TurboFlow: Information Rich Flow Record Generation on Commodity Switches

John Sonchack
University of Pennsylvania
jsonch@cis.upenn.edu

Adam J. Aviv
United States Naval Academy
aviv@usna.edu

Eric Keller
University of Colorado, Boulder
eric.keller@colorado.edu

Jonathan M. Smith
University of Pennsylvania
jms@cis.upenn.edu

ABSTRACT

Fine-grained traffic flow records enable many powerful applications, especially in combination with telemetry systems that supports high coverage, i.e., of every link and at all times. Current solutions, however, make undesirable trade-offs between infrastructure cost and information richness. Switches that generate flow records, e.g., NetFlow switches, are a low cost solution but current designs sacrifice information richness, e.g., by sampling. Information rich alternatives rely heavily on servers, which increases cost to the point that they are impractical for high coverage. In this paper, we present the design, implementation, and evaluation of TurboFlow, a flow record generator for programmable switches that does not compromise on either cost or information richness. TurboFlow produces fine- grained and unsampled flow records with custom features entirely at the switch *without relying on any support from external servers*. This is a challenge given high traffic rates and the limitations of switch hardware. To overcome, we decompose the flow record generation algorithm and optimize it for the heterogeneous processors in programmable switches. We show that with this design, TurboFlow can support multi-terabit workloads on readily available commodity switches to enable information rich monitoring with high coverage.

CCS CONCEPTS

• **Networks** → **Bridges and switches**; **Network measurement**; **Programmable networks**; **In-network processing**; **Network monitoring**; *Routers*; *Network economics*;

KEYWORDS

NetFlow, Network Monitoring, P4, Programmable Switch Hardware

ACM Reference Format:
John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. 2018. TurboFlow: Information Rich Flow Record Generation on Commodity Switches. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3190508.3190558

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04.
https://doi.org/10.1145/3190508.3190558

1 INTRODUCTION

High coverage network monitoring, across all the switches in a network and over long periods of time, enables many powerful applications. Security systems detect distributed threats by correlating the communication patterns of large groups of hosts [38], debuggers localize issues such as routing loops or black holes by comparing statistics from each switch along the path of problematic TCP flows [50], load balancers maximize bisection bandwidth by computing globally optimized routes [2], and networks themselves are designed based on large scale, long term measurement campaigns [4, 74, 78].

These applications all rely on telemetry support from the data plane to scale efficiently. In particular, many [5, 49, 53, 66, 84, 89] operate on flow records (FRs) that contain statistics about network traffic and performance aggregated by IP 5-tuple. FRs are appealing because they are extremely compact, e.g., 2-3 orders of magnitude smaller than the packets that they summarize [41], but are also fine-grained, preserving information about each individual TCP connection or UDP stream.

The information richness makes FRs well suited for many applications, but scaling the telemetry infrastructure up to support high coverage monitoring with FRs can be expensive. For example, server based FR generation appliances are rated for around 100 Gb/s of traffic [19, 33]. In large networks with thousands of switches [4, 74, 78], which can each forward traffic at multi-terabit rates, the telemetry infrastructure for high coverage FR monitoring would comprise many racks full of servers.

A more practical and cost effective approach is scaling with flow monitoring switches, e.g., NetFlow switches [20, 72, 94]. But generating FRs at the switch is challenging and current solutions require sacrificing the *information richness* of FRs, with respect to either their accuracy or feature set. A common approach is to generate FRs with software running on the switch CPU, based on sampled packets [67]. The sampling reduces accuracy, and therefore applications effectiveness. For example, sampling 1 out of every 1000 packets, a recommended ratio for 10 Gb/s links [76], will miss low rate attack flows [14]. Specialized monitoring ASICs are the alternative [20], which use hardware data paths and high speed memory to generate FRs at line rate. They offer high performance but lock the switch into exporting FRs with fixed, usually simple, features. This limits the applications that can use the FRs because many require more advanced or custom features [49, 84, 91].

Ultimately, the only solutions capable of generating *unsampled* FRs with *custom features* all place significant workloads on

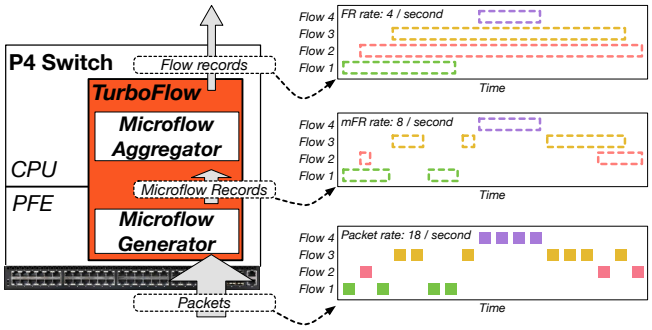

Figure 1: Overview of TurboFlow.

servers [19, 33, 50, 61], which makes them prohibitively expensive for high coverage monitoring in large or high speed networks.

Introducing TurboFlow. Motivated by the desire for practical high coverage FR monitoring without sacrificing information richness, we introduce `TurboFlow`, a FR generator optimized for programmable switches. `TurboFlow` produces fully customizable FRs for extremely high rate traffic, i.e., > 1 Tb/s, without sampling or relying on *any* support from servers. It can be deployed to readily available programmable switches [54], allowing them to serve as drop in replacements for current monitoring switches but with higher quality FRs that better support the hundreds of available monitoring applications.

Design. Generating information rich FRs at Tb/s rates with the processing resources available in a commodity switch is challenging. Switches have two types of programmable processors: programmable forwarding engines (PFEs), e.g., P4 [8] hardware; and standard CPUs, e.g., low power 2-8 core x86s. By themselves, neither processor can support an information rich FR generator. PFEs do not have enough memory to track all concurrently active flows, and have restrictive computational models that prevent implementation of the complex data structures required. Software on the switch CPU, on the other hand, does not have the throughput to support per-packet key value operations at the necessary rates.

Rather than trying to shoehorn the FR generator into either the PFE or CPU, we decompose it into two complementary parts that are well suited for the individual processing units. The PFE does preprocessing to reduce the workload of the CPU, while the CPU handles all the complex logic that the PFE could not support. Each processor relies on the other to overcome its limitations, and the modular design enables optimizations that improve performance by orders of magnitude.

Figure 3 depicts the high level idea. The PFE generates *microflow records* (mFRs), which are similar to FRs but only account for the most recent packets in each flow. Generating mFRs in the PFE instead of full FRs reduces the amount of state it must store, and allows us to use a simpler data structure better suited to the PFE's limitations. The data structure is similar to a hash table but with one important simplification. Whenever two flows collide, the PFE sends a mFR for the older colliding flow up to the switch CPU and replaces it with the newer entry. This simple logic can be implemented even on PFEs with highly restrictive computational models. The switch's DMA engine transfers mFRs to the switch CPU's main memory, where an aggregator groups them into full FRs using a hash table optimized for the task. These FRs can be exported directly to the analysis applications without any additional processing.

Even though it runs on a switch CPU, the aggregator does not need to sample to keep up with high rate traffic, i.e., multiple Tb/s. We optimize its main bottleneck, the hash table that maps mFRs to FRs, to reduce cache misses, accelerate key comparison operations, and mask memory latency. All of this maximizes the rate of mFRs that it can process. Additionally, since the PFE combines multiple packets into each mFR, the mFR rate is already much lower than the rate of packet arrivals.

Evaluation. We implemented `TurboFlow` on two P4 [8] switches with significantly different PFE architectures. First, the Wedge 100BF- 32X [54], a 32x100 GbE switch with a Tofino [64] PFE; second, a 4x10 GbE prototype switch using a NFP-4000 PFE [62]. On both platforms, `TurboFlow` could scale to monitor all links with traffic workloads from Internet and data center traces.

Benchmarks showed that our aggressive optimizations play a large role, improving the performance of the switch CPU component by a factor of 20. PFE acceleration was also significant, further improving performance by a factor of 10 or more, depending on the workload. Based on analysis of the benchmark results, high coverage and information rich monitoring with `TurboFlow` is cost effective in Internet and data center scenarios. Compared with other recent PFE accelerated telemetry systems, `TurboFlow` reduces the equipment and power cost of generating FRs by a factor of more than 5.

The implementation, benchmarks, and analysis demonstrate that `TurboFlow` *is a powerful telemetry system for high coverage and information rich network monitoring* that can be deployed to readily available commodity equipment.

Contributions. This paper makes 4 contributions. First, an analysis of the trade offs between richness and cost for high coverage flow monitoring. Second, `TurboFlow`, a FR generator optimized for the architectures of commodity programmable switches, enabling them to generate information rich FRs for high rate traffic without assistance from servers. Third, a working implementation of `TurboFlow` that can be deployed onto readily available equipment. Fourth, a thorough evaluation and cost analysis of `TurboFlow`, demonstrating that it enables high coverage and information rich network monitoring at low cost.

2 FLOW MONITORING SWITCHES

Flow records (FRs), depicted in Table 1, compactly summarize information about packet flows and how they were processed by the network. `TurboFlow` focuses on FRs that aggregate packets at the level of IP 5-tuple, i.e., by TCP connection or UDP stream, which are commonly referred to as *NetFlow* [18] or IPFIX [23] records, and used by many applications, as Table 2 shows.

FRs are an appealing record format because they are extremely compact, which makes network-wide monitoring practical. For example, an hour-long full packet trace from a 10 Gb/s Internet router link would contain nearly 1 TB of data and over 1 billion packets [12]. At such high rates, it is not practical to collect or

	Flow 1	Flow 2
Flow Key		
Source IP	10.1.1.1	10.1.1.6
Dest. IP	10.1.1.2	10.1.1.7
Source Port	34562	12520
Dest. Port	80	88
Protocol	TCP	UDP
Flow Features		
Packet Count	5	7
Byte Count	88647	3452
Max Queue Length	0	34
Avg. End-to-end Latency (us)	10	300

Table 1: Example flow records.

Feature Type	Examples	Applications
Traffic Characteristics		
Metadata	QoS type, IP options, TCP options & flags	Security [84], flow scheduling [2, 40], auditing [49], heavy hitter detection [92], QoS monitoring [61]
Statistics	duration, packet count, byte count, jitter, max packet size	
Network Performance		
Metadata	ingress port, egress port, selected route	Loop and black hole debugging [50], performance queries [61], load balancing [79], network design [74]
Statistics	max queue depth, avg. latency, dropped packet count	

Table 2: Types of FR features and example applications.

analyze data from more than a hand full of links. A FR trace that summarizes the packets with FRs that have the format depicted in Table 1, on the other hand, is around 5 GB with 50 million records. For an application that analyzes or collects FRs, this represents a 20X reduction in processing rate and a 200X reduction in bit rate. At these much lower rates, an application implemented with an efficient stream analytics framework [56] could monitor *hundreds* of such 10 Gb/s links with a single server.

FRs are also appealing because they summarize traffic at the level of individual TCP or UDP streams. The fine granularity preserves information that is important for many applications. For example, host communication patterns can reveal botnets [38] or other attacks [84] while per-stream packet counts can make it easier to localize misconfiguration in the network core [50] and perform traffic engineering [88]. Additionally, the fine granularity makes FRs flexible – FRs with the same features can be used for many different applications. This is in contrast with other approaches to scalable monitoring, such as sketching [92], where the network is configured to measure coarse grained statistics that are relevant to specific applications.

The compactness and fine granularity of FRs enables efficient and powerful network-wide monitoring applications. However, generating the FRs represents additional (and potentially significant) work for the network infrastructure, especially when high coverage is the goal. There are two main approaches to FR generation. First, dedicated appliances or *NetFlow probes* [29]; commodity servers that convert packets, mirrored from network switches, into FRs. Appliances are designed to cover individual links, e.g., they are rated for around 40 - 100 Gb/s of traffic [19, 33], which makes them a cost prohibitive solution for high coverage monitoring.

A more cost effective approach is to use *flow monitoring switches*, which generate FRs summarizing the packets that they forward. Eliminating the reliance on servers in the FR generation process greatly reduces infrastructure cost. Operating at the switch has the additional benefit of providing visibility into statistics and metadata related to how a switch processes packets, for example, input and output ports, queue lengths, and ingress timestamps. This visibility enables *network performance*, such as those shown in Table 2, that an appliance could not compute.

FR generation at the switch is highly desirable, but also challenging because of high packet and flow arrival rates. On a single 10 Gb/s Internet link, flow and packet arrival rates are around 10 - 50 K and 300 - 500 K per second, respectively [13]. Switches have aggregate throughput hundreds of times larger, in the multi-terabit range [54], meaning a switch based FR generator needs to keep up with tens and hundreds of millions of flows and packets per second.

Current systems work around the challenge by sacrificing *feature richness* or *accuracy*, or relying on resource intensive post processing with servers. We describe these design goals and summarize prior systems below.

2.1 Design Goals

Feature Richness. Feature richness is the capability of a flow monitoring switch to include custom features in FRs. Supporting custom features enables a wider range of applications because each application relies on different features based on its objective. For example, consider bot detection [38], QoS measurement [88], and incast debugging [61]. The goal of a bot detection system is often to understand communication patterns between hosts, which only requires FRs with packet counters, byte counters, and timestamps. QoS measurement, on the other hand, also requires statistics about relevant QoS metrics, such as dropped packet counts and path delays. Incast debugging requires completely different features that describe switch performance, e.g., queue depth, rather than traffic characteristics.

Feature richness also improves the effectiveness of ML applications, e.g., traffic classifiers [91] or anomaly detectors [5], which can take advantage of a wide variety of features.

Accuracy. Accuracy describes how closely FRs represent the underlying traffic. There are two dimensions to accuracy. First, *feature accuracy* of the statistics in a single FR. Feature accuracy directly impacts the accuracy of the monitoring applications. For example, traffic load balancers [2, 40] change the routes of elephant

	Low Generation Cost	Accurate Records	Rich Features
Fixed FE			
Packet Sampling	✓	✗	✓
NetFlow ASICs	✓	✓	✗
PFE Accelerated			
FlowRadar	✗	✓	✗
Marple	✗	✓	✓
TurboFlow	✓	✓	✓

Table 3: Comparison with prior switch FR generators.

flows contending for the same links, to maximize network bisection bandwidth. Inaccurate traffic statistics can cause the load balancers to identify the wrong flows as heavy hitters, or miss actual heavy hitters [88], which reduces their effectiveness. As another example, inaccurate statistics can cause security systems to miss anomalies [10], such as the outbreak of an attack.

An orthogonal type of accuracy is *flow accuracy*, or what fraction of the flows are represented in the FRs. Flow accuracy impacts security applications. For example, if the FRs are biased to miss short flows, which occurs naturally when generating FRs from randomly sampled packets [34], intrusion detection systems [84] can underestimate the magnitude of DDoS attacks [58] with many small flows. FR generation systems that do not represent all flows are also vulnerable to attacks that specifically try to avoid detection [36].

Generation Cost. Generation cost quantifies the number of servers required to convert data exported from the switch into FRs, which directly increases equipment and power costs of the network. Operators [37] and researchers [1, 40, 47] go to great lengths to minimize these costs, which makes it unlikely that systems with high generation overhead will be deployed for full coverage monitoring in practice, regardless of how much they benefit applications. Instead, operators often must rely on lower cost approaches that sacrifice feature richness or accuracy. For example, recent data center measurement studies used packet sampling at extremely high ratios, e.g., 1:30000 [74] packets, which skews the accuracy of both FRs and FR features [31].

2.2 Prior Systems

Flow monitoring has a long history of use [49], and many systems for switch based FR generation have been developed. Prior literature describe these systems in detail [41], we classify them into two broad categories that have similar general properties with respect to feature richness, accuracy, and generation cost. Table 3 summarizes the systems, along with more recent telemetry platforms that, like `TurboFlow`, leverage programmable forwarding engines in modern switches.

Sampling. Sampling systems clone a fraction of packets or flows [34, 67, 75], sometimes along with processing metadata, from the switch forwarding engine to its CPU, which generates FRs in software. The sampling is necessary to prevent overloading the CPU, but reduces the feature and flow accuracy of the records [31]. Sampling is widely used in practice because it has low cost and is available on many commodity switches [67].

NetFlow ASICs. Some switches integrate custom hardware to generate FRs [20, 72, 94]. This approach produces accurate flow records, but locks the infrastructure into using a fixed set of flow features. Older ASICs were limited to simple counters [94], though some newer ASICs offer more features. Cisco's FlowCache ASIC, for example, supports additional performance features including latency, TCP window size, packet size, TTL, and TCP option variation [68]. These features are useful, however since the ASICs are still fixed-function hardware, they do not support custom features. An additional limitation of ASICs is integration, as they are usually designed for vendor-specific frameworks [68] and embedded into non- commodity switches.

PFE Accelerated. An recent trend in networking is the emergence of programmable forwarding engines (PFEs)[15, 63, 64], hardware in next generation switches and line cards that is capable of custom line rate packet processing. PFEs are emerging now because the chip area and power difference between programmable and fixed function hardware is becoming negligible, while the ever increasing number of protocols is making fixed function ASICs impractical [6]. PFEs are appealing for monitoring because they can compute custom statistics at line rate [61, 79] over a rich set of packet header and processing metadata.

Several recent systems [50, 61] have proposed to leverage their capabilities for telemetry similar to FR generation. The main drawback of these systems is that using them to generate FRs would require significant post processing at servers. In FlowRadar [50], the PFE encodes per-flow statistics into counting Bloom filters [7], which a server must later decode. The decoding is expensive, especially for FRs that include features besides IP 5-tuple.

Marple [61] is a system for streaming queries for network statistics. It splits the query processing between a PFE and a scale out key-value store, e.g., Redis [71]. The PFE partially computes the statistics requested by the query and streams multiple updates for each flow to the key-value store, which aggregates the updates together. A Marple query can request a stream of FRs, which can include a rich feature set containing a large class of statistics that are efficiently mergable [61], i.e., each update only requires the PFE to send a bounded amount of state to the backing store. However, using Marple for FR generation is expensive because of its reliance on scale out key-value stores. For example, it is reported to require around 1 key-value server to support queries for a single 64 x 10 GbE switch.

3 TURBOFLOW OVERVIEW

`TurboFlow` is a FR generator for commodity programmable switches with P4 PFEs [62, 64] that produces accurate and feature rich FRs for terabit rate traffic without requiring *any* support from collection servers. Figure 2 depicts how `TurboFlow` integrates into a network infrastructure. The ingress pipeline of the switch's PFE includes a `TurboFlow` module in its P4 ingress pipeline that generates microflow records (mFRs) and the switch's CPU runs a `TurboFlow` process that converts the mFRs into FRs. Operators can customize

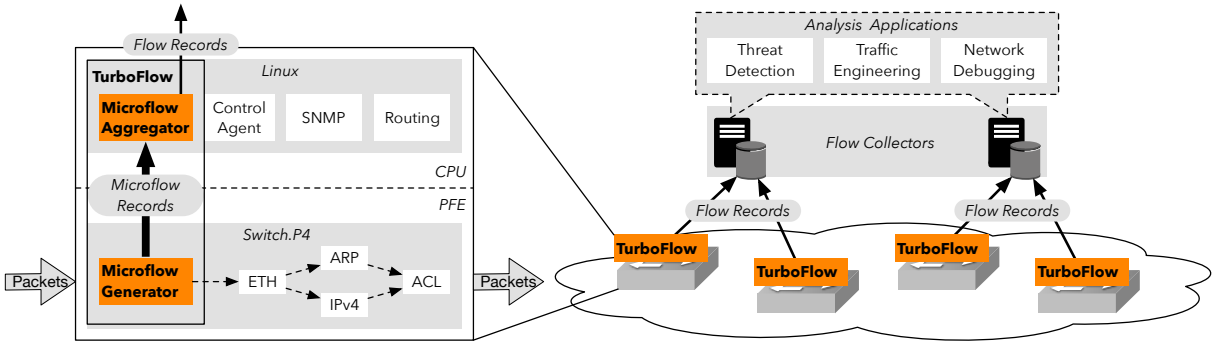

Figure 2: Deployment model of TurboFlow.

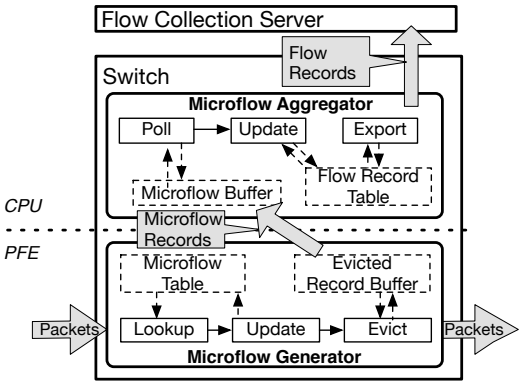

Figure 3: TurboFlow architecture.

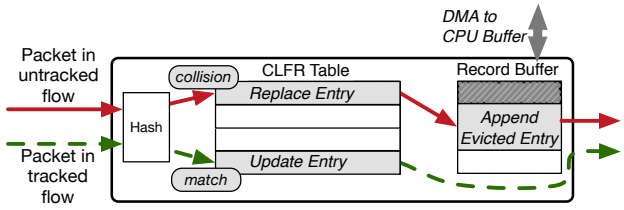

Figure 4: MFR generator data structure.

the features that TurboFlow includes in the FRs, which can also be packed into common formats, such as IPFIX [23] or NetFlow [22], that are used by many flow collectors [82] and analysis applications [5, 49, 53, 65, 66, 82, 84, 89, 91].

Challenges. Programmable switches have two types of processors: general purpose CPUs and specialized programmable forwarding engines (PFEs). Neither processor can support the full FR generation workload itself. Switch CPUs cannot support the required packet rates, e.g., hundreds of millions of packets per second for terabit rate traffic. The main bottleneck is mapping packets to FRs. In a terabit rate switch, packet arrival rates can be in the hundreds of millions per second. Each packet must be mapped to a FR, which takes many cycles because of high memory latencies and expensive key comparison operations. For example, using Redis [71] to map packet to FRs with the Wedge 100BF- 32X's CPU (a quad core Intel D1517 [43]) provides a throughput of around 500 K per core – over two orders of magnitude lower than necessary.

PFEs, on the other hand, can support key lookups at high enough rates using on chip memory, but the memory is too small to store FRs for the full set of active flows. Additionally, to guarantee high line rates while meeting chip space and timing budgets [81], many PFEs have restricted computational models that prevent implementing a full key-value data structure that can support operations *other* than lookups, e.g., insertions, at line rate.

Design. With TurboFlow, depicted in Figure 3, we overcome these challenges by decomposing the FR generation algorithm into components that can be optimized for each processor in a programmable switch. The PFE produces microflow records (mFRs) that summarize active flows over short timescales. Focusing on mFR generation instead of complete FR generation reduces the set of concurrently active flows to lower memory requirements, and permits simpler data structures that map well to PFE hardware.

A mFR aggregator, running on the switch CPU, stitches the mFRs together into complete flow records, using a key value data structure we optimized to leverage performance features in modern CPUs. Operating on mFRs instead of packets lowers the rate of key value operations that the CPU must sustain, while the optimizations reduces their individual cost to further maximize throughput.

4 THE MFR GENERATOR

The mFR generator produces mFRs that summarize short sequences of packets within flows. MFRs have the same format as FRs. As depicted in Table 1, they contain a flow key and features describing packets in the flow or how they were processed by the switch. MFRs can include any features that the PFE can compute, based on the limitations described in Section 2.2. This includes all statistics supported by NetFlow ASICs, and all performance metrics that can be represented in scalable Marple [61] queries.

The mFR generator, illustrated in Figure 4, contains a mFR table and a mFR record buffer. The *mFR table* maps packet keys to records based on their hash values. If the record has the same key as the packet, its features are updated. If the keys do not match, the current record is replaced with a new one and appended to the end of an

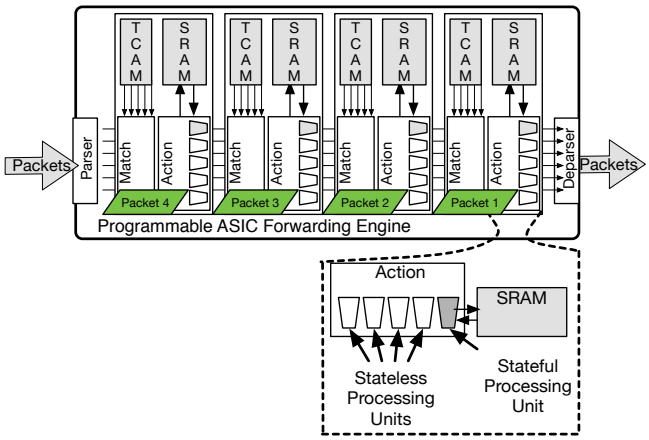

Figure 5: Generalized architecture of a P4 programmable ASIC.

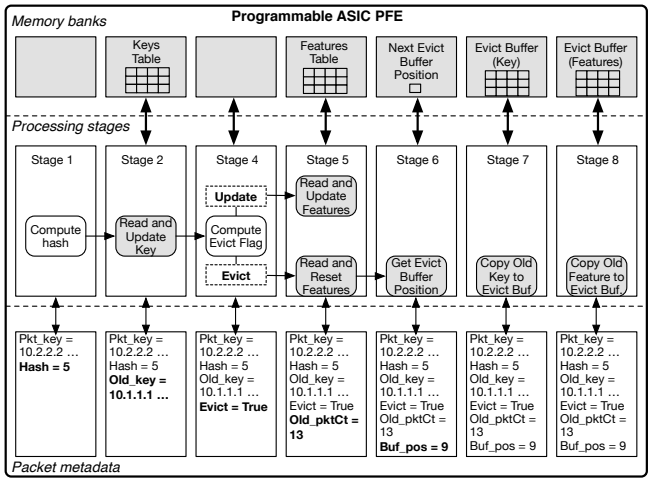

Figure 6: MFR generator mapped to a programmable ASIC.

evicted record buffer, a ring buffer that is DMAed to the switch CPU's main memory.

The mFR generator is designed specifically so that it can map to the restrictive computational models of real PFE hardware. Below, we describe how it maps to programmable P4 ASICs [9, 81], which are highly restrictive; and less restrictive but lower throughput NPUs [62, 90].

Mapping to P4 Programmable ASICs. Figure 5 illustrates the general architecture of a P4 programmable ASIC. It has a pipeline of stages that each spends a fixed number of cycles processing each packet. A stage contains TCAM to store forwarding rules, SRAM for counters and other state that persists across packets, a vector of processing units to modify packet headers or metadata carried along with it, and a small number of stateful processing units that can update state in SRAM that persists between packets, i.e., *register arrays* in P4 14 [8].

There are two benefits to this architecture. First, it provides an extremely high and guaranteed throughput – any P4 program that compiles to the PFE will run at line rate, which is around 1 billion packets per second for recent designs [9, 64, 81]. Second, the primitives of P4 map almost directly to the hardware, so match + action packet processing, which has many applications, is straightforward to implement.

It is not as simple for more complex packet processing to take advantage of the hardware's high performance. Sequential logic, i.e., operations that depend on each other, must be implemented as a series of actions in multiple stages. Stateful operations, which `TurboFlow` also relies on, are also highly constrained. Each variable that persists across packets can only be accessed once per packet, to meet the one cycle time budget. To enable more complex algorithms, the stateful units that access memory can execute small atomic programs, e.g., to simultaneously read and write, predicate updates, or perform simple mathematical operations [79, 81].

Figure 6 illustrates how the mFR generator maps to a pipeline of logical tables that use register arrays (in SRAM banks local to each stage), packet metadata, and P4 primitive instructions to implement its functionality. The pipeline operates on packets in parallel with any other forwarding functionality. There are no back branches or loops in the pipeline, which is a requirement of both the P4 language and ASICs. Also, not depicted in the figure, all the persistent state is striped across memory banks by word. This allows the pipeline to be implemented with at most 1 read or write to any memory bank, per packet. The logical stages in Figure 6 implement the following logic.

(1) Computes the hash of the packet's key.
(2) Loads the key of the last flow with that hash from the *key table* into a metadata register, then writes the current packet's key back.
(3) Sets a metadata flag indicating whether or not an evict is needed by comparing the current key with the previous key, which is now in metadata.
(4) Loads the features values of the flow from the feature table into metadata, and resets or updates the features depending on the evict flag.
(5) Loads the next free position in the output buffer, writes back a conditionally updated value: if the evict flag is set, the previous position plus the length of a mFR record; if not, the original value.
(6) Writes the key of the previous flow (loaded in stage 2) to the evicted key buffer.
(7) Writes the features of the previous flow (loaded in stage 5) to the evicted feature buffer.

Mapping to an NPU. Network processors (NPUs) [63, 90] are an older and more flexible architecture for high throughput packet processing. The trade off is lower maximum throughput than programmable ASICs, with no hardware guarantees. There are many NPU architectures, the most widely available today us a *run to completion* execution model where the NPU processes packets with a pool of RISC cores, as Figure 7 depicts. The cores are arranged into islands with local SRAM for code and data storage, a large shared

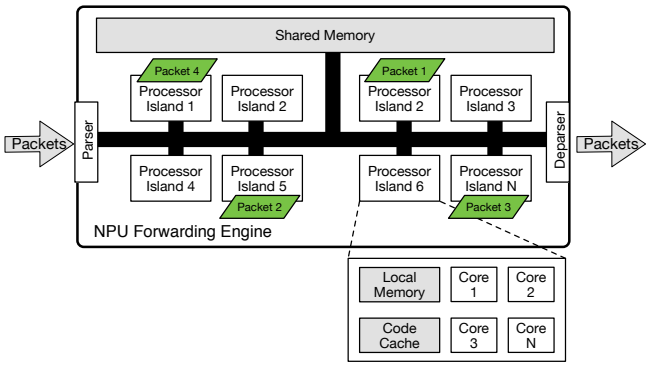

Figure 7: Generalized architecture of a NPU.

Added Instruction	Example P4 14 Code	Throughput Change
Control flow branch	`if (...){ apply(...); }`	-15.5%
Apply table	`apply(...);`	-9.5%
Read memory	`register_read(...);`	-3.0%
Write memory	`register_write(...);`	-3.0%
Modify header	`modify_field(...);`	-0.5%

Table 4: P4 primitives cost on the NFP-4000.

off-chip DRAM for forwarding tables and other persistent state, and a high bandwidth switch fabric interconnecting the components.

Mapping the mFR generator to an NPU is more straightforward than mapping to a programmable ASIC, but also requires optimization to maximize throughput. One consideration is minimizing the number of cycles required to process each packet, which directly impacts throughput. Figure 4 shows the cost of primitive P4 instructions in a simple forwarding application running on one core of the NFP-4000 NPU [63]. Applying tables and branching in the outer control flow of the program is surprisingly expensive because these P4 primitives do not map natively to the underlying hardware, and are instead implemented as software routines. To eliminate the overhead of applying many tables, we designed the mFR generator to minimize branches. Figure 8 shows P4-14 psuedocode for the mFR generator, which only requires applying 2 tables per packet.

A second performance concern is synchronizing state without contention. Each core operates on a different packet concurrently, and needs thread safe access to the shared mFR state. The P4 library for the NFP-4000 has a coarse grained semaphore that locks an entire array. This causes high contention and only allows one thread to operate at a time. To synchronize more efficiently, we implemented as simple spin-lock semaphore that allows a core to lock the row of the mFR table or record buffer that it needs to access, as shown in Figure 9. This reduces contention as there are many rows in the mFR table and record buffer. It also only requires 4 synchronization operations per packet (a lock / unlock for the mFR table and a lock / unlock for the record buffer).

```
// Metadata.
metadata tempMfr_t tempMfr;
metadata pktMeta_t md;

// Register arrays to store mFR table and evict buffer.
register keyArr[NUM_MICROFLOWS_TRACKED];
register pktCtArr[NUM_MICROFLOWS_TRACKED];
register evictBufArr[1];
register evictBufKey[BUF_SIZE];
register evictBufPktCt[BUF_SIZE];

// Control function -- call from P4 ingress.
control MfrGenerator {
   apply(UpdateKey);
   if (md.keyXor == 0) {
      apply(UpdateFeatures);
   } else {
      apply(ResetFeatures);
   }
}

// Tables.
table UpdateKey { default_action :UpdateKeyAction(); }
table UpdateFeatures { default_action
     :UpdateFeaturesAction(); }
table ResetFeatures { default_action
     :ResetFeaturesAction(); }

// Actions.
// Update key for every packet.
action UpdateKeyAction() {
   modify_field_with_hash_based_offset(md.hash, 0,
        key_field_list, HASH_SIZE);
   register_read(tempMfr.key, keyArr, md.hash);
   register_write(keyArr, md.hash, pkt.key);
   modify_field(tempMfr.keyXor,
      (pkt.key string^ tempMfr.key));
}
// Update features when there is no collision.
action UpdateFeaturesAction() {
   register_read(tempMfr.pktCt, pktCtArr, md.hash);
   register_write(pktCtArr, md.hash, tempMfr.pktCt+1);
}
// Reset features and evict on collision.
action ResetFeaturesAction() {
   register_read(tempMfr.pktCt, pktCtArr, md.hash);
   register_write(pktCtArr, md.hash, 1);
   register_read(tempMfr.evictBufPos, evictBufArr, 0);
   register_write(evictBufArr, 0, tempMfr.evictBufPos+1);
   register_write(evictBufKey, tempMfr.evictBufPos,
        tempMfr.key);
   register_write(evictBufPktCt, tempMfr.evictBufPos,
        tempMfr.pktCt);
}
```

Figure 8: mFR generator psuedocode for NFP-4000.

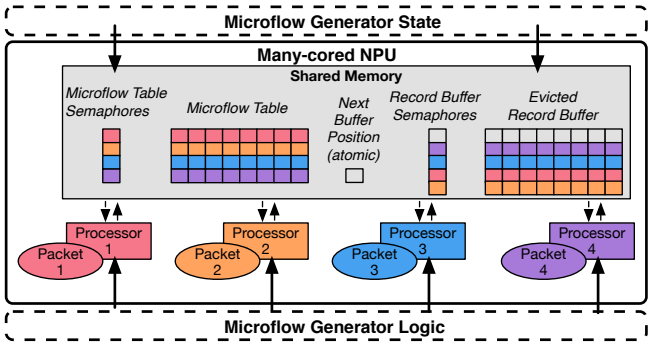

Figure 9: Mapping the mFR generator to a NPU with fine-grained semaphores.

5 THE MFR AGGREGATOR

The mFR aggregator runs on the switch CPU and stitches mFRs, streamed up from the PFE, into full FRs. The challenge for the mFR aggregator is optimizing the key-value data structure that maps mFRs to complete FRs, which is the main bottleneck.

5.1 Reading mFRs

The DMA engine of the PFE copies mFRs from the evicted records buffer into free cells in a larger ring buffer in main memory. The mFR aggregator processes the mFRs in batches and sends the addresses of free cells back to the DMA engine. This design is similar to high performance NIC drivers [73], and allows the DMA engine to dynamically vary the copy rate, as long as there are free cells available.

When running on multiple cores, each mFR aggregator maintains its own buffer and a corresponding evicted record buffer in the PFE. The PFE statically load balances mFRs across the buffers based on key.

5.2 Aggregating mFRs

The aggregator stores complete FRs in a hash table. For each mFR, the aggregator either updates the features of an existing record or inserts a new entry. Whenever the aggregator processes a `TCP` mFR with a `FIN` flag, it marks the flow as expired, copies it to an output buffer, and removes it from the hash table. The hash table is the core bottleneck for the aggregator, and optimizing it was the focus of our implementation. We found four optimizations that had significant performance benefit.

Linear Probing. `TurboFlow` uses a linear probing hash table [69]. Compared to linked list [71] or quadratic probing implementations, which are more common, linear probing requires more memory to keep collision rates low. However, it has an important benefit for `TurboFlow`: better cache locality. In the case of a miss, the next mFRs to check will already be in the cache.

Flat Tables. The aggregator stores mFRs directly in the hash table, i.e., each slot stores a FR, rather than a pointer to a data structure. This saves cycles by eliminating dereferencing, and also reduces cache misses.

128 Bit Integer Keys. The aggregator represents flow keys as two 64 bit integers: the first stores IP addresses; the second stores ports, protocol, and (optionally) physical link ID. This allows the key comparison function to use SSE 4.1 operations to compare a pair of 128 bit keys in only 2 instructions [42].

Lookup Prefetching. The aggregator batches lookups to mask memory latency. It prefetches the hash table slots where each record in the batch is most likely to be before processing, so the memory lookups occur in parallel with the processing of the first few records. Prefetching also compliments linear probing: when a record is *not* in the expected slot, the linear probing algorithm is likely to have placed it in the slot immediately proceeding, which would also be loaded by a prefetch of the expected slot.

5.3 Exporting Flow Records

A separate thread of the aggregator packs the evicted FRs into packets and exports them to collection servers. It also periodically scans the hash table entries and expires all flows that have been inactive for longer than a pre- configured length of time.

5.4 Worst Case Performance

Allocating more PFE memory to the mFR generator decreases collisions and reduces the rate of mFRs sent to the CPU, letting `TurboFlow` scale to higher rates and leaving more headroom for other applications. But how much PFE memory does an operator need to allocate?

Table Size	a	$2 \times a$	$3 \times a$	$4 \times a$	$5 \times a$
$P[eviction]$	.65	.40	.28	.22	.18
Packets : mFR	1.53	2.5	3.57	4.54	5.55

Table 5: Eviction chance with a active flows.

To guide configuration, we derive Equation 2, an expected worst case bound for the mFR stream based on the size of the mFR table in the PFE (T), expected packet rate ($E[p]$), flow rate ($E[f]$), and number of simultaneously active flows ($\hat{a}$). Table 5 lists the expected worst case rates in terms of $\hat{a}$ given a fixed packet rate and flow rate.

The expected worst case rate depends on the probability that an individual packet causes an eviction, which Equation 1 describes. Evictions occur whenever there is a collision. The probability of a packet colliding with a prior entry is equal to 1 minus the probability that all other active flows map to *different* slots than the packet. $\frac{1}{T}$ is the probability of a single flow having the same hash value as the packet, $(1 - \frac{1}{T})$ is the probability of a single flow having a *different* hash value, and $(1 - \frac{1}{T})^{\hat{a}})$ is the probability that all $\hat{a}$ active flows have different hash values than the current packet.

$$P[eviction] = 1 - (1 - \frac{1}{T})^{\hat{a}}) \quad (1)$$

$$E[m] = E[f] + (E[p] - E[f]) * P[eviction] \quad (2)$$

Trace	Capacity	Flow Rate	Packet Rate
Internet Router Link	10 Gb/s	5K - 40K	0.250 M - 0.6M
DC ToR Switch	1280 Gb/s	478K	124.0M
DC Agg. Switch	1440 Gb/s	1291K	134.0M

Table 6: Evaluation workloads.

	Logical Stage	# Tables	# VLIWs	# SALUs	# TCAMs
1.	Compute hash	0	0	0	0
2.	Update key	4	3	4	0
3.	Set evict flag	1	1	0	0
4.	Update features	4	3	4	12
5.	Load buffer pos	1	2	1	3
6&7.	Update evicted buf.	8	2	8	0
	Total	9.38%	2.86%	35.42%	5.21%

Table 7: Tofino pipeline usage for `TurboFlow`.

6 EVALUATION

We implemented `TurboFlow` and used benchmarks, simulation, and analysis to answer the following questions:

- What are the computational and memory requirements?
- How much do optimizations in the CPU help?
- What is the overall monitoring capacity of a switch running `TurboFlow`, and how difficult is it to tune?
- What is the cost of high coverage monitoring with `TurboFlow`?

6.1 Experimental Setup

Evaluation Platforms. We implemented the programmable ASIC and NPU designs of TurboFlow [1]. The programmable ASIC implementation targets the Wedge 100BF-32X [54], a 32x100 GbE switch with a Tofino [64] PFE and an Intel D1517 quad core CPU with 8 GB of RAM. The NPU implementation targets a switch built using a commodity server with a 4x10 GbE NFP-4000 [62] NPU, packaged as a PCIe card, and an AMD Opteron-6272 CPU.

Each implementation had the same mFR aggregator, written in C++, but different mFR generation code, written in a combination of P4 and platform specific languages, for access to hardware features not supported by P4. The Tofino implementation required platform specific code for the stateful operations, while the Netronome implementation used our custom semaphore, which we wrote in Micro-C.

6.2 Benchmark Workloads

We configured `TurboFlow` to produce FRs that included IP 5-tuples and 4 features: packet count, byte count, start timestamp, and end timestamp. We benchmarked it with traces that represent Internet router and data center switch workloads, as Table 6 summarizes.

Internet Routers. We used 8 1-hour long traces from a 10 Gb/s link between core Internet routers [11], collected in 2015. Each trace contains 1 - 2 billion anonymized packet headers, representing over 99% of the packets that crossed the links during the collection periods. To scale resource requirements up to Tb/s rates, we model a router that monitors many 10 Gb/s links with independent traffic flows. We allocate a different segment of the mFR table for each link, and statically load balance mFRs from each link to the CPU buffers. This emulates a scenario where the packet rate, flow rate, and number of active flows, i.e., all the variables in the worst case performance equation, scale linearly with link capacity.

Data Center Switches. We generated packet traces of a simulated data center using YAPS [48], an event based simulator parameterized by the data center traffic statistics reported in [4]. YAPS is based on the simulators used in other recent work [3, 35]. We modeled a 40 GbE two tier datacenter network composed of 144 *end hosts* that generated traffic, 9 *ToR switches* that connected to end hosts, and 4 *aggregation switches* that interconnected the ToR switches.

[1] `TurboFlow` code repository: https://github.com/jsonch/turboflow

6.3 Microbenchmarks

We first measured the computational and memory requirements of the PFE and CPU components of `TurboFlow`.

Tofino PFE. On the Tofino, the mFR generator is guaranteed to run at line rate. The primary questions was how many of the Tofino's computational resources `TurboFlow` requires, which determines how much room there is for other functionality. Table 7 shows requirements for three important resources, based on output from the Tofino compiler. The mFR generator required no more than 36% of any resource, which leaves room for many other functions to process packets in parallel with `TurboFlow`.

Common data plane functions such as forwarding, access control, and mirroring, rely mostly on tables for match-action processing, VLIWs for modifying headers, and TCAM for forwarding rules. `TurboFlow` used under 10% of all of these resources.

`TurboFlow` consumed a larger portion of the Tofino's stateful ALUs (SALUs), which implement the stateful operations to update the mFR table and evicted mFR buffer. Recent prototype data plane applications would also use SALUs [28, 44, 50, 61], if they were implemented for the Tofino. Although not all of these applications may be able to map to the Tofino, we can estimate an upper bound for the number of SALUs they require by counting the number of register reads or writes in their P4 code. By this metric, we estimated that a data plane cache for read heavy key- value stores would require 7 SALUs [44], a Paxos implementation [28] would require 9, and a simple EWMA estimate of link utilization for traffic load balancing [19] could be implemented with 1. Based on SALU requirements, all of these applications could be deployed concurrently with `TurboFlow`.

Table 7 also shows that the hash computation required no additional resources, because we configured the SALUs to address register arrays using the packet key's hash, avoiding the need to precompute it.

NFP-4000 PFE. In an NPU architecture the main computational resources is CPU time, which is multiplexed across all the functions running in the NPU. Running in a single thread of the debugger, we measured an average per-packet cycle count of 3423 for the

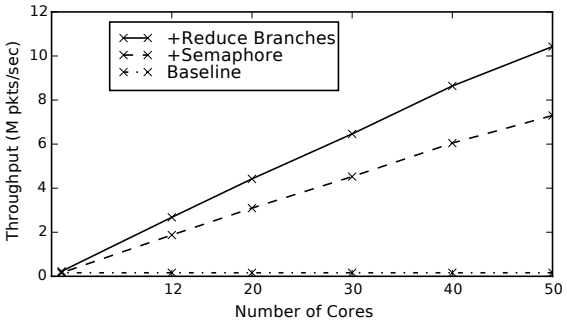

Figure 10: Packet rate throughput on the NFP-4000.

Cycles	Mem Ops.	Hashing	Apply Tables
3423	66.76%	2.13 %	27.81%

Table 8: Single thread cycle count on NFP-4000.

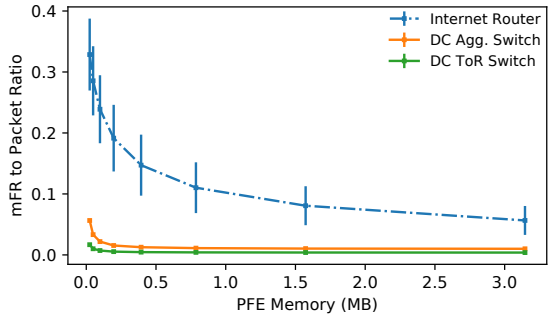

Figure 11: PFE memory vs. ratio of mFRs to packets.

TurboFlow function. As Table 8 shows, memory accesses dominated the cost, which took approximately 200 cycles per read or write. Applying the evict or update tables in the control flow was also expensive, and could be merged with the key update table to optimize future code.

Figure 10 plots the multi-core performance of TurboFlow on the NFP-4000. With all cores of the device enabled, it sustained around 11 M packets per second, enough to saturate the 40 Gb/s interface even with small 300B packets. The figure shows the performance benefit of the finer grained semaphore and table merging optimizations described in Section 4.

Running additional functions alongside TurboFlow would reduce throughput. To estimate how much, we measured the cost of all P4 primitives. The most expensive operations were applying tables (around 1200 cycles each) and reading / writing to register arrays (around 200 cycles each). Based on these measurements, we estimate that running TurboFlow along with 5 custom forwarding tables and another stateful P4 program that requires 9 register reads and 9 register writes, e.g., a Paxos [28] implementation, would cost around 13,000 cycles per packet, resulting in a throughput of around

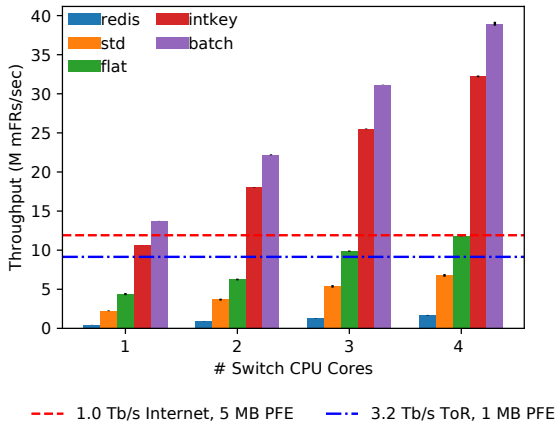

Figure 12: mFR aggregator throughput with optimizations.

3 M packets per second, or 20 - 30 Gb/s with average [11] sized packets.

mFR to Packet Ratio. The PFE reduces the switch CPU's workload by aggregating packets into mFRs. Figure 11 plots *mFR to packet ratios* for the Internet router and DC switch traces, which shows how large the workload reduction is as PFE memory allocated to TurboFlow varies. 1 MB of PFE memory reduced the workload of its CPU component by at least a factor of 10 in all traces. The PFE component was more effective in the data center traces because there were fewer active flows and flow arrival rates were lower. There, 100 KB of PFE memory reduced the CPU workload by a factor of 45 (for aggregation switches) and 135 (for the top of rack switches).

Switch CPU Throughput. Figure 12 plots throughput of the mFR aggregator with different optimizations, averaged over 100 trials. The rightmost bar of each cluster (*batch*) shows throughput with all the optimizations described in Section 5. The mFR aggregator had an average throughput of 13.73 mFRs/s with 1 core, and scaled almost linearly with additional cores; cores 2 through 4 increased throughput by 8.4, 8.9, and 7.79 mFRs/s, respectively.

The leftmost bar (*redis*) in Figure 12 shows a baseline TurboFlow aggregator implemented using Redis [71]. It also scaled well, but was much less efficient than TurboFlow. *The difference is a factor of 20.* The Redis implementation is a strawman that emphasizes the benefit of optimization when constrained to the switch CPU. However, even compared to a much more efficient C++ baseline implementation using a `std::unordered_map` (*std*), the TurboFlow optimizations still provided a 5.67X throughput increase.

The horizontal lines in Figure 12 illustrate *why* the extra throughput matters, with two workloads that require the switch CPU to process around 10M mFRs/s. First, 1 Tb/s of traffic with the multi-link Internet router workload with and 5 MB of PFE memory dedicated to TurboFlow; second, 3.2 Tb/s of DC traffic at a ToR switch (derived by time-accelerating the 1.28 Tb/s ToR trace by a factor of 2.5). The mFR rates for these workloads are computed based on the packet rates of the traces and the packet to mFR ratios shown in Figure 11.

Resource	4 Features	8 Features	Rel. Cost
(Tofino) Tables	18/64	26/64	44%
(Tofino) sALUs	17/44	25/44	47%
(NFP-4000) Cycles	2915	4415	29%
(Switch CPU) mFR Throughput	13.73M	12.57M	8.4%

Table 9: The cost of generating additional features.

PFE Memory	Internet Link	ToR Switch	Aggregation Switch
PCIe load (mFRs to CPU)			
49 KB	22.57 Mb/s	242.93 Mb/s	856.99 Mb/s
98 KB	19.01 Mb/s	173.06 Mb/s	563.17 Mb/s
196 KB	15.39 Mb/s	129.52 Mb/s	397.63 Mb/s
786 KB	9.27 Mb/s	100.04 Mb/s	286.92 Mb/s
1572 KB	7.05 Mb/s	94.65 Mb/s	267.52 Mb/s
Network load (FRs to collector)			
-	1.75 Mb/s	89.71 Mb/s	247.92 Mb/s

Table 10: Communication overheads for `TurboFlow`.

The `TurboFlow` aggregator can support both of these workloads with 1 or 2 cores, which neither the Redis nor C++ baselines could support, even using all 4 cores.

Feature Richness. Table 9 shows the cost of generating FRs with 4 additional features (maximum queue depth, packet size, inter-arrival time, and average queue depth). On the Tofino, this required 8 additional tables and sALUs to widen the mFR table and evicted mFR buffer. On the NFP-4000, the additional memory operations added cycles. For the switch CPU, the new features had a smaller impact on throughput because of prefetching.

PCIe and Network Overhead. Table 10 shows that even with small amounts of PFE memory, the PCIe 3.0 x4 interfaces of the Tofino and NFP-4000 have enough throughput to carry mFRs for over 1000 10 Gb/s core Internet links. Network overhead for exporting the complete FRs to collection servers is even lower, requiring less than $\frac{1}{1000}$ as much bandwidth as the original monitored traffic.

6.4 Monitoring Capacity

Using the benchmark results and statistics from the workload traces, we analyzed the effective monitoring capacity of a Wedge BF32-100X running `TurboFlow`.

Aggregate Capacity. Table 11 summarizes the monitoring capacity of a single Wedge BF32-100X with different amounts of PFE memory and numbers of switch CPU cores, based on the average packet sizes in the workload traces, mFR to packet ratios in Figure 11, and average mFR throughputs in Figure 12. The table shows that `TurboFlow` can scale to terabit rates for both Internet and data center workloads, using reasonable amounts of PFE memory and CPU cores.

Without *any* PFE processing, i.e., the PFE has no mFR table and sends every packet to the CPU as a mFR representing 1 packet,

	Number of CPU Cores			
PFE Memory	1	2	3	4
Internet Router				
0 MB	300 Gb/s	600 Gb/s	900 Gb/s	1200 Gb/s
1 MB	600 Gb/s	1200 Gb/s	1700 Gb/s	2200 Gb/s
4 MB	800 Gb/s	1400 Gb/s	1900 Gb/s	2500 Gb/s
8 MB	900 Gb/s	1600 Gb/s	2300 Gb/s	2800 Gb/s
16 MB	1100 Gb/s	1800 Gb/s	2600 Gb/s	3200 Gb/s
24 MB	1200 Gb/s	2000 Gb/s	2800 Gb/s	3500 Gb/s
Data Center Aggregation Switch				
1 MB	6.4 Tb/s	9.3 Tb/s	13.6 Tb/s	14.4 Tb/s

Table 11: Aggregate monitoring capacity for `TurboFlow`.

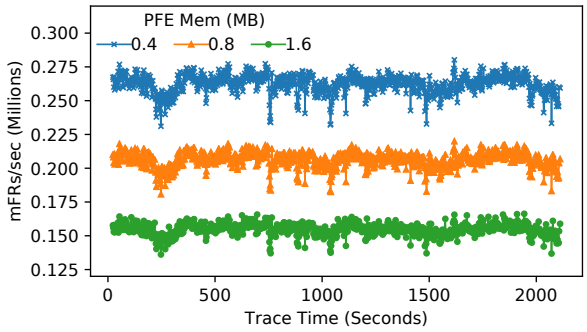

Figure 13: mFR rates over time in 1 core Internet trace.

`TurboFlow` scales to 1.2 Tb/s of aggregate Internet links when using all 4 cores. A small amount of PFE memory, 1 MB, can replace 2 of those cores to reach the same capacity. At the other extreme, the switch can also scale to the same traffic rates using 24 MB of PFE memory and only 1 CPU core.

For perspective, Marple [61], the only other recent PFE accelerated system capable of efficiently generating FRs with these features, is estimated to require an 8 core dedicated server to support a 64x10 GbE switch [61] with Internet scale workloads derived from the same links.

Tuning. To analyze the difficulty of tuning `TurboFlow`, i.e., selecting how much PFE memory to use, which determines mFR rate and thus CPU load, we measured the *stability* of configurations over time and the *safety* of the analytic formulas we derived in Section 5.4.

Figure 13 plots mFR rates during .5 intervals in the 12/2015 core Internet router trace. MFR rates were stable throughout the duration of the trace, and variance was low. Given the stability, it would be practical to tune `TurboFlow` based on a short initial sample of traffic in these workloads.

Figure 14 plots a histogram of the ratio of worst case expected mFR rate to average measured mFR rate, with trials for all 8 2015 core Internet router traces. The ratio is always above 1, demonstrating that the worst case formula in Section 5.4 provided a safe

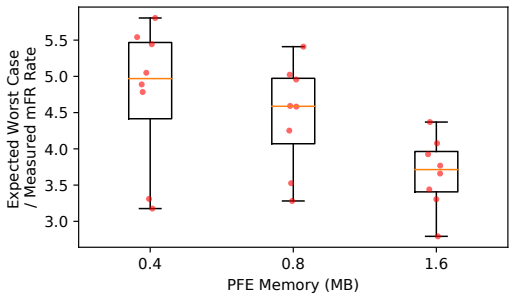

Figure 14: Worst case to measured mFR rates for 2015 Internet router traces.

Workload	Switches	+ Generation	+ Analysis
Equipment Cost (per Tb/s)			
DC ToR	$3600	$3603 (+ 0.1%)	$3642 (+ 1.2%)
DC Agg.	$3600	$3608 (+ 0.2%)	$3702 (+ 2.9%)
Internet	$3600	$3636 (+ 1.0%)	$4059 (+ 12.8%)
Power Cost (per Tb/s)			
DC ToR	150 W	158 W (+ 5.6%)	164 W (+ 10.0%)
DC Agg.	150 W	159 W (+ 6.1%)	174 W (+ 16.7%)
Internet	150 W	163 W (+ 9.2%)	234 W (+ 56.3%)

Table 12: Cost of monitoring infrastructure with `TurboFlow`.

bound in all scenarios. Since switch CPU load is stable over time, operators can use the formulas to find an initial configuration and later tune PFE memory allocation to meet target switch CPU loads.

6.5 Telemetry Infrastructure Cost

We analyzed the cost of high coverage monitoring in a network built from Wedge 100BF-32X switches, focusing on the equipment and power costs of generating and analyzing FRs. We calculated cost based on the number of commodity servers required for the work, normalized by the aggregate capacity of the monitored links, in Tb/s. We compare the cost with `TurboFlow` to a lower bound estimate of the cost with Marple [61] or FlowRadar [50].

6.6 Cost Model

The cost model calculates the *equipment cost* (in dollars) and *power consumption* (in Watts) of monitoring with respect to the capacity of the monitored links (in Tb/s). We assume that cost is continuous, e.g., it is possible to deploy a fraction of a server, and that it scales linearly with the total capacity of the monitored links. Linear scaling is a reasonable assumption because monitoring workload can be statically load balanced across the servers, e.g., based on link ID, and the overhead of transferring FRs across the network is negligible (Table 10).

$$Cost = \frac{FlowRate(w)/Capacity(w)}{ServerTput(t, w, i)/ServerCost} \quad (3)$$

Equation 3 is the cost function, in cost per Tb/s of monitored traffic, for a monitoring task t (either generating FRs or analyzing FRs), using a telemetry system i (either `TurboFlow` or the model of FlowRadar / Marple) on a traffic workload w (Internet router, DC ToR switch, or DC aggregation switch). The numerator describes the workload, normalizing their flow rates by capacity for equal comparison. The denominator expresses the number of FRs that can be processed per cost unit (dollar or Watt), which depends on the monitoring task, the traffic workload, and the cost of the servers.

Raw Cost. We based server cost on a reference server with an Intel Silver 4110 CPU 8 core CPU, which costed approximately $3500 in 2017 and uses 600 Watts under full load.

For perspective, we also reference the cost of the raw switches as a lower bound on the cost of a data plane with no monitoring. Wedge 100 series switches cost around $3600 per Tb/s and have a typical power consumption of around 150W per Tb/s [32].

Server Throughput. We estimate per-core server throughput for FR generation and analysis.

We define *FR generation* work as any processing required to collect telemetry data from switches, convert it into FRs, and copy the records into in-memory buffers for analysis applications to consume. For `TurboFlow`, the processing is simply collecting FRs from switches and copying them to buffers for analysis. We use a conservative throughput of 50M FRs/s, which corresponds to 10 Gb/s of 25B FRs, well within the capacity of a single core [73]. We also factored in `TurboFlow`'s usage of the switch CPU by adding 7.8W to the power cost of FR generation, modeling a scenario where `TurboFlow` fully utilizes the entire 25W CPU to generate FRs for its 3.2 Tb/s of switch links.

Marple and FlowRadar must also do post processing to aggregate data from the switch into complete FRs. We estimate an upper bound on their throughput based on the optimistic assumption that the flow servers will only need to do 1 key- value operation per flow for the post-processing. In practice, this would be higher since both systems export multiple records per FR. We measured the per-core throughput of Redis, widely used scale out key-value at 625K key-value updates/s per core of the reference server, consistent with other benchmarks [71].

Analysis work includes any processing that extracts higher level information from FRs. To estimate the throughput of a FR analysis process, we implemented a simple traffic classifier in C++ using Dlib [46]. The classifier predicts the application that generated a flow record (e.g., https, ssh, dns), using flow features described in prior work [91]. We benchmarked the per-core throughput of the classifier at 4.25 M FRs/s on the reference server.

Workload Profiles. We use the flow rates and capacities listed in Table 6 to normalize cost. For the Internet workload, where flow rate depended on link and month during which the trace was taken, we used a high flow rate of 40K / s.

6.7 Cost Analysis

Table 12 summarizes the modeled cost of high coverage monitoring with `TurboFlow`. The costs of generating FRs was low, under 1 % for equipment costs and 10 % for power costs. The cost of analyzing

Workload	DC ToR				DC Aggregation				Core Internet			
System	TurboFlow		PFE Accelerated		TurboFlow		PFE Accelerated		TurboFlow		PFE Accelerated	
Cost Metric *(Per Tb/s)*	*Unit*	*Power*	*Unit*	*Power*	*Unit*	*Power*	*Unit*	*Power*	*Unit*	*Power*	*Unit*	*Power*
Generation	$3	8 W	$268	44 W	$8	9 W	$645	107 W	$36	13 W	$2880	479 W
Analysis	$39	6 W	$39	6 W	$94	15 W	$94	15 W	$423	70 W	$423	70 W
Total	$42	14 W	$308	51 W	$102	24 W	$740	123 W	$459	84 W	$3303	550 W
TurboFlow Saving	**$265, 36 W per Tb/s**				**$637, 98 W per Tb/s**				**$2844, 466 W per Tb/s**			

Table 13: Cost comparison between TurboFlow and other PFE accelerated telemetry systems.

the FRs, which depends entirely on the analysis application rather than the underlying telemetry infrastructure, was higher. The traffic classifier that we used in the benchmarks did computationally expensive machine learning and, even though it was not designed or efficiency, the overall monitoring cost was still reasonable in data center workloads because of the highly efficient FR generation with TurboFlow. In practice, analysis applications can be optimized for much higher throughput [86], or may not even be necessary, e.g., for auditing.

Table 13 compares the cost of TurboFlow to other recent PFE accelerated telemetry systems. TurboFlow reduced both equipment and power costs of generating FRs by a factor of 5 or more for all workloads. This corresponded to a cost reduction of 3 or more when also running the analysis application.

7 DISCUSSION

7.1 Lessons Learned

TurboFlow is one of the first published systems with an implementation for commodity multi-terabit rate P4 switches. Here, we summarize some of the lessons we learned for future work.

Target Hardware Early. Although P4 is a flexible high level language, there are many hardware-specific restrictions that come to bear in practice. If the ultimate goal of a project is to target hardware, it is useful to consider these restrictions early in the design. The first implementation of TurboFlow was for the reference P4 behavioral software switch [24], and did not fully consider the restrictions of hardware. It was effective in theory but required a complete redesign for either the NPU or ASIC PFE. In proceeding work, we have found that it is most efficient to target hardware from day one. Although there is a learning curve, all current P4 hardware vendors have SDEs and cycle-accurate simulators that make this more straightforward than it may seem.

Decompose Complex Processing. The most challenging restriction in ASIC PFEs, for TurboFlow, was that each position in a stateful array could only be accessed once per packet. This restriction is necessary for the hardware, but likely to be an impediment for many complex packet processing applications. We learned that while the limitation may prevent an entire algorithm from being implemented in the PFE, it is often possible to decompose or redesign parts of it to take advantage of the hardware.

Leverage Switch CPUs. Programmable PFEs are a major change in the architecture of commodity switches. Another change that has not received as much attention is the increased throughput between the PFE and CPU. Both the Tofino and NFP-4000 have PCIe 3.0 links with DMA engines and > 10 Gb/s throughput. This is orders of magnitude more than in traditional switches, where it was on the order of Mb/s [26]. TurboFlow shows that the extra bandwidth makes the switch CPU an valuable asset that can enable powerful new features without requiring new hardware.

7.2 Looking Forward

TurboFlow leverages commodity programmable switches that are coming to market for powerful and cost effective monitoring in next- generation networks. Here, we discuss how TurboFlow may interact with advances that are further out.

Future PFEs. As transistor sizes continue to shrink, PFEs will continue to evolve towards increased flexibility and programmability. A recent example is the proposed dRMT ASIC [16], which implements stateful operations using a pool of memory processors that can be invoked multiple times per packet. This design supports a flexible *run to completion* model, similar to the NPUs, but with the capability to provide throughput guarantees that depend on the number of memory operations per packet. TurboFlow motivates research into how the flexibility can be used for more advanced traffic metrics, such as streaming estimates of complex statistics [25], and in-PFE memory management techniques [61].

Although future PFEs will be more flexible, they are still likely to have memory constraints because SRAM density [27] does not increase as rapidly as network demand [21]. This suggests that TurboFlow, and the more general idea of systems that combine PFE and optimized CPU processing, will remain relevant for future generations of network devices.

Network Growth. Network traffic is predicted to continue growing at an annual rate of around 20-30%, for both data centers and the larger Internet [21]. To meet this demand, switches will likely need to scale horizontally, with additional processing cores and more parallel memory banks. PFE ASICs scale by integrating multiple pipelines that each handle a subset of ports, while PFE NPUs and CPUs scale horizontally by adding cores, co-processors, and memory channels [27]. TurboFlow is well suited to taking advantage of horizontal scaling and can serve as a starting point for building more powerful telemetry systems optimized for higher throughput hardware.

8 RELATED WORK

Network monitoring is important, and TurboFlow builds on many prior works. At a high level, TurboFlow is the first system that can generate *feature rich* and *unsampled* FRs at terabit rates using *only commodity switch hardware*.

Scalable Monitoring. Many previous systems achieved scalability by sampling packets [34, 67, 70, 85], or using sketches to estimate certain statistics [92]. These approaches require less resources in the PFE than TurboFlow, but sacrifice information richness.

An orthogonal approach to scaling is balancing the monitoring workload across multiple devices. CSamp [75], OpenNetMon [88], and other systems [17, 57, 87] all load balance monitoring work across routers or switches based on traffic flow. A similar technique would also reduce workload for TurboFlow switches, but at the expense of sacrificing coverage and flow accuracy.

PFE Acceleration. TurboFlow builds on other recent PFE accelerated telemetry systems [50, 61] with different design goals. These systems are designed for periodic measurement tasks, e.g., a network administrator manually debugging an incast by querying specific switches for statistics about certain flows. TurboFlow, on the other hand, is designed for high coverage and always-on monitoring. To meet these more aggressive design goals, we optimized the entire flow generation process to meet the resource constraints of a switch and analytically bound the expected worst case performance. Intead of optimizing for the switch, Marple and FlowRadar rely on external servers to post process data exported from the PFE, which adds overhead that makes high coverage monitoring cost prohibitive.

Query Refinement. TurboFlow is complementary to concurrent work on network query refinement [39], where the data plane of a network streams an increasingly selective flow of packets to a software processor. The refinement improves scalability of packet level monitoring, which is orthogonal to flow level monitoring. While a packet stream has more detailed information about individual flows, it loses information about all the flows filtered out. TurboFlow accounts for all packets in its information rich flow records, and is lightweight enough to run at all times. The flow records from TurboFlow can help provide context for a packet query or determine how an ongoing query should be refined or changed.

Switch CPUs. Several prior systems have proposed coupling traditional fixed function FEs with switch CPUs, using the CPUs for caching forwarding rules [26, 45, 60], more flexible packet processing [52, 80, 83], or offloading counter processing [59]. In these systems the CPU has a fixed interface to the FE, similar to OpenFlow [55], that allows it to send and receive packets, poll fixed counters, and install forwarding rules. The fixed interface and high cost of forwarding rule installation are obstacles to using these systems for FR generation. The only way for the CPU to offload FR generation work to the FE is by installing per-flow forwarding rules and periodically polling their counter values. This strategy leaves the CPU with too much work because forwarding rule installation rates are much lower than flow arrival rates, e.g., 300-1000 per second [26], compared to >10,000 new flows per second for a single 10 Gb/s link [13]. TurboFlow leverages the increased flexibility of PFEs to implement a custom mFR based interface between the PFE and CPU that allows the work to be partitioned at a finer granularity for better PFE utilization.

CPU Optimizations. TurboFlow is also related to work that optimizes network functions such as lookup tables [30, 93], key value stores [51], software switches [77], and sampled FR generation [29] on general purpose CPUs. There is overlap in some of the optimizations that all of these systems use, e.g., batching is generally effective. To our knowledge, TurboFlow is the first to propose and evaluate the specific hash table optimizations described in Section 5 for the task of FR generation with commodity switch CPUs, which are much less powerful than server CPUs.

Energy Savings. A primary goal of TurboFlow is fine-grained and information rich monitoring at low energy cost. Many other works have demonstrate the practical importance and challenge of reducing power consumption, with energy saving load balancers [40, 47] and architectures [1]. TurboFlow is an orthogonal way to reduce power consumption in networks that use flow monitoring, independent of routing or architecture. Additionally, many systems designed to reduce energy consumption themselves rely on a network monitoring infrastructure, for example, to understand network demand and balance load more effectively. TurboFlow can serve as a platform for adopting such systems without increasing infrastructure cost and opens the door to exploring new designs for energy efficiency based on insights gleaned from unsampled, feature rich FRs.

9 CONCLUSION

High coverage network monitoring with FRs is useful for many applications, but current systems either sacrifice the richness of the FRs or rely heavily on servers, which drives up the cost of the telemetry infrastructure. We overcome this issue with TurboFlow, a FR generator that produces unsampled and feature rich flow records on Tb/s scale commodity programmable switches without requiring *any assistance from servers*. To achieve this goal, we carefully decomposed the FR generation algorithm into components that can be optimized for the two available processing units in a commodity programmable switch: its PFE and CPU. The result is a powerful and efficient system that operates well within the constraints of real switch hardware. Our implementations for commodity switches and evaluation with Internet and data center traces show that TurboFlow can generate information rich FRs, scales to multi-terabit rates, provides a tunable and efficient trade off between PFE memory and switch CPU utilization, and minimizes the cost of telemetry infrastructures without sacrificing accuracy or feature richness. These attributes make TurboFlow a powerful tool for monitoring at any scale.

Acknowledgements. We thank our shepherd, Dejan Kostić, and the anonymous reviewers for their input on this paper. This research was supported by: NSF grant numbers 1406225 and 1406192 (SaTC); ONR grant number N00014-15-1-2006; and DARPA Contract No. HR001117C0047.

REFERENCES

[1] Dennis Abts, Michael R Marty, Philip M Wells, Peter Klausler, and Hong Liu. 2010. Energy proportional datacenter networks. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 338–347.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Data Center Networks.. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, Vol. 7. 19–19.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 435–446.

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

[5] Monowar H Bhuyan, Dhruba Kumar Bhattacharyya, and Jugal K Kalita. 2014. Network anomaly detection: methods, systems and tools. *IEEE Communications Surveys & Tutorials* 16, 1 (2014), 303–336.

[6] Nikolaj Bjorner, Marco Canini, and Nik Sultana. 2017. Report on Networking and Programming Languages 2017. *ACM SIGCOMM Computer Communication Review* 47, 5 (2017), 39–41.

[7] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting bloom filters. In *European Symposium on Algorithms*. Springer, 684–695.

[8] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (July 2014), 87–95.

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

[10] Daniela Brauckhoff, Bernhard Tellenbach, Arno Wagner, Martin May, and Anukool Lakhina. 2006. Impact of packet sampling on anomaly detection metrics. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 159–164.

[11] Caida. 2015. The CAIDA Anonymized Internet Traces 2015 Dataset. http://www.caida.org/data/passive/passive_2015_dataset.xml. (2015).

[12] Caida. 2015. Trace Statistics for CAIDA Passive OC48 and OC192 Traces – 2015-2-19. https://www.caida.org/data/passive/trace_stats/. (February 2015).

[13] Caida. 2018. Statistical information for the CAIDA Anonymized Internet Traces. http://www.caida.org/data/passive/passive_trace_statistics.xml. (February 2018).

[14] Enrico Cambiaso, Gianluca Papaleo, Giovanni Chiola, and Maurizio Aiello. 2013. Slow DoS attacks: definition and categorisation. *International Journal of Trust Management in Computing and Communications* 1, 3-4 (2013), 300–319.

[15] Cavium. 2015. Cavium / XPliant CNX880xx Product Brief. https://www.cavium.com/pdfFiles/CNX880XX_PB_Rev1.pdf?x=2. (2015).

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

[17] Shihabur Rahman Chowdhury, Md Faizul Bari, Reaz Ahmed, and Raouf Boutaba. 2014. Payless: A low cost network monitoring framework for software defined networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 1–9.

[18] Cisco. 2012. 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. 2015. 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. 2016. 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] Cisco. 2018. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html. (February 2018).

[22] Benoit Claise. 2004. Cisco systems NetFlow services export version 9. https://tools.ietf.org/html/rfc3954. (2004).

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

[24] P4 Language Consortium. 2014. Behavioral Model (bmv2). https://github.com/p4lang/behavioral-model. (2014).

[25] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[26] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proc. SIGCOMM*.

[27] Denis C Daly, Laura C Fujino, and Kenneth C Smith. 2017. Through the Looking Glass-The 2017 Edition: Trends in Solid-State Circuits from ISSCC. *IEEE Solid-State Circuits Magazine* 9, 1 (2017), 12–22.

[28] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review* 46, 2 (2016), 18–24.

[29] Luca Deri and NETikos SpA. 2003. nProbe: an open source netflow probe for gigabit networks. In *TERENA Networking Conference*.

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

[31] Nick Duffield, Carsten Lund, and Mikkel Thorup. 2003. Estimating flow distributions from sampled flow statistics. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 325–336.

[32] EdgeCore. 2017. Wedge 100 Data Sheet. https://www.edge-core.com/_upload/images/Wedge_100-32X_DS_R04_20170615.pdf. (June 2017).

[33] Endace. 2016. EndaceFlow 4000 Series NetFlow Generators. https://www.endace.com/endace-netflow-datasheet.pdf. (2016).

[34] Cristian Estan, Ken Keys, David Moore, and George Varghese. 2004. Building a better NetFlow. In *ACM SIGCOMM Computer Communication Review*, Vol. 34. ACM, 245–256.

[35] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 1.

[36] Sharon Goldberg and Jennifer Rexford. 2007. Security vulnerabilities and solutions for packet sampling. In *Sarnoff Symposium, 2007 IEEE*. IEEE, 1–7.

[37] Google. 2018. Google Data Centers Efficiency. https://www.google.com/about/datacenters/efficiency/. (February 2018).

[38] Guofei Gu, Roberto Perdisci, Junjie Zhang, Wenke Lee, et al. 2008. BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection.. In *USENIX Security Symposium*, Vol. 5. 139–154.

[39] Arpit Gupta, Rüdiger Birkner, Marco Canini, Nick Feamster, Chris Mac-Stoker, and Walter Willinger. 2016. Network Monitoring as a Streaming Analytics Problem. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 106–112.

[40] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. 2010. ElasticTree: Saving Energy in Data Center Networks.. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, Vol. 7. 249–264.

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

[42] Intel. 2007. Intel® SSE4 Programming Reference. (2007).

[43] Intel. 2014. Intel Pentium Processor D1517. https://ark.intel.com/products/91557/Intel-Pentium-Processor-D1517-6M-Cache-1_60-GHz. (2014).

[44] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 121–136.

[45] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. 2014. Infinite cacheflow in software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 175–180.

[46] Davis E. King. 2009. Dlib-ml: A Machine Learning Toolkit. *Journal of Machine Learning Research* 10 (2009), 1755–1758.

[47] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. 2013. DENS: data center energy-efficient network-aware scheduling. *Cluster computing* 16, 1 (2013), 65–75.

[48] Guatam Kumar, Akshay Narayan, and Peter Gao. [n. d.]. Yet Another Packet Simulator. https://github.com/NetSys/simulator. ([n. d.]).

[49] Bingdong Li, Jeff Springer, George Bebis, and Mehmet Hadi Gunes. 2013. A survey of network flow applications. *Journal of Network and Computer Applications* 36, 2 (2013), 567–581.

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

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

[52] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. 2011. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks.. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Vol. 8. 2–2.

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

[54] Marketwired. 2017. Barefoot Networks Shares Tofino-based Wedge 100B Switch Designs with the Open Compute Project (OCP). (January 2017).

[55] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev. (CCR)* 38, 2 (2008).

[56] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. 2017. StreamBox: Modern stream processing on a multicore machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 617–629.

[57] Samuel Micka, Sean Yaw, Brittany Terese Fasy, Brendan Mumey, and Mike Wittie. 2017. Efficient multipath flow monitoring. In *IFIP Networking Conference (IFIP Networking) and Workshops, 2017*. IEEE, 1–9.

[58] Jelena Mirkovic and Peter Reiher. 2004. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review* 34, 2 (2004), 39–53.

[59] Jeffrey C Mogul and Paul Congdon. 2012. Hey, you darned counters!: get off my ASIC!. In *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 25–30.

[60] Masoud Moshref, Minlan Yu, Abhishek B Sharma, and Ramesh Govindan. 2012. vCRIB: Virtualized Rule Management in the Cloud.. In *HotCloud*.

[61] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 85–98.

[62] Netronome. 2018. Agilio CX Intelligent Server Adapters Agilio CX Intelligent Server Adapters. https://www.netronome.com/products/agilio-cx/. (2018).

[63] Netronome. 2018. OpenNFP. https://open-nfp.org/. (2018).

[64] Barefoot Networks. 2018. Barefoot Tofino. https://www.barefootnetworks.com/technology/. (2018).

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

[66] Andriy Panchenko, Fabian Lanze, Andreas Zinnen, Martin Henze, Jan Pennekamp, Klaus Wehrle, and Thomas Engel. 2016. Website Fingerprinting at Internet Scale. In *Proceedings of the 23rd Internet Society (ISOC) Network and Distributed System Security Symposium (NDSS 2016)*.

[67] Peter Phaal, Sonia Panchen, and Neil McKee. 2001. *InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks.* Technical Report.

[68] Remi Philippe. 2016. Cisco Advantage Series Next Generation Data Center Flow Telemetry. (2016).

[69] Jeff Preshing. 2013. This Hash Table Is Faster Than a Judy Array. http://preshing.com/20130107/this-hash-table-is-faster-than-a-judy-array/. (January 2013).

[70] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2015. Planck: Millisecond-scale monitoring and control for commodity networks. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 407–418.

[71] Redis. 2018. Redis. https://redis.io/. (February 2018).

[72] Wikipedia contributors. 2018. NetFlow — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=NetFlow&oldid=823922835. (2018). https://en.wikipedia.org/w/index.php?title=NetFlow&oldid=823922835 [Online; accessed 23-February-2018].

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

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

[75] Vyas Sekar, Michael K Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G Andersen. 2008. cSamp: A System for Network-Wide Flow Monitoring.. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, Vol. 8. 233–246.

[76] sFlow. 2009. sFlow Sampling Rates. http://blog.sflow.com/2009/06/sampling-rates.html. (June 2009).

[77] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. 2016. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 525–538.

[78] Danfeng Shan, Fengyuan Ren, Peng Cheng, and Ran Shu. 2016. Micro-burst in Data Centers: Observations, Implications, and Applications. *arXiv preprint arXiv:1604.07621* (2016).

[79] Naveen Kr Sharma, Antoine Kaufmann, Thomas E Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation.. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 67–82.

[80] Alan Shieh, Srikanth Kandula, and Emin Gun Sirer. 2010. SideCar: building programmable datacenter networks without programmable switches. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 21.

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

[82] Chakchai So-In. 2009. A survey of network traffic monitoring and analysis tools. http://www.cs.wustl.edu/~jain/cse567-06/ftp/net_traffic_monitors3/#Section2.1.1.1. *Cse 576m computer system analysis project, Washington University in St. Louis* (2009).

[83] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2016. Enabling practical software-defined networking security applications with ofx. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*.

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

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

[86] D. Tong, Y. R. Qu, and V. K. Prasanna. 2014. High-throughput traffic classification on multi-core processors. In *2014 IEEE 15th International Conference on High Performance Switching and Routing (HPSR)*. 138–145. https://doi.org/10.1109/HPSR.2014.6900894

[87] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. 2010. OpenTM: traffic matrix estimator for OpenFlow networks. In *International Conference on Passive and Active Network Measurement*. Springer, 201–210.

[88] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. 2014. Opennetmon: Network monitoring in openflow software-defined networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 1–8.

[89] Liang Wang, Kevin P Dyer, Aditya Akella, Thomas Ristenpart, and Thomas Shrimpton. 2015. Seeing through network-protocol obfuscation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 57–69.

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

[91] Nigel Williams, Sebastian Zander, and Grenville Armitage. 2006. A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *ACM SIGCOMM Computer Communication Review* 36, 5 (2006), 5–16.

[92] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Vol. 10. 29–42.

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

[94] Daniel Zobel. 2010. 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).