USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SketchPipe: Toward Accurate Sketch-based Network Measurement on Multi-Pipeline Switches with Splitless Sketch Placement

Xiang Chen, Longlong Zhu, and Linying Zheng, *Zhejiang University;* Hongyang Du, *The University of Hong Kong;* Dong Zhang, *Fuzhou University;* Jianshan Zhang, *Minjiang University;* Xuan Liu, *Yangzhou University;* Qun Huang, *Peking University;* Dusit Niyato, *Nanyang Technological University;* Haifeng Zhou and Chunming Wu, *Zhejiang University;* Hongyan Liu, *Fuzhou University;* Kui Ren, *Zhejiang University*

https://www.usenix.org/conference/nsdi26/presentation/chen-xiang

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

SketchPipe: Toward Accurate Sketch-based Network Measurement on Multi-Pipeline Switches with Splitless Sketch Placement

Xiang Chen[1] *Longlong Zhu*[1] *Linying Zheng*[1] *Hongyang Du*[2] *Dong Zhang*[3] *Jianshan Zhang*[4] *Xuan Liu*[5]
Qun Huang[6] *Dusit Niyato*[7] *Haifeng Zhou*[1] *Chunming Wu*[1] *Hongyan Liu*[3] *Kui Ren*[1]

[1]*Zhejiang University* [2]*The University of Hong Kong* [3]*Fuzhou University* [4]*Minjiang University*
[5]*Yangzhou University* [6]*Peking University* [7]*Nanyang Technological University*

Abstract

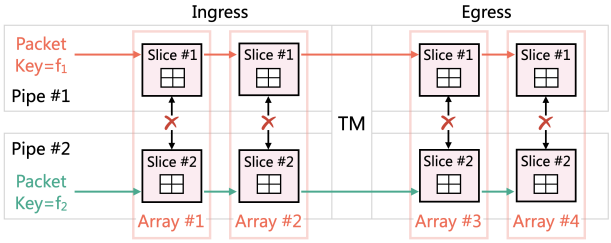
Sketches compactly measure traffic statistics with limited resource usage and are compatible with high-speed multi-pipeline switches. Existing solutions place sketches on multi-pipeline switches via *array-split placement* that splits each sketch array into several identical slices and places each slice on a specific pipeline. However, they incur two issues that drop measurement accuracy: (1) Different slices in different pipelines redundantly measure the same flows, wasting scarce switch resources. (2) Some slices receive much higher traffic loads than other slices. Hence, the data of different slices are highly varied, making their analysis inaccurate.

We propose SketchPipe, a framework that offers accurate sketch-based measurement for multi-pipeline switches. The key ideas are two-fold. First, SketchPipe places each sketch array *exclusively* on a pipeline. Second, it caches the flow keys of normal packets while using synthetic state packets to *asynchronously* transfer cached keys to sketch arrays. As such, it avoids splitting arrays to enable accurate measurement while avoiding affecting normal packet processing. Our experiments on 12.8 Tbps Tofino2 switches show that SketchPipe improves accuracy by up to two orders of magnitude for various sketches and network monitoring applications.

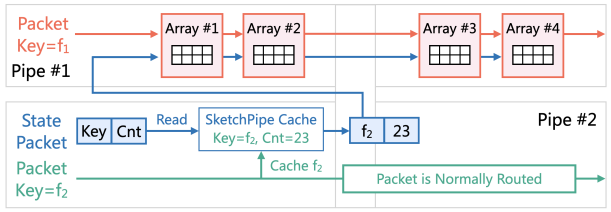
1 Introduction

In network measurement, sketches [2, 28, 33, 35, 36, 55] compactly summarize traffic statistics, e.g., the *heavy keys* with packet counts above a threshold, in several counter arrays. They have formed the basis of network monitoring applications, e.g., heavy hitter [33] and DDoS flow detection [18].

The literature uses high-speed multi-pipeline switches to deploy sketches to keep up with Tbps-level traffic [2, 28, 35, 36, 55]. Each switch include many *compound pipelines* [6, 11, 49]. Each compound pipeline contains several ingress and egress pipelines, which perform packet processing with sketch operations. Each ingress pipeline receives packets from a port and sends them to the traffic manager (TM). TM interconnects ingress and egress pipelines. It routes packets to egress pipelines that emit packets.



(a) *Array-split placement. Each sketch array is split into two 2×2 slices in different pipelines. Each packet arriving at a pipeline can only be measured by the slices in that pipeline*



(b) *Splitless placement. Each 2×4 array locates in only one compound pipeline. If a flow key, e.g., f_2, arrives at a pipeline that does not place any arrays, it will be cached. A synthetic state packet will later transfer this key to all arrays.*

Figure 1: *Array-split vs. splitless sketch placement on a switch comprising two compound pipelines #1-#2.*

In particular, the number of compound pipelines in each switch keeps increasing, e.g., both Broadcom Tomahawk and Intel Tofino have doubled the number of compound pipelines in a few years [6, 11, 49]. The reason is that switch vendors are always looking for ways to improve switch throughput. Since increasing the clock frequency of each individual compound pipeline brings high power consumption and non-trivial latency [6], their current practice is to push for more hardware parallelism, i.e., using more compound pipelines. To keep parallelism, the resources of each compound pipeline are naturally *isolated* from those of other pipelines [11, 49].

Array-split placement. In Figure 1(a), existing solutions adopt array-split sketch placement, which is a typical strategy when adapting prior sketches to multi-pipeline switches [17, 19, 20, 28, 33, 35, 36, 46, 55, 57]. They place each slice on a specific compound pipeline. When a packet of a flow arrives

at a slice, the slice invokes sketch operations to summarize flow statistics and records them in its counters. Finally, the counter values of slices are aggregated to detect heavy keys.

However, this approach introduces two issues that significantly affect the measurement accuracy of sketches.

(1) *Redundant measurements*. Ideally, for each flow, the sketch only uses the minimum number of counters (denoted by K) to keep track of the packets of that flow to conserve scarce switch resources. In array-split placement, each sketch array is split into identical slices on different compound pipelines. Each slice uses its own hash functions. When packets from the same flow arrive at different pipelines (e.g., due to ECMP routing or per-packet load balancing), each slice measures and stores the flow's statistics separately. Such redundancy reduces the effective capacity of each sketch and increases hash collisions, leading to accuracy drops. Also, ECMP, its variants [59], and per-burst or per-packet load balancing [1, 14, 16, 53] can cause this issue [11].

To exemplify, consider a count-min sketch with two slices. If flow A's packets arrive at both Pipe 1 and Pipe 2, Slice 1 increments a counter at index $h_1(A)$ while Slice 2 increments another counter at index $h_2(A)$. These counters occupy separate memory in the sketch data structure rather than the same space. As a result, each flow consumes $K \times N_{slice}$ counters instead of the expected number of K counters, where K is the number of hash functions.

(2) *Measurement imbalances*. The workloads of different slices are *unbalanced*, leading to highly varied counter values. Directly utilizing these values reduces accuracy [11], e.g., an 80% precision drop (§5). This is because some compound pipelines (i.e., hot pipelines) receive more packets than others due to two reasons. First, traffic is skewed, i.e., most packets come from a few flows with heavy keys [4, 20, 40, 58]. Second, which compound pipeline each flow arrives at is uncertain. Thus, the sketch slices on the pipelines that receive the flows with heavy keys suffer from higher workloads.

Key idea. We propose SketchPipe, a framework that offers accurate sketch-based measurement for multi-pipeline switches. Our key ideas are two-fold. First, SketchPipe *exclusively* places each array to only one compound pipeline (Figure 1(b)). This avoids partitioning sketches, eliminating redundant measurements and measurement imbalances.

However, splitless placement may still bring high performance overheads. In detail, every normal incoming packet should be routed to every compound pipeline that places sketch arrays to perform measurement operations. Such inter-pipeline routing increases the per-packet processing latency several times and degrades flow completion time.

In response, the second key idea is to disable the inter-pipeline routing of normal packets. Instead, SketchPipe *asynchronously* completes sketch measurement when handling normal packets: each normal packet is routed as usual while SketchPipe retrieves and sends its key to the sketch arrays in other pipelines, which update their data with the key. Specifically, it caches the key at the end of ingress while using the in-switch capability of packet generation [62] to create *state packets*. State packets retrieve cached keys and piggyback keys in their headers. Then they traverse the remaining arrays to complete measurement operations with their piggybacked keys. For example, in Figure 1(b), normal packets are routed as usual while their keys (e.g., f_2) are cached and retrieved by state packets to complete measurement.

Design challenge. Nevertheless, SketchPipe still faces the design challenge of avoiding high overheads.

(1) Its cache needs to occupy a portion of switch resources. Thus, caching the key of every incoming packet can easily saturate scarce switch resources.

(2) Processing the state packets created by SketchPipe may affect the performance of normal packet processing due to their competition on switch bandwidth.

Our design. SketchPipe addresses the above challenge:

(1) SketchPipe adopts light cache design, i.e., its cache uses a small amount of switch resources. Since traffic is skewed, the number of distinct flows is orders-of-magnitude smaller than that of packets [9, 43, 44, 60, 61]. So our cache aggregates the flow keys of the packets that belong to the same flow. It stores per-flow information, making its size small.

(2) SketchPipe minimizes the overheads of processing state packets in two steps. First, SketchPipe limits the number of state packets to its cache size. For each cache entry, it only creates one state packet that is dedicated to drain the entry. The state packet extracts the aggregated information from the entry and uses them to update sketch arrays. Once it completes measurement, it is routed back to the cache to measure the next key. Second, to avoid bandwidth competition and throughput reduction, SketchPipe isolates state packets from normal packets by handling state packets entirely in *internal recirculation bandwidth*. In detail, each compound pipeline offers an internal port [11, 61, 62], which O(100 Gbps) bandwidth is independent of normal packet processing. We dedicate such bandwidth to handle state packets.

Contributions. We make the following contributions.

- Using real-world traces and testbed experiments, we highlight the need of building a new framework for accurate sketch placement on multi-pipeline switches (§2).
- We propose and design SketchPipe to achieve low overheads and high accuracy in the sketch measurement running on multi-pipeline switches (§3-§4).
- We have implemented SketchPipe atop a 12.8-Tbps Intel Tofino2 switch [49]. Our experiments indicate that compared to existing solutions, SketchPipe improves accuracy by two orders of magnitude for a wide spectrum of network monitoring applications (§5).

2 Preliminaries and Motivation

2.1 Sketches for Multi-Pipeline Switches

Sketch-based measurement. Sketches measure traffic statistics with theoretical bounds on the tradeoff between resource

footprints and accuracy. They enable diverse network monitoring applications, e.g., the count-min sketch [12] measures per-flow counts for heavy hitter detection. Each sketch uses three steps to update its counter arrays [33, 35, 36].

- **Flow key hashing**. For each packet, the sketch extracts a set of packet headers (e.g., IP addresses) as the flow key and performs a hashing operation with the key in each array to acquire an index for value updating.
- **Counter array updates**. In each array, the sketch locates a counter with the index and updates the counter value, e.g., in count-min, the value is simply incremented by one; in UnivMon [33], the value is added by v where v is produced by uniformly mapping the key to 1 or -1.
- **Heavy key processing**. The sketch accepts user-specified thresholds to detect heavy keys. These thresholds are compared against the flow size estimates derived from updated counter values to identify heavy keys. The sketch periodically reports heavy keys to control plane applications.

Such a workflow keeps simplicity and thus allows sketches to be resource-efficient in traffic measurement, making sketches compatible with high-performance programmable switches.

Multi-pipeline switches. In Figure 1, programmable switches use multiple compound pipelines to parallelize packet processing and boost throughput [6, 11, 49]. Each compound pipeline has several match-action stages, each offering memory and computational resources. The memory like SRAM forms sketch arrays. The computational resources such as ALUs realize sketch operations.

Internal recirculation bandwidth. Each compound pipeline offers an internal port for handling synthetic or recirculated packets with O(100 Gbps) bandwidth [11, 61, 62]. Such bandwidth is isolated from the bandwidth of normal packet processing in the same compound pipeline.

Rules of sharing switch resources. Multi-pipeline switches have two rules on the usage of their resources [11, 49].

- **No resource sharing between compound pipelines**. The resources in *different* compound pipelines are *isolated*, e.g., in Figure 1(a), Pipe #1 and #2 do not share resources.
- **Resource sharing within each compound pipeline**. While the bandwidth for handling synthetic or recirculated packets and that for normal packets are distinguished in each compound pipeline, these packets manipulate the same memory and computational resources in the pipeline.

2.2 Motivation: Limitations of Existing Solutions

We illustrate why existing solutions that place sketches on multi-pipeline switches fail to achieve high accuracy.

Array-split sketch placement refers to a common practical approach that places sketches on multiple pipelines. As shown in Figure 1(a), it comprises three steps.

(1) Each sketch array is split into several identical slices. Each slice is placed on a specific compound pipeline.

(2) Each slice uses the three classes of sketch operations (§2.1) to measure incoming packets.

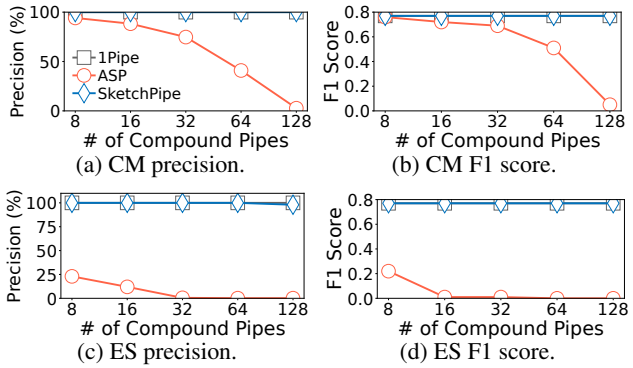


Figure 2: *Accuracy drops of existing placement.* 1Pipe *stands for the ideal one-pipeline placement (see §5.1).*

(3) The data of sketch slices are analyzed to detect heavy keys. Each slice sets a new threshold to detect heavy keys by multiplying the original threshold with a *coefficient* [11], e.g., for the threshold of 10^5 packets, the coefficient of 0.1 guides each slice to detect a heavy key once the number of packets that use this key exceeds 10^4. By default, the coefficient is the inverse of number of compound pipelines.

However, array-split placement comes with two problems directly affecting measurement accuracy.

- **Redundant measurements**. Different sketch slices measure the packets identified by the same flow key while each slice allocates some counters for the flow. Thus, a lot of counters are wasted, reducing measurement accuracy.
- **Measurement imbalances**. Traffic is highly uncertain and dynamic at runtime [4, 40, 58]. The workloads in different compound pipelines are unbalanced by nature. Hence, the measurement workloads of different sketch slices in different compound pipelines are highly diverse. Analyzing various data from different slices to detect heavy keys suffers from poor accuracy (e.g., $O(10^3)$ false positives [11]).

To date, there are a lot of frameworks that place sketches on switches [2, 10, 20, 28, 33, 35, 36, 55, 57]. To the best of our knowledge, most of them consider a single pipeline when placing sketches, e.g., FCM-Sketch [45] targets a single multi-stage pipeline, which deviates from the multi-pipeline nature of modern switches. As such, to enforce successful deployment, they adopt array-split placement to match multiple pipelines, inevitably suffering from the above issues.

Microbenchmarks. We measure the impact of array-split placement. We use two representative sketches, the count-min sketch (CM) [12] and elastic sketch (ES) [55]. We refer to their papers to set configurations as follows. For CM, we use three hash functions. For ES, we set its heavy part to use two counter arrays while dedicating a hash function to its light part. We deploy the two sketches using two methods. First, we simulate a single-pipeline switch and configure the pipeline with the full resource capacity of our testbed switch [49]. We set CM and ES to consume all pipeline resources, respectively. Second, we increase the number N of compound pipelines from 8 to 128 [11]. We set each pipeline to use $1/N$ of the total

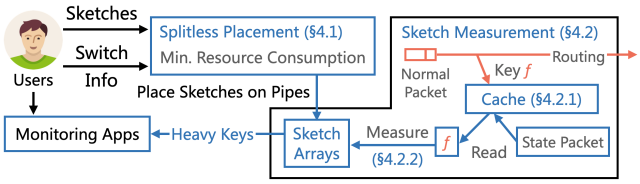


Figure 3: *SketchPipe Architecture.*

resource capacity while using array-split placement to deploy sketches. This setting comes from our observation that the resource capacity of next-generation switches does not grow too much due to high chip footprints and heat consumption [3, 11, 25, 49].

In this context, we use the two sketches to measure heavy hitters in a CAIDA 2019 trace [48] comprising around 26 M packets. The threshold is set to 0.02% of the number of packets in the trace [55]. For routing decisions, we randomly set the arrival ingress pipeline and destination egress pipeline of each flow while routing all its packets to traverse the same path in the switch. As flow sizes vary among flows, such a random setting results in several hot pipelines that process massive packets from heavy hitters. Thus, it simulates measurement imbalances among compound pipelines. Note that our results under other routing settings (e.g., ECMP) are similar while these settings are considered in §5.

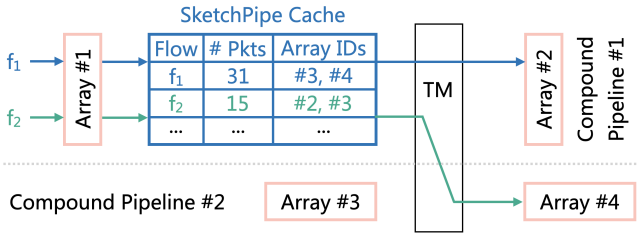
We obtain the ground truth from the trace while summing up the results of sketch slices in each compound pipeline to acquire measurement results. We compare them to compute the precision, $tp/(tp+fp)$, and the F1 score, $2tp/(2tp+fp+fn)$, of sketches, where tp, fp, and tn denote the number of true positives, false positives, and false negatives, respectively. In Figure 2, compared to the ideal one-pipeline placement, the original array-split sketch placement (ASP) in most existing frameworks significantly drops sketch accuracy.

3 Overview of SketchPipe

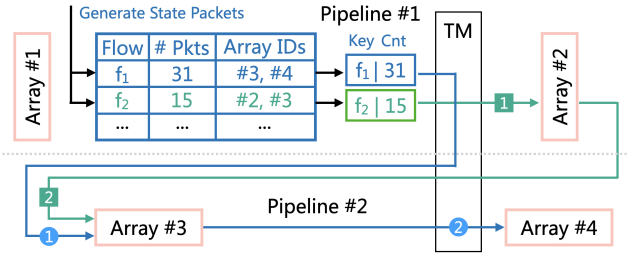
Goals. We aim to build a framework that achieves two goals of placing sketches on multi-pipeline switches.

- **G1**: **High accuracy**. Compared to the ideal one-pipeline placement, the sketches on multiple compound pipelines should offer comparable accuracy in detecting heavy keys.
- **G2**: **Low overheads**. Sketch-based measurement should not impose significant resource and performance overheads on normal packet processing.

Overview. We propose SketchPipe, a framework for accurate sketch-based measurement in multi-pipeline switches. In Figure 3, users (e.g., administrators) submit sketches to SketchPipe. They program the sketch as usual without additional programming burdens. SketchPipe chooses to place each sketch array on *only* one compound pipeline, i.e., *splitless* sketch placement. It frames splitless sketch placement into an optimization problem. Solving this problem places every sketch array exclusively on a specific compound pipeline while minimizing switch resource consumption. It augments the sketch with its logic of caching flow keys and handling



(a) *Normal packet processing while caching flow keys.*



(b) *Processing cached keys in the compound pipeline #1.*

Figure 4: *Example of SketchPipe.*

state packets and places the sketch based on its decisions.

At runtime, SketchPipe performs its asynchronous algorithm for sketch-based measurement. For normal packets, it caches flow keys in each compound pipeline. Meanwhile, it creates state packets in each compound pipeline. The number of state packets equals the cache size. These state packets only occupy the internal recirculation bandwidth without affecting normal packet processing bandwidth. They retrieve keys from the cache and update sketch arrays with these keys. This asynchronous workflow completes measurement across multiple compound pipelines (§4.2). Moreover, sketches will report their measured flow keys to network monitoring applications such as heavy hitter detection for making network management decisions (§5).

By this means, SketchPipe achieves the two goals: (1) With the help of its asynchronous algorithm and state packets, each sketch array still has a complete view of all keys, retaining high accuracy (**G1** achieved). (2) Normal packets are only processed by the arrival ingress pipeline and destination egress pipeline without visiting irrelevant pipelines. The number of state packets is limited while the processing of state packets and that of normal packets are isolated, avoiding high performance or resource overheads (**G2** achieved).

Example. Figure 4 shows an example of SketchPipe when handling two flows, f_1 and f_2, in a two-pipeline switch. In Figure 4(a), users deploy a four-array count-min sketch on the switch via SketchPipe. Each array only resides in the ingress or egress of a specific compound pipeline.

At runtime, the packets of f_1 and f_2 arrive at the ingress of Pipeline #1. They are measured by Array #1. Then they arrive at the end of ingress, where their keys are cached by SketchPipe. Each cache entry records the flow key, the number of packets that use the key, and the IDs of sketch arrays waiting to measure these packets. Then normal packets are routed to the egress by TM based on routing rules, e.g., the

packets of f_2 are routed to the egress of Pipeline #2.

In this context, some arrays still have some unmeasured flow keys, e.g., Array #2 waits for measuring f_2. Thus, SketchPipe generates two state packets in Pipeline #1 and injects them to the cache, i.e., Figure 4(b). Each state packet retrieves the flow key and packet count from its target cache entry. It is then routed to the remaining arrays based on the array IDs in the cache entry, e.g., the state packet that records f_1 (in blue) is routed to Array #3 and #4 in Pipeline #2.

Scope. Our scope is three-fold. (1) SketchPipe focuses on the sketches that are designed for multi-pipeline switches, which are adopted by some production networks such as Alibaba and Tencent data center networks [27, 29, 47, 61]. Each sketch uses less resources than the entire switch capacity. SketchPipe does not support the sketches designed for software [18, 32], and the sketches that strictly rely on packet orders. In the literature [2, 19, 20, 28, 33, 35, 36, 55, 57], the above spectrum already covers a lot of existing sketches since these sketches are inherently designed for multi-pipeline switches while their measurements are agnostic of unpredictable packet orders. We plan to support the sketches with insertion-order-dependent data structures, e.g., distinct counting with martingale transform, in the future. (2) Some sketches (e.g., NZE [20]) need the control plane to decode their data before detecting heavy keys. SketchPipe supports such decoding by setting switches to report sketch counter values to the control plane. (3) SketchPipe does *not* aim to provide better sketch algorithms. Instead, it complements their sketches with an accurate and low-overhead method that runs sketches over multi-pipeline switches.

Assumptions. (1) Traffic is skewed [4, 20, 40, 55, 58, 61], i.e., most packets come from a few flows. This is already a standard traffic pattern in modern networks. (2) SketchPipe assumes that each array fits in the capacity of the ingress of a single compound pipeline. This guarantees that sketch arrays are successfully installed without being split among multiple compound pipelines. This assumption holds because modern sketches limit the size of each array with the maximum range of hash functions available in switches, e.g., 2^{16}. Thus, each array has up to $O(10^5)$ counters and can fit in a single pipeline.

4 Design of SketchPipe

4.1 Splitless Sketch Placement with SketchPipe

Goals. SketchPipe is designed to place sketches on a target multi-pipeline switch with two primary goals. (1) It ensures successful sketch placement by assigning each sketch array to a particular compound pipeline. (2) It aims to minimize the utilization of compound pipelines running sketch arrays, thereby optimizing switch resource consumption.

SketchPipe completes these goals by framing the problem of splitless sketch placement as an optimization challenge. The first goal is encoded as a constraint. The second goal is formulated as the objective for finding optimal solutions. We provide a detailed explanation below, where the notation of symbols is detailed in Table 1.

Table 1: Notation of main symbols used by this paper.

$\mathcal{S}$	Sketch $\mathcal{S}=\{\mathbb{A},\mathbb{H},\mathbb{U}\}$.
$\mathbb{A},\mathbb{H},\mathbb{U}$	Sets of sketch arrays, hashing, and updating operations.
$\mathbb{P},\mathbb{M},\mathbb{C},X$	Packet, bitmap, SketchPipe cache, and traffic imbalance.
N,m	Number of compound pipelines, and number of sketch arrays.
a_i,n_i,l_i	The i-th array in $\mathbb{A}$, its counter number, and per-counter size.
$D(a,b)$	Visiting order among sketch arrays.
$\mathcal{P}$	The switch formulated by a set of compound pipelines.
P_i,p_i^{ig},p_i^{eg}	The i-th compound pipeline, its ingress and egress.
K	Number of match-action stages in the ingress p_i^{ig} or egress p_i^{eg}.
α	Number of compound pipelines occupied by sketch arrays.
λ	Buffer space dedicated to the processing of state packets.
$\gamma(P_i)$	Variable indicating if $P_i\in\mathcal{P}$ is occupied by sketch arrays.
$x(a,p)$	Variable indicating if a is placed on the ingress/egress p.
$y(a,i,p)$	Variable indicating if a is placed on the i-th stage in p.

Input. The input of SketchPipe accepts a sketch, the model of the multi-pipeline switch, and a set of routing rules.

(1) The sketch is defined by an input P4 program [5]. P4 is a language for programming packet processing logic and has been utilized by numerous systems for sketch implementation [2, 20, 35, 36, 55]. SketchPipe uses existing toolchains [38] to transform the P4 program into the intermediate representation (IR). The IR eliminates irrelevant programming details and exhibits sketch arrays and operations.

In this context, SketchPipe extracts sketch arrays and operations from the IR and uniformly represents them with $\mathcal{S}=\{\mathbb{A},\mathbb{H},\mathbb{U}\}$: $\mathcal{S}$ denotes the sketch while $\mathbb{A}$, $\mathbb{H}$, and $\mathbb{U}$ denote the set of sketch arrays, the set of hashing operations, and the set of updating operations, respectively. In $\mathbb{A}$, there exist m sketch arrays while the i-th array a_i has n_i fixed-size counters. Each counter uses l_i bytes to record data. In $\mathbb{H}$, there are m hash functions. The i-th hash function computes an index j to address the j-th counter $a_i[j]$. With the index j, the i-th updating function in $\mathbb{U}$ is invoked to update $a_i[j]$ (e.g., $a_i[j]=a_i[j]+1$ in count-min sketches). Also, we use a matrix of boolean variables, $D(a,b)$, to record the visiting order among sketch arrays. If the array b must be visited after its former array a, $D(a,b)=1$; otherwise, $D(a,b)=0$. For example, for elastic sketches [55], the arrays that realize count-min sketches as light parts should be visited after the arrays that realize heavy parts. So the variables that specify the visiting orders among heavy and light parts equal true.

According to our investigation, existing sketches, which are compatible with switch ASIC pipelines, can be uniformly represented by the above model. This is because sketches rely on similar types of data structures (e.g., count-min sketches and bloom filters) and simple operations to offer strong theoretical guarantees of high accuracy and low resource consumption [2, 20, 35, 36]. So such a model is general [20, 35].

(2) Since each sketch comprises arrays and operations, both of which are placed on switch ASIC pipelines, we model the multi-pipeline switch as a set $\mathcal{P}$ of compound pipelines. The i-th compound pipeline P_i consists of an ingress p_i^{ig} and an egress p_i^{eg}. Each of them has K match-action stages. The j-th stage s_j in p_i^{ig} or p_i^{eg} offers finite memory and computational

resources. The switch $\mathcal{P}$ has N compound pipelines.

Output. SketchPipe enforces its decisions in two steps.

(1) SketchPipe decides where to place each array on compound pipelines. These decisions are two sets of boolean variables. First, $\{x(a,p)\}$ specifies the compound pipeline that runs a specific array. If $x(a,p)=1$, where a is an array while p is the ingress or egress of a compound pipeline, a is placed on p. Second, $\{y(a,i,p)\}$ shows which stages in p places a. If $y(a,i,p)=1$, a is placed on the i-th stage in p.

(2) SketchPipe inserts its logic of caching flow keys and handling state packets into the control flows defined by IR. Then it inputs both its decisions and the modified IR into the low-level compiler to configure compound pipelines.

Objective. The objectives of SketchPipe are to minimize the switch resource consumption of sketches. Similar to previous studies [2, 17, 35], it aims to (1) minimize the number of compound pipelines occupied by sketch arrays, i.e.

$$\textbf{minimize } \alpha,\ \alpha = \sum_{\forall P_i \in \mathcal{P}} \gamma(P_i),\ \text{where:}$$
$$\gamma(P_i)=1 \text{ iff } \sum_{\forall a \in \mathbb{A}} x(a,p_i^{ig}) \geq 1 \text{ or } \sum_{\forall a \in \mathbb{A}} x(a,p_i^{eg}) \geq 1 \quad (1)$$

where γ is the boolean variable indicating if the pipeline P_i has been used; and (2) minimize the number of match-action stages occupied by each sketch in each pipeline, i.e.

$$\textbf{minimize} \sum_{\forall a \in \mathbb{A}} \sum_{\forall P_i \in \mathcal{P}} \left(\sum_{j=1}^{K} y(a,j,p_i^{ig}) + \sum_{j=1}^{K} y(a,j,p_i^{eg})\right) \quad (2)$$

SketchPipe simultaneously optimizes these objectives via hierarchical objectives [54]. By default, these objectives have the same priority, which can be tuned by users.

Constraints. First, SketchPipe considers general constraints for successful sketch placement [17, 35]. These constraints are: (1) the placement should preserve switch resource limitations, (2) each sketch component (i.e., an array or operation) will be placed on at least one compound pipeline and at least one stage, (3) the execution dependencies among sketch components should be preserved, e.g., hashing operations should be invoked in the stages before the stages that run other components, which depend on the indexes computed by hashing operations, and (4) for an arbitrary array, it and its associated updating function should be placed on the same stages [23].

Second, SketchPipe employs specific constraints to realize splitless sketch placement, i.e., each array can be placed on the ingress/egress of only one compound pipeline.

$$\forall a \in \mathbb{A}: \sum_{\forall P_i \in \mathcal{P}} \left(x(a,p_i^{ig}) + x(a,p_i^{eg})\right) = 1 \quad (3)$$

Solving. Similar to existing frameworks [17, 35], SketchPipe uses the standard solver, Gurobi [15], for optimal solving.

4.2 Sketch Measurement with SketchPipe

Goals. On the basis of splitless sketch placement in §4.1, SketchPipe has two goals when building accurate sketch-based measurement for multi-pipeline switches. (1) It aims to successfully perform measurement with the sketch arrays running on different compound pipelines by transferring every key to visit every array. (2) It aims to minimize the impact of sketch-based measurement on normal packet processing throughput and buffer memory.

Algorithm 1 SketchPipe Cache

```
Input: Packet ℙ, bitmap 𝕄 of IDs of visited arrays
Variables: SketchPipe cache index i, flow key f, cache ℂ
1:  function HANDLING_NORMAL_PACKET(ℙ, 𝕄)            ▷ See §4.2.1
2:      Extract the flow key f from ℙ, and then route ℙ to TM
3:      SketchPipe cache index i ← HASH(f)
4:      if ℂ[i].key = f or ℂ[i] is empty then
5:          ℂ[i].key ← f
6:          ℂ[i].cnt ← ℂ[i].cnt+1
7:          ℂ[i].bitmap ← ¬𝕄          ▷ Record the IDs of unvisited arrays
8:      else      ▷ Evict on collision: replace the old key with the new key
9:          ℂ[i].key ← f
10:         ℂ[i].cnt ← 1
11:         ℂ[i].bitmap ← ¬𝕄
12: function HANDLING_STATE_PACKET(ℙ)               ▷ See §4.2.2
13:     Extract the cache index i from ℙ
14:     if ℂ[i] is empty then Recirculate ℙ
15:     ℙ ← ℂ[i], and then reset ℂ[i]
16:     Route ℙ to other compound pipelines based on ℙ.bitmap
```

To simultaneously achieve these goals, SketchPipe seperates its measurement from normal packet processing via an asynchronous algorithm: normal packets are processed and routed as usual to satisfy the second goal. Instead, their keys are cached at the end of ingress of each compound pipeline (§4.2.1). These keys will be retrieved by synthetic state packets, which send keys to the unvisited sketch arrays in other compound pipelines to achieve the first goal (§4.2.2).

4.2.1 Light Cache Design

Challenge 1: How to design a cache with limited switch resources? The limited switch resource capacity constrains the cache of SketchPipe. Each compound pipeline has a few match-action stages while most stages need to be allocated to the sketch and other functions [35, 36]. Thus, it is essential to minimize the resources consumed by the cache.

Solution: light cache design. Algorithm 1 summarizes the workflow of SketchPipe cache. Our solution is three-fold.

(1) We decide what to cache in SketchPipe (lines 1-7). We choose flow keys because traffic is skewed (i.e., most packets come from a few flows) [4, 20, 40, 55, 58, 61]. Therefore, the number of *distinct keys* is much smaller than that of packets, e.g., the former (8.9M) is three orders of magnitude lower than the latter (1,835.1M) in a one-hour trace collected in the Equinix-Chicago monitor from CAIDA [55].

However, compared to per-packet caching, per-key caching sacrifices the full visibility of traffic and misses the records of most packets, impeding accurate measurement. In response, when caching a flow key, SketchPipe counts the packets that use this key and await measurement in the cache entry. The intuition behind such aggregation is two-fold. (1) Sketches process the packets that use the same key in the same way (e.g., hashing them to the same counters). Hence, using the summarized count of these packets to update sketch arrays

Table 2: Hash collision rates of SketchPipe cache.

Cache Size	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
CAIDA 2019 Trace [48]	4.70%	4.61%	4.44%	4.09%	3.44%
MAWI 2024 Trace [13]	17.7%	11.1%	6.30%	3.31%	1.63%
IMC Data Center Trace [24]	<0.01%	<0.01%	0.00%	0.00%	0.00%

contributes to measurement results in the same way that individual packets update these arrays. (2) Packet counting is a basic hardware-compatible operation and can be easily integrated into SketchPipe caches.

In addition, each cache entry also records the IDs of the arrays waiting to measure the cached key. This guides subsequent state packets to direct the key to correct pipelines.

(2) We determine the cache size. We follow prior studies [37, 44] to limit SketchPipe to use O(10^6) cache entries. Such a choice uses a few switch resources (§5).

One concern is that the limited cache size leads to frequent hash collisions. If so, cached keys may be evicted by new keys due to collisions. However, since traffic is skewed, the keys of most packets are recorded by a few cache entries, making the probability of cache collisions small. Also, the state packets created by SketchPipe will rapidly drain the cache (O(1 μs), see Appendix 1), leaving enough room for caching new keys and mitigating hash collisions.

In Table 2, we validate the low probability of hash collisions by processing real-world traces, a CAIDA 2019 wide-area network trace [48], an MAWI 2024 trace [13], and the IMC data center traffic trace [24], using the SketchPipe cache. We partition these traces into consecutive epochs with the large size of 100 M packets, and present the average after processing every epoch. Even with a small cache size of 2^{16}, the probability remains below 3.5%. In particular, the probability in the IMC trace is much lower than other traces as data center traffic exhibits higher skewness, i.e., <5% flows have >95% packets. As such, the cache size of 2^{14} is already larger than the total number of flows, eliminating collisions.

(3) When hash collisions happen, we determine the mechanism of handling collisions in SketchPipe (lines 8-11). In particular, there exist various types of standard practices.

- We consider a strawman solution that selects a few keys to cache based on a sampling rate, e.g., 1:100. This solution inevitably leads to high cache miss ratios and thus serves as a baseline that demonstrates the necessity for SketchPipe to adopt a suitable strategy for handling collisions.
- First in first out (FIFO) replaces the oldest key in the cache with the new key. However, it is infeasible in switch ASIC pipelines as it needs excessive memory to record the insertion order of all cached keys to remove the oldest key.
- Random replacement (RR) randomly chooses an old key and replaces it with the new key when a collision occurs. While simple, it unintentionally evicts frequently accessed keys, leading to poor hit rates and low cache performance.
- Least frequently used (LFU) removes the least frequently accessed key to leave room for the new key. But similar to FIFO, it consumes excessive memory to monitor the key.

Table 3: Cache miss ratios (lower is better).

Cache	Sample	FIFO	RR	LFU	SketchPipe
CAIDA 2019 Trace [48]	99.4%	99.6%	99.1%	50.07%	11.4%
MAWI 2024 Trace [13]	99.5%	98.4%	92.6%	45.6%	29.27%
IMC Data Center Trace [24]	99.0%	1%	1%	1%	1%

- Least recently used (LRU) deletes the most recently used key. But realizing it suffers from the same issue as LFU.

In response, when collisions occur, SketchPipe adopts the evict-on-collision policy that approximates LRU [37, 44, 61]. It has proven to be surprisingly effective and is compatible with resource-restrictive switch ASIC pipelines [30, 37, 43]. In detail, when a collision happens, it simply evicts the old key to save the new key. We demonstrate its effectiveness by measuring the cache performance of the above cache mechanisms. We allocate 2^{16} entries to each cache and deploy caches on our testbed switch by default. For the caches that are hardware-incompatible, we present simulation results. Table 3 shows that SketchPipe achieves the best performance in terms of low cache miss ratios across all traces.

4.2.2 State Packet Management

Challenge 2: How to limit overheads? SketchPipe generates state packets to retrieve cached keys and deliver them to unvisited sketch arrays. However, without a careful way of managing state packets, massive state packets may overwhelm the available pipeline bandwidth and buffer space. As a result, normal packets have to compete with state packets for bandwidth and buffer, affecting their performance.

Understanding packet recirculation and generation. Before diving into the details of how SketchPipe addresses the challenge, we introduce two capabilities provided by the ASIC pipelines of existing programmable switches [49].

C1: *Synthetic packet generation*. Each compound pipeline has a packet generator that creates synthetic packets entirely in the pipeline. Once created, synthetic packets are distinguished from normal packets via their specific format.

C2: *Packet recirculation*. Every compound pipeline allows an arbitrary packet to be looped back to the ingress of its internal recirculation port and to begin a new round of processing. Such recirculation only occupies internal recirculation bandwidth instead of consuming normal packet processing bandwidth (§2). Thus, recirculated packets avoid affecting normal packet processing and switch throughput.

Solution: State packet management. We start by illustrating how SketchPipe uses state packets to complete the measurement of cached keys. Then we explain how SketchPipe controls its overheads when utilizing state packets.

State packet processing: First, SketchPipe uses the generator (**C1**) to generate state packets. It inputs the format of state packets to the generator. Each state packet initially comprises a cache index and a bitmap. The index addresses a specific cache entry while the j-th bit b_j in the bitmap specifies if the packet should visit the j-th array: if so, $b_j = 1$; otherwise, $b_j = 0$. The generator creates k state packets corresponding

to the k cache entries. Each state packet will be reused for the next key when it completes the measurement of a key. So the k state packets are created only once.

Second, SketchPipe handles state packets in two steps.

(1) *Cache processing*: The generator in each compound pipeline injects state packets to the internal recirculation port. Each state packet addresses one cache entry with its index. If the entry does not record any key, it will be recirculated (**C2**) until the entry records a key. Otherwise, it retrieves the key and packet count from the cache and piggybacks them on its header. Also, according to the array IDs in the entry, it flips its bitmap bits. For example, if the array IDs are #3 and #4, the third bit and fourth bit are set to one.

(2) *State packet routing*: For each state packet, SketchPipe matches the bitmap and picks the j-th bit. It directs the packet to the compound pipeline that runs the j-th array. Here, the j-th array is prioritized over other arrays, indicating by the matrix D recording the visiting order among arrays (§4.1): $D(a,b) = 1$, the measurement of the k-th array b relies on the results of the j-th array a, and thus a must be visited before b. When the state packet arrives at the j-th array, it updates the array with its key and packet count. For example, if the key is f_1 while the count is 31, it hashes f_1 to locate a counter in the array and updates the counter value based on 31 (e.g., incrementing the value by 31 in count-min sketches). Then the j-th bit b_j is reset to zero while the state packet is routed to the next array through TM or packet recirculation. If all bits equal zero, the measurement of the piggybacked key was completed. So the state packet is recirculated to the original pipeline to measure the next key.

Overhead control: State packets may compete with normal packets in both compound pipelines and TM buffer. SketchPipe limits their overheads with two strategies.

First, SketchPipe limits that the processing of all state packets can only occupy the internal recirculation bandwidth. This strategy isolates the processing of state packets from that of normal packets, thereby eliminating the competition between those packets on pipeline bandwidth.

Second, it restricts the buffer overheads incurred by state packets via a threshold λ (in MB), i.e., the TM buffer space occupied by state packets will never exceed λ. This strategy has two steps. (1) SketchPipe computes λ as the worst-case buffer size that caches all state packets at the same time. Recall that each state packet addresses a specific cache entry. Given k cache entries, there are at most k state packets in the buffer. Let l denote the size of each state packet. So $\lambda = k \cdot l$. (2) SketchPipe statically allocates a dedicated buffer space with the size of λ to recirculation through TM commands. By this means, it eliminates the competition between normal packets and state packets in the TM buffer.

5 Evaluation

We evaluate SketchPipe via testbed experiments and large-scale simulation. We report the average after 100 runs.

- (Exp#1-#5) In testbed, the sketches placed by SketchPipe achieve near-ideal accuracy for various network monitoring applications. In all cases, SketchPipe outperforms existing solutions by improving the accuracy by up to **81%** and reducing the errors by **two orders of magnitude**.
- (Exp#6) When scaling to multiple applications, SketchPipe beats existing solutions by **30% higher accuracy**.
- (Exp#7) Even with extreme traffic imbalances, SketchPipe still retains **near-ideal accuracy** in our testbed.
- (Exp#8) SketchPipe improves the accuracy of existing solutions by **two orders of magnitude** when scaling to tens of compound pipelines in simulation.
- (Exp#9) SketchPipe uses **at least 20% fewer resources** to achieve the same level of accuracy in our testbed.
- (Exp#10-16) SketchPipe only brings **small and acceptable overheads** to switch performance and resources.

5.1 Experimental Settings

Implementation. We realize SketchPipe on a 64×100 Gbps Tofino switch and a 32×400 Gbps Tofino2 switch [3, 49]. The former is a two-pipeline switch while the latter has four compound pipelines. We realize SketchPipe in $P4_{16}$.

(1) We realize the compile-time splitless sketch placement (§4.1) in C++ and Gurobi [15] for problem solving.

(2) We implement the data plane logic of SketchPipe (§4.2) in $P4_{16}$. First, we realize the SketchPipe cache through two register arrays. The first array comprises 64-bit entries, each of which has 32-bit higher bits and 32-bit lower bits. The higher bits cache the flow key while the lower bits count the number of packets using the cached key. The second array consists of 64-bit entries. Each entry records the bitmap of the cached key. Second, we realize the operations of manipulating the cache and processing normal packets and state packets with stateful ALUs and metadata.

(3) We exploit the switch development environment (SDE) 9.12.0 on the switch OS to invoke the packet generator to create state packets and inject them to the internal recirculation port. We invoke the TM commands in the SDE to dedicate the buffer memory of λ MB to internal recirculation ports.

(4) SketchPipe utilizes the switch capability that supports configuring different pipelines with different processing logic. It compiles the logic of different sketch arrays into binaries and loads them on corresponding compound pipelines determined by the placement. This ensures that each array runs exclusively on its assigned pipeline.

(5) SketchPipe utilizes bitwise operations to map packets to array IDs efficiently. Specifically, the state packet piggybacks a bitmap where each bit corresponds to a sketch array. SketchPipe determines the target array by checking the state of these bits (e.g., identifying which bit is set). This bitmap is precomputed and stored in the cache entry. It enables the state packet to be correctly routed to target arrays.

Testbed and simulator. We build a testbed comprising both a control plane and a data plane. We run the control plane on

a server. The server directly connects to data plane switches with 100 Gbps links. It runs network monitoring applications entirely in its user space, which collects events (e.g., heavy hitters) from switches through DPDK APIs [21]. Moreover, the data plane contains a cluster of eight servers directly connected by a 32×400 Gbps Tofino2 switch [49]. This switch has four compound pipelines, each comprising eight 400 Gbps ingress pipelines, eight 400 Gbps egress pipelines, and one 400 Gbps internal port. It offers a 64 MB TM buffer. For each compound pipeline, we connect two servers to two ingress pipelines of the compound pipeline. The two servers use Pkt-Gen [39] to inject traffic workloads to the compound pipeline at 200 Gbps and receive traffic from other servers. Each server has 36-core Intel(R) Xeon(R) Gold 6240C CPU (2.60 GHz), 128 GB RAM, and a two-port 100 Gbps NIC.

Since the number of compound pipelines in our switch is limited, we also simulate a multi-pipeline switch that has a large number of compound pipelines. We configure the performance and resource capacity of each compound pipeline based on the specifications of our testbed switch.

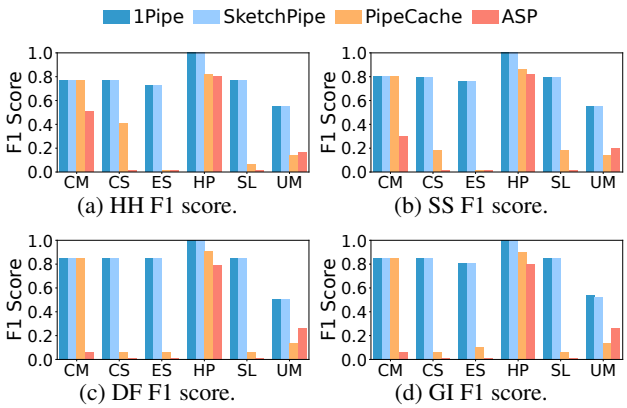
Parameters. Unless specified elsewhere, we set the parameters in SketchPipe as follows. (1) We set the SketchPipe cache size in all compound pipelines to 2^{16}, which collision rate is below 4% (Table 2). (2) We set the buffer threshold λ to 1 MB (1.5% of the total capacity) based on the cache size.

Baselines. To better quantify the accuracy and performance of SketchPipe, we implement existing approaches of sketch placement for comparison. We elaborate on them as follows.

(1) ASP is the original array-split sketch placement in prior studies, including FlowRadar (FR) [28], UnivMon (UM) [33], SketchLearn (SL) [19], Elastic Sketch (ES) [55], CocoSketch [57], NZE-Sketch [20], FlightPlan [46], P4All [17], Sketchlib [35], and Sketchovsky [36]. All these studies *vertically* place sketches on multiple compound pipelines (§2.2). Note that while some baselines (e.g., P4All) were not originally designed for multiple pipelines, we adapt them by applying array-split placement to enable their deployment on multi-pipeline switches for a fair comparison.

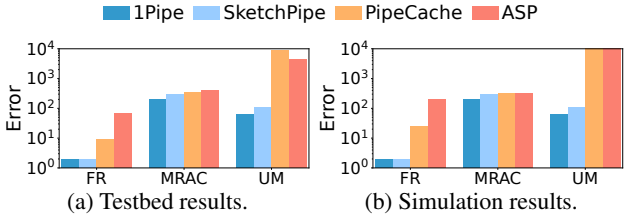
(2) PipeCache [11] is a recent ASP-based approach. We implement its open-source codes. However, it remains preliminary. First, it places only sketches on the egress while dedicating ingress resources to deliver flow keys to the correct egress sketch slices. While these keys are only measured once, PipeCache halves the number of counter arrays available for sketches and limits sketch accuracy. Second, it relies on normal packet routing to transfer keys, leading to imbalances. In comparison, SketchPipe does not limit placement locations, enabling sketches to use more resources and achieve higher accuracy. Also, it adopts synthetic state packets, which can be flexibly routed to every array, and thus eliminates measurement imbalances.

(3) 1Pipe refers to the ideal *theoretical* results, assuming that the switch only has one single compound pipeline. Thus, we obtain its results via simulation while leveraging them to



(a) HH F1 score. (b) SS F1 score. (c) DF F1 score. (d) GI F1 score.

Figure 5: (Exp#1-#4) Accuracy of HH, SS, DF, and GI.



(a) Testbed results. (b) Simulation results.

Figure 6: (Exp#5) Accuracy of EE.

demonstrate the accuracy gaps between ASP, PipeCache, or SketchPipe and the maximum possible accuracy of sketches.

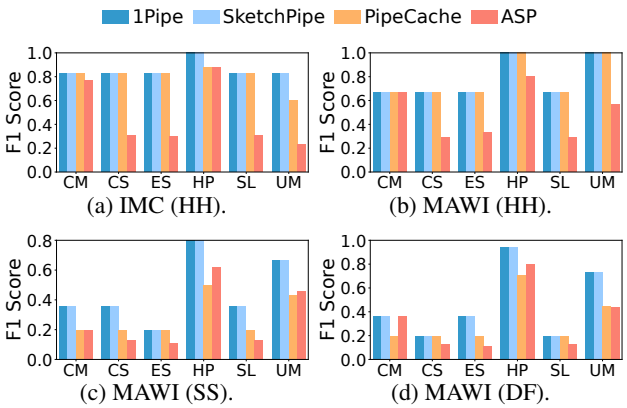
Sketches. We realize eight existing sketches: CM [12], the count sketch (CS) [7], ES [55], FR [28], HashPipe (HP) [42], MARC [26], SL [19], and UM [33]. We refer to the theoretical bounds in their original papers to configure their parameters to maximize accuracy. We use baselines and SketchPipe to deploy these sketches on our switch, respectively. Moreover, the events observed by sketches, or sketch data, will be collected to applications when every measurement epoch ends. The default epoch length is 10 seconds [36]. We set each sketch to measure the flows identified by IP pairs.

Monitoring applications. We realize five applications: heavy hitter detection (HH) [42], superspreader detection (SS) [20], DDoS flow detection (DF) [34], global iceberg detection (GI) [33], and entropy estimation (EE) [11]. We set the thresholds of HH and GI to 10K packets, set the threshold of SS to 0.5% of the total number of IP addresses, and set the threshold of DF to 0.5% of the total number of five-tuples [8]. We realize them with CM, CS, ES, HP, SL, and UM, respectively. For EE, we compute the entropy of measured flow distribution based on [11]. We realize EE with FR, MARC, and UM, respectively.

Metrics. (1) Precision = $tp/(tp+fp)$; (2) Recall = $tp/(tp+fn)$; (3) F1 score = $2tp/(2tp+fp+fn)$.

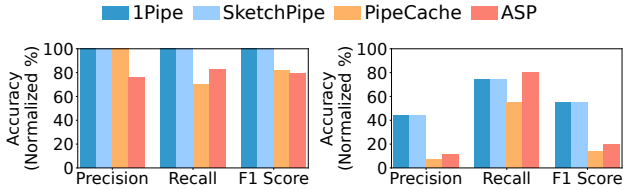
Workloads. By default, we consider representative traffic traces, i.e., WAN traces from CAIDA 2018 [48]. Each trace lasts for one minute and comprises around 26 M packets. We also consider the two traces in Table 3, i.e., an IMC data center trace [24] and an MAWI 2022 trace [13] in Exp#4.

In our testbed, we set the servers that inject traffic into the switch to replay CAIDA traces at their maximum speeds, i.e.,



(a) IMC (HH). (b) MAWI (HH).
(c) MAWI (SS). (d) MAWI (DF).

Figure 7: (Exp#6) Accuracy under diverse traces.



(a) HH (HP) accuracy. (b) SS (UM) accuracy.

Figure 8: (Exp#7) Scaling to multiple applications.

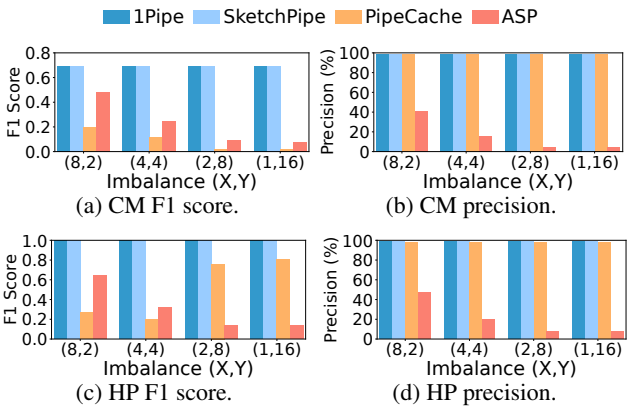
200 Gbps per compound pipeline and 800 Gbps in total. We set each server to only send a specific portion of traces to emulate routing dynamics and measurement imbalances. We split the trace to obtain these portions based on a weighted ECMP-like flow-consistent mechanism [11]. This mechanism enables us to create traffic imbalances by varying the weights of different portions. Here, we define the metric of traffic imbalances: an imbalance of (X,Y) indicates that X random pipelines will receive Y times more traffic than other pipelines. By default, we set $X = 0$, i.e., traffic is uniformly distributed among all compound pipelines.

5.2 Measurement Accuracy

(Exp#1-Exp#4) Detection of heavy hitters, superspreaders, DDoS flows, and global icebergs. Figure 5 presents the accuracy of detecting the flows of applications' interest, e.g., heavy hitters, on our testbed switch. It indicates that SketchPipe achieves near-ideal accuracy. It outperforms other baselines by up to 81% accuracy improvement. Instead, PipeCache only uses half of stages for sketches, leading to suboptimal results.

(Exp#5) Entropy estimation. Figure 6 plots the relative error of computing the entropy of measured flow distributions, i.e., the absolute divergence between the true entropy of our trace and the entropy estimated by each sketch. We highlight: (a) testbed results are the errors measured in our testbed switch; (b) simulation results are the errors when the number of compound pipelines is set to 32. They show a 60% error reduction in SketchPipe over the baselines.

(Exp#6) Accuracy under diverse traces. We repeat Exp#1 under other types of traffic traces, including the MAWI trans-Pacific packet trace [13] and the IMC data center trace [24],



(a) CM F1 score. (b) CM precision.
(c) HP F1 score. (d) HP precision.

Figure 9: (Exp#8) Impact of traffic imbalances.

to evaluate SketchPipe with diverse traffic characteristics. For IMC, we present the results of HH while the results of other applications exhibit the same trend. For MAWI, we consider HH, SS, and DDoS since the MAWI trace exhibits more complex traffic characteristics than IMC, which only contains a few large flows. In Figure 7, the results indicate that SketchPipe achieves the best accuracy and significantly outperforms comparison solutions.

(Exp#7) Scaling to multiple applications. We evaluate if SketchPipe enables high accuracy when scaling to multiple applications. We deploy two sketches, HP and UM, on the switch at the same time. HP supports heavy hitter detection, while UM enables superspreader detection. In particular, we employ such a setting due to two reasons. (1) HP and UM have different designs. HP embraces a space-saving algorithm for HH, while UM adopts universal measurements towards different types of flows of interest. Thus, adopting HP and UM corresponds to two measurement choices: utilizing advanced techniques to address traditional problems, or using an universal and stable approach. (2) HH and SS differ in their monitoring strategies: HH counts the number of packets, while SS counts IP addresses.

In this context, we utilize SketchPipe to place HP on two compound pipelines and UM on another two pipelines of our testbed switch. Figure 8 presents the results of measuring the WAN trace. It indicates that SketchPipe retains high accuracy, which outperforms the baselines by 30%.

Note that for UM, the recall of 1Pipe and SketchPipe are lower than those of ASP. The reason is that ASP suffers from high collision rates, yielding abnormal large counts for each flow. As a result, its number of false negatives is low, which dramatically increases the recall. Correspondingly, its number of false positives is far more than that of SketchPipe, significantly reducing precision and F1 score.

(Exp#8) Impact of traffic imbalances. We evaluate the accuracy of SketchPipe under significant traffic imbalances. Here, we update (X,Y), i.e., X randomly selected pipelines receive Y times more traffic than other pipelines. Figure 9 shows that even with extreme traffic imbalances, SketchPipe still retains

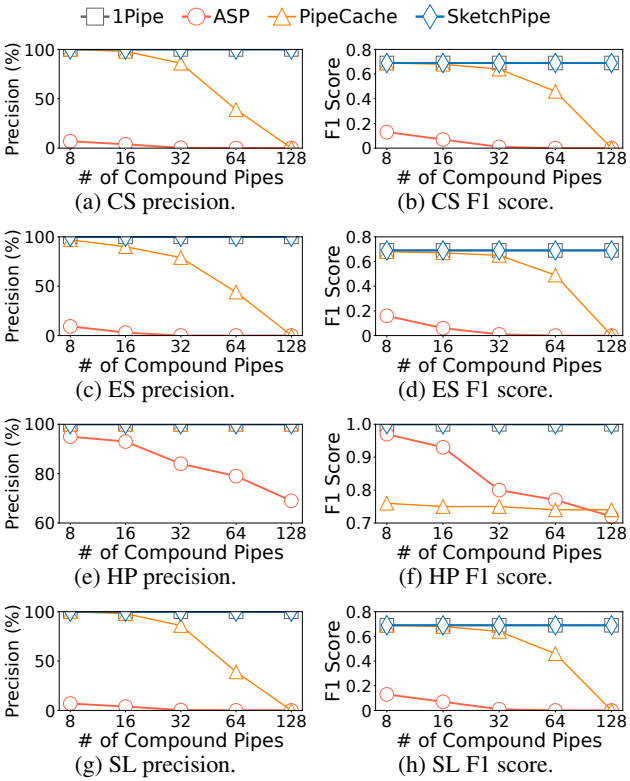


Figure 10: (Exp#9) Scaling to multiple pipelines.

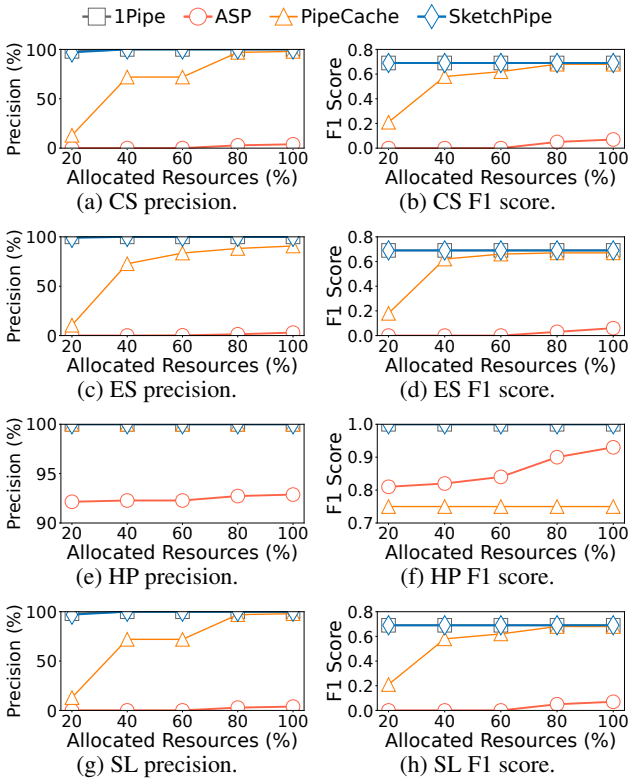


Figure 11: (Exp#10) Impact of switch resources.

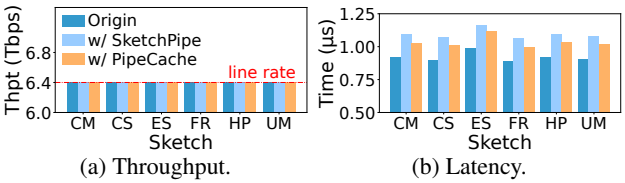


Figure 12: (Exp#11) Impact on switch performance.

near-ideal accuracy, validating its capability of coping with traffic imbalances. We highlight CM and HP. Other sketches follow the same trend.

(Exp#9) Scaling to multiple pipelines. Our testbed switch is limited by the number of compound pipelines. Thus, we simulate the switches with more pipelines. We increase the number of compound pipelines from 8 to 128 while fixing the switch resource capacity to be the same as our testbed switch. At the same time, we repeat Exp#1. In Figure 10, SketchPipe retains high accuracy. ASP suffers from non-trivial accuracy drops when the number of pipelines increases. Even though PipeCache mitigates this issue, it yields sub-optimal results. It drops optimality for most sketches (e.g., Figure 10) and applications (Exp#5). For simplicity, we elide the results of FR and UM, which follow the same trend as CS.

(Exp#10) Impact of switch resources. In some cases, where the switch also executes other in-network functions such as DDoS defense [34], sketches can only use a portion of switch resources. Thus, we vary the amount of switch resources allocated to each sketch in our testbed switch from 20% to 100% to quantify the impact. In Figure 11, SketchPipe uses less resources than other solutions to achieve the same level of accuracy during Exp#1. The reasons are two-fold. (1) ASP suffers from low accuracy in all cases due to its redundant measurements and measurement imbalances. (2) PipeCache only places sketches on the egress so as to enable its ingress caches to direct flows to correct pipelines to reduce redundant measurements (§2.2). As such, sketches can only use a half of stages and cannot access ingress resources. Instead, SketchPipe does not limit the placement locations of sketches, enabling sketches to use all available stages and resources.

5.3 Measurement Overheads

(Exp#11) Impact on switch performance. We study the impact of SketchPipe on the switch performance in terms of throughput and per-packet processing latency. To stress-test the switch performance, we use snake testing [22], a standard way of traffic testing in industry. In each compound pipeline, the first port and last port are traffic ports and connect to two testbed servers. Snake testing connects the $(2i-1)$-th normal port to the $(2i)$-th normal port for $0 < i < \beta/2$, where β is the number of normal ports in each compound pipeline. $\beta = 8$ in our testbed switch. Next, each server injects 100 Gbps traffic to a specific traffic port while the switch directs the traffic to another server via all connected pairs of normal ports. Thus, the above structure sets all ports to receive and send traffic at the same time, thus stress-testing switch performance. Our evaluation at 50% throughput serves as a stress test. Real-world measurement studies indicate that the av-

Table 4: (Exp#12) Switch resource consumption.

Type	PHV	SRAM	MapRAM	TCAM	VLIW	mALU	sALU	Stages
PipeCache	4.64%	2.50%	1.39%	0.00%	4.17%	1.25%	0.00%	15%
SketchPipe	4.64%	1.88%	3.12%	0.00%	3.12%	3.75%	0.00%	15%

Table 5: (Exp#13-#14) Recirc. and buffer consumption.

SketchPipe Cache Size	2^8	2^{10}	2^{12}	2^{14}	2^{16}
Total Number of State Packets	2^8	2^{10}	2^{12}	2^{14}	2^{16}
Buffer Memory Consumption	<0.01%	0.01%	0.05%	0.23%	0.92%
Bandwidth Consumption	<0.01%	<0.01%	<0.01%	<0.01%	0.21%

erage link utilization in production data centers is typically around 20%-40% [40, 56], and WAN link utilization in traces such as CAIDA and MAWI is often below 50%. Thus, our setting represents a high-load scenario in practice.

In Figure 12, we compare the performance of direct port forwarding with the performance when SketchPipe is activated. The impact of SketchPipe on throughput is negligible while the per-packet latency is only increased by 175 ns. SketchPipe does not degrade normal packet processing even at the maximum load. This is because state packets are handled exclusively by the internal recirculation bandwidth, which is isolated from normal packet processing. Furthermore, the number of state packets to drain cached keys is determined by the cache size rather than the traffic volume. As shown in Table 4, the overhead remains constant and small. We also plan to include the maximum load testing in our future work.

(Exp#12) Pipeline resource consumption. We evaluate the switch resource consumption of SketchPipe. Table 4 shows that SketchPipe achieves less than 10% resource consumption, which leaves enough resources to sketch placement.

(Exp#13) Buffer resource consumption. SketchPipe utilizes buffer memory due to its processing of state packets. In detail, since each state packet corresponds to a specific entry in the SketchPipe cache, the total number of state packets equals the SketchPipe cache size (§4.2.2). Hence, we vary the size from 2^8 to 2^{16} when measuring buffer consumption. Table 5 shows that in all cases, the consumption is below 1%, leaving enough room for traffic scheduling.

(Exp#14) Recirculation bandwidth consumption. We also study the impact of state packets on recirculation bandwidth, i.e., the bandwidth consumption on the internal recirculation bandwidth. We repeat Exp#10 while measuring the consumption on recirculation bandwidth. Table 5 presents that SketchPipe can support most i state packets per epoch, where i represents the number of cache entries, and if the injection speed is lower than 2^{16} packets/s, the recirculation utilization can be within 0.21%.

(Exp#15) Scaling to multiple pipelines. Next, we quantify the buffer and recirculation bandwidth overhead of SketchPipe when scaling to a large number of compound pipelines in our simulator. We repeat Exp#7 while fixing the cache size to 2^{16}. In Table 6, the consumption of SketchPipe linearly grows when the pipeline number grows. This is because the number of state packets in each compound pipeline is fixed when the cache size is fixed, limiting the consumption.

Table 6: (Exp#15) Scaling to multiple pipelines.

# of Compound Pipes (N)	4	8	16	32	64
Buffer Memory Consumption	0.92%	1.84%	3.68%	7.36%	14.8%
Bandwidth Consumption	0.21%	0.43%	0.86%	1.72%	3.45%

Table 7: (Exp#16) Impact of hash collision ratios on application-level accuracy (F1 score).

Collision	2%	4%	6%	8%	10%	Collision	2%	4%	6%	8%	10%
HH	0.99	0.99	0.97	0.95	0.94	SS	0.99	0.98	0.98	0.96	0.95

(Exp#16) Impact of hash collision handling. When hash collisions occur, SketchPipe drops new keys (§4.2.1). While this case occurs with a low rate in practice (<5% in Table 2), one may concern that dropping keys affects application-level accuracy. To this end, we intentionally modify the workloads to cause the hash collision rate varying from 2% to 10%. In Table 7, handling hash collisions in SketchPipe brings a negligible impact (<1%) on accuracy.

6 Conclusion

SketchPipe places sketches on multi-pipeline switches while achieving high measurement accuracy. It harnesses splitless sketch placement to eliminate redundant measurements and measurement imbalances. To prevent high overheads, it separates its measurement from packet processing with light cache design and state packet management. Our testbed experiments indicate that SketchPipe improves accuracy by two orders of magnitude for various applications.

In our appendices, we elaborate more on implementation details and discuss several properties of SketchPipe.

Acknowledgement

We would like to thank our shepherd, Prof. Alan Zaoxing Liu, and anonymous TPC reviewers for their constructive comments and insightful suggestions. This work is supported by National Key Research and Development (R&D) Program of China (2023YFB2904000), National Natural Science Foundation of China (No. 623B2090), Joint Funds of the National Natural Science Foundation of China (No. U25B2024, and No. U25B2038), Research Grants Council of Hong Kong (GRF 14201523), Natural Science Foundation of Fujian Province of China (No. 2024J08277), and Science Foundation of the Fuzhou (No. 2025-ZD-023).

Corresponding authors: Chunming Wu, and Haifeng Zhou.

References

[1] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM*, pages 503–514, 2014.

[2] A. Anup, L. Zaoxing, and S. Srinivasan. Heterosketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *USENIX NSDI*, pages 1–23, 2022.

[3] Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.

[4] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, pages 267–280, 2010.

[5] P. Bosshart, D. Daly, G. Gibb, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[6] Broadcom. Tomahawk4, 2019. https://docs.broadcom.com/doc/12398014.

[7] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703, 2002.

[8] X. Chen, Q. Huang, P. Wang, H. Liu, Y. Chen, D. Zhang, H. Zhou, and C. Wu. Mtp: Avoiding control plane overload with measurement task placement. In *IEEE INFOCOM*, pages 1–10, 2021.

[9] X. Chen, Q. Huang, D. Zhang, H. Zhou, and C. Wu. Approsync: approximate state synchronization for programmable networks. In *IEEE ICNP*, pages 1–12, 2020.

[10] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*, pages 226–239, 2020.

[11] M. Chiesa and F. L. Verdi. Network monitoring on multi-pipe switches. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(1):1–31, 2023.

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

[13] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda. Mawilab: combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking. In *ACM CoNEXT*, pages 1–12, 2010.

[14] S. Ghorbani, Z. Yang, et al. Drill: Micro load balancing for low-latency data center networks. In *ACM SIGCOMM*, pages 225–238, 2017.

[15] Gurobi Optimizer. http://www.gurobi.com.

[16] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*, pages 29–42, 2017.

[17] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker. Modular switch programming under resource constraints. In *USENIX NSDI*, pages 1–15, 2022.

[18] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, pages 113–126, 2017.

[19] Q. Huang, P. P. Lee, and Y. Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *ACM SIGCOMM*, pages 576–590, 2018.

[20] Q. Huang, S. Sheng, X. Chen, et al. Toward nearly-zero-error sketching via compressive sensing. In *USENIX NSDI*, pages 1027–1044, 2021.

[21] Intel Corporation. Data Plane Development Kit. http://dpdk.org.

[22] X. Jin, X. Li, H. Zhang, et al. Netcache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*, pages 121–136, 2017.

[23] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. In *USENIX NSDI*, pages 103–115, 2015.

[24] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *ACM IMC*, pages 202–208, 2009.

[25] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*, pages 90–106, 2020.

[26] A. Kumar, M. Sung, J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):177–188, 2004.

[27] Y. Li, J. Gao, E. Zhai, M. Liu, K. Liu, and H. H. Liu. Cetus: Releasing p4 programmers from the chore of trial and error compiling. In *USENIX NSDI*, pages 371–385, 2022.

[28] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: a better netflow for data centers. In *USENIX NSDI*, pages 311–324, 2016.

[29] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, et al. Hpcc: High precision congestion control. In *ACM SIGCOMM*, pages 44–58. 2019.

[30] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *USENIX NSDI*, pages 429–444, 2014.

[31] B. Liu, X. Huang, Q. Li, Z. Huang, Y. Sun, W. Li, J. Zhang, P. Yin, and K. Chen. Ceio: A cache-efficient network i/o architecture for nic-cpu data paths. In *ACM SIGCOMM*, page 381–394, 2025.

[32] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*, pages 334–350, 2019.

[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 *ACM SIGCOMM*, pages 101–114, 2016.

[34] Z. Liu, H. Namkung, G. Nikolaidis, et al. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *USENIX Security*, 2021.

[35] H. Namkung, Z. Liu, D. Kim, et al. Sketchlib: Enabling efficient sketch-based monitoring on programmable switches. In *USENIX NSDI*, pages 1–17, 2022.

[36] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste. Sketchovsky: Enabling ensembles of sketches on programmable switches. In *USENIX NSDI*, pages 1273–1292, 2023.

[37] S. Narayana, A. Sivaraman, V. Nathan, et al. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, pages 85–98, 2017.

[38] P4C. https://github.com/p4lang/p4c.

[39] PktGen. https://pktgen-dpdk.readthedocs.io/.

[40] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, pages 123–137, 2015.

[41] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen. Csamp: a system for network-wide flow monitoring. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 233–246, 2008.

[42] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SOSR*, pages 164–176, 2017.

[43] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *ACM EuroSys*, page 11, 2018.

[44] 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 *USENIX ATC*, pages 823–835, 2018.

[45] C. H. Song, P. G. Kannan, B. K. H. Low, and M. C. Chan. Fcm-sketch: generic network measurements with data plane support. In *ACM CoNEXT*, pages 78–92, 2020.

[46] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *USENIX NSDI*, pages 571–592, 2021.

[47] Tencent Implements Gateway Functionality in P4 Switching Appliance. https://tinyurl.com/4b57cbkk.

[48] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[49] Tofino2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[50] Tomahawk 5 / BCM78900 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78900-series.

[51] Tomahawk 6 / BCM78910 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78910-series.

[52] Tomahawk Ultra / BCM78920 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78920-series.

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

[54] Working With Multiple Objective. https://www.gurobi.com/documentation/9.0/refman/working_with_multiple_obje.html.

[55] 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 *ACM SIGCOMM*, pages 561–575, 2018.

[56] N. Yaseen, J. Sonchack, and V. Liu. tpprof: A network traffic pattern profiler. In *USENIX NSDI*, pages 1015–1030, 2020.

[57] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *ACM SIGCOMM*, pages 207–222, 2021.

[58] Y. Zhang, M. Roughan, W. Willinger, and L. Qiu. Spatio-temporal compressive sensing and internet traffic matrices. In *ACM SIGCOMM*, pages 267–278, 2009.

[59] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *ACM EuroSys*, pages 1–14, 2014.

[60] 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 *IEEE ICNP*, pages 291–301, 2018.

[61] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*, pages 76–89, 2020.

[62] Y. Zhou, Z. Xi, D. Zhang, Y. Wang, J. Wang, M. Xu, and J. Wu. Hypertester: high-performance network testing driven by programmable switches. In *ACM CoNEXT*, pages 30–43, 2019.

[63] L. Zhu, J. Yu, X. Chen, Z. Jiang, X. Liu, J. Zhang, X. Yang, R. Zhang, D. Niyato, X. Yi, et al. Leveraging large language models for multi-objective and adaptive sfc deployment: Techniques, case study, and promising directions. *IEEE Communications Magazine*, 64(1):32–39, 2025.

[64] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*, pages 479–491, 2015.

Appendix 1: More Implementation Details

Measurement keys. SketchPipe supports arbitrary measurement keys such as IP pairs and five tuples. Also, changing flow keys will not affect the quality of SketchPipe's splitless placement or runtime measurement, as it only changes how sketches compute keys. We also evaluate SketchPipe when flow keys are five-tuples. We observe the results similar to §5, indicating no impact of diverse keys.

Measurement epochs. If the current epoch ends, and there are still some state packets waiting to be measured by some sketch arrays, SketchPipe offers two options to users to handle these in-flight state packets. First, in-flight state packets are measured in the subsequent epoch. Second, the flow keys and counts piggybacked in these packets are sent to monitoring applications for measurement, i.e., they are measured in the current epoch. Users can choose one option on demand.

Handling state packet loss. Since state packets are isolated from other packets, the probability of state packet loss is always near zero during our evaluation. So SketchPipe does not intentionally handle such loss. But if a state packet was lost, its target keys will never be measured. To mitigate this issue, SketchPipe allows its users to periodically inject new state packets. Doing so eliminates the impact of loss at the cost of more state packets and slightly higher overheads.

Runtime updates. SketchPipe supports two types of runtime updates, i.e., (1) incremental updates where new sketches are placed on residual resources without affecting existing ones, and (2) recomputing overall placement decisions during device maintenance. If users submit the sketches, which resource requirements far exceed the resource capacity of the switch, placing these sketches saturates resources and leads to failures. In this case, SketchPipe stops placement and sends an alarm indicating switch resource shortage to users. It lets users make decisions on which sketches to place.

Appendix 2: Discussion

Impact of highly skewed traffic and large pipeline number. In SketchPipe, state packet processing will not become a bottleneck even under highly skewed traffic and a large pipeline number. Under highly skewed traffic, a few heavy flows dominate, i.e., only a small number of SketchPipe cache entries are occupied. Thus, only a limited number of state packets are required to drain these entries, keeping the overhead low. These packets operate within the dedicated internal recirculation bandwidth, making them isolated from normal traffic. Also, when the system scales to more pipelines, the available recirculation bandwidth and buffer resources scale proportionally. So the relative overhead of state packets remains minimal.

Handling non-skewed traffic. SketchPipe assumes skewed traffic, i.e., most packets belong to a few flows. But in the worst case, non-skewed traffic, such as sudden bursts, flash crowds, and adversarial workloads like DDoS attacks, increases the number of distinct flows significantly. While this case occurs in a low probability, it brings more hash collisions in the cache. More collisions trigger more frequent evictions, leading to additional state packets to transfer evicted keys to sketch arrays. As such, these in-flight packets may not have time to transfer the data from the cache to target arrays.

SketchPipe offers two options to handle such cases. First, in-flight state packets are measured in the subsequent epoch. In this case, only the measurement in the current epoch suffers from a few accuracy loss, which will soon be mitigated in the next epoch. Also, since large flows consist of massive packets, their errors remain relatively small even when the measurement of some packets is delayed to the next epoch. Second, the flow keys and counts piggybacked in these packets are sent to monitoring applications for measurement, i.e., they are measured in the current epoch. This option preserves full accuracy at the cost of additional bandwidth overhead.

In the future, we plan to integrate sampling-based strategies to further mitigate this issue. Sampling-based techniques [10, 32, 41, 64] are backed by well-established theories and achieve high accuracy under strict resource constraints. We plan to use them for improving the cache design in SketchPipe, e.g., rather than caching every flow key, SketchPipe can selectively sample keys to reduce cache pressure and limit the number of state packets. This can avoid high overhead on internal recirculation bandwidth even under non-skewed traffic.

Cache-enhanced designs. Over the past decade, the benefits offered by building an additional cache have been widely acknowledged by prior studies. For example, Agg-Evict runs a cache before sketches to aggregate the incoming packets on the same flows into batches. As such, sketches operate on batches rather than massive raw packets and thus save the number of memory accesses in software. Similar cache-based techniques have been adopted by other works [22, 31] to reduce per-packet processing overheads.

While SketchPipe also uses a cache, its design goal differs from these studies. Its cache enables asynchronous inter-pipeline measurements rather than a mechanism for reducing memory accesses in prior studies. It temporarily stores flow keys at the end of each ingress pipeline so that synthetic state packets can later retrieve and transfer these keys to sketch arrays in other compound pipelines. Without this cache, achieving splitless placement would require routing every normal packet to traverse all pipelines, which would severely degrade packet processing performance. We leave comprehensive comparisons between SketchPipe 's cache and the caches for reducing memory accesses on other platforms (e.g., software switches) as future work.

Extending to multiple switches. To date, SketchPipe focuses on sketch deployment on a single multi-pipeline switch. But we can extend SketchPipe to the scenarios where multiple sketches are placed on multiple switches. In detail, SketchPipe can be used as a backend that receives network-wide decisions from existing network-wide sketch placement frameworks [2, 63] and enumerates every target switch to perform its workflow. We leave such exploration to future work.

Moreover, SketchPipe relies on simple operations such as hashing, register-based caching, and packet generation, which are common in commodity switches. These operations are not target-specific and can be implemented on other switches that provide similar capabilities. We plan to implement SketchPipe on emerging switches [50, 51, 52] in the future.

Extending to other applications. SketchPipe mainly targets the applications that define heavy keys based on packet counts. It is able to support other applications that consider other metrics. For example, if an application counts per-flow bytes rather than packets, SketchPipe enables this application by simply replacing packet counts in its cache with byte counts. Such modifications also apply to other metrics.