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

Many applications for network operation and management require information about traffic flows and how they are processed throughout the network. Currently, the telemetry systems that support these applications at scale force users to choose between infrastructure cost and information richness. The root cause lies in how the systems cope with high traffic rates: by either sampling or integrating custom hardware in the switch, which limits information richness; or by relying on post-processing with servers, which drives up cost. This paper introduces TurboFlow, a telemetry system for commodity programmable switches that does not compromise on either design goal. TurboFlow generates information rich traffic flow records using only the processing resources available on the switch. To overcome the challenge of high traffic rates, we decompose the flow record generation process and carefully optimize it for the heterogeneous processors in programmable switches. We show that the resulting design allows TurboFlow to support multi-terabit workloads on commodity programmable switches, enabling high coverage flow monitoring that is both information rich and cost effective.

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

Monitoring of network traffic and performance is critical for many applications. Security systems correlate communication patterns across large groups of hosts to detect botnets [39]; debuggers compare statistics from each switch along the path of a problematic TCP flow to diagnose misconfiguration [51]; load balancers analyze per-flow utilization to reduce congestion [2]; and researchers carry out large scale, long term measurement campaigns [4, 74, 78] to guide future work.

For all of the above examples and many other monitoring applications [5, 50, 54, 66, 84, 88], high coverage is important. To function most effectively, or in some cases at all, the applications need information about *all* the TCP/UDP flows in the network and how packets in each one were processed by the switches along their path. Rather than measure traffic and switch state directly, which would not scale, high coverage monitoring applications typically operate on flow records (FRs) streamed up from telemetry systems in the data plane [51, 62, 68]. FRs are appealing because they are compact summaries of fine-grained flows, 2-3 orders of magnitude smaller than the corresponding packets [42], which makes them ideal for large scale collection and analysis.

FRs are powerful for applications, but scaling telemetry infrastructure up to support high coverage FR generation can be expensive. For example, dedicated appliances for FR generation are rated for around 100 Gb/s of traffic [19, 33]. In large networks with thousands of switches [4, 74, 78], each capable of forwarding traffic at multi-terabit rates, the telemetry infrastructure for high coverage FR generation would comprise many racks of servers.

Generating FRs at the switch, e.g., with NetFlow switches [20, 72, 93], is more cost effective, but also more challenging due to higher traffic rates. Current systems sacrifice *information richness* of the 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 allows the CPU to cope with high traffic rates, but reduces accuracy and therefore application 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 performance without sampling, but lock the switch into exporting FRs with fixed, usually simple, features. This limits the applications that can be supported and also reduces their effectiveness [50, 84, 90].

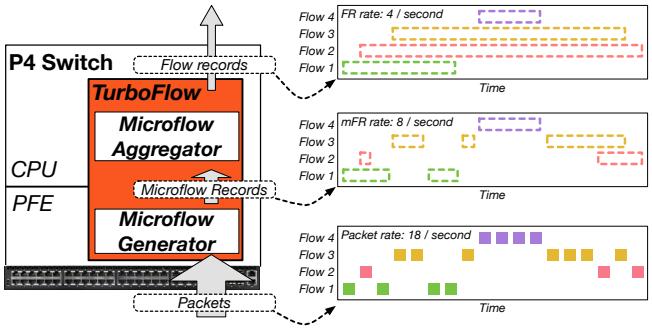

Figure 1: Overview of TurboFlow.

Ultimately, the only telemetry systems capable of generating *unsampled* FRs with *custom features* all place significant workloads on servers [19, 33, 51, 62], 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 flow 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 commodity programmable switches [55], allowing them to serve as drop in replacements for current monitoring switches but with higher quality FRs that better support the hundreds of currently available (or yet to be developed) flow monitoring applications.

Design. Generating information rich FRs at Tb/s rates with the processing resources available in a programmable switch is challenging. Switches have two types of processors: programmable forwarding engines (PFEs), e.g., P4 [8] hardware; and standard CPUs, e.g., low power 2-8 core Xeons. 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 for FR generation. Software on the switch CPU, on the other hand, does not have necessary throughput.

Rather than trying to shoehorn FR generation into either the PFE or CPU, we decompose it into two complementary parts that are well suited for the individual processors. The PFE does preprocessing to reduce the workload of the CPU, while the CPU handles complex logic that the PFE cannot 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 memory requirements and allows us to use a 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 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 PFE's DMA engine transfers mFRs to the 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 collection or analysis servers without any additional processing.

Even though it runs on a switch CPU, the aggregator does not need to sample to monitor multi-terabit traffic with high packet rates. 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 it needs to support is already much lower than packet rate.

Evaluation. We implemented `TurboFlow` on two P4 [8] switches with significantly different PFE architectures. First, the Wedge 100BF-32X [55], a 32x100 GbE switch with a Tofino [65] PFE; second, a 4x10 GbE prototype switch using a NFP-4000 PFE [63]. On both platforms, `TurboFlow` scales to monitor all links with workloads from Internet and simulated data center traces.

Benchmarks show that our aggressive optimizations play a large role, improving the performance of the switch CPU component by a factor of 20. PFE acceleration is also significant, further improving performance by a factor of 10 or more, depending on the workload. Based on analysis of the benchmark results, `TurboFlow` makes high coverage and information rich monitoring 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 flow monitoring* that can be deployed to commodity programmable switches.

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 programmable switches, enabling them to generate information rich FRs for high rate traffic without assistance from servers. Third, a working implementation of `TurboFlow` that demonstrates effectiveness on commodity programmable switches. Fourth, a thorough evaluation and cost analysis of `TurboFlow`, demonstrating that it can enable 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. FRs 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 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 analyze

	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, 41], auditing [50], heavy hitter detection [91], QoS monitoring [62]
Statistics	duration, packet count, byte count, jitter, max packet size	
Network Performance		
Metadata	ingress port, egress port, selected route	Loop and black hole debugging [51], performance queries [62], load balancing [79], network design [74]
Statistics	max queue depth, avg. latency, dropped packet count	

Table 2: Types of FR features and example applications.

data from more than a hand full of links. On the other hand, a FR trace that summarizes the flows with the record format depicted in Table 1 is around 5 GB with 50 million records. This represents a 20X reduction in processing rate and a 200X reduction in bit rate for the application. At these much lower rates, an analysis application implemented on an efficient general purpose stream processing platform [57] could monitor *hundreds* of 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 [39] or other attacks [84] while per-stream packet counts can make it easier to localize misconfiguration in the network core [51] and perform traffic engineering [87]. 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 [91], where the network is configured to measure coarse grained statistics that are only 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 metrics*, 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 rates hundreds of times higher, in the multi-terabit range [55], meaning a switch based FR generator needs to keep up with tens of millions of flow and hundreds of millions of packets per second.

Current systems work around the challenge by sacrificing one of three important design goals: *feature richness*, *accuracy*, or the *cost of FR generation*. We describe these design goals and summarize prior systems below.

2.1 Design Goals

Feature Richness. Feature richness is the capability to include custom statistics and metadata in the FRs. Custom features enable a wider range of applications because different types of applications require different features. For example, consider bot detection [39], QoS measurement [87], and incast debugging [62]. Bot detection systems analyze communication graphs between hosts, which can be generated using only basic features that describe properties of the packets in the flow, for example packet counters, byte counters, and timestamps. QoS measurement, on the other hand, also required statistics about the performance of the network, such as dropped packet counts and path delays. Incast debugging requires completely different features that describe internal operation of the switch, e.g., queue depth.

Besides supporting more applications, feature richness can also improve the effectiveness of many applications, such as machine learning classifiers [90] or anomaly detectors [5] that can take advantage of a wide 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. Inaccurate features directly

	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.

reduce the effectiveness of many applications. For example, consider traffic load balancers [2, 41] that re-route flows to maximize network bisection bandwidth. Inaccurate traffic statistics can cause the load balancers to identify the wrong flows for re-routing, or incorrectly estimate the amount of bandwidth a flow is utilizing [91]. The end result is lower bisection bandwidth.

Inaccurate statistics can also cause security systems to miss anomalies [10], such as the outbreak of an attack.

An orthogonal dimension of accuracy is *flow accuracy*, or what fraction of traffic flows are represented in the FRs. Flow accuracy matters for most applications, but for security applications in particular. For example, if FRs are biased against short flows, as occurs when generating FRs from randomly sampled packets [34], intrusion detection systems can underestimate the magnitude of DDoS attacks [59]. Poor flow accuracy also leaves applications vulnerable to adversaries that specifically try to avoid monitoring [37], e.g., to circumvent intrusion detection or billing.

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 [38] and researchers [1, 41, 48] go to great lengths to minimize these costs, which makes it unlikely that systems with high generation cost would be deployed for high coverage monitoring, regardless of how much they could benefit applications. In practice, operators rely on low 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

FRs have a long history of use [50], and many systems have been developed for FR generation using switches. Prior literature describes earlier systems in detail [42], which were designed for switches with traditional, fixed-function hardware. We classify these systems into two broad categories that have similar general properties with respect to feature richness, accuracy, and generation cost. Table 3 summarizes them, along with recent telemetry platforms that, like `TurboFlow`, are designed to utilize the programmable forwarding engines in modern switches.

Sampling. Sampling systems clone a fraction of packets or flows [34, 67, 75], sometimes along with internal switch statistics, from the forwarding engine to the switch 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. Sampling is widely used in practice because it has low cost and available on many commodity switches [67].

NetFlow ASICs. Some switches use custom hardware to generate FRs [20, 72, 93]. Depending on the design, the hardware may either sample [35] packets, but at a lower ratio than software, or even account for every packet [68]. A common design that can account for every packet is including a FR generation module in the forwarding engine ASIC [68] that stores per- flow statistics in high speed TCAM counters. Older ASICs were limited to simple features, e.g., byte and packet counters [93]. Newer ASICs, such as those in switches designed for the Cisco Tetration platform [68], support additional features including latency, TCP window size, packet size, TTL, and TCP option variation [68]. Custom hardware provides accuracy, since it can account for every packet, but locks the monitoring applications into a fixed set of flow features.

PFE Accelerated. A recent trend in networking is the commoditization of programmable forwarding engines (PFEs)[15, 64, 65] in next generation switches, line cards, and network interfaces. PFEs are emerging now because the chip area and power cost of programmability 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 over a rich set of packet header and processing metadata.

Several recent systems [51, 62] have proposed to leverage their capabilities for telemetry. These systems could be configured to generate FRs, but would have a high generation cost because they rely on post processing at server clusters, to aggregate data streamed from the switch into complete records.

In FlowRadar [51], the PFE encodes per-flow statistics into counting Bloom filters [7], which a server can later decode and convert into FRs. The decoding is expensive, especially for FRs that include features besides IP 5-tuple.

Marple [62] is a system for streaming queries of network performance statistics, which can include queries for FRs. Marple splits the query processing between PFEs and a scale out key-value store, e.g., Redis [71]. The PFE partially computes the statistics requested by the query. It streams updates for each flow to the key-value store, which aggregates the updates together. For efficient generation, the fields in the query are limited statistics and metadata that are *efficiently mergeable* [62], i.e., each update only requires the PFE to send a bounded amount of state to the backing store. The class of efficiently mergeable features is large. However, even with efficiently mergeable features, generating FRs with Marple is still expensive because it relies on scale out key-value stores for post processing. For example, Marple is reported to require around 1 key-value server to service queries for a single 64 x 10 GbE switch [62].

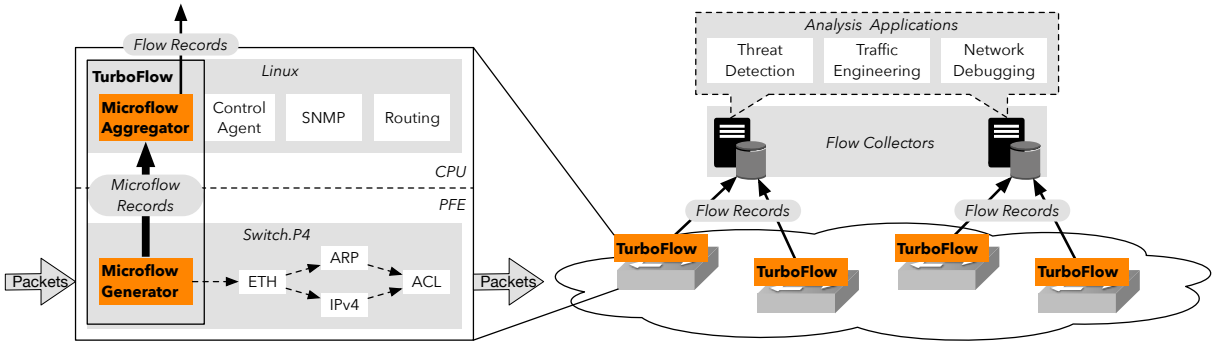

Figure 2: Deployment model of `TurboFlow`.

3 TURBOFLOW OVERVIEW

`TurboFlow` is a FR generator for commodity programmable switches with P4 PFEs [63, 65]. It produces accurate and feature rich FRs for terabit rate traffic without requiring *any* post processing at servers. Figure 2 depicts how `TurboFlow` integrates into a network infrastructure. The ingress pipeline of a switch PFE includes a `TurboFlow` module that generates microflow records (mFRs). The switch CPU runs a `TurboFlow` process that converts the mFRs into FRs and exports them to collection or analysis servers. Operators can customize the features in the FRs, which can also be packed into standard formats, such as IPFIX [23] or NetFlow [22], that are used by many existing flow collectors [82].

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, which can take many cycles because of high memory latencies and expensive key comparison operations. For example, using Redis [71] on the Wedge 100BF- 32X's CPU (a quad core Intel D1517 [44]) provides a throughput of around 500 K packets per second per core – two orders of magnitude lower than necessary.

PFEs, on the other hand, can support key lookups at high rates using on chip memory, but the memory is too small to store FRs for the full set of active flows. Additionally, many PFEs have restricted computational models [81] 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. The PFE produces microflow records (mFRs) that summarize active flows over short timescales. Focusing on mFRs 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 optimized to leverage performance features in modern

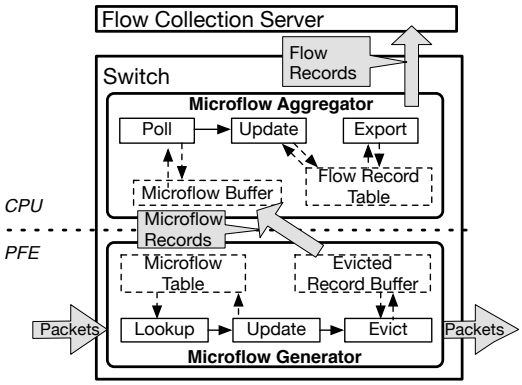

Figure 3: `TurboFlow` architecture.

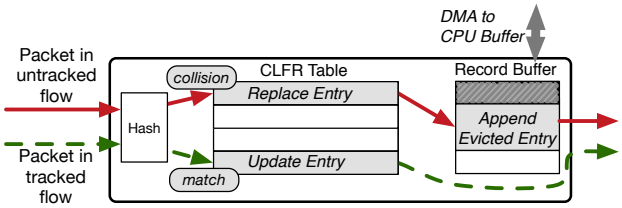

Figure 4: MFR generator data structure.

CPUs. Operating on mFRs instead of packets lowers the rate of key value operations that the CPU must sustain and the optimizations reduce their individual cost to maximize throughput.

4 THE MFR GENERATOR

The mFR generator produces mFRs that summarize burst of packets within flows. MFRs have the same format as FRs, depicted in Table 1. They can include any custom features that the PFE can compute. Capabilities vary by hardware. the most limited PFEs can perform logical operations, simple arithmetic, and estimate floating point operations [79] with limited precision. Additionally, we focus on the large class of efficiently mergeable statistics, as described in Section 2.2 and prior work [62]. This includes the average, minimum, and maximum of any packet header field or metadata associated

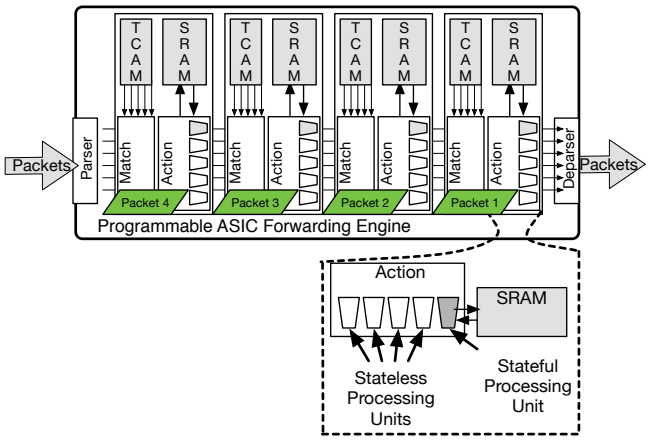

Figure 5: Generalized architecture of a P4 programmable ASIC.

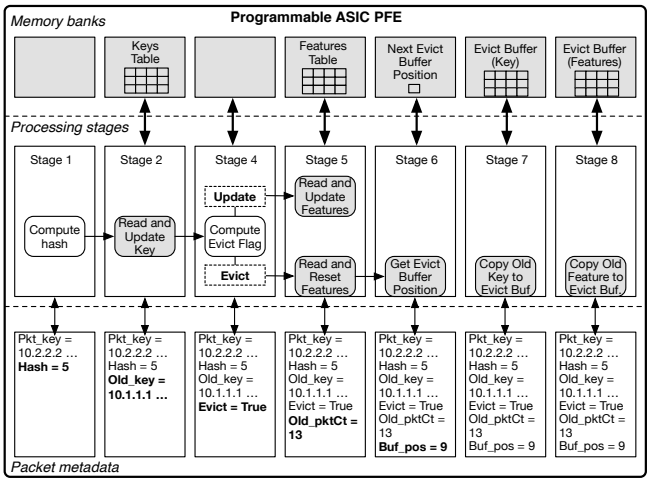

Figure 6: MFR generator mapped to a programmable ASIC.

with processing, all of the features listed in Tables 1 and 2, and all statistics supported by NetFlow ASICs.

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 *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 NPUs [63, 89], which are less restrictive but have lower throughput.

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, i.e., *register arrays* in P4 [8], 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 execute simple stateful programs while accessing the SRAM.

The architecture has two important benefits. 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, or 1 packet per clock cycle, for recent designs [9, 65, 81]. Second, it is straightforward to implement functionality that can be expressed as match + action packet processing because the primitives of P4 map directly to the hardware.

Since the architecture is so specialized for match + action processing, it can be difficult to map more complex functions to the hardware. Sequential operations must be implemented as a sequence of actions in multiple stages. Stateful operations, which `TurboFlow` relies on, are highly constrained. In current ASICs, each register array can only be accessed once per packet, at a single location, to meet the per-stage time budget. Stateful processing units enable programmable atomic updates to the register arrays, e.g., with simultaneous reads and writes or predicated updates. However, the atomic operations must be simple to meet chip space and timing budgets [**?**].

Figure 6 illustrates how the mFR generator maps to the processing stages in a programmable ASIC. It uses register arrays in SRAM banks to store the mFR table and evicted mFR buffer, packet metadata as a scratchpad for decision logic, and P4 tables to define control flow. The pipeline operates on packets in parallel with other forwarding functionality. There are no back branches or loops in the pipeline, which is a requirement of both the P4 language and programmable ASICs. Also, not depicted in the figure, all the persistent state is striped across memory banks. For example, an array of N 13 byte IP 5-tuple keys is actually 4 register arrays: 3 32-bit arrays and 1 8-bit array, each with length N and stored in a separate bank. This allows the pipeline to only need 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 metadata, 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) (through 8) Writes the key and features of the previous flow to the evicted buffers.

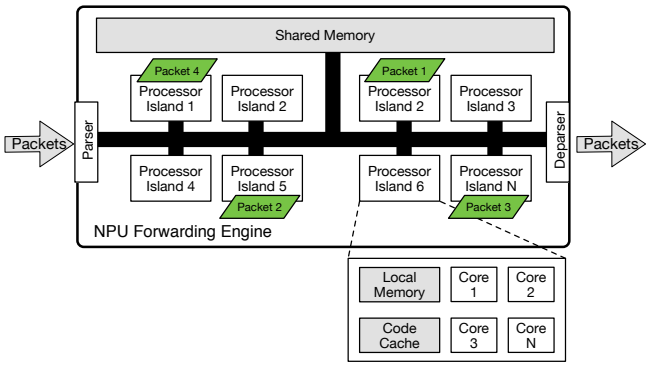

Figure 7: Generalized architecture of a NPU.

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.

Mapping to an NPU. Network processors (NPUs) [64, 89] are an older and more flexible architecture for high throughput packet processing. The trade off is lower maximum throughput than programmable ASICs and fewer compiler guarantees on performance. There are many NPU architectures, the most flexible of which use a pool of RISC cores to process packets, as Figure 7 depicts. The cores are arranged into islands with local SRAM for code and data storage, a large shared off-chip DRAM for forwarding tables and other persistent state, and a high bandwidth switch fabric interconnecting the components.

Unlike programmable P4 ASICs, P4 primitives do not map directly to the hardware of NPUs. Instead, NPUs support P4 with software libraries and toolchains [64] that compile P4 into routines for lower level languages, e.g., micro-C.

Mapping the mFR generator to an NPU requires optimization to maximize throughput. One consideration is minimizing the number of cycles required to process each packet. As Figure 4 shows, we found that some P4 primitives, most importantly applying tables and branching in the outer control flow of the pipeline, can be expensive on NPUs. We optimized the mFR generator to minimize the number of tables and branches in the outer control flow. Figure 8 shows P4-14 psuedocode after optimization, 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 NPU that we targetted, the NFP-4000 [64], has a coarse grained semaphore that locks an entire array. This caused high contention and only allowed one thread to operate at a time. To eliminate contention, we implemented a simple spin-lock semaphore

```
// 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.

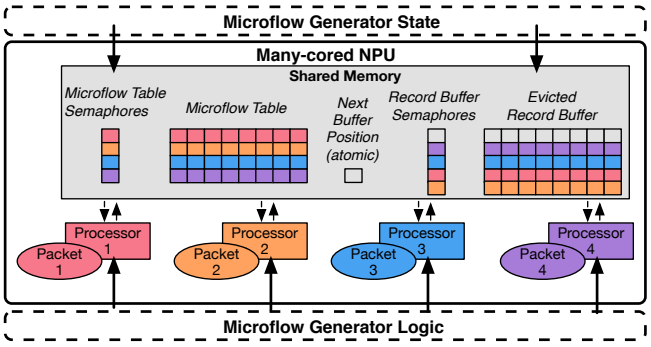

Figure 9: Mapping the mFR generator to a NPU with fine-grained semaphores.

that allows a core to lock a single row of the mFR table or record buffer, as shown in Figure 9.

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 in designing the mFR aggregator is optimizing the key-value data structure to maximize the rate at which it can map mFRs to complete FRs.

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.

To use multiple cores, `TurboFlow` spawns multiple mFR aggregators that each maintain their own buffers. The PFE statically load balances mFRs across the buffers based on key.

5.2 Aggregating mFRs

The aggregator stores FRs for active flows in a hash table. For each mFR, the aggregator either updates the features of an existing FR, inserts a new FR, or, for `TCP` packets with `FIN` flags, copies a FR 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 significantly improved performance.

Linear Probing. `TurboFlow` uses a linear probing hash table [69]. Linear probing has high cache locality, which significantly increases throughput given the small size of individual FRs. When a hash table miss occurs, the next FR to check will likely be in the CPU's cache already.

Flat Tables. The aggregator stores FRs directly in the hash table, i.e., each slot stores a FR, rather than a pointer to a container. This eliminates dereferencing, which saves cycles and further 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 [43].

Lookup Prefetching. The aggregator batches lookups to mask memory latency. It prefetches the hash table slots where each FR 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 the prefetch.

5.3 Exporting Flow Records

A separate thread of the aggregator packs the evicted FRs into packets and exports them to collection or analysis 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 reduces the rate of mFRs sent to the CPU by decreasing collisions, which lets `TurboFlow` scale to traffic higher rates and leaves more CPU cycles for other applications. But how much PFE memory does should an operator allocate to meet a target mFR rate?

To guide configuration, we derive Equation 2, an expected worst case bound for the rate of 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.

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.

The expected worst case rate depends on the probability that an individual packet causes an eviction, which Equation 1 describes. An Eviction occurs when 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 for `TurboFlow`?
- How much do optimizations in the mFR aggregator improve throughput?
- What is the overall monitoring capacity of a switch running `TurboFlow`, and how difficult is it to tune?
- What is the infrastructure 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 targetted the Wedge 100BF-32X [55], a 32x100 GbE switch with a Tofino [65] PFE and an Intel D1517 quad core CPU with 8 GB of RAM. The NPU implementation targetted a switch built using a commodity server with a 4x10 GbE NFP-4000 [63] 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, written 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 10 Gb/s links 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 the workload up to Tb/s rates, we modeled a router that monitors many 10 Gb/s links with independent traffic flows. In `TurboFlow`, we allocated a different segment of the mFR table for each link, and statically load balance mFRs from each link to the CPU buffers. This represented a scenario where the packet rate, flow rate, and number of active flows, i.e., all the variables in the worst case performance equation, scaled linearly with link capacity.

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

Data Center Switches. We generated packet traces of a simulated data center using YAPS [49], 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, 36]. We modeled a 40 GbE two tier data center 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.

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 was compiler guaranteed to run at line rate. The primary questions was how many of the Tofino's computational resources `TurboFlow` required, 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`.

`TurboFlow` used under 10% of the table, VLIW, and TCAM resources, which common data plane functions such as forwarding and access control rely on.

`TurboFlow` consumed a larger portion of the Tofino's stateful ALUs (SALUs). Recent prototype data plane applications would also use SALUs [28, 45, 51, 62]. 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 would 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 [45], 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. We avoided the need to compute it by configuring the SALUs to compute the hash of the packet's key while accessing the register arrays.

NFP-4000 PFE. In an NPU architecture the main computational resources is CPU time, which is shared by all the data plane functions running in the NPU. We measured an average per-packet cycle count of 3423 for the `TurboFlow` mFR generator, using the

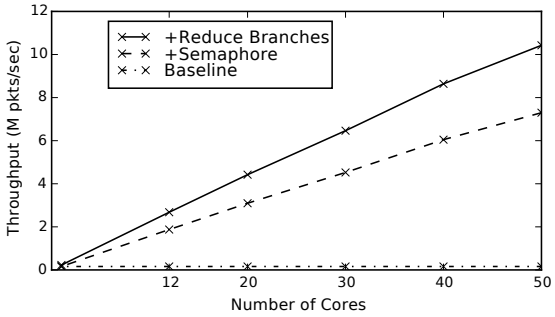

Figure 10: Packet rate throughput on the NFP-4000.

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

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

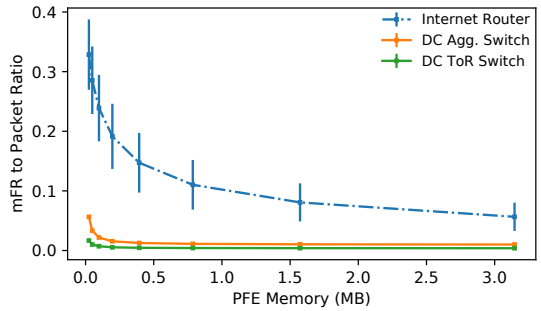

Figure 11: PFE memory vs. ratio of mFRs to packets.

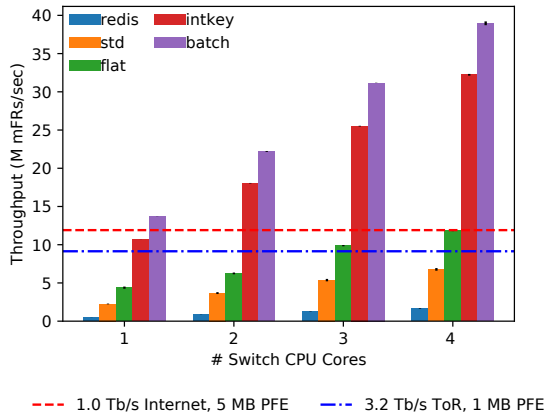

Figure 12: mFR aggregator throughput with optimizations.

NFP-4000's single thread debugger. 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 further 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 finer grained semaphore and branch reduction/table merging optimizations described in Section 4 were important in reaching this throughput, as Figure 10 also shows.

Running additional data plane 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 3 M packets per second, or 20 - 30 Gb/s with average packet sizes around 600 B - 1 KB, which is common [11].

mFR to Packet Ratio. The PFE reduces the switch CPU's workload by aggregating packets into mFRs. We quantify the workload reduction in terms of the *mFR to packet ratios*. Figure 11 plots mFR to packet ratios for the Internet router and DC switch traces. 1 MB of PFE memory reduced CPU workload by a factor of at least 10 in all traces. Workload reduction was more significant 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 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++ 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. The `TurboFlow` aggregator can support both of these workloads

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`.

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 the main bottlenecks were per-mFR operations rather than per-byte operations.

PCIe and Network Overhead. Table 10 shows that even when only using small amounts of PFE memory, data rates to the CPU are low, in the sub 1 Gb/s range. For context, the PCIe 3.0 x4 interfaces of the Tofino and NFP-4000 have theoretical maximum throughputs of 32 Gb/s. 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 and Tuning

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

Aggregate Capacity. Table 11 summarizes the monitoring capacity of `TurboFlow` in terms of Tb/s of link capacity, using different amounts of PFE memory and numbers of switch CPU. We derived these capacities based on the average packet sizes in the workload traces, the mFR to packet ratios in Figure 11, and the average mFR throughputs in Figure 12. The table shows that `TurboFlow` can scale to produce FRs for terabit rate traffic with both Internet and data center workloads, using reasonable amounts of PFE memory and CPU cores.

	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`.

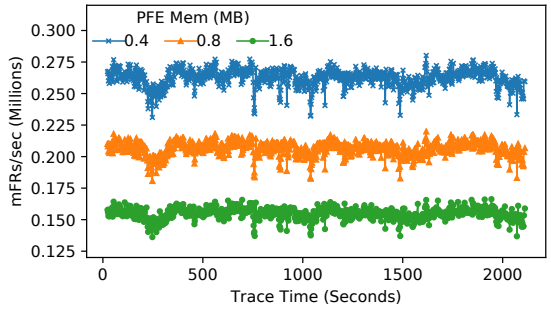

Figure 13: mFR rates over time in 1 core Internet trace.

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, `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 [62], 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 [62] 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 second 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

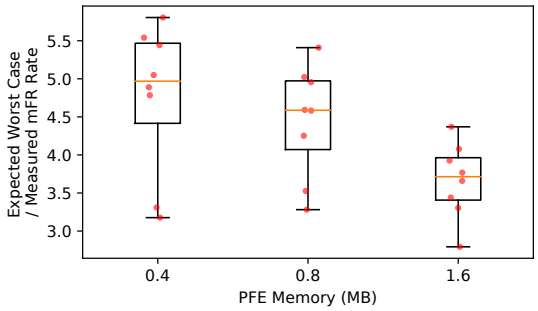

Figure 14: Worst case to measured mFR rates for 2015 Internet router traces.

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`.

core Internet router traces. The ratio is always above 1, demonstrating that the worst case formula in Section 5.4 provided a safe 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

To analyze the cost of high coverage monitoring with `TurboFlow`, we modeled the equipment and power costs of a generating and analyzing FRs in a network built from Wedge 100BF-32X switches and commodity servers. We compared the cost of monitoring with `TurboFlow` to a model that represents a lower bound estimate of cost using either Marple [62] or FlowRadar [51].

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 cost scales linearly with the total capacity of the monitored links. Linear scaling is a reasonable assumption because monitoring work can be statically load balanced across the servers, e.g., based on link or traffic flow, and the overhead of transferring FRs across the network is negligible (Table 10).

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

Equation 3 is the cost function per Tb/s of traffic for a monitoring task t (either generating FRs or analyzing FRs), using a telemetry system i (either `TurboFlow` or FlowRadar / Marple) with a traffic workload w (Internet router, DC ToR switch, or DC aggregation switch). The numerator describes the workload, it normalizes 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 task, the telemetry system, 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 around 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 costed around $3600 per Tb/s in 2017 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 filling the buffer with 25B FRs at a bit-rate of 10 Gb/s, well within the capacity of a single server core [73]. We 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 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 processing servers will only need to do 1 key- value operation per flow. 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 of FRs to extract higher level informatino. To estimate the throughput of a FR analysis process, we implemented a simple traffic classifier in C++ using Dlib [47]. The classifier predicts the application that generated a flow record (e.g., https, ssh, dns), using flow features described in prior work [90]. 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 average flow rate depends on trace, we used the highest rate of 40K / s.

6.7 Cost Analysis

Table 12 summarizes the modeled cost of high coverage monitoring with `TurboFlow`. Generating FRs added under 1 % to equipment cost and under 10 % to power cost. Analyzing the FRs, which depends

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.

entirely on the analysis application rather than the telemetry infrastructure, was more expensive. However, even though the traffic classifier that we benchmarked was unoptimized and did computationally expensive machine learning, the overall monitoring cost was still reasonable because of the highly efficient FR generation with TurboFlow.

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 for all workloads. This corresponded to a cost reduction of > 3, even 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. For future work, we summarize some of the lessons we learned in developing it.

Target Hardware Early. Even though PFEs can be programmed in high level languages like P4, they have many hardware-specific restrictions and extensions. If the ultimate goal of a projects is a PFE implementation, it is useful to consider the lower level details of hardware platform 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 significant optimization for the NPU and a complete redesign for the programmable ASIC. 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 restrictions of the programmable ASIC and, to a lesser extent, the NPU, are obstacles for complex applications. The most challenging restriction for TurboFlow was that in the programmable ASIC, each position in a register array could only be accessed once per packet. We learned that although these restrictions may prevent the implementation of a full system in the PFE, it is often possible to decompose or redesign parts of it to take advantage of the high performance of PFEs.

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 Ahead

TurboFlow leverages commodity programmable switches for powerful and cost effective monitoring that can be deployed to 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 to support 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 additional fine-grained throughput guarantees that depend on the number of memory operations per packet. TurboFlow motivates research into how the flexibility can be used for both more advanced traffic metrics, such as streaming estimates of complex statistics [25], and also more advanced in-PFE memory management techniques.

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 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 future telemetry systems that are optimized for higher throughput hardware.

8 RELATED WORK

TurboFlow builds on many prior network monitoring and telemetry systems. 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 [91]. These approaches require fewer PFE resources than TurboFlow, but sacrifice information richness.

An orthogonal approach to scaling is balancing the monitoring workload across multiple devices. CSamp [75], OpenNetMon [87], and other systems [17, 58, 86] load balance monitoring work across routers or switches based on traffic flow. A similar technique would also reduce workload for switches running TurboFlow, but at the expense of sacrificing coverage and flow accuracy.

PFE Accelerated Telemetry. TurboFlow builds on other recent PFE accelerated telemetry systems [51, 62] 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. On the other hand, TurboFlow 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 [40], 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.

Switch CPUs. Several prior systems have proposed coupling traditional fixed function FEs with switch CPUs, using the CPUs for caching forwarding rules [26, 46, 61], more flexible packet processing [53, 80, 83], or counter processing [60]. In these systems the CPU has a fixed interface to the FE, similar to OpenFlow [56], that allows it to send and receive packets, poll 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, 92], key value stores [52], 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 information rich monitoring at low energy cost. Many other works have demonstrate the practical importance and challenge of reducing power consumption with, for example, energy saving load balancers [41, 48] and architectures [1]. TurboFlow is a complimentary 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 analyze flow records, for example, to understand network demand and balance load more effectively. TurboFlow can serve as a platform to support the necessary monitoring at low cost. Wider availability of information rich flow monitoring can also open the door to more sophisticated algorithms to further improve efficiency.

9 CONCLUSION

High coverage flow monitoring has many applications, but current systems either sacrifice the information richness of the FRs or the cost of the telemetry infrastructure. As a solution, we introduced TurboFlow, a FR generator for commodity programmable switches that produces unsampled and feature rich FRs for multi-terabit rate traffic 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 showed 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 high coverage flow monitoring.

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] Nicolas Fevrier. 2018. Netflow, Sampling-Interval and the Mythical Internet Packet Size. https://xrdocs.github.io/cloud-scale-networking/tutorials/2018-02-19-netflow-sampling-interval-and-the-mythical-internet-packet-size/. (February 2018).

[36] 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.

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

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

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

[45] 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.

[46] 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.

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

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

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

[50] 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.

[51] 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.

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

[53] 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.

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

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

[56] 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).

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

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

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

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

[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] 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.

[87] 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.

[88] 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.

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

[90] 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.

[91] 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.

[92] 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.

[93] 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).