FastKeeper: A Fast Algorithm for Identifying Top-k Real-time Large Flows

Yanshu Wang*†, Dan Li*†, Jianping Wu*
*Department of Computer Science and Technology, Tsinghua University, Beijing, China
†Tsinghua Shenzhen International Graduate School, Tsinghua University, Shenzhen, China
Emails: wys17@mails.tsinghua.edu.cn, tolidan@tsinghua.edu.cn, jianping@cernet.edu.cn

GLOBECOM 2021 - 2021 IEEE Global Communications Conference | 978-1-7281-8104-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/GLOBECOM46510.2021.9685170

***Abstract*—Precise identification of large flows is a critical task in network traffic measurement. Previous works focus on identification of elephant flows (i.e. large flows from the beginning of the measurement). However, we generally observe that the flow rates change periodically and abruptly. In addition, the large flows may become small flows over time. Thus, elephant flows are not equal to the real-time large flows, and previous works cannot be used for the identification of the real-time large flows that is more meaningful for modern network applications. Nevertheless, identification of real-time large flows is challenging in that it requires accurate measurement of real-time flow rates and timely replacement of flows that have become small in the measurement data structure.**

In this paper, we propose FastKeeper to identify real-time large flows with a primary goal of simultaneously achieving low overhead, high performance and high accuracy. FastKeeper employs a sliding-window-based algorithm for accurate measurement of real-time flow rates and a bitmap-voting algorithm for timely replacement of flows that have become small in the measurement data structure. We evaluate it on DPDK using the traces from an operator network, and the evaluation demonstrates that it achieves high accuracy (98%) and processing throughput (25.33Mpps).

***Index Terms*—Traffic Measurement, Sketch-based Algorithms**

I. Introduction

Precise identification of large flows plays an important role in load balance [1], flow schedule [2], elephant flow offloading [3] and anomaly detection [4]. Previous works [5]–[7] focus on elephant flow identification. Elephant flow identification is to find the top-k flows with the largest flow sizes from the beginning of the measurement, where a flow is usually defined as a combination of certain fields in the packet header. However, according to our measurement of the operator network traffic, the flow rates change periodically and abruptly. Besides, the large flows may become small flows over time (detailed in Section II-A). Hence, elephant flows are not equal to the real-time large flows. Identification of real-time large flows is more meaningful, because most network applications are processing flows based on real-time flow rates. We define real-time large flows as top-k flows with the largest flow rates (dividing the flow sizes in a fine-grained time window by the time window size).

Identification of the real-time large flows is challenging, because real-time large flow identification requires accurate measurement of real-time flow rates and timely replacement of flows that have become small in the measurement data structure. Hence, elephant flow identification algorithms can not be directly used to identify the real-time large flows.

Besides, the real-time large flow identification systems are required to have high performance (i.e. processing throughput) and consume as few resources as possible to avoid impacts on other running applications.

In this paper, we present FastKeeper, a novel measurement algorithm that can identify real-time large flows, with a primary goal of simultaneously achieving low overhead, high performance and high accuracy. Specifically, FastKeeper employs a sliding-window-based measurement algorithm and a novel reset strategy to identify the real-time large flows. For the sliding-window-based measurement algorithm, FastKeeper measures the flow rates in a time window by a sketch data structure. This design can provide accurate measurement of real-time flow rates. For the novel reset strategy, FastKeeper records the flow rates in a time window by a bitmap and employs the bitmap-voting algorithm to reset the flow counters. The novel reset strategy can guarantee timely replacement of flows that have become small in the sketch data structure. Besides, FastKeeper achieves high performance under the constraint of limited storage by the sketch and bitmap data structure.

We implement a FastKeeper prototype on top of DPDK and evaluate it on commodity servers with 100GbE Mellanox ConnectX-5 NICs. Experimental results demonstrate that FastKeeper achieves high precision (above 90% in most cases), compared with other algorithms [6], [8]–[12] (below 60% in most cases). The throughput of FastKeeper is high (25.33Mpps) as well.

II. Background and Motivation

In this section, we first demonstrate the network traffic characteristics which motivate our work of real-time large flow identification, and then we analyze the challenges we face.

A. Network traffic characteristic analysis

In this section, we measure the traffic characteristics in a cloud gateway of a large public cloud provider and present our

This work was supported by the National Key Research and Development Program of China under Grant Grant 2018YFB1800100, the Research and Development Program in Key Areas of Guangdong Province under Grant 2018B010113001, and the National Natural Science Foundation of China under Grant 61772305.

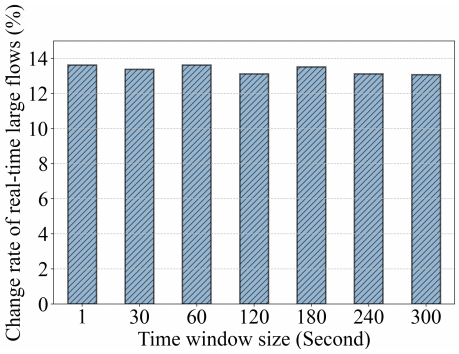
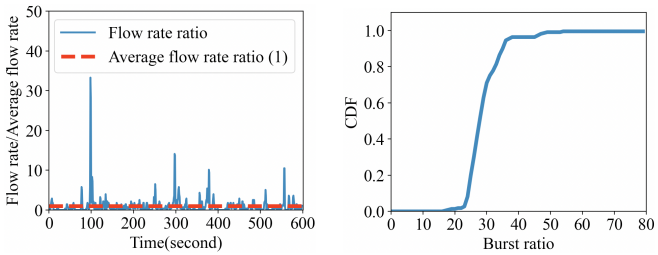


Fig. 1. Change rate of real-time large flows at different time scales.



(a) Burstiness of a typical flow. (b) Distribution of the burst ratio.

Fig. 2. The characteristics of bursty flows.

findings. The gateway manipulates the received packet headers and forwards the packets with tunnels, e.g. VxLAN [13] or GRE [14]. Sometimes, ACL and NAT may also be applied. We collect packet-level traces at different times in a typical work day, namely 8:00 am, 12:00 pm and 10:00 pm. The traces span three different regions with large inbound and outbound traffic.

We use pps (packet per second) as the measurement unit as most network functions are based on packet processing.

Flow rates change periodically. We examine the top 10% flows with the largest real-time flow rates in every time window (referred as real-time large flows) to check their time variance characteristics in consecutive time windows. We use different time windows, namely 1s, 30s, 1min, 2min, 3min, 4min and 5min. For each window size, we count the number of changed real-time large flows between consecutive windows. Dividing this number by the total number of the flows, we can get the change rates of real-time large flows. As shown in Fig. 1, real-time large flows constantly change over time and above 13% of the real-time large flows will be no longer real-time large flows in next time window. This phenomenon indicates that the real-time large flows will change over time periodically.

Flow rates change abruptly. In addition to the periodical change, we also observe the abrupt change of the flow rates. The flow rate can burst to a high rate in a quite short time (most below 1 second). Fig. 2(a) shows the burstiness of a typical flow. We define the ratio of the peak rate to the average rate as the burst ratio. As shown in Fig. 2(b), most flows have a burst ratio above 30 and the burst ratio can be as high as 80 for some flows.

B. Related works and motivation for real-time large flow identification

Previous flow measurement algorithms mainly focus on the elephant flow identification rather than the real-time large flow identification. Two different categories have emerged in the field of flow measurement, i.e. sketch-based and counter-based algorithms. Sketch-based algorithms [6], [8], [9] employ a two-dimensional matrix to record the flow statistics. Each element in the matrix is a bucket, which consists of several counters. When the packet arrives, the sketch-based algorithm hashes the packet to different rows of the matrix and updates the corresponding counters. The statistics of individual packets can be then estimated by reading the counters in the sketch. The sketch-based algorithms can provide accurate measurement and high performance. The well-known algorithm CM sketch [8] employs a counter in each bucket to record the flow size. UnivMon [9] uses an application-agnostic data plane monitoring primitive to achieve both generality and high accuracy. Heavykeeper [6] improves the accuracy of elephant flow measurement by sacrificing the accuracy of mice flow measurement. Counter-based algorithms [10]–[12] are deterministic. They maintain a summary of the flows. The summary consists of a small subset of the packets with associated counters approximating the flow statistics. It was experimentally observed that counter-based algorithms provide better guarantees than sketch-based algorithms. However, the counter-based algorithms have lower performance than that of sketch-based algorithms.

These algorithms focus on the measurement of the flow sizes. Besides, WCSS [12] refreshes the flow sizes regularly, but the granularity of refresh time is coarse and WCSS which is counter-based algorithm has low performance. According to our analysis in Section II-A, the flow rates in the network are changing periodically and abruptly. Hence, the flows with large flow sizes may have low real-time flow rates and vice versa. The elephant flows (the flows with large flow sizes) are not equal to the real-time large flows (the flows with large real-time flow rates). However, in most of network application scenarios, the real-time large flows (rather than elephant flows) are the focus of processing and scheduling. For example, the network load balancer should schedule the real-time large flows to achieve the balanced load for each processor. The anomaly detector should focus on the anomaly real-time large flows. In hardware offloading scenario, the real-time large flows should be offloaded to the hardware, in order to offload most of the traffic. These network applications require accurate and timely identification of real-time large flows. Therefore, identifying real-time large flows is more meaningful and challenging.

C. Challenges of real-time large flow identification

In this section, we introduce two challenges in real-time large flow identification.

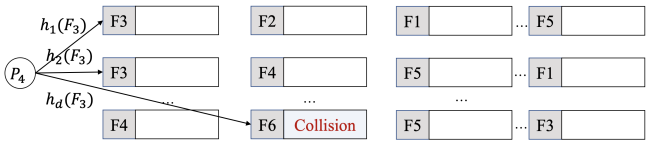


Fig. 3. Flow collision in data structure. (F_3 is the fingerprint of f_3)

How to measure the real-time flow rates accurately? To identify top-k real-time large flows, the flow measurement algorithm should record the real-time flow rates. When the flow rates are out-of-date, the measurement algorithm should forget the old flow rates and start to measure the new flow rates. How to design the measurement data structure and update algorithm is challenging.

How to timely replace the flows that have become small? Real-time large flow identification algorithm requires a data structure to store the top-k real-time large flows. However, the rates of the flows stored in the data structure keep changing. The new real-time large flows should replace the past real-time large flows that have become small flows. As shown in Fig. 3, when flow f_3 becomes a real-time large flow and collides with flow f_6, our algorithm should replace f_6 with f_3. With the proposed reset strategies, the small flows can be replaced by the new real-time large flows.

We propose FastKeeper to solve the above two challenges.

III. The Design of FastKeeper

A. Design overview

To identify real-time large flows accurately, FastKeeper employs a sliding-window-based approach to identifying the real-time large flows. The flow counters only record the real-time flow rates in the window. Each time the window slides, the counters will forget the old flow rates and begin to record new flow rates. The time window slides a time block each time, so the flows can be measured in a more fine-grained manner. The sliding-window-based approach is based on a hash table data structure to store all the rates of real-time large flows. As shown in Fig. 4, FastKeeper uses K counters to record a flow's received packets. Each counter records the number of packets received within a time block b. The flow rate is the sum of all the counters in each block. Each time the window slides over a time block (b packets), the counters which record the oldest flow rates are reset. In this way, only the packets received in the sliding window are summed as the real-time flow rates.

To timely replace the flows that have become small, we propose reset strategies to quickly reset the buckets. We maintain collision counters to record the collision times of the flows. When the collision counter is several times larger than the flow rate counter, FastKeeper will reset the bucket and a faster flow will be inserted into the bucket. In addition, an improved version of the reset strategy is proposed in Section III-D to replace the collision counters with a bitmap. The optimized reset strategy can record the flow rates with less storage overhead and fast reset the flow counters.

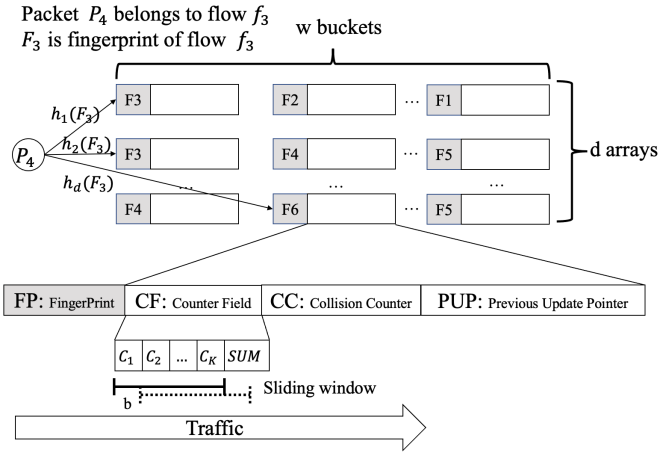


Fig. 4. The data structure of FastKeeper

B. Data structure and algorithm design of FastKeeper

In this section, we expand on the design details of FastKeeper's data structure and how to insert, update and query the flows in FastKeeper. We also discuss some optimizations of FastKeeper.

1) Data structure of FastKeeper: As shown in Fig. 4, FastKeeper is comprised of d arrays, and each array is composed of w buckets. Each bucket consists of four fields: a fingerprint field, a counter field, a collision counter field and a previous update pointer field. The counter field consists of $K+1$ counters. For convenience, we use $A_j[t]$ to represent the t^{th} bucket in the j^{th} array, and use $A_j[t].FP$, $A_j[t].C_i$, $A_j[t].CC$, $A_j[t].SUM$, and $A_j[t].PUP$, to represent the flow's fingerprint, i^{th} counter (block counter), collision counter, the flow counter (the sum of block counter) and the previous update pointer respectively. Arrays $A_1...A_d$ are associated with hash functions $h_1(.)...h_d(.)$. These d hash functions $h_1(.)...h_d(.)$ need to be pairwise independent.

2) The insertion and query algorithms of FastKeeper : **Insertion algorithm:** For each packet P_l belonging to flow f_i, FastKeeper computes the d hash functions. d hash functions will map f_i to d buckets $A_j[h_j(f_i)]$ $(1 \le j \le d)$ (one bucket in each array). As shown in Fig. 4, for each mapped bucket, FastKeeper updates the bucket according to Algorithm 1. The update process consists of two stages: the counter updating stage and window sliding stage. In the counter updating stage, the flow counters SUM and CC are updated according to the update strategy. In the window sliding stage, FastKeeper resets the old block counter and start to measure new flow rates. We will introduce the counter updating and window sliding stage in detail.

Counter updating stage: Initially, all fingerprint fields are $NULL$, and all counter fields are 0.

Case 1: When $A_j[h_j(f_i)].FP = NULL$. It means that the flow does not store any flow, and we can insert new flows to the bucket. FastKeeper sets $A_j[h_j(f_i)].FP = F_i$ (F_i is the fingerprint of f_i) and $A_j[h_j(f_i)].SUM = A_j[h_j(f_i)].SUM + 1 = 1$ which corresponds to lines 1∼2 in Algorithm 1.

Case 2: When $A_j[h_j(f_i)].FP = F_i$. It means that the flow f_i is stored in the bucket $A_j[h_j(f_i)]$, so FastKeeper sets $A_j[h_j(f_i)].SUM = A_j[h_j(f_i)].SUM + 1$, which corresponds to lines 1∼2 in Algorithm 1.

Case 3: When $A_j[h_j(f_i)].FP \neq F_i$. It means that the bucket does not store flow f_i. Two flows collide in the same bucket. FastKeeper sets $A_j[h_j(f_i)].CC = A_j[h_j(f_i)].CC + 1$, which corresponds to lines 3∼4 in Algorithm 1.

Window sliding stage: In the window sliding stage, FastKeeper resets the old block counter and update the global packet counter C and the block pointer p according to Algorithm 2. For each incoming packet, FastKeeper increases the global packet counter C by one (lines 1 in Algorithm 2), and when the global counter reaches the boundary of the block, FastKeeper will increase the block pointer p by one (lines 2∼4 in Algorithm 2). If the block pointer p is not equal to the previous update pointer PUP, FastKeeper performs window sliding stage.

Algorithm 1 Update Bucket(UpdateBucket(B,FP))

Input: The packet counter C, the time window size W, the number of blocks in the time window K, the block pointer p, the fingerprint of the flow FP, a predefined threshold λ.

1: **if** $(FP = B.FP)$ **then**
2: $B.SUM \leftarrow B.SUM + 1$
3: **else**
4: $B.CC \leftarrow B.CC + 1$
5: **end if**
6: **if** $p \neq B.PUP$ **then**
7: **if** $p \neq B.PUP + 1$ **then**
8: **for all** i such that $B.PUP \leq i \leq p - 2$ **do**
9: $C_i \leftarrow 0$
10: **end for**
11: **end if**
12: $B.SUM \leftarrow B.SUM - C_p$
13: $C_{p-1} \leftarrow B.SUM - \sum_{i=1,2\ldots,p-2,p,p+1\ldots,K} C_i$
14: **end if**
15: **if** $\frac{B.CC}{B.SUM} \geqslant \lambda$ **then**
16: $B.FP \leftarrow FP$, $B.SUM \leftarrow 0$
17: **for all** i such that $0 \leq i \leq K$ **do**
18: $C_i \leftarrow 0$
19: **end for**
20: $C_p \leftarrow 1$, $B.CC \leftarrow 0$
21: **end if**
22: $B.PUP \leftarrow p$

Algorithm 2 Update window(UpdateWindow())

Input: The packet counter C, the time window size W, the number of blocks in the time window K, the block pointer p, initialized as 0.

1: $C \leftarrow (C + 1)\ mod\ W$
2: **if** C mod $\frac{W}{K} = 0$ **then**
3: $p \leftarrow (p + 1)\ mod\ K$
4: **end if**

Case 1: When $p \neq PUP$ and $p = PUP + 1$, it means that the block counter reaches the boundary of block, FastKeeper updates the flow counters according to lines 12∼13 in Algorithm 1.

Case 2: When $p \neq PUP$ and $p \neq PUP + 1$, it means that the block counter reaches the boundary of block and no packets arrives in the bucket in a few blocks. FastKeeper resets the block counters where no packets arrives (lines 7∼11 in Algorithm 1) and updates the flow counters (lines 12∼13 in Algorithm 1).

Bucket reset strategy: FastKeeper needs a reset strategy to timely replace the flows that have become small. In the basic reset strategy, FastKeeper employs a collision counter CC to address this problem. When a flow collision occurs in a bucket, we increase the collision counter CC of that bucket, and then we calculate $\frac{CC}{SUM}$. A large $\frac{CC}{SUM}$ indicates that the flow in the bucket has already become smaller than the colliding flow. So the bucket should be reset and the new real-time large flow should be inserted into the bucket (lines 15∼21 in Algorithm 1).

Query algorithm: To query the flow rate of f_i, FastKeeper first calculates the d hash functions. d hash functions will map f_i to d buckets $A_j[h_j(f_i)]$ $(1 \leq j \leq d)$. FastKeeper then chooses the mapped buckets where $A_j[h_j(f_i)].FP = F_i$ and returns the maximum value of SUM in the selected buckets. Because there are flow collisions in the same bucket, the flow rate will be underestimated.

Algorithm 3 FastKeeper for real-time large flow identification

Input: A packet P_l belonging to the flow f_i, the sampling rate r.

1: $exist \leftarrow false$
2: **if** $f_i \in min\_heap$ **then**
3: $exist \leftarrow true$
4: **end if**
5: $max\_counter \leftarrow 0$
6: $UpdateWindow()$
7: **for** $j = 1$ to d **do**
8: **if** $random(0, 1) \leq r$ **then**
9: $UpdateBucket(A_j[h_j[f_i]], calculateFP(f_i))$
10: **end if**
11: $C = A_j[h_j[f_i]].SUM$
12: **if** $C \geq max\_counter$ **then**
13: $max\_counter = C$
14: **end if**
15: **end for**
16: **if** $exist = true$ **then**
17: $min\_heap[f_i] \leftarrow max\_counter$
18: **else if** min_heap has empty buckets || C > min_heap_{min} **then**
19: $min\_heap.insert(f_i)$
20: **end if**

C. FastKeeper for real-time large flow identification

In order to identify top-k real-time large flows, we employ a min-heap data structure together with FastKeeper. The min-heap is used to store the fingerprints and the flow rates of top-k real-time large flows. The algorithm for top-k real-time large flow identification is shown in Algorithm 3. For each incoming packet P_l belonging to flow f_i, we first insert the packet into FastKeeper (line 9 in Algorithm 3), and query the flow rate of f_i. FastKeeper will return the rate of f_i (lines 11∼14 in Algorithm 3). If the rate of flow f_i is already in the min-heap, we update the heap with the returned flow rate

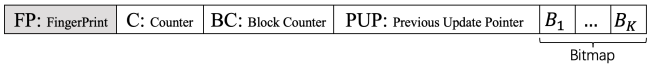


Fig. 5. The bucket of bitmap-based FastKeeper

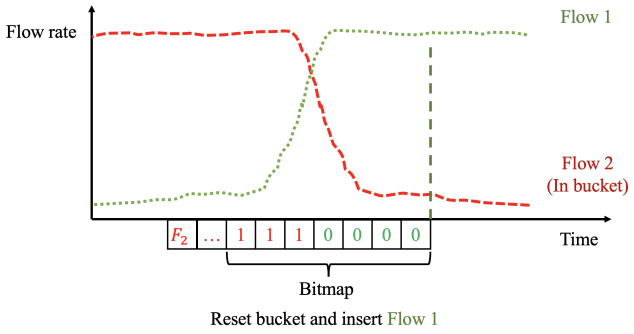


Fig. 6. Bitmap-voting algorithm

(lines 16∼17 in Algorithm 3). If the rate of the flow f_i is not in the min-heap and the min-heap is empty, we directly insert the flow f_i and its rate to the heap. If the rate of flow f_i is not in min-heap and the returned rate is larger than the rate of flow in the root node of the min-heap, we delete the root node and insert the flow into the min-heap (lines 18∼19 in Algorithm 3). When we query the top-k real-time large flows, min-heap directly returns the flow fingerprints.

D. Optimizations

Storage optimization: FastKeeper arranges the counters of the same time block into the same array. Every time the window slides, only the array which records the oldest counters is reset, which is a piece of continuous memory. Continuous memory can leverage spatial locality of cache, and the access of cache can be 20 × faster than memory.

Sampling technique: The throughput of network traffic is extremely high in some scenarios, for example, the gateways need to process the incoming and outgoing traffic at 10s of Tbps. The sampling technique can help FastKeeper achieve higher processing throughput (line 8 in Algorithm 3).

Bitmap voting: In FastKeeper (bitmap), i.e. the FastKeeper which uses a bitmap-voting strategy, we use a bitmap to record the relative flow rates in a sliding window with only one bit for each block. The reset strategies are based on the bitmap. The structure of bitmap-based bucket is shown in Fig. 5. The bucket contains $A_j[t].FP$, $A_j[t].C$, $A_j[t].BC$, $A_j[t].PUP$ and $A_j[t].B_i$ to represent its fingerprint field, flow counter field, block counter field, previous update pointer field and i^{th} bit of the bitmap respectively. The insertion process and query process is the same as the description in Section III-B2. For each incoming packet P_l belonging to flow f_i, we compute d hash functions and map f_i to d buckets $A_j[h_j(f_i)]$ $(1 \leq j \leq d)$. For each bucket mapped to, we update the bucket. First, FastKeeper (bitmap) updates counter C and block counter BC according to the fingerprint FP. Then FastKeeper (bitmap) updates the bitmap according to the block counter BC. The window update process is the same as FastKeeper in Section III-B2. The counter update stage consists of the update processes of the block counter and the bitmap.

Updating the block counter: Initially, all fingerprint fields are initialized to $NULL$, and all counter fields are initialized to 0.

Case 1: When $A_j[h_j(f_i)].FP = NULL$. It means that the bucket do not store any flow, and we can insert the new flow into the bucket. FastKeeper sets $A_j[h_j(f_i)].FP = F_i$, $A_j[h_j(f_i)].C = 1$ and $A_j[h_j(f_i)].BC = 1$.

Case 2: When $A_j[h_j(f_i)].FP = F_i$. It means that the flow f_i is stored in the bucket $A_j[h_j(f_i)]$, so FastKeeper sets $A_j[h_j(f_i)].C = A_j[h_j(f_i)].C + 1$ and $A_j[h_j(f_i)].BC = A_j[h_j(f_i)].BC + 1$.

Case 3: When $A_j[h_j(f_i)].FP \neq F_i$. It means that the flow does not store the flow f_i. Two flows collide in the same bucket. FastKeeper sets $A_j[h_j(f_i)].BC = A_j[h_j(f_i)].BC - 1$.

At the boundary of each block, $A_j[h_j(f_i)].BC$ will be reset and the bitmap will be updated.

Updating the bitmap: Initially, all the bits of the bitmap are set to 1.

Case 1: When $A_j[h_j(f_i)].BC > 0$. It means that the flow in the mapped bucket is large than the colliding flows, we record the history in bitmap $A_j[h_j(f_i)].B_i = 1$.

Case 2: When $A_j[h_j(f_i)].BC \leq 0$. It means that the flow in the bucket is smaller than the colliding flows, we record the history in bitmap $A_j[h_j(f_i)].B_i = 0$.

Reset strategy: We reset the bucket based on the relative flow rates in the bitmap by the bitmap-voting algorithm. If the number of 0 in the bitmap is larger than the number of 1 in bitmap, it means that the colliding flow is faster than the flow in the bucket, and the bucket should be reset.

For example, as shown in Fig. 6, the bucket stores flow 2. The flow rates are changing. Flow 2 gets slow and flow 1 becomes fast. Bitmap is updated by the algorithm. And we use the majority voting to decide whether the bucket will be reset. In Fig. 6, because the number of 0 in the bitmap is larger than the number of 1, FastKeeper will reset the bucket and insert the flow 1 to the bucket. The actual situation is that flow 1 becomes faster than flow 2. The results of the bitmap-voting algorithm accord with the actual situation.

IV. Evaluation

A. Experiment Setup

Testbed setup: Our testbed consists of 2 servers which are directly connected. Each server is a Dell PowerEdge R740 with two 8-core Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz and 64GB RAM. One server plays the role as the sender and the other server as the receiver. FastKeeper is deployed in the receiver. Each server has one Mellanox ConnectX-5 NIC with one 100GbE port. We use *dpdk_pkg_gen* version 3.2.4 [15] together with DPDK version 17.11 [16] to send and receive packets.

We implement FastKeeper as a C library and integrate it with DPDK. The total library has nearly 3K lines of C code.

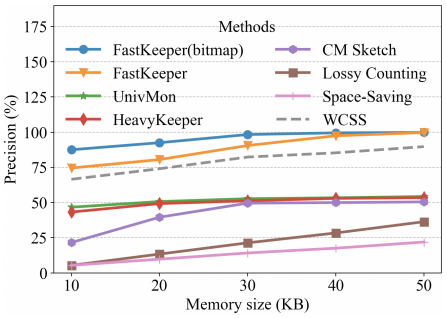


Fig. 7. Precision vs. memory size.

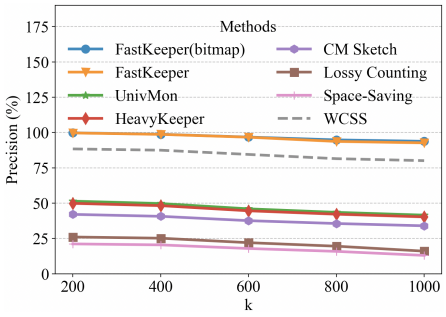


Fig. 8. Precision vs. k.

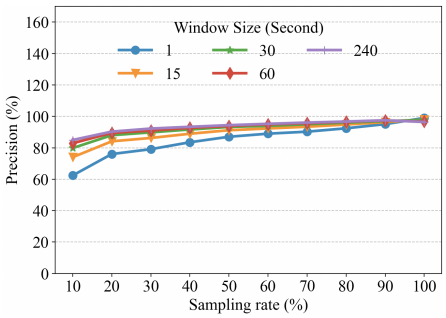


Fig. 9. Precision vs. sampling rate in different window sizes.

FastKeeper uses a CPU core for the real-time large flow identification and other CPU cores run network function logic (receiving, forwarding, ACL or NAT).

Trace: We use the packet-level trace collected from the operator network of a large cloud service provider throughout this paper. This trace contains 33M flows and 77M packets.

B. Algorithms and metrics

We compare FastKeeper with six popular algorithms, Count-Min Sketch [8], UnivMon [9], HeavyKeeper [6], Space-Saving [10], Lossy Counting [11] and WCSS [12]. WCSS can also forget the flow information periodically. We denote the FastKeeper algorithm with storage optimization and bitmap-voting optimization as FastKeeper (bitmap) and FastKeeper with all optimizations as optimized FastKeeper. We use 5 tuples (source IP address, destination IP address, source port, destination port and protocol type) as the flow key and consider the following performance metrics:

Precision and recall: Precision is the ratio of the number of correctly returned real-time large flows to k. We can define precision as $\frac{\sigma}{k}$, where σ is the number of correctly returned real-time large flows by FastKeeper. Because we aim at identifying top-k real-time large flows, the recall is also $\frac{\sigma}{k}$. If many precision results are generated from the window-based algorithms, we aggregate these results by averaging.

Throughput: Throughput is the insertion speed of the algorithms in the testbed. We define throughput as $\frac{N}{T}$, where N is the number of packets inserted into FastKeeper and T is the time FastKeeper uses to insert N packets. We use Million of packets per second (Mpps) to measure the throughput.

C. Experimental results of precision

We analyse the precision results for varying memory sizes, k values (without sampling) and sampling rates. In this experiment, we set the window size W to 1290k packets (5 seconds), the block size b to 258k packets (1 second), the number of array d to 3 and the threshold λ to 2. For the experiments of varying memory sizes, we set k to 100, and for experiments of varying k values, we set the memory size to 100KB. For the experiment of varying sampling rates and window sizes, we set k to 100 and memory size to 100KB.

Precision vs. memory size: As shown in Fig. 7, FastKeeper and FastKeeper (bitmap) have higher precision compared with other algorithms. When we use 30KB memory size, the precision of FastKeeper and FastKeeper (bitmap) reaches 90% and 98%, but the precision of UnivMon, HeavyKeeper, CM sketch, Lossy Counting, Space-Saving and WCSS is 52%, 51%, 49%, 21%, 14% and 82% respectively. In addition, we notice that when we use larger memory size, the precision of FastKeeper and FastKeeper (bitmap) becomes closer. To be specific, when the memory is 10KB, the precision of FastKeeper (bitmap) and FastKeeper is 87% and 74%, and when the memory is 50KB, the precision of FastKeeper (bitmap) and FastKeeper is both 99%. Because FastKeeper (bitmap) uses bitmap to save memory, FastKeeper (bitmap) has more buckets than FastKeeper when we use the same memory size. Hence, FastKeeper (bitmap) has a lower probability of flow collisions than that of FastKeeper. FastKeeper (bitmap) and FastKeeper have similar precision when enough buckets has eliminated most of the flow collisions.

Precision vs. k: As shown in Fig. 8, when k becomes larger, the precision of all algorithms decreases. FastKeeper and FastKeeper (bitmap) still have high precision when k is large. Even when k increases to 1000, the precision of FastKeeper and FastKeeper (bitmap) is still above 92%. Precision of other algorithms also decreases when k increases. For example, when k is 200, the precision of UnivMon, HeavyKeeper, CM sketch, Lossy Counting, Space-Saving and WCSS is 51%, 49%, 42%, 26%, 21%, and 88%. But when k increases to 1000, the precision of UnivMon, HeavyKeeper, CM sketch, Lossy Counting, Space-Saving and WCSS is 41%, 40%, 33%, 16%, 13% and 80%. It is worth noting that when we have enough memory to eliminate most of the flow collisions, both FastKeeper and FastKeeper (bitmap) have high precision.

Precision vs. sampling rate in different window sizes: As shown in Fig. 9 (the unit of window size is a second or 258k packets, and we use optimized FastKeeper), a small sampling rate leads to low precision, and a large window can benefit the precision, because the optimized FastKeeper needs to get enough packets to converge. When the sampling rate is small, the optimized FastKeeper needs more packets (large window) to maintain the same precision. But even the sampling rate is 10%, the precision of the optimized FastKeeper is above 60%, which is higher than UnivMon, HeavyKeeper, CM sketch,

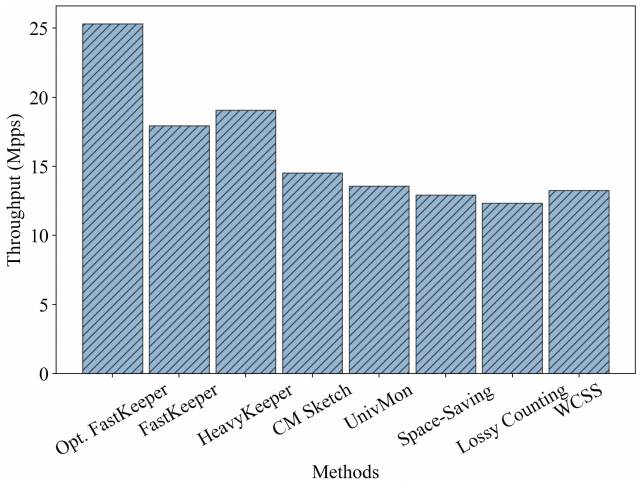


Fig. 10. Throughput of different algorithms.

Lossy Counting and Space-Saving. We notice that when the sampling rate is 100%, the precision will decrease as the window size increases, because a larger window contains more flows, and the collisions of different flows in the same bucket lead to low precision.

D. Experimental results of throughput

The throughput evaluation is based on the testbed described in Section IV-A. In the sender, we use 5 CPU cores to generate packets and the throughput can reach 25Mpps. In the receiver, we use two receiving cores to receive the packets, and maximum throughput of each core is 13 Mpps. The real-time large flow identification algorithms are deployed in other CPU core to receive the traffic from the receiving cores.

In the experiment, the receiving cores receive the packets and enqueue the packets to a lockless queue and the real-time large flow identification algorithm fetches the packets from the queue. We measure the maximum throughput of the receiver which can not fill up the lockless queue, because if the lockless queue is full, it indicates that the real-time large flow identification algorithms can not process these packets in time. We set the queue capacity to 128 packets, which is shallow enough to buffer few packets for the accurate throughput measurement. We set k to 100, memory size to 100MB, d to 3, λ to 2, sampling rate to 0.7, the sliding window size to 1290k packets (5 seconds) and the block size b to 258k packets (1 second).

The throughput of FastKeeper on DPDK is shown in Fig. 10, and the throughput of optimized FastKeeper, FastKeeper, HeavyKeeper, CM sketch, UnivMon, Space-Saving, Lossy Counting and WCSS is 25.33 Mpps, 17.95 Mpps, 19.06 Mpps, 14.51 Mpps, 13.57 Mpps, 12.93 Mpps, 12.34 Mpps and 13.25 Mpps. FastKeeper has a sliding-window-based update logic, so the throughput is slightly lower than HeavyKeeper. But optimized FastKeeper uses bitmap-voting and sampling techniques to accelerate the real-time large flow identification process, so the throughput of optimized FastKeeper is higher than other algorithms.

V. Conclusion

We propose FastKeeper, a sketch-based top-k real-time large flow identification algorithm, which can identify the real-time large flows accurately and timely. For the accurate measurement, FastKeeper uses a sliding window to forget the flow history and maintain the recent flow rates. For the timely replacement, FastKeeper leverages a bitmap-voting strategy to store the relative flow rates efficiently and reset the bucket quickly. Our experiments demonstrate that FastKeeper achieves higher throughput (25.33Mpps) and accuracy (98%) in top-k real-time large flow identification compared to the state-of-the-art solutions.

References

[1] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 523–535.

[2] A. B. Faisal, H. M. Bashir, I. A. Qazi, Z. Uzmi, and F. R. Dogar, "Workload adaptive flow scheduling," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, 2018, pp. 241–253.

[3] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging zipf's law for traffic offloading," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 1, pp. 16–22, 2012.

[4] A. Lakhina, M. Crovella, and C. Diot, "Characterization of network-wide anomalies in traffic flows," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004, pp. 201–206.

[5] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "Heavyguardian: Separate and guard hot items in data streams," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 2584–2593.

[6] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[7] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 741–756.

[8] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[9] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 101–114.

[10] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International conference on database theory*. Springer, 2005, pp. 398–412.

[11] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 2002, pp. 346–357.

[12] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[13] M. Mahalingam, D. G. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks." *RFC*, vol. 7348, pp. 1–22, 2014.

[14] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina, "Rfc2784: Generic routing encapsulation (gre)," 2000.

[15] K. Wiles, "Pktgen-dpdk," in *https://github.com/pktgen/Pktgen-DPDK*, 2010.

[16] D. Intel, "Data plane development kit," 2014.