Network Monitoring on Multi-Pipe Switches

MARCO CHIESA*, KTH Royal Institute of Technology, Sweden
FÁBIO L. VERDI†, Federal University of São Carlos, Brazil

Programmable switches have been widely used to design network monitoring solutions that operate in the fast data-plane level, *e.g.*, detecting heavy hitters, super-spreaders, computing flow size distributions and their entropy. Many existing works on networking monitoring assume switches deploy a single memory that is accessible by each processed packet. However, high-speed ASIC switches increasingly deploy *multiple* independent pipes, each equipped with its own independent memory that *cannot* be accessed by other pipes.

In this work, we initiate the study of deploying existing heavy-hitter data-plane monitoring solutions on multi-pipe switches where packets of a "flow" may spread over multiple pipes, *i.e.*, stored into distinct memories. We first quantify the accuracy degradation due to splitting a monitoring data structure across multiple pipes (*e.g.*, up to 3000x worse flow-size estimation average error). We then present PipeCache, a system that adapts *existing* data-plane mechanisms to multi-pipe switches by carefully storing all the monitoring information of each traffic class into exactly one specific pipe (as opposed to replicate the information on multiple pipes). PipeCache relies on the idea of briefly storing monitoring information into a per-pipe cache and then piggybacking this information onto existing data packets to the correct pipe *entirely* at data-plane speed. We implement PipeCache on ASIC switches and we evaluate it using a real-world trace. We show that existing data-plane mechanisms achieves accuracy levels and memory requirements similar to single-pipe deployments when augmented with PipeCache (*i.e.*, up to 16x lower memory requirements).

CCS Concepts: • **Networks** → **Network measurement**; **Network monitoring**; *Bridges and switches.*

Additional Key Words and Phrases: multi-pipe switches; ASIC switches; network monitoring; P4

ACM Reference Format:
Marco Chiesa and Fábio L. Verdi. 2023. Network Monitoring on Multi-Pipe Switches. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 1, Article 8 (March 2023), 31 pages. https://doi.org/10.1145/3579321

1 INTRODUCTION

Network monitoring is an indispensable component of today's networks. Network operators rely on monitoring systems to collect key statistics about the underlying traffic, optimize network performance, detect misconfigurations, attacks, and beyond. Three key requirements of today's monitoring solutions are: *accuracy*, *i.e.*, the ability to correctly identify network events, *low memory overheads*, *i.e.*, minimizing the amount of resources used on a switch, and *reactiveness*, *i.e.*, the time required to detect an event. We discuss the rich body of literature that has long explored the trade-off among these requirements [11, 28, 32, 41, 45, 53, 56, 63, 71, 73–75].

Traditional monitoring approaches rely on *sampling* packets from the data-plane of a switch and updating per-flow counters in the control-plane (*e.g.*, NetFlow [32]). These approaches trade lower memory overheads in the data-plane (as control-plane memory is abundant and cheap) at the price of *both* lower accuracy (as packets are sampled) and higher computational & reactiveness overheads due to high sampling ratios.

Authors' addresses: Marco Chiesa, KTH Royal Institute of Technology, Kistagången 16, Kista, Sweden, 19139; Fábio L. Verdi, Federal University of São Carlos, Rod. João Leme dos Santos, km 110, Sorocaba, Brazil, 18052-780.

2476-1249/2023/3-ART8
https://doi.org/10.1145/3579321

In recent years, researchers have leveraged new advancements in programmable ASIC data-planes to cleverly keep and update monitoring data structures entirely in the data-plane [63, 73]. These data-plane approaches monitor each single packet (*i.e.*, no sampling) and therefore achieve *higher accuracy* than traditional approaches at the cost of higher data-plane on-chip memory utilization, which is renowned to be a scarce and expensive resource compared to control-plane memory [14]. We focus on these data-plane approaches in this paper due to their higher accuracy.

An emerging aspect of ASIC data-planes is that high-speed switches increasingly rely on parallelized packet processing pipe architectures due to slowing in Dennard's and Moore's laws [16, 39]. Each *pipe* interconnects a subset of the ports of the switch to a crossbar element and it only processes packets received/sent from/to a port on the switch. Importantly for this work, each pipe contains *dedicated* memory resources, *i.e.*, a pipe *cannot* access the memory of another pipe [37]. Today's switches may contain up to 16 parallel packet processing pipes [16].

In this work, we initiate the study of the problem of deploying *existing* data-plane-based monitoring solutions on multi-pipe switches. We focus on monitoring data structures deployed on ASIC switches and do not address data structures deployed on external devices such as SmartNICs or CPUs. Existing monitoring solutions store statistics about the forwarded traffic at different granularities, which we call *monitored traffic classes*, depending on the specific use case. For instance, network operators monitor each single TCP connection for rerouting large flows [3] whereas they monitor all traffic from each source IP addresses for super spreader detection [18].

We identify three main problems with multi-pipe deployments that result in lowered accuracy and reactiveness compared to an ideal single-pipe deployment (using the same amount of memory):

- *Inefficient memory utilization.* In a multi-pipe deployment, all the monitoring data structures are replicated across all pipes. However, packets belonging to the same monitored traffic class may arrive and leave the switch from multiple pipes, for instance, when monitoring source IPs or using a per-packet load balancer (*e.g.*, Flowlet [65], NDP [27]). In these cases, packets of the same monitored traffic class will *spread* over all these pipes, resulting in additional memory usage to store information about each traffic class.
- *Strict per-pipe memory resources.* Consider an 8-pipe switching chip where each pipe has only space to store statistics for 1 k flows. If there are 8 k incoming flows and these are non-uniformly spread across pipes, some flows will not be monitored. Conversely, in a single-pipe deployment, all the 8 k flows would be monitored.
- *Slow or inaccurate detection of network events.* When packets of the same monitored traffic class spread over multiple pipes, it becomes harder to detect a network event at data-plane speed as switches rely on internal CPUs to aggregate statistics across multiple pipes. Fetching arrays with hundreds of thousands of elements on the CPU of a switch may take up to seconds [55]. Delays in, for instance, detecting heavy hitters impact the ability to de-prioritize large flows, which leads to performance degradation for latency-sensitive flows with microsecond-level latency requirements (*e.g.*, high-priority RPCs) [5, 47].

In this work, we first quantify the impact of the above problems on two existing state-of-the-art data-plane monitoring solutions, *i.e.*, FCM-Sketch [63] and Elastic-Sketch [73]. We select these two mechanisms because they are both lightweights in terms of memory requirements and they can both be used to implement many monitoring tasks (*e.g.*, heavy-hitter detection, flow size entropy estimation, super-spreader detection). Our main results using real-world traces show that a naïve 16-pipe deployment of Elastic-Sketch or FCM-Sketch *requires up to* $8.5x$ *additional on-chip memory* to achieve a similar level of accuracy of a single-pipe deployment.

We therefore design a novel mechanism, called PipeCache, to *adapt existing* data-plane monitoring solutions to multi-pipe switch architectures. Our key idea is to deploy an existing monitoring solution on each pipe, partition traffic classes across all the available pipes, and store *all* monitoring

information about each monitored traffic class in the assigned pipe, *i.e.*, monitoring information about one monitored traffic class is *not* spread among pipes. To achieve this goal, we augment existing data-plane monitoring solutions with a cache to temporarily store monitoring information in a pipe before piggybacking it onto any existing packet that must be forwarded to the pipe where the cached state should be stored. Building PipeCache entails addressing one main challenge: keeping the cache small so as to achieve close-to-single-pipe accuracy and memory requirements. A large cache may outweigh the accuracy gains obtained by storing information about a traffic class in one location. Conversely, a small cache risks filling up too quickly, losing monitoring information, and lowering accuracy. We rely on three approaches in PipeCache to keep the cache small: *i)* piggybacking as many packets as possible given hardware constraints, *ii)* cloning a specific number of ad-hoc packets to drain the cache, and *iii)* relying on a fallback mechanism when the cache cannot be drained with the given specific number of ad-hoc packets. Creating and recirculating ad-hoc packets is a controllable overhead in PipeCache: the fallback mechanism kicks in whenever the recirculation overhead goes beyond a predefined threshold. Moreover, switches are equipped with internal recirculation ports that avoid consuming the bandwidth available on the external ports connected to the switch [38].

We evaluate PipeCache using existing real-world traffic traces augmenting both FCM-Sketch [63], Elastic-Sketch (ES) [73], HyperLogLog [24], MRAC [46], Beaucoup [18], and FlowLens [6] for different monitoring tasks: heavy-hitter detection, super-spreader detection, and entropy estimation of the flow distribution. Figure 1 shows that PipeCache reduces the memory requirements to achieve a certain accuracy level (*i.e.*, an F1-score of 0.9 for heavy-hitter detection) by up to 7x and 2x compared to a baseline deployment of Elastic-Sketch and FCM-Sketch on a 16-pipe switch, respectively. Moreover, we show in our evaluation (not in the figure) that PipeCache allows existing monitoring mechanisms to achieve close-to-single-pipe accuracy and memory requirements. We also show that cloning packets only results in minimal bandwidth overheads and that our fallback mechanism preserves information about packets that do not fit in the cache. Finally, we show that PipeCache is not specific to heavy-hitter detection, showing similar memory gains for a different monitoring task: super-spreader detection.

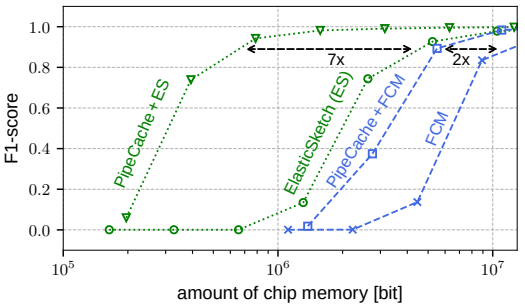

Fig. 1. PipeCache performance.

We summarize our contributions as:

- To the best of our knowledge, this is the first study to quantify the impact of deploying existing heavy-hitter monitoring data-plane solutions on today's high-speed multi-pipe switches. We highlight three main problems of deploying today's monitoring solutions on multi-pipe switches.
- We show that the on-chip memory requirements for multi-pipe deployment are significantly higher than with a single pipe, *i.e.*, up to 8.5x additional memory.
- A novel and general system, PipeCache, that temporarily stores and quickly moves monitoring information of each traffic class to its corresponding storage pipe.
- An implementation of PipeCache on a programmable ASIC switch.
- PipeCache reduces memory between 6*x* and 25x compared to pipe-oblivious deployments for computing heavy hitters, super spreaders, and flow-size and packet-length distributions.
- Showing that PipeCache can augment *any* existing monitoring work is left as future work.

2 MOTIVATION AND TRAFFIC ANALYSIS

In this section, we first introduce the architectural switch model based on multiple pipes. We then investigate the impact of deploying existing monitoring solutions across multiple pipes and we compare them to an ideal architecture based on a single pipe. Our results show that, under identical memory constraints, deploying a monitoring solution on multiple pipes may result in up to 3000x higher average error for the heavy-hitter size estimation task (compared to a single-pipe deployment). Similarly, a 16-pipe deployment may require up to 8.5x additional memory than a single-pipe deployment for achieving a close-to-zero error estimation.

2.1 Multi-pipe switch architectures

High-throughput multi-pipe switch architectures. Similarly to multi-core CPU architectures, switch vendors deploy multiple packet processing *pipes* (or packet processing engines) to sustain higher throughput rates. In fact, scaling throughput by increasing the clock frequency of each individual pipe results in excessive power consumption and increased latency [16]. We show a simplified representation of a switch architecture chip with two pipes in Figure 2.

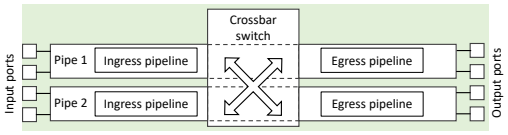

Fig. 2. Overview of a multi-pipe switch architecture.

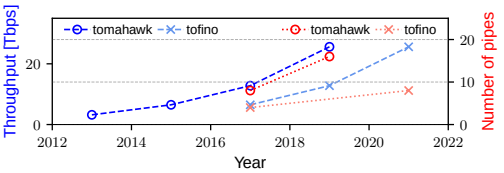

Fig. 3. Throughput and number of pipes trends.

A pipe connects a subset of the switch input/output ports to a crossbar entity, which is responsible for moving packets across pipes. Each pipe contains a certain amount of memory and computational resources (not shown in the figure), including SRAM, TCAM, and ALUs. These resources can be used to store any information needed to process packets according to an operator-configured packet processing pipeline. Each pipe executes two packet processing pipelines called the *ingress* and the *egress* pipelines. A switch executes the ingress pipeline before the packet reaches the crossbar entity and it executes the egress pipeline after it.

There is one constraint of multi-pipe switches that is key for this work [38]: *packets processed through one pipe can only read/write/update memory in that specific pipe.* Accessing memory on a different pipe at terabits-per-second speeds requires supporting concurrent data structures across pipes, which is a complex operation that may reduce the switch throughput because of the higher memory access latency of a shared memory [40, 72].

The number of pipes deployed on high-throughput switches has been steadily growing. We investigate current trends in the switching chip design industry with respect to multi-pipe deployments. Figure 3 shows on the left y-axis (bluish lines) the switch throughput of different generations of Broadcom Tomahawk (circle marks) and Intel Tofino switches (cross marks).

For both these vendors, throughput has doubled every two years reaching 25.6 Tbps in 2019 and 2021, respectively. Doubling the throughput of a single switch reduces power consumption in a data center network by a factor of 6x [36]. On the right y-axis of Figure 3, we show the number of pipes deployed on different generations of the same switches. We only show data that is publicly available and not covered by NDAs. Broadcom Tomahawk chips doubled their number of pipes between 2015 and 2017, from 8 to 16 pipes [15, 16] while Intel Tofino3 doubled the number of pipes from Tofino1, moving from 4 to 8 pipes[26, 39].

2.2 Deployment Issues

Deployment of existing data-plane network monitoring solutions on multi-pipe switches. With the emergence of programmable high-speed ASIC switches, researchers and practitioners have devised a large plethora of highly sophisticated data-plane network monitoring solutions running at data-plane speeds [11, 28, 41, 45, 53, 57, 63, 71, 73–75]. All these solutions operate at the level of a stream of packets that updates data structures in a single memory resource. However, on multi-pipe switch, packets belonging to the same monitored traffic class may traverse different pipes as we show in this section.

We first define more formally a *monitored traffic class* as the set of packets that share an identical subset of packet header values. The subset of packet header depends on the use case and define the granularity of the monitored traffic class. For instance, a monitored traffic class may be defined as all the packets with the same source IP address or all the packets with the same TCP/IP tuple (*i.e.*, the same TCP connection). In both these two cases, traffic belonging to the same monitored class may spread across different pipes because packets belonging to the same traffic class may arrive and leave a switch from different ports. For instance, packets that have the same IP source address may have different destination IP addresses. Since forwarding is performed based on the destination address, packets sharing the same source IP address may arrive and leave a switch from different pipes. Even when a traffic class is defined at the granularity of a TCP connection, packets belonging to the same TCP connection may be spread by a switch performing per-packet or per-burst load balancing (*e.g.*, NDP [27], LetItFlow [65], Drill [25], Conga [4], HULA [43], StarDust [80]). In both cases, due to the lack of a shared inter-pipe memory, the monitoring statistics would be stored on different locations (*i.e.*, different pipes). We now illustrate the implications of the above problem with three problems accompanied by examples. In all the following examples, it is irrelevant how a (monitored) traffic class is defined. It can be defined as a single flow, a source IP address, or something different. We will simply refer to a generic traffic class (or a flow). In the examples, we focus on the flow size estimation task.

Problem #1: Inefficient memory utilization due to spreading of monitoring information. The first problem with a multi-pipe deployment is that the state associated with the same monitored traffic class may spread over multiple pipes, fundamentally *reducing* the amount of available memory to store monitoring information. For example, consider a traffic class to be all packets with the same IP source address. Assume we use a simple array data structure to store the number of packets from each IP source address by indexing the entry in the array using the hash of the IP source address (as it is common in existing super-spreader sketch-based mechanisms [63]). Since flows with the same IP source address may be forwarded on arbitrary output ports, the traffic statistics associated with one traffic class would spread over all the existing pipes. This means that the state for storing one traffic class on a switch must be replicated over all pipes which, ultimately, reduces the available memory on the switch to store traffic statistics.

Consider the example in Figure 4 where the switch spreads the flows associated with three IP source addresses over two pipes. Without loss of generality, we assume that the array monitoring data structure is deployed on the egress packet processing pipeline. The first (red dashed) traffic class consists of 10 k packets, the second (blue solid) traffic class consists of 10 packets, and the third (green dotted) traffic class consists of 6 k packets. The switch equally splits the packets of the red and blue traffic classes between the two egress pipes while it splits

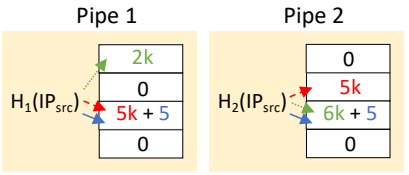

Fig. 4. Inefficient memory usage example.

the green flow with a 1:3 ratio between the two pipes. The switch uses two different hashes H_1 and H_2 to access the arrays in the two pipes to reduce collisions. Assume that the traffic of the blue and red (green and blue) IP addresses collide on the array in the first (second) pipe and does not collide on the second (first) pipe. To estimate the size of a traffic class, a natural approach would be to iterate over all pipes and sum up all the array entries where an IP address hashes to. This approach leads, however, to a large incorrect estimate of the blue traffic class whose size is estimated at 11 k packets instead of 10. The key problem is that information of one flow is spread on multiple pipes, increasing collisions, and effectively reducing the available memory to store traffic statistics.

Problem #2: Strict per-pipe memory resources. Our second example considers a network monitoring data structure that stores exact per-flow packet counters. Consider the same 2-pipe switch of Figure 4 where each pipe can store up to 4 flow entries and there are 8 incoming flows. If the switch forwards 2 flows to Pipe 1 and 6 flows to Pipe 2, then the switch will not have space to store state for the two extra flows forwarded to Pipe 2. The higher the traffic forwarding imbalance, the worst the utilization of the memory. The key problem is that a switch is constrained within the memory of each pipe as opposed to leveraging a shared memory across all pipes.

Problem #3: Slow/inaccurate detection of network events. An advantage of data-plane-based network monitoring is that the data-plane can either immediately trigger a modification of the forwarding state or trigger a notification to the control-plane about an observed network event. For instance, many existing systems keep a counter for each traffic class and trigger an action when a large traffic class is detected [3, 42]. In these works, the control-plane fetches all the data-plane data structures to identify large traffic classes, which has been shown to take up to seconds [55]. Ideally, when a counter reaches a pre-configured threshold, the data-plane should either modify the forwarding in the data-plane (*e.g.*, de-prioritizing a traffic class [5]) or report the traffic class to the controller [42, 48]. Monitoring and reacting entirely in the data-plane leads to faster reaction times as packets are processed in hundreds of nanoseconds. Unfortunately, data-plane monitoring and detection only works as long as monitoring information about a traffic class is stored on a single pipe. When the statistics are spread over multiple pipes, several problems arise. If one applies the same single-pipe threshold to each pipe, there is a risk of not identifying some events as the packets triggering that event have spread over multiple pipes. For instance, in Figure 4, if the threshold is set to 10 k packets, then none of the pipes identifies the red traffic class as the per-pipe counters are $\sim$ 5 k. Instead, one can reduce the threshold proportionally to the number of pipes, *e.g.*, a 10 k threshold translates to a 5 k per-pipe threshold on a 2-pipe switch. This mechanism catches all the large traffic classes of a single-pipe deployment, but it also erroneously catches a significant amount of small traffic classes that go above the threshold on some pipes (because packets may spread non-uniformly). For instance, in Figure 4, Pipe 2 erroneously detects the green traffic class as a large class as it receives 6 k packets in Pipe 1.

2.3 Traffic Analysis

To quantify the accuracy degradation due to the above problems, we simulate both Elastic-Sketch [73] and FCM-Sketch [63]. We first provide a minimal background on these mechanisms and we then analyze them. In this section, we focus on detecting heavy hitters, *i.e.*, traffic classes sending $\geq$ 10 k packets. Heavy-hitter detection is key in many use cases,*e.g.*, traffic engineering/prioritization [5], capacity planning [23], anomaly identification [49], DDoS detection [68], and more. We refer the reader to Appendix F for additional background information on detecting heavy hitters. In this subsection, we define a traffic class as a source IP address as in the FCM-Sketch work [63].

Background on Elastic-Sketch [73]. *Elastic-Sketch* contains a "heavy" and a "light" part. The "heavy" part of Elastic-Sketch consists of a set of array data structures that hold the *exact* traffic class identifier, the number of packets, and a "vote" metric that is used to evict entries from the

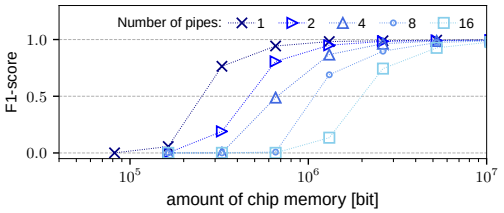

Fig. 5. Impact of multi-pipe deployment on the F1-score of an Elastic-Sketch monitoring data structure.

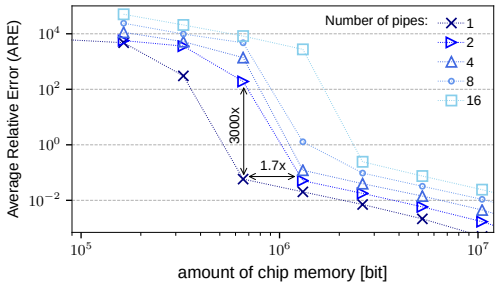

Fig. 6. Impact of multi-pipe deployment on the ARE obtained with Elastic-Sketch.

heavy part. The light part is a Count-Min sketch that stores traffic statistics for traffic classes that are not in the heavy part. To query the size of a traffic class, Elastic-Sketch sums up the counters of that traffic class from the heavy and the light parts. Since the heavy part contains the exact traffic class identifier, errors in the estimate only arise because of the light part. This mechanism keeps large traffic classes in the heavy part, limiting collisions between small and large traffic classes.

Background on FCM-Sketch [63]. *FCM-Sketch* has three levels of Count-Min sketches with decreasing the number of elements and increasing element size (*e.g.*, the 1^{st} level contains 2^{19} 8-bit entries while the 3^{rd} level contains 2^{13} 32-bit entries). A packet belonging to one traffic class increases an entry by 1 at level i only if the corresponding entry of the indexed element at level $i-1$ has reached the maximum value (*e.g.*, an entry at level 2 is increased by 1 only if the traffic class entry at the level 1 is 255). To query the size of a traffic class, one iterates from the first level and sums up the values until it reaches a level whose value is not the maximum one for that level. FCM-Sketch distills large from small traffic classes as only large classes appear in higher levels.

The accuracy of existing monitoring solutions quickly degrades with more pipes. We show through simulations that the higher the number of pipes on a switch, the lower the accuracy of a data-plane monitoring solution. We only show graph results for Elastic-Sketch (because of its lower memory requirements) and we mention summarized results for FCM-Sketch (whose graphs can be found in App. A). We implement Elastic-Sketch using a heavy part consisting of two array data structures and one light part consisting of a Count-Min sketch with a single hash (as suggested in the original paper for the hardware implementation). We deploy Elastic-Sketch on multiple egress pipes and estimate the size of a traffic class by summing up the per-pipe size estimates. We evaluate Elastic-Sketch using the CAIDA Equinix-NYC data monitor [17] from 2019 containing 21M packets, ~ 500 k distinct IP source addresses, of which 166 carry more than 10 K packets. We spread packets uniformly across pipes using a connection-consistent ECMP mechanism [19], which means connections with the same source IP address will be spread across the egress pipes.

Figure 5 shows the impact of a multi-pipe deployment on the F1-score (y-axis) of a heavy-hitter detector (with a threshold set at 10 k packets) with respect to the amount of *on-chip* memory used. The F1-score is defined as $\frac{2tp}{2tp+2p+fn}$ where tp, fp, and fn are the number of true positives, false positives, and false negatives, resp. An F1-score of one (zero) indicates high (low) accuracy. Our results show that on a 16-pipe deployment, it takes roughly 10x more memory to achieve the same F1-score of a single-pipe deployment. The results with fewer pipes also suggest that when ECMP spreads traffic on just two, four, or eight pipes, we still incur accuracy degradation.

Figure 6 shows the Average Relative Error (ARE) of the traffic class estimation for the reported heavy hitters, *i.e.*, $\sum_{tc \in HH} \frac{tc^* - tc'}{tc^*}$, where HH is the set of reported heavy hitters, tc^* is the real size of traffic class tc, and tc' is the estimated traffic class size.

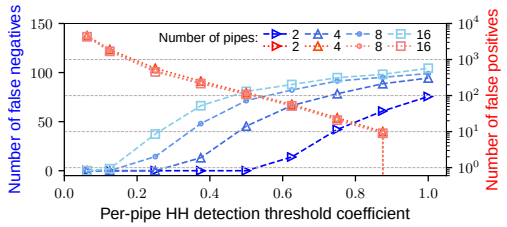

Fig. 7. Impact on Elastic-Sketch of using different per-pipe thresholds on the number of false negative and false positives for different number of pipes.

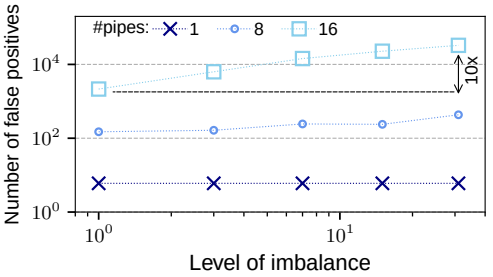

Fig. 8. Impact of traffic imbalances on the number of false positives with Elastic-Sketch.

Our results show that by using the same amount of memory, a single-pipe deployment reduces ARE by a factor of up to 3000x (*i.e.*, from a large ARE of $1.5 \times 10^2 \longrightarrow 15\,000\%$ to a much lower ARE of $0.05 \longrightarrow 5\%$). With a target ARE of $0.05 \longrightarrow 5\%$, we observe that it takes 1.7x additional chip memory every time we double the number of pipes on the switch. With the FCM-Sketch mechanism, we also observe some accuracy degradation. More specifically, it takes roughly $\sim$ 4x additional memory to achieve an F1-score above 0.9 on a 16-pipe switch. Moreover, using the same amount of memory, a single-pipe deployment reduces ARE by up to a 1000x factor compared to a 16-pipe switch.

Multi-pipe deployments make it harder to quickly detect heavy hitters. We showed in Section 2.2 that when a monitoring solution spreads over multiple pipes, it may be harder to quickly detect a network event such as a heavy hitter. Based on the same aforementioned CAIDA trace and Elastic-Sketch implementation, we measure in Figure 7 the number of false negatives (left y-axis, bluish lines) and false positives (right y-axis, reddish lines) using a different *per-pipe* threshold coefficient (x-axis) to compute the heavy-hitter threshold. This per-pipe coefficient is used to derive the per-pipe threshold. For instance, if the threshold on an 8-pipe switch is 10 K packets, a coefficient of 0.125 means each pipe reports a heavy hitter if the number of packets of a traffic class in any pipe exceeds 1 250 packets. Our results show that a substantial amount of either false positives or false negatives arise, *e.g.*, with 4 pipes and a coefficient of 0.6, we measure > 60 false negatives (out of 166 real heavy hitters) and > 50 false positives. Conversely, a single-pipe deployment correctly identifies *all and only* the correct heavy hitters (not shown in the figure). With a coefficient of 1.0, we have zero false positives (shown as a vertical red line). We observe identical results for FCM-Sketch. The reason why Elastic-Sketch and FCM-Sketch perform similarly is that both mechanisms achieve an F1-score of 1 in a single-pipe deployment. The false negatives and positives are only due to the per-pipe threshold mechanism and how traffic spreads across pipes.

Traffic imbalances degrade accuracy in multi-pipe switches. We showed in Problem 2 of Section 2.2 a case where a traffic imbalance prevented storing state for two flows. We now quantify this accuracy drop by configuring each heavy part array of Elastic-Sketch with 2^{13} elements. We generate an imbalance using weighted ECMP [79]. Figure 8 shows the number of false positives (y-axis) in single- and multi-pipe deployment as we vary the level of imbalance across pipes (x-axis). An *imbalance level* of x means that some pipes (*i.e.*, half of them in this experiment) receive up to x % less traffic than some other pipes. We see that the higher the number of pipes, the higher the number of false positives, up to 10x more false positives with 16 pipes and an imbalance of 31. We observe similar results for FCM-Sketch.

Other monitoring tasks. In this subsection, we focused on heavy-hitter detection, which is a key building block for different monitoring tasks [11, 29, 45, 53, 61, 63, 71, 73, 74]. In our evaluation section, we show a similar accuracy degradation for a variety of additional monitoring tasks.

In the rest of this paper, we tackle the following question:

"Can we devise a mechanism that supports existing monitoring solutions and achieves the accuracy and memory requirements of a single-pipe deployment on a multi-pipe switch?"

3 SYSTEM DESIGN

We now present a novel approach, called PipeCache, to reduce the overheads of deploying existing data-plane network monitoring solutions onto multi-pipe switches. The main goal of PipeCache is to achieve *both* the accuracy and memory requirements of a single-pipe deployment and support general existing monitoring solutions. We first present the general design of PipeCache (without considering hardware implementation constraints) and then present a hardware-aware design.

3.1 Overview

Design overview. Our system relies on a simple idea: storing *all* statistics information about *each* traffic class in one single pipe. To achieve this goal, we deploy existing monitoring data structures on the egress pipes and we assign each single traffic class to one egress pipe, *i.e.*, the monitoring information of each traffic class is stored only inside one egress pipe, which we call the *monitoring pipe* of that traffic class. This mapping can be realized using hash-based mechanisms that hash the identifier of a traffic class and use it to select an egress pipe. We show a high-level representation in Figure 9 where, for instance, the Blue traffic class is stored only on Pipe 1. The challenge is to update the monitoring information in the monitoring pipes *while* forwarding packets according to their forwarding state installed on the device. Packets belonging to the same traffic class may be forwarded on different egress pipes, making it difficult to aggregate information on a single pipe. For this reason, we introduce a *cache* data structure in the ingress pipe to briefly store information about those packets that must be forwarded to an egress pipe that is not the monitoring pipe of the traffic class of these packets. To transfer information from the cache to the correct monitoring pipes, we *piggyback* any information stored in the cache on existing data packets that are forwarded to the monitoring pipe of the cached elements.

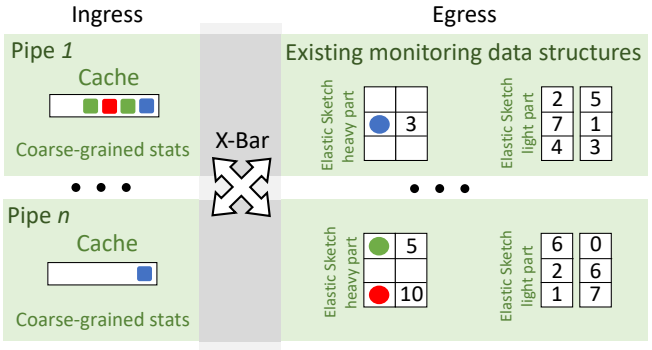

Fig. 9. Overview of PipeCache. Each monitored traffic class is stored in one egress pipe, called the monitoring pipe of the traffic class. For instance, the Red and Green traffic classes are stored in Pipe *n* while the Blue traffic class is in Pipe 1. A cache in each ingress pipe temporarily stores monitoring information for any packet that has been forwarded to an egress pipe that was not their corresponding monitoring pipe.

Simplified example. Consider the example of Figure 9 depicting a switch with *n* pipes. The egress pipes store the data structures needed by an Elastic-Sketch-like monitoring approach [73] but it could be a different mechanism. We refer the reader to Section 2 for background on Elastic-Sketch.

The heavy part of Elastic-Sketch is a key-value-like data structure while the light part is a Count-Min sketch. PipeCache maps each traffic class to an egress pipe (using the hash of the traffic class identifier), *e.g.*, the monitoring information about the blue traffic class is stored in Pipe 1 while information about the red and green traffic classes is stored on Pipe n. PipeCache deploys a cache on each ingress pipe. We do not store the entire packet in the cache but only any information needed to update the monitoring data structures (*e.g.*, the traffic class identifier or its hashes). In this example, our cache simply stores a list of traffic class identifiers, one for each packet that has been added to the cache. The cache of Pipe 1 currently stores information about three traffic classes (red, green, blue) while Pipe n only for one traffic class (blue). This information has been added into the cache whenever (red, green, or blue) packets were routed by the switch to a pipe that was not their monitoring pipe. For instance, assume that a packet belonging to the blue traffic class arrives at Pipe 1 and must be forwarded to Pipe n. PipeCache first checks whether the monitoring pipe of the blue traffic class is Pipe n. As this is not the case, PipeCache caches the identifier of the blue packet, it fetches from the cache any traffic class identifier that belongs to traffic classes stored in Pipe n, and it piggybacks them as packet metadata (*e.g.*, the green and red identifiers). When the packet arrives at the egress pipe of Pipe n, PipeCache updates the monitoring information with the carried metadata, *i.e.*, it updates the counter of the green (red) traffic class by 2 (1) units.

3.2 Design

The main challenge of PipeCache is to keep the cache data structure small. A large cache would deliver close-to-single-pipe accuracy yet at a very high memory cost due to the large cache. If these costs are too high, then traditional approaches would perform similarly if not even better than PipeCache. We describe three approaches that we use to keep the cache data structure as small as possible while preserving high heavy-hitter detection accuracy.

Approach #1: Piggybacking metadata. The first approach has been introduced in the previous overview of PipeCache. It boils down to piggybacking cached information on top of existing data-plane traffic to move it to the correct monitoring pipe. There is one important trade-off that we need to keep into account in the design of PipeCache. On the one hand, the more metadata we piggyback on a packet, the quicker a cache will be drained, supporting our goal to have a small cache. On the other hand, the more metadata we piggyback on a packet, the higher the cost in terms of switch resources needed to move this information to the egress pipe (more details in Section 3.3). It is therefore crucial to assess the impact of the amount of information that we piggyback on the ability of PipeCache to process traffic with a small cache while achieving high accuracy.

Trade-off analysis with piggybacked metadata. Consider the following example in which a 4-pipe switch receives a stream of 2^{20} packets where *i)* only a fraction $\eta = \frac{1}{8}$ of the packets (*i.e.*, 2^{17} packets) go to Pipe 1 and *ii)* Pipe 1 is *not* the monitoring pipe for these packets. Now, assume that the monitoring pipe for a fraction $\gamma = \frac{1}{4}$ of the packets (*i.e.*, 2^{18} packets) in the stream is Pipe 1. In other words, the number of packets whose monitoring pipe is Pipe 1 (*i.e.*, 2^{18}) is twice as much as the number of packets that are forwarded to Pipe 1 (*i.e.*, 2^{17}). According to basic queueing theory, if we want to prevent the cache occupancy from growing indefinitely, we need to piggyback *more than* $\frac{\gamma}{\eta} = 2$ traffic class identifiers from the cache every time we forward a packet. Note that, by carrying only 2 identifiers the cache occupancy would still grow indefinitely. In this way, we can use the 2^{17} packets forwarded to Pipe 1 to carry the 2^{18} identifiers that will be cached. In general, if we want to prevent the cache occupancy from growing indefinitely, we need to piggyback more than $\frac{\gamma}{\eta}$ traffic class identifiers. If $\frac{\gamma}{\eta}$ is higher than the amount of piggybacked identifiers, the cache will fill and information about the packets not fitting in the cache would be discarded. There exist therefore a tradeoff between the amount of information that must be piggybacked on every single packet, the

imbalance of the forwarded traffic, and the ability of the system to drain a cache. Whenever a cache fills completely and PipeCache does not manage to drain that cache using the available space in the metadata of a packet (which we evaluate in Section 5), PipeCache relies on the following two approaches to avoid losing monitoring information.

Approach #2: Generating ad-hoc packets. When a traffic packet hits a full cache, PipeCache clones the data packet and forwards the cloned copy to its monitoring pipe. When cloning the packet, the switch also fetches additional traffic class identifiers from the cache so that the cloned copy helps in draining the cache. This approach guarantees all the monitoring information is ultimately collected in its assigned monitoring pipe. However, creating clones of a packet has two main costs: *i)* switches have a constraint on the amount of packets that can be cloned and *ii)* a cloned packet is an additional packet that the switch must process, increasing the risk of packet drops. We note that the bandwidth requirements are unlikely to be the main issue as the cloned packets only contain a few traffic class identifiers and are therefore small in size. If the number of cloned packets exceeds a certain threshold (potentially "zero" if an operator does not want to clone & recirculate packets), we resort to our third approach.

Approach #3: Resorting to a fallback mechanism. When a traffic packet hits a full cache *and* the packet cannot be cloned as described in Approach #2 (because we already cloned more packets than what the pre-configured threshold allows us to do), PipeCache simply forwards the packet to its egress pipe without caching the traffic class identifier and it updates the monitoring data structures in the non-monitoring pipe. We say that such identifiers have been *deflected*. PipeCache keeps statistics about the number of deflected packets at a configurable granularity and uses these statistics to estimate the size of each traffic class. When PipeCache estimates the size of a traffic class, it reads the counters on the monitoring pipe and then uses the "deflected" statistics to adjust the traffic class size accordingly. The granularity at which PipeCache keeps deflection statistics impacts the overall memory consumption. We rely on coarse-grained statistics about the number of deflected packets from each ingress to each egress pipe. A critical challenge is the potential inaccuracy of the traffic class estimation due to the coarse granularity of the deflected statistics. When there are traffic classes with extreme bursts, there is a risk that the number of deflected packets for these traffic classes is much higher than the pipe-to-pipe amount. This inaccuracy could lead to underestimating the size of a traffic class. We therefore use a burst detector to mark such traffic classes. We estimate the size of these traffic classes by reading their counters across all pipes (from the control plane). The fallback mechanism guarantees that PipeCache does not perform worse than a naïve deployment under traffic imbalances among egress pipes or bursty traffic.

3.3 Tackling Hardware Constraints

There are several important aspects and constraints to keep in mind when deploying PipeCache on an ASIC switch. If PipeCache were to require performing complex operations, then its deployment could be undermined. In this section, we look at the existing constraints known from publicly available documents about programmable ASIC switches (*e.g.*, Intel Tofino [37]). These constraints are likely to be present (in some forms) on any high-speed switch ASIC.

The main constraint is the one related to the number of operations that can be performed on a memory location. Programmable ASIC switches configure portions of their SRAM memories to be read/written as indexed arrays directly at data-plane speed to be used as *registers*. A stateful ALU reads/writes at *one single* index of the register array per-processed packet. Therefore, one cannot read/write two elements at different indices in a register array.

The implications of the above constraint on PipeCache are relevant and worth discussing here. PipeCache must perform two critical operations: *Operation 1)* pushing and fetching *multiple* traffic

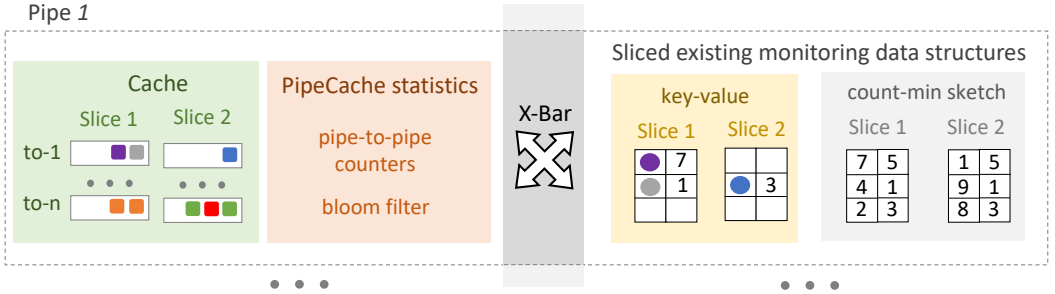

Fig. 10. Adapting PipeCache to hardware constraints.

class identifiers from the cache at data-plane speed, *Operation 2)* quickly identifying which traffic class identifiers in the cache should be stored on the egress pipe where a traffic packet is being forwarded. Operation 1 means the cache must be implemented with a register. However, fetching multiple identifiers requires to have independent registers as a single register can only read one element. We therefore split both the cache and the existing data structures into multiple *slices* where each traffic class is mapped exactly to one slice. See Figure 10 for a schematic architectural representation. When PipeCache fetches elements from the cache, it fetches at most one element from each slice of the cache. Similarly, the fetched identifiers will update distinct registers in the egress pipe. Operation 2 requires deploying per-egress-pipe caches (identified in the figure with "to-1,…, to-n") as we cannot perform a complex search inside a register array to find which identifiers should go to a certain monitoring pipe. Having dedicated per-slice per-egress-pipe caches increases the memory requirements compared to a shared cache as packets of a traffic class can only be cached in their assigned per-slice per-pipe cache.

Discussion. As explained in the motivation, PipeCache helps in reducing memory requirements by *i) storing per-monitored-class statistics at one single location and ii) leveraging memory resources across all pipes*. If the workload and the monitoring task are such that *1)* all packets belonging to the same monitored traffic class are not spread across distinct pipes and *2)* the set of incoming packets for all monitored traffic classes is spread uniformly across pipes, then PipeCache will not reduce the memory requirements. We show one such example in App. D in Fig. 30.

4 IMPLEMENTATION

PipeCache is a packet processing primitive that could be realized as part of an ASIC functionality. To verify its feasibility, we implement PipeCache on top of the FCM-Sketch Tofino implementation [22]. We split each Count-Min sketch into 4 slices implemented with registers. We implement the cache using a stack, which requires two register arrays. The number of register elements used for the caches grows linearly depending on the number of slices and not the number of pipes. We verify the correct implementation of the system running the same CAIDA traffic traces used in the evaluation.

ASIC Resource Usage. We summarize the overhead of our preliminary implementation of PipeCache (only Approach #1) on a 2-pipe Intel Tofino1 switch in Table 11. We set the size of the FCM levels as in the original FCM paper. We split each level into 4 slices with one-fourth the original size. We use a per-pipe per-slice cache of 100 elements. From the table, we observe that PipeCache requires a little bit of

Resource	FCM	PipeCache w/ FCM
SRAM	18.04%	20.94%
TCAM	0.00%	3.47%
VLIW Instruction	5.80%	13.80%
Exact Match X-Bar	4.13%	11.46%
TCAM Match X-Bar	0.00%	5.05%

Fig. 11. ASIC resources used by PipeCache.

additional memory compared to a straightforward FCM-Sketch multi-pipe implementation. The overhead is due to the additional cache and its logic. We currently use some TCAM to extract all bits from the header fields to select the target pipe, the forwarded pipe, and the slice. This could be optimized using bit-slice operations as a ternary match is not necessarily needed. We observe a higher usage of the exact match crossbar (X-Bar) due to the additional cache logic needed to update the stack data structures in the ingress pipelines. Our code could be optimized to reduce the impact of the cache logic on the number of used registers.

5 EVALUATION

We now evaluate the effectiveness of PipeCache on real-world traffic traces, quantifying both its benefits and overheads compared to state-of-the-art pipe-oblivious approaches. We augment Elastic-Sketch, FCM-Sketch, HyperLogLog, MRAC, Beaucoup, and FlowLens with our cache-based pipe-aware monitoring mechanism PipeCache. We first focus on heavy-hitter detection, which is a key part of many monitoring solutions. We then report results on super-spreader detection and entropy estimation of the flow size distribution. We answer the following questions:

- *"Does PipeCache reduce memory requirements?"*
- *"What are the packet recirculation overheads?"*
- *"Does the fallback mechanism identify all heavy hitters with high accuracy?"*
- *"Does PipeCache quickly detect heavy hitters?"*
- *"What is the best number of slices?"*
- *"Does PipeCache cope with traffic imbalances and bursty traffic?"*
- *"Does PipeCache generalize to other monitoring tasks?"*

Our results indicate that four slices suffice to achieve close-to-single-pipe accuracy under load imbalances and bursty traffic. PipeCache requires minimal bandwidth overheads that fit within the internal recirculation port of a switch and it requires minimal additional memory resources to store cached traffic class identifiers, ultimately achieving up to $\sim$ 10x memory reduction compared to pipe-oblivious implementations of existing monitoring solutions. Our code is publicly available [1].

Heavy hitter detection and accuracy metrics. We focus on identifying IP source addresses carrying large amounts of traffic (as in [63]). Each traffic class is identified by the 32-bit IPv4 source address of each packet. We measure the following accuracy metrics:

- *F1-score*, defined as $\frac{2tp}{2tp+2p+fn}$ where tp, fp, and fn are the number of true positives, false positives, and false negatives, resp. An F1-score of 1 means high accuracy (*i.e.*, all and only heavy hitters are reported) while an F1-score of 0 means inaccuracy (*i.e.*, zero heavy hitters correctly reported).
- *Average Relative Error (ARE)* as the average traffic-class size error, *i.e.*, $\sum_{tc \in HH} \frac{tc^* - tc'}{tc^*}$, where HH is the set of reported heavy hitters, tc^* (tc') is the real (reported) size of traffic class tc.

Resource metrics. We quantify the amount of resources used by PipeCache and the baselines along two dimensions:

- *memory*: we measure the amount of used memory in bits across all pipes. The *cache memory overhead* parameter controls the size of the cache memory with respect to the memory of an existing monitoring solution. For example, if Elastic-Sketch consumes 1 MiB, a cache memory overhead of 10% consumes 100 KiB for the cache entities.
- *packet recirculation*: we measure the amount of additional bandwidth (in Gbps and packets per second) used by PipeCache when recirculating cloned packets.

Traffic traces and forwarding mechanism. We run our simulations using two distinct CAIDA traces from 2019. We expect to observe similar trends for datacenter traces. The first CAIDA trace contains 21 M packets, roughly 500 traffic classes (*i.e.*, source IP addresses), and 166 traffic classes generate at least 10 k packets. The second trace contains roughly 100 M packets, 2 M traffic classes,

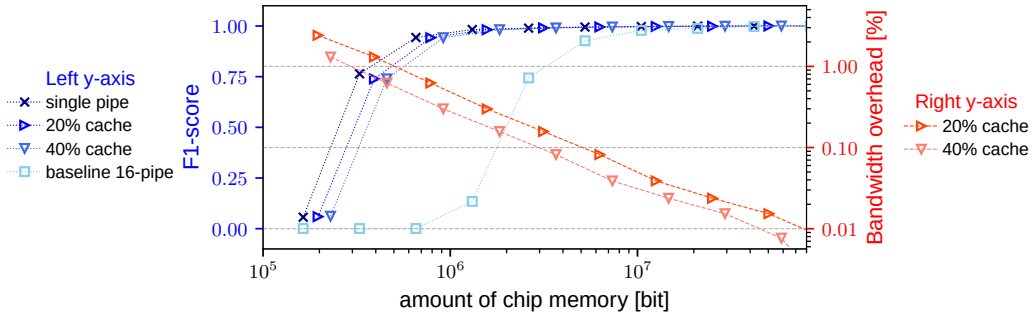

Fig. 12. PipeCache with packet recirculation when augmenting Elastic-Sketch [73]

and 564 traffic classes generate at least 10 k packets. We obtain quantitatively similar results and show them only for the first trace. We replay the traces through one ingress pipeline and spread the traffic over the egress pipelines using an *ECMP*-like flow-consistent spreader. We note that using an ECMP load balancer gives us a reasonably fair way of modeling how traffic towards different IP destinations could be forwarded. In fact, in this evaluation, a traffic class is a source IP address so traffic spreads across egress pipes based on the destination IP address. We generate a *traffic imbalance* by changing the weights of a traffic spreader. A traffic imbalance of x means that some pipes have a weight that is x times higher than some other pipes, *i.e.*, they receive x times more traffic. An imbalance of 1 means traffic is forwarded uniformly. An imbalance of 31 means one port receives 31% less traffic than another port. We repeat each simulation 5 times changing the seed.

Existing data-plane monitoring configuration. We configure Elastic-Sketch using two sets of arrays for the heavy part and a single array for the light part using the hardware approach described in the original paper. The two arrays in the heavy part are indexed using two different hash functions and each pipe uses a different set of hash functions. We set the λ value, which is used to determine when to evict a traffic class from the heavy part, to 8. We configure FCM-Sketch using 3 levels of Count-Min sketches. Each sketch contains two arrays indexed with two different hash functions. The bit size of the element is 8, 16, and 32 for the 1st, 2nd, and 3rd levels. The number of entries is reduced by 8x per increasing level. We show graph results for Elastic-Sketch and report summaries for FCM-Sketch (with key graphs in App. B). Packet recirculation overheads do not depend on the monitoring solutions but only on the number of pipes, number of slices, cache sizes, and traffic traces. We note that FCM-Sketch requires more memory than Elastic-Sketch to achieve the same F1-score (as previously shown in Figure 1, meaning that FCM-Sketch will use larger caches in our simulations (thus incurring significantly lower packet recirculation overheads).

PipeCache achieves close to single-pipe accuracy. We first quantify the accuracy and performance overheads of the PipeCache version that recirculates cloned packets when a cache is full and the fallback mechanism is disabled. Figure 12 shows the F1-score (left y-axis, blue) over the per-bit memory utilization (x-axis) for both the single-pipe (cross mark) and 16-pipe (square mark) deployments as well as the 16-pipe deployment with a cache memory overhead of 20% (right-facing triangle mark) and 40% (bottom-facing triangle mark). We let PipeCache recirculate as many packets as needed so that all information ends up being stored in the correct pipe.

We first observe that the F1-score of our cache-based approaches is close to the one of the single-pipe deployment. Compared to the 16-pipe baseline deployment (square marks), PipeCache reduces the memory requirements by 6x for any F1-score above 0.7. As for FCM-Sketch, we reduce memory consumption between a factor of 2 – 3.3x for F1-scores above 0.9 (see Figure 26 in App. B).

PipeCache detects heavy hitters without aggregating information from multiple pipes. A side result of Figure 12 is that PipeCache quickly identifies all heavy hitters without aggregating information from different pipes. Conversely, the baseline approach from Figure 7 in Section 2 suffers from either a large number of false positives or false negatives. Specifically, PipeCache detects *all and only* the heavy hitters of Figure 7 without any false positives while the baseline detects all the heavy hitters with 1000s of false positives with a per-pipe detection coefficient of $\frac{1}{16}$.

PipeCache *quickly* detects heavy hitters at the per-pipe level. Caching packet information as in PipeCache may delay the detection of a heavy-hitter. The impact of this delay is however negligible for one main reason. The larger per-slice queue in all our simulations is 256, which means we may delay at most 256 packets of a single traffic class. Any heavy-hitter with a size that is 256 packets larger than the heavy-hitter detection threshold would be detected as packets would start to be recirculated. Moreover, in expectation, each cache contains packets from different traffic classes, which would further lower the detection gap. We measure this delay by comparing an ideal single-pipe detection and PipeCache using the 16-pipe deployment of Figure 7. We measure the average *heavy-hitter detection packet delay*, which measures after how many additional packets (from the instant we generate the 10000th packet of a traffic class) PipeCache detects a heavy-hitter. Our results show that the average detection delay is 2 633 packets. On a 25.6-Tbps switch and average packet sizes of 1 kB, this delay translates to 800 nanoseconds. The CPU of a switch would require seconds to perform the same detection [55].

PipeCache requires minimal bandwidth overheads. We have shown that PipeCache achieves a similar performance to single-pipe deployment. However, we achieve this by recirculating cloned packets whenever caches are full, which increases bandwidth overheads. Without a cache, almost all packets need to recirculate as they are likely to be forwarded towards any of the 15 pipes that do not store state for that packet. Recirculating so many packets would pose high overheads on today's ASIC switches. Figure 12 shows the bandwidth overheads (right red y-axis) for our cache-based solutions. We see that the bandwidth overheads are always below 3% and quickly decrease as we use more memory for both Elastic-Sketch and our cache. The recirculation overheads are higher when we consider the percentage of recirculated packets (not explicitly shown in the graph). The percentage of recirculated packets can be obtained by multiplying the bandwidth overhead by $\frac{900\text{B}}{72\text{B}} = 12.5$, where 900 B is the average packet size in our trace and 72 B is our recirculated packet. Our graphs show that to achieve an F1-score above 0.8 (0.99), PipeCache only incurs a bandwidth overhead below 0.2% (0.08%) with a 40% cache memory overhead, which translates to recirculating 2.5% (1%) of the total number of packets. We make the following two important observations about these observed packet recirculation overheads. First, today's ASIC switches are designed to handle more than 2.5% of recirculated packets with internal recirculation ports. Second, by recirculating 2.5% of the packets, a switch would achieve line-rate for packets with an average size of 308 bytes instead of 300 bytes [39], which is minimal overhead.

False negatives arise without packet recirculation. We now show that limiting packet recirculation leads to false negatives, *i.e.*, undetected heavy hitters. Figure 13 shows the number of true positives (left blue y-axis) and the number of false positives (right red y-axis) over different memory utilizations (x-axis) for different thresholds on the number of packets that can be recirculated: 0% (which only counts packets forwarded to their monitoring pipe), 5%, and 20%. All these simulations use a cache memory overhead of 20%. Our graph shows several interesting results. First, when we use a small memory (left side of the graph), we miss detecting a significant amount of heavy hitters (blue lines below 166) and the number of false positives is relatively high. Second, when we increase the amount of memory, we observe that the number of true positives (*i.e.*, correctly detected heavy hitters) increases and the number of false positives decreases. In fact, by using more

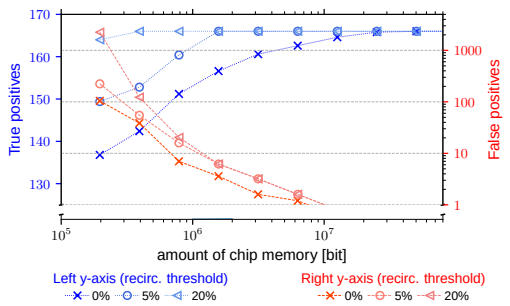

Fig. 13. Impact of packet recirculation threshold on the heavy hitter accuracy task (Elastic-Sketch).

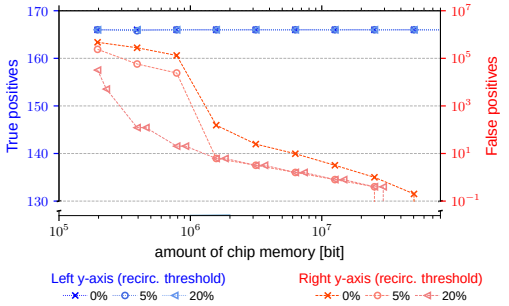

Fig. 14. Impact of the fallback mechanism on the accurate detection of heavy hitters (Elastic-Sketch).

memory, our cache becomes larger and we recirculate fewer packets. Surprisingly, even with a 0% threshold, PipeCache identifies *all* the 166 heavy hitters as the cache is large enough to absorb bursts of traffic going to non-monitoring pipes.

The fallback mechanism of PipeCache increases the detection of heavy hitters. We have shown in Figure 13 that when we limit the percentage of packets that can be recirculated, a non-negligible fraction of heavy hitters go undetected, especially when using less memory (left side of the graph). We now show the impact of using the fallback mechanism proposed in Section 3. The fallback mechanism kicks in whenever a cache is full and the percentage of recirculated packets has reached a pre-configured limit. Instead of disregarding a packet, the fallback mechanism forwards the packet to its designed egress port (and pipe) and updates the data structures on that pipe. Figure 14 shows again the number of true positives (left blue y-axis) and false positives (right red y-axis) over different memory sizes for PipeCache with different packet recirculation thresholds when using the fallback mechanism. Our results show that the fallback mechanism trades a higher number of false positives for higher true positives: it identifies all the 166 heavy hitters (blue lines at the top of the graph) at the cost of a higher number of false positives (compared to Figure 13). We note that the number of false positives at low memory utilization is high. Due to high packet recirculation, the fallback mechanism behaves similarly to the 16-pipe baseline.

Four slices better absorb bursts. The higher the number of slices, the larger the amount of information that can be transferred from the cache to the monitoring data structures in the egress pipes. Intuitively, the higher the number of slices, the lower the packet recirculation needs as the caches can be flushed quicker. However, the higher the number of slices, the smaller the per-slice cache size (as we have the same cache memory overhead partitioned over a higher number of cache slices). Figure 15 shows the percentage of recirculated packets (y-axis) over the amount of utilized memory for a different number of slices: 2 (cross mark), 4 (triangle mark), 8 (square mark), and 16 (circle mark). We set the number of pipes to 16 and the cache memory overhead to 20%. Our results show that with 4 slices we obtain the lower packet recirculation overhead. With 2 slices, PipeCache does not piggyback enough identifiers to drain the caches. With 8 or 16 slices, the caches are instead too small compared to 4 slices.

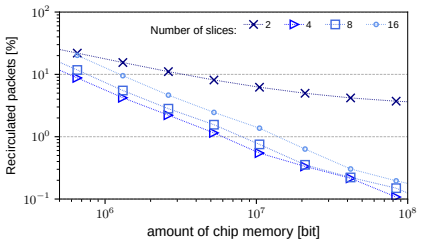

Fig. 15. Recirculation overheads vs. # slices.

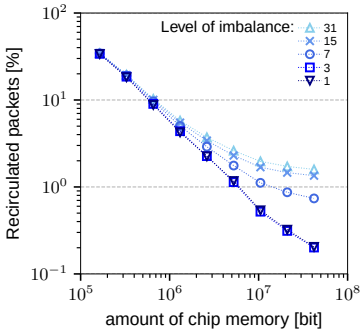

Fig. 16. Impact of load imbalances on packet recirculation.

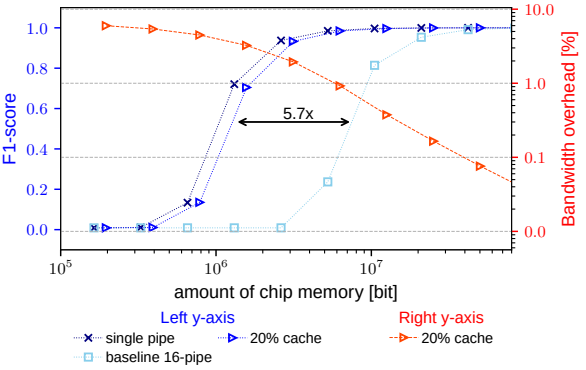

Fig. 17. Impact of bursty traffic.

PipeCache copes with traffic imbalances. We show how traffic imbalances affect the percentage of recirculated packets. Figure 16 shows the percentage of recirculated packets (y-axis) over the amount of utilized memory (x-axis) for different levels of imbalance: 1 (bottom-facing triangle marks), 3 (square marks), 7 (circle marks), 15 (cross marks), and 31 (top-facing triangle marks). An imbalance of 1 is a uniform load balance while an imbalance of 31 means that some pipes (*i.e.*, one pipe in this test) receive 31x less traffic than other pipes. Our results show that imbalances affect the amount of recirculated packets. Even with an imbalance of 31, the amount of recirculated packets is below 2% with higher memory utilization. We note that with an "infinite" imbalance, the fallback mechanism performs close to the baseline (not shown in the graphs). One way to better handle traffic imbalances would be to change the assignment of the traffic classes to egress pipes and map more traffic classes on pipes that receive more traffic. We leave such optimizations as future work.

PipeCache handles bursty traces with little recirculation bandwidth overheads. When there are many packets of the same flow that arrive "back-to-back", the cache data structure fills up and more packets must be recirculated to drain the cache. We measure the impact of bursty traffic on PipeCache by synthetically increasing the number of back-to-back packets in the analyzed trace. We define the *level of burstiness* of a trace as the average number of back-to-back packets with the same flow identifier. For example, if the trace contains the sequence of packets $< A, A, A, A >$, the burstiness level is 1 while a sequence $< A, B, A, B >$ has a burstiness level of 0. The original Caida trace has a burstiness level of 0.07. In this paragraph, we increase the burstiness level of the CAIDA trace to 0.86 (highly bursty) by replicating packets. Figure 17 shows the F1-score (right blue y-axis) and the extra bandwidth requirements (left red y-axis) while varying the available memory (x-axis) for the single-pipe (cross marks), 16-pipe baseline (square marks), and PipeCache with 20% cache (triangle marks) deployments and a burstiness level of 0.86. Our results show that PipeCache retains almost a 6x memory reduction w.r.t. to the 16-pipe baseline. Compared to the original trace of Figure 12 (which has a burstiness level of 0.07), we see that the bandwidth overhead increases by a factor of ~10x because of the higher number of packet recirculations needed to drain the caches. To achieve an F1-score of 0.9, the bandwidth overhead of PipeCache is just 2%, which is within the available internal recirculation resources of existing programmable switches.

PipeCache is not specific to heavy hitters. Case #1: detecting superspreaders. We implement HyperLogLog [24] to identify super spreaders in the same traffic trace and augment it with PipeCache. A *super spreader* is a source IP address that sends traffic to more than a certain number of *distinct* destination IP addresses (as opposed to the large amount of packets as in a heavy-hitter detection task). We set the detection threshold to 128 addresses. We obtain

similar results with other threshold values. Figure 18 shows the number of true positives (left blue y-axis) and false positives (right red y-axis) over different memory sizes for the single-pipe deployment (cross marks), a 16-pipe deployment HyperLogLog baseline (square marks), and PipeCache with HyperLogLog (triangle marks). The results show that PipeCache performs very closely to a single-pipe deployment, both in terms of true and false positives. We note that the 16-pipe deployment baseline requires up to 11.7x more memory to achieve the same number of false positives of PipeCache.

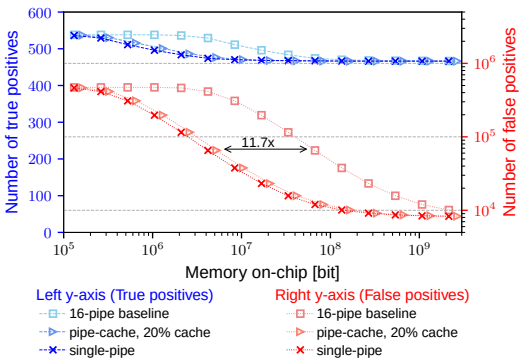

Fig. 18. Super-spreader detection with PipeCache.

We observe that the true positive line for the 16-pipe baseline deployment is higher than PipeCache and single-pipe deployments. This should *not* be interpreted as a gain of the 16-pipe baseline. The true positive line of the 16-pipe deployment is higher because there are also many false positives, *i.e.*, almost every monitored traffic class is reported as a super spreader. This is also the case for the leftmost points of the single-pipe and PipeCache deployments, which detect ~550 true positives with ~ 10^5 bits of memory. We note again that the gains of PipeCache over the 16-pipe baseline come from storing all the monitoring information in a single pipe instead of spreading it over all pipes. This translates to a larger available memory, especially on next-generation high-speed switches with multiple pipes. Recirculation overheads are identical to the above heavy-hitter detection evaluation.

PipeCache is not specific to heavy hitters. Case #2: Entropy estimation. We implement the basic version of the MRAC algorithm [46] (as in the Elastic-Sketch work [73]) to compute the entropy of the flow distribution. The *entropy* of the flows distribution is computed as $-\sum_i (i\frac{n_i}{m} \log \frac{n_i}{m})$ where n_i is the number of flows whose size is i packets, m is the sum of all the n_i, and i iterates between 1 and the size of the largest flow. Figure 19 shows the estimated entropy (y-axis) along different memory constraints (x-axis) for both a single-pipe (cross marks) and a 16-pipe MRAC baseline deployment (square marks) as well as PipeCache with MRAC (triangle marks). The horizontal red dashed line represents the real entropy of the trace. We observe that PipeCache reduces the memory requirements for implementing MRAC on a 16-pipe switch by 13x compared to the baseline.

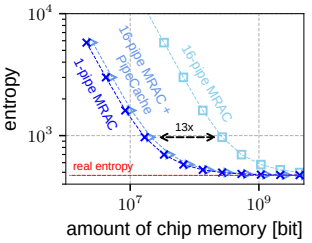

Fig. 19. Entropy estimation.

PipeCache brings benefits to existing multi-query monitoring systems. We implement Beaucoup [18] to support two queries at the same time: *i)* detecting IP source address super spreaders and *ii)* IPpair heavy hitters. We select these two types of queries for two reasons. First, the traffic classes are defined at different granularities, *i.e.*, in the super-spreader query a traffic class is an IP address while in the heavy-hitter query it is a pair of IP addresses. Second, the type of "counting" (*i.e.*, the so-called attribute of the query [18]) is different, *i.e.*, in the super-spreader query, we count distinct elements while in the heavy-hitter query we count the number of packets. We note that the size of the cache element in PipeCache does not necessarily increase with the number of supported queries. In fact, Beaucoup only executes one query per packet. Therefore the size of the cache element is bounded by the largest number of hashes needed in a single query. In

this example, the super-spreader query needs to store both the hash of the IP source (to update the correct element) and that of the IP destination (to update the element with the correct value). In the heavy-hitter query, we only need to store the 32-bit hash of the IP pair, which means the size of the cache element is 64 bits. We analyze the accuracy of Beaucoup under two different traffic spreading mechanisms: ECMP-based (as in previous sections) and per-packet-random (where packets with the same 5-tuple may be spread across pipes, as in NDP [27], Hula [43], etc.). Figure 20 and Figure 21 show the F1-score (y-axis) for detecting superspreaders and heavy hitters (resp.) along with different memory requirements (x-axis) using either a single-pipe (cross marks) or a 16-pipe Beaucoup baseline deployment (square marks) or PipeCache with Beaucoup (triangle marks), resp. In both figures, packets are spread using per-packet-random. The results show that PipeCache reduces the memory requirements to achieve an F1-score of 0.75 by 14x and 4x for the super-spreader and the heavy-hitter tasks, resp. We now consider an ECMP packet spreader (figures in App. C). In this case, we observe different behaviors for the super-spreader and the heavy-hitter tasks. In the super-spreader case, packets belonging to the same traffic class spread over multiple pipes since packets with the same IP source address may have different destinations. However, for the heavy-hitter task, the CAIDA trace is such that packets belonging to the same IP-pair are unlikely to belong to different 5-tuples. This means that packets of the same monitored traffic class are likely to traverse always the same pipe. In this case, we do not observe benefits in deploying PipeCache for reducing the memory requirements in the heavy-hitter task. We note that it helps to deploy PipeCache even if in only one of the monitored tasks packets from the same monitored traffic class spread across pipes (*e.g.*, super-spreader monitoring).

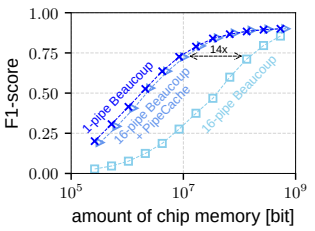

Fig. 20. Beaucoup superspreader

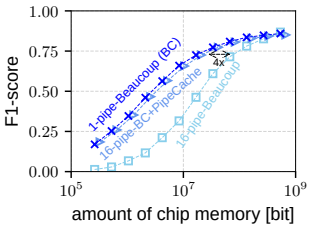

Fig. 21. Beaucoup heavy-hitter.

PipeCache brings benefits for flow-sampling-based monitoring queries. Some monitoring systems sample a set of flows from the currently forwarded ones and keep statistics only about those flows. FlowLens [6] is an example of such a system, which relies on a data structure to store per-connection information. We implement FlowLens to support the monitoring task of computing the per-flow packet length distribution. We use the difference between the real packet length distribution and the estimated one by normalizing the two distributions and summing up the estimation errors for each "packet length bin" used in FlowLens. If the sum of the errors is smaller than 5%, we say that the packet length distribution of that flow is *accurate*. Figure 22 shows the number of accurately estimated per-flow packet-length distributions (y-axis) under different memory requirements (x-axis) using either a single-pipe (cross marks) or a 16-pipe FlowLens baseline deployment (square marks) or PipeCache with FlowLens (triangle marks). We first use per-packet-random to spread packets. The results show that there are significant gain in using PipeCache to correctly estimate the packet length flow distributions. We then repeated the simulation with the ECMP-based packet spreader (refer to App. D for graphs). As expected, in this case, we do not see any gains because the monitored traffic classes (*i.e.*, identified with 5-tuple) do not spread across multiple pipes. However, when traffic imbalances across pipes arise, we observe a 2x memory reduction when using PipeCache.

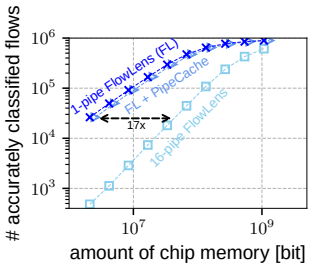

Fig. 22. FlowLens comparison.

6 DISCUSSION

We discuss aspects of PipeCache that relate to *i)* its potential memory gains, *ii)* its limitations, *iii)* its deployment within general network monitoring tools, and (iv) covert channel attacks.

6.1 Conditions for Memory Gains in PipeCache

There are different factors that affect whether PipeCache will bring any memory reduction:

- *Spreading packets of a traffic class.* When packets belonging to the same traffic class spread across multiple pipes, PipeCache brings substantial memory gains by avoiding duplicating information on multiple pipes. If packets belonging to the same traffic class do not spread across multiple pipes, then PipeCache may not bring memory benefits (as in Figure 30 in Appendix D).
- *The level of imbalance across pipes.* Even if packets belonging to the same traffic class do not spread on multiple pipes, PipeCache may still reduce memory requirements if packets are forwarded with some levels of imbalance since some pipes may end up not fully utilizing their memory resources (see Figure 31 in App D).
- *The amount of information that should be stored in the cache per packet.* PipeCache requires storing a certain amount of information about a packet in the cache. In an IP-source-address heavy-hitter monitoring task, this information is just the IP source address. In a 5-tuple heavy-hitter, PipeCache does not have to store the entire 5-tuple as it can simply store the hash of that tuple, which can be used to access the data structures in the egress pipe later on. In the IP-source super-spreader task, PipeCache needs to store both the IP-source address (to access the correct element in the egress pipe) and the destination IP address (to correctly update the value of the accessed element). Therefore, in this case, the memory requirements of PipeCache doubles. Memory requirements also increase with multi-query monitoring systems. In the worst case, PipeCache must store the exact 5-tuple as well as other per-packet attributes (*e.g.*, packet length, inter-packet gap, TCP options).
- *The data structures used to monitor traffic.* PipeCache brings different levels of memory reduction when using Elastic-Sketch or FCM-Sketch for the same monitoring task. Our evaluation shows that PipeCache brings high memory benefits with all the mechanisms that store an identifier of the traffic class (see Figure 12 for Elastic-Sketch, Figure 20 for Beaucoup, and Figure 22 for FlowLens). Our evaluation shows that such high gains may also be achievable without storing per-traffic-class identifiers (as in HyperLogLog, Figure 18). Formally understanding the impact of multi-pipe deployments on monitoring mechanisms is future work.

6.2 PipeCache limitations

There are two aspects of the PipeCache implementation that require future work.

Reducing the number of register arrays. The number of registers used in PipeCache grows quadratically in the number of pipes and this may become a limiting factor on ASIC programmable switches. We note that our implementation of PipeCache is just a feasibility check. The cache functionality of PipeCache could be an in-built primitive in the hardware of each pipe.

Parameter tuning. There exist one key parameter in PipeCache that needs to be tuned depending on the burstiness and level of imbalance of the input trace: the threshold used to enable Approach #3. A threshold of 0 would disable PipeCache. A threshold that is too high would result in a high recirculation overhead. In our simulations, we set the threshold to 8, *i.e.*, a packet is not cached and directly forwarded to the egress pipe based on its forwarding information whenever PipeCache detects in the countable bloom filter that it has cloned and recirculated packets in the element pointed by the forwarded packet 8 times. A more thorough study of the impact of this parameter on the performance of PipeCache is left as future work.

6.3 PipeCache and covert channel attacks

We discuss the ability of PipeCache to detect different types of covert channel network attacks:

- *Packet size distributions.* In this type of attack, information is leaked using different packet sizes. A common way to detect this attack is to track sudden changes in the packet size distribution. Tracking packet size distributions per-traffic-class requires storing information on registers and therefore suffers from memory overhead in multi-pipe switches. We showed a 25x memory reduction by using PipeCache for these type of covert channel attacks.
- *Inter-packet gap (IPG) time.* In this type of attack, information is leaked by delaying consecutive packets by different amounts of time. These types of attacks are cumbersome to detect on multi-pipe switches because packets spreading on multiple pipes lead to incorrect inter-packet-gap measurements. IPGs are measured by subtracting timestamps between consecutive packets. If consecutive packets spread across multiple pipes, they arrive on different pipes one cannot measure the inter-packet gap. PipeCache helps in moving all the IPG measurements on the same pipe. PipeCache could compute the *average* inter-packet gap of each traffic class, however, computing the distribution of inter-packet gaps would be more complex because inter-pipe cached information is not moved to the egress in order.
- *Storage in packet headers.* In this type of attack, information is leaked inside TCP header fields and is detected by monitoring rare values in these fields using data structures in the data plane. Similarly to detecting packet length distributions, PipeCache could lower memory requirements as it would store information about a single traffic class on a single pipe.

6.4 Deployment beyond ASIC switches

In this work, we focused on monitoring data structures that are deployed on ASIC switches. Networking monitoring is, however, a complex operation that entails collecting and analyzing traffic information at different locations using a different types of devices. Monitoring solutions such as *Flow[62], SmartWatch [58], NetWarden [70], and FlowLens [6]) rely on a combination of ASIC switches, SmartNICs, and general-purpose CPUs to detect network events at different levels of granularity and performance. ASIC switches offer high packet processing throughput with constrained packet processing logic (*e.g.*, no decryption) and constrained memory. Conversely, general-purpose CPUs support expressive packet processing logic at a lower throughput. The goal is to offload as much as possible the monitoring logic to ASIC switches and rely on SmartNICs or CPUs to perform more complex (and fine-grained) monitoring operations.

PipeCache brings benefits to such monitoring solutions by reducing the memory footprint on ASIC devices. We showed in our evaluation that PipeCache significantly improves the accuracy of the packet length distribution in FlowLens [6], which would intuitively result in better detection accuracy on external devices. Solutions that rely on a two-level detection mechanism (*e.g.*, SmartWatch [58]), where the ASIC identifies suspicious traffic at a coarse grainularity and forwards it to external devices for fine-grained analysis, may also benefit from PipeCache. Our mechanism would reduce memory requirements for the data structures deployed on the switch for identifying suspicious traffic.

7 RELATED WORK

There exists a rich body of literature on network monitoring.

Single-switch monitoring. Extensive research has been carried out in recent years on data-plane monitoring on a single switch [73] [9, 11–13, 18, 33, 34, 45, 48, 52–54, 56, 59, 62, 63, 77]. The goal of these systems is to design algorithms for extracting traffic statistics from traffic given the constrained memory and computational capabilities of a switch. These works do not address issues arising from multi-pipe switch architectures and are therefore orthogonal to our work.

Network-wide monitoring. Several existing works have explored the problems of collecting traffic statistics from individual switches to build a network-wide view of the data-plane conditions [2, 8, 10, 10, 20, 21, 30, 31, 35, 50–54, 62, 64, 71, 78]. These works model each switch as a single node, which we show is not correct when using multi-pipe switches. One can see a multi-pipe switch as a special instance of a network where a pipe is equivalent to a node of the network. Most of the existing work of network-wide monitoring aggregates statistics using a controller. Some of these works (*e.g.*, FlowRadar [52]) pull the data-plane data structures at the control-plane level (*e.g.*, through the CPU of the switch), which is renowned to be slow. Some other works (*e.g.*,, LightGuardian [78]) extract the traffic statistics directly at the data-plane and may piggyback it on data packets to reach end hosts and transmit those statistics to a controller. Conversely, we focus on the impact of multi-pipe deployments and we collect all information in one pipe, thus achieving faster detection. Some network-wide works tackle the problem of selecting which traffic classes are monitored at a switch (*e.g.*, CFS [10], AROMA [7]). PipeCache solves a completely different problem, *i.e.*, collecting monitoring information of packets that are spread over multiple locations at a single location.

Awareness of multi-pipe switches. Khooi et al. [44] first raised awareness about designing multi-pipe-aware systems. The authors state that "*in-network traffic monitoring applications operate as intended on multi-pipeline switches*". In this work, we show the opposite conclusion. Finally, MP5 [60] is a recent work on multi-pipe ASIC programmable switches. It does not however tackle the problem of heavy-hitter detection, it does not show the impact of multi-pipe deployments on the accuracy of the detection, and we do not see how MP5 could be easily extended to solve it.

Alternative hardware implementations. The dRMT switch architecture envisions a shared memory inside a pipe that can be accessed from all its match/action stages. However, dRMT does not aim for inter-pipe shared memory and, as far as we know, there are not yet commercial products that support it. Solving the inter-pipe memory problem is renowned to be an extremely difficult problem [40]. Trio [72] is a programmable ASIC switch that supports a DRAM memory shared across packet processor engines. DRAM is a memory for bulk transfers and suffers from performance limitation when random access is needed [69] as in the case of different packets pointing to a different part of the memory, making it difficult to use for per-packet level telemetry.

Distributed shared state. SwiSh [76] handles distributed state in a network using programmable switches by replicating the same information across multiple switches. Thus, SwiSh replicates information in multiple locations instead of collecting it at one location. Conversely, PipeCache stores monitoring information at one single location with substantial memory savings.

8 CONCLUSIONS

In this work, we unveiled teh impact of deploying existing monitoring solutions on multi-pipe switches. We presented PipeCache, a system that adapts a variety of monitoring mechanisms to multi-pipe switches. Our evaluation shows up to 16x memory savings. PipeCache relies on piggybacking metadata, generating ad-hoc packets, and resorting to a fallback mechanism to store monitoring information on a single pipe. We envision extending PipeCache to network-wide monitoring and store monitoring data in exactly *one* location in the network.

ACKNOWLEDGEMENTS

We would like to thank Amedeo Sapio, Gianni Antichi, the anonymous reviewers, and our shepherd K.K. Ramakrishnan for their insightful comments. This work has been partially supported by the Swedish Research Council (agreement No. 2021-04212), KTH Digital Futures, and CNPq/SAAB (No. 200220/2020-9).

REFERENCES

[1] PipeCache code repository. https://bitbucket.org/pipecache/pipecache-simulations/.

[2] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. HeteroSketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 719–741, Renton, WA, April 2022. USENIX Association.

[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 19, USA, 2010. USENIX Association.

[4] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 503–514, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Pias: Practical information-agnostic flow scheduling for commodity data centers. *IEEE/ACM Trans. Netw.*, 25(4):1954–1967, aug 2017.

[6] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando M. V. Ramos, and André Madeira. Flowlens: Enabling efficient flow classification for ml-based network security applications. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[7] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. Routing oblivious measurement analytics. In *2020 IFIP Networking Conference (Networking)*, pages 449–457, 2020.

[8] Ran Ben Basat, Gil Einziger, Shir Landau Feibish, Jalil Moraney, and Danny Raz. Network-wide routing-oblivious heavy hitters. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, page 66–73, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Ran Ben Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, and Erez Waisbard. Memento: Making sliding windows efficient for heavy hitters. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 254–266, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Ran Ben Basat, Gil Einziger, and Bilal Tayh. Cooperative network-wide flow selection. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–11, 2020.

[11] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking*, 28(3):1172–1185, 2020.

[12] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C. Luizelli, and Erez Waisbard. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 127–140, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, aug 2013.

[15] Broadcom. Tomahawk3 - BCM56980 12.8 Tb/s Multilayer Switch , 2019. https://docs.broadcom.com/doc/56980-DS.

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

[17] CAIDA. The caida ucsd anonymized internet traces - 20190117., 2019. https://docs.broadcom.com/doc/12398014.

[18] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 226–239, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Marco Chiesa, Guy Kindler, and Michael Schapira. Traffic engineering with equal-cost-multipath: An algorithmic perspective. *IEEE/ACM Trans. Netw.*, 25(2):779–792, 2017.

[20] Vitalii Demianiuk, Sergey Gorinsky, Sergey I. Nikolenko, and Kirill Kogan. Robust distributed monitoring of traffic flows. *IEEE/ACM Transactions on Networking*, 29(1):275–288, 2021.

[21] Damu Ding, Marco Savi, Gianni Antichi, and Domenico Siracusa. An incrementally-deployable p4-enabled architecture for network-wide heavy-hitter detection. *IEEE Transactions on Network and Service Management*, 17(1):75–88, 2020.

[22] fcm project. Fcm-sketch: Generic network measurements with data plane support (github), 2021.

[23] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational ip networks: Methodology and experience. *SIGCOMM Comput. Commun. Rev.*, 30(4):257–270, aug 2000.

[24] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[25] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 225–238, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Vladimir Gurevich and Andy Fingerhut. P4_16 Programming for Intel Tofino using Intel P4 Studio, 2021. https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf.

[27] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery.

[28] Zijun Hang, Mei Wen, Yang Shi, and Chunyuan Zhang. Interleaved sketch: Toward consistent network telemetry for commodity programmable switches. *IEEE Access*, 7:146745–146758, 2019.

[29] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S. Muthukrishnan, and Jennifer Rexford. Carpe elephants: Seize the global heavy hitters. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, SPIN '20, page 15–21, New York, NY, USA, 2020. Association for Computing Machinery.

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

[33] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 576–590, New York, NY, USA, 2018. Association for Computing Machinery.

[34] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. Toward Nearly-Zero-Error sketching via compressive sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 1027–1044. USENIX Association, April 2021.

[35] Qun Huang, Haifeng Sun, Patrick P. C. Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 404–421, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Ian Cox. Broadcom Tomahawk4: Industry's Highest Bandwidth Chip, 2019. https://www.youtube.com/watch?v=B-COGMbaUg4.

[37] Intel. P416 intel tofino native architecture - public version, 2021.

[38] Intel. P416 Intel® Tofino™ Native Architecture – Public Version, 2021. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf.

[39] Intel. Intel Tofino 3 Intelligent Fabric Processor Brief, 2022. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-brief.html.

[40] Intel Research Academy Forum (confidential access only). TX at ingress stage, 2021. https://community.intel.com/t5/Intel-Connectivity-Research/TX-at-ingress-stage/m-p/1284697/highlight/true#M2362.

[41] Rhongho Jang, DaeHong Min, SeongKwang Moon, David Mohaisen, and DaeHun Nyang. Sketchflow: Per-flow systematic sampling using sketch saturation event. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 1339–1348, 2020.

[42] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. *SOSR '16: Proceedings of the Symposium on SDN Research*, (10):1–12, mar 2016.

[44] Xin Zhe Khooi, Levente Csikor, Jialin Li, and Dinil Mon Divakaran. In-network applications: Beyond single switch pipelines. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 1–8, 2021.

[45] Xin Zhe Khooi, Levente Csikor, Jialin Li, Min Suk Kang, and Dinil Mon Divakaran. Revisiting heavy-hitter detection on commodity programmable switches. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*,

pages 79–87, 2021.

[46] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, page 177–188, New York, NY, USA, 2004. Association for Computing Machinery.

[47] Kumar, Gautam and Zhang, Yiwen and Dukkipati, Nandita and Wu, Xian and Vahdat, Amin. Admission Control for Latency-Critical Remote Procedure Calls in Datacenters, 2022. United States Patent Application 20220239598 https://www.freepatentsonline.com/y2022/0239598.html.

[48] Jan Kučera, Diana Andreea Popescu, Han Wang, Andrew Moore, Jan Kořenek, and Gianni Antichi. Enabling event-triggered data plane monitoring. In *Proceedings of the Symposium on SDN Research*, SOSR '20, page 14–26, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, page 201–206, New York, NY, USA, 2004. Association for Computing Machinery.

[50] Jonatan Langlet, Ran Ben-Basat, Sivaramakrishnan Ramanathan, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. Zero-cpu collection with direct telemetry access. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, page 108–115, New York, NY, USA, 2021. Association for Computing Machinery.

[51] Yiran Li, Kevin Gao, Xin Jin, and Wei Xu. Concerto: Cooperative network-wide telemetry with controllable error rate. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 114–121, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 311–324, Santa Clara, CA, March 2016. USENIX Association.

[53] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 101–114, New York, NY, USA, 2016. Association for Computing Machinery.

[54] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A High-Performance Switch-Native approach for detecting and mitigating volumetric DDoS attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3829–3846. USENIX Association, August 2021.

[55] Hun Namkung, Daehyeok Kim, Zaoxing Liu, Vyas SekaR, and Peter Steenkiste. *Telemetry Retrieval Inaccuracy in Programmable Switches: Analysis and Recommendations*, page 176–182. Association for Computing Machinery, New York, NY, USA, 2021.

[56] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. SketchLib: Enabling efficient sketch-based monitoring on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 743–759, Renton, WA, April 2022. USENIX Association.

[57] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. SketchLib: Enabling efficient sketch-based monitoring on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association.

[58] Sourav Panda, Yixiao Feng, Sameer G Kulkarni, K. K. Ramakrishnan, Nick Duffield, and Laxmi N. Bhuyan. Smartwatch: Accurate traffic analysis and flow-state tracking for intrusion prevention using smartnics. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, page 60–75, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. Continuous in-network round-trip time monitoring. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 473–485, New York, NY, USA, 2022. Association for Computing Machinery.

[60] Vishal Shrivastav. Stateful multi-pipelined programmable switches. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '22, 2022.

[61] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 164–176, New York, NY, USA, 2017. Association for Computing Machinery.

[62] John Sonchack, Oliver Michel, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 823–835, Boston, MA, July 2018. USENIX Association.

[63] Cha Hwan Song, Pravein Govindan Kannan, Bryan Kian Hsiang Low, and Mun Choon Chan. Fcm-sketch: Generic network measurements with data plane support. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 78–92, New York, NY, USA, 2020. Association for

Computing Machinery.
[64] Lu Tang, Qun Huang, and Patrick P. C. Lee. A fast and compact invertible sketch for network-wide heavy flow detection. *IEEE/ACM Transactions on Networking*, 28(5):2350–2363, 2020.
[65] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 407–420, Boston, MA, March 2017. USENIX Association.
[66] Shobha Venkataraman, Dawn Song, Phillip B Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. Technical report, Carnegie-Mellon Univ Pittsburgh Pa School Of Computer Science, 2004.
[67] Fábio Luciano Verdi and Marco Chiesa. Heavy hitter detection on multi-pipeline switches. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '21, page 121–124, New York, NY, USA, 2021. Association for Computing Machinery.
[68] Chenxu Wang, Tony T. N. Miu, Xiapu Luo, and Jinhe Wang. Skyshield: A sketch-based defense system against application layer ddos attacks. *IEEE Transactions on Information Forensics and Security*, 13(3):559–573, 2018.
[69] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. Shuhai: Benchmarking high bandwidth memory on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119. IEEE, 2020.
[70] Jiarong Xing, Qiao Kang, and Ang Chen. Netwarden: Mitigating network covert channels while preserving performance. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.
[71] Kaicheng Yang, Yuanpeng Li, Zirui Liu, Tong Yang, Yu Zhou, Jintao He, Jing'an Xue, Tong Zhao, Zhengyi Jia, and Yongqiang Yang. Sketchint: Empowering int with towersketch for per-flow per-switch measurement. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–12, 2021.
[72] Mingran Yang, Baban Alex, Kugel Valery, Libby Jeff, Mackie Scott, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Ghobadi Manya. Using Trio – Juniper Networks' Programmable Chipset – for Emerging In-Network Applications. In *ACM SIGCOMM '22*.
[73] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 561–575, New York, NY, USA, 2018. Association for Computing Machinery.
[74] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proc. VLDB Endow.*, 10(11):1442–1453, aug 2017.
[75] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 29–42, USA, 2013. USENIX Association.
[76] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. Swish: Distributed shared state abstractions for programmable switches. In *USENIX NSDI 2022*. USENIX, April 2022.
[77] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 207–222, New York, NY, USA, 2021. Association for Computing Machinery.
[78] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. LightGuardian: A Full-Visibility, lightweight, in-band telemetry system using sketchlets. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 991–1010. USENIX Association, April 2021.
[79] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *EuroSys'14*, New York, NY, USA, 2014. Association for Computing Machinery.
[80] Noa Zilberman, Gabi Bracha, and Golan Schzukin. Stardust: Divide and conquer in the data center network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 141–160, Boston, MA, February 2019. USENIX Association.

APPENDIX

A MOTIVATION (FCM-SKETCH)

In this appendix, we report the analogous graphs of Section 2 for the FCM-Sketch mechanism.

In Figure 23, we observe that it takes roughly ~ 4x additional memory to achieve an F1-score above 0.9 with a 16-pipe switch compared to the single-pipe deployment.

In Figure 24, we observe that under the same memory constraints, a single-pipe deployment reduces ARE by up to a 1000x factor compared to a 16-pipe switch (from roughly 4000% to 4%). This is visible for a memory utilization of roughly 10 Mb.

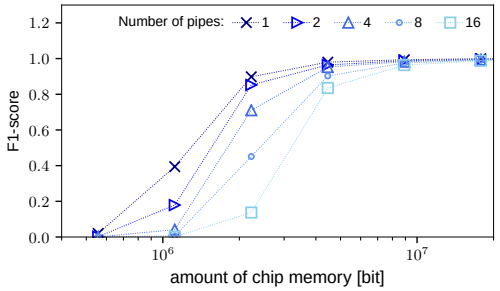

Fig. 23. Impact of multi-pipe deployment on the F1-score of an FCM-Sketch-like monitoring data structure.

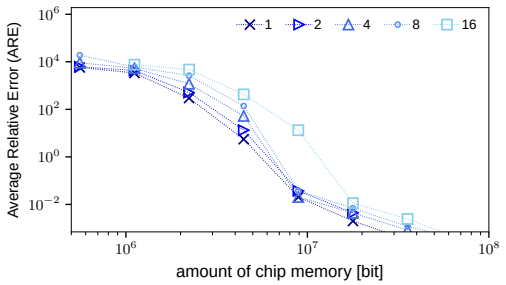

Fig. 24. Impact of multi-pipe deployment on the average relative flow estimate error (ARE) of an FCM-Sketch-like monitoring data structure.

In Figure 25, we report the number of false negatives and false positives for different HH detection thresholds. With the FCM-Sketch mechanism, we observe an identical level of accuracy degradation as in Elastic-Sketch. The reason why Elastic-Sketch and FCM-Sketch performs similarly is that both mechanisms achieve an F1-score of 1 in a single-pipe deployment. The false negatives and positives are only due to the per-pipe threshold mechanism. We indeed observe that *the set* of false positives and false negatives is roughly identical between Elastic-Sketch and FCM-Sketch.

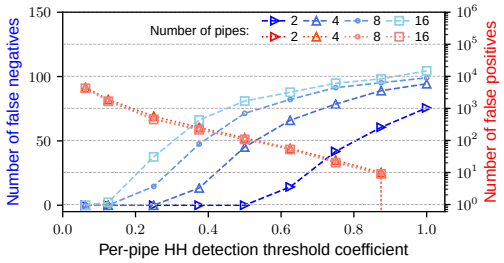

Fig. 25. Impact on FCM-Sketch of using different per-pipe thresholds on the number of false negative (blue lines) and false positives (red lines) for different number of pipes using the same amount of per-chip memory.

B EVALUATION (FCM-SKETCH)

For FCM-Sketch, we reduce memory consumption between a factor of $2 - 3.3$x for F1-scores above 0.9 (see Figure 26) with a cache memory overhead of 20%. The bandwidth overhead are much lower than Elastic-Sketch because FCM-Sketch requires higher memory to achieve a high F1-score. This means that the caches used by PipeCache with FCM-Sketch are also larger than Elastic-Sketch for for similar F1-scores.

Figure 27 shows the number of true positives and false negatives with FCM-Sketch when using the fallback mechanism. We observe that also in this case, PipeCache identifies all the 166 heavy hitters at the cost of a higher number of false positives. We note that the lines at 5% and 20%

recirculation rates overlaps completely as PipeCache recirculates below 5% of the packets. We observe that the number of false positives goes to 0 with a memory utilization of 86 Mb.

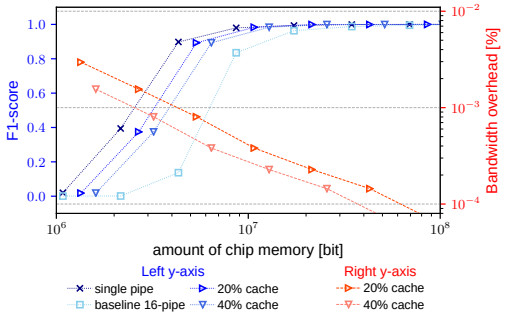

Fig. 26. Accuracy of PipeCache with FCM-Sketch and packet recirculation.

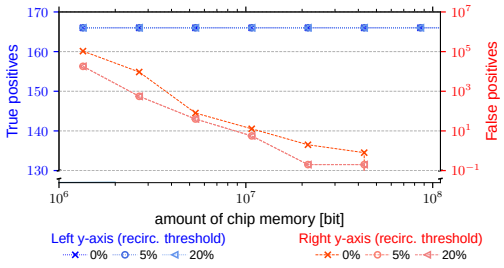

Fig. 27. Impact of the packet recirculation threshold with a fallback mechanism with FCM-Sketch.

Packet recirculation. We do not show for FCM-Sketch the graphs that focus on packet recirculation. In fact, packet recirculation overheads do not depend on the monitoring solution but only on the number of pipes, number of slices, cache sizes, and traffic trace. This means that the lines are *identical* to Elastic-Sketch.

C BEAUCOUP WITH ECMP

In this section, we quantify the impact of multi-pipe deployment for the Beaucoup monitoring system and the gains obtainable with PipeCache. We spread packets across pipes using ECMP, which is flow-consistent (*i.e.*, all packets with the same 5-tuple traverse the same pipe). We implement two queries: super-spreader detection and heavy-hitter detection. Following the design of Beaucoup, we let each packet only update either the super-spreader detector or the heavy-hitter detector. Figure 28 and Figure 29 show the F1-score (y-axis) under different memory requirements (x-axis) for Beaucoup deployed on a single-pipe, Beaucoup deployed on a 16-pipe switch, and PipeCache deployed together with Beaucoup on a 16-pipe switch.

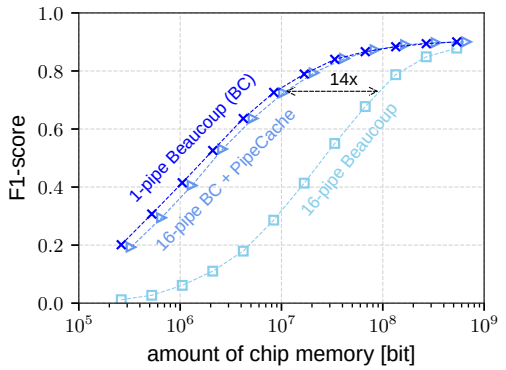

Fig. 28. Accuracy of PipeCache when extending Beaucoup for the super-spreader monitoring task (with ECMP packet spreading).

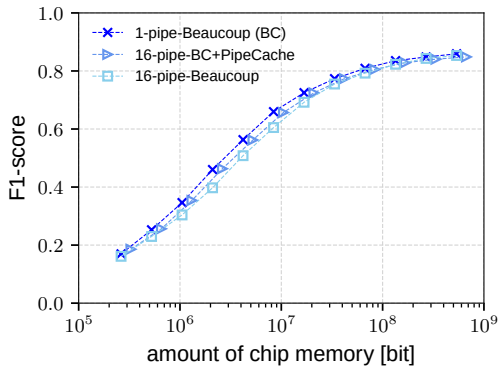

Fig. 29. Accuracy of PipeCache when extending Beaucoup for the heavy-hitter monitoring task (with ECMP packet spreading).

We observe that the memory requirement gains for the super-spreader task (Fig. 28) are consistent with those observed when the packets were spread using per-packet-random (Fig. 20). This is an

intuitive result since a traffic class in the super spreader monitoring task aims at detecting IP source address with many distinct destinations. Since the CAIDA trace contains many IP source addresses with multiple destinations, even with an ECMP packet spreader, information about the same IP source address would be stored at different pipes. Conversely, the memory gains for the heavy-hitter monitoring task (Fig. 29) almost disappears when packets are spread using ECMP instead of using per-packet-random (Fig 21). This can be explained as follows. In the heavy-hitter monitoring task, a traffic class consists of an IP-pair and, in the CAIDA trace, different IP pairs tend to have one or a few different 5-tuple connections. This means that most flows traverse a single or a few pipes and therefore their information is not spread across all pipes. It is worth observing that PipeCache still brings some small gains. It is worth noting that, even in the absence of gains for one or more monitoring tasks, an operator should deploy PipeCache whenever at least one monitoring task the gains are significant. We note that PipeCache could be deployed in Beaucoup just for the monitoring tasks that exhibit large gains by using PipeCache. For the other queries that do not exhibit large gains (*e.g.*, heavy-hitter monitoring), one should fallback to a traditional approach and avoid allocating cache memory for updating their data structures.

D FLOWLENS WITH A FLOW CONSISTENT PACKET SPREADER (ECMP)

We now show how PipeCache performs on workloads where packets belonging to the same monitored traffic class do not spread across different pipes. We consider the same implementation of FlowLens used for Figure 22 but we now spread packets across pipes using ECMP. Since a traffic class in FlowLens is a 5-tuple and ECMP is flow-consistent, this means that all packets belonging to the same monitored traffic class will traverse a single ingress and egress pipe. Figure 30 shows the number of accurately estimated per-flow packet-length distributions (y-axis) under different memory requirements (x-axis) using either a single-pipe (cross marks) or a 16-pipe FlowLens baseline deployments (square marks) or PipeCache with FlowLens (triangle marks).

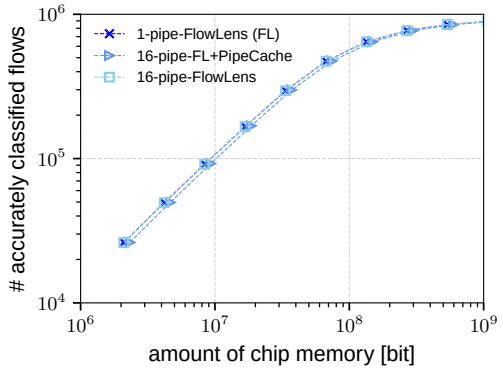

Fig. 30. Accuracy of PipeCache when extending FlowLens for computing the packet-length distribution of the 5-tuple connections with ECMP packet spreading.

We can observe that all the lines are close to each other. PipeCache retains the same accuracy of the baselines, however, with the additional cost of storing the cache data structure.

We now show that, however, when there are traffic imbalances across pipes, PipeCache may lead to memory savings. We therefore quantify the memory requirement gains of using PipeCache when the traffic across the pipes is imbalanced. We replay the CAIDA trace with a level of imbalance of 31. Figure 31 shows the number of accurately estimated per-flow packet-length distributions (y-axis) under different memory requirements (x-axis) using either a single-pipe (cross marks) or

a 16-pipe FlowLens baseline deployments (square marks) or PipeCache with FlowLens (triangle marks). Our results show that, even when each single traffic class does not spread across different pipes, PipeCache reduces the memory requirements up to a factor of 2x under an imbalanced traffic workload.

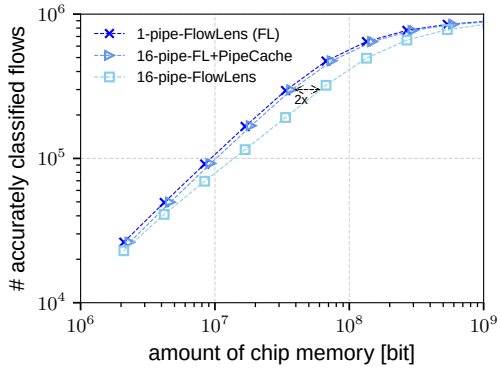

Fig. 31. Accuracy of PipeCache when extending FlowLens for computing the packet-length distribution of the 5-tuple connections with ECMP packet spreading under a high imbalance.

E SINGLE- VS. ALL- EGRESS-PIPE CACHE DEPLOYMENT

A natural question to ask when using PipeCache is whether it is best to deploy an existing monitoring solution across all egress pipes or just on one single egress pipe as was done in a previous work [67]. In the *all-egress-pipes* deployment scenario, we use all the egress pipes to store per-traffic-class statistics so that each traffic class is stored exactly in one of those egress pipes. In the *single-egress-pipe* deployment scenario, we store all the traffic classes in exactly one single egress pipe, *i.e.*, the other egress pipes do not store anything. A single-egress-pipe deployment simplifies the logic of the system and can potentially better absorb bursty traffic as explained in the following. Consider an 8-pipe switch in which we deploy all the monitoring data structures on a single egress pipe, for instance, Pipe 1. Assume we use two slices. Since there is a single egress pipe, we only need to deploy two caches in one ingress pipe: one cache for each of the two slices. Conversely, in a deployment where we store the monitoring data structures across all pipes, we would need 16 caches (two slices times eight egress pipes). Having fewer caches means that, under the same memory constraints, one can configure larger caches in the single-egress-deployment compared to the all-egress-pipes deployment, which helps in absorbing bursty traffic. There is however a cost in using a single-egress-pipe deployment: one needs more slices (than an all-egress-pipe deployment) to prevent the cache occupancy to grow indefinitely even under uniformly-balanced traffic across pipes. Consider the same calculations that we carried out in Section 3.2 and assume a uniform spreading of the packets across all egress pipes. With a single-pipe-deployment, a fraction $\eta = \frac{1}{8}$ of the packets go to the only monitoring pipe (*e.g.*, Pipe 1) and all packets (*i.e.*, $\gamma = 1$) must be stored on that pipe. This means that one needs > 8 slices to keep the cache occupancy low. Conversely, in an all-egress-pipes deployment, we have that only a fraction $\gamma = \frac{1}{8}$ of the packets have Pipe 1 as their monitoring pipe. This means that two slices are sufficient to prevent the cache occupancy from growing indefinitely under uniform traffic balance across pipes. We evaluate these two approaches in Section 5.

We compare the two PipeCache deployment approaches described in Section 3: when we deploy an existing monitoring mechanism (*e.g.*, Elastic-Sketch) on just one *single* egress pipe (which uses larger per-slice caches) or across *all* of them (which uses smaller per-slice caches).

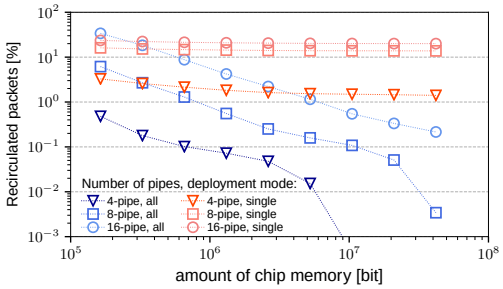

Fig. 32. All-egress-pipes vs. single-egress-pipe.

Figure 32 shows the percentage of recirculated packets (y-axis) over the amount of utilized memory for both single-egress-pipe (reddish lines) and all-egress-pipes (bluish lines) cache deployments and for different number of pipes, *i.e.*, with 4 (triangle marks), 8 (square marks), and 16 (circle marks) pipes. We set the memory cache overhead to 20% and use 4 slices. Our results show that the percentage of recirculated packets for all single-egress-pipe cache deployments remains stable even when additional memory is used. The reason lies in the calculations of Section 3: not enough packets to drain the cache. Conversely, all-egress-pipe reduces the percentage of recirculated packets when increasing the memory utilization.

F BACKGROUND ON HEAVY HITTERS AND SUPER SPREADERS

We provide some minimal background on heavy hitters and super spreaders.

Heavy hitters. A heavy hitter in a networking context is a traffic class that contributes to a large fraction of the forwarded traffic. There exist a variety of definitions for heavy hitters. One may define a heavy hitter based on a static threshold that defines the minimum number of packets to characterize a heavy hitter. Alternatively, a heavy hitter may be defined as a fraction of a traffic or as the top k traffic classes in a traffic trace. Most existing works on ASIC switches rely on thresholds as these are simpler to realize in hardware [42, 63]. In this work, we also consider a threshold based approach. Analyzing the memory benefits of PipeCache with other heavy hitter definitions is left as future work.

Super spreaders. While a heavy hitter is a traffic class that generates a large fraction of traffic, a super spreader is a traffic class that either generates traffic towards many destinations or receives traffic from many sources (as in a DDoS attack [18]). The common way to detect super spreaders is to rely on approximate data structures for counting distinct elements such as HyperLogLog [24, 66].

Received August 2022; revised October 2022; accepted January 2023