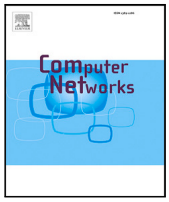
Computer Networks 254 (2024) 110764

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



MpScope: Enabling multi-pipeline monitoring inside a switch

Chengyuan Huang [a], Tianfan Zhang [a], Li Wang [a], Yibo Xiao [a], Chao Yang [a], Chen Tian [a], Xiaoliang Wang [a,*], Dong Zhang [b], Bingheng Yan [b], Ahmed M. Abdelmoniem [c], Wanchun Dou [a], Guihai Chen [a]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, China
[b] Jinan Inspur Data Co., Ltd., China
[c] School of Electronics Engineering and Computer Science, Queen Mary University of London, UK

ARTICLE INFO

Keywords:
Programmable switch
Network monitoring
Multiple pipelines

ABSTRACT

The core of programmable switches is the flexible data plane, composed of multiple programmable pipelines in existing programmable switches. These pipelines are isolated from each other and cannot share state and data. However, most of network monitoring systems ignore this condition and implicitly assume that the switch has only a single pipeline. This results in an inaccurate measurement and high communication overhead with the practical switch. To tackle this problem, we propose *MpScope*, a general multi-pipeline monitoring framework, which centers around the control plane, supporting accurate and efficient network monitoring. Specifically, *MpScope* combines the switch's data plane and control plane to achieve comprehensive network monitoring of the whole switch scope. The control plane aggregates the statistical results from multiple pipelines and tunes the monitoring module residing in the different pipelines in the data plane dynamically. The data plane is responsible for real-time traffic measurement and statistic reports. Its behaviors can be adjusted periodically with the instructions from the control plane. Two typical monitoring applications, *i.e.*, heavy hitter detection and distinct counting, are developed with *MpScope* to validate the effectiveness of multi-pipeline monitoring. Experiments show that *MpScope* significantly reduces communication overhead compared to the static threshold scheme while maintaining high detection accuracy over time.

1. Introduction

With the explosive growth of network traffic, problems such as configuration errors, malicious attacks, hardware failures, *etc.*, are inevitable in increasingly complex networks, which will affect the security and stability of the network. Network monitoring can help engineers understand the behavior and state of the network, which is essential for the successful operation of the network. It efficiently collects information from the data plane to detect network bottlenecks and abnormal behaviors, supporting network management.

In network measurement, the period of information collection is called an epoch. Usually, a switch collects data within an epoch, sends the statistics to the server, and then clears the records and starts the next epoch measurement. Common tasks of network monitoring include heavy hitter [1], SuperSpreader [2], and DDoS [3] detection. Heavy hitter refers to a flow whose size exceeds a certain threshold in a measurement epoch; SuperSpreader refers to a source host that sends data to more than a threshold number of destination hosts in an epoch; DDoS is when a destination host is under DDoS attack, *i.e.*, it receives messages from more than a threshold number of source hosts in an epoch. SuperSpreader and DDoS are similar, they both belong to the distinct counting problem, *i.e.*, a flow should be counted only once. Moreover, there are many other metrics [4] for network monitoring, such as Top-K, Hierarchical heavy hitter, heavy changer and flow cardinality statistics, *etc.*

Traditional network monitoring tools and protocols, such as SNMP [5], NetFlow [6], and sFlow [7], collect either sampled or coarse-grained statistical data, and cannot provide sufficiently accurate information for network troubleshooting. Moreover, sampling-based measurement systems [6,7] cannot monitor every packet in the network. Programmable switches bring new opportunities for network monitoring. The programmability of the data plane allows monitoring modules to be deployed directly on switches. Unlike sampling-based measurements, the modules deployed on switches can count all packets entering the switch, achieving fine-grained network measurement [4]. As a result, many P4-based network monitoring systems and measurement algorithms have emerged, *e.g.*, a large number of sketch-based approaches [8–14] are proposed to achieve fast and accurate network monitoring.

* Corresponding author.
E-mail address: waxili@nju.edu.cn (X. Wang).

https://doi.org/10.1016/j.comnet.2024.110764
Received 26 January 2024; Received in revised form 29 August 2024; Accepted 31 August 2024
Available online 3 September 2024

The existing programmable switches are composed of multiple programmable pipelines, which process network packets in parallel to obtain an extremely high processing rate. For instance, the Intel Tofino 2 chip can support up to 12.8 Tbps of non-blocking throughput and up to 6 billion pps of forwarding capacity. Currently, these pipelines are isolated and cannot share state and data. The configuration and action on one pipeline cannot interfere with others, which provides simplicity and security. Hence, network engineers should be aware of the limitations caused by the internal architecture of multiple pipelines when designing and implementing P4 programs.

However, most of current network monitoring systems make an implicit assumption that the programmable switch has only a single pipeline for simplicity [8–13,15–20]. When a monitoring job is launched on a switch, it implements only the monitoring module in a pipeline and neglects the others. Consequently, the traffic of interest passing through the pipelines, not deployed with a monitoring module, is not counted. As a result, if the existing monitoring approaches are leveraged directly, they cannot accurately measure the real situation of the network with the multi-pipeline condition. For example, a heavy hitter on the entire switch may not be a heavy hitter on any pipeline, and existing systems will miss these heavy hitters. Recently, several related works have begun to focus on conducting network monitoring across multiple pipelines, e.g., PipeCache [21]. It aggregates the flow statistics at the egress of a specific pipeline (monitoring pipe) and conveys the necessary monitoring information to the monitoring pipe. Specifically, it temporarily caches the metadata of packets that are not routed to the monitoring pipe at the ingress, then piggybacks the corresponding cached metadata onto the packets being forwarded to the monitoring pipe. However, PipeCache consumes additional data plane resources that are scarce in the programmable switch and can lead to significant monitoring errors.

To overcome the practical issue of multi-pipeline monitoring, we propose *MpScope*, a general framework, which centers around the control plane, supporting multiple network monitoring applications. Inspired by the network-wide distributed network monitoring work [22], *MpScope* combines the data plane and control plane inside a switch to achieve network monitoring under multiple pipelines, thereby solving serious measurement limitations. The key idea of *MpScope* is to implement the monitoring modules on all pipelines and coordinate the behaviors of pipelines to complete the switch-wide monitoring job. Specifically, the pipelines in the data plane measure the traffic passing through and report the results to the control plane periodically. Meanwhile, the control plane has a global view of the multiple pipelines and dynamically adjusts the measurement behaviors based on the data plane reports and control plane applications. Meanwhile, unlike existing multi-pipeline monitoring work that focuses on the data plane, *MpScope* does not require additional data structures in the data plane. This approach maintains the simplicity of the data plane while achieving good performance with limited data plane resources.

Moreover, based on the framework of *MpScope*, we design the corresponding solutions to improve performance for two specific monitoring applications, *i.e.*, heavy hitter detection and distinct counting. To detect heavy hitters, *MpScope* assigns the global threshold of the entire switch to each pipeline in the data plane, which is responsible for counting and reporting local heavy hitters. The control program in the control plane aggregates the reported information to obtain the heavy hitters of the entire switch. It dynamically adjusts the pipeline's local threshold according to historical information to reduce communication overhead. Similarly, *MpScope* follows the above idea with the BeauCoup algorithm [15] to detect the global SuperSpreader. Experiments show that the communication overhead of *MpScope* with the dynamic threshold (*MpScope dynamic*) is several orders of magnitude and up to 37.6% lower than that of existing network-wide distributed work [22] (`distributed`) and *MpScope* with the static threshold (*MpScope static*). Moreover, in the heavy hitter detection and distinct counting, *MpScope* achieves zero false negatives while introducing an acceptable number of false positives.

In summary, the contributions of this paper are as follows:

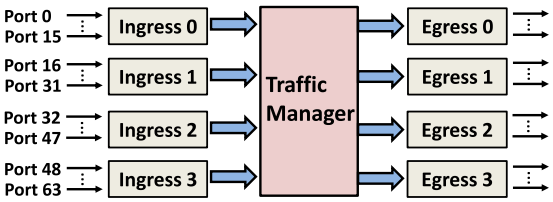


Fig. 1. The internal architecture of commodity programmable switch.

- We design *MpScope*, a general framework, which centers around the control plane, to enable accurate multi-pipeline monitoring in the commodity programmable switch.
- Based on the *MpScope* framework, we modify and propose two specific network monitoring solutions, *i.e.*, heavy hitter detection and distinct counting, to adapt to the practical multi-pipeline architecture.
- We conduct extensive experiments to show that *MpScope dynamic* can reduce the communication overhead significantly compared to `distributed` and *MpScope static*, while maintaining high detection accuracy over time.

2. Background and motivation

In this section, we show the commodity switch architecture in Section 2.1, introduce the existing network-wide and multi-pipeline network monitoring approach in Section 2.2, and describe the motivation of *MpScope* in Section 2.3.

2.1. Internal switch architecture

The core of the programmable switch is the programmable pipeline, which can process packets at line rate. Engineers can use the P4 [23] programming language to customize the programs on the pipeline, offloading some functions that used to be run by the CPU to hardware, achieving huge performance advantages.

The existing programmable switches adopt an internal architecture with multiple parallel packet processing pipelines to further improve their processing rate. For example, the Tofino [24] switch comprises 4 pipelines, and its total packet processing rate can reach 6.4 Tbps. As shown in Fig. 1, the internal architecture of each switch pipeline is divided into Ingress and Egress: Ingress connects to the switch's input port, and Egress connects to the switch's output port, and the Traffic Manager links them.

The mapping relationship between ports and pipelines is fixed. For example, ports 0 to 15 belong to pipeline 0, ports 16 to 31 belong to pipeline 1, and so on. The multiple pipelines in the current switch architecture are independent, they cannot share state and data, *i.e.*, when a pipeline processes a packet, it cannot access resources on other pipelines, such as values stored in the registers or counters. MP5 [25] also pointed out this problem and proposed a novel stateful multi-pipeline programmable switch architecture to achieve state sharing between different pipelines. However, this architecture is currently only a prototype and no commercial switch supports it. Therefore, engineers still have to handle the limitations of the current switch architecture, *i.e.*, the strict resource isolation between multiple pipelines.

2.2. Distributed network monitoring

Distributed network monitoring involves multiple nodes that conduct network monitoring locally and a central coordinator that collects the local statistics from distributed nodes to complete the global network monitoring task. According to the coordinator's scope, we can categorize the existing distributed network monitoring works into two

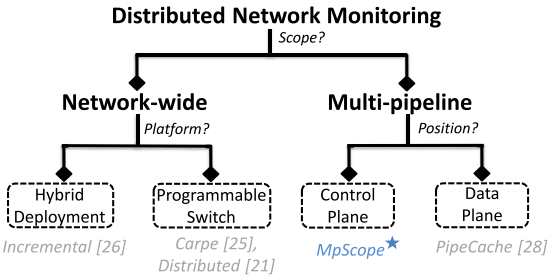


Fig. 2. Taxonomy of distributed network monitoring systems.

kinds, *i.e.*, network-wide and multi-pipeline network monitoring, as illustrated in Fig. 2.

Network-wide Monitoring. One line of distributed network monitoring aims to monitor targeted traffic on a network-wide basis, *i.e.*, identifying the traffic of interest with the statistics collected from different switches in a network. Rob Harrison et al. [22] studies the heavy hitter detection with multiple switches in the whole network and reduces the communication overhead between multiple switches and the central coordinator while ensuring accuracy. The proposed network-wide algorithm counts the traffic entering the network at each edge switch and applies local, per-key thresholds to trigger reports to a central coordinator. The coordinator adapts these thresholds to the prevailing traffic to reduce the total number of reports. As a follow-up work, Carpe [26] further overcomes the linearly increasing communication overhead with the number of monitored switches by probabilistically identifying and reporting potentially important flows. It benefits from the probabilistic reporting, limiting the required memory on the switches while introducing the bounded errors. In contrast, Damu Ding et al. [27] focuses on the hybrid deployment of fixed-function and programmable switches. They propose an incremental algorithm capable of detecting network-wide heavy hitters using only partial information from the data plane as input.

Multi-Pipeline Monitoring. The other line of distributed network monitoring focuses on orchestrating multiple monitoring functions on different switch pipelines to complete a network monitoring task. Xin et al. [28] first points out the fundamental constraints in designing stateful applications on multi-pipeline commodity programmable switches and proposes several high-level ideas to make any in-network application feasible to run on multi-pipeline switches. PipeCache [21] and its preceding work [29] share a similar idea, *i.e.*, storing all the monitoring information of each monitored traffic class into exactly one specific pipeline, instead of replicating the information on multiple pipelines. Specifically, PipeCache briefly stores monitoring information into a per-pipeline ingress cache and then piggybacks this information onto existing data packets to the correct egress pipe. Therefore, all the monitoring statistics are updated completely in the data plane, can be reflected correctly on one pipeline, and naturally, avoid the inaccuracy caused by the local scope of different pipelines. However, PipeCache's additional cache requires additional memory and logic, which results in higher usage of resources such as SRAM, TCAM, VLIW Instruction, Exact Match X-Bar, and TCAM Match X-Bar.

2.3. Motivation

On the one hand, we observe that most of the current network monitoring systems, such as Marple [18], Sonata [16], and PacketScope [17], and network measurement algorithms, such as various Sketches [10–14] and BeauCoup [15], assume that programmable switches have only one pipeline and do not consider the actual architecture of the switches. Therefore, when applied to existing programmable switches, they cannot accurately measure the corresponding metrics and thus do not accurately reflect the real network situation. For example, Sonata [16] is a monitoring system that supports various query requests. Suppose we use Sonata to detect which flows are heavy hitters. Each flow uses source IP and destination IP as the flow key. Sonata extracts the flow key of each packet in the switch data plane and performs hash operations on it. Later, it uses the hash value as the index of the register, accumulating the count in the corresponding register, thereby counting the flow size. Once the count corresponding to a certain flow reaches the threshold, the data plane will report the flow. Because switches have multiple pipelines and load-balancing network schemes such as Equal Cost Multi-Path (ECMP) are widely used, traffic with the same pair of source and destination IPs may enter different pipelines in the switch. This means that a heavy hitter at the global level of the switch may not be a heavy hitter in any pipeline. Therefore, Sonata may miss some heavy hitters. For example, a heavy-hitter flow is missed if the threshold is set to 1000 and the flow hits 800, 100, 100, and 100 times in four pipelines, respectively. In this case, the sum of the four pipelines exceeds the threshold, but Sonata does not report the flow because none of the records in each pipeline is greater than the global threshold. The same missed reports also occur in distinct counting problems such as DDoS and SuperSpreader detection.

On the other hand, as mentioned above, the recently proposed network monitoring work targeting the multi-pipeline architecture, PipeCache, introduces additional data structure and resource consumption in the data plane. The resource consumption linearly grows with the number of pipelines and is impacted by the traffic distribution. That is, PipeCache deploys a cache on each ingress pipe, and the cache may be overwhelmed if the traffic distribution is highly skewed and the majority of traffic is not routed to the egress pipe that stores the flow statistics. This would exacerbate the scarcity of resources in the data plane, especially when the data plane needs to support other network functions simultaneously, *e.g.*, normal routing/forwarding, IP address translation, and stateful load balancer. To mitigate this problem, PipeCache manually generates the cloned packets, piggybacking the cache to their corresponding monitoring pipelines. However, this may cause additional packet processing overhead and consume more bandwidth of the internal recirculation port.

The serious monitoring inaccuracy with the existing single-pipeline monitoring and the additional data plane overhead introduced by the existing multi-pipeline monitoring motivate us to design a method that achieves high accuracy while avoiding additional resource consumption in the data plane. To this end, we place the complexity on the control plane and keep the data plane the same as the single-pipeline architecture without introducing additional storage and computation. Specifically, we leverage the similarity between the network-wide and multi-pipeline network monitoring, *i.e.*, each pipeline is analogous to a switch in the network-wide monitoring model and the switch control plane is analogous to the global coordinator in the network. Therefore, inspired by the work of Rob Harrison et al. [22] mentioned in Section 2.2, which concentrates on the functions of the global coordinator, we can follow the same philosophy and design a framework that adjusts the separated multiple pipelines dynamically inside a switch, in the switch control plane, to implement the monitoring job. Besides, it should be able to support various monitoring applications, *e.g.*, heavy hitter detection and distinct counting, to accomplish the flexibility for multiple objectives.

3. MpScope design

In this section, we first give an overview of *MpScope* in Section 3.1 and introduce the basic workflow and framework of it. Later, based on the framework, we modify and redesign two typical network monitoring applications, *i.e.*, heavy hitter detection and distinct counting, with multiple pipelines in Sections 3.2 and 3.3, respectively.

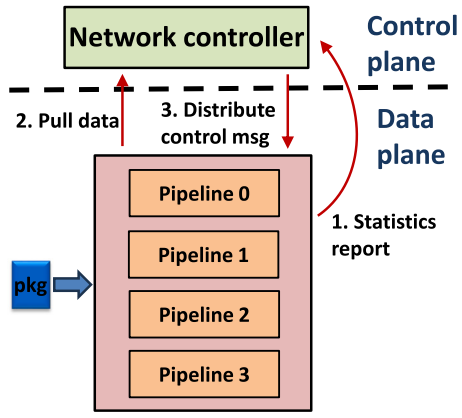


Fig. 3. The design overview of *MpScope*.

3.1. Overview

To fully utilize the line-rate processing capability of programmable switches, existing works usually offload the measurement tasks to the data plane, where the data plane independently completes the statistics and then reports the results (such as detected heavy hitters) to the control plane. However, due to the switches' multi-pipeline architecture, each pipeline in the data plane has only a local view and cannot complete global statistics. Instead, the control plane has a global view of the entire switch. Hence, an intuitive method is to copy every packet passing through the pipeline to the control plane, and the control plane program completely detects the target flows. However, the overhead is unacceptable because the bandwidth between the data and control planes is limited. The control plane comprises general-purpose CPUs and does not have line-rate processing capability, making it difficult to process Tbps-level traffic. Therefore, we propose a communication-efficient monitoring approach to eliminate the per-packet report from the data plane to the control plane and minimize the communication overhead. Moreover, the proposed approach is also control plane-centric to avoid introducing additional resource consumption in the data plane. *Specifically, the main idea of this method is that the data plane updates the statistics of each flow and reports the potential target statistics to the control plane, independently. The control plane aims to summarize and analyze the traffic statistics. It then sends the analysis results to the data plane to guide the data plane in achieving flexible and accurate monitoring.*

Fig. 3 shows the overview of the proposed approach, *MpScope*. The basic workflow is as follows: (1). When a data packet enters a pipeline in the data plane, the pipeline program determines whether it is a potential target (such as a heavy hitter or a SuperSpreader). If so, the pipeline reports the packet to the control plane. (2). At the end of each epoch, the controller in the control plane pulls and merges potential target statistics from all pipelines in the data plane and verifies whether the flows reported by the data plane are real targets. (3). After processing the data collected in this epoch based on the control logic of the monitoring application, the controller sends new control instructions to the pipelines in the data plane to conduct multiple pipeline cooperation. This collaborative method efficiently monitors the network by combining the data plane's line-rate processing capability with the control plane's global view.

Later, the following sections will be based on the general framework of *MpScope*, and use the heavy hitter detection and distinct counting under the multi-pipeline architecture as specific network monitoring application cases, and propose corresponding designs in Section 3.2 and Section 3.3, respectively.

Algorithm 1: Heavy Hitter Detection in the Data Plane

Input: Switch Setting $\langle pipeID, pipeThreshold\rangle$, Packet Key $\langle pkg\rangle$
Output: The marked packet is reported to the control plane

Function `ProcessPacket(`pkg`)`:
  $fkey \leftarrow \langle pkt.srcIP, pkt.dstIP\rangle$
  $hashValue \leftarrow HASH\langle fkey\rangle$
  $register[hashValue] \leftarrow register[hashValue] + 1$
  $count \leftarrow register[hashValue]$
  if $count = pipeThreshold[pipeID][fkey]$ **then**
    report this packet to the control plane
  end
End Function

Algorithm 2: Heavy Hitter Detection in the Control Plane

Input: Pipeline ID $\langle pipe0, ..., pipe(N-1)\rangle$, Switch Threshold $\langle switchThreshold\rangle$
Output: Heavy Hitter Set $\langle HeavyHitterSet\rangle$, Pipeline Threshold $\langle pipeThreshold\rangle$

Function `HandleReport(`pkg`)`:
  $possibleHeavyHitter$.insert($pkt.fkey$)
End Function
Function `GetHH(`$possibleHeavyHitter$`)`:
  foreach $fkey$ *in* $possibleHeavyHitter$ **do**
    $globalSize \leftarrow 0$
    for $pipeID \leftarrow 0$ *to* n **do**
      $globalSize \leftarrow globalSize + localSize[pipeID][fkey]$
    end
    if $globalSize \geq switchThreshold$ **then**
      heavyHitterSet.insert($fkey$)
    else
      `ResetTH(`$fkey, globalSize, localSize$`)`
    end
  end
  return *heavyHitterSet*
End Function
Function `ResetTH(`$fkey, globalSize, localSize$`)`:
  $EWMAglobalSize[fkey] \leftarrow (1-\alpha) \times EWMAglobalSize[fkey] + \alpha \times globalSize$
  for $pipeID \leftarrow 0$ *to* n **do**
    $EWMAlocalSize[pipeID][fkey] \leftarrow (1-\alpha) \times EWMAlocalSize[pipeID][fkey] + \alpha \times localSize[pipeID][fkey]$
    $frac \leftarrow EWMAlocalSize[pipeID][fkey] \div EWMAglobalSize[fkey]$
    $pipeThreshold[pipeID][fkey] \leftarrow frac \times switchThreshold$
  end
End Function

3.2. Heavy hitter detection

MpScope combines the switch's data and control planes to detect heavy hitters. Specifically, it distinguishes the heavy hitter threshold into the threshold on the entire switch and the threshold on a single pipeline, the former denoted as *switchThreshold* and the latter denoted as *pipeThreshold*. We aim to detect all heavy hitters that pass through the switch, for which we need to allocate *switchThreshold* to *pipeThreshold*. We can prove that as long as *switchThreshold* $= \Sigma$ *pipeThreshold*, then any flow that is a heavy hitter at the switch is a heavy hitter on at least one pipeline, which is presented in the proof of Theorem 1. Therefore, as long as the data plane's pipelines report their local heavy hitters, and the control plane decides on the global view, it can detect all heavy hitters at the switch level. In the first epoch

of network monitoring, we evenly allocate the thresholds of different pipelines, *i.e.*, $pipeThreshold = switchThreshold/n$, where n is the number of pipelines. Over time, the controller will dynamically adjust the threshold according to the traffic distribution of each pipeline.

Algorithm 1 shows the processing procedure in the data plane. Its input is the pipeline ID *pipeID*, the packet pkt that flows through the pipeline, and the local threshold *pipeThreshold* set by the controller. When a packet enters the data plane pipeline, the data plane extracts the flow key from the packet header and puts it in $fkey$. Then, it calculates the hash value based on $fkey$, which is used as the register's index to count the flow size. If the flow size is equal to the local threshold set on the pipeline, then the flow is a potential heavy hitter and is reported to the control plane. As a result, each flow will be reported at most once on a single pipeline in each epoch, greatly reducing the communication overhead between the data plane and the control plane.

Algorithm 2 shows the processing phases in the control plane. Its inputs are the pipeline IDs of the switch, the global threshold *switchThreshold*, and the outputs include the set of heavy hitters of the switch in this epoch *HeavyHitterSet* and the pipeline threshold *pipeThreshold*. During each epoch, if the control plane receives a report message from the pipeline (line 1–3), it will add the reported $fkey$ to the *possibleHeavyHitter* set. Multiple pipelines may report a flow, so this step also achieves deduplication.

At the end of each epoch, the controller runs the `GetHH` function (line 4–17) to count the global heavy hitters. The controller first pulls the counts from each pipeline and then clears the counters in the data plane. Next, the controller iterates through the *possibleHeavyHitter* set and merges the data from each pipeline to get the global count *globalSize*. If the overall count is greater than the global threshold *switchThreshold*, then the flow is a heavy hitter on the switch, and the controller adds it to the *HeavyHitterSet*. Otherwise, this is a false positive, and the controller adjusts the threshold for this flow (line 13).

`ResetTH` function (line 18–25) describes the process of the controller adjusting the thresholds on each pipeline based on the information collected from current and previous epochs, predicting the traffic distribution in the next epoch. The function uses the exponentially weighted moving average (EWMA) to synthesize information from different epochs, and the weight is set to account for data closer to the current epoch as more important. The algorithm first calculates the flow counts in the entire switch after applying the weighted average *EWMAglobalSize[fkey]* (line 19), then calculates the flow counts in each pipeline after applying the weighted average *EWMAlocalSize[pipeID][fkey]* (line 21). Later, it divides the two values to get the traffic distribution of the pipeline in the switch globally (line 22). Then, according to this ratio, *switchThreshold* is allocated to the pipeline's local threshold *pipeThreshold* (line 23). The adjusted threshold can adapt to the real-time traffic dynamics, reducing the number of messages reported by the data plane and the size of the *possibleHeavyHitter* set. For example, if the switch threshold is 1000, each pipeline threshold is 250, and a flow has a stable size on each pipeline of (300, 150, 150, 0), this flow will be accidentally reported by one of the pipelines. After adjusting by `ResetTH`, the pipeline threshold can be changed to (500, 250, 250, 0), thus avoiding the reporting of this flow.

Finally, we prove that Algorithm 1 and Algorithm 2 achieve 100% detection accuracy and can capture every heavy hitter across different pipelines.

Theorem 1. *Any global heavy hitter whose aggregated statistics exceed the global threshold will be reported to the control plane by at least one of its pipelines.*

Proof. Assume there are n pipelines in a switch, with the local threshold for Pipeline i denoted as T_i, and the global threshold for the switch as T. For a global heavy hitter H (where $H \geq T$), which is the sum of multiple local statistics H_i from different pipelines, one of its local statistics must be greater than the local threshold, *i.e.*, $\exists i, H_i \geq T_i$. We prove Theorem 1 by contradiction as follows. Assume a global heavy hitter H is not reported by any pipeline, *i.e.*, $\forall i, H_i < T_i$. Then, $H = \sum_{i=1}^{n} H_i < \sum_{i=1}^{n} T_i = T$, which contradicts the nature of the global heavy hitter, *i.e.*, $H \geq T$. Therefore, Algorithm 1 and Algorithm 2 ensure that the global heavy hitter must be reported by at least one of the pipelines, capturing every heavy hitter in any case. □

Table 1
The common distinct queries

Name	Key	Attribute
SuperSpreader	srcIP	dstIP
DDoS Victim	dstIP	srcIP
TCP Port Scan	{srcIP, dstIP}	dstPort

Algorithm 3: Distinct Counting in the Data Plane

Input: Switch Setting ⟨*pipeID, pipeCouponThreshold*⟩, Packet Key ⟨pkg⟩
Output: The marked packet is reported to the control plane

```
1 Function ProcessPacket(pkg):
2     fkey ← ⟨pkt.srcIP⟩
3     attr ← ⟨pkt.dstIP⟩
4     coupons[pipeID][fkey] ← BeauCoup(fkey, attr)
5     count ← Σ coupons[pipeID][fkey]
6     if count = pipeCouponThreshold[pipeID][fkey]  ς pkt generates
        new coupon then
7         report this packet to the control plane
8     end
9 End Function
```

3.3. Distinct counting

Unlike the heavy hitter detection, in the distinct counting problem, packets of the same flow should only be counted once. Common distinct counting problems include flow cardinality, DDoS, and SuperSpreader. Although Sonata [16] uses a counter-based method to handle distinct counting on a single pipeline, it cannot solve the problem under multiple pipelines because the packets of the flows of interest may appear in multiple pipelines. Suppose there are three flows: flow1, flow2, and flow3. Flow1 and flow2 pass through pipeline 1, and flow2 and flow3 pass through pipeline 2. Sonata will detect two flows in pipeline 1 and pipeline 2, and add them up to get a total of 4 flows, which violates the ground truth of 3 flows.

In this work, we leverage BeauCoup [15], a probability-based network measurement algorithm, to solve the distinct counting problem on multiple pipelines. It is inspired by the "coupon collection problem", which has been proven to successfully approximate the distinct counting of flows in the data plane. BeauCoup extracts two parts from the packets, a **Key** and an **Attribute**, and maintains a coupon bit array for each received Key. For example, in the SuperSpreader problem, the Key is the source IP, and the Attribute is the destination IP. When a packet enters the switch, BeauCoup locates the coupon bit array based on the packet's flow key. Then it maps the packet to an element of the array according to its destination IP, indicating that this packet will collect the coupon. Note that packets with different destination IPs may be mapped to the same coupon but are only counted once. By approximating the number of collected coupons, we can infer the number of destination IPs and detect SuperSpreaders. BeauCoup can also support different queries by modifying the Key and Attribute. We present some common queries in Table 1. BeauCoup has similarities with HyperLogLog [30] or Bloom Filter [31] in design, but the difference is that BeauCoup can support multiple query requests within a limited number of memory operations. Since programmable

Algorithm 4: Distinct Counting in the Control Plane

Input: Pipeline ID ⟨$pipe0, ..., pipe(N-1)$⟩, Switch Threshold ⟨*switchThreshold*⟩, Switch Coupon Threshold ⟨*switchCouponThreshold*⟩, Coupon Map ⟨*couponToNumber*⟩
Output: SuperSpreader Set ⟨*SuperSpreaderSet*⟩, Pipeline Coupon Threshold ⟨*pipeCouponThreshold*⟩

1 **Function** `HandleReport`(*pkg*):
2 *possibleSuperSpreader*.insert(*pkt.fkey*)
3 **End Function**
4 **Function** `GetSS`(*possibleSuperSpreader*):
5 **foreach** *fkey in possibleSuperSpreader* **do**
6 *globalCoupon*, *localSizeSum* ← 0
7 **for** *pipeID* ← 0 *to n* **do**
8 ▷ coupons are the bit arrays pulled from the data plane
9 *globalCoupon* ← *globalCoupon*|*coupons*[*pipeID*][*fkey*]
10 *localCount* ← Σ *coupons[pipeID][fkey]*
11 *localSize* ← *couponToNumber[localCount]*
12 *localSizeSum* ← *localSizeSum* + *localSize*
13 **end**
14 *globalCouponCount* ← Σ*globalCoupon*
15 **if** *globalCouponCount* ≥ *switchCouponThreshold* **then**
16 *SuperSpreaderSet*.insert(*fkey*)
17 **else**
18 `ResetTH`(*fkey*, *localSizeSum*, *coupons*)
19 **end**
20 **end**
21 **return** *SuperSpreaderSet*
22 **End Function**
23 **Function** `ResetTH`(*fkey, localSizeSum, coupons*):
24 *EWMAlocalSizeSum[fkey]* ← $(1-\alpha) \times$ *EWMAlocalSizeSum[fkey]* + $\alpha \times$ *localSizeSum*
25 **for** *pipeID* ← 0 *to n* **do**
26 *localCount* ← Σ*coupons*[*pipeID*][*fkey*]
27 *localSize* ← *couponToNumber[localCount]*
28 *EWMAlocalSize[pipeID][fkey]* ← $(1-\alpha) \times$ *EWMAlocalSize[pipeID][fkey]* + $\alpha \times$ *localSize*
29 *frac* ← *EWMAlocalSize[pipeID][fkey]* ÷ *EWMAlocalSizeSum[fkey]*
30 *pipeThreshold*[*pipeID*][*fkey*] ← *frac* × *switchThreshold*
31 *couponNumber* ← $\arg\max_k$ {*couponToNumber[k]* ≤ *pipeThreshold [pipeId][fkey]*}
32 *pipeCouponThreshold[pipeID][fkey]* ← *couponNumber*
33 **end**
34 **End Function**

switches are limited in memory operations, BeauCoup is the most suitable algorithm for the scenario considered in this work.

Thus, BeauCoup algorithm is deployed on different pipelines in the data plane, and the collected coupons are merged in the control plane to estimate the distinct value of the entire switch. Note that in this section, we take SuperSpreader detection as an example to introduce how to solve the distinct counting problem under a multi-pipeline architecture. Following the approach in Section 3.2, the controller dynamically adjusts the thresholds on each pipeline. In this case, the algorithm introduces additional thresholds for the number of coupons, namely *pipeCouponThreshold* and *switchCouponThreshold*. The BeauCoup algorithm estimates the number of distinct target IPs based on the collected number of coupons, *i.e.*, the sum of the coupon bit array. Similar to Theorem 1, we can prove that as long as *switchThreshold* = Σ *pipeThreshold*, then our proposed algorithms would not introduce more false negatives and achieve high accuracy, which is presented in the proof of Theorem 2.

Algorithm 3 presents the processing procedure in the data plane. In this case, the difference from Algorithm 1 is that the *pipeCouponThreshold* refers to the coupon threshold set by the controller. When a packet enters the pipeline, the data plane extracts information from the packet header: the key (*fkey*) and the attribute (*attr*) to be used by the BeauCoup algorithm. Taking SuperSpreader as an example, the source IP is extracted as *fkey*, and the destination IP is extracted as *attr*. Subsequently, based on *fkey* and *attr*, the BeauCoup algorithm generates the coupon bit array (*coupons*) for *fkey* (line 4). If the collected coupon count (*count*) reaches the set threshold (*pipeCouponThreshold*), the data plane reports it to the control plane (line 6–8). To reduce the number of reports, the data plane checks whether the packet has generated a new coupon, ensuring that a pipeline is only reported when the coupon count reaches the threshold for the first time.

Algorithm 4 describes the processing procedure in the control plane. The input includes pipeline IDs, the switch threshold *switchThreshold*, the switch coupon threshold *switchCouponThreshold*, and the map *couponToNumber* which keeps the relation between the number of coupons and the number of distinct flows. The outputs of Algorithm 4 include the *SuperSpreader* set and the pipeline coupon threshold *pipeCouponThreshold*. The `HandleReport` function handles the reports received in each epoch and adds *fkey* to the *possibleSuperSpreader* set. At the end of each epoch, the controller runs the `GetSS` function, which fetches and clears the data in all pipelines. Then, it traverses each *fkey* in the *possibleSuperSpreader* set, performs a bitwise OR operation on the coupon arrays of *fkey* on each pipeline, and obtains the sum of the overall distinct counts over the entire switch (line 9–12). Later, it decides whether the number of collected coupons reaches the coupon number threshold *switchCouponThreshold*. The *fkey* that exceeds *switchCouponThreshold* will be added to *SuperSpreaderSet* (line 16). Similar to the heavy hitter detection, if the report is a false positive, the controller also calls `ResetTH` to adjust the threshold allocation on different pipelines (line 18).

Algorithm 4 also uses EWMA to effectively utilize the historical information of previous epochs. The difference from Algorithm 2 is that `ResetTH` function needs to convert the coupon numbers to the targeted distinct counts according to the *couponToNumber* array generated by the BeauCoup algorithm (line 27, 31). For example, *couponToNumber* converts the local coupon count *localCount* of each pipeline to the number of targeted local distinct counts *localSize* (line 27). Next, the distribution ratio *frac* of the distinct counts is calculated using a weighted average (line 29). Finally, the coupon number is assigned (the key in *couponToNumber* map) corresponding to the maximum value in *couponToNumber* map that is less than or equal to *pipeThreshold[pipeID][fkey]*, to *couponNumber* (line 31–32).

At last, we prove that Algorithm 3 and Algorithm 4 introduce no more false negatives than the corresponding sing-pipeline algorithms and can capture every SuperSpreader across different pipelines.

Theorem 2. *Any global SuperSpreader whose aggregated statistics exceed the global threshold will be reported to the control plane by at least one of its pipelines.*

Proof. Assume there are n pipelines in a switch, with the global threshold of the switch denoted as T and the local threshold for Pipeline i denoted as T_i. Then, with Algorithm 3 and Algorithm 4, we have $T = \sum_{i=1}^{n} T_i$. Meanwhile, the local coupon array of Pipeline i at the end of an epoch is denoted as CA_i, and a coupon array can be mapped to the number of the distinct counters with the $F[\bullet]$ function. We prove Theorem 2 by contradiction as follows. Assume a global SuperSpreader S is not reported by any pipeline, then the local threshold T_i is not hit by any pipeline, *i.e.*, $\forall i, F[CA_i] < T_i$. Meanwhile, $S = F[(CA_1|CA_2|...|CA_i...|CA_n)] \le \sum_{i=1}^{n} F[CA_i] < \sum_{i=1}^{n} T_i = T$, *i.e.*, $S < T$, which contradicts the nature of the global SuperSpreader,

i.e., $S \geq T$. Therefore, Algorithm 3 and Algorithm 4 ensure that the global SuperSpreader must be reported by at least one of the pipelines, capturing every SuperSpreader in any case. □

4. Evaluation

In this section, we present the detailed evaluation setup in Section 4.1 and conduct extensive experiments to answer the following questions:

- **Does *MpScope* reduce system overhead compared to existing approaches?** We show that in heavy hitter detection, *MpScope* with the dynamic threshold reduces 22.1% ∼ 37.6% of the communication traffic compared to the static threshold approach (Section 4.2). Also, in SuperSpreader detection, *MpScope* with the dynamic threshold can save 19.3% ∼ 25.1% of the communication traffic (Section 4.3).
- **Does *MpScope* achieve high measurement accuracy in different applications?** We show that in heavy hitter detection, *MpScope* results in zero false negatives while introducing an acceptable number of false positives, with an average value of up to 377, under different settings (Section 4.2). Similarly, in SuperSpreader detection, *MpScope* results in zero false negatives while introducing an acceptable number of false positives, with an average value of up to 86, under different settings (Section 4.3).

4.1. Evaluation setup

We develop a packet-level simulator that replays the traffic trace and realizes the control plane and data plane as two separate modules in the simulator. The data plane is composed of 4 pipelines. When the simulation starts, a packet is mapped to one of the pipelines, and the monitoring function updates its statistics accordingly. If the statistics reach the local threshold, the control plane is called, simulating the communication between the control and data plane. Later, the control plane periodically calls the data plane and pulls and clears the recorded statistics. For PipeCache, we use the simulator provided by its authors, varying the configurations to show performance under different environments.

Traffic Distribution: Although some projects and organizations [32,33] have collected network traces in reality, they have not considered the distribution of traffic on a switch. This section refers to the distributed traffic distribution [22] to simulate network traffic distribution in reality on n different pipelines of a switch. Specifically, we replay the traces through one ingress pipeline and spread the traffic over the egress pipelines using an ECMP-like flow-consistent spreader. 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, which follows the same way as PipeCache. A traffic imbalance of x means that some pipes have a weight that is x times higher than the other pipes, i.e., they receive x times more traffic. For example, an imbalance of 1 means traffic is forwarded uniformly, and an imbalance of 31 means one pipeline receives 31 times less traffic than the other pipelines. In the simulation, we vary x from 1, 7 to 31 to show the performance of different schemes with diverse traffic distributions across pipelines.

Traffic Trace: The experiment uses the network trace collected by the WIDE MAWI [32] database in 2020 to evaluate the performance of different schemes. The MAWI database collects traces from real networks, helping researchers evaluate their traffic anomaly detection methods. We replay the trace at the rate of 1Mpkt/s and set the epoch length to 1 s.

The Compared Schemes: We evaluate the performance of 4 schemes, *i.e.*, the existing network-wide distributed monitoring approach (`Distributed`) [22], PipeCache [21], *MpScope* with the static threshold (*MpScope static*), and *MpScope* with the dynamic threshold (*MpScope dynamic*).

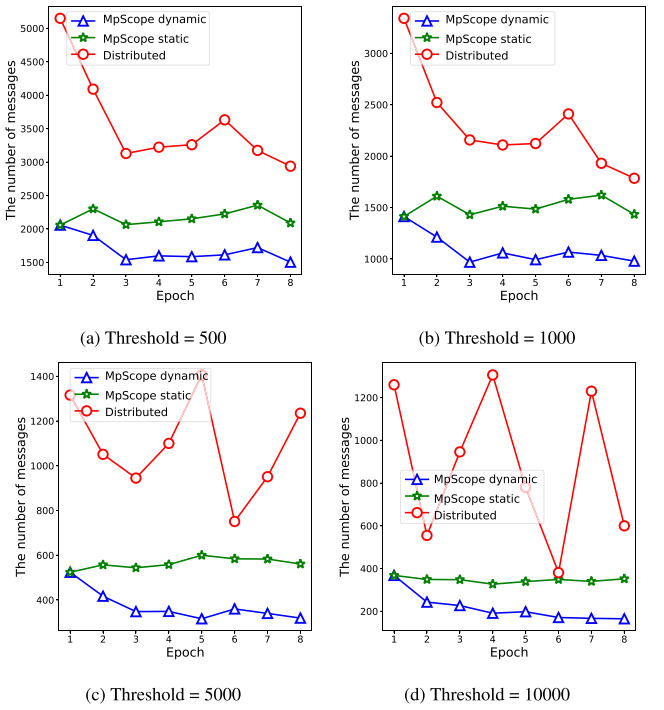


Fig. 4. Communication overhead of heavy hitter detection with different threshold settings.

Table 2
Communication overhead with different thresholds.

Threshold	Static threshold	Dynamic threshold	Reduction ratio (%)
500	17 354	13 525	22.1
1000	12 087	8728	27.8
5000	4511	2973	34.1
10000	2772	1730	37.6

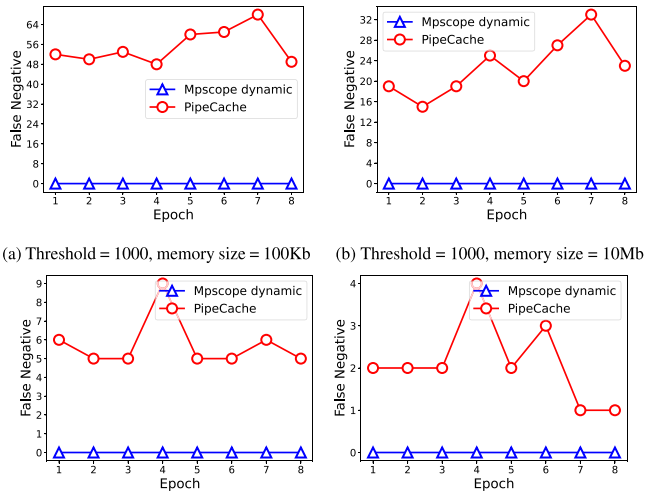
Performance Metrics: (1). We use the number of messages needed to be delivered between the control and the data plane as the metric to evaluate the communication overhead, *i.e.*, the number of calls between the control and data plane in our simulator. (2). We use false negatives (miss the flows of interests) and false positives (mistakenly report the non-interest flows) over time to quantify the detection accuracy of different schemes.

4.2. Heavy hitter detection

We show that *MpScope dynamic* significantly reduces communication overhead in Section 4.2.1 compared to the preliminary `Distributed` and static threshold approach. Then, in Section 4.2.2, we show that *MpScope* maintains no false negatives, at the cost of introducing additional false positives, in heavy hitter detection. Finally, the performance of different schemes under varied traffic distribution is exhibited in Section 4.2.3.

4.2.1. Communication overhead

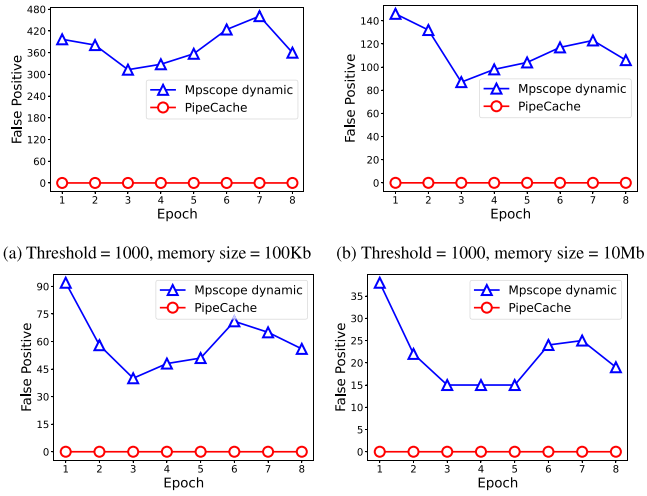
We measure the communication overhead by the total number of messages sent between the data and control plane, including the report counts and responses to pull requests. The key difference between the dynamic and static thresholds is that the static threshold scheme does not adjust the threshold based on historical flow distribution information. The static scheme consistently keeps the threshold on each pipeline as *switchThreshold*/n, which is the global threshold of the switch divided by the number of pipelines. Meanwhile, with `Distributed`, each pipeline continuously reports all counts



(a) Threshold = 1000, memory size = 100Kb (b) Threshold = 1000, memory size = 10Mb

(c) Threshold = 10000, memory size = 100Kb (d) Threshold = 10000, memory size = 10Mb

Fig. 5. The false negatives of heavy hitter detection.



(a) Threshold = 1000, memory size = 100Kb (b) Threshold = 1000, memory size = 10Mb

(c) Threshold = 10000, memory size = 100Kb (d) Threshold = 10000, memory size = 10Mb

Fig. 6. The false positives of heavy hitter detection.

that are greater than or equal to the local threshold to the control plane, leading to a significant increase in communication overhead. In contrast, with *MpScope*, each pipeline only reports once per period, specifically when the local count matches the local threshold. Similarly, with `Distributed`, the control plane pulls data from all pipelines multiple times within a period whenever it receives a report and its estimate of the aggregated count surpasses the global threshold. With *MpScope*, however, the control plane only pulls data that has been reported once per period. The primary difference between *MpScope* and `Distributed` comes from their different target environments. In the original `Distributed` environment, a centralized coordinator must monitor numerous observation nodes dispersed across the network. Each node needs to continuously report its current count to mitigate the risk of occasional report loss or delay. The coordinator also frequently adjusts the local threshold for each node to incur timely adjustments and avoid command loss or delay. However, the environment in this paper is confined to inside a switch, which is highly controlled with a limited number of pipelines (typically 2 to 8). This allows the control plane to quickly gather statistics within a period, enabling *MpScope* to monitor the pipelines more lazily. Specifically, each monitor reports at most once per period to notify the control plane, and the control plane subsequently queries once at the end of the period. This results in a significant reduction in communication overhead.

Fig. 4 compares the communication overhead of different schemes under various threshold settings. As shown in Fig. 4(a), for the threshold 500, we observe that the communication overhead with *MpScope* is significantly reduced, compared to the `Distributed` scheme. Moreover, we observe that the communication volume of the dynamic and static threshold schemes is the same in the first epoch. As the number of epochs increases, the advantage of the dynamic threshold scheme gradually becomes apparent, with its communication volume being much smaller than that of the static threshold scheme. This is because, in the first epoch, the thresholds of both schemes are the same. Over time, the dynamic threshold scheme effectively estimates traffic distribution using historical information. It improves data plane reporting accuracy by adjusting thresholds on different pipelines, thereby reducing the number of reported messages. Table 2 summarizes the communication overhead for all epochs under different threshold settings. It shows that the dynamic threshold scheme reduces communication volume by 22.1% to 37.6% compared to the static threshold scheme. In the following experiments, we compare the better *MpScope* with the dynamic threshold with the related work.

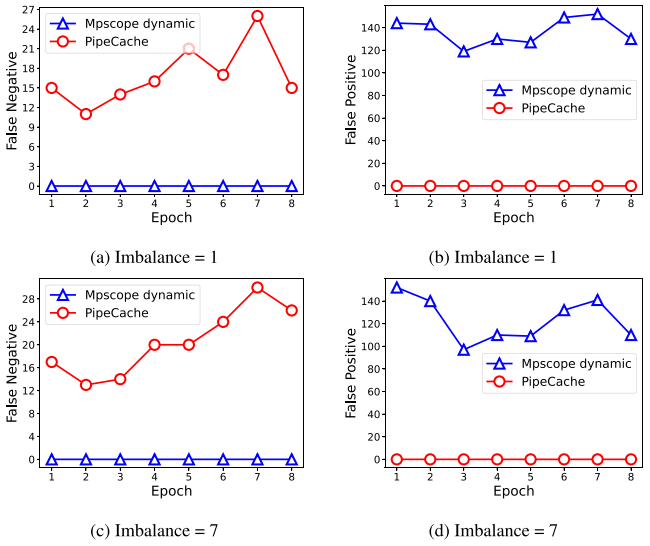
4.2.2. Detection accuracy

In this experiment, we show that *MpScope* can avoid false negatives (capture every heavy hitter) and cause a moderate number of false positives (additional communication overhead between the data and control plane).

Setup: We tune the available memory of the data plane from 100Kb to 10Mb to show the performance of *MpScope* and PipeCache with limited memory and abundant memory. Moreover, we vary the heavy hitter threshold from 1000 to 10,000 to reveal the performance difference between small and big thresholds. With threshold = 1000, the ground truth list of heavy hitters over time is [75,76,73,74,86,87,92,77]; With threshold = 10,000, the ground truth list of heavy hitters over time is [9,9,9,11,8,10,8,8]. The imbalance of the traffic distribution is 31 in this experiment. Note that we limit the ratio of manually generated recirculation packets with PipeCache, to 2%, as this moderate ratio avoids overwhelming the processing capacity of the switch chip.

Results: Fig. 5 shows the false negatives during heavy hitter detection with different schemes. We find that *MpScope* avoids false negatives (zero false negatives with different settings) and captures every heavy hitter successfully. However, PipeCache results in a great number of false negatives over time, *e.g.*, in Fig. 5(a), the average false negatives over time of PipeCache is 55 (the average ground truth over time is 80), leading to a 68.8% miss ratio. We attribute this to the limited memory of PipeCache, as the cache structure in every ingress consumes additional memory. When the available memory is so limited that the PipeCache cannot cache the necessary information in the ingress, the packets may go to the egress that is not the monitoring pipe and stores no statistics of these packets. This causes the miss of heavy hitter detection with PipeCache. On the contrary, *MpScope* requires no additional data structures in the data plane, and every egress can report the flow hitting the threshold.

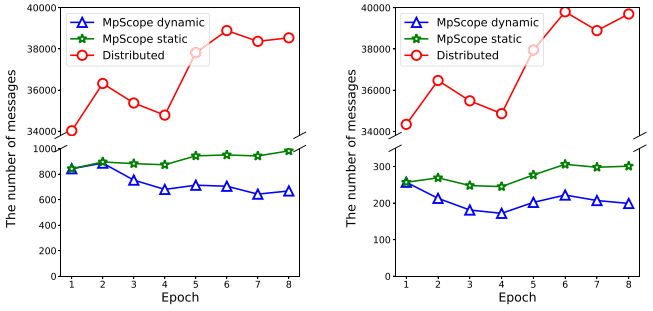
However, we observe that *MpScope* achieves zero false negatives at the cost of introducing some false positives. For example, as depicted in Fig. 6(a), *MpScope* can cause the average false positives of 377, 4.7 times the ground truth. On the contrary, PipeCache incurs no false positives with diverse settings. This is because PipeCache aggregates the flow statistics at the egress of the monitoring pipe, and every captured heavy hitter certainly hits the threshold. Instead, in *MpScope*, the statistics of a flow spreads over all pipelines, and the controller tunes the threshold dynamically based on its estimation. Thus, it is more sensitive and reports the possible heavy hitter aggressively. However, we found that even the largest false positive, *i.e.*, 461 per second in



(a) Imbalance = 1 (b) Imbalance = 1

(c) Imbalance = 7 (d) Imbalance = 7

Fig. 7. The performance of heavy hitter detection under varied traffic distribution.



(a) Threshold = 100 (b) Threshold = 200

Fig. 8. Communication overhead of distinct counting with different threshold settings.

Fig. 6(a), is still an acceptable traffic load for the control plane, and much smaller than the overall traffic volume, *i.e.*, 1 Mpkts per second.

4.2.3. Varied traffic distribution

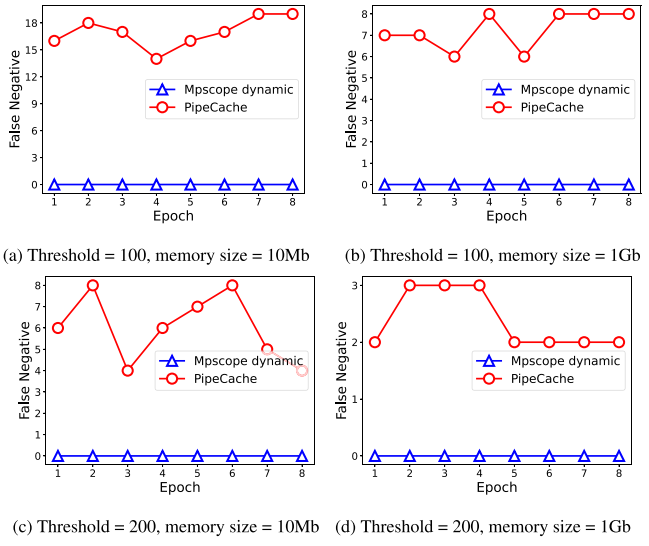
We show that *MpScope* can maintain accuracy with the diverse traffic distribution across the pipelines.

Setup: We change the traffic distribution across different pipelines (imbalance 1 and 7), to show the performance of different schemes with varied traffic distribution. The available memory in this experiment is set to 10Mb. The ratio of manually generated recirculation packets with PipeCache is set to 2%.

Results: By comparing Figs. 7(a) and 7(c), we found that the skewed traffic distribution incurs more false negatives with PipeCache, *i.e.*, 16 with imbalance = 1 versus 21 with imbalance = 7. This is because, in the worst case, a minority of traffic goes through the monitoring pipe and has fewer chances to piggyback information to the correct statistics, which causes more false negatives. Meanwhile, we found that the traffic distribution imbalance has a negligible impact on the performance of *MpScope*. We attribute this to the homogeneous characteristics of the pipeline, *i.e.*, every pipeline has the same structure used to store the statistics for the same flow.

4.3. SuperSpreader detection

We show that *MpScope dynamic* significantly reduces communication overhead in Section 4.3.1 compared to the preliminary `Distributed` and static threshold approach. Then, in Section 4.3.2, we show that *MpScope* maintains no false negatives, at the cost of introducing additional false positives, in SuperSpreader detection. Finally, the performance of different schemes under varied traffic distribution is exhibited in Section 4.3.3.



(a) Threshold = 100, memory size = 10Mb (b) Threshold = 100, memory size = 1Gb

(c) Threshold = 200, memory size = 10Mb (d) Threshold = 200, memory size = 1Gb

Fig. 9. The false negatives of SuperSpreader detection.

4.3.1. Communication overhead

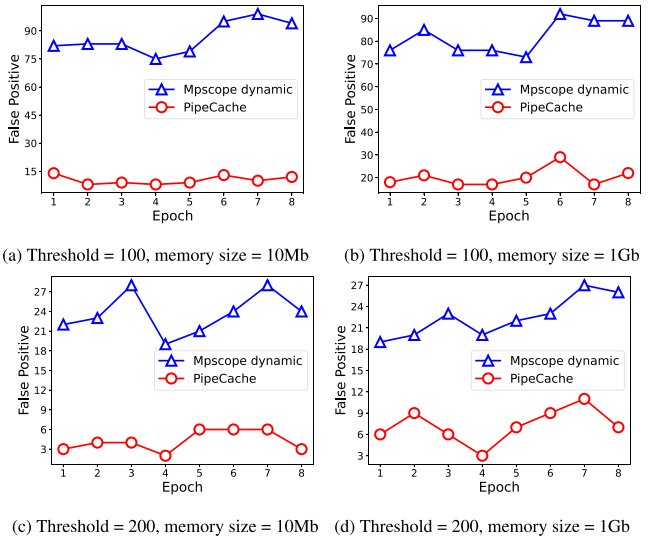
Fig. 8 shows the communication overhead of SuperSpreader detection with different threshold settings. Note that BeauCoup's distributed scheme involves each pipeline of the data plane running the algorithm independently with the same hash function. Whenever BeauCoup generates a new coupon, it reports to the control plane, and the control plane aggregates the coupons reported by each pipeline, thus obtaining the distinct count of the whole switch.

It is evident that *MpScope* has several orders of magnitude less communication overhead than the `distributed` scheme. This is because in *MpScope*, only when the coupon count reaches the preset pipeline threshold, the data plane will report the flow, while the `Distributed` scheme will report every new coupon, elaborated in Section 4.2.1. We also find that the communication overhead of the dynamic threshold scheme is lower than that of the static threshold scheme, *e.g.*, the communication overhead reductions with threshold = 100 and threshold = 200, are 19.3% and 25.1%, respectively. The reason is similar to that of the heavy hitter case explained earlier. In the following experiments, we compare the better *MpScope* with the dynamic threshold with the related work.

4.3.2. Detection accuracy

This experiment shows that *MpScope* can avoid false negatives (capture every SuperSpreader) and cause a moderate number of false positives (additional communication overhead between the data and control plane).

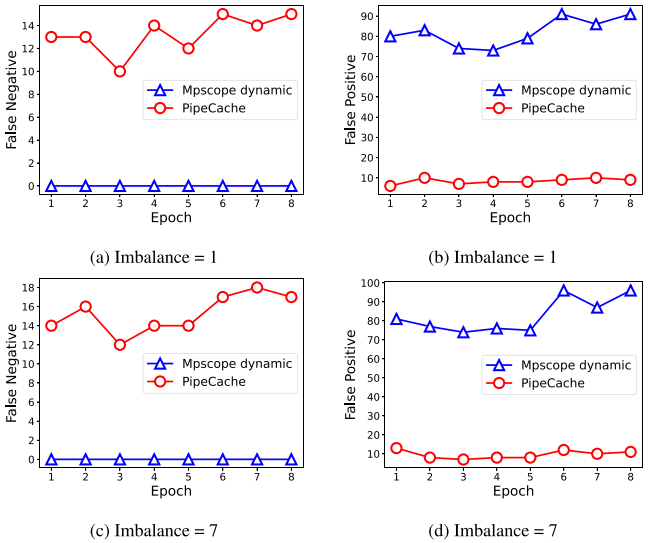
Setup: We tune the available memory of the data plane from 10 Mb to 1 Gb, to show the performance of *MpScope* and PipeCache with limited memory and abundant memory. Moreover, we vary the SuperSpreader threshold from 100 to 200 to reveal the performance difference between small and big thresholds. With threshold = 100, the ground truth list of SuperSpreaders over time is [29,31,32,29,32,32,34,32]; With threshold = 200, the ground truth list of SuperSpreaders over time is [15,16,14,16,16,16,15,14]. The imbalance of the traffic distribution is 31 in this experiment. Note that we limit the ratio of manually generated recirculation packets with PipeCache to 2%, as this moderate ratio avoids overwhelming the processing capacity of the switch chip.



(a) Threshold = 100, memory size = 10Mb (b) Threshold = 100, memory size = 1Gb

(c) Threshold = 200, memory size = 10Mb (d) Threshold = 200, memory size = 1Gb

Fig. 10. The false positives of SuperSpreader detection.



(a) Imbalance = 1 (b) Imbalance = 1

(c) Imbalance = 7 (d) Imbalance = 7

Fig. 11. The performance of SuperSpreader detection under varied traffic distribution.

Results: Fig. 9 shows the false negatives during SuperSpreader detection with different schemes. We find that *MpScope* avoids false negatives (zero false negatives with different settings) and captures every SuperSpreader successfully. However, PipeCache results in many false negatives over time, *e.g.*, in Fig. 9(a), the average false negatives over time of PipeCache is over 17 (the average ground truth over time is 31), leading to a 45.1% miss ratio. We attribute this to the memory sensitivity nature of PipeCache, similar to the reason explained in the heavy hitter detection.

However, we observe that *MpScope* achieves zero false negatives at the cost of introducing some false positives. For example, as depicted in Fig. 10(a), *MpScope* can cause the average false positives of 86, which is 2.8 times the ground truth. On the contrary, PipeCache incurs a few false positives with diverse settings. This is because PipeCache aggregates the flow statistics at the egress of the monitoring pipe, and every captured SuperSpreader certainly hits the threshold. Note that the non-zero false positives of PipeCache result from the inaccuracy of the probability-based BeauCoup algorithm. Instead, in *MpScope*, the statistics of a flow spreads over all pipelines, and the controller tunes the threshold dynamically based on its estimation. Thus, it is more sensitive and tends to report the possible SuperSpreader aggressively. However, we found that even the largest false positive, *i.e.*, 99 per second in Fig. 10(a), is still an acceptable traffic load for the control plane, and much smaller than the overall traffic volume, *i.e.*, 1 Mpkts per second.

4.3.3. Varied traffic distribution

We show that *MpScope* can maintain accuracy with the diverse traffic distribution across the pipelines.

Setup: We change the traffic distribution across different pipelines (imbalance 1 and 7), to show the performance of different schemes with varied traffic distribution. The available memory in this experiment is set to 1 Gb. The ratio of manually generated recirculation packets with PipeCache is set to 2%.

Results: By comparing Figs. 11(a) and 11(c), we found that the skewed traffic distribution incurs more false negatives with PipeCache compared to the balanced traffic distribution, *i.e.*, 13 with imbalance = 1 versus 15 with imbalance = 7. This is because, in the worst case, a minority of traffic goes through the monitoring pipe and has fewer chances to piggyback information to the correct statistics, which causes more false negatives. Meanwhile, by comparing Figs. 11(b) and 11(d), we found that the traffic distribution imbalance has negligible impact on the performance of *MpScope*. We attribute this to the homogeneous characteristics of the pipeline, *i.e.*, every pipeline has the same structure used to store the statistics for the same flow.

5. Related work

Most of the existing network monitoring approaches implicitly assume using a single pipeline, and we overview the most related ideas and can be enhanced by the idea in this paper.

Query-based Monitoring. Marple [18] is a novel monitoring system that supports flexible query jobs from network operators, utilizing the data flow primitives, *e.g.*, Map, Reduce, and Groupby. Similar to Marple, Sonata [16] provides the query interface, and it divides the measurement requirements and network traffic between the switch and flow processor, making them jointly undertake the monitoring tasks. PacketScope [17] is an extension of Sonata, which can monitor the internal situation of switches, such as packet header changes, internal queue packet loss, *etc.* OmniMon [19] is a network-wide flow-level telemetry architecture combining multiple network entities' capabilities (endpoints, switches, and controllers) to achieve high statistical accuracy and low resource overhead. Different from the aforementioned approaches that aim for a convenient query interface, *MpScope* focuses on a general framework to support multiple monitoring applications on multiple internal pipelines. *MpScope* is suited to be the monitoring backend and to be integrated into the existing query-based system.

Sketch-based Monitoring. UnivMon [12] is a Sketch-based flow monitoring framework that can support different monitoring tasks while aiming for generality and high accuracy. ElasticSketch [8] is an adaptive Sketch that can dynamically adjust according to different traffic characteristics. SpreadSketch [11] is a reversible Sketch data structure for detecting SuperSpreader across the network, which has theoretical guarantees on memory space, performance and accuracy. CocoSketch [9] is a Sketch-based measurement algorithm that supports arbitrary partial key queries. FlyMon [13] is the first Sketch measurement system that can achieve runtime dynamic reconfiguration of measurement tasks. SketchLib [10] is an efficient Sketch development library for programmable switch platforms. It systematically analyzes the resource bottlenecks in implementing various Sketch algorithms on programmable switches and proposes feasible optimization techniques for these bottlenecks. Generally, Sketch-based approaches count the traffic of all observed packets and perform lossy compression on the statistical data. They are orthogonal to *MpScope*, and designing a novel sketch supporting multiple pipeline monitoring is part of our future work.

6. Conclusion

Neglect of the internal switch architecture can cause serious overhead and inaccuracy in network monitoring. To overcome the limitation, we propose *MpScope*, a monitoring system that centers around the control plane to support multi-pipeline network monitoring inside a programmable switch. The measurement modules in the data plane report the measurement results to the control plane, periodically. The switch's control plane dynamically adjusts the measurement modules residing in different pipelines based on different monitoring applications. The experimental results show that *MpScope* can significantly reduce communication overhead and maintain accuracy compared to the previous works. Integrating more network monitoring applications in *MpScope* and developing new data structures natively supporting multiple pipelines are our future work.

CRediT authorship contribution statement

Chengyuan Huang: Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **Tianfan Zhang:** Conceptualization. **Li Wang:** Validation. **Yibo Xiao:** Formal analysis. **Chao Yang:** Software, Project administration, Methodology. **Chen Tian:** Conceptualization. **Xiaoliang Wang:** Formal analysis, Conceptualization. **Dong Zhang:** Methodology. **Bingheng Yan:** Methodology. **Ahmed M. Abdelmoniem:** Writing – original draft, Formal analysis. **Wanchun Dou:** Formal analysis, Conceptualization. **Guihai Chen:** Resources, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgments

The project was supported in part by the National Key R&D Program of China under Grant Number 2022YFB2901502, and in part by the National Natural Science Foundation of China under Grant Numbers 62325205, 62072228, and 62172204.

References

[1] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Elsevier VLDB, 2002, pp. 346–357.
[2] N. Kamiyama, T. Mori, R. Kawahara, Simple and adaptive identification of superspreaders by flow sampling, in: IEEE INFOCOM, 2007, pp. 2481–2485.
[3] Y. Xu, Y. Liu, Ddos attack detection under SDN context, in: IEEE INFOCOM, 2016, pp. 1–9.
[4] H. Zheng, Y. Jiang, C. Tian, L. Cheng, Q. Huang, W. Li, Y. Wang, Q. Huang, J. Zheng, R. Xia, et al., Rethinking fine-grained measurement from software-defined perspective: A survey, IEEE Trans. Serv. Comput. 15 (2021) 3649–3667.
[5] J.D. Case, M. Fedor, M.L. Schoffstall, J. Davin, Rfc1157: Simple network management protocol (snmp), 1990, RFC Editor.
[6] B. Claise, Cisco Systems Netflow Services Export Version 9, Tech. Rep., 2004.
[7] M. Wang, B. Li, Z. Li, Sflow: Towards resource-efficient and agile service federation in service overlay networks, in: IEEE ICDCS, 2004, pp. 628–635.
[8] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, S. Uhlig, Elastic sketch: Adaptive and fast network-wide measurements, in: ACM SIGCOMM, 2018, pp. 561–575.
[9] R. Miao, Y. Zhang, Z. Zheng, R. Wang, R. Zhang, T. Yang, Z. Liu, J. Jiang, CocoSketch: High-performance sketch-based measurement over arbitrary partial key query, IEEE/ACM Trans. Netw. (2023) 1–16.
[10] H. Namkung, Z. Liu, D. Kim, V. Sekar, P. Steenkiste, SketchLib: Enabling efficient sketch-based monitoring on programmable switches, in: USENIX NSDI, 2022, pp. 743–759.
[11] L. Tang, Q. Huang, P.P. Lee, SpreadSketch: Toward invertible and network-wide detection of superspreaders, in: IEEE INFOCOM, 2020, pp. 1608–1617.
[12] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, V. Braverman, One sketch to rule them all: Rethinking network flow monitoring with univmon, in: ACM SIGCOMM, 2016, pp. 101–114.
[13] H. Zheng, C. Tian, T. Yang, H. Lin, C. Liu, Z. Zhang, W. Dou, G. Chen, Flymon: enabling on-the-fly task reconfiguration for network measurement, in: ACM SIGCOMM, 2022, pp. 486–502.
[14] K. Yang, Y. Li, Z. Liu, T. Yang, Y. Zhou, J. He, J. Xue, T. Zhao, Z. Jia, Y. Yang, SketchINT: Empowering INT with TowerSketch for per-flow per-switch measurement, in: IEEE ICNP, 2021, pp. 1–12.
[15] X. Chen, S. Landau-Feibish, M. Braverman, J. Rexford, Beaucoup: Answering many network traffic queries, one memory update at a time, in: ACM SIGCOMM, 2020, pp. 226–239.
[16] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, W. Willinger, Sonata: Query-driven streaming network telemetry, in: ACM SIGCOMM, 2018, pp. 357–371.
[17] R. Teixeira, R. Harrison, A. Gupta, J. Rexford, PacketScope: Monitoring the packet lifecycle inside a switch, in: ACM SOSR, 2020, pp. 76–82.
[18] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, C. Kim, Language-directed hardware design for network performance monitoring, in: ACM SIGCOMM, 2017, pp. 85–98.
[19] Q. Huang, H. Sun, P.P. Lee, W. Bai, F. Zhu, Y. Bao, Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy, in: ACM SIGCOMM, 2020, pp. 404–421.
[20] Z. Liu, J. Bi, Y. Zhou, Y. Wang, Y. Lin, NetVision: Towards network telemetry as a service, in: IEEE ICNP, 2018, pp. 247–248.
[21] M. Chiesa, F.L. Verdi, Network monitoring on multi-pipe switches, Proc. ACM Meas. Anal. Comput. Syst. 7 (1) (2023) http://dx.doi.org/10.1145/3579321.
[22] R. Harrison, Q. Cai, A. Gupta, J. Rexford, Network-wide heavy hitter detection with commodity switches, in: ACM SOSR, 2018, pp. 1–7.
[23] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al., P4: Programming protocol-independent packet processors, ACM SIGCOMM Comput. Commun. Rev. 44 (2014) 87–95.
[24] Intel, Barefoot's Tofino, https://www.barefootnetworks.com/technology/.
[25] V. Shrivastav, Stateful multi-pipelined programmable switches, in: ACM SIGCOMM, 2022, pp. 663–676.
[26] R. Harrison, S.L. Feibish, A. Gupta, R. Teixeira, S. Muthukrishnan, J. Rexford, Carpe elephants: Seize the global heavy hitters, in: Proceedings of the Workshop on Secure Programmable Network Infrastructure, SPIN '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 15–21, http://dx.doi.org/10.1145/3405669.3405820.
[27] D. Ding, M. Savi, G. Antichi, D. Siracusa, An incrementally-deployable P4-enabled architecture for network-wide heavy-hitter detection, IEEE Trans. Netw. Serv. Manag. 17 (2020) 75–88.
[28] X.Z. Khooi, L. Csikor, J. Li, D.M. Divakaran, In-network applications: Beyond single switch pipelines, in: 2021 IEEE 7th International Conference on Network Softwarization, NetSoft, 2021, pp. 1–8, http://dx.doi.org/10.1109/NetSoft51509.2021.9492665.
[29] F.L. Verdi, M. Chiesa, Heavy hitter detection on multi-pipeline switches, in: Proceedings of the Symposium on Architectures for Networking and Communications Systems, ANCS '21, Association for Computing Machinery, New York, NY, USA, 2022, pp. 121–124, http://dx.doi.org/10.1145/3493425.3502760.
[30] P. Flajolet, É. Fusy, O. Gandouet, F. Meunier, Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm, Discrete Math. Theor. Comput. Sci. (2007).
[31] B. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.
[32] R. Fontugne, P. Borgnat, P. Abry, K. Fukuda, Mawilab: combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking, in: ACM CoNEXT, 2010, pp. 1–12.
[33] CAIDA, Passive monitor: Equinix-nyc, 2023, https://www.caida.org/data/monitors/passive-equinix-nyc.xml.

Chengyuan Huang received the B.Eng. and Ph.D. degrees from Beijing University of Posts and Telecommunications in 2015 and 2021, respectively. From 2021 to 2023, he was a postdoctoral researcher with Purple Mountain Laboratories, Nanjing, China. He is currently an assistant researcher with the Department of Computer Science and Technology, Nanjing University. His research interests include data center networks, software-defined networking and distributed systems.

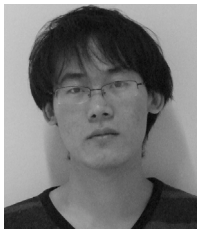
Tianfan Zhang received his Master degree from the Department of Computer Science and Technology, Nanjing University, China in 2023. Before that, he got the bachelor degree from Nanjing University of Aeronautics and Astronautics in 2020. His research interests include Computer Network and distributed system.

Li Wang received the B.S. degree from the Department of Computer Science and Technology, Nanjing University, China, in 2024. He is currently working toward the 1st-year ME degrees with the Department of Computer Science and Technology, Nanjing University. His research interests include in-network computing and network testing.

Yibo Xiao is currently a senior undergraduate student majoring in Computer Science and Technology at Nanjing University. His research interests include network system design.

Chao Yang received the B.S. degree from the Department of Computer Science and Technology, Nanjing University, China, in 2022. He is currently working toward the 2nd-year ME degrees with the Department of Computer Science and Technology, Nanjing University. His research interests include in-network computing and congestion control.

Chen Tian received the B.S., M.S., and Ph.D. degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. He is a professor with the State Key Laboratory for Novel Software Technology, Nanjing University, China. He was previously an associate professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology, China. From 2012 to 2013, he was a postdoctoral researcher with the Department of Computer Science, Yale University. His research interests include data center networks, network function virtualization, distributed systems, Internet streaming and urban computing.



Xiaoliang Wang received the Ph.D. degree from the Graduate School of Information Sciences, Tohoku University, Japan. From 2010 to 2014, he was an Assistant Professor with the Department of Computer Science and Technology, Nanjing University, China, where he is currently an Associate Professor. He has published more than 30 technical articles at premium international journals and conferences, including the IEEE TIT, the IEEE TCOM, the IEEE INFOCOM, USENIX ATC, and USENIX FAST. His research interests include network systems and optical switching networks.

Dong Zhang as the chairman of Jinan Inspur Data Technology Co., Ltd., has led the development of the world's highest computing and storage density rack server, the first China UNIX operating system. He has made creative contributions in areas such as converged architecture and high-end system software, earning one national award, and eleven provincial-level awards.

Bingheng Yan received his Ph.D. at Xi'an JiaoTong University in 2010. He is broadly interested in the area of operating system, virtualization, and cloud computing. He is the cloud R&D director of Jinan Inspur Data Co., Ltd., where he has led a wide range of virtualization research projects, and the development of Inspur's server virtualization product InCloud Sphere, which break the global world record of SpecVirt.

Ahmed M. Abdelmoniem (Member ACM, IEEE, USENIX) received his PhD in Computer Science and Engineering from the Hong Kong University of Science and Technology, HK in 2017. He is an Associate Professor at the School of Electronic Engineering and Computer Science, Queen Mary University of London, UK. Formerly, he was a Research Scientist at KAUST, Saudi Arabia, and a Senior Researcher with Huawei's Future Networks Lab, HK. He is an investigator on projects totaling USD 1.5mil in funding. His research interests lie in the intersection of distributed systems, networks, and machine learning. His work appears in top-tier conferences and journals, including NeurIPS, AAAI, MLSys, ACM EuroSys, IEEE INFOCOM and ICDCS, IEEE/ACM ToN, IEEE IoTJ, IEEE TCC, IEEE TIFS and Elsevier Computer Networks.

Wanchun Dou received the Ph.D. degree in mechanical and electronic engineering from the Nanjing University of Science and Technology, China, in 2001. He is currently a full professor at the State Key Laboratory for Novel Software Technology, Nanjing University. From April 2005 to June 2005 and from November 2008 to February 2009, he respectively visited the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, as a visiting scholar. He has chaired three National Natural Science Foundation of China projects and published more than 60 research papers in international journals and international conferences. His research interests include workflow, cloud computing, and service computing.

Guihai Chen received the B.S. degree in computer software from Nanjing University, in 1984, the ME degree in computer applications from Southeast University in 1987, and the Ph.D. degree in computer science from the University of Hong Kong, in 1997. He is a distinguished professor of Nanjing University. He had been invited as a visiting professor by Kyushu Institute of Technology, Japan, University of Queensland, Australia and Wayne State University. He has a wide range of research interests with focus on parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture and data engineering. He has published more than 350 peer-reviewed papers, and more than 200 of them are in well-archived international journals such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, IEEE Transactions on Knowledge and Data Engineering, IEEE/ACM Transactions on Networking and ACM Transactions on Sensor Networks, and also in well-known conference proceedings such as HPCA, MOBIHOC, INFOCOM, ICNP, ICDCS, CoNext and AAAI. He has won 9 paper awards including ICNP 2015 best paper award and DASFAA 2017 best paper award.