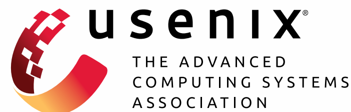

Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow

John Sonchack, *University of Pennsylvania;* Oliver Michel, *University of Colorado Boulder;* Adam J. Aviv, *United States Naval Academy;* Eric Keller, *University of Colorado Boulder;* Jonathan M. Smith, *University of Pennsylvania*

https://www.usenix.org/conference/atc18/presentation/sonchack

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow

John Sonchack[⋆], *Oliver Michel*[†], *Adam J. Aviv*[‡], *Eric Keller*[†], *and Jonathan M. Smith*[⋆]
[⋆]University of Pennsylvania, [‡]United States Naval Academy, and [†]University of Colorado, Boulder

Abstract

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

1 Introduction

Measurement plays a critical role in networking. Monitoring systems measure traffic for security [6, 35, 51, 36], load balancing [1, 26] and traffic engineering [55, 23, 27]; while engineers measure traffic and data plane performance to diagnose problems [14, 59, 25, 58, 56] and design new network architectures and systems [47, 5].

In high speed networks, which in 2018 have 100 Gb/s links and multi-Tb/s switches, it is challenging to support measurement without compromising on important practical requirements. Traditional switch hardware is inflexible and supports only coarse grained statistics [21, 45], while servers are prohibitively expensive to scale [50].

Fortunately, advances in switch hardware are presenting new opportunities. As the chip space and power cost of programmability drops [49, 7], switches are quickly moving towards reconfigurable ASICs [42, 44] that are capable of custom packet processing at high line rates. Recent telemetry systems [50, 40] have shown that these programmable forwarding engines (PFEs) can implement custom streaming measurement queries for fine-grained traffic and network performance statistics.

An open question, however, is whether telemetry systems can harness the flexibility and performance of PFEs *while also meeting requirements for practical deployment.* Current PFE accelerated telemetry systems [50, 40] focus on efficiency, compiling queries to minimize workload on servers in the telemetry infrastructure. Efficiency matters, but compiled queries do not address two other practical requirements that are equally important: concurrent measurement and dynamic queries.

First, support for concurrent measurement. In practice, there are likely to be multiple applications measuring the network concurrently, with queries for different statistics. A practical telemetry system needs to multiplex the PFE across all the simultaneously active queries. This is a challenge with compiled queries. Each query requires different computation that, given the line-rate processing model of a PFE [49], must map to dedicated computational resources, which are limited in PFEs.

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

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

At a high level, a query can be decomposed into three logical operations: a *select* operation that determines which packet header and metadata features to capture; a *grouping* operation that describes how to map packets to flows; and a *aggregation function* that defines how to compute statistics over the streams of grouped packet features. The primary benefit of using the PFE lies in its capability to implement the select and grouping operations efficiently because it has direct access to packet headers and low latency SRAM [40]. The challenge is implementing aggregation functions in the PFE, which are computationally complex and query dependent.

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

`*Flow`'s design, depicted in Figure 2, plays to the strengths of both PFEs and servers. Instead of compiling entire queries to the PFE, `*Flow` places parts of the *select* and *grouping* logic that are common to all queries into a match+action pipeline in the PFE. The pipeline operates at line rate and exports a stream of records that software can compute a diverse range of custom streaming statistics from *without needing to group per-packet records*. This design maintains the efficiency benefits of using a PFE while eliminating the root causes of concurrency and disruption issues. Further, it *increases* flexibility by enabling more complex aggregation functions than a PFE can support.

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

Each application can efficiently measure different aggregate statistics from the packet feature tuples in the same GPV stream. Applications can also dynamically change measurement without impacting the network, similar to what a stream of raw packet headers [25] would allow, but without the cost of cloning each packet to a server or grouping in software.

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

To utilize limited PFE memory, e.g., around 10MB as efficiently as possible, we introduce a key-value cache that supports dynamic memory allocation and can be implemented as a sequence of match+action tables for PFEs. It builds on recent match+action implementations of fixed width key-value caches [40, 29, 50] by introducing a line rate memory pool to support variable sized entries. Ultimately, dynamic memory allocation increases the average number of packet feature tuples that accumulate in a GPV before it needs to be evicted, which lowers the rate of processing that software must support.

Implementation and Evaluation. We implemented the `*Flow` cache [1] for a 100BF-32X switch, a 3.2 Tb/s switch with a Barefoot Tofino [42] PFE that is programmable with P4 [8]. The cache is compiler-guaranteed to run at line rate and uses a fixed amount of hardware resources regardless of the number or form of measurement queries.

To demonstrate the practicality of `*Flow`, we implemented three example monitoring applications that measure traffic in different ways: a host profiler that collects packet level timing details; a traffic classifier that measures complex flow statistics; and a micro-burst attributer that analyzes per-packet queue depths. Although these applications measure different statistics, they all can operate concurrently on the same stream of GPVs and dynamically change measurements without disrupting the network. Benchmarks show that the applications can scale to process GPVs at rates corresponding to Terabit-rate traffic while using under 10 cores.

To further demonstrate the practicality of `*Flow`, we also introduce a simple adapter for executing Marple [40] traffic queries on GPV streams. The adapter, built on top of RaftLib [3], supports high level query primitives (map, filter, groupby, and zip) designed for operator-driven performance monitoring. Using `*Flow` along with the adapter allows operators to run many different queries concurrently, without having to compile them all to the PFE or pause the network to change queries. Analysis shows that the `*Flow` PFE pipeline requires only as many computational resources in the PFE as *one* compiled Marple query. Currently, the adapter scales to measure 15-50 Gb/s of traffic per core, bottlenecked only by overheads in our proof-of-concept implementation.

Contributions. This paper has four main contributions. First, the idea of using *grouped packet vectors* (GPVs) to lift the aggregation functions of traffic queries out of data plane hardware. Second, the design of a novel PFE cache data structure with dynamic memory allocation for efficient GPV generation. Third, the evaluation of a prototype of `*Flow` implemented on a readily available commodity P4 switch. Finally, four monitoring applications that demonstrate the practicality of `*Flow`.

[1] `https://github.com/jsonch/starflow`

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

Table 1: Practical requirements for PFE supported network queries.

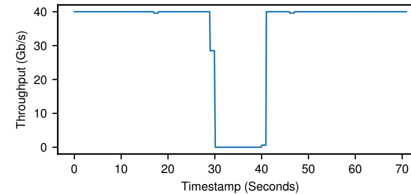

Figure 1: Network disruption from recompiling a PFE.

2 Background

In this section, we motivate the design goals of `*Flow` and describe prior telemetry systems.

2.1 Design Goals

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

Efficient. We focus on efficient usage of processing servers in the telemetry and monitoring infrastructure of a network. Efficiency is important because telemetry and monitoring systems need to scale to high throughputs [50] and network coverage [32]. An inefficient telemetry system deployed at scale can significantly increase the total cost of a network, in terms of dollars and power consumption.

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

Flexibility is also important for supporting future applications that may identify new useful metrics and systems that apply machine learning algorithms to analyze the network in many dimensions [43].

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

Consider a scenario where an operator is debugging an incast [15] and a network-wide security system is auditing for compromised hosts [36]. These applications would ideally run concurrently and need to measure different statistics. Debugging, for example, may benefit from measuring the number of simultaneously active TCP flows in a switch queue over small epochs, while a security application many require per-flow packet counters and timing statistics.

Dynamic. Support for dynamic queries is the capability to introduce or modify new queries at run time. It is important for monitoring applications, which may need to adapt as network conditions change, or themselves be launched at network run-time. Dynamic queries also enable interactive measurement [40] that can help network operators diagnose performance issues, e.g., *which queue is dropping packets between these hosts*?

2.2 Prior Telemetry Systems

Prior telemetry systems meet some, but not all, of the above design goals, as summarized in Table 1.

NetFlow Hardware. Many switches integrate hardware to generate NetFlow records [18] that summarize flows at the granularity of `IP` 5-tuple. NetFlow records are compact because they contain fully- computed aggregate statistics. ASICs [60, 20] in the switch data path do all the work of generating the records, so the overhead for monitoring and measurement applications is low. NetFlow is also dynamic. The ASICs are not embedded into the forwarding path, so a user can select different NetFlow features without pausing forwarding.

However, NetFlow sacrifices flexibility. Flow records have a fixed granularity and users choose statistics from a fixed list. Newer NetFlow ASICs [20] offer more statistics, but cannot support custom user-defined statistics or different granularities.

Software Processing. A more flexible approach is mirroring packets, or packet headers, to commodity servers that compute traffic statistics [19, 24, 22, 37]. Servers can also support concurrent and dynamic telemetry, as they are not in-line with data plane forwarding.

The drawback of software is efficiency. Two of the largest overheads for measurement in software are I/O [46], to get each packet or header to the measurement process, and hash table operations, to group packets by flow [50, 40, 33]. To demonstrate, we implemented a simple C++ application that reads packets from a PCAP, using lpcap, and computes the average packet length for each TCP flow. The application spent an average of 1535 cycles per packet on hash operations alone, using the relatively efficient C++ `std::unordered_map` [4]. In another application, which computed average packet length over pre-grouped vectors of packet lengths, the computation only took an average of 45 cycles per packet.

The benchmarks illustrate that mathematical opera-

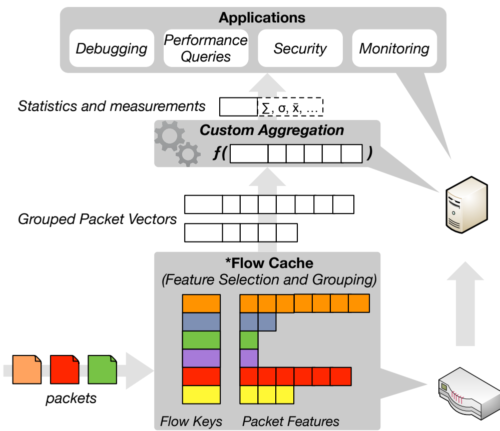

Figure 2: Overview of *Flow.

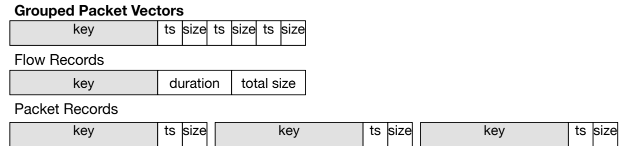

Figure 3: Comparison of grouped packet vectors, flow records, and packet records.

tions for computing aggregate statistics are *not* a significant bottleneck for measurement in software. Modern CPUs with vector instructions can perform upwards of 1 trillion floating point operations per second [39].

PFE Compiled Queries. Programmable forwarding engines (PFEs), the forwarding ASICs in next generation commodity switches [42, 13, 44, 41, 17, 53], are appealing for telemetry because they can perform stateful line-rate computation on packets. Several recent systems have shown that traffic measurement queries can compile to PFE configurations [50, 40]. These systems allow applications (or users) to define custom statistics computation functions and export records that include the aggregate statistics. Compiled queries provide efficiency and flexibility. However, they are not well suited for concurrent or dynamic measurement.

Concurrency is a challenge because of the processing models and computational resources available in a PFE. Each measurement query compiles to its own dedicated computational and memory resources in the PFE, to run in parallel at line rate. Computational resources are extremely limited, particularly those for stateful computation [49], making it challenging to fit more than a few queries concurrently.

Dynamic queries are a challenge because PFEs programs are statically compiled into configurations for the ALUs in the PFE. Adding a compiled query requires reloading the entire PFE program, which pauses all forwarding for multiple seconds, as Figure 1 shows. While it is possible to change forwarding rules at run-time to direct traffic through different pre-compiled functions, the actual computation can only be changed at compile time.

3 PFE Accelerated Telemetry with `*Flow`

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

In this section, we overview the architecture of `*Flow`, depicted in Figure 2.

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

`*Flow` Telemetry Switches. Switches with programmable forwarding engines [42, 49] (PFEs) compile the `*Flow` cache to their PFEs to generates GPVs. The cache is implemented as a sequence of match+action tables that applies to packets at line rate and in parallel with other data plane logic. The tables extract features tuples from packets; insert them into per-flow GPVs; and stream the GPVs to monitoring servers, using multicast if there are more than 1.

GPV Processing. A thin `*Flow` agent running on a server receives GPVs from the switch and copies them to per-application queues. Each application defines its own statistics to compute over the packet tuples in GPVs and can dynamically change them as needed. Since the packet tuples are pre-grouped, the computation is extremely efficient because the bottleneck of mapping packets to flows is removed. Further, if fine granularity is needed, the applications can analyze the individual packet feature themselves, e.g., to identify the root cause of short lived congestion events, as Section 6.2 describes.

4 Grouped Packet Vectors (GPVs)

*Flow exports telemetry data from the switch in the grouped packet vector (GPV) format, illustrated in Figure 3, a new record format designed to support the decoupling of packet feature selection and grouping from aggregate statistics computation. A grouped packet vector contains an IP 5-tuple flow key and a variable length vector of feature tuples from sequential packets in the respective flow. As Figure 3 shows, a GPV is a hybrid between a packet record and a flow record. It inherits some of the best attributes of both formats and also has unique benefits that are critical for *Flow.

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

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

5 Generating GPVs

The core of *Flow is a cache that maps packets to GPVs and runs at line rate in a switch's programmable forwarding engine (PFE). A GPV cache would be simple to implement in software. However, the target platforms for *Flow are the hardware data planes of next-generation networks; PFE ASICs that process packets at guaranteed line rates exceeding 1 billion packets per second [49, 9, 16]. To meet chip space and timing requirements, PFEs significantly restrict stateful operations, which makes it challenging to implement cache eviction and dynamic memory allocation.

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

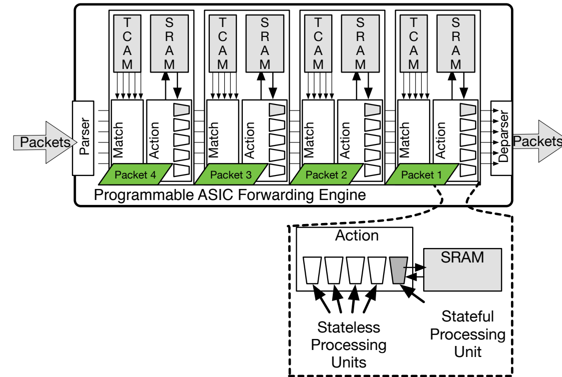

Figure 4: PFE architecture.

5.1 PFE Architecture

Figure 4 illustrates the general architecture of a PFE ASIC. It receives packets from multiple network interfaces, parses their headers, processes them with a pipeline of match tables and action processors, and finally deparses the packets and sends them to an output buffer. PFEs are designed specifically to implement match+action forwarding applications, e.g., P4 [8] programs, at guaranteed line rates that are orders of magnitude higher than other programmable platforms, such as CPUs, network processors [54], or FPGAs, assuming the same chip space and power budgets. They meet this goal with highly specialized architectures that exploit pipelining and instruction level parallelism [49, 9]. PFEs make it straightforward to implement custom terabit rate data planes, so long as they are limited to functionality that maps naturally to the match+action model, e.g., forwarding, access control, encapsulation, or address translation.

It can be challenging to take advantage of PFEs for more complex applications, especially those that require state persisting across packets, e.g., a cache. Persistent arrays, called "register arrays" in P4 programs, are stored in SRAM banks local to each action processor. They are limited in three important ways. First, a program can only access a register array from tables and actions implemented in the same stage. Second, each register array can only be accessed once per packet, using a *stateful ALU* that can implement simple programs for simultaneous reads and writes, conditional updates, and basic mathematical operations. Finally, the sequential dependencies between register arrays in the same stage are limited. In currently available PFEs [42], there can be no sequential dependencies; all of the registers in a stage must be accessed in parallel. Recent work, however, has demonstrated that future PFEs can ease this restriction to support pairwise dependencies, at the cost of slightly increased chip space [49] or lower line rates [16].

5.2 Design

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

Cache Eviction. The *Flow cache uses a simple *evict on collision* policy. Whenever a packet from an untracked flow arrives and the cache needs to make room for a new entry, it simply evicts the entry of a currently tracked flow with the same hash value. This policy is surprisingly effective in practice, as prior work has shown [34, 50, 40], because it approximates a least recently used policy.

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

This simple memory allocation policy is effective for *Flow because it leverages the long-tailed nature of packet inter-arrival time distributions [5]. In any given time interval, most of the packets arriving will be from a few highly active flows. A flow that fills up its narrow buffer in the short period of time before it is evicted is more likely to be one of the highly active flows. Allocating a wide buffer to such a flow will reduce the overall rate of messages to software, and thus its workload, by allowing the cache to accumulate more packet tuples in the ring buffer before needing to flush its contents to software.

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

5.3 Implementation

Using the above heuristics for cache eviction and memory allocation, we implemented the *Flow cache as a pipeline of P4 match+action tables for the Tofino [42]. The implementation consists of approximately 2000 lines of P4 code that implements the tables, 900 lines of Python code that implements a minimal control program to install rules into the tables at runtime, and a large library that is autogenerated by the Tofino's compiler toolchain. The source code is available at our repository [2] and has been tested on both the Tofino's cycle-accurate simulator and a Wedge 100BF-32X.

Figure 5 depicts the control flow of the pipeline. It extracts a tuple of features from each packet, maps the tuple to a GPV using a hash of the packet's key, and then either appends the tuple to a dynamically sized ring buffer (if the packet's flow is currently tracked), or evicts the GPV of a prior flow, frees memory, and replaces it with a new entry (if the packet's flow is not currently tracked).

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

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

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

5.4 Configuration

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

Feature tuple width depends on application requirements. For the other parameters, we implemented an OpenTuner [2] script that operates on a trace of packet arrival timestamps and a software model of the *Flow cache. The benchmarks in Section 7 show that performance under specific parameters is stable for long periods of time.

[2] https://github.com/jsonch/starflow

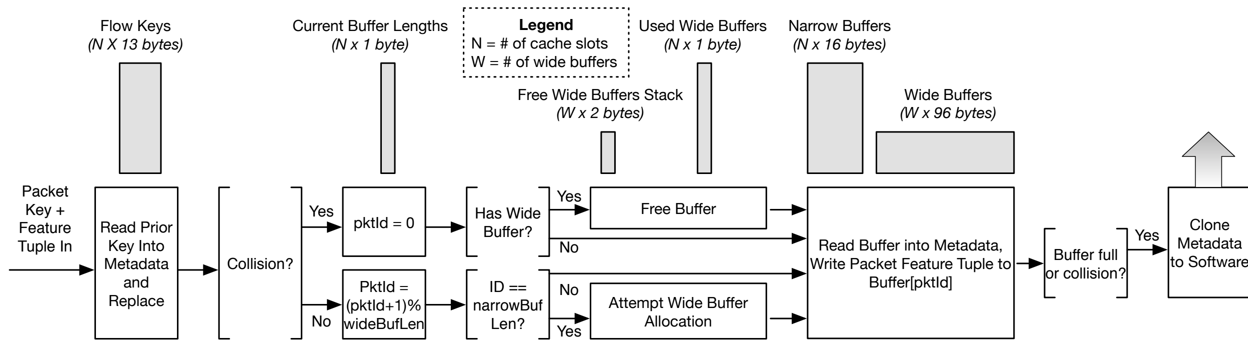

Figure 5: The *Flow cache as a match+action pipeline. White boxes represent sequences of actions, brackets represent conditions implemented as match rules, and gray boxed represent register arrays.

Run-time. The *Flow cache also allows operators to configure the following parameters at run-time by installing rules into P4 match+action tables. Immediately proceeding the *Flow cache, a filtering table lets operators install rules that determine which flows *Flow applies to, and which packet header and metadata fields go into packet feature tuples. After the *Flow cache, a table sets the destination of each exported GPV. The table can be configured to multicast GPVs to multiple servers and filter the GPV stream that each multicast group receives.

6 Processing GPVs

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

In this section, we describe the *Flow agent that receives GPVs from the *Flow cache, three motivating *Flow monitoring applications, and the *Flow adapter to execute operator-driven network performance measurement queries on GPV streams.

6.1 The *Flow Agent

The *Flow agent, implemented as a RaftLib [3] application, reads GPV packets from queues filled by NIC drivers and pushes them to application queues. While applications can process GPVs directly, the *Flow agent implements three performance and housekeeping functions that are generally useful.

Load Balancing. The *Flow agent supports load balancing in two directions. First, a single *Flow agent can load balance a GPV stream across multiple queues to support applications that require multiple per-core instances to support the rate of the GPV stream. Second, multiple *Flow agents can push GPVs to the same queue, to support applications that operate at higher rates than a single *Flow agent can support.

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

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

6.2 *Flow Monitoring Applications

To demonstrate the practicality of *Flow, we implemented three monitoring applications that require concurrent measurement of traffic in multiple dimensions or packet-level visibility into flows. These requirements go beyond what prior PFE accelerated systems could support with compiled queries. With *Flow, however, they can operate efficiently, concurrently, and dynamically.

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

Host Timing Profiler. The host timing profiler generates vectors that each contain the arrival times of all packets from a specific host within a time interval. Such timing profiles are used for protocol optimizers [55], simulators [10], and experiments [52].

Prior to *Flow, an application would build these vec-

tors by processing per- packet records in software, performing an expensive hash table operation to determine which host transmitted each packet.

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

Traffic Classifier. The traffic classifier uses machine learning models to predict which type of application generated a traffic flow. Many systems use flow classification, such as for QoS aware routing [23, 27], security [35, 51], or identifying applications using random port numbers or share ports. To maximize accuracy, these applications typically rely on feature vectors that contain dozens or even hundreds of different flow statistics [35]. The high cardinality is an obstacle to using PFEs for accelerating traffic classifiers, because it requires concurrent measurement in many dimensions.

*Flow is an ideal solution, since it allows an application to efficiently compute many features from the GPV stream generated by the *Flow cache. Our example classifier, based on prior work [43], measures the packet sizes of up to the first 8 packets, the means of packet sizes and inter-arrival times, and the standard deviations of packet size and inter-arrival times.

We implemented both training and classification applications, which use the same shared measurement and feature extraction code. The training application reads labeled "ground truth" GPVs from a binary file and builds a model using Dlib [30]; the classifier reads GPVs and predicts application classes using the model.

Micro-burst Diagnostics. This application detects micro-bursts [28, 48, 15], short lived congestion events in the network, and identifies the network hosts with packets in the congested queue at the point in time when the micro-burst occurred. This knowledge can help an operator or control application diagnose the root cause of periodic micro-bursts, e.g., TCP incasts [15], and also understand which hosts are affected by them.

Micro-bursts are difficult to debug because they occur at extremely small timescales, e.g., on the order of 10 microseconds [57]. At these timescales, visibility into host behavior at the granularity of individual packets is essential. Prior to *Flow, the only way for a monitoring system to have such visibility was to process a record from each packet in software [59, 25, 58, 56] and pay the overhead of frequent hash table operations.

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

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

6.3 Interactive Measurement Framework

An important motivation for network measurement, besides monitoring applications, is operator-driven performance measurement. Marple [40] is a recent system that lets PFEs accelerate this task. It presents a high level language for queries based around simple primitives (filter, map, group, and zip) and statistics computation functions. These queries, which can express a rich variety of measurement objectives, compile directly to the PFE, where they operate at high rates.

As discussed in Section 2, compiled queries make it challenging to support concurrent or dynamic measurement. Using *Flow, a measurement framework can gain the throughput benefits of PFE acceleration *without* sacrificing concurrency or dynamic queries, by implementing measurement queries in software, over a stream of GPVs, instead of in hardware, over a stream of packets.

To demonstrate, we extended the RaftLib [3] C++ stream processing framework with kernels that implement each of Marple's query primitives on a GPV stream. A user can define any Marple query by connecting the primitive kernels together in a connected graph defined in a short configuration file, similar to a Click [31] configuration file, but written in C++. The configuration compiles to a compact Linux application that operates on a stream of GPVs from the *Flow agent.

We re-wrote 6 example Marple queries from the original publication [40] as RaftLib configurations, listed in Table 4. The queries are functionally equivalent to the originals, but can all run concurrently and dynamically, without impacting each other or the network. These applications operate on GPVs with features used by the *Flow monitoring application, plus a 32 bit TCP sequence number in each packet feature tuple.

7 Evaluation

In this section, we evaluate our implementations of the *Flow cache, *Flow agent, and GPV processing applications. First, we analyze the PFE resource requirements and eviction rates of the *Flow cache to show that it is practical on real hardware. Next, we benchmark the *Flow agent and monitoring applications to quantify the scalability and flexibility benefits of GPVs. Finally, we

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

Table 2: Resource requirements for `*Flow` on the Tofino, configured with 16384 cache slots, 16384 16-byte short buffers, and 4096 96-byte wide buffers.

compare the `*Flow` measurement query framework with Marple, to showcase `*Flow`'s support for concurrent and dynamic measurement.

All benchmarks were done with 8 unsampled traces from 10 Gbit/s core Internet routers taken in 2015 [11]. Each trace contained around 1.5 billion packets.

7.1 The `*Flow` Cache

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

PFE Resource Usage. We analyzed the resource requirements of the `*Flow` cache configured with a tuple size of 32-bits, to support the `*Flow` monitoring applications, and a maximum GPV buffer length of 28, the maximum length possible while still fitting entirely into an ingress or egress pipeline of the Tofino. We used the tuning script, described in Section 5.4, to choose the remaining parameters using a 60 second trace from the 12/2015 dataset [12] and a limit of 1 MB of PFE memory.

Table 7.1 shows the computational and memory resource requirements for the `*Flow` cache on the Tofino, broken down by function. Utilization was low for most resources, besides stateful ALUs and stages. The cache used stateful ALUs heavily because it striped flow keys and packet feature vectors across the tofino's 32 bit register arrays, and each register array requires a separate sALU. It required 12 stages because many of the stateful operations were sequential: it had to access the key and packet count before attempting a memory allocation or free; and it had to perform the memory operation before updating the feature tuple buffer.

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

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

PFE Resources Vs. Eviction Rate. Figure 6 shows the average packet and GPV rates for the Internet router traces, using the `*Flow` cache with the Tofino pipeline configuration described above. Shaded areas represent the range of values observed. An application operating on GPVs from the `*Flow` cache instead of packet headers needed to process under 18% as many records, on average, while still having access to the features of individual packets. The cache tracked GPVs for an average of 640MS and a maximum of 131 seconds. 14% of GPVs were cached for longer than 1 second and 1.3% were cached for longer than 5 seconds.

To analyze workload reduction with other configurations, we measured *eviction ratio*: the ratio of evicted GPVs to packets. Eviction ratio depends on the configuration of the cache: the amount of memory it has available; the maximum possible buffer length; whether it uses the dynamic memory allocator; and its eviction policy. We measured eviction ratio as these parameters varied using a software model of the `*Flow` cache. The software model allowed us to evaluate how `*Flow` performs on not only today's PFEs, but also on future architectures. We analyzed configurations that use up to 32 MB of memory, pipelines long enough to store buffers for 32 packet feature tuples, and hardware support for an 8 way LRU eviction policy. Larger memories, longer pipelines, and more advanced eviction policies are all proposed features that are practical to include in next generation PFEs [9, 16, 40].

Figure 7 plots eviction ratio as cache memory size varies, for 4 configurations of caches: with or without dynamic memory allocation; and with either a hash on collision eviction policy or an 8 way LRU. Division of memory between and buffer slots between the narrow and wide buffers was selected by the AutoTuner script. With dynamic memory allocation, the eviction ratio was between 0.25 and 0.071. This corresponds to an event rate reduction of between $4X$ and $14X$ for software, compared to processing packet headers directly.

On average, dynamic memory allocation reduced the amount of SRAM required to achieve a target eviction ratio by a factor of 2. It provided as much benefit as an 8

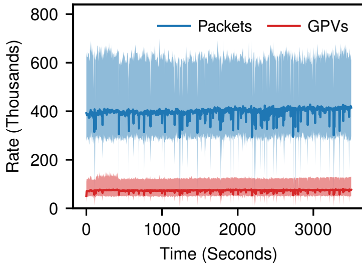

Figure 6: Min/avg./max of packet and GPV rates with *Flow for Tofino.

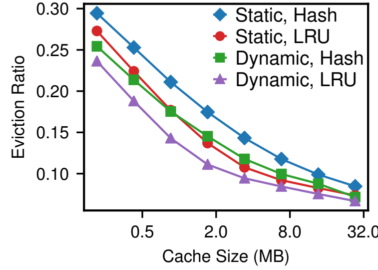

Figure 7: PFE memory vs eviction ratio.

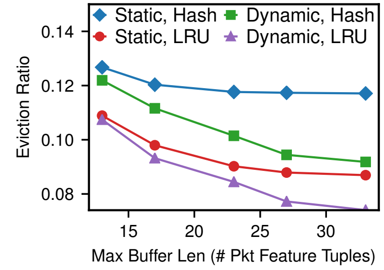

Figure 8: GPV buffer length vs eviction ratio.

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

Table 3: Average throughput, in GPVs per second, for *Flow agent and applications.

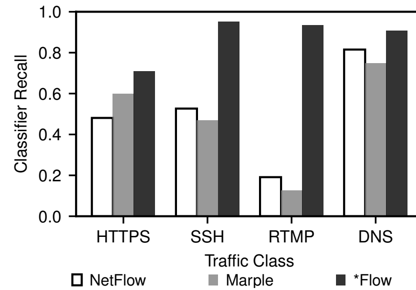

Figure 9: Recall of *Flow and baseline classifiers.

way LRU, but without requiring new hardware.

Figure 8 shows eviction rates as the maximum buffer length varied. Longer buffers required more pipeline stages, but significantly reduced eviction ratio when dynamic memory allocation was enabled.

7.2 *Flow Agent and Applications

We benchmarked the *Flow agent and monitoring applications, described in Section 6.2, to measure their throughput and quantify the flexibility of GPVs.

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

Throughput. Table 7.2 shows the average throughput of the *Flow agent and monitoring applications, in units of reassembled GPVs processed per second. For perspective, the average reassembled GPV rates for the 2015 10 Gbit/s Internet router traces, which are equal to their flow rates, are under 20 *thousand* per second [11].

The high throughput makes it practical for a single server to scale to terabit rate monitoring. A server using 10 cores, for example, can scale to cover over 100 such 10 Gb/s links by dedicating 8 cores to the *Flow agent and 2 cores to the profiler or classifier.

Throughput was highest for the profiler and classifier. Both applications scaled to over 10 M reassembled GPVs per second, each of which contained an average of 33 packet feature tuples. This corresponds to a processing rate of over 300 M packet tuples per second, around 750X the average packet rate of an individual 10 Gb/s Internet router link.

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

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

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

Table 4: Banzai pipeline usage for the `*Flow` cache and compiled Marple queries.

Classifier Accuracy. To quantify the flexibility benefits of GPVs, we compared the `*Flow` traffic classifier to traffic classifiers that only use features that prior, less flexible, telemetry systems can measure. The *NetFlow* classifier uses metrics available from a traditional NetFlow switch: duration, byte count, and packet count. The *Marple classifier* also includes the average and maximum packet sizes as features, representing a query that compiles to use approximately the same amount of PFE resources as the `*Flow` cache.

Figure 9 shows the recall of the traffic classifiers on the 12/2015 Internet router trace. The `*Flow` classifier performed best because it had access to additional features from the GPVs. This demonstrates the inherent benefit of `*Flow`, and flexible GPV records, for monitoring applications that rely on machine learning and data mining. Also, as Table 7.2 shows, the classifier was performant enough to classify >1 million GPVs per second per core, making it well suited to live processing.

7.3 Comparison with Marple

Finally, to showcase `*Flow`'s support for concurrent and dynamic measurement, we compare the resource requirements for operator driven measurements using compiled Marple queries against the requirements using `*Flow` and the framework described in Section 6.3.

PFE Resources. For comparison, we implemented the `*Flow` cache for the same platform that Marple queries compile to: Banzai [49], a configurable machine model of PFE ASICs. In Banzai, the computational resources of a PFE are abstracted as *atoms*, similar to sALUs, that are spread across a configurable number of stages. The pipeline has a fixed width, which defines the number of atoms in each stage.

Table 4 summarizes the resource usage for the Banzai implementation. The requirements for `*Flow` were similar to those of a *single* statically compiled Marple query. Implementing all 6 queries, which represent only a small fraction of the possible queries, would require 79 atoms, over 2X more than the `*Flow` cache. A GPV stream contains the information necessary to support *all* the queries concurrently, and software can dynamically change them as needed without interrupting the network.

Server Resources. The throughput of the `*Flow` analytics framework was between 40 to 45K GPVs/s per core. This corresponded to a per-core monitoring capacity of 15 - 50 Gb/s, depending on trace. Analysis suggested that the bottleneck in our current prototype is message passing overheads in the underlying stream processing library that can be significantly optimized [38].

Even without optimization, the server resource requirements of the `*Flow` analytics framework are similar to Marple, which required around one 8 core server per 640 Gb/s switch [40] to support measurement of flows that were evicted from the PFE early.

8 Conclusion

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

Acknowledgements

We thank the anonymous reviewers for their input on this paper. This research was supported in part by the National Science Foundation under grants 1406192, 1406225, and 1406177 (SaTC) and 1652698 (CAREER); ONR under grant N00014-15-1-2006; and DARPA under contract HR001117C0047.

References

[1] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)* (2010), vol. 7, pp. 19–19.

[2] ANSEL, J., KAMIL, S., VEERAMACHANENI, K., RAGAN-KELLEY, J., BOSBOOM, J., O'REILLY, U.-M., AND AMARASINGHE, S. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation* (2014), ACM, pp. 303–316.

[3] BEARD, J. C., LI, P., AND CHAMBERLAIN, R. D. Raftlib: a c++ template library for high performance stream parallel processing. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores* (2015), ACM, pp. 96–105.

[4] BELTON, J. Hash table shootout. `https://jimbelton.wordpress.com/2015/11/27/hash-table-shootout-updated/`.

[5] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), ACM, pp. 267–280.

[6] 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.

[7] BJORNER, N., CANINI, M., AND SULTANA, N. Report on networking and programming languages 2017. *ACM SIGCOMM Computer Communication Review 47*, 5 (2017), 39–41.

[8] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (July 2014), 87–95.

[9] 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 *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 99–110.

[10] BOTTA, A., DAINOTTI, A., AND PESCAPÉ, A. Do you trust your software-based traffic generator? *IEEE Communications Magazine 48*, 9 (2010).

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

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

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

[14] CHEN, A., WU, Y., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference* (2016), ACM, pp. 115–128.

[15] CHEN, Y., GRIFFITH, R., LIU, J., KATZ, R. H., AND JOSEPH, A. D. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking* (2009), ACM, pp. 73–82.

[16] CHOLE, S., FINGERHUT, A., MA, S., SIVARAMAN, A., VARGAFTIK, S., BERGER, A., MENDELSON, G., ALIZADEH, M., CHUANG, S.-T., KESLASSY, I., ET AL. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 1–14.

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

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

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

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

[21] CLAISE, B. Cisco systems netflow services export version 9. `https://tools.ietf.org/html/rfc3954`, 2004.

[22] DERI, L., AND SPA, N. nprobe: an open source netflow probe for gigabit networks. In *TERENA Networking Conference* (2003).

[23] EGILMEZ, H. E., CIVANLAR, S., AND TEKALP, A. M. An optimization framework for qos-enabled adaptive video streaming over openflow networks. *IEEE Transactions on Multimedia 15*, 3 (2013), 710–715.

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

[25] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., MAZIÈRES, D., AND MCKEOWN, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), vol. 14, pp. 71–85.

[26] HELLER, B., SEETHARAMAN, S., MAHADEVAN, P., YIAKOUMIS, Y., SHARMA, P., BANERJEE, S., AND MCKEOWN, N. Elastictree: Saving energy in data center networks. In *NSDI* (2010), vol. 10, pp. 249–264.

[27] HICKS, M., MOORE, J. T., WETHERALL, D., AND NETTLES, S. Experiences with capsule-based active networking. In *DARPA Active NEtworks Conference and Exposition, 2002. Proceedings* (2002), IEEE, pp. 16–24.

[28] JEYAKUMAR, V., ALIZADEH, M., GENG, Y., KIM, C., AND MAZIÈRES, D. Millions of little minions: Using packets for low latency network programming and visibility. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 3–14.

[29] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 121–136.

[30] KING, D. E. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research 10* (2009), 1755–1758.

[31] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (Aug 2000), 263–297.

[32] Li, Y., Miao, R., Kim, C., and Yu, M. Flowradar: a better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), pp. 311–324.

[33] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.

[34] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. Mica: A holistic approach to fast in-memory key-value storage. USENIX.

[35] Livadas, C., Walsh, R., Lapsley, D., and Strayer, W. T. Using machine learning technliques to identify botnet traffic. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on* (2006), IEEE, pp. 967–974.

[36] Lu, W., and Ghorbani, A. A. Network anomaly detection based on wavelet analysis. *EURASIP Journal on Advances in Signal Processing 2009* (2009), 4.

[37] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), USENIX Association, pp. 459–473.

[38] Michel, O., Sonchack, J., Keller, E., and Smith, J. M. Packet-level analytics in software without compromises. In *HotCloud* (2018).

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

[40] 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* (2017), ACM, pp. 85–98.

[41] Netronome. Agilio cx intelligent server adapters agilio cx intelligent server adapters. https://www.netronome.com/products/agilio-cx/, 2018.

[42] Networks, B. Barefoot tofino. https://www.barefootnetworks.com/technology/#tofino.

[43] 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.

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

[45] Phaal, P., Panchen, S., and McKee, N. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. Tech. rep., 2001.

[46] Rizzo, L., and Landi, M. Netmap: Memory Mapped Access to Network Devices. In *Proc. ACM SIGCOMM* (2011).

[47] 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.

[48] Shan, D., Jiang, W., and Ren, F. Absorbing micro-burst traffic by enhancing dynamic threshold policy of data center switches. In *Computer Communications (INFOCOM), 2015 IEEE Conference on* (2015), IEEE, pp. 118–126.

[49] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 15–28.

[50] Sonchack, J., Aviv, A. J., Keller, E., and Smith, J. M. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 11:1–11:16.

[51] Sperotto, A., Schaffrath, G., Sadre, R., Morariu, C., Pras, A., and Stiller, B. An overview of ip flow-based intrusion detection. *IEEE communications surveys & tutorials 12*, 3 (2010), 343–356.

[52] Vanini, E., Pan, R., Alizadeh, M., Taheri, P., and Edsall, T. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *NSDI* (2017), pp. 407–420.

[53] Wheeler, B. A new era of network processing. *The Linley Group, Technical Report* (2013).

[54] Wheeler, B. A new era of network processing. *The Linley Group, Tech. Rep* (2013).

[55] Winstein, K., and Balakrishnan, H. Tcp ex machina: Computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 123–134.

[56] Wu, Y., Chen, A., Haeberlen, A., Zhou, W., and Loo, B. T. Automated bug removal for software-defined networks. In *NSDI* (2017), pp. 719–733.

[57] Zhang, Q., Liu, V., Zeng, H., and Krishnamurthy, A. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference* (New York, NY, USA, 2017), IMC '17, ACM, pp. 78–85.

[58] Zhou, W., Sherr, M., Tao, T., Li, X., Loo, B. T., and Mao, Y. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 615–626.

[59] Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B. Y., et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 479–491.

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