HyperSight: Towards Scalable, High-coverage, and Dynamic Network Monitoring Queries

Yu Zhou, Jun Bi, *Senior Member*, *IEEE*, Tong Yang, *Member*, *IEEE*, Kai Gao, Jiamin Cao, Dai Zhang, Yangyang Wang, Cheng Zhang

***Abstract*—Performing fine-grained and real-time network monitoring is the core logic of various data center operation applications, such as traffic engineering, network troubleshooting, and anomaly detecting. However, the state-of-the-art network monitoring solutions either fall short of completely detecting all network incidents (*i.e.,* congestion), yielding limited monitoring coverage, or introduce large overheads, yielding limited scalability. In this paper, we present *HyperSight*, a network traffic monitor with both high coverage and low overheads. The key idea of *HyperSight* is to monitor networks at the *behavior level* via tracking packet behavior changes. *HyperSight* proposes three designs for behavior-level monitoring. First, to facilitate expressing various network monitoring tasks, *HyperSight* presents a declarative query language based on the streaming processing model. Second, *HyperSight* proposes Bloom Filter Queue (BFQ), a memory-efficient algorithm to empower in-network capability for monitoring packet behavior changes. BFQ can be implemented on commodity programmable switches. Third, to support dynamic deployment and execution of packet behavior change monitoring tasks without interrupting on-service switches, *HyperSight* proposes virtual BFQ to support dynamic query compilation. We build a prototype of *HyperSight* and deploy it on commodity programmable switches. Evaluation results show that *HyperSight* supports a wide range of network event queries and can monitor over 99% packet behavior changes while keeping remarkably low overheads.**

***Index Terms*—Network monitoring, programmable switch, packet behavior**

I. Introduction

Network monitoring is critical for data center network management. In particular, the management tasks, such as traffic engineering [2–6], troubleshooting [7–10], attack detection [11], and network planning [12, 13] require always-on, fine-grained, and real-time visibility of network incidents, such as congestion and throughput degradation. To obtain a completed view of network status, network operators should be able to provide *high-coverage* network monitoring. Furthermore, as data center networks expand rapidly in scale, speed, and traffic volume, operators should also guarantee that network monitoring systems can *scale* with low overheads.

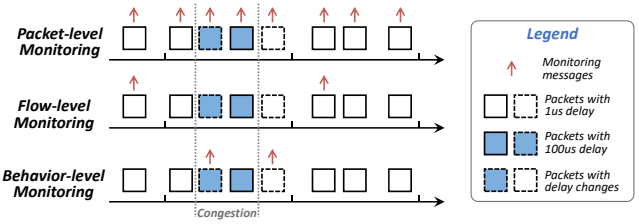


Figure 1. **Behavior-level monitoring *vs.* packet-level and flow-level monitoring. Packet-level monitoring provides high-coverage monitoring but is limited in scalability. Flow-level monitoring yields good scalability but cannot guarantee monitoring coverage. Behavior-level monitoring can reconcile both the monitoring coverage and scalability.**

There have been many off-the-shelf solutions for network monitoring in the literature. Based on the monitoring granularity, we categorize them into two types. First, the *packet-level* monitoring solutions [7–9, 14] display how each packet traverses the networks, but they have to inspect all packets, which introduces unacceptable overheads and comes with significantly constrained scalability. Second, the *flow-level* monitoring solutions present flow-level information via sampling [15–17], aggregation [18–21], or sketching [22–24]. However, the coarse-grained flow-level information might miss some subtle network incidents, compromising monitoring coverage. For example, sFlow [16] performs sampling over packets and potentially misses congestion events experienced by unsampled packets. In summary, packet-level monitoring and flow-level monitoring fail in either scalability or coverage.

Given the limitations of existing solutions, we argue for a shift of network monitoring granularity to the *behavior level*: only monitor the packets that encounter behavior changes. Packet behaviors can include path, throughput, delay, and field modification when packets are forwarded in networks. Behavior changes denote that the behavior of a packet differs from the behaviors of the previous packets in the same flow. Figure 1 shows a comparison of the three monitoring granularities. Furthermore, behavior-level monitoring is motivated by the following observations. First, network incidents are always accompanied by packet behavior changes. For example, congestion incurs latency inflation. Second, the packets experiencing behavior changes only take up a small portion of overall traffic (see §VII-B). Driven by the above observations, we propose to monitor *all packet behavior changes* on switches to provide visibility of network incidents with both high scalability and high coverage.

Yu Zhou, Jiamin Cao, and Dai Zhang are with Institute for Network Sciences and Cyberspace, Tsinghua University, Department of Computer Science, Tsinghua University, and Beijing National Research Center for Information Science and Technology (e-mail: {y-zhou16, cjm18, zhangd15, zhang-cheng13}@mails.tsinghua.edu.cn).

Jun Bi and Yangyang Wang are with Institute for Network Sciences and Cyberspace, Tsinghua University, Department of Computer Science, Tsinghua University, and Beijing National Research Center for Information Science and Technology, and CERNET Network Center (e-mail: {junbi, wangyy}@cernet.edu.cn).

Tong Yang is with the Department of Computer and Science, Peking University, China (e-mail: yangtongemail@gmail.com).

Kai Gao is with the College of Cybersecurity, Sichuan University (e-mail: kaigao@scu.edu.cn).

Cheng Zhang is with Huawei Technologies Co., Ltd. (toer-iczhang@gmail.com)

The conference version of this paper was presented at the IEEE ICNP 2018, Cambridge, UK, September 17, 2018 [1].

In this paper, we present *HyperSight*, a practical system for monitoring packet behavior changes. However, there are three challenges in designing *HyperSight*.

Lack of a convenient way to express various network monitoring tasks. There are various network monitoring tasks, such as load imbalance profiling, congestion detection, and flow path monitoring. Operators need a convenient way to express the tasks they intend to deploy in networks. However, no such network programming language that can specify various monitoring tasks over packet behavior changes. To address this issue, we propose a high-level *Packet Behavior Query Language* (*PBQL*). PBQL introduces stream processing into packet behavior change monitoring and abstracts packet behaviors as dynamic tables. Then, in PBQL, operators can use sliding-window-based primitives, including `distinct` and `duplicate`, to extract packet behavior changes.

Limited switch telemetry capability for monitoring packet behavior changes. Although many switch telemetry methods come into being with the flourish of programmable switches [25] and P4 [26], none of them can be directly applied for monitoring packet behavior changes. On the one hand, some telemetry methods can record packet behaviors of a small portion of flows, which is far from satisfactory for high-coverage monitoring. On the other hand, some telemetry methods record aggregated states (*e.g.,* average delay) of all flows, which are always helpless for monitoring packet behavior changes. Packet behavior change monitoring requires exact states of all flow and can exhaust limited data plane memory due to massive traffic. To address this issue, we propose *Bloom Filter Queue* (*BFQ*), a memory-efficient algorithm for recording packet behaviors on data planes. BFQ can identify packet behavior changes by finding distinct or duplicate behaviors among recently-arrived packets. Moreover, BFQ runs in the data plane entirely and can be deployed into P4-programmable switches, *e.g.,* Tofino [27].

Incapable of implementing dynamic monitoring tasks without interrupting running switches. As network condition changes over time, network operators or applications need to change monitoring tasks running in switches at runtime. However, updating monitoring tasks needs to change P4 programs, which will interrupt on-service switches. As interrupting switches introduces service pause in milliseconds and is intolerable for network operation, supporting dynamic deployment and execution of monitoring tasks is of great importance. However, none of existing monitoring solutions in programmable switches supports dynamic queries. Borrowing from the idea of virtual programmable data plane [28, 29], we propose *virtual BFQ* (*vBFQ*) and make BFQ support dynamic reconfiguration. Operators can dynamically compile network monitoring tasks into running switches with no interruption.

In this paper, we make the following contributions:

- We propose *HyperSight* which exploits packet behavior changes to monitor network incidents while achieving both high coverage and high scalability.
- We provide PBQL, a declarative query language for operators to specify network monitoring intents. PBQL introduces a simple programming model for network monitoring based on stream processing. (§IV)
- We propose BFQ, a new telemetry algorithm compatible with commodity programmable switches to support packet behavior change monitoring. (§V)
- We propose vBFQ to support dynamic compilation of network event queries, enabling update of monitoring tasks without switch termination. (§VI)
- We implement a prototype of *HyperSight* atop Tofino [27] and SmartNIC [30]. The open-source code of *HyperSight* is at [31]. Evaluation results indicate that *HyperSight* supports a wide range of network monitoring queries and can monitor over 99% packet behavior changes while reducing overheads by two orders of magnitude. (§VII)

II. Motivation and Related Work

A. Motivation Cases

In this section, we outline four use cases for always-on monitoring of packet behavior changes.

Network-wide per-flow delay change. *HyperSight* discretizes delay for every flow and reports all discretized delay changes. Monitoring flow delay enables operators to identify the current network congestion status, as network congestion is the main deciding factor for packet forwarding delay in data center networks. Combining delay of different flows, network operators can easily locate hot congestion spots (*i.e.,* caused by incast), which helps evaluate the effectiveness of congestion control algorithms. Furthermore, monitoring flow delay can help operators debug whether application glitch should be attributed to networks. The state-of-the-art solutions can easily monitor the delay of a specific path (*e.g.,* Pingmesh [10]) or a specific flow (*e.g.,* EverFlow [7]). However, performing always-on network-wide delay monitoring is non-trivial for the state of the arts which potentially introduces large overheads.

Network-wide per-flow throughput change. *HyperSight* can report changes that happen to the port-level throughput of all flows. Load imbalance introduces limited utilization of network fabric and impedes the performance of normal traffic. Measuring throughput changes of all links in a continuous manner enables operators to understand whether the load balance scheme (*e.g.,* ECMP) works normally. Furthermore, knowing which flows traverse highly-loaded links is essential for traffic engineering.

Network-wide per-flow packet retransmission. *HyperSight* can monitor packet retransmission via checking whether duplicate TCP packets with the same five-tuples and sequence number exist. On the one hand, in-network packet retransmission detection helps identify random packet loss, which enables operators to take timely mitigation operations such as device rebooting. On the other hand, continuous packet retransmission reveals severe congestion or link failures, calling for traffic redirecting. On-data-plane packet retransmission detection paves the path to real-time traffic redirecting and high-quality network services.

Denial-of-service attacks. Some denial-of-service (DoS) attacks lead to a large amount of the same type of traffic to the same destination to exhaust victims' resources. Thus,

attack packets might have the same destination IP addresses and packet patterns, which distinguish them from regular packets. For example, DNS reflection DoS attacks could result in massive DNS response packets to victims simultaneously, while in ordinary cases, victims only have a small number of response packets. Given the above nature of DoS attack traffic, *HyperSight* can check the duplicate packets among recently-arrived packets. When there is a lot of duplicate packets from too many different source IP addresses to the same destination address, *HyperSight* can reasonably identify that the network undergoes DoS attacks. Furthermore, checking duplicate packets empowers *HyperSight* more generality to monitor various DoS attacks.

Existing solutions can support the above cases for a specific flow or a specific device. However, when applying existing solutions to check packet behaviors for all flows continuously, they inevitably introduce significant bandwidth overheads and processing overheads, posing limited feasibility. Compared to the existing solutions, *HyperSight* can provide always-on high-coverage network monitoring services. Meanwhile, *HyperSight* significantly reduces the monitoring overheads and can scale to large networks with high-volume traffic.

B. Related Work

Network monitoring has long been a challenging task drawing intensive researching interests. First, some tools can provide fine-grained packet-level monitoring. NetSight [9] will generate packet records (called postcards), but NetSight introduces high costs, because it needs to generate postcards for each packet. EverFlow [7] also generate postcards but employs match-mirror and proactive test packet injection to reduce monitoring overheads. However, if EverFlow wants to monitor all packet behaviors, it has to generate postcards for all packets and encounter the same scalability issue of NetSight.

Second, some solutions can provide aggregated flow-level monitoring statistics. sFlow [16] and NetFlow [15] perform sampling over packets. FlowRadar [21], LossRadar [20], and TurboFlow [18] can provide flow-level counters, but they cannot provide network incident information. HashPipe [32], Elastic Sketch [22], SketchLearn [23], OpenSketch [24], and UnivMon [33] employ sketches to monitor heavy hitters, flow size distribution, traffic change detection, and so on. Sonata [34] and Marple [35] also provide language-directed network monitoring. However, Sonata and Marple have to interrupt on-service switches when updating their queries in switches, falling short of supporting dynamic queries.

III. Overview of *HyperSight*

In this section, we will illustrate how *HyperSight* implements queries and how *HyperSight* monitors packet behavior changes (PBC).

A. Workflow and Dataflow of HyperSight

As shown in Figure 2, *HyperSight* is composed of four layers. The first layer consists of queries from various applications, and the queries can be specified with PBQL. The

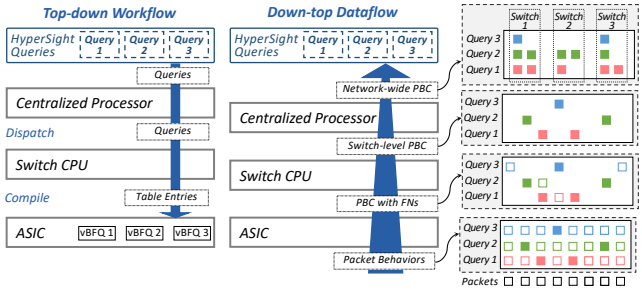


Figure 2. **Top-down workflow and down-top dataflow of *HyperSight*. Solid squares denote packets experiencing behavior changes, while hollow squares denote those that do not experience behavior changes.**

second layer is a centralized processor which collects data from all switches and pushes data to queries. The third layer consists of CPUs of all switches and takes responsibility for cleaning raw data from programmable ASIC and reports data to the centralized processor. The fourth layer consists of ASICs which collect PBC at line rate. Next, we will introduce the workflow of *HyperSight* and how monitoring data of *HyperSight* moves among the four layers, named *dataflow*.

Top-down workflow of *HyperSight*. As shown in Figure 2, the workflow of *HyperSight* starts from the first layer and ends at the fourth layer, *i.e.,* in a *top-down* manner. *HyperSight* could execute multiple queries simultaneously in the same network, and network operators could specify queries with PBQL. Then, the centralized processor dispatches all queries to switches. Next, the switch CPU compiles queries to configurations (*i.e.,* table entries) of the P4 program, and the entries can be dynamically installed into ASIC. Correspondingly, *HyperSight* will instantiate a vBFQ for each query. On the one hand, the top-down workflow supplies language-directed telemetry [35] and enables a flexible way to express various network event monitoring tasks. On the other hand, the top-down workflow prevents collecting redundant data that is undesired by network operations and enhances the overall scalability of *HyperSight*.

Down-top dataflow of *HyperSight*. Before being provisioned to upper-layer applications, PBC data will flow through ASIC, switch CPU, and the centralized processor sequentially, *i.e.,* in a *down-top* manner. First, vBFQ in switch ASIC reports PBC with false negatives (FN). Then, switch CPU can faithfully remove FNs in PBCs and report switch-level PBCs. Last, the centralized processor merges the data from different switches into the same data store and pushes the data to application queries. Such down-top dataflow will reduce the volume of data layer by layer. Leveraging computing power on switch ASIC, switch CPU, and the centralized processor, the dataflow can achieve high PBC coverage while keeping overheads of collecting and transmitting data as low as possible.

B. Architecture of HyperSight

Figure 3 demonstrates the *HyperSight* architecture which comprises two parts. First, the centralized processor provides query interfaces for various applications and collects data from all switches. Second, switches supporting *HyperSight* provide high-coverage and scalable PBC monitoring. *HyperSight* integrates the great programmability of switch CPU

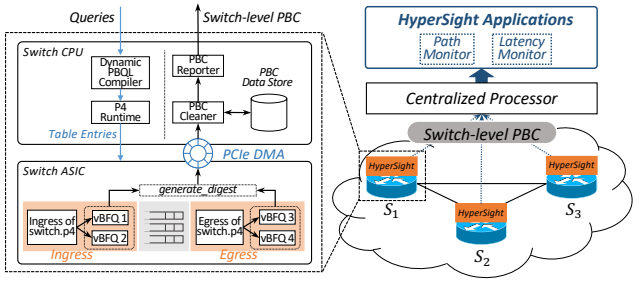


Figure 3. **Architecture of *HyperSight*.**

and the high performance of switch ASIC. On the one hand, *HyperSight* implements an efficient algorithm (BFQ) on ASIC to get PBCs. On the other hand, the switch CPU performs the complex logic of removing FNs and reports FN-free PBCs to the centralized processor. Next, we will introduce the design of switch CPU and switch ASIC.

Switch CPU. There are two functions in switch CPU. First, switch CPU performs *query compilation*. *HyperSight* provides a dynamic compiler for PBQL queries on PBC, and the compiler will convert queries into P4 program configurations, *i.e.,* table entries. Then, the compiler invokes P4 runtime [36], which is a south-bound protocol for controlling programmable ASIC. §VI presents more details about query compilation. In this manner, *HyperSight* supports hot recompilation and reconfiguration of queries without switch service interruption. Second, switch CPU performs *PBC cleaning*. As PBC reported by switch ASIC could have FNs, PBC cleaning is required to keep switch-level PBC redundancy-free and cut down the overheads of reporting PBC. Polling PBCs form PCIe DMA, PBC cleaner records data in PBC data store. PBC cleaner will check whether the reported data correctly identifies PBCs. If there is no FN, PBC cleaner informs PBC reporter which uploads PBCs to the processor.

Switch ASIC. vBFQ can reside at the end of ingress pipelines or egress pipelines in switch ASIC, which is decided by the monitored packet behaviors. For example, if a vBFQ is deigned to monitor queuing delay which can only be observed in egress pipelines, it must be placed in egress pipelines. BFQ is a memory-efficient algorithm which inevitably comes with errors (*i.e.,* FNs). Specifically, BFQ might report some packet behaviors that do not undergo changes. The insight of *HyperSight* to handle FNs is to employ the processing power of switch CPU. Whenever a BFQ finds a PBC, it will invoke *generate_digest* which reports PBC to switch CPU via PCIe DMA. Evaluation results show that BFQ incurs minor overheads on packet forwarding performance (§VII-D). In §V, we will present the detailed design and analysis of BFQ.

IV. Packet Behavior Query Language

To provide a convenient way for network operators to describe their intents on packet behavior change monitoring, *HyperSight* provides a unified language, named packet behavior language (PBQL). In this section, we will introduce the programming model and the syntax of PBQL. Furthermore, we will show the expressibility of PBQL via nine applications.

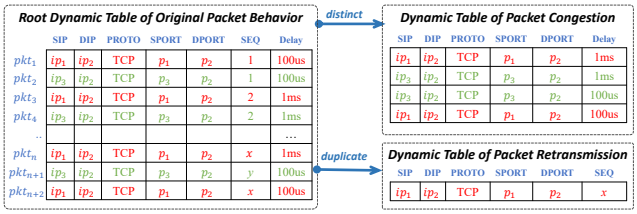


Figure 4. **Dynamic tables of *HyperSight*. Green packets belong to flow 1, and red packets belong to flow 2.**

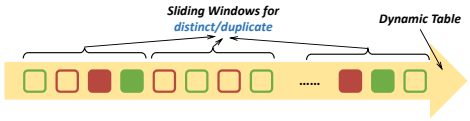


Figure 5. **`distinct` and `duplicate` based on sliding windows over dynamic tables.**

A. Programming Model

PBQL is inspired by distributed stream processing frameworks, such as Flink [37]. These frameworks take a stream of data records as input and make transformation on records to build various applications. Correspondingly, packet behaviors in switches can be abstracted as data records in stream processing framework, and we use *dynamic tables* to denote the stream of packet behavior records. In this part, we will introduce *HyperSight*'s programming model, which includes dynamic tables and the two operations over dynamic tables.

Dynamic table. In Figure 4, columns of a dynamic table are the fields that can identify packet behaviors, such as destination IP address (*SIP*) and delay (*Delay*), and see §IV-B for a completed packet behavior field list. A dynamic table has an infinite number of rows, each of which is a packet behavior record. The left part of Figure 4 shows a root dynamic table that stores original behaviors of each packet. *HyperSight* generates a dynamic table for each query. In essence, *HyperSight* transforms the root dynamic table to the dynamic tables corresponding to different queries. Next, we introduce two transformations to monitor PBCs.

Table I
Packet Behavior Fields.

Category	Field	Syntax	Pipeline
Header	Header Field	$header.field$	Ingress and Egress
Path	Ingress Port	in_port	Ingress and Egress
	Egress Port	eg_port	Ingress and Egress
	Queue	qid	Ingress and Egress
Performance	Queue Length	$queue$	Egress
	Delay	$delay$	Egress
	Throughput	$throughput$	Egress

Table II
PBQL Syntax.

f	$\in$ Packet Behavior Fields	
V	$\in \mathbb{N}$	Field Value
V^m	$\in \mathbb{N}$	Field Value Mask
W	$\in \mathbb{N}$	Window Size
F	$::= f \mid f \ \& \ V^m$	
T	$::= " < " F \ \{, F\} " > " \mid *$	Keys
C	$::= " < " f \ \{, f\} " > " \mid *$	Columns
DIS	$::= distinct(C, T, W)$	
DUP	$::= duplicate(C, T, W)$	
Q	$::= DUP \mid DIS \mid Q_1.Q_2$	Statement
OP	$::= Q.start(V, V^m\{, V, V^m\}) \mid Q.end()$	Query Operations

Table III
EXAMPLE APPLICATIONS EXPRESSED BY THE PACKET BEHAVIOR QUERY LANGUAGE.

Applications	Queries
Large flow	Q_1=`distinct` (<flow>, <flow, tcp.seq&0xFFFF8000>).`duplicate` (<flow>, <flow>)
Flow path	Q_2=`distinct` (<flow, sid, in_port, eg_port>, <flow, sid, in_port, eg_port>)
Loop freedom	Q_3=Q_2.`duplicate` (*, <flow, sid, eg_port>)
Flow delay change	Q_4=`distinct` (<flow, sid, delay>, <flow,sid, delay&0xFC00 >)
Flow congestion	Q_5=`distinct` (*,<flow, sid, delay&0x8000>) .`duplicate` (*, <flow>)
Throughput change	Q_6=`duplicate` (<ip.dip>, <ip.sip, ip.dip, throughput&0xFC00>) .`distinct` (*, <ip.dip>)
Packet retransmission	Q_7=`duplicate` (<flow,tcp.seq>,<flow,tcp.seq>). `distinct` (*,<flow>)
DNS reflection attack victims	Q_8=`duplicate` (<ip.dip>,<ip.sip,ip.dip,dns.address>) .`distinct` (*,<ip.dip>)
Packet modification	Q_9=`distinct` (<flow,modified_fields>,<flow,modified_fields>)

Sliding-window-based `distinct` and `duplicate`. Intuitively, dynamic tables could support various streaming processing primitives like *map*, *reduce*, and *join*. To monitor packet behavior changes, *HyperSight* proposes two dedicated primitives shown in Figure 4. First, `distinct` is the same as the distinct primitive of standard SQL and extracts distinct elements in the dynamic table. Second, `duplicate` is contrary to `distinct` and extracts the duplicate elements in the dynamic table. However, as a dynamic table could have an infinite number of packet behavior records, performing `distinct` or `duplicate` over the whole dynamic table is impossible. Therefore, `distinct` and `duplicate` are executed over sliding windows and extract distinct or duplicate packet behaviors for the recently-arrived packets. Intuitively, `distinct` extracts the behaviors of a packet when its behavior is distinct among the recently W packets (W is the window size), and `duplicate` works in the similar manner.

B. Packet Behavior Query Language

Based on the above programming model, *HyperSight* proposes PBQL for network operators to specify PBC monitoring tasks. PBQL enables operators to express queries over fields that represent packet behaviors, which are referred to as *packet behavior fields*. Next, we will introduce all packet behavior fields and details of the language syntax.

Packet behavior fields. As shown in Table I, there are three types of packet behavior fields. The first type is related with *packet headers* and contains fields from parsed headers. In this paper, we only support some widely-used header fields, including destination IP, source IP, TCP/UDP ports, and so on. The second one is related with packet forwarding paths and includes forwarding ports and queues. The third one is related to packet forwarding performance.

Syntax. Table II shows the basic syntax of the query language. Both `distinct` and `duplicate` have three parameters. The first parameter C denotes the columns of the dynamic table. The second parameter T specifies the keys for the primitives. For example, if we want to find distinct destination IP addresses, T should be $< ip.dip >$. Some fields, such as delay, in T can be discretized via the mask operation (&). The third parameter W is the window size. Furthermore, PBQL supports query refinement, and operators can utilize existing statements to implement more complex queries. Furthermore, PBQL also provides `start` and `end` primitives for operators to manipulate their query at runtime dynamically. Furthermore, operators can designate a query to monitor specified flows through the parameters V and V^m. `start` uses V and V^m to create ternary match rules of five-tuples.

C. Expressibility

To demonstrate how PBQL comes into force in reality, Table III shows nine example applications. For brevity, we use *flow* to denote five-tuples throughout the paper. These applications enhance network monitoring in various perspectives. For example, with the ingress port and the egress port, we can get the forwarding path of each flow in Q_2. With Q_1, we can also attain the large flows whose sizes are over 2^{15} bytes. With PBQL, network operators can conveniently specify packet behavior change monitoring tasks. We should claim that it is impossible for *HyperSight* to implement the completed logic of the applications with PBQL. The goal of *HyperSight* is to provide an interface for operators to attain the data required by these applications from networks.

V. DESIGN OF BLOOM FILTER QUEUE

In this section, we first show the technical background and constraints of P4 and programmable switches. Then, we will demonstrate the detailed design of BFQ. Table IV summarizes the symbols used by this paper.

A. Preliminary to P4 and Programmable Switches

PISA and P4. The protocol independent switch architecture (PISA) [38] is a typical programming model for programmable switches. PISA is composed of several reconfigurable components. First, PISA has *parser* and *deparser* to encode and decode headers with arbitrary protocol formats. Second, PISA processes packets with a pipeline of stages, each of which has multiple match-action tables and memory resources, such as SRAM and TCAM. Third, a table has various match fields (*e.g.,* destination IP address) with types (*i.e., exact*, *lpm*, *range*, and *ternary*). Forth, a table has multiple *compound actions* constructed by primitive actions, such as *modify_field*. Last, each stage has a fixed amount of stateful components,

Table IV
SYMBOLS USED THROUGHOUT THE PAPER.

Symbol	Description
K	The number of arrays in a bloom filter
M	The number of cells in an array
S	The number of bloom filters in a BFQ
N	The number of packets in a block
W	The window size, $W = (S-1)N$
$H_1, ..., H_K$	Hash functions for arrays

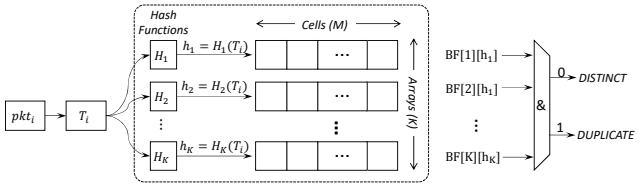


Figure 6. **Processing a packet with a Bloom Filter.**

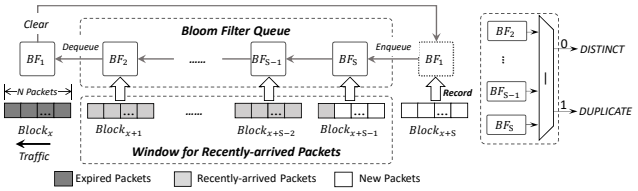


Figure 7. **Bloom Filter Queue design to monitor packet behavior changes.**

including registers, counters, and meters. Only registers can be read and written by PISA in a transaction manner while the other two only support either reading or writing operations.

P4 [26] is a data plane programming language. Network operators can develop P4 programs to specify packet processing behaviors of the above PISA components. The lifecycle of a P4 program has two phases. At *compile time*, the P4 compiler transforms P4 programs to executable code and generates control API. Then, at *runtime*, PISA switches process packets according to P4 programs. Meanwhile, the controller populates match-action table entries through control API.

Constraints of PISA. To guarantee high-performance packet forwarding, PISA inevitably introduces programmability compromise with constraints for complex logic. We present some of the PISA constraints, which drive the design choices of BFQ. The first one is *constrained access to stages*. In PISA, each packet can traverse tables and stateful components only once. Thus, we can not implement the logic (such as minimal calculation) requiring multiple accesses to states. The second one is *constrained operations on registers*. Registers are crucial to implementing stateful packet processing logic on data planes, but PISA only supports transactional register operations including read, write, read-write, write-read and so on. The constrained transactional operations prevent implementing complex logic over states stored in registers.

There are some off-the-shelf Bloom Filter variants, including SBF [39], RSBF, BSBFSD, and RLBSBF [40], which own the similar capability with BFQ, *i.e.,* finding distinct or duplicate items in infinite streams. However, none of them can be implemented on PISA, as they need either multiple times of access to same stages or non-transactional operations over registers. Therefore, the limited feasibility of existing solutions on PISA motivates the design of BFQ.

B. Bloom Filter Queue

In this part, we first demonstrate the philosophy of monitoring packet behavior changes. Next, we will briefly introduce Bloom Filter (BF) which can find distinct or duplicate elements in a data set with finite elements. Last, we show the design of BFQ that can find distinct or duplicate elements in the data stream with an infinite number of elements.

Philosophy of Monitoring Packet Behavior Changes. An intuitive approach to monitoring packet behavior changes is to track packet behaviors of each flow and compare packet behaviors of previous packets with the incoming packets. However, this approach needs to record states for each flow and consumes unacceptable data plane resources.

Given the limitation of the above approach, we propose to check and record the *appearance* of packet behaviors for incoming packets. If some packet behavior never appears, we

Algorithm 1: Pseudo-code for BFQ

```
1  pkt_counter ← 0 ;
2  foreach pkt entering a switch do
3      pkt_counter ← pkt_counter + 1 ;
       /* block_id denote the current block ID for
          pkt.                                      */
4      block_id ← pkt_counter / N;
       /* bf_id and next_bf_id denotes the current
          and next BF ID.                           */
5      bf_id ← block_id mod S;
6      next_bf_id ← (block_id + 1) mod S;
7      report_flag ← 0;
8      Get a T for pkt;
9      for 1 ≤ k ≤ K do
10         h_k ← H_k(T) mod M;
11         for 1 ≤ s ≤ S do
12             if s ≠ next_bf_id then
13                 if BFQ[s][k][h_k] = 1 then
14                     report_flag = report_flag + 1;
15                     Break;
16         BFQ[bf_id][k][h_k] ← 1;
17         BFQ[next_bf_id][k][pkt_counter mod M] ← 0;
18     if report_flag < K then
19         Generate a digest;
```

can identify a packet behavior change, which is similar to finding *distinct* packet behaviors. We can record packet behavior appearance with BF, which is a probabilistic algorithm with high memory efficiency and consumes bounded memory resource. However, BF can only find distinct packet behaviors for finite packets, but there could be infinite packets passing through the switches. Moreover, an error might happen to BF when a packet behavior changes back to a previously existing one. To overcome the limitation of BF, we propose BFQ, which tries to find distinct behaviors for recently-arrived packets, or in a sliding window of packets. BFQ can find PBCs even when the behavior of the incoming packet change back to the behavior of previous packets that are outside the sliding window. There is a risk that BFQ might fail when the behavior of the incoming packet changes back to the previous packet behavior inside the window. Luckily, this risk can be relieved with careful adjustment of the sliding window size.

Bloom Filter. First of all, we show how a BF finds distinct packet behaviors. As shown in Figure 6, a BF comprises K arrays, each of which comprises M cells. The behavior of packet pkt_i is T_i. Then, BF generates K positions (h_1, ...) through K different hash functions over T_i. Afterwards, BF can get K cells ($BF[1][h_1]$, ..., $BF[K][h_K]$) for T_i. In BF, a cell contains only one bit. If all the K cells are 1, BF marks T_i as duplicate (*i.e.,* T_i has been seen before). Otherwise, BF marks pkt_i as distinct (*i.e.,* T_i has never been seen).

Bloom Filter Queue. After presenting BF, we illustrate how packets sequentially traverse BFQ. As shown in Figure 7,

BFQ is composed of S BFs. BFQ sequentially divides packets into fixed-size blocks, each of which has N packets. Each BF exclusively records behaviors for a block of packets. For example, BF_1 records behaviors for $Block_x$. Every packet queries recent $S-1$ BFs to check whether its behavior has been recorded. If all BFs mark this packet as distinct, BFQ has never seen the packet behavior in recently-arrived packets. Then, BFQ marks this packet as distinct and report packet behaviors. Otherwise, BFQ marks the packet as positive and does not report. Meanwhile, to prevent influences from expired packets, BFQ should clear all cells in the dequeued BF, *i.e.,* BF_1. When all packets in the current block ($Block_{x+S-1}$) complete, the new block ($Block_{x+S}$) will be shifted into the window, and its BF (cleared BF_1) will be enqueued at the same time. Similar to the above procedure, BFQ dequeues BF_2 whose block becomes expired.

BFQ maintains a window which slides at the block level and always scans recently-arrived packets. The window size can be calculated by $(S-1)N$. For every packet, BFQ checks all BFs except for the oldest BF. If outputs of all the checked BFs are distinct, BFQ classifies this packet behavior as distinct (never seen in the recently-arrived packets), otherwise duplicate. We provide a completed description of BFQ in Algorithm 1.

For BFQ, there could be two types of errors, *false positive (FP)* and *false negative (FN)*. The FP is referred to that a distinct packet behavior is classified as the duplicate packet behavior, while the FN is referred to that a duplicate packet behavior is classified to be distinct. We name the possibility that FP and FN happen as *false positive rate (FPR)* and *false negative rate (FNR)* respectively. We present a detailed analysis of FPR and FNR of BFQ in §VII.

Report messages of BFQ. Instead of forwarding the whole packets, BFQ only reports the column fields for each query to the switch CPU with *generate_digest* which is general primitive action supported by almost all P4 targets. As the packet behavior fields only occupy tens of bytes and are much smaller than the packet sizes, BFQ takes up a small bandwidth of the ASIC-CPU PCIe channel. To further optimize the workload of switch CPU, *HyperSight* proposes to buffer BFQ reports on the data plane and forward reports in batches, which borrows the insight of *Flow [19]. Intuitively, In this manner, *HyperSight* could effectively reduce the number of messages between ASIC and CPU, improving CPU efficiency.

VI. Dynamic Compilation

With network conditions changing over time, operators might need different queries at different time over different flows. In other words, *HyperSight* should be able to support dynamically starting and terminating queries. To achieve this goal, we propose to support dynamic compilation of PBQL in this section. First, we explore the language elements that differ among queries, which is referred to as dynamics in the query language and determines the design space of dynamic compilation. Next, we show the design of a BFQ extension that supports reconfiguring PBQL dynamics without disrupting on-service switches. As we borrow the idea of programmable data plane virtualization [28, 29, 41], we name the extension as *virtual bloom filter queue* (*vBFQ*).

A. Dynamics of PBQL

In this part, we outline the dynamics in the query language. The primary design goal of dynamic compilation is to make these dynamics dynamically-reconfigurable.

Types and numbers of query statements. Each query has different types, *i.e., distinct* and *duplicate*. Furthermore, as network operators might write refinement queries, implying that one query might have multiple query statements, just like Q_3, Q_6, and Q_8 in Table III. Thus, vBFQ should support different types and number of query statements.

Columns and keys. In one query, network operators can specify different packet behavior fields as the columns (C) and keys (T). Furthermore, packet behavior fields in keys need *mask* operations. Therefore, vBFQ should be able to select packet behavior fields to construct columns and keys.

Parameters of BFQ. There are four parameters of BFQ, including the number of cells per array (M), the number of arrays per BF (K), the number of BF (S), and the number of packets per block (N). On the one hand, N and S jointly determine the window size W, *i.e.,* $W = NS - N$. On the other hand, M, K, and S jointly determine the memory occupied by vBFQ. To be able to allocate memory and windows for different queries at runtime, vBFQ should support dynamically changing the four BFQ parameters.

B. Virtual Bloom Filter Queue

Figure 8 shows the overall design of vBFQ. In the P4 pipeline, we allocate multiple physical arrays, each of which has a fixed amount of cells and a hash function. As shown in the figure, physical arrays work as a resource pool, and we can allocate registers from the pool for different vBFQ. Furthermore, in Classification, we use a table to perform ternary matching over five tuples. Thus, Classification enables operators to perform monitoring tasks with various granularities. Moreover, vBFQ is functionally equal to BFQ without any compromise or wastage of register resources. Next, we show the design of vBFQ that supports the dynamics of PBQL.

Types and numbers of query statements. The two types of query statements, *i.e.,* distinct and duplicate, have opposite functions. As the query results of vBFQ are enumerable, we make a look-up table to enumerate all results and determine whether to perform digest generation. For example, when K = 3 and S = 3, then there will be 2^9 kinds of results, and the look-up table should have 512 entries for each query. Currently, we use SRAM to perform exact match over query results, and we can also employ ternary match to reduce the number of table entries at the cost of consuming scarce TCAM. With the look-up table, we can dynamically configure the query type. To support query refinement, we install the structure shown in Figure 8 in ingress pipelines and egress pipelines. Currently, each structure only supports one vBFQ for one query, thus switch ASIC can support up to 2 query statements. To remove such constrain, we further employ switch CPU to run the remaining query statements on software. In this manner, *HyperSight* has no limit in terms of the number of statements in one query.

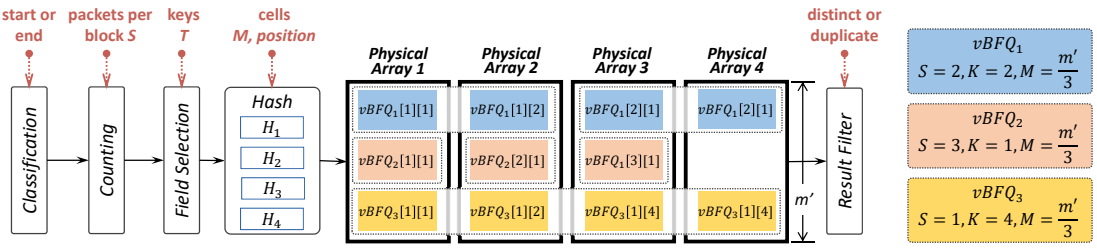


Figure 8. **Design of virtual Bloom Filter Queue.**

Columns and keys. We use the field list of P4 to store all packet behavior fields in Filed Selection. There are two corresponding field lists, one for columns and the other for keys. To be general for column field selection, we report all packet behavior fields in the column field list. As for key field selection, we perform masked modification over fields. Take $modify\_field(delay, delay, mask)$ as an example (assume delay has 8 bits). If keys do not include delay, the mask should be 0, and delay will be 0. Otherwise, the mask is 0xFF. If operators want to bin delay with 16ns, the mask should be 0xF0. After the mask operation, the key field list can be used as the keys of Hash operations. In this manner, *HyperSight* support flexible column and key selection.

Parameters of BFQ. In Counting, we maintain a packet counter for each query, and can automatically change current BF in BFQ according to N. In Hash, we use *modify_field_with_hash_based_offset (dest, offset, field_list_calc, M)* to get the cell index for each array. $offset$ denotes the position of vBFQ in physical arrays, $field\_list\_calc$ contains the key field list attained in Field Selection, and M is the number of cells per array for the vBFQ. In this manner, we can flexibly allocate memory for vBFQ from the physical resource pool. Furthermore, we use a table in front of each physical array to determine operations over the physical array for each query. There are three kinds of operations, doing nothing, querying the array, and clearing the array. With the controlled operations, we can easily set S and K for each vBFQ. Based on the above methods, we can dynamically adjust all the four parameters of vBFQ.

C. Limitations of vBFQ

The design of vBFQ can never support dynamic reconfiguration of all elements and encounters two constrains. On the one hand, vBFQ cannot add new packet behavior fields dynamically. The column field list and key field list include all behavior fields. When intending to add newly-found fields into the two lists, operators have to re-load the P4 program. On the other hand, vBFQ cannot allocate or release memory resource of physical arrays dynamically. The amount and size of physical arrays are fixed after the P4 program is deployed. When operators want to install new physical arrays or extend the physical array, they have to re-load the P4 program. Fortunately, the above cases are infrequent for real-world networks and are tolerable for *HyperSight*.

VII. Evaluation

Prototype and setup. We built a prototype of vBFQ with 1000 lines of P4 code, and we deployed the prototype on two P4 targets, *i.e.,* a P4-programmable switch and a SmartNIC. The programmable switch is equipped with a double-pipeline Tofino ASIC, 32 100GE QSFP ports, and 4 Intel 1.60GHz CPU cores. The SmartNIC is equipped with NFP-4000 [30] and two 10GE ports. We build a prototype of *HyperSight* dynamic compiler with 4000 lines of python code. The dynamic compiler runs in switch CPU and can automatically convert queries into the Tofino chip. Moreover, we employ two packet traces captured from real-world networks to evaluate vBFQ. First, *UNIV* is from a data center network [42]. Second, *MAWI* spans about one hour and is collected from an Internet exchange point [43]. Moreover, we evaluate *HyperSight* under three widely-used queries. Table V shows code, monitoring targets, and usage of the queries.

Evaluation goals. To comprehensively understand trade-offs made by *HyperSight*, we design experiments with the following four evaluation goals. First, based on real-world packet traces, we compare *HyperSight* with existing network monitoring tools in terms of implementing two widely-used queries, which monitors flow paths and large flows (§VII-A). Second, we present the scalability of *HyperSight* via simulating it in a typical data center network (§VII-B). Third, we present an in-depth analysis of BFQ and compare BFQ with five existing algorithms that can de-duplicate elements in an infinite stream (§VII-C). Fourth, we evaluate the performance and hardware resource overhead of *HyperSight* (§VII-D).

Result overview. We summarize evaluation results as below.

- For monitoring flow paths, *HyperSight* achieves 100% coverage and reduces transmission overhead by about two orders of magnitude when comparing with NetSight. For monitoring large flows, *HyperSight* completely captures all large flows with lower than 0.1% transmission overhead on UNIV and 1% transmission overhead on MAWI.
- In the simulation-based experiments, *HyperSight* can monitor all congestion events and over 99% congested flows, which is close to 5x of sampling at 1:10. Meanwhile, *HyperSight* keep the overall transmission overhead lower than 500Mbps in the fat-tree network.
- *HyperSight* incurs moderate resource overheads and minor performance overheads, which brings about a small impact on other data plane functions (*e.g.,* routing and ACL).

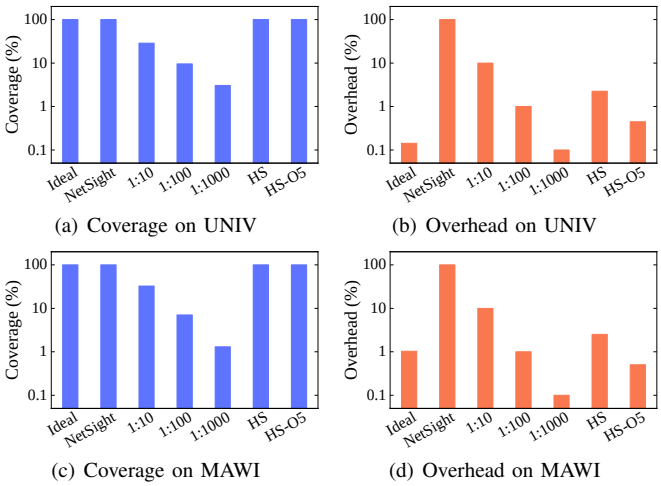
A. Evaluation of HyperSight

We use *HyperSight* to implement $\mathbb{Q}_1$ and $\mathbb{Q}_2$, and present evaluation results as below.

Analysis on $\mathbb{Q}_1$. An intuitive solution for ECMP paths is to only monitoring forwarding paths of TCP SYN packets

Table V
QUERIES USED IN THE EVALUATION OF *HyperSight*.

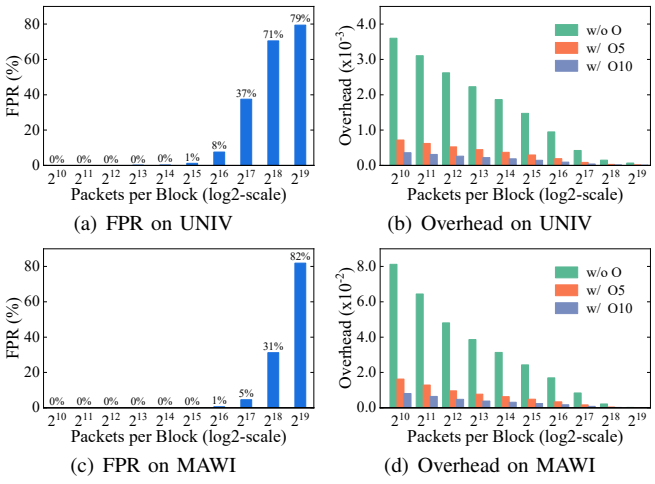
Query	Monitoring targets	Used by
$\mathbb{Q}_1$ = `distinct` (<flow, in_port, eg_port>, <flow, in_port, eg_port>)	ECMP flow paths	§VII-B, §VII-D, §VII-E
$\mathbb{Q}_2$ = `distinct` (<flow, tcp.seq>, <flow, tcp.seq&0xFFF00000>).`duplicate` (*, <flow>)	Large flows	§VII-B
$\mathbb{Q}_3$ = `distinct` (<flow, qid, delay>, <flow, qid, delay&0xFC00>)	Congested flows	§VII-C



(a) Coverage on UNIV (b) Overhead on UNIV

(c) Coverage on MAWI (d) Overhead on MAWI

Figure 9. **Coverage and overhead comparison in terms of $\mathbb{Q}_1$.**



(a) FPR on UNIV (b) Overhead on UNIV

(c) FPR on MAWI (d) Overhead on MAWI

Figure 10. **False positive rates and overheads in terms of $\mathbb{Q}_2$.**

at the beginning of flows. However, this solution fails when ECMP paths change due to link failures or load balancing. $\mathbb{Q}_1$ enables monitoring paths continuously and provides visibility of path changes. In the experiments of $\mathbb{Q}_1$, we compare *HyperSight* with three countermeasures, including the *ideal* baseline, NetSight [9], and sampling. For the ideal baseline, we only generate one report for one ECMP flow path and can cover all flows. NetSight reports messages for all packets. Sampling is configured with different ratios, including 1:10, 1:100, and 1:1000. As for *HyperSight*, we show coverage and overheads of *HyperSight* (*HS*) and optimized *HyperSight* whose batch size is five (*HS-O5*).

As shown in Figure 9(a) and Figure 9(c), the ideal baseline and NetSight can monitor all ECMP paths for all flows. With the sampling ratio increasing from 1:10 to 1:1000, the coverage of sampling decreases dramatically. For 1:10, the coverage ratio is about 28%, while the ratio is as low as 3% for 1:1000. For both packet traces, the coverage ratio of HS and HS-O5 are larger than 99.99%. Figure 9(b) and Figure 9(d) shows the monitoring overheads, *i.e.,* the ratio of monitoring messages to normal packets. NetSight generates monitoring messages for all packets and incurs unacceptable overheads. As for sampling, the overhead decreases linearly as the sampling ratio increases. HS incurs several percentages of overheads, which is comparable to sampling 1:100. Furthermore, HS-O5 reduces overheads by 80%. In summary, as for $\mathbb{Q}_1$, *HyperSight* is the only one that is close to the ideal baseline and can achieve both good coverage and low overheads.

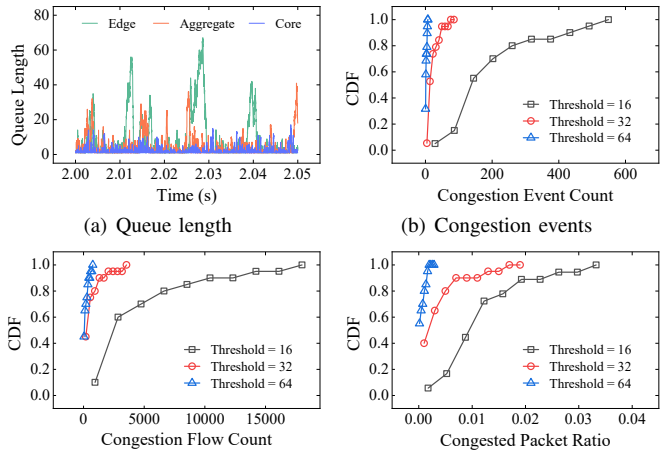
Analysis on $\mathbb{Q}_2$. In $\mathbb{Q}_2$, we utilize *HyperSight* to monitor the flows whose sizes are larger than 2^{20} bytes. In the experiments, we measure FPR and overheads under different numbers of packets per block. Furthermore, we measure the overheads of *HyperSight* when the optimization batch size is 1 *w/o O*, 5*w/ O5*, and 10 (*w/ O10*).

As shown in Figure 10(a) and Figure 10(c), *HyperSight* can keep 0% FPR when the window size is small. When the window size further increases, FPR increases accordingly. As shown in Figure 10(b) and Figure 10(d), the overhead decreases dramatically with the window size increasing. For example, when there are 2^{14} packets per block for UNIV, we can monitor all large flows. If the switch ASIC is forwarding packets at 1Gpps, the switch CPU should be about to process monitoring messages at 2Mpps. With the optimization, the switch CPU should only work at 200Kpps, which is within the capability of the switch CPU. Notably, the query running in ASIC inevitably generate false negatives, but the agent in switch CPU can faithfully remove those false negatives.

B. Simulation of HyperSight

We simulate a fat-tree network with k = 4 using NS3 (20 switches and 32 hosts). In the simulation, links are configured with 1Gbps bandwidth and 1ms delay, and switches ports are configured with one queue whose size is 1000 packets. During the simulation, each host emits 1000 TCP flows to any other host, and the start time of flows follows a uniform distribution. We employ a heavy-tailed Pareto distribution to derive flow sizes [44] whose mean is 10000 bytes. We run $\mathbb{Q}_3$ in switches to monitoring congestion events and flows.

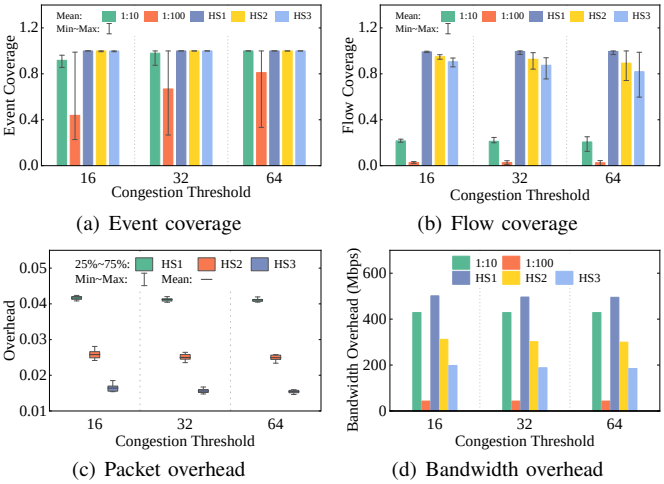
Congestion characteristics of simulation traffic. We trace en-queuing length of each packet in each switch to understand queuing length evolvement, congestion events, congested flows, and congested packets, named congestion characteristics, which are shown in Figure 11(a). We use different queue thresholds to classify whether queues are congested. More specifically, when the queue length exceeds the threshold, the congestion happens, and the congestion ends as soon as the queue length drops down below the threshold. In our experiments, the congestion thresholds can be 16, 32, and



(a) Queue length (b) Congestion events

(c) Congestion flows (d) Congestion packets

Figure 11. **Queue, congestion events, congestion flows, and congestion packets in the simulation experiment.**



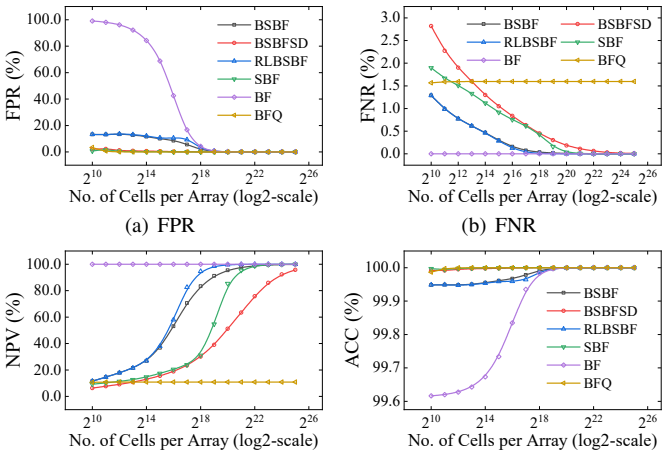
(a) Event coverage (b) Flow coverage

(c) Packet overhead (d) Bandwidth overhead

Figure 12. **Using sampling and *HyperSight* to detect congestion events and flows.**

64. Figure 11(a) shows the queuing length evolvement of three ports in an edge switch, an aggregate switch, and a core switch. Edge switches encounter more congestion events than aggregate switches and core switches. Figure 11(b) and Figure 11(c) show the cumulative distribution of congestion events and flows in one switch. When the congestion threshold rises, the numbers of congestion events and congestion flows decrease. Furthermore, we show the ratio of packets experiencing congestion in Figure 11(d), revealing that congested packets only take up a small portion of overall traffic.

Analysis on $\mathbb{Q}_3$. As for $\mathbb{Q}_3$, we present the congestion event coverage, congestion flow coverage, as well as overheads in packets and bandwidth in Figure 12. In the experiment, we compare *HyperSight* (optimization batch size is 5) with sampling (1:10 and 1:100). Moreover, we change the numbers of packet per block to 4000 (*HS1*), 10000 (*HS2*), and 20000 (*HS3*) for *HyperSight*. As for event coverage shown in Figure 12(a), *HyperSight* can monitor over 99% congestion events in all switches while sampling at 1:100 can only supply information of 43%-81% congestion events on average. As for flow coverage shown in Figure 12(b), sampling performs worse. Sampling at 1:10 can only monitor no more than 22% congestion flows. *HyperSight* keeps flow coverage over 99% with 4000 packets per block. As for packet overheads



(a) FPR (b) FNR

(c) NPV (d) ACC

Figure 13. **Algorithm comparison with varied numbers of cells per array.**

shown in Figure 12(c), monitoring messages generated by *HyperSight* take up several percents of normal packets, and packet overheads decrease significantly with the packets per block increasing. As for bandwidth overheads shown in Figure 12(d), *HyperSight* generates 500Mbps network-wide monitoring traffic at most. Each switch generates 25Mbps monitoring traffic, which occupies 0.3% of switch bandwidth. Overall, *HyperSight* enables high-coverage and low-overhead congestion monitoring, and largely outperforms sampling.

C. Analysis of BFQ

To comprehensively understand the characteristics of BFQ, we compare it with the BF baseline and other four widely-used algorithms, including SBF [39], RSBF, BSBFSD, and RLBSBF [40]. All algorithms are tested against UNIV under $\mathbb{Q}_1$. Due to space constraints, we only show four metrics, including FPR, FNR, negative predicate value (NPV), and accuracy (ACC).

Analysis of varied M. Figure 13 shows compression of the algorithms with M increasing from 2^{10} to 2^{25} when K is 3. Bigger M is, the more memory algorithms occupy. For *HyperSight*, S is 4, and N is 65536. As shown in Figure 13(a) and Figure 13(d), BF encounters almost 100% FPR and zero ACC when the available memory is small, revealing that BF cannot be applied to infinite packet streams. Other algorithms including BFQ extend BF to work over the infinite stream. With M increasing, FPRs of all algorithms decrease significantly. For BFQ, it can keep FPR low and accuracy high even if there is a small amount of memory, which is preferable to performing monitoring tasks over resource-constrained switches. Furthermore, as shown in Figure 13(b) and Figure 13(c), FNR and NPV of BFQ are constant.

Analysis of varied K. Figure 14 compares the algorithms with K increasing from 2 to 5 when M is 65536. For *HyperSight*, S is 4, and N is 65536. As shown in Figure 14(a), BF keeps a high FPR, and FPRs of The other algorithms decrease with the number of arrays. As shown in Figure 14(b), BF has a zero FNR. The FNR of the other algorithms except BFQ increases with K, while BFQ has a decreased FNR when K increases. Because when K decreases, the window size of recently-arrived packets decreases. This increases the possibility that

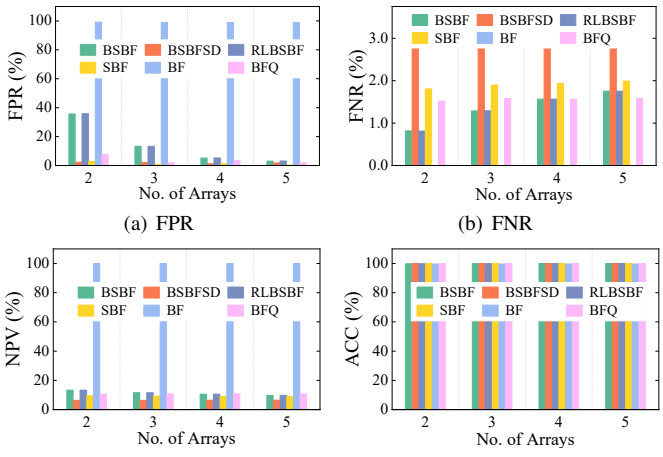


Figure 14. **Comparison of different algorithms with varied numbers of arrays (K).**

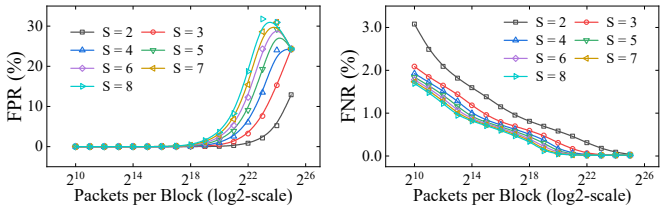


Figure 15. **FPR and FNR of BFQ with varied packet per block (N).**

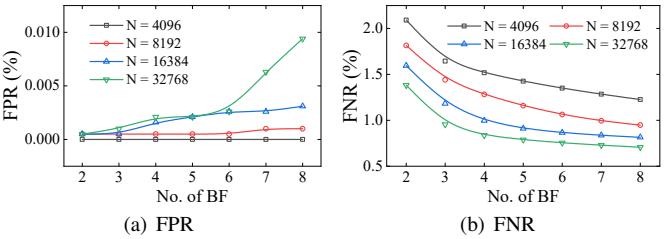


Figure 16. **FPR and FNR of BFQ with varied numbers of BF (S).**

duplicated packet behaviors are classified as distinct packet behaviors, and packets with duplicated behaviors are more likely to be in different windows. Figure 14(d) demonstrates that BFQ has similar accuracy with the other algorithms.

Analysis of varied N and S. Figure 15 and Figure 16 shows FPR and FNR of BFQ with N increasing from 2^{10} to 2^{25} when M, K are 65536 and 4 respectively. As shown in Figure 15(a), FPR of BFQ is close to zero when the number of packets per block is small. Then, FPR will grow linearly with N. It is intuitive as bigger N is, one BF in BFQ should check more packets simultaneously, which inevitably brings FPR up. As for FNR shown in Figure 15(b), FNR decreases with S increasing. In the experiments, N ranges from 4096 to 32768. As shown in Figure 16(a), FPR of BFQ increases slightly with S, and N has an impact on the increasing speed. As for FNR shown in Figure 16(b), FNR decreases with S increasing.

Summary. Based on the above results, we reasonably claim that BFQ performs comparably to existing algorithms for infinite streams. Meanwhile, BFQ is implementable on programmable switches while the other BF extension algorithms are too complex for programmable switches. Furthermore, BFQ's parameters can be dynamically adjusted by operators at runtime to make a desirable trade-off between FPR and FNR.

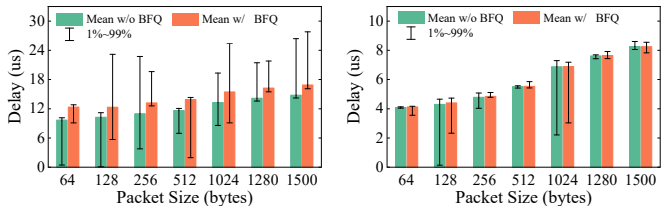


Figure 17. **Performance overhead introduced by BFQ.**

D. Overheads of vBFQ

HyperSight needs to deploy vBFQ on different P4 targets, such as Tofino, SmartNIC, and FPGA. First, vBFQ needs to perform field selection, store counters, and get final results, which introduce additional usage of data plane resources, such as matching crossbar, SRAM, and TCAM. Second, vBFQ introduces additional processing logic, which might influence packet forwarding performance. Next, we will analyze resource overheads and performance overheads of vBFQ.

Resource usage. In this part, we show the hardware resource usage of vBFQ, and we build a prototype of vBFQ that is configured with eight 65536-cell arrays. Table VI shows the results normalized by the resource usage of Switch.P4. vBFQ only needs 11.97% VLIW actions to implement compound actions and 29.09% packet header vector to accommodate metadata. Furthermore, vBFQ can consume a moderate number of stateful ALUs, which occupies 66.67% of Switch.P4. As the resource results are normalized by Switch.P4 which uses few registers, which makes the stateful ALU consumption of *HyperSight* seem high. When compared to the overall switches resources, the stateful ALU consumption of vBFQ is small. In the previous publication [1], we also propose an algorithm that can significantly reduce the amount of stateful ALUs consumed by vBFQ. In summary, vBFQ brings acceptable resource overheads to hardware targets and have a small impact on the other data plane functions.

Performance overhead. There raises a concern that vBFQ could bring performance overheads. We deploy vBFQ on Tofino and SmartNIC and measure the forwarding delay of *ECMP* with (w/) or without (w/o) vBFQ against packets of varied sizes. Figure 17 shows the evaluation results. We present not only mean delay but also the 1-99% interval to show whether vBFQ affects delay jitters. As shown in Figure 17(a), vBFQ introduces a modest delay increase in SmartNIC. For packets of all sizes, the delay increases by about 2µs. As shown in Figure 17(b), vBFQ introduces dozens of nanoseconds delay increase on Tofino. On both targets, vBFQ has a close-to-zero impact on delay jitter.

Table VI
ADDITIONAL HARDWARE RESOURCES CONSUMED WITH 16 SOURCE ROUTING LABELS. THE VALUES ARE NORMALIZED BY THE USAGE OF SWITCH.P4.

Resources	Normalized Usage
Match Crossbar	15.53%
SRAM	20.48%
TCAM	33.33%
Very Long Instruction Word (VLIW) Actions	11.97%
Hash Bits	13.54%
Stateful Arithmetic Logical Units (Stateful ALU)	66.67%
Packet Header Vector (PHV)	29.09%

VIII. Conclusion

This paper presents *HyperSight*, an efficient network monitor for packet behavior changes. *HyperSight* proposes *behavior-level* monitoring, which brings a remarkable decrease in monitoring cost with high monitoring coverage. *HyperSight* presents a declarative query language to enable convenient expression of various packet behavior monitoring tasks. *HyperSight* proposes BFQ to empower a powerful capability for monitoring packet behavior changes. *HyperSight* proposes to virtualize BFQ to implement dynamic monitoring tasks without disrupting switches. *HyperSight* supports a wide range of network monitoring queries and can monitor over 99% packet behavior changes while reducing overheads by two orders of magnitude. In our future work, *HyperSight* will explore automatically tuning the number of cells per array in vBFQ for accuracy guarantee and monitoring network-wide packet behavior changes.

Acknowledgement

This research is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (No. 61872426, No. 61625203, and No. 61832013). Tong Yang is the corresponding author. We thank Chen Sun, Zhilong Zheng, Yiran Zhang, Yunsenxiao Lin, and Heng Yu for their insightful suggestions.

References

[1] Y. Zhou, J. Bi, T. Yang, K. Gao, C. Zhang, J. Cao, and Y. Wang, "Keysight: Troubleshooting programmable switches via scalable high-coverage behavior tracking," in *Proceedings of ICNP*, 2018.

[2] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of SOSR*, 2016.

[3] N. Katta, M. Hira, A. Ghag, C. Kim, I. Keslassy, and J. Rexford, "Clove: How i learned to stop worrying about the core and love the edge," in *Proceedings of HotNets*, 2016.

[4] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proceedings of SIGCOMM*, 2018.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of CoNEXT*, 2011.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of NSDI*, 2010.

[7] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," in *Proceedings of SIGCOMM*, 2015.

[8] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, and L. Yuan, "dshark: A general, easy to program and scalable framework for analyzing in-network packet traces," in *Proceedings of NSDI*, 2019.

[9] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proceedings of NSDI*, 2014.

[10] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of SIGCOMM*, 2015.

[11] I. Farris, T. Taleb, Y. Khettab, and J. Song, "A survey on emerging sdn and nfv security mechanisms for iot systems," *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, 2019.

[12] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, "SIMON: A simple and scalable method for sensing, inference and measurement in data center networks," in *Proceeding of NSDI*, 2019.

[13] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proceedings of SIGCOMM*, 2016.

[14] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," in *Proceedings of SIGCOMM*, 2014.

[15] B. Claise, "Cisco systems netflow services export version 9," Website, http://www.rfc-editor.org/rfc/rfc3954.txt.

[16] sFlow, "sflow," Website, https://sflow.org/.

[17] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *Proceedings of ICDCS*, 2014.

[18] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: Information Rich Flow Record Generation on Commodity Switches," in *Proceedings of EuroSys*, 2018.

[19] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow," in *Proceedings of ATC*, 2018.

[20] Y. Li *et al.*, "Lossradar: Fast detection of lost packets in data center networks," in *Proceedings of CoNEXT*, 2016.

[21] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *Proceedings of NSDI*, 2016.

[22] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of SIGCOMM*, 2018.

[23] Q. Huang, P. P. C. Lee, and Y. Bao, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of SIGCOMM*, 2018.

[24] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proceedings of NSDI*, 2013.

[25] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of SIGCOMM*, 2013.

[26] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM CCR*, vol. 44, no. 3, Jul. 2014.

[27] Barefoot Networks, "Barefoot tofino switch," Website, https://barefootnetworks.com/technology/.

[28] C. Zhang, J. Bi, Y. Zhou, A. Basit, and J. Wu, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *Proceedings of ICCCN*, 2017.

[29] D. Hancock and J. van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *Proceedings of CoNEXT*, 2016.

[30] Netronome, "Netronome flow processor," Website, https://netronome.com/product/nfp-6xxx/.

[31] "The code of keysight," Website, https://github.com/KeySight-P4.

[32] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of SOSR*, 2017.

[33] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of SIGCOMM*, 2016.

[34] A. Gupta, R. Harrison, A. Pawar, R. Birkner, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proceedings of SIGCOMM*, 2018.

[35] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proceedings of SIGCOMM*, 2017.

[36] P4 Language Consortium, "P4 runtime," Website, https://github.com/p4lang/PI.

[37] The Apache Software Foundation, "Flink: Stateful computations over data streams," Website, https://flink.apache.org.

[38] Barefoot Networks, "The world's fastest & most programmable networks," Website, https://goo.gl/1mtjpf.

[39] F. Deng and D. Rafiei, "Approximately detecting duplicates for streaming data using stable bloom filters," in *Proceedings of SIGMOD*, 2006.

[40] S. K. Bera, S. Dutta, A. Narang, and S. Bhattacherjee, "Advanced bloom filter based algorithms for efficient approximate data de-duplication in streams," *CoRR*, 2012.

[41] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "Mpvisor: A modular programmable data plane hypervisor," in *Proceedings of SOSR*, 2017.

[42] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of IMC*, 2010.

[43] WIDE Project, "Mawi working group traffic archive," Website, http://mawi.wide.ad.jp/mawi/.

[44] V. Paxson, "Empirically derived analytic models of wide-area tcp connections," *IEEE/ACM Trans. Netw.*, vol. 2, no. 4, p. 316–336, Aug. 1994. [Online]. Available: https://doi.org/10.1109/90.330413