Confluence: Improving Network Monitoring Accuracy on Multi-pipeline Data Plane

Cenman Wang, Ye Tian, Yiwen Wu, and Xinming Zhang

Anhui Key Laboratory on High-Performance Computing, School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, 230026, China
Email: {wangcenman,wyw0530}@mail.ustc.edu.cn, {yetian,xinming}@ustc.edu.cn

Sketch-based method is promising for traffic monitoring in data center networks. Existing data plane programming model (e.g., P4) assumes target switch as one single pipeline, while state-of-the-art programmable switches actually contain multiple independent pipelines. The status quo approach for deploying a sketch-based measurement application on a multi-pipeline switch is to deploy a sketch instance in each pipeline individually. However, under multi-path routing, such a naive approach leads to poor accuracy. To overcome this problem, in this paper, we present *Confluence*, a sketch-based network measurement system for multi-pipeline switches. For monitoring network flows that have packets arrived in bursts and spread over multiple pipelines, Confluence introduces novel data structures to collect short-term traffic statistics in ingress pipelines, and converge the measurement data to egress pipelines. Confluence is carefully designed under the switch hardware constraints, and in particular, to resolve the circular dependency in querying and updating a flow's measurement data from sketch buckets, we propose a novel algorithm and theoretically prove its effectiveness. Both theoretical analysis and experiments driven by real-world traffic traces show that Confluence delivers higher measurement accuracies than existing solutions, especially in the critical task of detecting heavy hitters. Assessment on hardware switch suggests that Confluence is practical for real-world deployment.

Keywords: Network monitoring; Sketch; Network data plane; Multi-pipeline switch

1. INTRODUCTION

Per-flow network measurement, which aims to estimate the size of each flow in the network, plays a critical role in managing high-speed data center networks. To overcome the limitations of the conventional sample-based measurement method (e.g., NetFlow [1], sFlow [2]), sketch-based method emerges and becomes a promising direction in recent years. In such a measurement system, a probabilistic data structure, namely *sketch*, is placed within a switch for collecting per-flow statistics. Sketch-based methods can fulfill a wide range of traffic monitoring tasks including flow size estimation [3], flow size distribution estimation [4], heavy hitters detection [5], etc., and a number of sketch-based measurement systems were proposed and successfully deployed on programmable switches in recent years [6, 7, 8, 9, 3, 10].

Existing sketch-based network monitoring applications are developed with domain-specific data plane programming models like P4 [11]. In such a model, target switch is generally assumed as one single pipeline. On the other hand, for sustaining higher packet rates, state-of-the-art programmable switches already contain multiple pipelines, where each pipeline has dedicated resources and processes packets independently. For example, Intel Tofino 2 contains 4 pipelines [12], and Broadcom Tomahawk has up to 8 pipelines [13]. When a sketch-based application programmed for a single logical pipeline is deployed on a multi-pipeline switch, the status quo approach is to deploy multiple identical sketch instances in all the pipelines in parallel, and each instance independently counts the packets passing its residing pipeline [14, 15].

Although the status quo approach provides a logical equivalence of a single pipeline on a multi-pipeline switch, however, our analysis shows that under the multi-path routing (e.g., ECMP [16]), which is widely applied in production networks, the parallel sketch instances will have a higher hash collision rate, when comparing with a logically equivalent sketch in a single pipeline, under the condition that the two solutions consume same overall memory. The root cause is that when network flows' packets are spread over K ($K \geq 2$) pipelines, each sketch instance must maintain a hash space that is identical to the one maintained by the single-pipeline sketch for accommodating all the flow IDs, but it has only $\frac{1}{K}$ memory resource. Clearly, comparing with a same-sized single-pipeline sketch, the parallel sketch instances will have a lower measurement accuracy due to the higher hash collision rate.

To solve the problem within the data plane, one

feasible solution, with *PipeCache* [17, 18] as the only example, is to decouple a packet's egress pipeline from the pipeline that counts this packet, and allow only one pipeline to count all packets of a network flow assigned to it. In PipeCache, packets store their summaries in a number of caches maintained by ingress pipelines, and the cached summaries are piggybacked on subsequent packets to their corresponding egress pipelines, where they are counted by dedicated sketch instances. Although PipeCache reduces hash collisions, however, the cache-based solution is susceptible to cache overflow, due to the bursty nature of the network traffic in data centers [19, 20], as when a burst size exceeds the cache size, the cache overflows.

In this paper, we present *Confluence*, a sketch-based network measurement system for multi-pipeline data plane. In a Confluence system on a switch containing K ($K \geq 2$) pipelines, we divide the entire flow ID hash space into K equal-sized sub-spaces. Each egress pipeline hosts a sketch (namely the *main sketch*) that covers a sub-space, and monitors the network flows with their flow IDs falling in this sub-space. Since each main sketch maintains only $\frac{1}{K}$ of the entire hash space with $\frac{1}{K}$ memory resource, Confluence avoids the high hash collision rate as in the status quo approach. For overcoming the problem that a network flow is routed to an egress pipeline different from the one that monitors the flow, we place an auxiliary sketch (referred to as the *delta sketch*) in the ingress pipeline. The delta sketch accumulates short-term traffic statistics of the flow received by the ingress pipeline, and piggybacks the statistics on packets routed to the corresponding egress pipeline that monitors the flow to update the main sketch. Note that unlike PipeCache, Confluence does not clone and recirculate any packet, thus avoids the significant overhead incurred by packet recirculation. In the design, implementation, and evaluation of Confluence in this paper, we make the following contributions.

- We present Confluence for monitoring network flows on multi-pipeline data plane. By introducing two data structures, namely *pipeline table* and *delta sketch*, in each ingress pipeline, we design a novel mechanism for a switch to accumulate short-term flow statistics in ingress pipelines, and piggyback the statistics to update the *main sketches* in egress pipelines. We show that Confluence is accurate under bursty traffic, and theoretically prove that it has a lower error bound than the status quo approach.
- We extend the basic design of Confluence to cope with the packet processing constraints of commodity hardware programmable switches. In particular, we interleave two instances of the data structures in ingress pipeline to make sure that all the packets access the pipeline stages in a fixed sequential order, and propose an algorithm to ensure that when querying and updating the delta sketch, a packet accesses at most one address of a stage-local register in its pipeline pass, as constrained by the hardware switch.
- We evaluate Confluence and compare it with PipeCache and other benchmarks. We find that Confluence delivers higher measurement accuracy under bursty traffic than its counterparts, and establishes its advantages under many circumstances, including the ones when there are limited in-switch memories, when the switch contains many pipelines, and when the workloads imposed on different pipelines are imbalanced. By implementing and assessing with a hardware programmable switch, we show that Confluence is practical for real-world deployment.

For the remainder part of this paper, Sec. 2 introduces background and motivation; We present and analyze Confluence in Sec. 3; Experiment results are discussed in Sec. 4; Sec. 5 surveys the related works and we conclude this paper in Sec. 6.

2. BACKGROUND AND MOTIVATION

In this section, we first introduce the architectural model of multi-pipeline switches, then we describe the challenges in applying sketch-based measurement applications on multi-pipeline switches. We analytically show that the status quo approach leads to low measurement accuracy, and the existing solution has its limitation.

2.1. Multi-pipeline Switch

In programmable switches based on Protocol Independent Switch Architecture (PISA) [21, 22, 23], packets are processed in programmable *pipelines*. Typically, a programmable pipeline contains an *ingress* pipeline and an *egress* pipeline, which are separated by components like packet buffer and traffic manager. A pipeline is composed of multiple match+action unit (MAU) stages, where each stage applies the specified actions on the packets filtered out by the match conditions. Moreover, each MAU stage has a stage-local memory, and stateful elements such as counters and sketch bucket arrays are realized as *registers* in the memory.

Similar to multi-core CPUs, state-of-the-art programmable switches currently employ multiple pipelines for sustaining higher packet rates. For example, Intel Tofino switch contains 4 pipelines [12] and Broadcom Tomahawk switch has up to 8 pipelines [13]. As illustrated in Fig. 1, in a typical multi-pipeline switch, ports are statically assigned to pipelines without overlaps, pipelines are parallel without sharing any resources and process packets independently, and multiple ingress and egress pipelines are inter-connected with a crossbar. Unfortunately, although a physical switch may contain multiple pipelines, the domain-specific data plane programming model like P4 [11] still abstracts the target switch as one single logical pipeline, regardless how

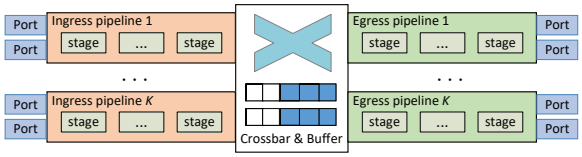
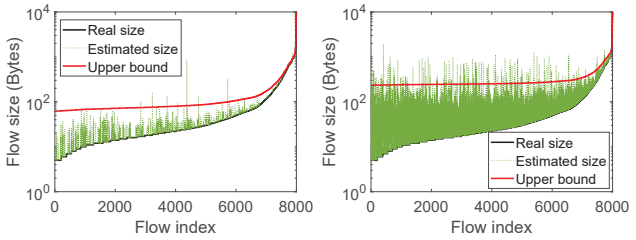


FIGURE 1. Illustration of multi-pipeline switch architectural model.



(a) Flow size estimation (b) Heavy hitter detection

FIGURE 2. Flow size estimations made by (a) a single-pipeline CM sketch and (b) ParallelSketch using 8 CM instances.

many physical pipelines the switch actually has. For deploying an application written for a single logical pipeline on a multi-pipeline target switch, the application is indeed deployed to each pipeline individually [14, 15].

To enable the sharing of packet processing states (e.g., counters) across multiple pipelines, new switch architectures such as MP5 [24] and OptimusPrime [25] are proposed in recent years. However, the new generation architectures have yet been supported by commodity hardware switches, and how to ensure the correctness and performance of the applications written for a logical single pipeline on the multi-pipeline switch is a critical issue [14, 15].

2.2. Challenge and Motivation

Sketch-based method is a promising direction for fast and accurate per-flow measurement in data center networks [26, 27, 28, 6, 29, 30, 31, 7, 8, 10, 3]. In a sketch-based measurement system, a probabilistic data structure, namely *sketch*, is usually placed within a switch for aggregating per-flow statistics. More specifically, when receiving a packet, a switch computes hash values from the packet's flow ID (e.g., the 5-tuple), locates a bucket from each array, and updates the bucket value.

2.2.1. Limitation of status quo approach

As above introduced, existing sketch-based network measurement applications are written for a single logical pipeline. In particular, when an application constructs a sketch data structure for single logical pipeline, after deploying on a multi-pipeline switch, multiple sketch instances will be deployed in all the pipelines and count packets in parallel [14, 15]. We refer to such a status quo approach as *ParallelSketch* in this paper.

Although ParallelSketch provides logically equivalent functionalities on a multi-pipeline switch, however, its performance is poor due to the wide usage of multi-path routing techniques. Taking Google's B4 SDWAN network [32, 33] as an example, in B4, each packet has two IP headers, where the destination address of the outer IP header is indeed a tunnel ID, which decides the egress port (and consequently, the egress pipeline) that the packet should be directed to in an ECMP manner, while the packet's 5-tuple flow ID is defined by the inner IP and the transport layer header. In B4, a flow's packets may carry different tunnel IDs and could be received by and routed to different ingress and egress pipelines. If we place parallel sketch instances in egress pipelines, then for monitoring a network flow, measurement data from all the instances in different egress pipelines must be aggregated.

When a network flow has its packets spread over multiple egress pipelines due to the multi-path routing, each sketch instance in an egress pipeline may encounter packets from all the flows, thus it needs to maintain a hash space for accommodating all the flow IDs. Comparing with a single-pipeline switch, given the same overall memory usage, sketch instances in multiple parallel pipelines would have higher collision rates and be less accurate. For a particular instance, consider that a count-min (CM) sketch [26], which has d rows and w columns of buckets, is placed in a single-pipeline switch to measure a set of network flows with a total number of N packets. According to [26], for a flow f of size n_f, with a probability of $1 - e^{-d}$, its estimation n'_f by the CM sketch is bounded by

$$n'_f \leq n_f + \frac{e}{w} \times N \tag{1}$$

where e is Euler's number.

Now suppose that we apply ParallelSketch on a switch containing K parallel pipelines to measure the same set of network flows, and with the same overall in-switch memory usage, we resize the CM sketch instance in each egress pipeline as d rows and $\frac{w}{K}$ columns of buckets. If each sketch instance randomly encounters $\frac{1}{K}$ of the total N packets, then similar to the single-pipeline case, with a probability of $1 - e^{-d}$, the size of the flow f estimated by the sketch instance in the i^{th} egress pipeline, denoted as $n'_{f,i}$, is bounded by

$$n'_{f,i} \leq n_{f,i} + \frac{Ke}{w} \times \frac{N}{K} \tag{2}$$

where $n_{f,i}$ is the ground-truth size of the flow f in the i^{th} pipeline. By aggregating the estimations from all the parallel sketch instances, it is easy to see that with a probability of $(1-e^{-d})^K \approx 1-K\times e^{-d}$, the aggregated estimation of the flow f is bounded by

$$n'_f = \sum_{i=1}^{K} n'_{f,i} \leq n_f + \frac{Ke}{w} \times N \tag{3}$$

By comparing (3) with (1), one can see that given the same in-switch memory budget, ParallelSketch is much less accurate than a single-pipeline switch. To show this, Fig. 2 compares the sizes of 8,000 flows estimated by the two approaches. In Fig. 2(a), we employ a 2×2^{15} CM sketch in a single-pipeline switch, and in Fig. 2(b), eight 2×2^{12} CM sketch instances are deployed in 8 pipelines. We also plot the upper bounds in (1) and (3) in the figures for comparison. From the figures, we can see that ParallelSketch has higher upper bound and over-estimates more severely than single-pipeline, and its estimations exceed the upper bound more often.

Note that although we focus on sketch-based network monitoring applications in this paper, however, the lack of inter-pipeline state-sharing likewise affects non-sketch applications. For example, to detect heavy hitter flows (i.e., the top-k largest flows), if we deploy a detector such as HashPipe [34] in each individual pipeline, then different pipelines will make different decisions based on their pipeline-local views independently, and lead to errors in the final detection result. Given that multi-path routing is widely adopted by many network designs besides B4, such as VL2 [35], Presto[36], Hermes [37], RotorNet [38], etc, the problem widely exists.

2.2.2. Limitation of existing solution

PipeCache [17, 18] is, to our best knowledge, the only work that addresses the above-described problem within the switch data plane. In PipeCache, a sketch instance is deployed at each egress pipeline, and a flow is deterministically mapped to one fixed egress pipeline, which is referred to as the flow's *monitor* pipeline, for counting all its packets. Since a flow's packets could be spread over all the pipelines, in each ingress pipeline, PipeCache places multiple caches, one for each egress pipeline, to store summaries of the packets that should be counted by that pipeline.

More specifically, when an ingress pipeline receives a packet that should be routed to the j^{th} egress pipeline and be monitored by the sketch instance in the k^{th} egress pipeline, besides caching the packet's summary to the cache corresponding to the k^{th} pipeline, PipeCache piggybacks the summaries stored in the cache corresponding to the j^{th} pipeline to this packet, and the packet carries the summaries to update the sketch instance in the j^{th} egress pipeline. By counting all the packets of a flow with one single sketch instance, PipeCache reduces the collisions in ParallelSketch.

In PipeCache, a cache is built up by the packets whose flows are mapped to the corresponding monitor pipeline, and is drained by the packets that are routed to that pipeline. If we use λ and μ to denote the arrival rates of the two types of the packets, then to avoid cache overflow, a packet must carry more than $\frac{\lambda}{\mu}$ summaries in its metadata.

However, the bursty nature of the network traffic, which is widely observed in data center networks [19, 20], leads to very large instantaneous λ values and causes caches to over-flow. For example, Kapoor et al. [19] reports that end-host tends to send bursts of packets to the same destination back-to-back. There are many sources for bursts, for example, the mechanisms of Generalized Receive Offload (GRO) and Generalized Segmentation Offload (GSO) in the Linux kernel, and the TCP Segmentation Offload (TSO) and Large Receive Offload (LRO) features supported by the NIC hardware, enable end-hosts to send and receive a large virtual packet, whose size is up to 64 kB, by breaking the virtual packet into many MTU-sized packets back-to-back in the network. Furthermore, study shows that packet bursts can be further enlarged with Interrupt Coalescing (IC), TCP congestion controls, and Linux queueing disciplines [20]. Clearly, when traffic under the monitoring contains bursts whose sizes exceed the size of a cache in PipeCache, the cache overflows.

On the other hand, the number of packet summaries a packet can carry in its metadata, is constrained by two factors. The first constraint is that today's programmable switch only allows a packet to carry limited user-defined metadata when being routed from an ingress pipeline to an egress one. For example, our experience on the Intel Tofino switch shows that the egress parser can parse at most 160 Bytes, which include all the layer-2 to layer-4 headers as well as the intrinsic metadata such as `in_port` and `timestamp`, which means that only a few dozens of Bytes are left for the piggybacked packet summaries. The second and most critical factor is that in hardware programmable switch, a packet can access a register only once in its pipeline pass [8, 9, 18], which means that for enabling a packet to carry multiple summaries, the switch needs to divide a cache into multiple registers and place in different MAU stages. In fact in PipeCache, caches and other data structures are divided into four slices and realized as separated registers, and a packet can carry up to four summaries in its metadata.

We use an example in Fig. 3 to show the limitation of PipeCache under bursty traffic. In the figure we consider a switch containing two pipelines, and handles packets of two network flows f and g, which have their packets counted by the sketch instances in egress pipeline #1 and #2 respectively. Packets of flow f are routed to egress pipeline #2, and each packet can carry at most one packet summary in its metadata due to the register access constraint. As shown in the figure, at time t_0, a burst of four packets of flow g arrives to ingress pipeline #1, followed by one packet of flow f, and at time t_1, the four packets of flow g have passed through ingress pipeline #1, but only the first two packets have their summaries cached due to the limited cache space. At time t_2, the packet of flow f, which is routed to egress pipeline #2, carries one packet summary of flow g to update the sketch instance in the pipeline. From the example we can see that although 4 packets of flow g arrives, the sketch instance in egress

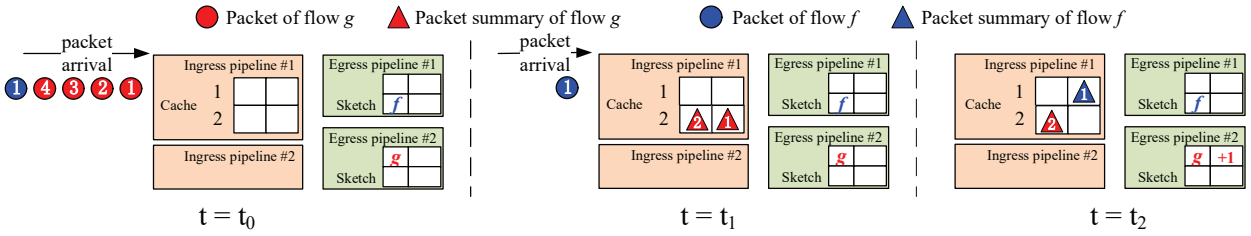


FIGURE 3. Demonstration of PipeCache's limitation. PipeCache counts only one packet of flow g at time t_2 eventually, despite that four packets of the flow arrive at time t_0.

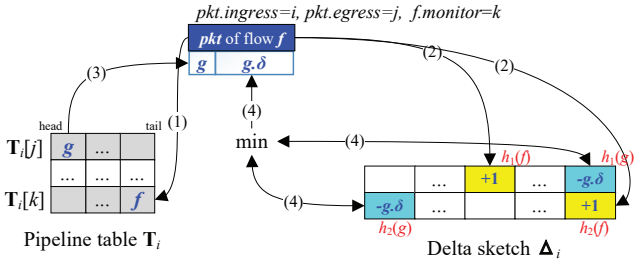


FIGURE 4. Steps of ingress pipeline packet processing in the basic design of Confluence.

pipeline #2 counts only one packet, due to the reasons of packet burst, limited cache size, and register access constraint.

To combat cache overflow, PipeCache proposes a method based on packet clone and recirculation. In such a method, when a packet finds that the cache is full, PipeCache clones a copy of the packet, and recirculates the copy to enter the ingress pipeline again, but this time the packet is routed to its monitor pipeline. Nevertheless, packet cloning and recirculating incur non-trivial overhead, as the cloned packets consume additional processing capacity and queueing space within the switch. Moreover, cloning and recirculating a large amount of packets may cause congestions, increase the chance of packet dropping, and reduce the switch's forwarding performance. For this reason, *in this work, we aim to avoid packet clone and recirculation.*

3. THE CONFLUENCE SYSTEM

In this section, we present *Confluence*, a network measurement system for multi-pipeline switches. We first present the basic design of Confluence that is agnostic of the hardware constraints, then we develop a more practical system ready to be deployed on commodity programmable switches.

3.1. Basic Design and Analysis

3.1.1. Measurement data structures

In Confluence, a packet pkt of a network flow f is associated with three pipelines: ingress pipeline, egress pipeline, and monitor pipeline. The ingress pipeline $pkt.ingress$ is the pipeline that receives the packet, the switch routes the packet to the egress pipeline $pkt.egress$, and the monitor pipeline $f.monitor$ is the egress pipeline responsible for counting all the packets of flow f. Note that a flow's packets do not necessarily have the same ingress and egress pipelines, but they share the same monitor pipeline, which is decided by hashing the flow ID.

Considering a multi-pipeline switch containing K parallel pipelines, Confluence introduces two data structures in each ingress pipeline: a *pipeline table* and a *delta sketch*. For the i^{th} ingress pipeline, we denote its pipeline table as $\mathbf{T}_i$ and its delta sketch as $\mathbf{\Delta}_i$. The pipeline table $\mathbf{T}_i$ has K entries, one for each egress pipeline, and each pipeline table entry is logically a FIFO queue that can store up to c flow IDs. When $\mathbf{T}_i[k].tail = f$, it means that flow f's monitor pipeline is the k^{th} egress pipeline, i.e., $f.monitor = k$, and one or more packets of f have recently been received by the i^{th} ingress pipeline.

The delta sketch $\mathbf{\Delta}_i$ counts the packets that have recently been received by the i^{th} ingress pipeline but their flow statistics have yet been updated to the corresponding monitor pipelines. As a standard count-min (CM) sketch, $\mathbf{\Delta}_i$ has d_1 counter arrays, and each array contains w_1 buckets. $\mathbf{\Delta}_i$ employs d_1 pair-wise independent hash functions, denoted as $h_r(\cdot)$, $0 \le r < d_1$, to map network flows to different buckets in each array.

Each monitor pipeline, say the j^{th} egress pipeline, maintains a sketch for monitoring any network flow f with the flow's monitor pipeline assigned to it, i.e., $f.monitor = j$. We refer to such a sketch as *main sketch*, which keeps the final measurement results. Main sketch is updated by the data piggybacked on the packets from different ingress pipelines. Any sketch-based data structure can be used for realizing the main sketch, and in this work, we choose Elastic Sketch [6], because of its high accuracy and hardware deployability.

3.1.2. Packet processing

In Confluence, measurement data structures are updated by packets. Suppose that the i^{th} ingress pipeline receives a packet pkt of flow f, i.e., $pkt.ingress = i$, and the packet pkt will be routed to the j^{th} egress pipeline and should be counted by the k^{th} egress pipeline, i.e., $pkt.egress = j$ and $f.monitor = k$. As demonstrated in Fig. 4, pkt is processed by the i^{th} ingress pipeline with the following steps:

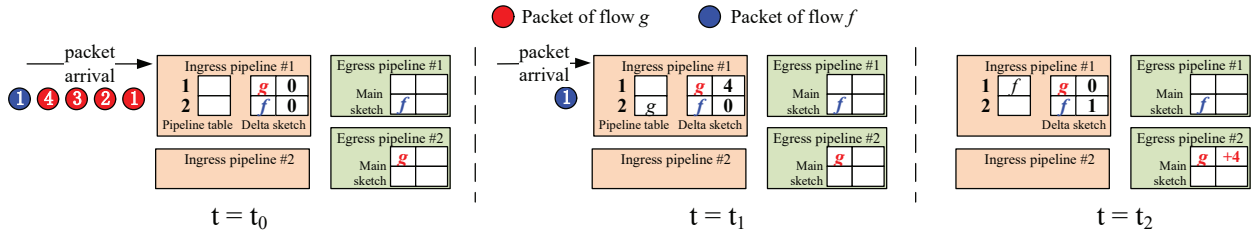


FIGURE 5. Demonstration of Confluence. At time t_2, Confluence counts all the four packets of flow g, under the presences of the packet burst.

- **Step 1**: Insert the flow ID f to the tail position of the pipeline table entry $\mathbf{T}_i[k]$, if the queue is full, evict the element at the head position.
- **Step 2**: Look up the delta sketch $\mathbf{\Delta}_i$ with f, and increase the value of each bucket $\mathbf{\Delta}_i[r][h_r(f)]$, for $r = 0, \cdots, d_1 - 1$.
- **Step 3**: Read and remove the flow ID g from the head position of the pipeline table entry $\mathbf{T}_i[j]$ and copy g to pkt's metadata. Note that g is a flow whose monitor pipeline is the j^{th} egress pipeline, i.e., $g.monitor = j$.
- **Step 4**: Look up the delta sketch $\mathbf{\Delta}_i$ with g, copy $g.\delta = \min\{\mathbf{\Delta}_i[r][h_r(g)] | r = 0, \cdots, d_1 - 1\}$ to pkt's metadata, and remove flow g's measurement data from the delta sketch by deducting each delta sketch bucket value by $g.\delta$, i.e., $\mathbf{\Delta}_i[r][h_r(g)] = \mathbf{\Delta}_i[r][h_r(g)] - g.\delta$, for $r = 0, \cdots, d_1 - 1$.
- **Step 5**: Route pkt with the metadata $\langle g, g.\delta \rangle$ to the j^{th} egress pipeline.

When the packet pkt with the metadata $\langle g, g.\delta \rangle$ is routed to the j^{th} egress pipeline, Confluence increases the counters corresponding to flow g in its main sketch by an amount of $g.\delta$.

To demonstrate how packets are handled by Confluence, we revisit the example in Fig. 3, and show it in Fig. 5, in which we simply replace PipeCache's packet summary cache with a pipeline table and a delta sketch in each ingress pipeline, and assume that each pipeline table entry can store only one flow ID (i.e., $c = 1$). As shown in the figure, at time t_0, a burst of four packets of flow g and one packet of flow f arrive to ingress pipeline #1. After the four packets of flow g have passed through the ingress pipeline at time t_1, they write their flow ID, i.e., g, to the pipeline table entry corresponding to egress pipeline #2, and the burst size, i.e., 4 packets, is counted by the delta sketch in ingress pipeline #1. At time t_2, after the packet of flow f writes its flow ID to the pipeline table, it reads flow ID g from the table entry corresponding to its egress pipeline (i.e., egress pipeline #2), retrieves and deducts the counter value of g from the delta sketch, and updates the main sketch in its egress pipeline. Note that different from the case in Fig. 3, all the 4 packets of flow g are counted by Confluence as shown in Fig. 5.

By comparing the examples in Fig. 3 and Fig. 5, we can see that unlike PipeCache, Confluence does not cache raw per-packet summaries, but accumulates short-term flow statistics temporarily in the delta sketches before updating the main sketches. Since counters are much less likely to overflow than caches, Confluence can better preserve the measurement accuracy under the bursty network traffic.

3.1.3. Analysis

We formally analyze Confluence's measurement accuracy. Without loss of generality, we consider a switch containing K pipelines, and besides the pipeline table and the CM delta sketch containing $d_1 \times w_1$ buckets in the ingress pipeline, we suppose that the main sketch in each egress pipeline is realized as an Elastic Sketch [6]. In particular, the Elastic Sketch contains a hash table heavy part and its light part is a standard CM sketch composed of $d_2 \times w_2$ buckets.

We have the following result regarding Confluence's measurement accuracy.

THEOREM 3.1. *For a network flow f of size n_f, with a probability of $(1-e^{-d_1}) \times (1-e^{-d_2}) \approx 1-e^{-d_1}-e^{-d_2}$, its size n'_f estimated by Confluence is bounded by*

$$n'_f \leq n_f + (\frac{e}{w_1} + \frac{1}{K} \times \frac{e}{w_2}) \times N \tag{4}$$

where N is the total number of the packets.

Proof. Since the heavy part of the Elastic Sketch does not introduce errors [6], there are only two sources of measurement errors in n'_f: 1) errors introduced by the delta sketches in the ingress pipelines, and 2) errors introduced by the CM sketches as the Elastic Sketch light parts in the egress pipelines. For the first error source, if we view the combination of the delta sketch instances in all the ingress pipelines as one $w_1 \times d_1$ CM sketch, where each bucket is K times larger, then according to [26], with a probability of $1 - e^{-d_1}$, the estimation n''_f of f accumulated by the combined CM sketch is bounded by

$$n''_f \leq n_f + (\frac{e}{w_1}) \times N \tag{5}$$

Since in Confluence, n''_f is used to update the main sketch, and each main sketch averagely covers $\frac{1}{K}$ of the total network traffic, therefore for the second error source, with a probability of $1 - e^{-d_2}$, the final

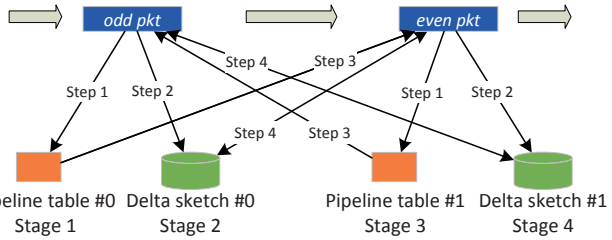


FIGURE 6. Placement of pipeline table and delta sketch instances and operations executed by odd/even packets.

estimation n'_f from the main sketch is bounded by

$$n'_f \leq n''_f + (\frac{e}{w_2}) \times \frac{N}{K} \tag{6}$$

Summing up the two errors, it is easy to see that with a probability of $(1-e^{-d_1}) \times (1-e^{-d_2}) \approx 1-e^{-d_1}-e^{-d_2}$, we have

$$n'_f \leq n_f + (\frac{e}{w_1} + \frac{1}{K} \times \frac{e}{w_2}) \times N \tag{7}$$

□

3.2. Tackling Hardware Constraints

3.2.1. Constraints from hardware programmable switches

We aim to deploy Confluence on commodity hardware programmable switches (e.g., the Tofino-based switch), which have the following constraints in processing network packets. First, a packet always moves forward within a pipeline, so if a number of measurement data structures are placed at different stages in a pipeline, all the packets access the structures in a fixed sequential order [22]. Second, as reported by many studies (e.g., [8, 9, 18]), a packet can access at most one address of a stage-local register in its pipeline pass.

Unfortunately, the basic design of Confluence that we have described in Sec. 3.1 violates the above constraints, and is infeasible to be implemented on commodity hardware programmable switches. First, as shown in Fig. 4, the pipeline table and each array of the delta sketch, which are realized as registers, are accessed *twice* per packet: each packet needs to write the flow ID f and read the flow ID g from/to different entries of the pipeline table, and it needs to increase the bucket value of f, retrieve and decrease the bucket value of g at different positions of each delta sketch bucket array. However, as we have mentioned, a register can be accessed by a packet at most once.

Second, as illustrated by Fig. 4, when retrieving $g.\delta$ and removing it from flow g's measurement data in the delta sketch as in step 4, there exists a *circular dependency*, that is, each bucket of flow g $\mathbf{\Delta}_i[r][h_r(g)]$, $r = 0, \cdots, d_1 - 1$, needs to be decreased by an amount of $g.\delta$ when being accessed by the packet of f, but $g.\delta = \min\{\mathbf{\Delta}_i[r][h_r(g)] | r = 0, \cdots, d_1 - 1\}$ is available only after all the buckets have been accessed by this packet.

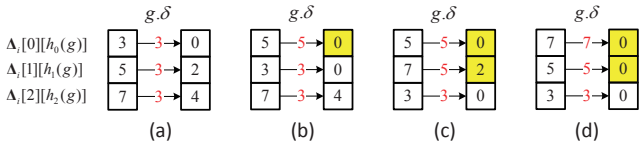


FIGURE 7. Examples of Algorithm 1 executions, where buckets being overly decreased are highlighted.

3.2.2. Tackling register access constraints

To overcome the first problem, we propose to place two instances of pipeline table and delta sketch, which are indexed as 0 and 1, in each ingress pipeline. As shown in Fig. 6, conceptually[1], the 0-indexed pipeline table is placed at stage 1, the 0-indexed delta sketch is placed at stage 2, the 1-indexed pipeline table is placed at stage 3, and the 1-indexed delta sketch is placed at stage 4 of the ingress pipeline.

We label each packet received by a pipeline as either *odd* or *even* alternatively. All the packets access the pipeline table and delta sketch instances with a fixed sequence order from stage 1 to 4, but as illustrated in Fig. 6, the odd/even packets perform different operations on the measurement data structures in different stages as the following.

- An odd packet executes step 1 as in Fig. 6 by writing f's flow ID to the 0-indexed pipeline table, increases f's sketch buckets as in step 2 in the 0-indexed delta sketch, executes step 3 by reading g's flow ID from the 1-indexed pipeline table, removes g's measurement data and retrieves $g.\delta$ from the 1-indexed delta sketch as in step 4.
- On the other hand, an even packet executes step 3 on the 0-indexed pipeline table, performs the operations in step 4 on the 0-indexed delta sketch, executes step 1 on the 1-indexed pipeline table, and performs the operations in step 2 on the 1-indexed delta sketch.

As a result, odd/even packets have their traffic statistics traced by the 0/1-indexed pipeline table and delta sketch, and their measurement data is removed and carried by the even/odd packets to their monitor pipelines. It is easy to see that in Fig. 6, each packet accesses the data structures in same fixed order, and a packet accesses each measurement data structure exactly once, thus avoids violating the register access constraint.

3.2.3. Resolving circular dependency

To resolve the circular dependency in retrieving $g.\delta$ and deducting it from the delta sketch buckets of flow g as in step 4, we propose Algorithm 1 as follows.

In Algorithm 1, a packet of flow g accesses the hashed bucket $\mathbf{\Delta}_i[r][h_r(g)]$ in each delta sketch array sequentially, thus eliminates the circular dependency.

[1]In practice, realizing a delta sketch instance may require d_1 consecutive stages, one stage for realizing a sketch bucket array.

Algorithm 1 Algorithm for step 4

INPUT: Flow ID g

```
1: Initially, g.δ ← null;
2: for r ← 0, ··· , d1 − 1 do
3:     if g.δ == null then
4:         g.δ ← Δi[r][hr(g)], Δi[r][hr(g)] ← 0;
5:     else
6:         g.δ ← min(Δi[r][hr(g)], g.δ);
7:         Δi[r][hr(g)] ← Δi[r][hr(g)] − g.δ;
8:     end if
9: end for
10: return g.δ;
```

When accessing a bucket, the algorithm first checks whether $g.\delta$ has a valid value (line 3), if not, it sets $g.\delta$ as value of the currently accessed bucket $\mathbf{\Delta}_i[r][h_r(g)]$, and clears the bucket as 0 (line 4). Otherwise, it compares the bucket value $\mathbf{\Delta}_i[r][h_r(g)]$ with $g.\delta$: if the former is greater, it deducts $g.\delta$ from $\mathbf{\Delta}_i[r][h_r(g)]$; otherwise, the algorithm sets $g.\delta$ as $\mathbf{\Delta}_i[r][h_r(g)]$, and clears the bucket as 0 (line 6-7).

Fig. 7 presents 4 cases of the algorithm execution. In each case, the algorithm accesses $r = 3$ buckets of flow g in different bucket arrays sequentially from top to bottom, and after accessing all the three buckets, $g.\delta$ is always set as $\min\{\mathbf{\Delta}_i[r][h_r(g)]|r = 0, 1, 2\} = 3$ correctly. Meanwhile, in the cases in Fig. 7(b)-(d), some buckets are overly decreased, that is, the bucket value is deducted by an amount greater than $g.\delta$.

Note that unlike CU [27, 39], which also updates bucket value conservatively, Algorithm 1 does not need to traceback the smallest bucket value, thus is hardware-friendly. Algorithm 1 shares some similarity with LightGuardian [8] and Count-Less [40], but LightGuardian and Count-Less apply conservative strategies to increase bucket values, while Algorithm 1 decreases the values of the sketch buckets and obtains the minimum value simultaneously. For Algorithm 1, we have the following result.

Theorem 3.2. *Algorithm 1 retrieves $g.\delta$ correctly, i.e., $g.\delta = \min\{\mathbf{\Delta}_i[r][h_r(g)]|r = 0, \cdots, d_1 - 1\}$, and the algorithm only overly decreases a bucket whose value is over-estimated.*

Proof. From Algorithm 1, one can see that the only way to change the value of $g.\delta$ is to assign it with the value of the currently accessed bucket, under the condition that $g.\delta == null$ (line 3-4) or $g.\delta$ is greater than the currently accessed bucket value (line 5-7). So after accessing all the buckets, $g.\delta$ will be assigned with the minimum value of the buckets, i.e., $g.\delta = \min\{\mathbf{\Delta}_i[r][h_r(g)]|r = 0, \cdots, d_1 - 1\}$.

In Algorithm 1, a bucket has its value overly decreased under the only condition that when this bucket is accessed, $g.\delta$ is greater than its final value, which means that the bucket corresponding to the minimum value $\min\{\mathbf{\Delta}_i[r][h_r(g)]|r = 0, \cdots, d_1 - 1\}$ has not been accessed by the packet yet. Since the currently accessed bucket is not the one that has the minimum value, its value is over-estimated. □

Since both the delta sketch and the main sketch tend to over-estimate [26], Theorem 3.2 suggests that by overly decreasing a bucket whose value is over-estimated, Confluence can partially offset the over-estimation errors.

4. EVALUATION

4.1. Experiment Setup

We conduct extensive experiments to evaluate Confluence, and in particular, we examine and compare the following network measurement systems.

- **SinglePipe**: This is the case that the switch has only one single pipeline, and hosts an Elastic Sketch for monitoring all the network flows.
- **ParallelSketch**: This is the status quo approach as we have described in Sec. 2.2.1, in which an Elastic Sketch instance is placed in each egress pipeline and works independently.
- **PipeCache**: PipeCache is proposed in [17, 18]. As in [18], we recirculate as many packets as needed so that all the packets are counted in the correct monitoring pipeline. A packet summary is its 5-tuple flow ID, which is the minimum information required for the egress pipeline to count this packet. We divide each data structure in PipeCache into 4 slices as suggested in [18].
- **PipeCache***: PipeCache* has the same configuration of PipeCache except that it disallows packet clone and recirculation for the reason as explained in Sec. 2.2.2.
- **Confluence**: This is the system presented in Sec. 3, and the system complies with all the hardware constraints.

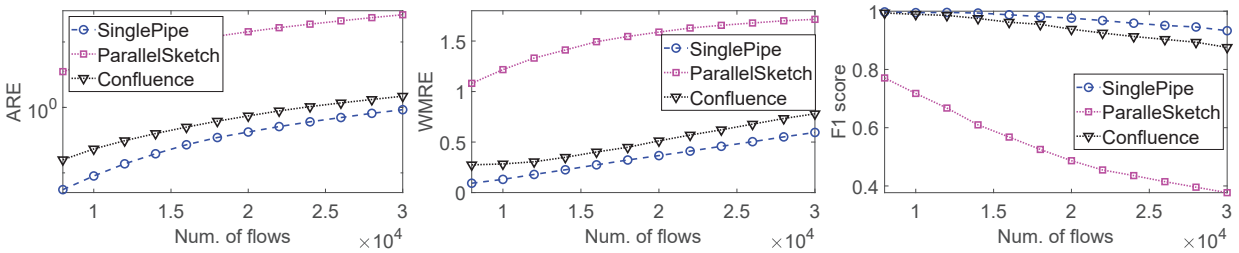
We have implemented the above systems with Python for simulation. We have also implemented Confluence on the `bmv2` software switch that supports P4 programming [41], and the Edgecore Wedge 100BF-32X Tofino-based hardware switch[2]. For enabling fair comparison, we stick to the following principles in our evaluation. First, all the systems should have approximately same overall memory usages for hosting their measurement data structures. Second, for the systems of PipeCache/PipeCache* and Confluence, which place measurement data structures in both ingress and egress pipelines, they should have approximately same memory usages in each pipeline.

We employ the measurement systems to perform three representative measurement tasks, which are:

[2]The P4 code of Confluence is available at `https://github.com/wyw0530/Confluence`

TABLE 1. Parameters of different measurement systems on a switch composed of $K = 8$ pipelines

	Ingress pipeline	Egress pipeline	Memory usage
SinglePipe	N/A	Entries in Elastic Sketch heavy part: 2240 Buckets in Elastic Sketch light part: 2×22664	90,688 Bytes
ParallelSketch	N/A	Entries in Elastic Sketch heavy part: 280 Buckets in Elastic Sketch light part: 2×2833	11,336 Bytes per pipeline Total: 90,688 Bytes
PipeCache/PipeCache*	Number of sliced caches: 4 Summaries in a sliced cache: $K \times 8$	Number of sliced Elastic Sketch: 4 Entries in sliced Elastic Sketch heavy part: 51 Buckets in sliced Elastic Sketch light part: 2×2^9	3,104 Bytes per ingress pipe 8,227 Bytes per egress pipe Total: 90,648 Bytes
Confluence	Entries in a pipeline table: K Buckets in a delta sketch: 2×2^{10}	Entries in Elastic Sketch heavy part: 204 Buckets in Elastic Sketch light part: 2×2^{11}	2,950 Bytes per ingress pipe 8,227 Bytes per egress pipe Total: 89,416 Bytes



(a) Flow size estimation (b) Flow size distribution estimation (c) Heavy hitter detection

FIGURE 8. Measurement accuracies for (a) flow size estimation, (b) flow size distribution estimation, and (b) heavy hitter detection tasks accomplished by SinglePipe, ParallelSketch, and Confluence when measuring various numbers of network flows.

- **Flow size estimation**: In this task, we estimate the size of each network flow, and assess the estimating accuracy with *Averaged Relative Error* (*ARE*), which is defined as

$$ARE = \frac{1}{|\mathbb{F}|} \sum_{f \in \mathbb{F}} \frac{|n_f - n'_f|}{n_f} \tag{8}$$

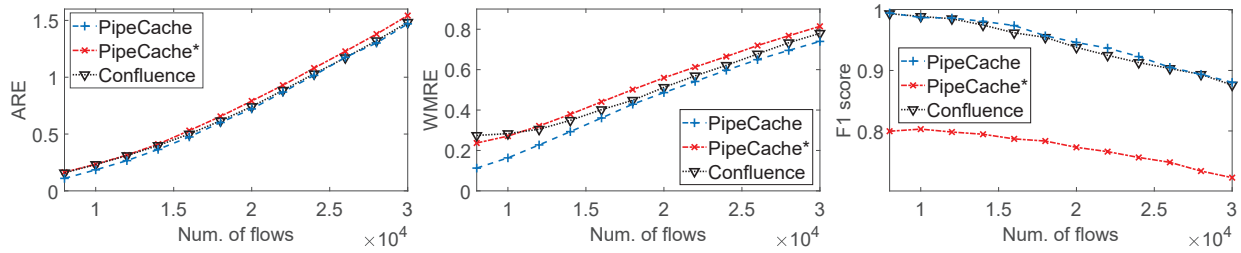
  where n_f and n'_f are the ground-truth and estimated sizes of a flow f, and $\mathbb{F}$ is the set of the flows being monitored.
- **Flow size distribution estimation**: We use the *Weighted Mean Relative Error* (*WMRE*) to measure the difference between an estimated flow size distribution and the ground truth as

$$WMRE = \frac{\sum_i |m_i - m'_i|}{\sum_i (\frac{|m_i + m'_i|}{2})} \tag{9}$$

  where m_i and m'_i are the ground-truth and estimated numbers of the flows of size i.
- **Heavy hitter detection**: This task seeks to identify the heavy hitter flows, which are defined as the top 10% largest flows in $\mathbb{F}$. We use the *F1 score* to assess the detecting accuracy.

Note that the three measurement tasks have different significance. The task of flow size estimation only considers the overall averaged error, but does not evaluate how the estimation of each individual flow deviates from the ground truth. The task of flow size distribution estimation requires that the distribution of the estimations should be close to the ground truth, which is more difficult. The heavy hitter detection task is the most difficult and important one, as it demands to identify all the heavy hitter flows individually, and the task provides valuable information for network traffic engineering [42], QoS optimization [43], anomaly and intrusion detections [44]. In addition to measurement accuracy, we also examine the ratios of the packets that cause cache overflows in PipeCache/PipeCache* and counter overflows in Confluence respectively.

Packet bursts are largely caused by TSO [19], which is a feature supported by new NICs after 2014 [45]. For early data center traces (e.g., [46, 47]) that were collected on old NICs, we find that they contain much fewer and much smaller packet bursts comparing with the ones observed in the recent studies [19, 20]. As data center traffic traces collected from TSO-enabled NICs are currently not publicly available, in this work, we use the MAWI packet trace [48] captured from the WIDE backbone network to drive the simulation. Usually, packets arrive to the simulated switch following their original sequences in the trace. However, traffics on ISP backbone links are highly statistically multiplexed and less bursty than the ones in data centers. For this reason, in our experiments, for all the network flows whose packet rates exceed 300 packets per second, we assume that they send packets in bursts. More specifically, for a network flow that has its packet rate exceeding 300, we group its packets into bursts, where each burst has a size following the distribution in [19] with a mean burst size of 30 packets. The traffic trace is transformed to contain bursts as the following: Each burst starts when its first packet appears in the original trace, but in the transformed trace, all the subsequent packets in the burst follow the first packet and arrive



(a) Flow size estimation (b) Flow size distribution estimation (c) Heavy hitter detection

FIGURE 9. Measurement accuracies for (a) flow size estimation, (b) flow size distribution estimation, and (c) heavy hitter detection tasks accomplished by PipCache, PipCache*, and Confluence when measuring various numbers of network flows.

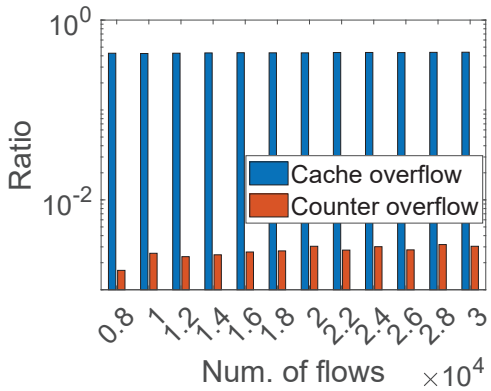
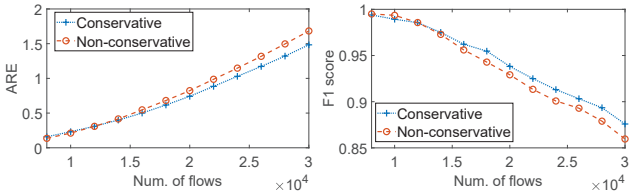


FIGURE 10. Comparison of ratios of packets encountering cache overflows in PipeCache/PipeCache* and counter overflows in Confluence when measuring various numbers of network flows. Note that Y-axis applies log scale.



(a) Flow size estimation (b) Heavy hitter detection

FIGURE 11. Measurement accuracies for (a) flow size estimation and (b) heavy hitter detection tasks accomplished by Confluence with and without conservative update algorithm.

to the switch back-to-back, while for the packet that does not belong to the burst but immediately follows the first packet in the original trace, it arrives after the burst in the transformed trace.

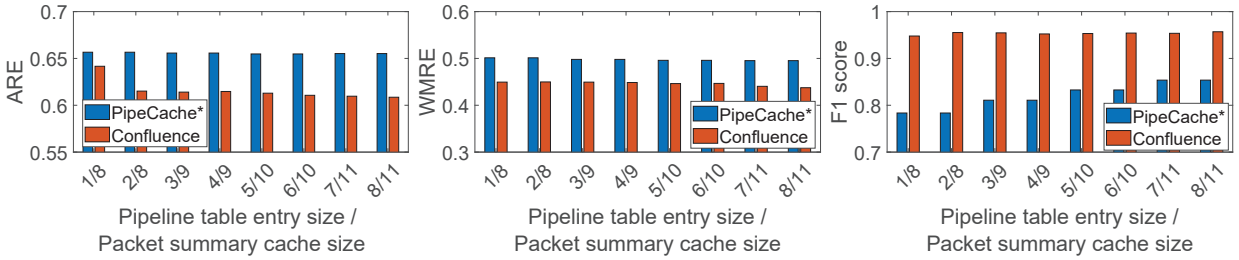
4.2. Measurement Accuracy

We evaluate the measurement systems by monitoring 8,000-30,000 network flows. If not otherwise specified, for a simulated multi-pipeline switch, we configure it to have $K = 8$ parallel pipelines. For a network flow, the switch routes its packets to each egress pipeline with an equal chance of $\frac{1}{K}$. We stick to the fair memory usage principles as described in Sec. 4.1. In particular, for Confluence, the size of the pipeline table entry is set as $c = 2$, and the delta sketch contains 2×2^{10} 5-bit buckets; in addition, each ingress pipeline hosts two instances of pipeline tables and delta sketches to avoid violating the constraints of the hardware switch. For PipeCache/PipeCache*, each sliced cache can store up to 8 packet summaries, so that its ingress memory usage is no smaller than Confluence. Table 1 lists parameters of the different systems under the evaluation[3].

We first compare Confluence with the benchmarks of SinglePipe and ParallelSketch, and present the results in Fig. 8. From the figure one can see that Confluence significantly outperforms ParallelSketch in all the three tasks, because of the reason as we have analyzed in Sec. 2.2.1. SinglePipe is more accurate than Confluence, as it has a lower hash collision rate with one single large sketch than the 8 small sketch instances in Confluence, and it avoids the errors in moving flow statistics from the ingress pipelines to the egress ones. Note that SinglePipe is for a switch with only one pipeline, which can not sustain a packet rate as high as a multi-pipeline switch.

In Fig. 9, we compare Confluence with PipeCache and PipeCache*, as the three solutions place measurement data structures in both ingress and egress pipelines of a multi-pipeline switch. Fig. 9(a) shows that in the flow size estimation task that is least significant, Confluence is slightly less accurate than PipeCache and slightly more accurate than PipeCache*, as the metric of ARE only considers the overall averaged errors. In the task of the flow size distribution estimation with the metric of WMRE by averaging errors weighted with flow sizes, when measuring over 14,000 flows, Confluence starts to outperform PipeCache* and approach to PipeCache. In the heavy hitter detection task that is most difficult and important, Confluence substantially outperforms PipeCahce* and has the F1 scores only slightly lower than PipeCache. This is be-

[3]Both packet summary and flow ID are 5-tuple, whose size is 97 bits. An entry of the Elastic Sketch heavy part has 162 bits, which contain two 32-bit counters, 1-bit flag, and 97 bits for the 5-tuple flow ID. A pipeline table entry has 195 bits, which stores $c = 2$ flow IDs and one bit to indicate the head position of the queue. A bucket in the Elastic Sketch has 8 bits.



(a) Flow size estimation (b) Flow size distribution estimation (c) Heavy hitter detection

FIGURE 12. Measurement accuracies for (a) flow size estimation, (b) flow size distribution estimation, and (c) heavy hitter detection tasks accomplished by PipeCache* and Confluence when increasing the size of the sliced cache for PipeCache* and the pipeline table entry size for Confluence.

cause when the heavy hitter flows have their packets arrived in bursts, PipeCache counts all the packets, at a cost of cloning and recirculating a large number of packets; Confluence manages to count most of the packets, as the 5-bit counters in the delta sketch can count up to $2^5 - 1 = 31$ packets and is unlikely to overflow; while PipeCache* miss-counts many packets due to overflows of the packet summary caches. From the experiment result, we can see that Confluence has a performance close to PipeCache without the cost of packet clone and recirculation, and it outperforms PipeCache*, especially in the heavy hitter detection task that is most valuable for network management and optimization.

The only difference between PipeCache and PipeCache* is that PipeCache clones and recirculates a packet on cache overflow, while PipeCache* does not allow packet recirculation and miss-counts the packet. Fig. 10 presents the ratio of the packets that encounter cache overflows in PipeCache/PipeCache*, and for comparison, we also present the ratio of the packets that encounter counter overflows in Confluence. From the figure, we can see that over 40% packets encounter cache overflows, which means that in PipeCache, these packets double their processing capacity and queueing space consumptions within the switch, while in PipeCache*, these packets are miss-counted. On the other hand, only fewer than 0.4% packets in Confluence are miss-counted due to counter overflows.

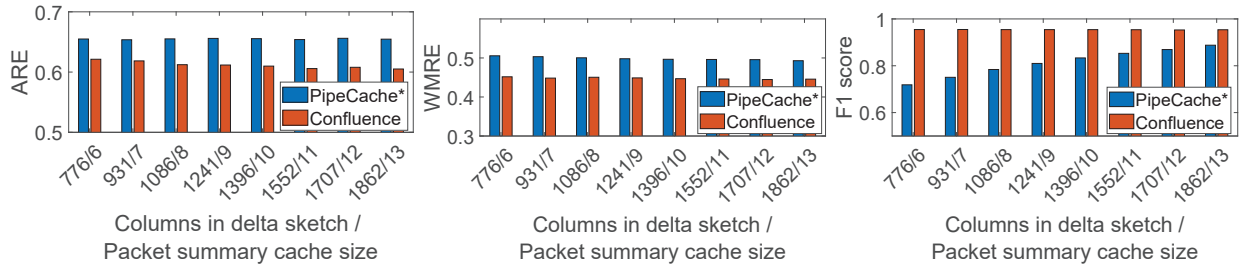
In Confluence, we propose Algorithm 1 to conservatively update the delta sketch buckets for resolving circular dependency. In the following experiment, we assume that a hypothetical Confluence system can violate the circular dependence constraint and directly deduct $\min\{\boldsymbol{\Delta}_i[r][h_r(g)]\}$ from each bucket, and compare it with the Confluence system applying Algorithm 1. Fig. 11 presents the AREs and F1 scores of the two systems in the tasks of flow size estimation and heavy hitter detection, from which we can see that by applying Algorithm 1, the system has slightly higher measurement accuracies. Our observation can be explained with Theorem 3.2, which states that Algorithm 1 only overly decreases a bucket whose value is overly estimated, thus can partially offset the overestimation errors.

4.3. Impact of Memory Usage

In this subsection, we study the impact of memory usages. Since in this work, we aim to avoid the resource-consuming packet clone and recirculation, in this and subsequent simulation experiments, we compare Confluence with PipeCache*, as both systems do not allow packet recirculation. As Confluence places two data structures in an ingress pipeline: the pipeline table and delta sketch, we increase the size of the two instances of each data structure separately. Intuitively, when the pipeline table has its entry size increased, it can "memorize" a network flow for a longer time, as the flow ID of a received packet is less likely to be "flushed out" by subsequently arrived packets. Likewise, when the delta sketch contains more bucket columns, hash collisions could be less likely to happen. Note that when increasing sizes of the Confluence data structures, we also enlarge the sliced caches in PipeCache* proportionally, so that PipeCache* consumes no less memory in each ingress pipeline than Confluence.

In the first experiment, we increase the size of the pipeline table entry in Confluence from $c = 1$ to 8 flow IDs, and proportionally enlarge the sliced cache in PipeCache* from 8 to 11 packet summaries. Fig. 12 presents the measurement accuracies achieved by the two systems. We can see that for Confluence, when increasing the pipeline table entry size from 1 to 2, the flow size estimation accuracy is substantially improved, but further increasing the entry size does not lead to significant improvement. For PipeCache*, increasing the sliced cache size obviously increase the F1 score, as for heavy hitter flows, a larger cache will miss fewer packets in bursts; but in the flow size estimation and flow size distribution estimation tasks, whose metrics are decided by all the network flows, increasing the cache size (from 8 to 11) does not make much difference.

Fig. 13 presents the measurement accuracies when we increase the size of the delta sketch in Confluence from 776 bucket columns to 1,862 columns, and increase the size of the sliced cache in PipeCache* from 6 to 13



(a) Flow size estimation (b) Flow size distribution estimation (c) Heavy hitter detection

FIGURE 13. Measurement accuracies for (a) flow size estimation, (b) flow size distribution estimation, and (c) heavy hitter detection tasks accomplished by PipeCache* and Confluence when increasing the size of the sliced cache for PipeCache* and the delta sketch columns for Confluence.

packet summaries proportionally. One can see that for PipeCache*, when enlarging the sliced cache, the F1 score in heavy hitter detection is improved considerably, for the same reason as above explained. To our surprise, increasing the size of the delta sketch does not lead to significant improvement on measurement accuracies for Confluence. We explain the observation with the fact that unlike conventional sketch-based systems, measurement data in a delta sketch instance is constantly removed and carried to the main sketches, so in most of the time, the sketch is close to empty with few hash collisions, despite that its size is small.

The experiment results in Fig. 12 and Fig. 13 suggest that with various-sized pipeline table and delta sketch instances, Confluence delivers higher measurement accuracies and substantially outperform PipeCache*, and its advantage over PipeCache* is more pronounced under small ingress memory usages. Note that such a property is highly desired, as fast-speed SRAM is a scarce resource in hardware programmable switches.

4.4. Impact of Number of Pipelines

In the next experiment, we examine the impact of number of the pipelines within a switch on PipeCache* and Confluence. Note that for PipeCache*, each ingress pipeline maintains K caches, one for each egress pipeline, and each cache is divided into 4 slices. For Confluence, each ingress pipeline maintains two pipeline table instances each containing K entries, however, the sizes of the two delta sketch instances are irrelevant to the number of the pipelines. In other words, Confluence is more scalable than PipeCache* by design.

We perform the three measurement tasks under the cases when the switches contain $K = 4$, 8, 16, 32, and 64 parallel pipelines, and feed $7000 \times K$ network flows to drive the experiments. To enforce the memory usage fairness principle, we fix the size of the pipeline table entry as $c = 2$ for Confluence, and with the same memory usages, we decrease the size of the sliced packet summary cache in PipeCache* from 15 to 2 when the number of the pipelines is increased from 4 to 64[4]. Fig. 14 presents the experiment results, from which we can see that for both PipeCache* and Confluence, in the tasks of flow size estimation and flow size distribution estimation, measurement accuracies are slightly reduced when a switch contains more pipelines. However, for the heavy hitter detection task, the F1 score remains relatively stable for Confluence, but is substantially decreased for PipeCache*, as PipeCache* has more cache overflows due to its relatively poor scalability. Our observation indicates that when the switch contains more pipelines, Confluence is more scalable than PipeCache* by keeping its measurement accuracies stable and outperforming PipeCache*, especially in the critical task of detecting heavy hitter flows.

4.5. Impact of Imbalanced Workload

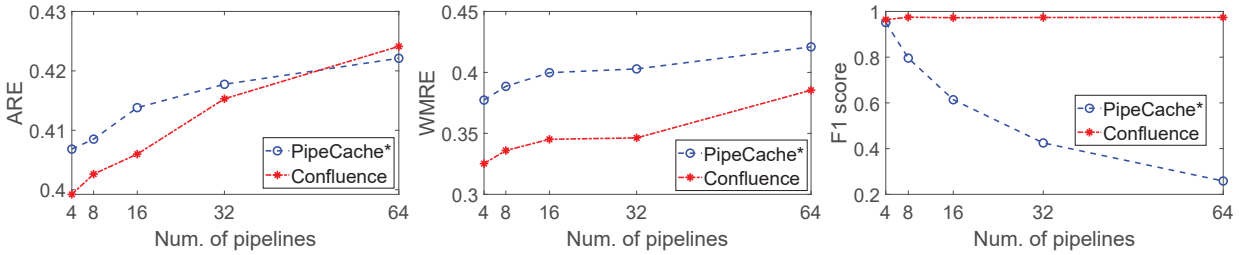
Our previous experiments assume balanced workloads among the pipelines, that is, a packet is routed to each pipeline with an equal chance of $\frac{1}{K}$. In this section, we examine the case that the workloads are imbalanced, in particular, we consider that a packet is routed to the k^{th} pipeline at a probability of p_k following a Zipf distribution as

$$p_k = \frac{\frac{1}{k^s}}{\sum_{i=1}^{K} \frac{1}{i^s}} \tag{10}$$

where $s \geq 0$ is the distribution's skewness factor.

We compare PipeCache* and Confluence under imbalanced workloads with $s = 0$, 1, 2, and 3 respectively. Note that when $s = 0$, the workload is indeed balanced. Fig. 15 presents the experiment results in monitoring 18,000 and 24,000 network flows, from which one can see under the imbalanced workload, Confluence has its measurement accuracies less reduced

[4]When there are $K = 4$ pipelines, for Confluence, a pipeline table instance consumes $K \times 195 = 780$ bits, and a delta sketch instance consumes $2 \times 2^{10} \times 5 = 10,240$ bits, so the ingress pipeline memory usage is $2 \times (780 + 10,240) = 22,040$ bits. For PipeCache*, as there are $4 \times K = 16$ sliced caches in each ingress pipeline, a cache should have a size of $\frac{22,040}{(16 \times 97)} = 14.2$, so we set the cache size as 15. With $K = 64$ pipelines, for Confluence, a pipeline table instance consumes $K \times 195 = 12,480$ bits, so the memory usage of the ingress pipeline is $2 \times (12,480 + 10,240) = 45,440$ bits. For PipeCache*, as there are $4 \times K = 256$ sliced caches in each ingress pipeline, a cache should have a size of $\frac{45,440}{(256 \times 97)} = 1.8$, so we set the cache size as 2.



(a) Flow size estimation (b) Flow size distribution estimation (c) Heavy hitter detection

FIGURE 14. Measurement accuracies for (a) flow size estimation, (b) flow size distribution estimation, and (c) heavy hitter detection tasks accomplished by PipeCache* and Confluence under various number of pipelines within a switch.

than PipeCache*. For example, in the heavy hitter detection task, when increasing s from 0 to 3, the F1 scores of PipeCache* are reduced 41.4% and 41.8% when measuring 16,000 and 24,000 network flows respectively, but the reductions on Confluence's F1 scores are 19.1% and 19.3% under the same conditions. We have experimented by monitoring other numbers of network flows and have similar observations. The poor performance of PipeCache* is easy to understand, as when the workloads are imbalanced, for some pipeline, there would be much more packets injecting their summaries to the cache than the packets carrying the cached summaries to egress pipelines, making the caches easier to over-flow.

4.6. Impact on Switch Performance

Confluence requires each packet to make $2 + 2 \times d_1$ register accesses when passing an ingress pipeline and access 1 or $1+d_2$ registers in its egress pipeline pass. In this subsection, we evaluate the impact of the register accesses on a hardware switch's packet forwarding performance. To this end, we implement Confluence on an Edgecore Wedge 100BF Tofino-based programmable switch.

We set up a simple testbed composed of eight servers inter-connected by the Tofino switch. All the servers are equipped with the Intel i7-8700 CPU and Intel X710-DA2 10GbE NIC. We set up 500 UDP flows from one server to another via the switch, and measure the switch's data forwarding rate with and without Confluence. Note that without Confluence, a packet does not make any register accesses in its pipeline pass.

We find that in both cases, the Tofino switch can achieve its line rate close to 10 Gbit/s, despite that Confluence demands more register accesses when processing a packet. We also examine the time required for the switch to process a packet. For the switch with Confluence, it takes 150 cycles and averagely 258.3 ns to process a packet, and for the switch that simply forwards packets, the per-packet processing time is 58 cycles and averagely 187.0 ns. The result suggests that Confluence does not impact a switch's forwarding performance, and is practical to be deployed in real-world data center networks.

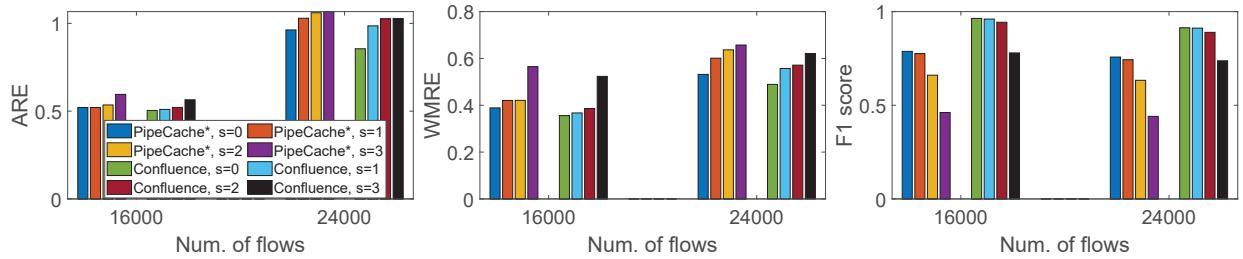
5. RELATED WORK

5.1. Programmable Data Plane

As network control plane requires more precise control on data plane resources and behaviors, the architecture of hardware programmable switch evolves rapidly in the last decade. Bosshart et al. [21] propose the RMT pipelined architecture for providing reconfigurable match-action tables in switching chips. Based on RMT, Sivaraman et al. [22] present the Banzai programmable packet processing pipeline architecture that supports stateful data plane algorithms. To overcome the limitation of no state sharing across pipelines, new generation of programmable switch architectures are proposed. Shrivastav [24] presents MP5, a switch architecture, compiler, and runtime for multi-pipeline programmable switches that is functionally equivalent to a single-pipeline switch. Chen et al. [25] propose OptimusPrime to enable an architecture that transforms between pipeline stages and multi-core RTC processors. Unfortunately, the new generation architectures have yet been supported by commodity hardware switches.

5.2. Sketch-based Network Traffic Monitoring

Sketch-based method is a promising direction for providing full-coverage and fine-grained network measurement in data center networks. To overcome the low memory utilizations of the classical sketches of CM [26], CU [27], and Count [28], a number of advanced sketches are proposed in recent years. For per-flow measurement, Yang et al. [6] present Elastic Sketch, which is composed of a heavy part hash table and a light part CM sketch to perform per-flow traffic monitoring; Zhang et al. [7] propose CocoSketch for supporting partial key query; Zhao et al. [8] develop a novel sketch namely SuMax to support various per-flow measurement tasks; Gu et al. [3] propose Distributed Sketch to perform sketch-based per-flow measurement with a network-wide cooperative method. For detecting persistent items in network data streams, Zhang et al. [49] present On-Off sketch for estimating number of epochs that an item persistently appears; Li et al. [10] propose P-Sketch to estimate item persistence by exploiting con-



(a) Flow size estimation (b) Flow size distribution estimation (c) Heavy hitter detection

FIGURE 15. Measurement accuracies for (a) flow size estimation, (b) flow size distribution estimation, and (c) heavy hitter detection tasks accomplished by PipeCache* and Confluence under imbalanced workloads with $s = 0$, 1, and 2.

tinuity of item appearance. For detecting spreaders in network traffic, Tang et al. [50] develop SpreaderSketch to estimate spreads of network flows with multiresolution bitmap. However, most of sketch-based measurement applications are developed for one single logical pipeline, and when being naively deployed on a multi-pipeline switch, they face correctness and performance issues [14, 15].

5.3. Traffic Monitoring on Multi-Pipeline Switches

Gebara et al. [14] point out that PISA-based programmable switch provides no state-sharing across pipelines, which is an obstacle for in-network applications. Khooi et al. [15] analyze the impact of multiple pipelines on stateful network applications. Chiesa et al. [17, 18] describe and address the problem of network measurement on multi-pipeline switches caused by multi-path routing, and present a solution named PipeCache. Rodrigues et al. [51] propose a recirculation-based method similar to PipeCache and a controller-assisted method to detect heavy hitter flows on multi-pipeline switches. Huang et al. [52] present MpScope, a multi-pipeline monitoring framework that employs controller to aggregate measurement results from multiple pipelines and tunes the monitoring module in the different pipelines dynamically. Comparing with the controller-assisted solutions [51, 52], Confluence accurately monitors traffic within a multi-pipeline data plane without the assistance from controller; and unlike the recirculation-based approaches [17, 18], Confluence does not employ the expensive packet clone and recirculation, and preserves accuracy under various circumstances.

6. CONCLUSION

Similar to multi-core CPUs, state-of-the-art programmable switches employ multiple pipelines to sustain higher packet rates, but on the other hand, the existing domain-specific data plane programming model (e.g., P4) still assumes the target switch as one single logical pipeline. Such a discrepancy causes existing sketch-based network measurement applications, which are programmed for single-pipeline switches, have poor measurement accuracy when being deployed naively on multi-pipeline switches under multi-path routing scenarios.

In this paper, we presented Confluence, a sketch-based network measurement system for multi-pipeline switches. By introducing novel data structures and methodologies, Confluence is capable to collect and converge measurement data of network flows, which have their packets spread over multiple pipelines, in an efficient and inexpensive way. Theoretical analysis and experimental evaluation show that Confluence substantially outperforms the existing solutions by delivering high measurement accuracy, and the system is practical for real-world deployment.

ACKNOWLEDGEMENTS

This work was supported in part by the National Natural Science Foundation of China [grant numbers 61672486, 62072425].

DATA AVAILABILITY

The data underlying this article are available at https://mawi.wide.ad.jp/mawi/.

REFERENCES

[1] RFC 3954 (2004) *Cisco Systems NetFlow Services Export Version 9.* Internet Engineering Task Force (IETF). Wilmington, DE, USA.

[2] Phaal, P. and Lavine, M. (2004) *sFlow Version 5.* InMon Corp. San Francisco, CA, USA. `https://sflow.org/sflow_version_5.txt`.

[3] Gu, L., Tian, Y., Chen, W., Wei, Z., Wang, C., and Wang, X. (2024) Per-flow network measurement with distributed sketch. *IEEE/ACM Trans. Netw.*, **32**, 411–426.

[4] Li, Y., Miao, R., Kim, C., and Yu, M. (2016) FlowRadar: A better NetFlow for data centers. *Proc. of NSDI'16*, Santa Clara, CA, USA, 16-18 March, pp. 311–324. USENIX Association, Berkeley, CA, USA.

[5] Basat, R. B., Chen, X., Einziger, G., and Rottenstreich, O. (2020) Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Trans. Netw.*, **28**, 1172–1185.

[6] Yang, T., Jiang, J., Liu, P., Huang, Q., Gong, J., Zhou, Y., Miao, R., Li, X., and Uhlig, S. (2018) Elastic Sketch: Adaptive and fast network-wide measurements. *Proc. of SIGCOMM'18*, Budapest, Hungary, 20-25 August, pp. 561–575. ACM, New York, NY, USA.

[7] Zhang, Y., Liu, Z., Wang, R., Yang, T., Li, J., Miao, R., Liu, P., Zhang, R., and Jiang, J. (2021) CocoSketch: High-performance sketch-based measurement over arbitrary partial key query. *Proc. of SIGCOMM'21*, Virtual Event, 23-27 August, pp. 207–222. ACM, New York, NY, USA.

[8] Zhao, Y., et al. (2021) LightGuardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. *Proc. of NSDI'21*, Virtual Event, 12-14 April, pp. 991–1010. USENIX Association, Berkeley, CA, USA.

[9] Wei, Z., Tian, Y., Chen, W., Gu, L., and Zhang, X. (2023) DUNE: Improving accuracy for sketch-int network measurement systems. *Proc. of IEEE INFOCOM'23*, New York, USA, 17-20 May, pp. 1–10. IEEE, Piscataway, NJ, USA.

[10] Li, W. and Patras, P. (2024) P-Sketch: A fast and accurate sketch for persistent item lookup. *IEEE/ACM Trans. Netw.*, **32**, 987–1002.

[11] *P4 Open Source Programming Language.* `https://p4.org/.AccessedonAug.20,2024`.

[12] *Intel Tofino 2.* `https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino-2/products.html`. Accessed on Aug. 20, 2024.

[13] *Broadcom. Tomahawk3 - BCM56980 12.8 Tb/s Multilayer Switch.* `https://docs.broadcom.com/doc/56980-DS`. Accessed on Aug. 20, 2024.

[14] Gebara, N., Lerner, A., Yang, M., Yu, M., Costa, P., and Ghobadi, M. (2020) Challenging the stateless quo of programmable switches. *Proc. of HotNets'20*, Virtual Event, 4-6 November, pp. 153–159. ACM, New York, NY, USA.

[15] Khooi, X. Z., Csikor, L., Li, J., and Divakaran, D. M. (2021) In-network applications: Beyond single switch pipelines. *Proc. of IEEE International Conference on Network Softwarization (NetSoft'21)*, Tokyo, Japan, 28 June-2 July, pp. 1–8. IEEE, Piscataway, NJ, USA.

[16] RFC 2991 (2000) *Multipath Issues in Unicast and Multicast Next-Hop Selection.* Internet Engineering Task Force (IETF).

[17] Chiesa, M. and Verdi, F. L. (2023) Network monitoring on multi-pipe switches. *Proc. of SIGMETRICS'23*, Orlando, Florida, USA, 19-23 June, pp. 49–50. ACM, New York, NY, USA.

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

[19] Kapoor, R., Snoeren, A. C., Voelker, G. M., and Porter, G. (2013) Bullet trains: A study of NIC burst behavior at microsecond timescales. *Proc. of CoNEXT'13*, Santa Barbara, CA, USA, 9-12 December, pp. 133–138. ACM, New York, NY, USA.

[20] Sharafzadeh, E., Abdous, S., and Ghorbani, S. (2023) Understanding the impact of host networking elements on traffic bursts. *Proc. of NSDI'23*, Boston, MA, USA, 17-19 April, pp. 237–254. USENIX Association, Berkeley, CA, USA.

[21] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. (2013) Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *Proc. of SIGCOMM'13*, Hong Kong, China, 12-16 August, pp. 99–110. ACM, New York, NY, USA.

[22] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. (2016) Packet transactions: High-level programming for line-rate switches. *Proc. of SIGCOMM'16*, Florianopolis , Brazil, 22-26 August, pp. 15–28. ACM, New York, NY, USA.

[23] P4.org (2021) *$P4_{16}$ Portable Switch Architecture (PSA).* The P4.org Architecture Working Group. `https://p4.org/p4-spec/docs/PSA.pdf`.

[24] Shrivastav, V. (2022) Stateful multi-pipelined programmable switches. *Proc. of SIGCOMM'22*, Amsterdam, Netherlands, 22-26 August, pp. 663–676. ACM, New York, NY, USA.

[25] Chen, Z., Feng, Y., Liu, S., Song, H., Zhou, H., Yun, T., Xu, W., Pan, T., and Liu, B. (2024) Optimusprime: Unleash dataplane programmability through a transformable architecture. *Proc. of SIGCOMM'24*, Sydney, Australia, 4-8 August, pp. 663–676. ACM, New York, NY, USA.

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

[27] Estan, C. and Varghese, G. (2002) New directions in traffic measurement and accounting. *ACM SIGMCOMM CCR*, **32**, 323–336.

[28] Charikar, M., Chen, K., and Farach-Colton, M. (2002) Finding frequent items in data streams. *Proc. of International Colloquium on Automata, Languages, and Programming (ICALP'02)*, Malaga, Spain, 8-13 July, pp. 693–703. Springer-Verlag Berlin, Heidelberg, Germany.

[29] Yang, T., Gao, S., Sun, Z., Wang, Y., Shen, Y., and Li, X. (2019) Diamond sketch: Accurate per-flow measurement for big streaming data. *IEEE Trans. Parallel Distrib. Syst.*, **30**, 2650–2662.

[30] Song, C. H., Kannan, P. G., Low, B. K. H., and Chan, M. C. (2020) FCM-Sketch: Generic network measurements with data plane support. *Proc. of CoNEXT'20*, Barcelona, Spain, 1-4 December, pp. 78–92. ACM, New York, NY, USA.

[31] Yang, K., Li, Y., Liu, Z., Yang, T., Zhou, Y., He, J., Xue, J., Zhao, T., Jia, Z., and Yang, Y. (2021) SketchINT: Empowering INT with TowerSketch for per-flow per-switch measurement. *Proc. of ICNP'21*, Virtual Event, 1-5 November, pp. 1–12. IEEE, Piscataway, NJ, USA.

[32] Jain, S., et al. (2013) B4: Experience with a globally-deployed software defined WAN. *Proc. of SIGCOMM'13*, Hong Kong, China, 12-16 August, pp. 3–14. ACM, New York, NY, USA.

[33] Hong, C.-Y., et al. (2018) B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. *Proc. of SIGCOMM'18*, Budapest, Hungary, 20-25 August, pp. 74–87. ACM, New York, NY, USA.

[34] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. (2017) Heavy-hitter detection entirely in the data plane. *Proc. of SOSR'17*, Santa Clara, CA, USA, 3-4 April, pp. 164–176. ACM, New York, NY, USA.

[35] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D., Patel, P., and Sengupta, S. (2009) VL2: A scalable and flexible data center network. *Proc. of SIGCOMM'09*, Barcelona, Spain, 16-21 August, pp. 51–62. ACM, New York, NY, USA.

[36] He, K., Rozner, E., Agarwal, K., Felter, W., Carter, J., and Akella, A. (2017) Presto: Edge-based load balancing for fast datacenter networks. *Proc. of SIGCOMM'15*, London, UK, 17-21 August, pp. 465–478. ACM, New York, NY, USA.

[37] Zhang, H., Zhang, J., Bai, W., Chen, K., and Chowdhury, M. (2017) Resilient datacenter load balancing in the wild. *Proc. of SIGCOMM'17*, Los Angeles, CA, USA, 21-25 August, pp. 253–266. ACM, New York, NY, USA.

[38] Mellette, W. M., McGuinness, R., Roy, A., Forencich, A., Papen, G., Snoeren, A. C., and Porter, G. (2017) RotorNet: A scalable, low-complexity, optical datacenter network. *Proc. of SIGCOMM'17*, Los Angeles, CA, USA, 21-25 August, pp. 267–280. ACM, New York, NY, USA.

[39] Mazziane, Y. B., Alouf, S., and Neglia, G. (2022) Analyzing count min sketch with conservative updates. *Computer Networks*, **217**, 1–10.

[40] Kim, S., Jung, C., Jang, R., Mohaisen, D., and Nyang, D. (2023) A robust counting sketch for data plane intrusion detection. *Proc. of NDSS*, San Diego, CA, USA, 27 February - 3 March, pp. 1–17. The Internet Society, Reston, VA, USA.

[41] *bmv2, the behavioral model for P4.* `https://github.com/p4lang/behavioral-model`. Accessed on Aug. 20, 2024.

[42] Benson, T., Anand, A., Akella, A., and Zhang, M. (2011) MicroTE: Fine grained traffic engineering for data centers. *Proc. of CoNEXT'11*, Tokyo, Japan, 6-9 December, pp. 1–12. ACM, New York, NY, USA.

[43] Kabbani, A., Alizadeh, M., Yasuda, M., Pan, R., and Prabhakar., B. (2010) AF-QCN: Approximate fairness with quantized congestion notification for multi-tenanted data centers. *Proc. of IEEE HOTI'10*, Mountain View, CA, USA, 18-20 August, pp. 58–65. IEEE, Piscataway, NJ, USA.

[44] Garcia-Teodoro, P., Daz-Verdejo, J. E., Maci-Fernndez, G., and Vzquez, E. (2009) Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, **28**, 18–28.

[45] *TCP offload engine.* `https://en.wikipedia.org/wiki/TCP_offload_engine`. Accessed on Feb. 25, 2025.

[46] Benson, T., Akella, A., and Maltz, D. A. (2010) Network traffic characteristics of data centers in the wild. *Proc. of IMC'10*, Melbourne, Australia, 1-3 November, pp. 267–280. ACM, New York, NY, USA.

[47] *Data Set for IMC 2010 Data Center Measurement.* `https://pages.cs.wisc.edu/~tbenson/IMC10_Data.html`. Accessed on Feb. 25, 2024.

[48] *MAWI Working Group Traffic Archive.* `https://mawi.wide.ad.jp/mawi/`. Accessed on Aug. 15, 2024.

[49] Zhang, Y., Li, J., Lei, Y., Yang, T., Li, Z., Zhang, G., and Cui, B. (2020) On-Off sketch: a fast and accurate sketch on persistence. *Proceedings of VLDB Endowment*, **14**, 128–140.

[50] Tang, L., Xiao, Y., Huang, Q., and Lee, P. P. C. (2023) A high-performance invertible sketch for network-wide superspreader detection. *IEEE/ACM Trans. Networking*, **31**, 724–737.

[51] Rodrigues, T. H. S. and Verdi, F. L. (2024) Detecting heavy hitters in network-wide programmable multi-pipe devices. *Proc. of IEEE/IFIP Network Operations and Management Symposium (NOMS 2024) - Poster*, Seoul, Korea, 6-10 May, pp. 1–5. IEEE, Piscataway, NJ, USA.

[52] Huang, C., et al. (2024) MpScope: Enabling multi-pipeline monitoring inside a switch. *Computer Networks*, **254**, 1–12.