Formed Records

Network packet records are also simple and well formed. Each packet record is the same size and contains the same fields. Within the fields, the values are also of fixed size and have simple types, e.g., counters or flags. Records are much more complex for general stream processing systems

because they represent complex objects, e.g., web pages, and are encoded in serialization formats such as JSON and protocol buffers.

Network Attached Input

Data for packet analytics comes from one source: the network. Be it a router, switch, or middlebox that exports them, they will ultimately arrive to the software via a network interface. In general stream processing workloads, the input source can be anything: a database, a sensor, or another stream processor.

Partionability

There are common ways to partition packet records, e.g., for load balancing, that are relevant to many different applications. Further, since the fields of a packet are well defined, the partitioning is straightforward to implement. In general stream processing workloads, partitioning is application specific and can require parsing fields from complex objects.

4.4.2 Optimization Opportunities

Based on these observations of the packet analytic workloads, we identified four important components of stream processing systems where there is significant potential for optimization. We measure the benefit of these optimizations in section 4.7.1.

Data Input

In general-purpose stream processing systems, data can be read from many sources such as a HTTP API, a message queue system (such as RabbitMQ [12] or Kafka [25]), or from specialized file systems like HDFS [9]. These frameworks can add overhead at many levels, including due to context switches and copy operations. Since packet analytics tasks all have the same source, the network, a stream processing system designed for packet analytics can use kernel bypass and related technologies, such as DPDK [5, 84], PF_RING [128], or netmap [141], to reduce overhead by mapping the packet records directly to buffers in the stream processing system.

Zero-Copy Message Passing

Through our initial experiments we have identified that for most applications the performance of

a single processor within the stream processing graph is I/O-bound. Specifically, frequent read, write, and copy operations into the queues connecting kernels introduce significant performance penalties. Since packet records are small and well formed, a stream processing framework for packet analytics can eliminate this overhead by pre-allocating buffers and simply passing pointers between processors, to significantly improve performance.

Concurrent Queues

Elements in a stream processing pipeline communicate using queues, which can themselves have significant impact on overall application performance. We identified thread-safety and memory layout primitives as primary bottlenecks in queue implementations. Jetstream's design, in which stream distribution and load balancing is offloaded to the data plane, means that most queues connect a single producer and consumer. This allows us to implement an efficient, lock-less queue with a simple memory layout that maximizes performance.

Batching

Batching can improve performance in multiple ways. Batching access to queues amortizes the cost of individual queue and dequeue operations. Batching packet records by flow, as done by *Flow, amortizes the cost of hash table operations necessary to map each packet record to a flow. We designed our queues such that they support enqueueing and dequeuing operations in batches with only a single atomic write per batch. Doing this reduces the number of atomic operations by a factor of the batch size. We found the optimal batch size to be 64.

Hash Tables

Often, network analytics software needs to map packet records (or batches of packet records) to prior flow state. This requires a hash table, which itself can be a bottleneck [191, 66]. Since packet records are well formed and have fixed width values, there are many optimizations that we can apply to the data structures used internally by the stream processor, for example, flat layout in memory to avoid pointer-chasing.

4.5 User Analysis and Monitoring with On-Demand Aggregation

The Jetstream analysis and monitoring system provides a way for users to extract and analyze user-defined flow metrics from the network for both on and offline analysis. This component consists of three parts that we describe below: an interface for stream processing pipelines to export application-specific flow metrics, a local metric collection proxy for each pipeline that exposes a poll-based interface, and a back-end database that stores the metrics as time series data that can be queried for statistics at custom levels flow of aggregation.

Exporting Flow Metrics

Flow metrics are calculated by Jetstream stream processing applications, which operate over packet records. Jetstream's `metrics_broker` API allows application pipelines to stream arbitrary flow metrics to a database proxy. The API converts metric data into the Protocol Buffers serialization format [75], streams it to the proxy, and using a remote procedure call over GRPC [8], tells the proxy to update the metrics in the database.

The Jetstream `metrics_broker` API accepts a metric name, metric value, and flow key. It packs the input into a protobuf stream structure that contains fields for the input parameters and an auto-generated timestamp. In order to prevent a Jetstream application from streaming metric data on every packet record, metrics are streamed to the proxy at a user-defined rate from the application.

Database Proxy

The database proxy sits between a Jetstream application and the database, converting data into the appropriate format. In order to prevent data aggregation in-line resulting in cross-core communication, a database proxy is instantiated for each instance of an application and subscribes to an instance's metric stream. Once subscribed, the proxy receives a stream of metric data from the application and executes the received GRPC update commands. In our prototype, we convert metric data into a format for the Prometheus time series database, which supports the following metric types.

The `counter` metric represents a cumulative and monotonically increasing value. For example, in a Jetstream application, a `counter` is ideal for counting the total number of bytes or packets in a connection. A `gauge` can be set to a specific value, reset to zero, increased, or decreased in value. For example, a `gauge` would be ideal for maintaining the active number of connections to a specific IP address. The `histogram` metric is used to store samples into buckets of user-defined size and maintains a per bucket count of samples. Similar to `histogram`, the `summary` metric maintains `histogram`-like buckets but over a sliding window.

Database

The Prometheus database polls the proxies' APIs for the current metrics. Prometheus takes the counters, which contain a metric name, metric value, and metadata, from the API and stores the data into its time series database. Prometheus supplies a query language and API, which allows a user to extract network traffic metrics from the database. In order to provide an accurate look into the network at all times, we utilize Grafana [7], a tool that constantly queries the Prometheus database and provides an up-to-date view of user-defined metrics.

Example Applications and Queries

For each application described in 4.6.4, we provide example Prometheus queries to illustrate how a user can interact with and extract user-defined metrics from their Jetstream applications.

For the traffic accounting application, Prometheus maintains individual counter metrics for each component of a GPV's 5-tuple. For example, `tp_src_bytes` represents the application's exported metric name for counting bytes per source port. A user can use Prometheus' `rate()` function to calculate the average number of bytes per second sourcing from port 443 over the last minute using the following query:

```
rate(tp_src_bytes{tp_src="443"}[1m])
```

Note that since load balancing across instances of the same application is done by hashing the GPV's 5-tuple, the same IP addresses, for example, will end up across multiple instances of the

application, resulting in multiple total packet counters exposed by the HTTP APIs for the same IP address. When Prometheus queries each proxy's API, the counters with the same metric name and metadata are aggregated in the database automatically. As a result, the database maintains the correct total GPVs, packets, and bytes per 5-tuple component.

The heavy hitter application, which looks for IP addresses sending or receiving $> \theta\%$ of the total packets in the network, exports heavy hitter candidates with the metric name `ip_heavy_hitters`. In order to identify the top k heavy hitters from the candidates stored in the database, we can issue a query as follows:

```
topk(k, ip_heavy_hitters)
```

This query takes a vector of the heavy hitters candidates and `topk()` returns the k largest values (and their IP addresses).

The TCP analysis application looks for out of order packets in a TCP flow. Flows with at least one out of order packet are exported to the database with the metric name `tcp_seq` and the metric value counting the number of out of order packets in the flow. If a user wants to find which flows originating from the 192.168.0.0/16 subnet and port 443 have more than 10 out of order packets, the user can issue the following query to the database:

```
tcp_seq{ip_src=~"192.168.+.+",tp_src="443"} > 10
```

The query for Slowloris detection is as simple as `slowloris_candidate`, since all of the processing can be done in Jetstream without cross-core communication and the application just exports which IP addresses are Slowloris candidates. The metric name is `slowloris_candidate` and the value of the metric is the connections per byte for the IP address.

For each of the above example queries, we plugged them into Grafana in order to constantly query the database, giving us real-time network statistics.

4.6 Programmability and Applications

Processing packet records in software as opposed to implementing basic computation in the telemetry plane allows for virtually unlimited programmability and flexibility. Analytics applications can be written in a general purpose language and for general-purpose hardware. Therefore Jetstream applications can easily be prototyped, tested, and deployed. Jetstream is implemented in C++ and compiles to a dynamic library that applications can easily link against. This library does not only include the stream processing core and runtime environment, but also a variety of pre-built processors that can be used to rapidly build network monitoring and analytics applications. Additionally, application developers can define custom processors.

4.6.1 Input/Output and Record Format

As Jetstream's telemetry component extends a prior telemetry system, *Flow, we leverage *Flow's record model, grouped packet vectors.

Jetstream connects with *Flow's analytics plane interface which exports grouped packet vectors. Unlike traditional flow records, GPVs still contain individual packet data (such as individual timestamps, byte counts or TCP flags) through feature vectors. We leverage GPVs that include individual microsecond timestamps, byte counts, hardware queuing delays, queue id's, queue depths, IP ids, and TCP sequence numbers.

The primary packet input mechanism in our system leverages netmap [141], a kernel-bypass mechanism allowing the mapping of NIC buffers directly into the stream processor's (user space) memory. Using this, we are able to inject packet records at high rates into the Jetstream analytics system without allowing costly and frequent system calls to become a bottleneck in the processing pipeline. While kernel-bypass NIC access is the primary packet interface in our system, we also implemented the ability to read GPVs from memory, from files, from standard sockets, or to receive raw packet records using PCAP [18] or the TaZmen sniffer protocol [181].

4.6.2 Programming Model

The stream processing paradigm provides a perfect starting point for a library of reusable code for network analytics. Code modules can simply be implemented as custom processors that define a number of input and output ports of different types. Interconnecting such pre-defined elements then assembles a full application. A simple application printing packet records received over a network interface can be defined like this:

```
int main() {
  jetstream::app app;
  auto receive = app.add_stage<jetstream::nm_gpv_receiver>("enp24s0f0");
  auto print   = app.add_stage<jetstream::printer<starflow::gpv_t>>(std::cout);
  app.connect<starflow::gpv_t>(receive, print);
  return app();
}
```

`jetstream::app` here describes a pipeline or application which can be executed by calling the function call operator on the application object. Using this API, each application defines the processing steps (stages) it requires.

Jetstream includes a standard library of common subtasks that can be chained to build a full network analytics application. Processors in this library include `filter`, `map`, `reduce`, `count`, `join`, and `print`.

4.6.3 Custom Processors

If an analytics application requires processing logic, data types, or interfaces that are not covered by Jetstream's library, application developers can implement their custom processing elements that can easily be integrated into applications and then automatically take advantage of Jetstream's scaling and load balancing capabilities. A custom processor is simply a subclass of `jetstream::proc`. In the constructor of the subclass, input and output ports can be defined before implementing the processor logic in the `operator()()` method. Every processor has a set of input and output ports

that are typed and have a unique id. The ports can then be used to receive or send elements, respectively. A full custom proc can then be implemented like this:

```
class print : public jetstream::proc {
public:
  print() { add_in_port<starflow::gpv_t>(0); }

  bool operator()() {
    starflow::gpv_t gpv; jetstream::signal sig;
    in_port<gpv_t>(0)->dequeue_wait(gpv, sig);
    std::cout << gpv << std::endl;
    return sig == sig::proceed;
  }
};
```

In addition to directly subclassing `jetstream::proc`, predefined processors for common input/output scenarios exist that can further simplify this code.

4.6.4 Standard Applications

Here we show the flexibility of Jetstream, and one's ability to query the backend database for arbitrary network statistics. We implemented four standard network monitoring applications, which range from basic traffic accounting to DoS detection.

Traffic Accounting

For traffic accounting, we count the total number of GPVs, packets, and bytes for each component of a GPV's 5-tuple (e.g. IP protocol, source IP, source port, destination IP, and destination port).

Heavy-Hitter Detection

The heavy hitter detection application looks for IP addresses sending or receiving $> \theta\%$ of the total packets in the network. We implemented the KPS heavy hitter detection algorithm described in [92] and stream heavy hitter candidates to the database proxy.

TCP Analysis

Here we monitor TCP flows that transmit out of order packets. Out of order packets can be a result of a number of network issues such as packet drops, loops, reordering, and duplication [87]. Flows

with more than one out-of-order packet are sent to the monitoring database for further analysis.

DoS Detection

Our DoS detection application looks for Slowloris attacks [58], a resource exhaustion attack that consumes a webserver's connection pool. We identify such attacks with two subqueries. One subquery counts the number of unique connections. The other subquery maintains the total number of bytes flowing toward a destination IP. These two queries are joined to calculate the average connections per byte. Hosts with more connections per byte than a predefined threshold are reported to the database. For this application, we load balance across the destination IP address instead of the 5-tuple to maintain accurate destination IP byte counts prior to joining the two queries. *Flow's analytics plane interface allows specifying the header fields over which a hash is computed on a per-application basis.

4.6.4.1 Advanced Applications

For our last application, we took Mirsky et al.'s autoencoder-based network intrusion detection system, Kitsune [118], ported it from Python to C++, and implemented it on top of Jetstream. We ran Kitsune on Jetstream to demonstrate Jetstream's ability to achieve high throughput for highly specialized and complex applications. While Kitsune is not part of our formal evaluation, and was implemented as a proof of concept, we were able to achieve a 76.9 thousand packets per core throughput, which is approximately twice as fast than the author's own implementation.

4.7 Evaluation

We thoroughly evaluate the performance of our prototype implementation using three different sets of benchmarks. First, we evaluate the effects of a subset of the different optimizations that we performed (Section 4.7.1). Second, we evaluate the overall system scalability and performance in a realistic network environment for different example applications (Section 4.7.2). Third, we evaluate the resource utilization of our PFE implementation.

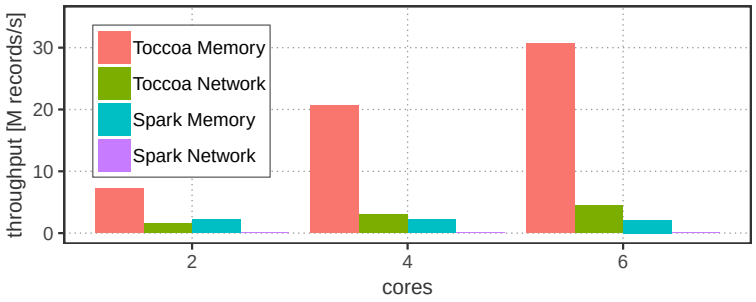


Figure 4.3: Comparing Jetstream with Spark

Jetstream can scale beyond 50 million packet records per second per pipeline when using GPVs and beyond 30 million packet records per second without GPVs. It outperforms similar applications implemented in Spark Streaming by 3.3X when reading from memory and by 17.7X when reading from the network for two cores. Jetstream scales linearly with core count where Spark does not provide scalability for this type of workload. Figure 4.3 shows the details of this experiment.

Experiment Setup

We used the Cloudlab network experimentation platform [59] for all of our benchmarks. Our experiment topology consisted of six servers interconnected over two separate 10GbE networks. The two nodes were equipped with two 10-core Intel Xeon E5-2660 v3 CPUs clocked at 2.6 Ghz. Each node had 160GB of ECC DDR4 memory. The nodes were connected to a central switch over two 10Gbit/s networks with Intel X520 converged Ethernet adapters. Each NIC has 40 independent receive and transmit queues per port that fit 2048 packets each. For all experiments we used packet traces from a 10Gbit Internet core link collected by CAIDA in February 2015 [43].

4.7.1 Stream Processing Optimizations

In Section 4.4.2, we outlined six software analytics optimizations for the unique characteristics of packet analytics workloads. We now evaluate a subset of these optimizations with a basic Jetstream application that counts the number of bytes per source IP address. While the implementation for

this application is not complex, it still examines every single packet record. We show the performance of computationally more demanding applications in Section 4.7.2. For each optimization, we demonstrate their effectiveness by comparing the throughput of our application with all optimizations enabled with the same application without the respective optimization.

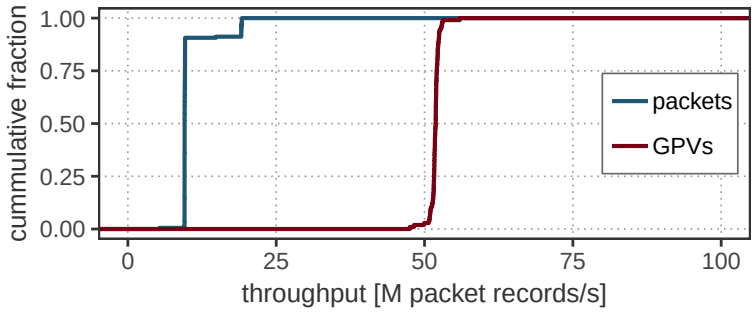


Figure 4.4: Jetstream throughput using GPVs vs. individual packet records

Record Format

Jetstream operates on grouped packet vectors (GPVs), batches of packet measurements grouped by IP 5-tuple in the data plane forwarding engine. Figure 4.4 depicts the distribution of measured application throughput using GPVs compared to using packet records. GPVs provide an average of 5.4× speedup over individual packet records by amortizing the overhead of hash table and queue operations. The mean GPV length in our traces was 8.2 — speedup was not equal to GPV length because a GPV is also larger than an individual packet record, representing a small overhead.

Name	Memory Layout	Thread Safety
queue1	Linked List	Locks
queue2	Array	Locks
queue3	Linked List	Memory Barriers
queue4	Array	Memory Barriers
queue5	Array (2^n slots)	Memory Barriers

Table 4.1: Properties of the different evaluated concurrent queue implementations.

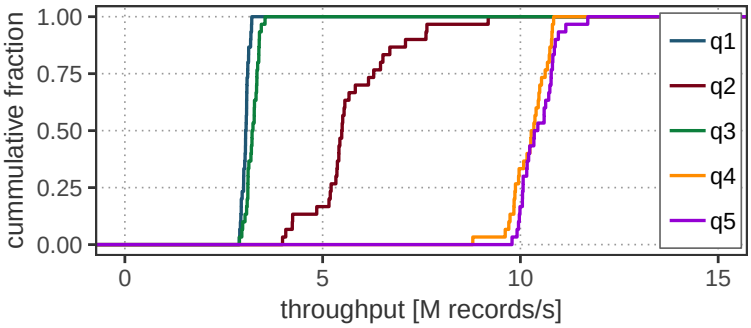


Figure 4.5: Throughput for different concurrent queue implementations for 32 Byte records

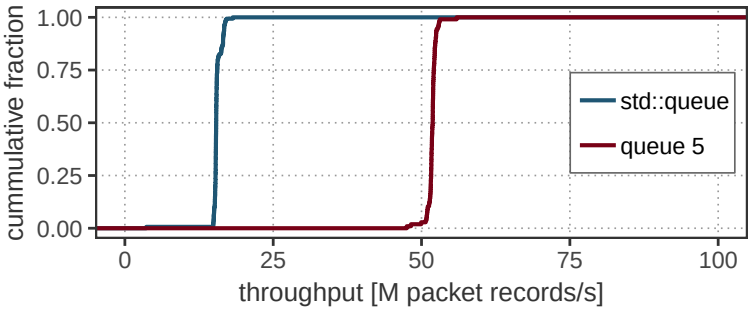


Figure 4.6: Jetstream throughput using our optimized queue implementation vs. the C++ STL standard queue

Concurrent Queues

Jetstream uses a highly optimized queue to minimize the overhead of communication between elements in the stream processor. Figure 4.6 compares its overall performance to that of the C++ STL

standard queue (made thread-safe through basic mutexes). Our optimized queue implementation improved application throughput by over a factor of 3X.

To drill down into the specific optimizations, we benchmarked different queue implementations (see Table 4.1) using a simple micro-benchmark connecting two kernels passing 32 B records. The full set of results is shown in figure 4.5. We started out with two simple lock-based queue implementations (queue 1 and queue 2) and improved this design to a lock-free linked list implementation (queue 3). The lock-free implementations achieved thread safety using C++11 memory ordering primitives and atomic variables.

Finally, we extended the lock-free queue to use a ring buffer instead of a linked list (queue 4 and queue 5) and limited buffer sizes to powers of two (queue 5), allowing us to use bit logic instead of more expensive modulo operations to handle slot index rollover.

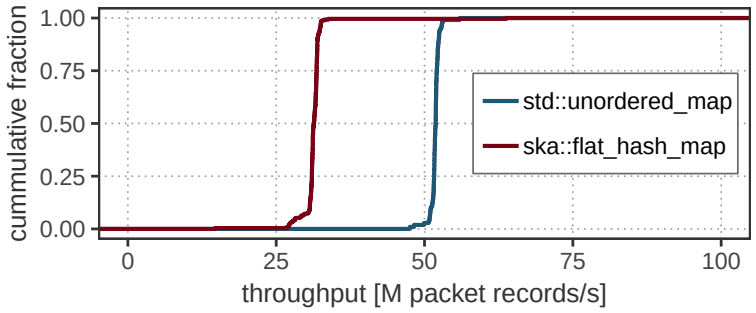


Figure 4.7: Jetstream throughput when using a flat hash table implementation vs. the C++ STL hash table

Hash Tables

Packet analytics often relies on hash table data structures, e.g., for mapping packet records to per-IP state. Jetstream uses a flat hash table with linear probing [153], where keys and values are stored directly into the data structure's slot array. This provides a 1.8X speedup compared to the `std::unordered_map`, an efficient container-based chaining implementation (Figure 4.7). The

speedup is due to eliminating pointer operations and improving cache locality [155].

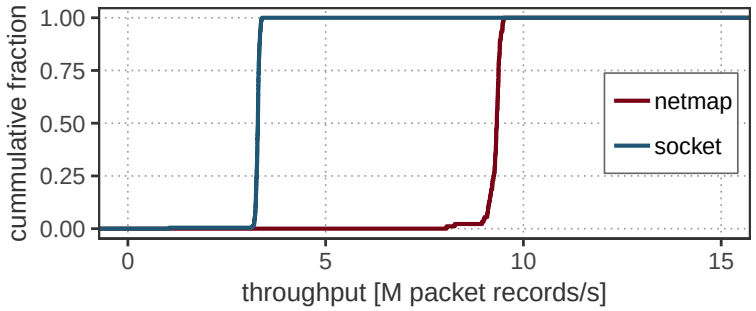


Figure 4.8: Jetstream throughput: netmap vs. Linux sockets

Network Input

Jetstream uses netmap [141] to map NIC buffers directly into the analytics pipeline's memory space and bypass the OS kernel. Figure 4.8 shows a single application pipeline's throughput with netmap compared to an identical implementation that uses a basic Linux datagram socket instead. Netmap provided a speedup of 2.8X by avoiding the overhead of transferring packets through the OS stack.

4.7.2 Scalability

Building on the previous benchmarks, we now benchmark Jetstream's performance and scalability at a macro level using the applications described in 4.6.4. In this experiment, we modeled a scenario where 3 switches stream GPVs across the network to 3 Jetstream analytics servers running application pipelines. We model the switches by running a software implementation of *Flow on 3 separate servers in the Cloudlab network, driven by real-world packet traces from CAIDA. In each run of the experiment, we add an additional application pipeline instance to the Jetstream cluster. Each pipeline uses two cores scaling to a total of 8 pipelines per server, or 24 pipelines using 48 cores across 3 servers.

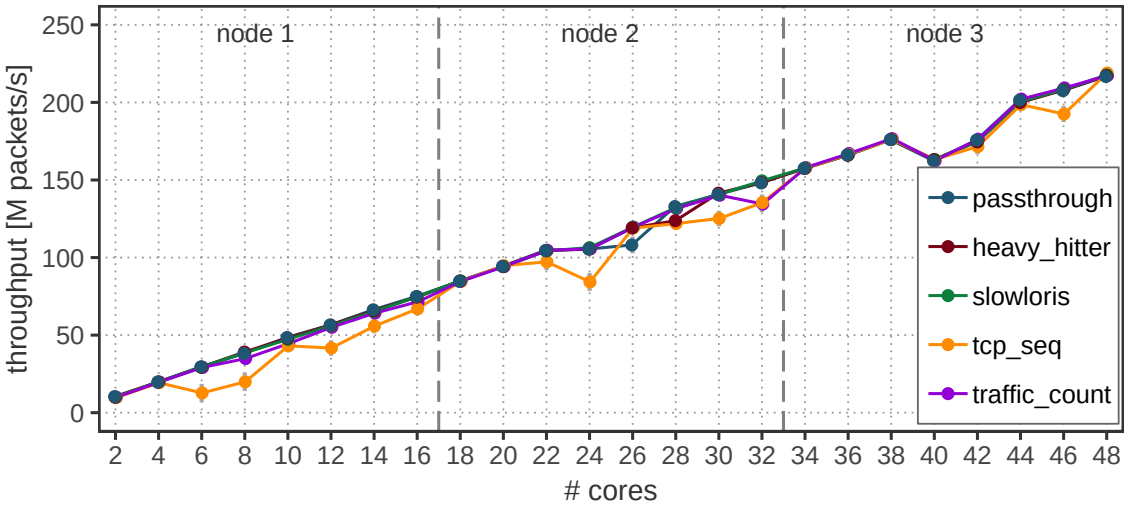


Figure 4.9: Scalability of different Jetstream applications across servers

Figure 4.9 shows the effectiveness of Jetstream's design of independent processing pipelines and offloaded load balancing and aggregation. Jetstream scales linearly with core count across machines and can process over 200 million packet records per second leveraging only 3 commodity servers. Section 4.8 discusses what this throughput means in terms of deployment cost using real-world data center traffic statistics. The bottleneck in all benchmarks was the interface between the netmap driver and Jetstream, suggesting that future optimizations there could further increase performance.

.

4.8 Deployment Analysis

We analyze the cost of network-wide telemetry with Jetstream based on the 24 hour traces from Facebook production clusters [144, 69]. Table 2.1 summarizes the clusters. We model a deployment scenario where packet-level telemetry and load balancing is done by ToR switch closest to the destination host. Each ToR forwards packet headers to a Jetstream processing server selected by load balancing based on hash of packet IP 5-tuple, which runs the applications described in prior sections to calculate flow metrics.

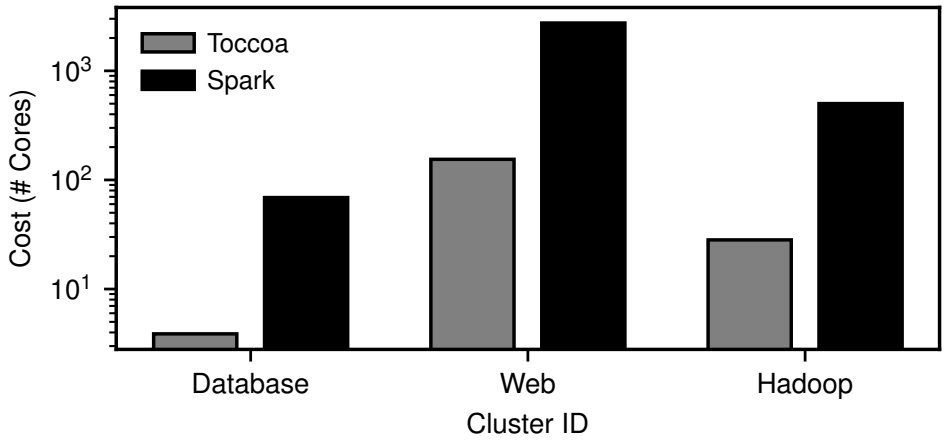


Figure 4.10: Analytics deployment cost with Jetstream and Spark.

Figure 4.10 shows the cost of an individual application, in terms of processing cores required during peak load in each cluster. We derived peak load by measuring the packet rate of each top-of-rack switch (ToR) in each cluster and then scaling it by a factor of 30,000, the sampling rate of the data set. Jetstream required 187 cores for analytics across all three clusters, or approximately 12 16-core servers. Spark, for comparison, would require over 3312 cores for the same workload – a roughly 200 node cluster just for analytics.

The other cost of monitoring is network utilization for carrying the telemetry data. In this trace, the ToR with the highest peak throughput would generate less than 1.6 Gb/s of telemetry data at all times, assuming 64B of data per packet (GPVs would reduce this). For every other ToR, the peak telemetry data generation rate would be under 0.41 Gb/s.

4.9 Related Work

Jetstream is a fast and flexible network analytics system. It builds on prior work on network monitoring, stream processing, and high performance packet processing.

Network Monitoring

Jetstream is a platform for push based network monitoring, in which the network data plane streams records describing traffic to analytics processes. Records can describe individual packets [80, 167, 94]

or aggregates of multiple packets in the same flow [155, 79, 122, 103].

Packet level records typically contain fields from the packet's header and additional metadata about network performance, e.g., queue depths [80, 167, 94]. These records are simple for switches to generate, as there is no stateful processing or metric calculation required. Packet records also contain more information than aggregate records, enabling applications that pinpoint the root causes of network issues [80] accurately detect network threats [118]. Jetstream compliments these packet-level monitoring systems and applications; it is the first general platform for processing packet level records efficiently once they leave the data plane. As the analysis in Section 4.8 showed, efficiency is crucial when deploying packet-level applications in real high speed networks.

Aggregate records, e.g., IPFIX [57] or NetFlow [83] records, contain summarize multiple packets to reduce the analytics workload. Many recent systems have focused on how to efficiently generate aggregate records [155, 79, 122, 103] with programmable switch hardware [152, 124]. Jetstream's efficiency makes it practical to move the flow record generation into software. This has two main benefits. First, it increases flexibility, as some applications [126] require flow metrics that are too complex for switch hardware to implement [148]. Second, it frees up switch memory, which is only around 10-32 MB [122], for more critical data plane services, e.g., forwarding.

Stream Processing

The distinguishing feature of Jetstream, compared to general purpose stream processors [187], is its optimization for network analytics workloads. Jetstream mirrors other recent work that optimizes stream processing for specific applications. AWStream [189] focuses on wide-area stream processing, Drizzle [176] balances latency, throughput, and fault tolerance for real-time applications, StreamBox [115] optimizes for single-node stream processing, and RaftLib [33] automates load balancing for compute-intensive applications.

High Performance Packet Processing

Jetstream is closely related to work on optimizing software packet processors. It leverages Netmap [141] to improve performance by bypassing the kernel stack. Like Click [96], Jetstream minimizes the overhead between invocations of functions in a software pipeline. However, it achieves this with

zero-copy queues, similar to NetBricks [131] rather than virtual function calls. Jetstream supports a more general programming model than these systems, in which the inputs and outputs of each function can be a stream of arbitrary C++ objects, rather than packet objects. Similar to RouteBricks [66], Jetstream exploits multi-core, multi-queue parallelism. However, Jetstream is designed for more complex and general processing, where a single record may be processed by an arbitrary number of cores.

4.10 Conclusion

This chapter introduces Jetstream, a flexible and high performance traffic analytics platform. The key is applying a holistic treatment to the problem of traffic analytics. We pinpoint choke points in software stream processors and design Jetstream's architecture to eliminate them and enable fully parallel analytics pipelines. We then identify domain specific optimizations for the core stream processing engine to alleviate remaining bottlenecks.

The resulting prototype of Jetstream scales to $>$ 70M packets per second per 16-core stream processing server, an improvement of over 3x compared with Spark. Benchmarks show that Jetstream's integration with the *Flow data distribution and load balancing system, enables linear scaling with addition of servers while only requiring moderate switch resources. Jetstream scales linearly with core count across machines and can process over 200 million packet records per second leveraging only 3 commodity servers. Using a large-scale trace based analysis, we demonstrate that Jetstream can analyze measurements from every packet in Facebook clusters representing over 20,000 servers using only 12 analytics servers.

Jetstream represents an ideal combination of performance and flexibility that makes packet and flow analytic practical at large scales in real networks. It also paves the way for new applications that perform more advanced analytics without being constrained by the underlying data plane hardware.

Chapter 5

Persistent Interactive Queries for Network Security Analytics

Network monitoring is an increasingly important task in the operation of today's large and complex computer networks. In recent years, technologies leveraging software defined networking and programmable hardware have been proposed. These innovations enable operators to get fine-grained insight into every single packet traversing their network at high rates. They generate packet or flow records of all or a subset of traffic in the network and send them to an analytics system that runs specific applications to detect performance or security issues at line rate in a *live* manner.

Unexplored, however, remains the area of detailed, interactive, and retrospective analysis of network records for debugging or auditing purposes. This is likely due to technical challenges in storing and querying large amounts of network monitoring data efficiently. In this work, we study these challenges in more detail. In particular, we explore recent advances in time series databases and find that these systems not only scale to millions of records per second but also allow for expressive queries significantly simplifying practical network debugging and data analysis in the context of computer network monitoring.

5.1 Introduction

Network analytics systems focus on the practical analysis of network records for performance monitoring, intrusion detection, and failure detection [116, 79]. They use advances in parallel software-based data processing, such as stream processing as well as kernel bypass technologies for data input [66, 131, 141, 5].

Together, telemetry and analytics systems provide fine-grained visibility into live network conditions that is useful for many applications. But equally important and un-addressed by current systems is visibility into past network conditions. There are a variety of reasons why such information matters. For some applications, such as network auditing, historic information is simply required. For others, such as debugging, it is essential to not only identify that the network is in a certain state, but also how it got into that state. In many cases, historic data is necessary because analysis is too expensive to do in real time, and many times need human interaction to investigate. For example, in security systems, a common scenario is to identify anomalies in real time and tag related network monitoring records (packet or flow records) for offline analysis using a heavier weight analysis system or assistance from a network administrator.

All of the above applications rely on retrospective queries about the network, which requires some level of record persistence. This poses a significant challenge given the volume and velocity of record-based monitoring data in networks. The challenge is not only due to the shear amount of records that modern telemetry systems can generate, but also due to the high rates (hundreds of millions of packets per second) at which today's wide-area and data center networks operate.

In this chapter, we analyze this challenge in more detail and take first steps towards a persistence system that supports not only live queries, but also retrospective queries. We explore the requirements of such a system, identify time-series databases as a promising starting point, and design strategies for using modern database engines with state of the art network telemetry and analytics systems. Finally, we sketch the design of a next generation network monitoring architecture, centered around programmable hardware that is composed of telemetry, analytics, and persistence planes, that supports high performance expressive and retrospective queries.

5.2 Background

Databases have been successfully deployed and used for decades in a wide range of applications and are the backbone of systems across all industries. Traditionally, databases have been used for online transaction processing workloads, like in the financial, production and transportation industries.

Advances over the past 10-15 years in data transmission rates and storage capacity have driven the demand for a revolution in database and data analytics technologies. These workloads that are significantly higher in both velocity and volume are commonly referred to as Big Data workloads. In this section, we look at this spectrum of database technology to understand suitability for network monitoring.

5.2.1 Database Models

Although initially database systems were designed around the rigid mathematical relational model, the emergence of Big Data has led to new database designs straying from this original model. These databases are often referred to as NoSQL databases as their relational counterparts use the Structured Query Language (SQL) as their interface.

Relational database management systems (RDBMS) require data to be in a fixed format that often needs to be broken up in several relations in order to fit in this model. Especially for modern workloads, this process comes with significant performance drawbacks due to frequent joins. On the other hand, relational databases are extremely powerful: They allow for complex data models, can enforce rigid integrity constraints, follow a strong transactional model and have a very expressive query interface through SQL.

NoSQL databases generally do not require this fixed storage format and are, as a result, easier to adapt to custom, irregular and unstructured data. Additionally, they are optimized for large volumes of data and often have better I/O performance and horizontal scalability properties than their relational counterparts. This is mostly due to the lack of features that SQL-based systems implement and enforce in the database layer, such as integrity constraints.

5.2.2 Time-Series Databases

Alongside the emergence of Big Data applications, the widespread deployment of IoT and general sensor data applications has triggered a shift from traditional transactional database applications to applications where data has a strong temporal character. A common example for such data is

measurement-related data where an observation is associated with a time. A key characteristic of such workloads is that this data is typically only written once, never updated and from that point on exclusively read (queried). As a result, database systems optimized for this type of workload and equipped with time-series related functions were proposed. These systems are referred to as time-series databases.

As network monitoring data normally consists of measurements of some sort that are associated with their time of observation, time-series databases are a natural fit for our problem domain. We experimented with several systems and soon realized that an expressive query interface, as well as join operations across data stored at different levels of granularity (e.g., packet records vs. flow records) are essential requirements for designing a practical and flexible persistence scheme for network monitoring data.

Unfortunately, the vast majority of time-series optimized databases are implemented as some sort of non-relational key-value store. While this is suitable for multiple independent series of measurements that are never put in context with each other, this is not suitable for network monitoring data (see section 5.5). TimescaleDB [172] provides a promising alternative: A relational database management system that is optimized for time-series data. Timescale is an extension for the PostgreSQL RDBMS, a database system that has been used for decades and is an industry standard because of its many features, reliability and scalability [137]. TimescaleDB provides high write throughput, good scalability and most importantly does not compromise on any traditional database features by providing a full-featured SQL query interface and allowing for constraints and joins.

5.2.3 Database Requirements for Network Record Persistence

TimescaleDB appears to be a good match for the application of network monitoring. In the remainder of this chapter we seek to evaluate the suitability of TimescaleDB for this application. In particular, we answer three key questions:

Is the query interface expressiveness enough? A network administrator must be able to

quickly query the database system using a flexible, fast, and intuitive query system that meets the needs of interactive analysis of historical network telemetry data. In Section 5.3 we provide examples of such queries.

At what rates can we insert data? Inserting data into the database is a key challenge that limits the applicability of the database. Network monitoring records are commonly generated at rates of several million per second. As a result, a database system must be optimized for high write rates, and we need to understand the degree of aggregation which is required to meet the insert limits. In Section 5.4, we evaluate optimizations for write performance into TimescaleDB.

At what scale can we store data? At these high insert rates, massive amounts of data can accumulate in short periods of time. The database system must be scalable enough to cope with data volumes of billions of records. This, coupled with the aggregation levels, then determines the window of time which network operators can interactively query. In section 5.5, we analyze the storage requirements for network records at different granularities.

5.3 Querying Network Records

Before even discussing performance, we look at the expressiveness of TimescaleDB in the context of network monitoring. We wish to run retrospective queries and allow for exploratory data analysis on network records.

5.3.1 Network Queries

To demonstrate the flexibility of SQL for network monitoring records, we show a set of example queries highlighting different language features of SQL and TimescaleDB:

1. *Bin packet and byte counts in 1s time intervals:* Timescale's `time_bucket()` function is useful to make large amounts of time-series data manageable. For example, this query can be used to generate a traffic graph over time.

```
SELECT time_bucket('1 seconds', ts_us) AS interval,
  SUM(pkt_count) AS pkt_count, SUM(byte_count) AS byte_count
```

```
FROM gpv GROUP BY interval ORDER BY interval ASC
```

2. *Count the number of packets per IP address from a particular IP subnet within the last hour:* PostgreSQL's `INET` datatypes allows specifying queries that are not limited to a direct match on an IP address but can efficiently query at the granularity of IP prefixes.

```
SELECT ip_src, SUM(pkt_count) AS pkt_count FROM gpv WHERE ip_src
  << inet '60.70.0.0/16' AND gpv.ts_us > NOW() - interval '1 hours'
  GROUP BY ip_src
```

3. *List packets that came from a particular IP subnet:* A `JOIN` allows to query data across relations and in this case can result in per-packet records including individual timestamps, packet sizes, or TCP information. If the underlying packet records were already deleted, this query would still succeed but only show aggregate information such as total byte and packet counts.

```
SELECT * FROM (gpv RIGHT JOIN pkt ON gpv.gpv_id = pkt.gpv_id)
  WHERE gpv.ip_src << inet '60.70.0.0/16'
```

5.3.2 Retrospective Queries and Debugging

In section 5.1 we explained how network analytics suites are optimized to process large amounts of data quickly. This is important for the timely detection of intrusions or other issues within the network. In the case of an anomaly, an analytics system can generate alerts but does not have the ability to inspect the problem.

For example, a network record stream processor could detect an unusually high queue occupancy and queuing delay in a single queue of a switch; a problem often caused by incorrect load balancing schemes or an adversarial traffic pattern. We now show how retrospective network queries can in this scenario help a network administrator to determine if indeed a misconfiguration is causing this effect or whether the network is simply at capacity.

As a first step it is of value to determine what packets actually were in the queue while the alert was raised:

```
SELECT DISTINCT gpv.ip_src, gpv.ip_dst, gpv.ip_proto, gpv.tp_src,
  gpv.tp_dst FROM (gpv RIGHT JOIN pkt ON gpv.gpv_id = pkt.gpv_id)
  WHERE pkt.ingress_ts >= '2018-02-19 12:59:11.595' AND
  pkt.ingress_ts < '2018-02-19 12:59:11.600' AND pkt.queue_id = '5'
```

This query returns a list of flows whose packets were in the respective queue at this specified time. The administrator sees that the majority of packets were destined for particular IP subnet, a cluster that serves video content. The routing configuration for this subnet shows, that it is reachable via an ECMP group. After determining the links that are part of this group, the following query can be used to get insight into the distribution of packets across these links:

```
SELECT pkt.queue_id, COUNT(pkt.queue_id) FROM (gpv RIGHT JOIN pkt
  ON gpv.gpv_id = pkt.gpv_id) WHERE gpv.ip_dst << inet '53.231/16'
  AND pkt.queue_id IN (4,5,6,7) GROUP BY pkt.queue_id;
```

If the returned distribution is roughly uniform, the load balancing scheme works, otherwise there is an issue with this particular ECMP group and its hashing algorithm.

While the stream processor could include a digest of the packets which were in the particular queue at the time, the problem may be located well beyond this single queue and finding the root cause can require a more in-depth analysis of the state of the network. Record persistence and an interactive query system allow an operator to analyze the problem in more detail across different network devices with the goal of identifying the underlying issue.

Furthermore, as all of these queries, took less than one second to complete on a 200M packet record dataset, this method is perfectly suitable for interactive debugging and exploratory data analysis. While query performance might degrade with larger databases, there are ways to overcome this issue through precompiling queries [6] or using specialized indices or views for frequently performed queries.

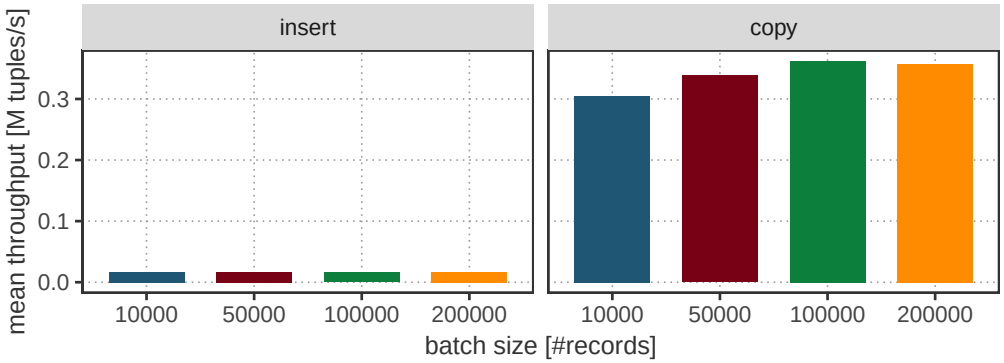


Figure 5.1: GPV INSERT vs. COPY performance

5.4 Inserting Network Records

A key challenge for the design and implementation of network record based monitoring systems lies in dealing with high traffic rates of today's networks. Database systems, as previously explained, are usually not optimized for these write-heavy workloads and represent a performance bottleneck. Therefore, the database write performance determines how many network records can be saved in a given amount of time and at which level of aggregation they can be stored for network analysis. We evaluate TimescaleDB in this regard and show optimizations that vastly improve insert performance.

The injection process (commonly implemented through SQL `INSERT` statements) is associated with complex underlying logic and tasks, such as updating indices, checking constraints, partitioning data, and running triggers. These tasks can significantly hurt injection performance. TimescaleDB helps in this case as it is optimized for mainly appending data as opposed to performing random insert and update operations. TimescaleDB organizes data in chunks indexed by timestamp that fit into memory. Each chunk has its own index. This means that the individual indices are small and efficiently manageable. Chunks get written to disk asynchronously only after a chunk has been filled entirely.

The insert performance can further be optimized by not using complex data constraints and injecting data in chunks using the SQL `COPY` statement as opposed to using `INSERT`. PostgreSQL

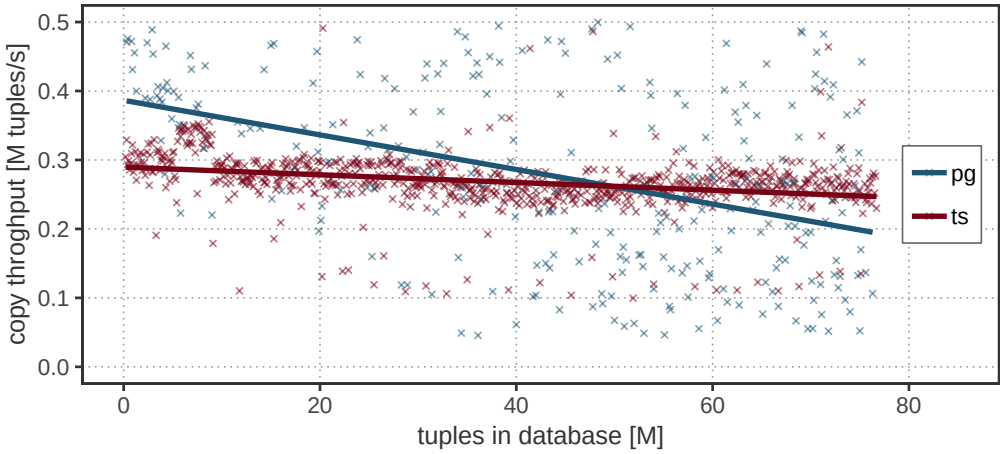


Figure 5.2: GPV COPY performance as a function of database size for PostgreSQL and TimescaleDB

(Timescale's underlying database) has a custom binary format and allows for fast inserts of chunks of data in this format. Details on this feature and the format can be found in [138]. We compared the performance of `COPY` and `INSERT` for different chunk sizes. Figure 5.1 shows the mean throughput of injecting 10 million rows. At a batch size of 100K packets using `COPY`, we achieve an average write throughput of 360K records per second. This is over an order of magnitude higher compared to using `INSERT`.

We also compared the write-performance of TimescaleDB and standard PostgreSQL with respect to the number of tuples already inserted. Figure 5.2 shows the results of this experiment. We can see that PostgreSQL's write performance is initially higher but degrades with database size. TimescaleDB's performance only slightly decreases with database size.

While a write throughput of 300K - 400K records per second is still over an order of magnitude lower than packet rates in high-speed networks, this rate can actually be sufficient for most applications since not necessarily every packet needs to be stored at the highest level of granularity. Using flow-based aggregation schemes, compression rates of 30-40×, while maintaining a good level of detail per flow, are possible. We further elaborate on this in the next section. Additionally, we believe that these write rates can be further improved by optimizing PostgreSQL storage and transaction settings, as well as inserting records using multiple threads simultaneously.

Field	Length [Byte]	Description
ts_us	8	absolute timestamp in μs
gpv_id	8	unique identifier
flow_key	26	IP 5-tuple
- ip_src	8	IP source address
- ip_dst	8	IP destination address
- tp_src	4	source port
- tp_dst	4	destination port
- ip_proto	2	IP Protocol
sample_id	2	sample identifier
tap_id	2	tap identifier (*e.g.*, switch)
duration	4	GPV duration in μs
pkt_count	2	number of packets in GPV
byte_count	3	number of Bytes in GPV

Table 5.1: Grouped Packet Vector Format

5.5 Storing Network Records

As our goal is to enable interactive retrospective queries, the storage of the data base is critical in determining the time window under which we can query. In order to maintain the ability to analyze stored network records using expressive and powerful queries, a carefully designed storage format is imperative. Furthermore, given the volume of records that can be generated, compression and data retention strategies must be addressed. In this section we detail the data format and analyze storage tradeoffs an operator can take for a particular use case.

5.5.1 Grouped Packet Vectors

We use the grouped packet vector format (GPV) described in section 3.4 as the record format for our prototype implementation. A grouped packet vector contains an IP 5-tuple flow key and a variable length vector of feature tuples from sequential packets in the respective flow.

In order to query GPV data in a relational database system in a flexible manner, the GPV format must be normalized through two relations, one for the GPV header (flow data) and one for the packet features. The columns for the two relations, including their storage size requirements in TimescaleDB, are listed in Table 5.1 and in Table 5.2.

Field	Length [Byte]	Description
gpv_id	8	unique identifier (foreign key)
ts_us	8	absolute timestamp in μs
queue_id	2	unique queue ID
tcp_flags	2	TCP Flags
egress_delta	4	ingress - egress timestamp
byte_count	2	total packet length
queue_depth	2	experienced queue length
ip_id	2	IP identification field
tcp_seq	4	TCP sequence number

Table 5.2: Packet Record Format

As opposed to proper flow record generation using TCP flags and timeouts, we generate GPVs using a simple hash table-based cache data structure as we use this format mainly for reasons of data compression. The cache is organized in a number of slots (cache height) with a fixed amount of packet features that fit in each slot (cache width). The slot index is determined through a hash function from the IP 5-tuple. Individual packets are then appended to the packet feature vector. A GPV is evicted from the cache when either a hash collision happens or when the feature vector is full. This data structure can be implemented in hardware and further optimized using secondary caches for high-activity flows.

For this work, however, we use a simple single cache implementation in software. The cache performance (in terms of eviction ratio) is directly dependent on the cache dimensions (number of slots and slot width). Figure 5.3 and figure 5.3 depicts the effects on the eviction behavior for different cache dimensions through experimentation using real-world WAN packet traces [43]. The larger the cache is in either dimension, the higher the achieved compression ratio (GPV length) is (see figure 5.3). On the other hand, a large cache requires more memory and also extends the eviction latency, *i.e.,* the time a record spends in the cache before being evicted (see figure 5.4). This can be important when running applications on live data. As a result, the cache dimensions must be carefully chosen for the application's requirements.

For the remainder of this paper, we chose a cache height of $2^{18} = 262144$ with a width of 32. With these parameters, the size of the cache in memory is 206 MB. In our simulations, the

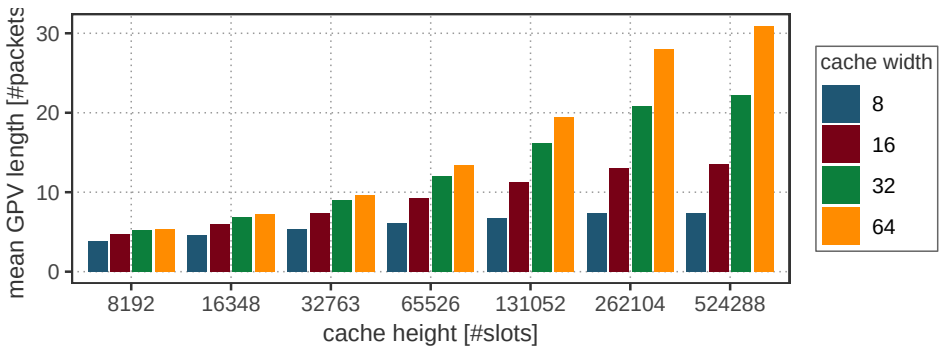


Figure 5.3: mean GPV length for different cache configurations

mean GPV length is 20.82 with a mean eviction latency of 0.85 seconds and a 99%ile latency of 2.60 seconds.

5.5.2 Storage and Record Retention

Storing data in a database generally requires more storage space than a custom, optimized binary format. In this case, a GPV header in C++ only occupies 48 Bytes, whereas its representation in TimescaleDB occupies 56 Bytes. A packet record requires 34 Bytes in Timescale, but only 24 Bytes in our custom storage format. The difference mainly stems from use of different data types, as well as added foreign key columns. Additionally, TimescaleDB maintains indices and other metadata for tables. We measured the physical disk storage required to store 100 million records of each type. Together with all metadata and indices, a GPV header occupies 122.7 Bytes and a packet record 97.8 Bytes. In either case. this is a more than 2× increase over the binary format. The storage requirements grow linearly with record count.

Looking at these numbers at scale, we can see that using our format and database layout, 1 billion packet records require approximately 1TB of disk storage. In addition, each packet record belongs to a GPV header which occupies storage. The storage requirements of the GPV header in respect to the total number of records depends on the maximum configured GPV length. At an maximum GPV length of 16, a billion packet records require approximately 76GB of GPV records. At a maximum length of 64, the requirement goes down to roughly 19GB. The total database size

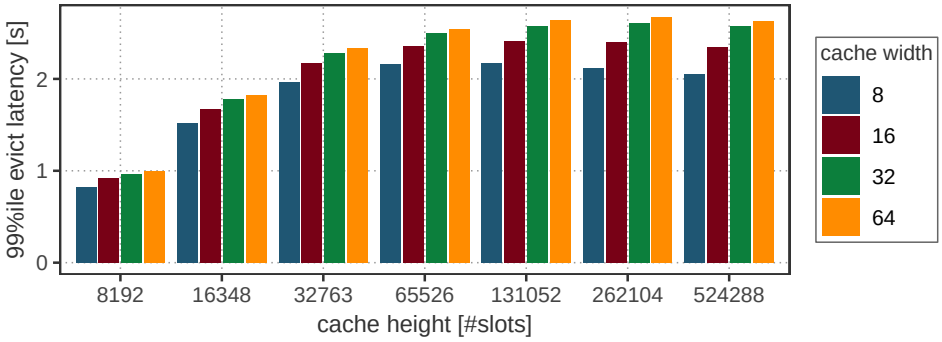


Figure 5.4: 99%ile eviction latency for different cache configurations

is the sum of the sizes of the `pkt` and `gpv` relations. Figure 5.5 shows these results in detail.

In our experiments, we inserted up to around 1 billion records into TimescaleDB. While we did not go beyond this, Timescale has been successfully used with database sizes beyond 500 billion records [17]. Our dataset of a 10Gbit/s wide-area link had an average packet rate of 330K/s [43]. Given a storage budget of 500 billion rows, at this packet rate, TimescaleDB could store around 17 days of packet-level data.

In a practical deployment we imagine that an operator would not necessarily store every single packet record forever. For example, a data retention policy could be defined in which every GPV header is stored and packet records are only retained within a storage budget. Various different policies are imaginable. We leave this discussion open for future research. Our relational model is designed such that most queries would still succeed when a GPV header does not reference any packet records anymore. The query would then return already aggregated instead of per-packet data.

5.6 Conclusion

Network monitoring is increasingly important in the operation of today's large and complex networks. With the introduction of modern programmable switches, there are more opportunities than ever before to collect high fidelity measurements. As recent real time telemetry and analytics systems have demonstrated, this can provide visibility into network conditions that enable powerful

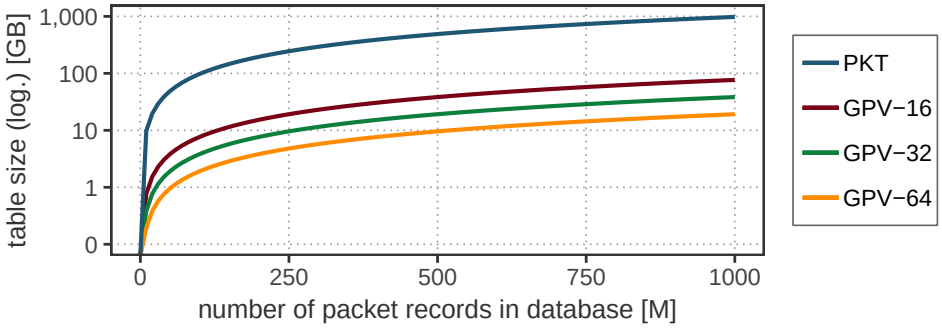


Figure 5.5: Timescale physical database size for different relations as a function of packets stored

new monitoring applications. But often, visibility into current network conditions is not enough. For debuggers, security systems, and many other applications, it is critical to also have visibility into past network conditions.

In this chapter, we identify this need for retrospective network analytics and show how such a system can be realized. We leverage recent trends in database technology, namely time-series databases. While most time series databases are implemented as NoSQL, key-value stores with custom query interfaces, we motivate why for this workload, a traditional relational database model is better-suited. We study the feasibility of using a relational model based time-series database (TimescaleDB) for network monitoring records.

We identify the main challenges of this approach and design strategies and optimizations to tailor an existing database engine for retrospective network analytics. These strategies improve system efficiency significantly and the resulting prototype serves as both a demonstration of feasibility and an important first step towards a complete solution. With this prototype, we explore features of retrospective queries and motivating use cases.

Chapter 6

Future Work and Conclusion

6.1 Future Work

We first lay out areas for future research of the three main components of Toccoa (sections 6.1.1, 6.1.2, and 6.1.3) before briefly elaborating on future work on the better integration of all three components through a unified, integrated network monitoring orchestration plane (section 6.1.4).

6.1.1 *Flow

*Flow is a high-performance network telemetry system that uses a novel in-network cache data structure to export grouped packet vectors (GPVs) at line rates of several Terabits per second on modern programmable forwarding engines.

Protocol-dependent Feature Extraction

Currently, the *Flow pipeline treats every packet that enters the PFE in the same way and as described in section 3.5. Since different traffic types have different characteristics and packet-level features that are of interest to monitoring applications, we imagine supporting different feature sets for different network protocols. For example, the field that is currently used for TCP sequence numbers could be used for ICMP error codes in case the packet is an ICMP packet. This would allow for more protocol-specific insight and allow for even finer-grained network monitoring.

Dynamic Feature Extraction

Additionally, we imagine the granularity at which records are generated to be runtime configurable. P4 data planes can expose a runtime API (commonly implemented through a remote procedure

call (RPC)) interface, such as gRPC[8] or Thrift[2]. Using a dynamic initial processing stage in *Flow and the corresponding runtime API it would be possible to temporarily export a subset of traffic at a higher granularity, e.g, including the first n bytes of payload data.

While it is certainly not feasible to export all packets including their payload permanently to the analytics platform, it might be useful to temporarily export packets coming from a specific source as full mirrored packets, for example during an attack. On the other hand, packets that are of low interest can always be aggregated as flow records without individual packet features. Changing these policies during network operation without reloading the PFE can significantly increase monitoring flexibility for the operator.

6.1.2 Jetstream

Using Jetstream, we showed that high coverage and high granularity network analytics is feasible in software. Jetstream is highly optimized for packet analytics workloads as it scales to tens of millions of packet records per second per core. It does that while still providing high programmability and flexibility for the network operator.

Runtime Configurability

Similar to *Flow, Jetstream is also missing features for runtime configurability. While we do not envision complete Jetstream applications to change at runtime, an API should be exposed that allows to scale up or scale down the processing graph to react to changes in processing demand. Additionally, it would be useful to change parameters for standard library processors (such as *filter* or *reduce*) at runtime.

Programming Interface

Jetstream provides a C++ API that is designed to speed up application development. Writing applications for Jetstream, still however requires some software engineering knowledge. As network administrators are likely the primary user group of network analytics systems, a more user-friendly query interface with richer abstractions would be beneficial. There is a rich body of work around programming language abstractions for network programming [73, 119, 161, 112, 186]. In particular

NetQRE [186] demonstrates a practical interface for quantitative network queries. We believe integrating such an interface into Jetstream can help adoption.

6.1.3 Persistent Interactive Queries

In our work on PIQ we identified the main challenges associated with record persistence for network monitoring records, i.e., injecting data, storing data, and finally querying data. We conclude that the relational database model together with time-series oriented functions and optimizations is an ideal and highly scalable platform to realize a persistence and retrospective query system for modern network monitoring solutions.

Performance Improvements

Even though we applied optimizations for better insert throughput of PostgreSQL, our prototype's performance is still not sufficient to store information about every single packet in a network at high rates. We conducted initial experiments with several ways to further optimize the write performance by parallelizing write operations and writing directly into PostgreSQL's underlying data structures. We will continue this line of work.

Application Integration

We also showed how SQL can be a powerful query and analytics tool based on its expressiveness, versatility and ecosystem of tools. Still, we would like to explore the area of object relational mappers (ORM) to build a more flexible and interactive command line query system. An ORM would have the advantage that fetched records from the database can be further analyzed and modified using the particular programming language. Most modern languages include functional programming mechanisms and library tools for mapping and reducing data making exploratory data analysis even simpler.

6.1.4 Orchestration

This thesis introduces Toccoa, a comprehensive network telemetry and analytics solution. Internally, Toccoa consists of several modules that expose various programmability and configuration

interfaces. For example, the telemetry component is controlled through a Thrift runtime API, while the analytics component exposes a GRPC API and is currently orchestrated through a collection of scripts.

We believe that a unified orchestration system is a key requirement for real-world deployment of Toccoa. Using an orchestration interface, operators should be able to configure at what levels the traffic in their network should be aggregated across the telemetry, analytics, and persistence planes. Furthermore, a unified configuration interface can be used (in a manner similar to the vision of software-defined networking) to obtain a bird's eye view of an entire network at the monitoring level instead of separately looking into exported packet or flow streams at the individual devices or sensors.

A significant part of enabling a network-wide view, is the correlation of network records across devices in order to trace back the exact history of a particular packet across an organizations's network. This feature can help pinpointing equipment failures to specific device queues and misconfigurations to a single device. Additionally, for auditing and retrospective analysis of attacks, a full packet history can indicate what areas of the infrastructure have been attacked or in which area sensitive information may have been extracted.

6.2 Conclusion

Monitoring plays a large role in network operation and management. As networks grow in size, traffic volume, and complexity, this will only become more important and challenging. In this thesis, we propose a novel network monitoring architecture, called Toccoa. Toccoa's design is based on the insight, that telemetry and analytics need to be tightly and carefully integrated to build cloud-scale network monitoring solutions.

While increasingly advanced telemetry planes can leverage line-rate programmable switch hardware to provide finer-grained and more information rich data than ever before, we answer the question how this vision can efficiently be implemented and deployed in practice. Prior telemetry plane systems have focused on leveraging PFEs to scale efficiently with respect to through-

put, but have not addressed the equally important requirement of scaling to support multiple concurrent applications with dynamic measurement needs. As a solution, we introduced *Flow, a PFE-accelerated, analytics-aware telemetry system that supports dynamic measurement from many concurrent applications without sacrificing efficiency or flexibility.

Up to this time, a crucial and almost entirely unaddressed challenge in network monitoring is analytics plane performance. Current systems work around the issue by reducing the amount of data, and information that the telemetry plane exports, e.g., by filtering or aggregating in switch hardware. This reduces the flexibility of the overall monitoring system, as it only works for certain applications and in networks with state-of-the-art programmable switches. To identify a practical solution, we examined the differences between general stream processing workloads and network analytics workloads and uncovered the root causes of bottlenecks in current systems. Based on our discoveries, we introduce Jetstream, a flexible stream processor optimized for network analytics workloads. Jetstream is a more practical and straightforward solution to mitigate analytics plane bottlenecks. Through this, Jetstream makes it practical for monitoring applications to analyze every packet in software. This enables advanced monitoring, increases flexibility, and simplifies the network data plane.

Finally, we elaborate on and give directions for the third required and important component for a comprehensive network monitoring and analytics solution, that is secondary and interactive queries. While our proposed telemetry and analytics planes can scale up to and sustain data center scale traffic rates, it would still be infeasible to implement intrusion detection or performance monitoring systems in this framework that analyze and detect every possible anomaly. Therefore, we introduce a third component that is optimized for interaction and effectiveness in making large amounts of data comprehensible.

In summary, we demonstrate how Toccoa can pave the way for next-generation monitoring applications. The capabilities of our systems go far beyond what is currently possible at data center scales. We believe that our holistic approach is a significant step toward the future of network monitoring and analytics.

Bibliography

[1] Apache Hadoop. https://hadoop.apache.org/.

[2] Apache Thrift RPC. https://thrift.apache.org.

[3] Broadcom ships tomahawk 3. https://people.ucsc.edu/~warner/Bufs/StrataXGS-Tomahawk3-Briefing-Deck-FINAL-DS.pdf.

[4] Center for applied internet data analysis (caida). https://www.caida.org.

[5] Data Plane Development Kit. https://dpdk.org.

[6] DB Toaster. https://dbtoaster.github.io.

[7] Grafana. https://grafana.com/.

[8] gRPC. https://grpc.io.

[9] Hdfs architecture guide. https://hadoop.apache.org/docs/current1/hdfs_design.html.

[10] OpenSOC: Big Data Security Analytics Framework.

[11] Prometheus. https://prometheus.io/.

[12] Rabbitmq. https://www.rabbitmq.com/.

[13] Redis: in-memory data structure store. https://redis.io/.

[14] S4: distributed stream computing platform. https://incubator.apache.org/s4/.

[15] Samza. https://samza.incubator.apache.org/.

[16] Storm, distributed and fault-tolerant realtime computation. https://storm.incubator.apache.org/.

[17] Talk: Rearchitecting a SQL Database for Time-Series Data. https://www.youtube.com/watch?v=eQKbbCg0NqE.

[18] Tcpdump & libpcap. https://www.tcpdump.org.

[19] Toccoa/ocoee river. https://en.wikipedia.org/wiki/Toccoa/Ocoee_River.

[20] Ethane: Taking control of the enterprise. ACM SIGCOMM Computer Communication Review, 37(4):1–12, 2007.

[21] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10), volume 7, pages 19–19, 2010.

[22] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive Online Scheduling in Storm. In Proceedings of the 7th ACM International Conference on Distributed Event-based Systems, DEBS '13, pages 207–218, New York, NY, USA, 2013. ACM.

[23] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In Proceedings of the 23rd international conference on Parallel architectures and compilation, pages 303–316. ACM, 2014.

[24] Apache Software Foundation. Flink. `https://flink.apache.org`.

[25] Apache Software Foundation. Kafka. `http://kafka.apache.org`.

[26] Apache Software Foundation. Spark streaming. `https://spark.apache.org/streaming/`.

[27] Marcos D. Assunção, Rodrigo N. Calheiros, Silvia Bianchi, Marco A.S. Netto, and Rajkumar Buyya. Big data computing and clouds. J. Parallel Distrib. Comput., 79(C):3–15, May 2015.

[28] T. Auld, A. W. Moore, and S. F. Gull. Bayesian neural networks for internet traffic classification. IEEE Transactions on Neural Networks, 18(1):223–239, Jan 2007.

[29] D Awduche. Requirements for traffic engineering over mpls. `https://tools.ietf.org/html/rfc2702`, 1999.

[30] F Baker. Cisco architecture for lawful intercept in ip networks. `https://tools.ietf.org/html/rfc3924`, 2004.

[31] F Baker. Requirements for ip flow information export (ipfix). `https://tools.ietf.org/html/rfc3917`, 2004.

[32] Jonathan C Beard, Peng Li, and Roger D Chamberlain. Raftlib: a c++ template library for high performance stream parallel processing. In Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, pages 96–105. ACM, 2015.

[33] Jonathan C Beard, Peng Li, and Roger D Chamberlain. Raftlib: A c++ template library for high performance stream parallel processing. International Journal of High Performance Computing Applications, 2016.

[34] Jim Belton. Hash table shootout. `https://jimbelton.wordpress.com/2015/11/27/hash-table-shootout-updated/`.

[35] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, pages 267–280. ACM, 2010.

[36] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies, page 8. ACM, 2011.

[37] Nikolaj Bjorner, Marco Canini, and Nik Sultana. Report on networking and programming languages 2017. ACM SIGCOMM Computer Communication Review, 47(5):39–41, 2017.

[38] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. SIGCOMM Comput. Commun. Rev., 44(3):87–95, jul 2014.

[39] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In ACM SIGCOMM Computer Communication Review, volume 43, pages 99–110. ACM, 2013.

[40] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[41] Alessio Botta, Alberto Dainotti, and Antonio Pescapé. Do you trust your software-based traffic generator? IEEE Communications Magazine, 48(9), 2010.

[42] Jake Brutlag. Speed matters. `https://ai.googleblog.com/2009/06/speed-matters.html`, 2009.

[43] CAIDA. Trace statistics for caida passive oc48 and oc192 traces – 2015-02-19. `https://www.caida.org/data/passive/trace_stats/`.

[44] CAIDA. Statistics for caida 2015 chicago traces. `https://www.caida.org/data/passive/trace_stats/`, 2015.

[45] CAIDA. Trace statistics for caida passive oc48 and oc192 traces – 2015-12-17. `https://www.caida.org/data/passive/trace_stats/`, 2015.

[46] Cavium. Cavium / xpliant cnx880xx product brief. `https://www.cavium.com/pdfFiles/CNX880XX_PB_Rev1.pdf?x=2`, 2015.

[47] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. ACM Comput. Surv., 41(3):15:1–15:58, July 2009.

[48] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. Understanding tcp incast throughput collapse in datacenter networks. In Proceedings of the 1st ACM workshop on Research on enterprise networking, pages 73–82. ACM, 2009.

[49] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. drmt: Disaggregated programmable switching. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pages 1–14. ACM, 2017.

[50] N M Mosharaf Kabir Chowdhury and Raouf Boutaba. A Survey of Network Virtualization. Comput. Netw., 54(5):862–876, apr 2010.

[51] Cisco. The cisco flow processor: Cisco's next generation network processor solution overview. http://www.cisco.com/c/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/solution_overview_c22-448936.html.

[52] Cisco. Introduction to cisco ios netflow. https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html, 2012.

[53] Cisco. Cisco netflow generation appliance 3340 data sheet. http://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/netflow-generation-3000-series-appliances/data_sheet_c78-720958.html, July 2015.

[54] Cisco. Cisco nexus 9200 platform switches architecture. https://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-737204.pdf, 2016.

[55] Cisco Systems Inc. Cisco Visual Networking Index: Forecast and Trends, 2017–2022. Technical report, 2018.

[56] Benoit Claise. Cisco systems netflow services export version 9. https://tools.ietf.org/html/rfc3954, 2004.

[57] Benoit Claise, Brian Trammell, and Paul Aitken. Specification of the ip flow information export (ipfix) protocol for the exchange of flow information. Technical report, 2013.

[58] Cloudflare. Slowloris ddos attack. https://www.cloudflare.com/learning/ddos/ddos-attack-tools/slowloris.

[59] Cloudlab. Cloudlab. https://www.cloudlab.us.

[60] Michael Collins. Network security through data analysis: building situational awareness. OReilly Media, 2014.

[61] P4 Language Consortium. P4 switch intrinsic metadata. https://github.com/p4lang/switch/blob/master/p4src/includes/intrinsic.p4.

[62] M Cotton. Internet assigned numbers authority (iana) procedures for the management of the service name and transport protocol port number registry. https://tools.ietf.org/html/rfc6335, 2011.

[63] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic classification through simple statistical fingerprinting. SIGCOMM Comput. Commun. Rev., 37(1):5–16, January 2007.

[64] Luca Deri and NETikos SpA. nprobe: an open source netflow probe for gigabit networks. In TERENA Networking Conference, 2003.

[65] Dimensional Research. Network Complexity, Change, and Human Factors are Failing the Business. Technical report.

[66] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 15–28. ACM, 2009.

[67] Hilmi E Egilmez, Seyhan Civanlar, and A Murat Tekalp. An optimization framework for qos-enabled adaptive video streaming over openflow networks. IEEE Transactions on Multimedia, 15(3):710–715, 2013.

[68] Endace. Endaceflow 4000 series netflow generators. `https://www.endace.com/endace-netflow-datasheet.pdf`, 2016.

[69] Facebook. Fbflow dataset. `https://www.facebook.com/network-analytics`.

[70] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN: An Intellectual History of Programmable Networks. ACM SIGCOMM Computer Communication Review, 44(2):87–98, apr 2014.

[71] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational ip networks: Methodology and experience. IEEE/ACM Trans. Netw., 9(3):265–280, June 2001.

[72] Michael Finsterbusch, Chris Richter, Eduardo Rocha, Jean-Alexander Muller, and Klaus Haenssgen. A survey of payload-based traffic classification approaches. IEEE Communications Surveys and Tutorials, PP:1–22, 12 2013.

[73] Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola, and David Walker. Frenetic: A high-level language for openflow networks. In Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow, PRESTO '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[74] Open Networking Foundation. OpenFlow 1.5.1 Switch Specification. `https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf`, 2015.

[75] Google. Protocol buffers. `https://developers.google.com/protocol-buffers/`.

[76] Google. Transparency report. `https://transparencyreport.google.com/https/overview?hl=en`, 2019.

[77] Albert Greenberg, Gisli Hjalmtysson, David A Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. ACM SIGCOMM Computer Communication Review, 35:41–54, 2005.

[78] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, pages 139–152, New York, NY, USA, 2015. ACM.

[79] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven Streaming Network Telemetry. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, pages 357–371, New York, NY, USA, 2018. ACM.

[80] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), volume 14, pages 71–85, 2014.

[81] Tony Hey and Gyuri Papay. The Computing Universe. Cambridge University Press, 2015.

[82] Michael Hicks, Jonathan T Moore, David Wetherall, and Scott Nettles. Experiences with capsule-based active networking. In DARPA Active NEtworks Conference and Exposition, 2002. Proceedings, pages 16–24. IEEE, 2002.

[83] Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto, and Aiko Pras. Flow monitoring explained: from packet capture to data analysis with netflow and ipfix. IEEE Communications Surveys & Tutorials, 16(4):2037–2064, 2014.

[84] Intel. DPDK Performance Report. `https://fast.dpdk.org/doc/perf/DPDK_16_11_Intel_NIC_performance_report.pdf`.

[85] Intel. Introduction to intel ethernet flow director and memcached performance. `https://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-flow-director.html`, 2014.

[86] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining network intents for self-driving networks. SIGCOMM Comput. Commun. Rev., 48(5):55–63, January 2019.

[87] Sharad Jaiswal, Gianluca Iannaccone, Christophe Diot, Jim Kurose, and Don Towsley. Measurement and classification of out-of-sequence packets in a tier-1 ip backbone. IEEE/ACM Trans. Netw., 15(1):54–66, February 2007.

[88] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In ACM SIGCOMM Computer Communication Review, volume 44, pages 3–14. ACM, 2014.

[89] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 35–49, Renton, WA, 2018. USENIX Association.

[90] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[91] T. Karagiannis, A. Broido, N. Brownlee, K. C. Claffy, and M. Faloutsos. Is p2p dying or just hiding? [p2p traffic measurement], Nov 2004.

[92] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst., 28(1):51–55, March 2003.

[93] Changhoon Kim, Parag Bhide, Ed Doe, Hugh Holbrook, Anoop Ghanwani, Dan Daly, Mukesh Hira, and Bruce Davie. In-band network telemetry (int). `https://p4.org/assets/INT-current-spec.pdf`, 2016.

[94] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In ACM SIGCOMM, 2015.

[95] Davis E. King. Dlib-ml: A machine learning toolkit. Journal of Machine Learning Research, 10:1755–1758, 2009.

[96] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. ACM Trans. Comput. Syst., 18(3):263–297, Aug 2000.

[97] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network Virtualization in Multi-tenant Datacenters. In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 203–216, Seattle, WA, 2014. USENIX.

[98] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G. Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of age: A longitudinal study of tls deployment. In Proceedings of the Internet Measurement Conference 2018, IMC '18, pages 415–428, New York, NY, USA, 2018. ACM.

[99] Christos Kozanitis, John Huber, Sushil Singh, and George Varghese. Leaping Multiple Headers in a Single Bound: Wire-speed Parsing Using the Kangaroo System. In Proceedings of the 29th Conference on Information Communications, INFOCOM'10, pages 830–838, Piscataway, NJ, USA, 2010. IEEE Press.

[100] Marshall A. Kuypers, Thomas Maillart, and Elisabeth Paté-Cornell. An empirical analysis of cyber security incidents at a large organization. 2016.

[101] Maciej Kuźniar, Peter Perešíni, Dejan Kostić, and Marco Canini. Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches. Computer Networks, 136:22 – 36, 2018.

[102] Sihyung Lee, Kyriaki Levanti, and Hyong S. Kim. Network monitoring: Present and future. Computer Networks, 65:94–98, 2014.

[103] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: a better netflow for data centers. In 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), pages 311–324, 2016.

[104] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 1–13. ACM, 2011.

[105] Hyeontaek Lim, Donsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. USENIX, 2014.

[106] Po-Ching Lin, Y.R. Lin, Tsern-Huei Lee, and Yuan-Cheng Lai. Using string matching for deep packet inspection. Computer, 41:23 – 28, 05 2008.

[107] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, pages 101–114, New York, NY, USA, 2016. ACM.

[108] Carl Livadas, Robert Walsh, David Lapsley, and W Timothy Strayer. Using machine learning technliques to identify botnet traffic. In Local Computer Networks, Proceedings 2006 31st IEEE Conference on, pages 967–974. IEEE, 2006.

[109] Wei Lu and Ali A Ghorbani. Network anomaly detection based on wavelet analysis. EURASIP Journal on Advances in Signal Processing, 2009:4, 2009.

[110] Duncan MacMichael. Introduction to receive side scaling. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling`, 2017.

[111] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, pages 459–473. USENIX Association, 2014.

[112] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. Event-driven network programming. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16, pages 369–385, New York, NY, USA, 2016. ACM.

[113] Anthony McGregor, Mark A. Hall, Perry Lorier, and James Brunskill. Flow clustering using machine learning techniques. In PAM, 2004.

[114] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev., 38(2):69–74, mar 2008.

[115] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. StreamBox: Modern Stream Processing on a Multicore Machine. In 2017 USENIX Annual Technical Conference (USENIX ATC 17), pages 617–629, Santa Clara, CA, 2017. {USENIX} Association.

[116] Oliver Michel, Eric Keller, John Sonchack, and Jonathan M Smith. Packet-level analytics in software without compromises. In Proceedings of the 10th USENIX Workshop on Hot Topics in Cloud Computing, Boston, MA, 2018. USENIX Association.

[117] Microway. Detailed specifications of the skylake-sp intel xeon processor family. https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-skylake-sp-intel-xeon-processor-scalable-family-cpus/.

[118] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection. In Proceedings of the 2018 Network and Distributed System Security Symposium, number February, San Diego, CA, 2018.

[119] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pages 1–13, Lombard, IL, 2013. USENIX Association.

[120] Andrew W. Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. In Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '05, pages 50–60, New York, NY, USA, 2005. ACM.

[121] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, pages 129–143, New York, NY, USA, 2016. ACM.

[122] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pages 85–98. ACM, 2017.

[123] Netronome. Agilio cx smartnics. https://www.netronome.com/products/agilio-cx/.

[124] Barefoot Networks. Barefoot tofino. https://www.barefootnetworks.com/technology/#tofino.

[125] Thuy Thanh Nguyen and Grenville J. Armitage. Synthetic sub-flow pairs for timely and stable ip traffic identification. 2006.

[126] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. IEEE Communications Surveys & Tutorials, 10(4):56–76, 2008.

[127] Norton. Zero-day vulnerability: What it is, and how it works. https://us.norton.com/internetsecurity-emerging-threats-how-do-zero-day-vulnerabilities-work-30sectech.html, 2019.

[128] Ntop. Pf_ring high-speed packet capture, filtering and analysis. https://www.ntop.org/products/packet-capture/pf_ring/.

[129] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. Big data technologies: A survey. Journal of King Saud University - Computer and Information Sciences, 30(4):431 – 448, 2018.

[130] Recep Ozdag. Intel® ethernet switch fm6000 series-software defined networking, 2012.

[131] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of nfv. In OSDI, pages 203–216, 2016.

[132] Christoforos Panos, Christos Xenakis, and Ioannis Stavrakakis. An evaluation of anomaly-based intrusion detection engines for mobile ad hoc networks. In Steven Furnell, Costas Lambrinoudakis, and Günther Pernul, editors, Trust, Privacy and Security in Digital Business, pages 150–160, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[133] Konstantina Papagiannaki, Rene Cruz, and Christophe Diot. Network performance monitoring at small time scales. In Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, IMC '03, pages 295–300, New York, NY, USA, 2003. ACM.

[134] J. Park, H. Tyan, and C. . J. Kuo. Internet traffic classification for scalable qos provision. In 2006 IEEE International Conference on Multimedia and Expo, pages 1221–1224, July 2006.

[135] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. Technical report, 2001.

[136] Remi Philippe. Cisco advantage series next generation data center flow telemetry, 2016.

[137] Postgres. Postgresql database system. `https://www.postgresql.org`.

[138] Postgres. Postgresql documentation: Copy. `https://www.postgresql.org/docs/current/sql-copy.html`.

[139] Arcot Rajasekar, Frank Vernon, Todd Hansen, Kent Lindquist, and John Orcutt. Virtual Object Ring Buffer : A Framework for Real-time Data Grid. Technical report, San Diego Supercomputer Center, UCSD.

[140] Ron Renwick. In-band network telemetry – it's not rocket science. `https://www.netronome.com/blog/in-band-network-telemetry-its-not-rocket-science/`, 2017.

[141] Luigi Rizzo and Matteo Landi. Netmap: Memory Mapped Access to Network Devices. In Proc. ACM SIGCOMM, 2011.

[142] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In Lisa, volume 99, pages 229–238, 1999.

[143] Matthew Roughan, Subhabrata Sen, Oliver Spatscheck, and Nick Duffield. Class-of-service mapping for qos: A statistical signature-based approach to ip traffic classification. In Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, IMC '04, pages 135–148, New York, NY, USA, 2004. ACM.

[144] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In ACM SIGCOMM Computer Communication Review, volume 45, pages 123–137. ACM, 2015.

[145] Barun Kumar Saha, Deepaknath Tandur, Luca Haab, and Lukasz Podleski. Intent-based networks: An industrial perspective. In Proceedings of the 1st International Workshop on Future Industrial Communication Networks, FICN '18, pages 35–40, New York, NY, USA, 2018. ACM.

[146] sFlow. sflow sampling rates. http://blog.sflow.com/2009/06/sampling-rates.html, June 2009.

[147] Danfeng Shan, Wanchun Jiang, and Fengyuan Ren. Absorbing micro-burst traffic by enhancing dynamic threshold policy of data center switches. In Computer Communications (INFOCOM), 2015 IEEE Conference on, pages 118–126. IEEE, 2015.

[148] Naveen Kr Sharma, Antoine Kaufmann, Thomas E Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 67–82, 2017.

[149] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[150] Z. Shu, J. Wan, J. Lin, S. Wang, D. Li, S. Rho, and C. Yang. Traffic engineering in software-defined networking: Measurement and management. IEEE Access, 4:3246–3256, 2016.

[151] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In Proceedings of the 2016 ACM SIGCOMM Conference, pages 15–28. ACM, 2016.

[152] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In Proceedings of the 2016 ACM SIGCOMM Conference, pages 15–28. ACM, 2016.

[153] Malte Skarupke. I wrote the fastest hashtable. https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/.

[154] Chakchai So-In. A survey of network traffic monitoring and analysis tools. Cse 576m computer system analysis project, Washington University in St. Louis, 2009.

[155] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Turboflow: Information rich flow record generation on commodity switches. In Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, pages 11:1–11:16, New York, NY, USA, 2018. ACM.

[156] John Sonchack, Adam J Aviv, and Jonathan M Smith. Enabling Practical Software-defined Networking Security Applications with OFX. In NDSS 2016, number February, pages 21–24, San Diego, CA, 2016.

[157] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated monitoring to concurrent and dynamic queries with *flow. In Proceedings of the 2018 USENIX Annual Technical Conference (ATC), Boston, MA, 2018. USENIX Association.

[158] Haoyu Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In Proceedings of the Second ACM SIGCOMM Workshop on Hot

Topics in Software Defined Networking, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[159] Steve Souders. Velocity and the bottom line. `http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html`, 2009.

[160] A. Soule, A. Nucci, R. L. Cruz, E. Leonardi, and N. Taft. Estimating dynamic traffic matrices by using viable routing changes. IEEE/ACM Transactions on Networking, 15(3):485–498, June 2007.

[161] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, CoNEXT '14, pages 213–226, New York, NY, USA, 2014. ACM.

[162] Anna Sperotto, Gregor Schaffrath, Ramin Sadre, Cristian Morariu, Aiko Pras, and Burkhard Stiller. An overview of ip flow-based intrusion detection. IEEE communications surveys & tutorials, 12(3):343–356, 2010.

[163] Anna Sperotto, Gregor Schaffrath, Ramin Sadre, Cristian Morariu, Aiko Pras, and Burkhard Stiller. An overview of ip flow-based intrusion detection. IEEE communications surveys & tutorials, 12(3):343–356, 2010.

[164] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on, pages 228–237. IEEE, 2014.

[165] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, pages 426–439, New York, NY, USA, 2016. ACM.

[166] T. T. t. Nguyen and G. Armitage. Training on multiple sub-flows to optimise the use of machine learning classifiers in real-world ip networks. In Proceedings. 2006 31st IEEE Conference on Local Computer Networks, pages 369–376, Nov 2006.

[167] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed Network Monitoring and Debugging with SwitchPointer. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 453–456, Renton, WA, 2018. USENIX Association.

[168] Mellanox Technologies. Rdma and roce for ethernet network efficiency performance. `https://www.mellanox.com/page/products_dyn?product_family=79`.

[169] Mellanox Technologies. What is nvme over fabrics? `https://community.mellanox.com/s/article/what-is-nvme-over-fabrics-x`.

[170] Martin Thompson, D Farley, M Barker, P Gee, and A Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. Technical report, code.google.com, 2011.

[171] Sameer Tilak, Paul Hubbard, Matt Miller, and Tony Fountain. The ring buffer network bus (RBNB) dataturbine streaming data middleware for environmental observing systems. In Proceedings - e-Science 2007, 3rd IEEE International Conference on e-Science and Grid Computing, pages 125–132, 2007.

[172] Timescale. Timescale db. https://www.timescale.com/.

[173] Twitter. The infrastructure behind twitter - scale. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html.

[174] Twitter. Observability at twitter - technical overview. https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-i.html.

[175] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In NSDI, pages 407–420, 2017.

[176] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and Adaptable Stream Processing at Scale. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 374–389, New York, NY, USA, 2017. ACM.

[177] Ben Wagner. Deep packet inspection and internet censorship: International convergence on an 'integrated technology of control'. 2009.

[178] Bob Wheeler. A new era of network processing. The Linley Group, Tech. Rep, 2013.

[179] Nigel Williams, Sebastian Zander, and Grenville Armitage. A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification. SIGCOMM Comput. Commun. Rev., 36(5):5–16, October 2006.

[180] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. In ACM SIGCOMM Computer Communication Review, volume 43, pages 123–134. ACM, 2013.

[181] Wireshark. Tzsp reference. https://www.wireshark.org/docs/dfref/t/tzsp.html.

[182] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. In Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '12, pages 419–430, New York, NY, USA, 2012. ACM.

[183] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated bug removal for software-defined networks. In NSDI, pages 719–733, 2017.

[184] Minlan Yu. Network telemetry: Towards a top-down approach. SIGCOMM Comput. Commun. Rev., 49(1):11–17, February 2019.

[185] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, NSDI '13, pages 29–42, Berkeley, CA, USA, 2013. USENIX Association.

[186] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with netqre. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, pages 99–112, New York, NY, USA, 2017. ACM.

[187] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: A unified engine for big data processing. Communications of the ACM, 59(11):56–65.

[188] S. Zander, T. Nguyen, and G. Armitage. Automated traffic classification and application identification using machine learning. In The IEEE Conference on Local Computer Networks 30th Anniversary (LCN'05)l, pages 250–257, Nov 2005.

[189] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. AWStream: Adaptive Wide-area Streaming Analytics. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, pages 236–252, New York, NY, USA, 2018. ACM.

[190] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In Proceedings of the 2017 Internet Measurement Conference, IMC '17, pages 78–85, New York, NY, USA, 2017. ACM.

[191] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In Proceedings of the ninth ACM conference on Emerging networking experiments and technologies, pages 97–108. ACM, 2013.

[192] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 615–626. ACM, 2010.

[193] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In ACM SIGCOMM Computer Communication Review, volume 45, pages 479–491. ACM, 2015.

[194] Daniel Zobel. Does my cisco device support netflow export? `https://kb.paessler.com/en/topic/5333-does-my-cisco-device-router-switch-support-netflow-export`, June 2010.