Software Packet-Level Network Analytics at Cloud Scale

Oliver Michel, John Sonchack, Greg Cusack, Maziyar Nazari, Eric Keller, and Jonathan M. Smith, *Fellow, IEEE*

***Abstract*—As networks grow in speed, scale, and complexity, operating them reliably requires continuous monitoring and increasingly sophisticated analytics. Because of these requirements, the platforms that support analytics in cloud-scale networks face demands for both higher throughput (to keep up with high packet rates) and increased generality and programmability (to cover a wider range of applications). Recent proposals have worked toward these goals by offloading analytics application logic to line-rate programmable data plane hardware, as scaling existing software analytics platforms is prohibitively expensive. The rigid design and constrained resources of data plane devices, however, fundamentally limit the types of analysis and the number of tasks that can run concurrently. In this article, we demonstrate that generality need not be sacrificed for high performance. Rather than offloading entire analytics applications to hardware, the core idea of our work is to offload only critical preprocessing tasks that are shared among applications (e.g., load balancing) to a line-rate hardware frontend while optimizing the core analytics software to exploit properties of network analytics workloads. Based on this design, we present Jetstream, a hybrid platform for network analytics that can run custom software-based analytics pipelines at throughputs of up to 250 million packets per second on a 16-core commodity server. Jetstream makes sophisticated, network-wide packet analytics feasible without compromising on generality or performance.**

***Index Terms*—Network monitoring and measurements, data center networks, performance management, security management, prototype implementation and testbed experimentation.**

I. Introduction

EFFECTIVE network management requires traffic analytics: the capability to mine critical information from packet streams, which can be used to trigger actions in the network or guide subsequent decisions. Traffic analytics is a core component in today's reliable networked systems that is used to help meet stringent security [1], correctness [2], [3], and performance guarantees [4], [5]. Historically, we largely relied on humans in a network operation center to watch some transformed version of the data (e.g., graphs) and manually interpret the data to then take action. This approach does not scale to today's data center or wide area networks which continue to grow in complexity, size, and traffic. Instead, today, the ability to continuously perform automated and sophisticated analytics across the entire infrastructure is imperative [6], [7].

Given the importance of the problem, in recent years, many novel and compelling architectures and systems for fine-grained network monitoring in cloud-scale environments have been presented [2], [8]–[11]. At the core of each proposed system is an underlying processing engine that analyzes raw data. The design of such a processing system is the focus of our work.

In an ideal world, the system would enable arbitrary, sophisticated analytics that consider every single packet traversing a network. A network operator should be able to write multiple custom analytics applications to run in parallel. These applications can be interactive queries or long-running, continuous analyses over a network packet data stream.

In a nutshell, the analytics system must be *general* to enable arbitrary and runtime-configurable applications through a programmable interface. Equally important is high *performance* to allow for economically feasible network-wide coverage and parallel analytics applications. This ideal of general, software-based analytics on every packet in a cloud-scale network is expensive to realize. Consequently, there has been a long history of work compromising on various dimensions with the goal of making this vision practical.

Historically, flow aggregation and sampling (e.g., with IPFIX [12]) have been the main tools of network operators to reduce the amount of information to analyze. Both approaches are appealing because they can be practically implemented in resource-constrained hardware switches. Aggregated network records, however, hinder fine-grained analytics that are required for a wide range of performance- and security monitoring use cases [8], [11], while sampling compromises on data fidelity and accuracy [13]–[15]. These limitations motivated researchers to propose custom algorithms and probabilistic data structures (e.g., sketches) that provide provable accuracy and can be implemented in hardware [16]–[18]. Still, sketching only supports basic statistical analysis, limiting generality. For example, more intricate analytics logic such as detecting a network loop, where a packet traverses the same switch twice, cannot be realized using sketches.

Manuscript received May 31, 2020; revised September 12, 2020 and November 16, 2020; accepted November 17, 2020. Date of publication February 11, 2021; date of current version March 11, 2021. This work has been funded by the National Science Foundation (NSF) under award 1652698 (CAREER) and NSF/VMWare award 1700527 (SDI-CSCS). The associate editor coordinating the review of this article and approving it for publication was T. Zinner. *(Corresponding author: Oliver Michel.)*

Oliver Michel was with the Department of Computer Science, University of Colorado Boulder, Boulder, CO 80309 USA. He is now with the Faculty of Computer Science, University of Vienna, 1010 Vienna, Austria (e-mail: oliver.michel@univie.ac.at).

John Sonchack is with Department of Computer Science, Princeton University, Princeton, NJ 08544 USA.

Greg Cusack, Maziyar Nazari, and Eric Keller are with the University of Colorado Boulder, Boulder, CO 80309 USA.

Jonathan M. Smith is with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104 USA.

Digital Object Identifier 10.1109/TNSM.2021.3058653

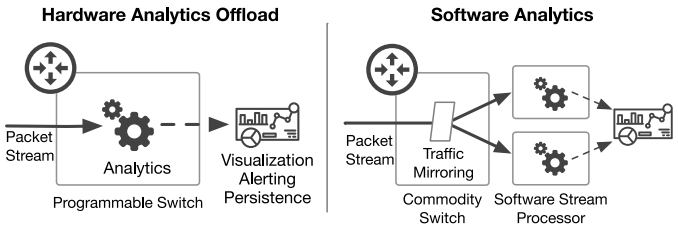


Fig. 1. Previous Network Analytics System Architectures.

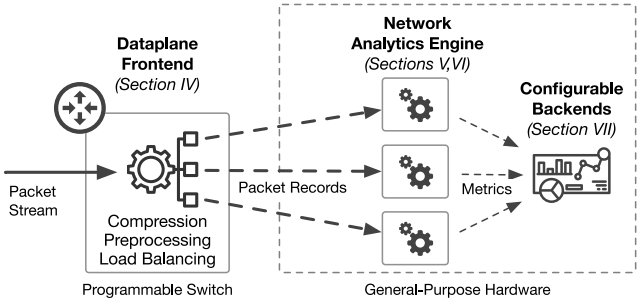


Fig. 2. Jetstream Architecture Overview.

These compromises on generality and data granularity are increasingly problematic today, as there is a growing number of applications that need to perform analytics on data from every single packet in a network, for example for machine learning in intrusion detection or traffic classification systems [19]–[26]. For these tasks, analysis is sophisticated and application-specific, and hence impractical to implement as a sketch or in hardware. To accommodate such applications, the community presented ways to analyze entire traces of packets in software. Performance limitations, however, meant that these proposals suffered from poor visibility, e.g., limited to a single switch [6] or a specific class of flows, which again makes them unsuitable for the many modern analytics applications mentioned above.

Today, we are left with two directions that research has taken toward the goal of being able to analyze every packet in a network for a wide range of applications. The first direction is to compile analytics tasks to run on modern programmable switches [8], [9] (see left side of Figure 1). This is challenging as hardware resources on these switches are heavily constrained. To illustrate this, we compiled the Sonata [9] queries available [27] to the Intel Tofino programmable forwarding engine (PFE) using two levels of refinement. Only two of the seven queries fit within the resource limits of the chip (see Section VIII-B). This leaves the other queries as not currently being practical and raises questions about the feasibility of enabling multiple queries to run simultaneously.

The second direction is to adapt a pure software architecture for network analytics, using a map-reduce-style, scale-out system such as dShark [10] (see right side of Figure 1). While this allows for horizontal scalability and supporting multiple queries simultaneously, performance is still a significant challenge. In an end-to-end performance evaluation, dShark's throughput is 10.6 million packets per second on a 16-core server. This would result in needing to dedicate 96 servers to monitor a single cluster in a modern data center [28] for a single application (see more in Section VIII-C).

In this article, we introduce a third direction that balances the two previously presented extremes. Our proposed system, Jetstream, uses a hardware-software co-design and can efficiently analyze hundreds of millions of packets per second for multiple simultaneous applications allowing for network-wide, packet-level analytics without compromises. Our design is based on two key strategies.

First, we leverage programmable switches for *system-level* offload: Rather than pushing down entire analytics applications to programmable data plane hardware (i.e., compiling a query to P4 [29]), Jetstream offloads system-level tasks that are necessary for *all* analytics applications to a programmable data plane frontend. For example, tasks such as extracting packet features, compressing and organizing packet records for efficient processing, and steering data streams can efficiently be implemented in hardware but are expensive to run in software. By offloading them to programmable switches, we eliminate much of a software analytics platform's work without overloading the programmable switch.

Our second strategy is to carefully optimize Jetstream's software component to exploit both the properties of network analytics workloads and our partitioning between hardware and software. For example, the structure of packet flows is inherently suitable for distribution across servers (see Section V-A). Since load balancing is offloaded to the programmable switch frontend, Jetstream's analytics pipelines (which run application-specific logic) can be designed to operate independently of each other. This eliminates resource contention to improve both performance and scalability. Finally, guided by workload characteristics, we apply a series of domain-specific system optimizations. These optimizations allow for significant performance gains over general-purpose systems without impacting application logic or accuracy. The resulting high-level architecture is depicted in Figure 2.

We implemented a complete prototype of Jetstream. The data plane frontend runs on a Barefoot Tofino PFE at line rate of 3.2 Tb per second and allows for dynamic adding, removing, and scaling of analytics tasks without reloading the programmable switch. The core software analytics engine is implemented in C++. It consists of a framework and a library for writing custom analytics pipelines that compute relevant metrics from network packet record input streams. It integrates optimizations that include kernel-bypass input/output, zero-copy message passing, high-throughput concurrent queues, batching, and accelerated hash tables. Lastly, a configurable backend for aggregating and querying metrics provides an interface to network operators or control platforms. We evaluate Jetstream with real-world traffic traces using seven example analytics applications: a heavy-hitter detector, a software load balancing profiler, a Slowloris DoS attack detector, a SSH brute force detector, a SYN flood detector, a TCP sequence analyzer, and a traffic statistics/accounting application.

We published partial results on an early design of the software analytics component of Jetstream in [30]. We now fundamentally extend the earlier processing engine by integrating

it with hardware-based telemetry systems and introducing an independent, parallel processing pipeline architecture as a core design strategy. Together with the data plane frontend and a database backend, this article provides an end-to-end system which we evaluate in a realistic multi-server deployment.

Our evaluation shows that individual application throughputs range from 5.4 to 15.9 million packets per second for a single core. Jetstream scales linearly with core count across machines (or between 86.4 and 254.4 million packets per second on a 16-core server). For comparison, using a 16-core server, Spark (Sonata's backend) can handle 1.4 million packets per second and dShark can handle 10.6 million packets per second. A task that would take 24 servers in dShark only requires a single Jetstream server, demonstrating how Jetstream's design and optimizations make the ideal of sophisticated and network-wide analytics practical.

In the remainder of this article, we first motivate the need for Jetstream by discussing the progression of analytics systems towards increasing generality (the ability to support a wider collection of applications) and performance (the ability to handle more traffic) in Section II. We then introduce Jetstream and its architecture in Section III. This architecture consists of three main components which are then detailed: the programmable data plane frontend (Section IV), the core software network analytics engine (Sections V and VI), and the on-demand aggregation and query backend (Section VII). We evaluate Jetstream in Section VIII and conclude in Section IX.

II. Motivation

With recent advances in networking technology, such as software-defined networking [31] and programmable data planes [29], [32], and the rapidly increasing scale of networks, there has been a flurry of research toward improving network monitoring and analytics. Each proposed system has moved us closer to the idealized goal of being able to perform general analytics on every packet in a network. The challenge, of course, is doing so in a cost- and resource-efficient manner. This is where each current analytics platform makes tradeoffs. In this section, we motivate the need for and the design of Jetstream by discussing the most relevant prior systems.

A. Sketching in the Data Plane

Sketching is among the most resource efficient approaches to custom analytics. Sketches leverage probabilistic data structures to compute summary statistics over large input datasets using a sub-linear amount of memory [33]. OpenSketch [17] provides a library of such sketches to be deployed in programmable hardware platforms, while UnivMon [18] introduces a universal streaming scheme, where a generic sketch in hardware preprocesses packet records at high rates and software applications compute application-specific metrics.

While extremely efficient in space requirements, sketches can only support certain classes of statistical functions and aggregate analysis as they lack visibility into individual packets. For example, an analysis task that cannot be represented with a sketch is the detection of packets that traverse the same switch twice, i.e., a network loop. By design, sketches also over- and under-count events and randomly lose information because of hash collisions in the underlying data structure.

In contrast, an analytics application running on top of Jetstream has visibility into every packet and can therefore calculate any statistic with full accuracy.

B. Packet-Level Software Analytics

There is a growing set of analytics tasks (particularly machine-learning intrusion detection and traffic classification systems) that cannot rely on sketching because they need to either analyze fields in each packet or perform sophisticated, application-specific analysis. Examples of the required packet-level data include packet inter-arrival times [19], TCP receive window [21], [24], and TCP flags [21], [24]. Analytics applications use these and other fields to compute: packet lengths statistics [20], [26], packet arrival order [23], and many other advanced and derived metrics (e.g., Fourier transforms of inter-arrival times, flow idle times, mean packet sizes, flow duration, number of TCP data packets) [19]–[26].

To support such applications, there have been proposals to process entire traces of packets in software. Planck [6] demonstrated the ability to mirror packets of interest to a management port of a switch which then sends traffic to an attached server for processing in software. Planck has limited scalability and incurs packet loss due to substantial oversubscription of the management port. To reduce the workload, NetSight [2] filters out traffic that is not of interest, using Berkeley packet-filter (BPF) style filters, before application-level processing, while Everflow [11] pushes both filtering and shuffling into data plane hardware. While these systems improve scalability, the heavy reliance on filtering limits their applicability to debugging tasks and increases operator burden, as operators must know what they are looking for a priori.

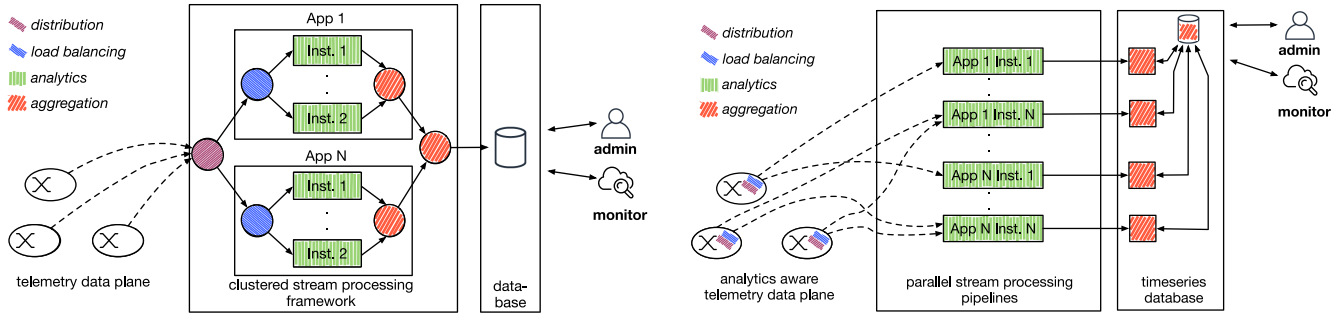
Finally, distributed measurement frameworks, such as SwitchPointer [34] or Confluo [35] collect features from regular network packets at the network's end hosts and perform lightweight analysis there. This approach lacks visibility into the core of the network and also requires analytics functionality and applications to be deployed on every single host at the network edge. Finally, Confluo applications must follow a rigid programming model limiting its applicability for the above mentioned applications.

In contrast, Jetstream's high throughput enables scaling without filtering, giving visibility into all packets collected from throughout the network. Applications can flexibly extract features and compute metrics of interest using a general-purpose language and an unrestricted programming model.

C. Compiled Queries in the Data Plane

With the emergence of programmable forwarding engine technologies (PFE) [29], [32], researchers have sought to use these platforms to solve scalability issues introduced by previous packet-level monitoring systems by compiling some or all of the processing into line rate hardware.

Marple [8] identified a set of fixed operators that can be compiled to a programmable forwarding engine and used to implement parts of a network monitoring query. This approach



(a) Straw man network analytics system architecture

(b) Jetstream network analytics architecture

Fig. 3. Telemetry-based network analytics system architectures.

offers great performance, but not all queries can fit within the resource constraints of a PFE. For those queries, performance is typically bottlenecked by the backend stream processor. Compiled queries are also problematic for other reasons. First, due to limited resources on these devices, only a small number of tasks can run in parallel [36], [37]. Second, reconfiguring data plane programs (i.e., changing the running monitoring query) is disruptive as it incurs device downtimes on the order of tens of seconds [37]. Third, applications are constrained to use the fixed set of operators available in the PFE programming model. While general, some applications [38] require metrics that are too complex for switch hardware to implement [39]. Fourth, deployment is challenging because overall system throughput is highly sensitive to the application, how it is split between hardware and software, and the workload characteristics (e.g., number of flows).

Sonata [9] reduced PFE memory requirements by introducing a method of iterative refinement for the PFE component of a query. This comes with two additional drawbacks, however. First, iterative refinement requires additional costly hardware resources. In our evaluation (see Section VIII-B), we find that refinement causes only two out of seven applications to be able to fit on the PFE. Second, refinement relaxes the temporal and logical constraints of a query. Events must last longer than a refinement window to be detected, which is on the order of seconds [9]. Further, even long-lived events can be missed because they may fail to match relaxed thresholds in the coarse-grained early stages of a refined query.

In contrast to these systems, Jetstream leverages hardware (switch) offload for preprocessing logic that is expensive in software and common to all applications. All example applications that we later discuss in Section VIII-A require feature extraction, record load balancing, and distribution. By offloading this system-level functionality (rather than application-specific tasks), Jetstream can accelerate *all* analytics tasks and scale predictably and efficiently with the number of running applications while eliminating the need to re-load switch logic to run new or additional applications.

D. General-Purpose Software Processing

An alternative approach to programmable data plane acceleration and offload is to optimize software-based analytics. Software platforms can support virtually any application and can be reconfigured without downtime; however, per-core processing rates are generally low, making operation in environments with high packet rates prohibitively expensive.

There are two orthogonal lines of work in this area. First, language-based tools, such as NetQRE [40] compile queries into efficient C++ programs. Second, and more related to Jetstream, are stream processing frameworks designed to run many applications concurrently and at scale, e.g., dShark [10]. While dShark performs much better than general-purpose stream processors (e.g., Spark, used as the backend of Marple [8] and Sonata [9]), its throughput is still low relative to network packet rates. Reported throughput for dShark, for example, is on the order of 10.6 million packets per second for a 16-core server running one application. This is several orders of magnitude lower than typical data center packet rates, effectively requiring racks full of servers just for analytics.

To understand the limitations of existing stream processing systems and build intuition for Jetstream's design, consider Figure 3(a), which shows the general architecture of a software stream processor used to analyze packets traces from across a network. The figure illustrates three main steps in the analytics process, each of which has a significant bottleneck that Jetstream eliminates.

First, the frontend of the stream processor must handle load balancing and distribution: forwarding a copy of each packet to an instance of each application that needs to analyze it. Given the high event rates in network analytics, this task of deciding where each packet should go and load balancing across servers and processor cores is expensive in software, and can easily bottleneck the whole system.

Second, in the application-specific analytics pipelines, sequences of operators transform input packet streams into streams of meaningful information (e.g., metrics or alerts). In these pipelines there are many sources of overhead that cumulatively reduce throughput by an order of magnitude. For example, copy and locking operations in inter-operator queues, pointer-chasing in container-based key-value data structures, and serialization overheads in message-passing subsystems. As Section V-A explains in more depth, for many network analytics tasks frequent message passing and lookup operations are required, making general-purpose stream processor overheads impact network analytics tasks significantly.

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 aggregation node — a third bottleneck.

Takeaway: All of these systems have benefits over traditional solutions (e.g., traffic mirroring or flow monitoring), but introduce compromises. Further, while some use programmable network hardware, all still rely, to varying degrees, on software for final metric computation and are therefore subject to the above mentioned bottlenecks. As a result, even for state-of-the-art telemetry systems, Jetstream's capability to support high-throughput and general analytics in software is essential for meeting novel, packet-level monitoring requirements in cloud-scale networks.

Further, software processing as it is possible in Jetstream enables applications written in a general-purpose language and does not limit the complexity of analysis or require sacrificing accuracy to gain performance. Instead, applications can fully leverage the flexibility of general-purpose hardware with ample memory and processing resources to implement complex analytics using, e.g., neural networks, sophisticated stateful logic, or third-party libraries.

As we describe next, Jetstream achieves these goals through a combination of system-level hardware offload and software optimization, which eliminate the bottlenecks outlined above to enable high-performance network analytics in software.

III. Introducing Jetstream

Jetstream is a high-performance network analytics system that makes no compromises on generality or performance of analytics tasks. It lets applications perform packet-level analytics, including the calculation of arbitrary metrics, entirely in software and scales linearly with server resources. To overcome the issues observed in Section II, we follow two main design strategies.

First, as Figure 3(b) illustrates, we move distribution and load balancing functionality into network switches. We call this *analytics-aware network telemetry*. We also push aggregation of computed, metric streams to external backend systems. At the core of our system then remain independent stream processing pipelines that are primarily bottlenecked by computational, input/output and data structure overheads. The second design strategy is to optimize these overheads away using a collection of techniques drawn from prior work but adapted for packet analytics workloads.

A. Using Jetstream

Jetstream is designed to run user-defined applications on records for every packet in a network. These applications are written in a general-purpose language (here C++) and can use a highly optimized set of common network-oriented stream processing operators that are part of the Jetstream library. In addition to using this standard library, a user can implement operators with entirely customized logic that still benefit from Jetstream's system-level optimizations.

Typical applications implement, for example, header-based intrusion detection [41] or performance monitoring applications, such as a queue depth monitor based on telemetry data from data plane hardware [37]. Common across all applications is the broader goal of network analytics, that is making the vast amount of records exported from network devices comprehensible and useful for the operator. This means that Jetstream applications must perform significant event rate reduction through (application-specific) data aggregation. As a result, the output data of a typical Jetstream application is again a (much lower frequency) event stream of applications-specific data tuples (metrics) for further, sometimes interactive, analysis [40], [42], visualization, network control [5], [43], or archiving [42] in a backend system. As a proof of concept we demonstrate the integration with a time series database system (Prometheus [44]) as one possible backend.

B. Analytics-Aware Network Telemetry

Switches are the source of network traffic data (i.e., packet headers or records), as prior network measurement systems [2], [3], [8], [9], [34], [45], [46] have observed. Offloading network analytics tasks directly to line-rate PFEs on network switches is therefore appealing but comes with previously explained drawbacks (Section II-C). Unlike prior systems, Jetstream does not push any application specific logic down to the switch level. Instead, we leverage programmable data plane technology for offloading functionality common across all applications, such as compressing, distributing, and load balancing telemetry data streams.

Jetstream's data plane frontend builds on *Flow [37], an existing high-performance network telemetry platform that exports digests containing per-packet measurements. We elaborate on how we extend and make *Flow *analytics-aware* by implementing application-specific, runtime-configurable distribution and load balancing of telemetry streams in Section IV.

C. Highly-Parallel Streaming Analytics

The streaming analytics engine performs the vast majority of analytical computations touching on every single exported packet in software. This is the core component of Jetstream, supporting custom applications implemented as stream processing programs.

In the stream processing paradigm, an application is a graph (or *pipeline*) that is organized in several stages. Each stage performs one computational task and is implemented using one or many parallel *kernels* (or *operators*) that transform (e.g., map, filter, or reduce) an unbounded stream of tuples [47]–[50]. In traditional stream processing, applications scale at the stage-level. Each operator typically runs in a separate thread and maps to a physical processor core. This model is a clean and simple abstraction for data processing applications, but presents two main challenges.

First, it requires load balancing between kernels in software, which introduces bottlenecks described in Section II-D. We overcome this challenge by scaling at the granularity of full pipelines. An application consists of multiple independent pipelines that each handle a distinct subset of flows. Jetstream's data plane component partitions packet records between these pipelines and encapsulates each record in a

UDP packet. The UDP destination port encodes the application instance selected to process the packet. At the analytics server, the NIC uses the port number to select the appropriate hardware queue for each packet; each Jetstream pipeline then only ever reads from its assigned queue.

Second, stream processing platforms add communication and data structure overheads. We address this challenge by carefully applying a set of software optimizations that are adapted and tuned for packet-level network analytics workloads. These optimizations have the goal to minimize the amount of costly copy operations, improve data locality within processing pipelines and amortize remaining, inevitable cost by using batching. We elaborate on the unique characteristics of packet analytics workloads and their impact on our design and optimization choices in Section V.

D. On-Demand Metric Aggregation and Analysis in Backend Systems

Finally, the results of stream processing pipelines, which will generally consist of high-level information at significantly lower rates, can be fed into different backend systems, such as security platforms as alerts [51], time series databases for visualization, auditing, offline analysis [52], or network controllers for automated network reconfiguration [7]. In order to mine meaningful and network-wide metrics and analytics results, event streams must eventually be merged and aggregated across analytics pipelines and servers. As explained before, this is costly when done within the stream processor and at rates of millions of records per second but becomes feasible when performed on event streams of hundreds or even thousands of records per second and outside of the critical analytics pipelines.

Consequently, to maintain pipeline independent processing, we push cross-pipeline data aggregation into the backend itself. As a proof of concept, we use a time series database system which is already optimized to aggregate data from many sources. Each metric calculation pipeline streams data directly into a database proxy, which exposes per-instance flow metrics through an interface that the database scrapes. We show that our model fits existing time series database systems well and dive into each phase of the on-demand aggregation and analysis part of our system in Section VII.

IV. Analytics-Aware Network Telemetry

The Jetstream data plane interface connects line-rate telemetry systems with the Jetstream analytics processing servers. As we view compression as an important system-level functionality to support, we build our prototype with concepts taken from *Flow [37], which emits *grouped packet vectors* (GPVs). A GPV is simply a variable-length list of packet features grouped by flow for more efficient processing with software. One can think of GPVs as a deduplication-based compression format for packet records. In an evaluation of a wide-area Internet packet trace collected by CAIDA [53], using GPVs results in a 7.7x reduction in bandwidth over packet records (which already provide significant compression over full packet traces).

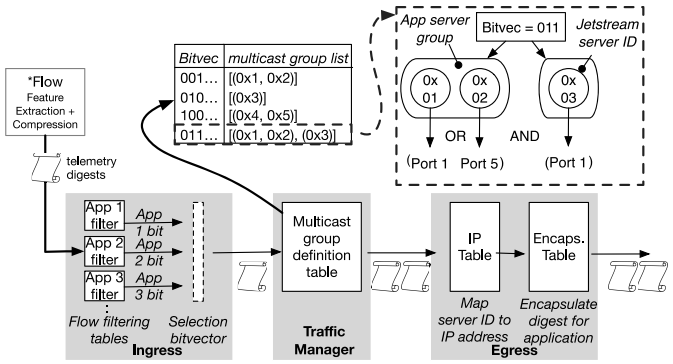


Fig. 4. Jetstream's data plane frontend for filtering, replication, and load balancing of telemetry digests written in P4.

Jetstream's data plane component, illustrated in Figure 4, extends *Flow to distribute and load balance 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 first bottleneck of general software stream processing for network analytics and allows application pipelines to operate entirely in parallel.

We leverage programmable switches (e.g., in our case the Barefoot Tofino [36]) to support three important functions at line rate: *replicating* streams of telemetry digests 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. We implemented these three functions on top of the feature extraction and compression functionality of *Flow.

Usage: The abstraction for the Jetstream data plane interface is simple and application-centric. Each application configures match+action tables used by the Jetstream P4 [29] program to set the IP addresses of its Jetstream processing servers. The switch will load balance telemetry digests destined for the application across these servers, based on a key. The key can be configured per application and is generally the IP 5-tuple or a subset of it. For example, for an application that only computes statistics per destination IP address, using only the destination IP address as the key means that packets with a particular destination IP address would always end up at the same Jetstream pipeline eliminating the need for later data aggregation. Each application also configures a dedicated filtering table that specifies which flows it needs to monitor. Only telemetry digests matching the filtering table are cloned to the application's servers. The filtering tables can either use exact or ternary matching over the flow key. A new application is added and configured at runtime by populating entries in the respective match+action tables using the RPC mechanisms exposed by the data plane target [54]. This means that adding, scaling or removing a Jetstream application does not require reloading the data plane as it is required in existing systems [8] incurring switch downtimes of tens of seconds [37].

Design: As illustrated by Figure 4, the Jetstream data plane interface is implemented as a sequence of match+action tables in the ingress, multicast engine, and egress stages of

a programmable switch. The input is a stream of telemetry digests from *Flow or any other data plane telemetry system. During the ingress stage, Jetstream applies a set of parallel match+action tables to determine which set of applications need to process each digest, based on its flow key. Each table holds the filtering policy of one application and sets a single bit in a *flow selection bitvector* packet metadata field, i.e., `bitvec[2] == 1` means that the third application needs a copy of the current digest.

After ingress, the digest and flow selection bitvector proceed to the switch traffic manager. The traffic manager (TM) uses the bitvector as a reference into its multicast configuration table. For modern switches, e.g., the Barefoot Tofino, each entry in this table maps a multicast ID to a set of multicast groups. As Figure 4 shows, Jetstream configures this tree structure so that each group represents the servers where a specific Jetstream application runs. The TM selects one member of each group using a hash of the load balancing key, clones the digest to the associated port, and annotates the packet with the ID of the selected member. Each ID is a 16-bit value, which we configure to encode the ID of a specific analytics server. Finally, in the egress pipeline, the switch encapsulates each replica of the digest. To determine the destination IP address, it uses a table that maps the Jetstream server ID to an IP address.

While we use *Flow as the underlying telemetry system, it is important to note, that Jetstream's data plane frontend is flexible and can be used with any data plane based telemetry source. For example, previous systems, such as Marple [8] and Sonata [9] can be integrated as telemetry sources and subsequently highly benefit from Jetstream's software processing performance and capabilities.

V. High-Performance Stream Processing of Network Records

The Jetstream data plane frontend sends telemetry records directly to the individual stream processing pipelines of one or more applications. This allows the pipelines to avoid interaction for distributing network records in software (i.e., the first bottleneck in Section II-D) 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 describe how we can leverage them to reduce communication and data structure overheads.

A. Packet Analytics Workloads

We identify six key differences between network packet analytics workloads and those of general stream processing.

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 that their stream processing cluster handles up to 46 million events per second [55], [56]. For comparison, the aggregate rate of packets leaving their cache network is over 320 million 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 (metrics) derived from packet headers and processing metadata, which are only a small portion of each packet. A 40 Byte 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 (e.g., by connection) before applying heavyweight data mining or visualization algorithms. This is not true and applicable for general stream processing workloads, where the backend algorithm may operate on features derived from each record.

Simple, Well-formed Records: Packet analytics records are also simple and well-formed. Each packet record has the same size and contains the same fields that can be accessed in constant time when in memory. 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, free-form text, and are encoded in serialization formats such as JSON and protocol buffers that require more complex parsing.

Network Attached Input: Data for packet analytics comes from one source: the network. Be it a router, switch, or middlebox that exports records, they will ultimately arrive in 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 that are relevant to many different applications, for example, using the flow key (e.g., IP 5-tuple) for load balancing. 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.

B. Jetstream Optimizations for Packet Analytics Workloads

Based on the observations about packet analytic workloads, we identified five important components of stream processing systems where we apply optimizations in Jetstream. We measure the benefit of these optimizations in Section VIII-A.

Data Input: In general-purpose stream processing systems, data can be read from many sources, such as a HTTP API, a message queue system (e.g., RabbitMQ or Kafka), or a specialized file system like HDFS. 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 [57], PF_RING [58], or netmap [59], to reduce overhead by mapping the packet records directly to buffers in the stream processing system. Jetstream uses netmap [59] to read packet records from individual NIC queues directly into the stream processor without introducing overheads from the operating system networking stack.

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, Jetstream can eliminate this overhead by pre-allocating buffers and passing pointers between kernels, to significantly improve performance. In Jetstream, for the output of kernels that do not alter the record data structure (e.g., filter operations), we amortize data copy overheads by using pointers together with C++ move semantics [60] that allow the compiler to avoid deep copies.

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 as primary bottlenecks in queue implementations. For example, basic concurrent queue implementations use expensive locks to ensure thread safety and use linked lists as their underlying data structure. Linked lists allow automatic resizing of the buffer but are expensive due to poor cache locality and frequent pointer dereferencing. 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. Based on this insight and leveraging several techniques used in concurrent data structures [61], we implemented an efficient, lock-free ring buffer. As records are small, we use a flat memory layout to avoid overheads of frequent pointer dereferencing. This means that the entire ring buffer is allocated as a single fixed size array and the array cells hold the actual data tuples as opposed to pointers to the data. The size of the ring buffer is also locked to powers of two allowing for cheaper bit shift operations instead of modulo operations to calculate offsets [62]. Finally, we use atomic types for head and tail indices to enable thread-safety [60].

Hash Tables: Often, network analytics applications need to map packet records to prior state. This requires a key-value store, which can easily be a bottleneck when processing high-rate packet streams. As a solution, Jetstream's library provides an optimized data structure that exploits the fact that packet records are small, well-formed, and have fixed width fields. The reduce operator and a flow table component that are commonly used by network analytics applications and are part of the Jetstream library use a hash table with a flat memory layout and open addressing with linear probing to reduce the overhead of pointer dereferencing and increase cache hit rates. Additionally, to minimize the cost of key comparisons during lookups, Jetstream's hash table encodes keys using 128-bit integers so that they can be compared using a single *Streaming SIMD (SSE)* vector instruction [63], [64].

Batching: Finally, the small size of individual network records makes batching appealing and improves 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 Jetstream's *Flow-based telemetry data plane, amortizes the cost of hash table operations necessary for a wide range of aggregation tasks that use the flow key or a subset of it as the aggregation key.

VI. Programmability and Applications

Jetstream analytics applications are written in C++, a popular, general-purpose language enabling easy prototyping, testing, and deployment. Applications leverage the Jetstream library of optimized stream processing primitives. This library not only includes the stream processing core and runtime, 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.

A. Input/Output and Record Format

As Jetstream's telemetry frontend extends a prior telemetry system, *Flow, we leverage *Flow's record model, 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 ids, queue depths, IP ids, and TCP sequence numbers. Further information on the GPV format and GPV generation in both software and hardware can be found in [37].

The primary packet input mechanism in our system leverages netmap [59], 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 or the TaZmen sniffer protocol.

B. Programming Model

Jetstream applications are written as stream processing pipelines. The simplest way for a developer to write an application is by composing Jetstream's builtin stream processors, for example those listed in Table I. Table II shows Jetstream's API for interconnecting these processors and launching pipelines. A simple application counting the number of packets per source IP address can be defined like this:

```
js::app a;
auto rx = a.add_stage<js::gpv_receiver>("enp2s0f0");
auto map
  = a.add_stage<js::map<gpv,pair<js::ipv4_addr,unsigned>>
    ([](gpv x){return std::make_pair(x.ipsrc,x.pktcount);});
auto reduce
  = a.add_stage<js::reduce<js::ipv4_addr,unsigned>>(plus());
a.connect<gpv>(rx,map);
a.connect<pair<js::ipv4_addr,unsigned>>(map,reduce);
a();
```

Here, `js::app a;` declares and instantiates a pipeline (or application). Calls to `a.add_stage()` and `a.connect()` define the pipeline, and its execution begins on the last line when we call the function operator (`a()`). Using this API, each application defines the processing steps it requires.

Jetstream includes a standard library (short `js`) of common processors that can be chained to build full network

TABLE I
PROCESSORS IN THE JETSTREAM STANDARD LIBRARY (NAMESPACE PREFIXES *js* AND *std* ARE OMITTED)

Processor	Parameters	Output	Description
filter<*In*>	function<bool(*In*&)> *p*	*In*	Filter out records of type *In* that do not satisfy boolean predicate *p*.
gpv_receiver	string iface_name	gpv	Receive GPVs from network interface *iface_name*.
join<*In1,In2,Out*>	function<bool(*In1*&, *In2*&) *c*, function<*Out*(*In1*&, *In2*&) *m*	*Out*	Joins two streams on matching condition *c* emitting tuples of new type *Out* assembled by *m*.
map<*In,Out*>	function<*Out*(*In*&)> *f*	*Out*	Apply function *f* to inputs of type *In*, emitting tuples of type *Out*.
print<*In*>	ostream& os	void	Print a summary of type *In* to a C++ output stream os.
reduce<*K,V*>	function<*V*(*V*&,*V*&)> *r*	pair<*K,V*>	Reduce values of type *V* grouped by keys of type *K* using reducing function *r* (e.g., std::plus to sum values by key).

TABLE II
API FOR COMPOSING AND RUNNING APPLICATIONS

Function	Description
a.add_stage<*P*>(*args*...)	Add stage with processors of type *P* to application *a*; initialize the processor with arguments *args*....
a.connect<*T*>(*s1*,*s2*)	Connect stage *s1* emitting type *T* with stage *s2* in the processing graph of application *a*.
a()	Run application *a*

analytics applications. The library includes common data flow operations listed in Table I. Additionally, specialized domain-specific operators exist to, for example, reduce by flow key (i.e., a flow table). All processors in the Jetstream library leverage different software optimizations outlined in Section V-B.

C. Custom Processors

If an analytics task requires processing logic, data types, or interfaces that are not covered by Jetstream's library, developers can implement custom processors that automatically take advantage of Jetstream's scaling and load balancing.

To write a custom processor, a developer first creates a subclass of `js::proc`. Next, the developer specifies input and output ports and types in the subclass's constructor. These ports are used to send or receive records to or from other processors, respectively. Finally, the developer implements processing logic in the `operator()` method. For example, a basic version of the *print* processor from Table I can be implemented like this:

```
class print : public js::proc {
public:
   print() { add_in_port<gpv>(0); }
   bool operator()() {
     gpv gpv; js::signal sig;
     in_port<gpv>(0)->dequeue_wait(gpv, sig);
     _os << gpv << std::endl;
     return sig == sig::proceed; }
private: std::ostream& _os; };
```

Custom processors allow developers to implement arbitrary applications that operate on packet records or GPV inputs. They are written as standard C++ code and can use custom algorithms and data structures, leverage third party libraries, or call external services.

VII. ON-DEMAND AGGREGATION IN BACKEND SYSTEMS

Processing pipelines in Jetstream are optimized to efficiently extract higher-level information from the input data packet stream. We refer to this higher-level information as *metrics*. In our prototype implementation such metric tuples consist of a numeric value, a timestamp, and a set of key-value metadata pairs. For example, to detect elephant flows, a heavy hitter detector implemented on top of Jetstream would periodically export the number of packets or bytes together with flow information (i.e., the IP 5-tuple) for the most active flows [65].

A. Integrating With Backend Systems

In Jetstream, the final aggregation of computed metrics is offloaded to configurable backend systems. This is possible as long as the analytics application already significantly (i.e., by several orders of magnitude) reduces the event rate. Intuitively, this is the common case for analytics applications because useful metrics aggregate data (e.g., in small time intervals), or report on anomalies that are by definition infrequent.

The backend can then be used to automatically or interactively query, analyze, or visualize metric data computed by Jetstream. Jetstream integrates with backend systems through an API that can be used by applications to export metrics from pipelines. A local metric collection proxy consumes app metrics and exposes an interface that can subsequently be polled by the backend system. The export API used within Jetstream applications is universal while the API exposed by the collection proxy is specific to the respective backend system. We imagine possible backend systems to be time series databases (e.g., Prometheus [44]), visualization systems (e.g., Grafana [66]), monitoring platforms (e.g., Nagios [67]), another stream processor, or a network control platform (e.g., ONOS [68]) to enable a network control loop.

Exporting Metrics: The metrics export API currently supports two types of metrics inspired by the Prometheus time series database: a *counter* and a *gauge*. A counter metric represents a cumulative and monotonically increasing value while a gauge can be set to a specific value, increased, or decreased in value. Each metric is associated with a name, a timestamp, and a set of meta data. Other metric types, such as snapshots of full metric distributions or vectors are imaginable. For example, upon detection, reporting a heavy hitter using a counter from within a Jetstream pipeline looks like this:

```
js::metrics.update_counter("heavy_hitters", hh.pkt_count,
    {{"ip_src", hh.ip_src}, ... });
```

Collection Proxy: Internally, the metrics export API adds a timestamp, serializes the metric object using Protocol Buffers [69], and sends a RPC message using gRPC [70] to the collection proxy. The collection proxy sits between a Jetstream application and the backend system, converting

data into the appropriate format. In order to prevent data aggregation in-line resulting in cross-core communication, a collection proxy is instantiated for each instance of a Jetstream application and subscribes to an instance's metric stream. We built a collection proxy prototype for the Prometheus time series database [44]. For this integration, the proxy exposes a HTTP API that a Prometheus instance periodically scrapes. Finally, Prometheus stores scraped metrics in its data store for continuous aggregation across Jetstream instances.

B. Querying Metrics

Prometheus supplies a query language and API, which allows a user to retrieve network traffic metrics from the database. Additionally, Prometheus allows configuring alerts and integrates with Grafana [66], a framework to easily visualize query results to, for example, build dashboards. All our example applications integrate with the metrics export system and can be queried from Prometheus. We now show example queries for three of those applications to illustrate how a user can interact with and extract relevant metrics from Jetstream.

For the traffic accounting application, Prometheus maintains individual counters for each component of a packet's IP 5-tuple. For example, a user can use the Prometheus `rate()` function to calculate the average number of bytes per second sourcing from port 443 over the last minute using this query:

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

The heavy hitter application, which looks for flows that cause more than a configurable percentage of the total bytes in the network, exports heavy hitter candidates with the metric name `heavy_hitters`. In order to identify the top 5 frequent flows from the candidates stored in the database, we can issue a query as follows:

```
topk(5, heavy_hitters)
```

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. To find which flows originating from the 192.168.0.0/16 subnet have more than 10 out-of-order packets, the user can issue the following query:

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

VIII. Evaluation

We evaluate the performance and efficacy of our prototype implementation through three different lenses. First, we measure Jetstream's overall system throughput and scalability from both an end-to-end standpoint as well as from an individual application throughput standpoint. We then look at how Jetstream's telemetry-aware data plane component compares with Sonata [9] in terms of PFE resource consumption and accuracy. Finally, we evaluate the performance of Jetstream's stream processor against both Spark [47] and dShark [10].

We used the Cloudlab network experimentation platform [71] for all of our benchmarks. Our experiment setup consisted of six servers with 2 × 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 over a 10Gbps network with two Intel X520 Ethernet adapters per server for ingestion of telemetry data. We used packet traces from a core Internet link collected by CAIDA in 2015 [53] for all experiments.

TABLE III
Jetstream's Per-Application Throughput [M pkts/s]. Two Cores Per Application

Application	Mean	Description
Passthrough	31.8	Simple GPV passthtough (no ops.)
Traffic Count	14.0	Count total GPVs, pkts, bytes
Heavy Hitter	16.2	Find IPs sending >θ% of total packets in network
TCP Seq.	15.0	Find TCP flows with out of order packets
Slowloris	14.7	Find IPs w/ many low traffic volume TCP connections
SLB Profiler	31.6	Software load balancer
SSH Brute Force	10.8	Identify SSH Brute Force attacks
SYN Flood	21.1	Identify SYN Flood DoS attacks

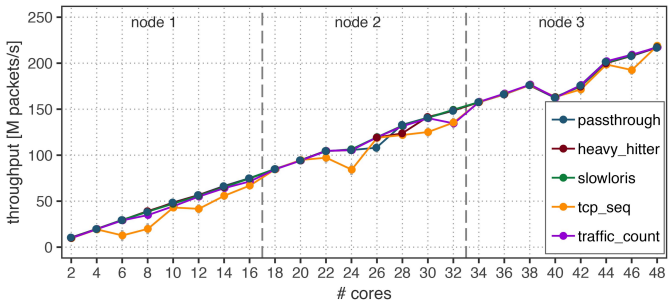


Fig. 5. Scalability of Jetstream applications across servers.

A. Macro Benchmarks

First, we benchmark Jetstream's performance and scalability at a macro-level using the applications described in Table III. In this experiment, we created a scenario where three switches stream GPVs across the network to three Jetstream analytics servers running application pipelines. Our programmable switch (Tofino) is currently not physically co-located with sufficient server resources. We therefore model the switches by running a software implementation of Jetstream's data plane component on three separate servers in the Cloudlab network, driven by real-world packet traces from CAIDA. Each pipeline uses two cores scaling to a total of eight pipelines per server, or 24 pipelines using 48 cores across three servers. Each 10GbE network interface serves up to four Jetstream pipelines.

Figure 5 shows the performance and scalability of Jetstream. We ran 24 rounds of this experiment where we added an additional analytics pipeline (2 cores) with each round, eventually using all 3 × 16 cores of our servers. Our system scales linearly with core count across machines and can process over 200 million packet records per second leveraging only three commodity servers. This demonstrates the effectiveness of key design choices in Jetstream.

The bottleneck in this set of benchmarks was the 10Gbps network interface card we used. With the assumption that telemetry packets are roughly 200 bytes on average (since a

TABLE IV
JETSTREAM NETWORK INTERFACE RESOURCE USAGE ON THE BAREFOOT TOFINO. STATEFUL ALU USAGE IS 0 FOR ALL APPLICATIONS

# Apps	Stages	Tables	VLIWs	Metadata	SRAM	TCAM
1	2	5	3	888b	128KB	3.84KB
4	2	8	5	912b	128KB	15.36KB
8	2	12	7	944b	128KB	30.72KB
12	3	16	9	976b	128KB	46.08KB
16	3	20	11	1008b	240KB	61.44KB

packet is a GPV record), the max rate of a 10Gbps network interface would be about 6M GPVs/sec. We had each of the two NICs feed 4 of the pipelines (8 of the cores), which led to roughly 1.5M GPVs/sec per pipeline. With an average of 8 packets per GPV in the trace that we used, this translates to roughly 12M packet records per second per pipeline that we can theoretically feed per pipeline, or a maximum theoretical rate of 96 M packet records per second per server with two 10Gbps NICs. In practice this rate is likely lower due to a variety of factors. We saw roughly 75M packet records per second in practice of just I/O performance. As we will see next, many of our applications scale beyond this number and would therefore benefit from higher throughput NICs.

To show the performance of the individual Jetstream applications without the NIC input bottleneck in our setup, we also stream network traffic from memory through Jetstream. Again, in this experiment, each application is assigned two cores as each application has one thread dedicated to consuming records while the other thread runs the application. Table III shows Jetstream's application performance numbers. We can see that Jetstream achieves a maximum throughput in excess of 31 million packets per second per pipeline while also attaining strong performance for complex, stateful applications such as SSH Brute Force detection. As a result, Jetstream pipelines process data between 1.5 to 3 times faster than the 10Gbit/s telemetry input over the network. In practice, a 40 Gbit/s NIC would be able to fully utilize the analytics pipelines.

Jetstream's high processing rates are a result of applying the different software optimization strategies outlined in Section V-B. Using GPVs provided a speedup of 5.4 over single packet records. Our optimized concurrent queue implementation was faster by a factor of 3.0 over the C++ standard template library queue (secured with locks). Our hash table implementation using a flat layout and linear probing provided a speedup of 1.8 over the STL standard unordered map. Finally, using netmap instead of standard sockets provided a throughput increase of a factor of 2.8. To obtain these numbers each optimization was isolated from all others.

B. Comparison With Hardware Analytics

We next evaluate Jetstream's data plane component, a line-rate data plane program written in P4 that filters, replicates, and load balances telemetry digests across analytics servers. We ran this program on a Barefoot Tofino PFE configured with ternary application filtering tables sized for 128 entries each. Table IV lists the major resource requirements of the Jetstream data plane interface. Overall, the component is lightweight: It requires only 3 stages and 20 tables to filter for 16 different applications because of its parallel design. The most-utilized resource is TCAM. Each application's table uses approximately 1% of the Tofino's TCAM. If wildcard and priority-based filtering is not required for all applications, some or all of the tables can be replaced with exact match tables in SRAM rather than TCAM.

TABLE V
RESOURCE USAGE FOR HARDWARE ANALYTICS QUERIES ON THE BAREFOOT TOFINO. SRAM REQUIREMENT ASSUMES < 65K CONCURRENT KEYS (E.G., ONE 10 GB/S INTERNET LINK [53])

Query	Stages	Tables	VLIWs	Metadata	sALUs	SRAM
Heavy Hit.	5	13	7	912b	1	112KB
New Conn.	6	16	8	1032b	1	128KB
S. Spreader	8	19	9	840b	2	192KB
Port Scan	8	20	10	1072b	2	208KB
SSH Brute	9	26	11	984b	2	224KB
SYN Flood	11	25	17	1312b	2	288KB
Cmpl. Flows	11	26	17	1312b	2	304KB
Slowloris	11	27	17	1316b	3	336KB
With one level of refinement (Sonata)						
Heavy Hit.	7	22	11	1152b	2	224KB
New Conn.	9	28	13	1184b	2	256KB
Others	***Compilation failed, insufficient resources***					

We now compare Jetstream's PFE resource consumption with that of Sonata [9], a state-of-the-art network telemetry and analytics platform that leverages switch hardware to accelerate network analytics. Sonata's primary goal is to reduce the load on the software stream processor by iteratively refining network queries and pushing them into hardware.

While Sonata is able to reduce the event rate at the stream processor, the system makes tradeoffs to realize this performance. First, Sonata's iterative refinement reduces the required state maintained by the switch to execute a query. However, refinement comes at the cost of an increasing number of match+action tables to perform the same query. Table V illustrates this point, as many queries that run with multiple levels of refinement fail to compile to the switch. If we compare Sonata (Table V) and Jetstream's (Table IV) resource usage, we can see that Jetstream only requires about as many resources (stages, tables, etc.) as a single Sonata query in hardware, even to support expensive load balancing and filtering for many concurrent applications.

The second of Sonata's tradeoffs also stems from query refinement and results in a reduction in accuracy. Each iteration of refinement that reduces load on the stream processor, requires another time window to pass by before packets are forwarded to the stream processor. As a result, in order to get the largest reduction in event rate at the stream processor, applications must wait multiple time windows before being able to process potentially time-critical data. Waiting one or more time windows negatively impacts accuracy for many applications as issues lasting fewer than one or more time windows (e.g., frequent micro-bursts [72]) will not be detected. Jetstream has no such accuracy limitation since processing is done in software. Detection performance is predictable and attacks will not slide through the cracks.

Finally note that, while Jetstream provides a telemetry replacement for Sonata at a lower PFE resource cost, Sonata (or other telemetry systems) and Jetstream can technically be used in conjunction. This may be beneficial in certain cases,

e.g., when a simple, static query fits entirely in the PFE. Doing so, however, sacrifices flexibility. For example, it makes runtime reconfiguration more challenging (see Section II-C).

C. Comparison With Pure Software Analytics

In this article, we argue that software provides the programmability and flexibility needed to support a wide range of network analytics applications. Existing software analytics platforms, however, do not provide sufficient performance for cloud-scale network analytics workloads. To support our argument, we now compare the performance of our system against both Spark (used by Sonata [9]) and dShark [10]. For each test, we used the same experimental setup as described in Section VIII.

a) *General-purpose Analytics (unmodified Spark):* To illustrate the impact that just the architectural changes have, we compare against Spark [47], a general-purpose stream processing system. We ran the Traffic Accounting application, which counts the number of packets and bytes per each component of the IP 5-tuple. We streamed GPVs as input data over the network to both Spark and Jetstream. With two CPU cores, Spark sustains 1.4 million packet records per second, whereas Jetstream runs at 9.9 million packet records per second. Most importantly, we found that for this workload Spark (unlike Jetstream) does not scale with core count (or number of threads). Spark's inability to scale in this scenario is due to the high-volume input streams in network telemetry that Spark distributes across worker threads in software. This imposes very high utilization in the distribution/load balancing thread and subsequently creates a bottleneck. In Jetstream, this critical task is offloaded to programmable line rate switches. We gave more intuition on this in Section II-D. Other Spark users have also found Spark to scale poorly for comparable workloads [73], confirming our tests.

b) *General-purpose Analytics (Spark with kernel bypass):* Of course, a question arises if Jetstream's benefit just comes from its use of kernel bypass technology. As it is non-trivial to modify Spark to include streamlined network I/O capabilities, we use streaming from memory within the application as a way to remove the I/O component from the evaluation. That is, we read an entire trace into memory and replay it directly within the application. With 2 cores, Spark runs at 2.0 million packet records per second, whereas Jetstream runs at 14.0 million packet records per second, further illustrating Spark's architectural bottleneck.

c) *Network Analytics Software (dShark):* To understand Jetstream's true software processing performance in the face of the NIC bottlenecks we experienced, we compare against dShark [10], a recently introduced software-based, packet-level, network analytics platform. A key innovation of dShark was the ability to analyze traffic in the face of network packet header transformations. One such application which requires this functionality is the software load-balancer (SLB) profiler in dShark. We re-created the SLB profiler application in Jetstream and validated its correctness in a live test. Our results illustrate Jetstream's comparable flexibility to dShark. We acknowledge, however, that because Jetstream relies on GPVs, which are fixed-format records, we can only support a fixed depth of header nesting, whereas dShark can support any depth. We believe this limitation is not impactful for this discussion, as in practice, it would be highly irregular to see a depth of nesting beyond some known amount. Since dShark is not open source, we reference the performance results in their publication. While not a perfect comparison, our results are still illustrative with Jetstream running on similar hardware. In the dShark experiments, packet records are streamed from memory directly into the analytics application. On a 16-core server, dShark runs at 10.6 million packets per second (Mpps) with 6 parsers and 9 groupers (or 0.625 Mpps per core), whereas Jetstream runs at 31.6 Mpps (or 15.9 Mpps per core), a 25.44x speedup. Here, we note that performance scales linearly with number of servers in both cases.

d) *Resource Cost Analysis:* To put the performance speedups into context, consider the resources needed to support analytics in a modern datacenter. Here, we look at reported traffic in a cluster at Facebook [28] where an analytics system needs to sustain at least 961 million packets per second in order to meet the Web server cluster's peak packet rates. Assuming 16-core servers, we would need ~96 servers for each analytics application to run on dShark, ~480 servers for systems using Spark, and a mere 4 servers for systems using Jetstream. These numbers also assume that dShark and Spark integrate optimized packet input through, for example, kernel-bypass technology, as Jetstream does.

IX. Conclusion

This article introduced Jetstream, a high-performance platform for network analytics that makes no compromises on performance or generality — records of every packet can efficiently be processed in software. The core insight of Jetstream is to utilize programmable networking hardware to improve the performance of software analytics platforms, rather than offloading analytics applications themselves.

The resulting prototype of Jetstream can analyze between 86.4 and 254.4 million packets per second on a 16-core commodity server. Benchmarks show that Jetstream's approach to telemetry data distribution and load balancing in the data plane enables linear scaling with addition of servers and only requires moderate switch resources. Compared with a high-performance network analytics software system (dShark), Jetstream supports over 25.4x higher processing rates. To process a published data center workload, this would require 96 servers in dShark, but only 4 in Jetstream — making fully flexible software-based network analytics practical.

Acknowledgment

The authors would like to thank Stefan Schmid for feedback on drafts of this article.

References

[1] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An overview of IP flow-based intrusion detection," *IEEE Commun. Surveys Tuts.*, vol. 12, no. 3, pp. 343–356, 3rd Quart., 2010.

[2] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2014, pp. 71–85.

[3] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better NetFlow for data centers," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2016, pp. 311–324.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. 7th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2010, pp. 281–296.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. 7th ACM Conf. Emerg. Netw. Exp. Technol.*, 2011, p. 8.

[6] J. Rasley *et al.*, "Planck: Millisecond-scale monitoring and control for commodity networks," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2014, pp. 407–418.

[7] R. Hand, M. Ton, and E. Keller, "Active Security," in *Proc. 12th ACM Workshop Hot Topics Netw. (HotNets-XII)*, 2013, pp. 1–7.

[8] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2017, pp. 85–98.

[9] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven Streaming Network Telemetry," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2018, pp. 357–371.

[10] D. Yu *et al.*, "dShark: A general, easy to program and scalable framework for analyzing in-network packet traces," in *Proc. 16th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2019, pp. 207–220.

[11] Y. Zhu *et al.*, "Packet-level telemetry in large datacenter networks," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2015, pp. 479–491.

[12] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP flow information export (IPFIX) protocol for the exchange of flow information," IETF, RFC 7011, 2013. Accessed: Feb. 21, 2021. [Online]. Available: https://tools.ietf.org/html/rfc7011

[13] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2003, pp. 325–336.

[14] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 270–313, 2002.

[15] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani, "Fast monitoring of traffic subpopulations," in *Proc. 8th ACM Conf. Internet Meas. (IMC)*, 2008, pp. 257–270.

[16] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Proc. 3rd ACM Conf. Internet Meas. (IMC)*, 2003, pp. 234–247.

[17] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc. 10th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 29–42.

[18] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2016, pp. 101–114.

[19] A. McGregor, M. A. Hall, P. Lorier, and J. Brunskill, "Flow clustering using machine learning techniques," in *Proc. Passive Active Meas. (PAM)*, 2004, pp. 205–214.

[20] A. W. Moore and D. Zuev, "Internet traffic classification using Bayesian analysis techniques," in *Proc. ACM SIGMETRICS Conf. (SIGMETRICS)*, 2005, pp. 50–60.

[21] J. Park, H. R. Tyan, and C. C. J. Kuo, "Internet traffic classification for scalable QoS provision," in *Proc. IEEE Int. Conf. Multimedia Expo*, 2006, pp. 1221–1224.

[22] T. T. T. Nguyen and G. J. Armitage, "Training on multiple sub-flows to optimise the use of machine learning classifiers in real-world IP networks," in *Proc. 31st IEEE Conf. Local Comput. Netw. (LCN)*, 2006, pp. 369–376.

[23] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through simple statistical fingerprinting," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 1, pp. 5–16, 2007.

[24] T. Auld, A. W. Moore, and S. F. Gull, "Bayesian neural networks for Internet traffic classification," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 223–239, Jan. 2007.

[25] N. Williams, S. Zander, and G. Armitage, "A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 5, pp. 5–16, 2006.

[26] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield, "Class-of-service mapping for QoS: A statistical signature-based approach to IP traffic classification," in *Proc. 4th ACM Internet Meas. Conf. (IMC)*, 2004, pp. 135–148.

[27] *Sonata Source Code*. Accessed: Feb. 21, 2021. [Online]. Available: https://github.com/Sonata-Princeton/SONATA-DEV

[28] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2015, pp. 123–137.

[29] P. Bosshart *et al.*, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[30] O. Michel, J. Sonchack, E. Keller, and J. M. Smith, "Packet-level analytics in software without compromises," in *Proc. 10th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2018, pp. 1–7.

[31] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[32] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2013, pp. 99–110.

[33] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proc. 28th Annu. ACM Symp. Theory Comput. (STOC)*, 1996, pp. 20–29.

[34] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with SwitchPointer," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 453–466.

[35] A. Khandelwal, R. Agarwal, and I. Stoica, "Confluo: Distributed monitoring and diagnosis stack for high-speed networks," in *Proc. 16th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2019, pp. 421–436.

[36] Intel Corporation. *Tofino*. Accessed: Feb. 21, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html

[37] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated monitoring to concurrent and dynamic queries with *flow," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2018, pp. 823–835.

[38] T. T. T. Nguyen and G. Armitage, "A survey of techniques for Internet traffic classification using machine learning," *IEEE Commun. Surveys Tuts.*, vol. 10, no. 4, pp. 56–76, 4th Quart., 2008.

[39] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 67–82.

[40] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo, "Quantitative network monitoring with netqre," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2017, pp. 99–112.

[41] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "KitSune: An ensemble of autoencoders for online network intrusion detection," in *Proc. Netw. Distrib. Syst. Security Symp. (NDSS)*, 2018, pp. 1–15.

[42] O. Michel, J. Sonchack, E. Keller, and J. M. Smith, "PIQ: Persistent interactive queries for network security analytics," in *Proc. ACM Int. Workshop Security Softw. Defined Netw. Netw. Function Virtualization (SDN-NFV Security)*, 2019, pp. 17–22.

[43] N. Foster *et al.*, "Frenetic: A network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[44] *Prometheus*. Accessed: Feb. 21, 2021. [Online]. Available: https://prometheus.io/

[45] C. Kim, A. Sivaraman, N. Katta, and L. O. Wobker, "In-band network telemetry via programmable dataplanes," in *Proc. ACM SIGCOMM Demos*, 2015, pp. 22–28.

[46] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "TurboFlow: Information rich flow record generation on commodity switches," in *Proc. 13th EuroSys Conf. (EuroSys)*, 2018, pp. 1–16.

[47] M. Zaharia *et al.*, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[48] J. C. Beard, P. Li, and R. D. Chamberlain, "RaftLib: A C++ template library for high performance stream parallel processing," *Int. J. High Perform. Comput. Appl.*, vol. 31, no. 5, pp. 391–404, 2017.

[49] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin,, "StreamBox: Modern stream processing on a multicore machine," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2017, pp. 617–629.

[50] *Apache Storm*. Accessed: Feb. 21, 2021. [Online]. Available: https://storm.apache.org

[51] M. Roesch, "Snort—Lightweight intrusion detection for networks," in *Proc. 13th USENIX Conf. Syst. Admin. (LISA)*, 1999, pp. 229–238.

[52] *TimescaleDB*. Accessed: Feb. 21, 2021. [Online]. Available: https://www.timescale.com/
[53] *Trace Statistics for CAIDA Passive OC48 and OC192 Traces—2015-02-19*. Accessed: Feb. 21, 2021. [Online]. Available: https://www.caida.org/data/passive/trace_stats/
[54] *P4runtime*. Accessed: Feb. 21, 2021. [Online]. Available: https://p4.org/p4-runtime/
[55] Twitter. *The Infrastructure Behind Twitter-Scale*. Accessed: Feb. 21, 2021. [Online]. Available: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html
[56] Twitter. *Observability at Twitter—Technical Overview*. Accessed: Feb. 21, 2021. [Online]. Available: https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-i.html
[57] *Data Plane Development Kit*. Accessed: Feb. 21, 2021. [Online]. Available: https://dpdk.org
[58] Ntop. *PF_RING*. Accessed: Feb. 21, 2021. [Online]. Available: https://www.ntop.org/products/packet-capture/pf_ring/
[59] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2012, pp. 101–112.
[60] Standard C++ Foundation. *C++11 Language Extensions—General Features*. Accessed: Feb. 21, 2021. [Online]. Available: https://isocpp.org/wiki/faq/cpp11-language
[61] A. Williams, *C++ Concurrency in Action*, 1st ed. Shelter Island, NY, USA: Manning, 2012.
[62] J. Yliluoma. *Bit Mathematics Cookbook*. Accessed: Feb. 21, 2021. [Online]. Available: https://bisqwit.iki.fi/story/howto/bitmath/
[63] *128bit Hash Comparison With SSE*. Accessed: Feb. 21, 2021. [Online]. Available: https://stackoverflow.com/questions/4534203/128bit-hash-comparison-with-sse
[64] Intel Corporation. *Intrinsics Guide*. Accessed: Feb. 21, 2021. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide
[65] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Trans. Database Syst.*, vol. 28, no. 1, pp. 51–55, 2003.
[66] *Grafana*. Accessed: Feb. 21, 2021. [Online]. Available: https://grafana.com/
[67] *Nagios Monitoring*. Accessed: Feb. 21, 2021. [Online]. Available: https://www.nagios.com/
[68] *ONOS*. Accessed: Feb. 21, 2021. [Online]. Available: https://www.opennetworking.org/onos/
[69] *Protocol Buffers*. Accessed: Feb. 21, 2021. [Online]. Available: https://developers.google.com/protocol-buffers/
[70] *gRPC*. Accessed: Feb. 21, 2021. [Online]. Available: https://grpc.io/
[71] *CloudLab*. Accessed: Feb. 21, 2021. [Online]. Available: https://www.cloudlab.us
[72] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo, "Micro-burst in data centers: Observations, analysis, and mitigations," in *Proc. 26th IEEE Int. Conf. Netw. Protocols (ICNP)*, 2018, pp. 88–98.
[73] *Spark: Inconsistent Performance Number in Scaling Number of Cores*. Accessed: Feb. 21, 2021. [Online]. Available: https://stackoverflow.com/questions/41090127/spark-inconsistent-performance-number-in-scaling-number-of-cores

Oliver Michel received the Ph.D. degree from the University of Colorado Boulder in 2019. He is currently a Postdoctoral Researcher with the Communication Technologies Group, University of Vienna. His research focuses on data plane programmability and architectures for scalable network telemetry and analytics.

John Sonchack received the Ph.D. degree from the University of Pennsylvania in 2020. He is currently a Postdoctoral Researcher with Princeton University. His research focuses on language and system design in hardware-accelerated network data planes.

Greg Cusack received the bachelor's degree from Santa Clara University in 2016. He is currently pursuing the Ph.D. degree with the University of Colorado Boulder advised by E. Keller.

Maziyar Nazari received the bachelor's degree from the University of Tehran in 2018. He is currently pursuing the Ph.D. degree with the University of Colorado Boulder advised by E. Keller. He also serves as a Research Assistant and is mainly doing research in the area of networked systems, virtualization, and cloud computing.

Eric Keller received the Ph.D. degree from Princeton University in 2011. He is currently an Associate Professor with the Electrical and Energy Engineering Department, University of Colorado Boulder. His research has been enabling and capitalizing on a more dynamic and programmable computing and network infrastructure, via such technologies as virtualization, software-defined networking, and the movement toward cloud-based services.

Jonathan M. Smith (Fellow, IEEE) received the Ph.D. degree from Columbia University. He is currently a Program Manager with the Information Innovation Office (I2O), Defense Advanced Research Projects Agency (DARPA) on leave from the University of Pennsylvania, where he holds the Olga and Alberico Pompa Professorship of Engineering and Applied Science and a Professor of Computer and Information Science. He was previously a Member of Technical Staff with Bell Telephone Laboratories and Bell Communications Research, joining Penn in 1989. He also served as a Program Manager with DARPA from 2004 to 2006, and was awarded the Office of the Secretary of Defense Medal for Exceptional Public Service in 2006.