Packet-Level Analytics in Software without Compromises

Oliver Michel
University of Colorado Boulder

John Sonchack
University of Pennsylvania

Eric Keller
University of Colorado Boulder

Jonathan M. Smith
University of Pennsylvania

Abstract

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 paper, we provide a first step towards 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 which enable the specialization of stream processing systems. As a result, an evaluation with our preliminary implementation shows that we can scale up to several million packets per second per core and together with load balancing and further optimizations, the vision of cloud-scale per-packet network analytics is possible.

1 Introduction

Network monitoring and analytics are crucial for the reliable and secure operation of cloud-scale networking environments. A monitoring system captures information about traffic. An analytics system processes that information to detect equipment failures, incorrect behavior (*e.g.,* packet drops or routing loops), and attacks, as well as to identify and locate performance issues within the infrastructure [34, 23, 21, 37, 15, 20]. In order to reduce the load on the analytics systems, compromises have been made in the monitoring systems. Aggregation (*e.g,* pre-calculating information about entire flows, and aggregating into a flow record) and sampling (only looking at a subset of the flows or packets), have been the de-facto standards for decades as the software systems just could not process the information fast enough. These approaches, of course, reduce information – 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 [3, 16]. Sampling and filtering reduces the number of packets or flows to be analyzed. Reducing information reduces load, but it also increases the chance of missing critical information, and restricts the set of possible applications [30, 28].

Recent advances in software-defined networking (SDN) and more programmable hardware have provided opportunities for more fine-grained monitoring, towards packet-level network analytics. Packet-level analytics systems provide the benefit of complete insight into the network and open up opportunities for applications that require per-packet data in the network [37]. But, compromises are still being made.

For example, [34, 23, 21] leverage processing primitives in hardware for custom aggregation (as opposed to fixed aggregation in legacy hardware). Although custom, the aggregation is still limited to a set of pre-defined functions that are implementable in programmable forwarding engines (PFE) [5, 10], and thus imposes restrictions on possible applications (*e.g.,* [9, 19, 24, 31]). These new programmable data planes have also been used for more granular and iteratively refined filtering in hardware [37, 15]. This, however, is only effective at reducing load if the application is only interested in a small subset of the overall traffic (*e.g.,* only DNS packets).

In summary, monitoring systems are historically designed to reduce the load of the software analytics platform based on the assumption that it cannot handle the load.

In this paper we challenge that assumption. We ask: *What are the limits of software analytics systems – is it possible to perform packet-level analytics on cloud scale infrastructures in software?*

We start with the assumption that hardware can send per-packet records [29] to the software-based analytics

system. The challenge is then processing those records as (i) modern networks run at traffic rates of several terabits of data per second, which can equate to hundreds of millions of packets per second [27], and (ii) modern analytics systems, specifically stream processing systems (which is an ideal programming model for network analytics), such as Apache Flink [6], or Kafka [7], simply are not capable of handling these rates. Even more recent work, such as StreamBox, a stream processor designed for efficiency [22], can only support around 30M records/s with a 56 core server for a basic example application unrelated to network processing. In modern networks, where even 10 Gb/s links have packet arrival rates of 500K packets per second [4], real time packet analytics with such general purpose systems would require racks full of servers.

Our insight is that stream processing for network analytics has fundamentally different characteristics than traditional uses of stream processing. In this paper we highlight some of these differences and show how they can impact the achievable processing rates (Section 3). As a step towards validating our vision, we outline an architecture (Section 4), and built a prototype with some preliminary optimizations (Section 5). With this, our evaluations indicate more than two orders of magnitude speedup (163×) over state-of-the-art solutions [15, 37, 22] (Section 6). Based on data from Facebook [27], we demonstrate that for an entire cluster in a Facebook data center our prototype can analyze every packet leaving the cluster on a single commodity server (representing only 0.5% of the cluster's hardware). As a result, we believe that powerful packet-level analytics in software at cloud scale (without compromise) is indeed within reach.

2 Stream Processing Overview

Stream processing is a paradigm for parallel data processing applications. Stream processors are composed of sequential data processing functions, that are connected by FIFO queues transferring the streams of data. The benefit of stream processing is that it compartmentalizes 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. Recently, with the increasing focus on real time and big data analysis, there have been many new streaming platforms that focus on efficient scaling and ease of use [7, 6, 35].

3 Stream Processing for Packet Records

Stream processing is widely used in many domains because it is a flexible and scalable way to process unbounded streams of records in real-time. These properties also make stream processing a natural fit for realizing the goal of a software platform capable of flexible analytics with visibility into every packet.

As explained in the introduction, using general purpose stream processors for packet analytics, however, is not efficient enough to be practical. The root cause of these issues are the differences between the workloads for packet analytics and those typical to stream processing. The differences present challenges, but also introduce new opportunities for optimization. By taking advantage of them, we can specialize stream processing systems for packet analytics to build systems that have not only the flexibility and scalability benefits inherent to the processing model, but also the efficiency to make software visibility into every packet practical at large scales.

In this section, we explore some of the distinct characteristics of packet analytics workloads and how we can optimize stream processors for them.

3.1 Packet Analytics Workloads

We illustrate the differences between a packet analytics workload and that of a typical stream processing task with a case study based on Twitter's well-documented use of stream processing [32, 33].

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 systems handle up to 46 M events per second. For comparison, the aggregate rate of packets leaving one small component of their platform is over 320 M per second. This network (the "Twitter Cache"), only represents 3% of the total infrastructure.

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

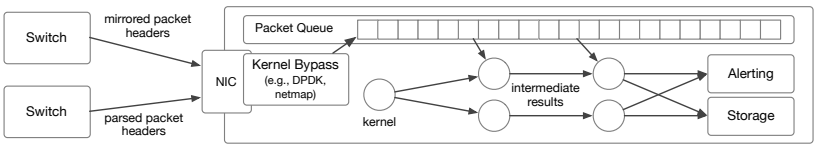

Figure 1: System Architecture

operate on features derived from each record.

Simple, Well Formed Records Packet analytics records are also simple and well formed. Each record has 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, they represent complex objects, *e.g.*, web pages, and are encoded in serialization formats such as JSON and ProtoBuf.

Common Preprocessing As a result of the standardized record formats, packet analytics tasks often rely on the same types of preprocessing logic, *e.g.*, grouping packets by source IP is the first step in many different applications. In typical stream processing workloads, the variability of the input types means that each application is more likely to require different, custom types of preprocessing.

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 NIC. 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.

3.2 Bottlenecks in Stream Processors

We identified five important components of stream processing systems where there is significant potential for optimization to account for the workload characteristics described above.

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 or Kafka), or from 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 [2], PF_RING [25], or netmap [26], to reduce overhead by mapping the packet records directly to buffers in the stream processing system. Benchmarks have shown that DPDK, for example, can process up to 48 million packets per second in user space [17]. Depending on the exact application and network interface cards, these numbers can be even higher.

Zero-Copy Message Passing Through our initial experiments we have identified that for most applications the performance of a single kernel is I/O-bound. Specifically, frequent read, write, and copy operations into the queues connecting kernels introduce significant performance penalties. This is true especially in early stages of the processing graph, where the data rate is still high as no aggregation has yet occurred. 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 kernels, to significantly improve performance.

(Off)load Balancing Load balancing is critical in stream processing, but sometimes the load balancing kernel itself can be a bottleneck. A stream processor designed for packet analytics can avoid this overhead by pushing the load balancing down to the NIC, *e.g.,* by using multiple receiver queues [26]. Instead of reading from a load balancer, the worker nodes could each read from independent buffers, populated by the NIC driver. This optimization takes advantage of the inherent partitionability of packet records, which maps well to the capabilities of the underlying hardware, and the fact that packet records all have a common source, the NIC itself.

Concurrent Queues Kernels in stream processing systems communicate using queues. The queue implementation itself can have significant impact on the overall application performance. Stream processors use generic queue implementations that allow for multiple concurrent readers and writers [8]. Additionally, many queue implementations use mutual exclusions and locks to ensure thread-safety. Through simple experiments, we identified locks to be one of the main performance bottlenecks of queue implementations. We will elaborate more on these experiments in Section 6. The main benefit of multiple readers and writers is flexibility to implement more custom load balancing schemes. This is less important for packet records because they are highly

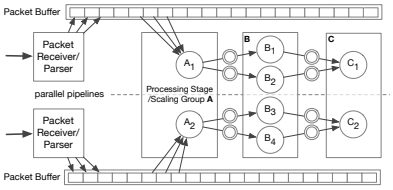

Figure 2: Detailed Stream Processor Architecture

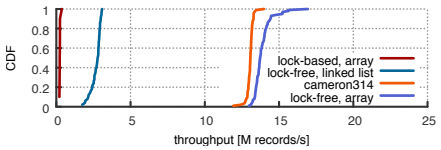

Figure 3: Performance of different concurrent queues

partitionable and we can largely offload this task to the NIC. Even without NIC support, load balancers can be implemented in preprocessing kernels, which would allow general reuse across many applications.

Hash Tables A common type of preprocessing in network analytics applications is aggregation, which requires a hash table. 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, encoding keys consisting of IP 5-tuple values as 128 bit integers to minimize the cost of key comparisons during lookup [1, 18]. The work on data plane implementations contains many other optimizations to draw from [36, 13].

4 A High-Throughput Stream Processing Architecture

To make high-performance packet-level network analytics feasible, we envision an architecture based on the paradigm of stream processing together with components optimized for the network domain. Our overall system is divided into three main parts, as depicted in Figure 1: A component that reads packet headers from a network interface card, the stream processor that does the analysis; and an alerting, visualization, or storage backend.

Packet Interface The packet interface is the part of the system that receives packet records from the respective network devices. Ideally, the records already come in a parsed format that only contains those header fields that are of interest for most applications. Otherwise, switches can generally mirror and truncate packets from a port. These packets can be relatively small (*e.g.,* 54 Bytes, which would include the Ethernet, IP and TCP/UDP headers), which is sufficient for most analytics applications since they do not perform deep-packet inspection. Packets in this case would have to be parsed, which however is computationally inexpensive due to the fixed memory layout of network packets. As explained before, operating system kernel bypass technologies can be used to read packets into user space memory at high rates.

Stream Processor The stream processor is the main and most critical component of our system. It needs to be able to properly load-balance traffic across cores while maintaining state in order to then perform the required computations on packets or intermediate results. In our model, all packets are stored in a large buffer and for the initial computation steps (where packets have not yet been transformed into tuples of a different type), pointers to packets are passed through the processing pipeline instead of copying data in or out of the FIFO queues connecting the processing elements as shown in Figure 2. Eventually, as soon as the original complete packet headers are not needed anymore, a packet can be marked as processed and will then be removed from the buffer to make space for new packets. Intermediate results that are passed between processing kernels are generally smaller in size than the full packet header (*e.g.* a counter for each 5-tuple) and are actually copied between processors. In order to scale to entire large-scale networks, multiple of such pipeline can run in parallel by making use of the partitionability of packet records (*cp.* Section 3.1).

Analytics Results Finally, results, which generally are being produced at significantly lower data rates and/or are also smaller in size, can be used to generate alerts or visualizations to easily monitor the status of the network. Alternatively, the results can simply be sent to storage for later, more detailed offline analysis, or auditing.

5 Prototype Implementation

As an initial proof of concept to illustrate that packet records processing in software at high data rates is feasible, we implemented prototype of the the stream processor part of the architecture described in Section 4 in C++11 only using the standard library. The processing kernels are implemented through instances of std::thread with a reference to a runtime manager, that maintains and provides access to the respective queues connecting a kernel to its neighbors in the processing graph. The application logic a kernel implements is specified by an anonymous function that is being called when a kernel begins execution.

As pointed out before, a suitable concurrent queue implementation is key to achieving high throughput. We tried different queue implementations using a simple experiment connecting two kernels and passing as many 4 Byte Integers through the queue as possible. These experiments were performed on a dual core 3.1 GHz Intel Core i7 CPU. The results are shown in Figure 3. We started out with a very simple array and lock-based queue implementation, improved this design to a lock-free and

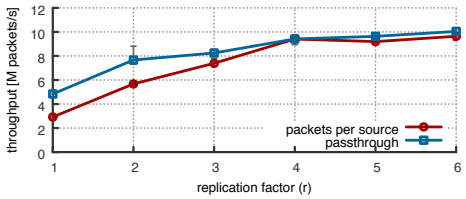

Figure 4: Throughput of example applications

linked list based implementation . We then found an open source high performing queue implementation [12] (cameron314). Finally, we implemented a custom lock-free concurrent queue using an array (*i.e.,* a ring buffer). For this implementation, which we are using in our prototype system, we use C++11 memory ordering primitives and atomic variables to achieve thread-safety. The results of this experiment show that a fast and carefully designed concurrent queue implementation is a key factor for achieving high performance in stream processors.

6 Evaluation

To evaluate the performance of our prototype, we implemented a simple (but somewhat typical) network monitoring application. The application computes the number of packets and bytes per source IP address on a packet stream. The first stage reads packets from a buffer, the second does the computation and maintains the state, and the last captures the results and performs benchmarking. The middle stage performs a hash table lookup for every single packet and can be parallelized by a replication factor r. The first kernel statically load balances across the replicas based on a hash of the source IP address.

Using a data set of 100 million packet records from a 10 Gb/s Internet backbone trace [4], we benchmarked the sustained throughput of the system on a 6 core 2.4 GHz Intel Xeon v3 CPU with 64 GB of RAM. Figure 4 shows the throughput as the number of replicas varies. Per socket, our system scales linearly to four replicas, excluding the source and sink kernels (*i.e.,* 6 threads – the number of cores). The application sustained around 2.9 M packets/s per core, enough to monitor around 58 Gb/s of aggregate link capacity with a single core, or 230 Gb/s of the evaluated workload with the entire server. For perspective, a 56 core server running StreamBox [22], sustained around 1 M packets/s (around 20 K packets/s per core) for a similar application that groups packets by IP pairs and computes average latency. For comparison, we also implemented a second application that just passes packet records through the same pipeline without any computations. This applications sustains around 4.8 M packets/s with a single core. Previous work [22] has shown, that multiple such independent pipelines can scale linearly with CPU core count.

To put these numbers into perspective of a cloud data center, we looked at traffic statistics reported by Facebook in [27, 14]. Facebook's data centers are organized in clusters, where each cluster contains either 4 or 8 racks with up to 44 servers each. In the web server cluster, the average server generates around 2 Gb/s of traffic, of which 500 Mb/s leaves the cluster. Given the reported median packet size of 120 bytes, this corresponds to 520K packets/s per server leaving the cluster. The aggregate egress traffic rate for a cluster is then 91M packets/s for 4 racks (176 servers) or 182M packets/s for 8 racks (352 servers). When processing around 2.9 M packets per second per core, this would require 32 cores for 4 racks or 64 cores for 8 racks, as compared to 5096 or 10192 cores for StreamBox [22] (a 163× improvement).

As a result, all egress traffic of such a cluster can be analyzed in software on a single server with between 32 and 64 cores. Typical high-performance commodity servers in 2U or 3U form factor (*e.g.*, [11]) have 4 sockets with 28 cores each. A single server in the context of such a cluster represents a maximum of 0.5% of the entire cluster hardware, which is more than reasonable to get insight into every single packet leaving the cluster.

7 Conclusion and Future Work

In this paper, we motivated why high performance network analytics at cloud data center or backbone WAN scales do not necessarily require sampling or aggregation. Instead, high-performance packet-level analytics can be realized with domain-specific optimizations and finely tuned software platforms. With this we get the benefits that software analytics systems have of simplified programmability through general-purpose languages, as well as deployability on commodity hardware.

Through preliminary experimentation we demonstrate that software analytics systems can indeed scale to several million packets per second per core with relatively simple optimizations and adaptations for the network domain. To further study our hypothesis that optimized and domain-specific software systems can process packet records at data center, we are in the process of building a complete end-to-end prototype. We plan to explore the rich set of possible optimizations including kernel-bypass technologies for packet input, optimized hash tables; along with load balancing strategies to distribute the analytics processing graphs across machines. This paper takes an important first step towards answering these questions and enabling powerful software analytics at the packet level in cloud-scale networks.

Acknowledgements

This research was supported in part by the National Science Foundation under grants 1700527 (SDI-CSCS) and 1652698 (CAREER), ONR grant N00014-15-1-2006, and DARPA contract HR001117C0047.

References

[1] 128bit hash comparison with sse. https://stackoverflow.com/questions/4534203/128bit-hash-comparison-with-sse.

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

[3] Rfc 3917: Requirements for ip flow information export (ipfix). https://tools.ietf.org/html/rfc3917.

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

[5] P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (jul 2014), 87–95.

[6] APACHE SOFTWARE FOUNDATION. Flink. https://flink.apache.org.

[7] APACHE SOFTWARE FOUNDATION. Kafka. http://kafka.apache.org.

[8] BEARD, J. C., LI, P., AND CHAMBERLAIN, R. D. Raftlib: A c++ template library for high performance stream parallel processing. *International Journal of High Performance Computing Applications* (2016).

[9] BHUYAN, M. H., BHATTACHARYYA, D. K., AND KALITA, J. K. Network anomaly detection: methods, systems and tools. *IEEE Communications Surveys & Tutorials 16*, 1 (2014), 303–336.

[10] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (2013), SIGCOMM '13, ACM, pp. 99–110.

[11] DELL. Poweredge r940 server. http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/poweredge-r940-spec-sheet.pdf.

[12] DESROCHERS, C. Concurrent queue. https://github.com/cameron314/concurrentqueue.

[13] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 15–28.

[14] FARRINGTON, N., AND ANDREYEV, A. Facebook's data center network architecture. In *2013 Optical Interconnects Conference* (may 2013), pp. 49–50.

[15] GUPTA, A., BIRKNER, R., CANINI, M., FEAMSTER, N., MACSTOKER, C., AND WILLINGER, W. Network Monitoring As a Streaming Analytics Problem. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2016), HotNets '16, ACM, pp. 106–112.

[16] HOFSTEDE, R., ČELEDA, P., TRAMMELL, B., DRAGO, I., SADRE, R., SPEROTTO, A., AND PRAS, A. Flow monitoring explained: from packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials 16*, 4 (2014), 2037–2064.

[17] INTEL. DPDK Performance Report. https://fast.dpdk.org/doc/perf/DPDK_16_11_Intel_NIC_performance_report.pdf.

[18] INTEL. Intel streaming simd extensions. https://software.intel.com/en-us/node/524253.

[19] KIM, H., CLAFFY, K. C., FOMENKOV, M., BARMAN, D., FALOUTSOS, M., AND LEE, K. Internet traffic classification demystified: myths, caveats, and the best practices. In *Proceedings of the 2008 ACM CoNEXT conference* (2008), ACM, p. 11.

[20] LI, Y., MIAO, R., KIM, C., AND YU, M. FlowRadar: A Better NetFlow for Data Centers. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation* (Santa Clara, CA, 2016), USENIX Association, pp. 311–324.

[21] LIU, Z., MANOUSIS, A., VORSANGER, G., SEKAR, V., AND BRAVERMAN, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 101–114.

[22] MIAO, H., PARK, H., JEON, M., PEKHIMENKO, G., MCKINLEY, K. S., AND LIN, F. X. StreamBox: Modern Stream Processing on a Multicore Machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), {USENIX} Association, pp. 617–629.

[23] NARAYANA, S., SIVARAMAN, A., NATHAN, V., GOYAL, P., ARUN, V., ALIZADEH, M., JEYAKUMAR, V., AND KIM, C. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 85–98.

[24] NGUYEN, T. T., AND ARMITAGE, G. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials 10*, 4 (2008), 56–76.

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

[26] RIZZO, L. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, 2012), USENIX Association, pp. 101–112.

[27] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 123–137.

[28] SFLOW. sflow sampling rates. http://blog.sflow.com/2009/06/sampling-rates.html.

[29] SONCHAK, J., AVIV, A. J., KELLER, E., AND SMITH, J. M. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings EuroSys 2018* (2018).

[30] SUH, J., KWON, T. T., DIXON, C., FELTER, W., AND CARTER, J. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on* (2014), IEEE, pp. 228–237.

[31] SUN, S., HUANG, Z., ZHONG, H., DAI, D., LIU, H., AND LI, J. Efficient monitoring of skyline queries over distributed data streams. *Knowledge and Information Systems 25*, 3 (Dec 2010), 575–606.

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

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

[34] YU, M., JOSE, L., AND MIAO, R. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), NSDI '13, USENIX Association, pp. 29–42.

[35] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., ET AL. Apache spark: A unified engine for big data processing. *Communications of the ACM 59*, 11, 56–65.

[36] ZHOU, D., FAN, B., LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2013), CoNEXT '13, ACM, pp. 97–108.

[37] ZHU, Y., KANG, N., CAO, J., GREENBERG, A., LU, G., MAHAJAN, R., MALTZ, D., YUAN, L., ZHANG, M., ZHAO, B. Y., AND ZHENG, H. Packet-Level Telemetry in Large Datacenter Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 479–491.