A Novel Space-Efficient Method for Detecting Network-Wide Heavy Hitters in Software-Defined Networking Using P4-Switch

Ali Alhaj
School of Computer and Information Sciences, University of Hyderabad
India
21mcpc16@uohyd.ac.in

Wilson Bhukya
School of Computer and Information Sciences, University of Hyderabad
India
rathore@uohyd.ac.in

Rajendra Lal
School of Computer and Information Sciences, University of Hyderabad
India
rajendraprasd@uohyd.ac.in

Abstract: *Software-Defined Networking (SDN) is a dynamic, programmable approach that enables centralized control and has become essential in modern networking environments such as data centers, Internet Service Providers (ISPs), and emerging 5G applications. A critical challenge within SDN environments is detecting and managing "heavy hitters" high-traffic flows often associated with malicious activities like Distributed Denial of Service (DDoS) attacks or real-time data-intensive applications. Identifying these flows across multiple network switches is complex due to constraints like memory limitations and processing accuracy. This paper proposes a novel, network-wide solution for detecting Heavy Hitters (HH), moving beyond the single-switch approaches found in previous research. In contrast, the new strategy introduces two algorithms to enhance detection. The first algorithm leverages the P4 programming language to identify the local Top-k heavy flows at individual P4-enabled switches. The second algorithm employs dynamic thresholding to efficiently combine the Top-k lists from multiple switches, creating a centralized, coordinated network-wide detection system. The proposed system was rigorously tested in an SDN environment utilizing P4 switches. The results show that it achieves a high detection accuracy (95%-100%) while using only 10KB of memory per programmable switch. Furthermore, the approach outperforms existing state-of-the-art methods, providing higher accuracy and lower error rates with minimal memory usage*.

Keywords: *Software-defined networking security, heavy-hitter detection, P4 switch, network monitoring, space-saving, sketches.*

Received June 14, 2024; accepted October 14, 2024
https://doi.org/10.34028/iajit/22/1/4

1. Introduction

Traffic monitoring in networks is critical to maintaining acceptable network Quality-Of-Service (QOS) [21, 32, 54]. Monitoring abnormal and deviated flow patterns is crucial to support various applications such as load balancing [33, 37], flow anomaly detection [25], and traffic engineering [5]. Two basic types of suspicious flows were focused on: The first type is elephant flows, more accurately called heavy flows, which are summarized as those in which the number of packets exceeds a specific limit or consumes a more significant amount of essential resources in the network. The second type is a heavy changer, summarized inflows that change significantly in volume or speed over a short time interval. The previous two types are generally referred to as Heavy Hitters (HH). Detecting HH helps mitigate the effects of attacks such as superspreader [19, 46] and Distributed Denial of Service (DDoS) attacks [1, 2, 40, 52]. Recent advancements in Software-Defined Networking (SDN) networks utilizing programmable switches [23] enable the execution of simple mathematical operations and the application of various algorithms to detect HH efficiently [4, 24, 45, 51].

With ever-increasing data packet size and flow speed, current flow measurement and monitoring approaches face the same three general challenges [18, 26]. Firstly, the switch memory is limited. Secondly, processing large volumes of flows is difficult according to the line rate. Thirdly, it is impossible to obtain accurate measurements based on a single switch, so Network-Wide Heavy-Hitters (NWHH) detection must be supported [14, 15, 17, 30, 50]. In recent years, many researchers have tried to study sketch-based streaming solutions with the emergence of programmable switches [6, 23]. Sketch is a data structure that uses limited persistent memory to collect statistics about network flows based on using several independent hash tables and mathematical operations to estimate the size of these flows. Sketch balances accuracy and resources compared to previous sampling solutions [10, 47]. However, sketches can be burdened for tracking all network flows, and it may be more efficient to focus on the Top-k volume flows or Top-heavy flows instead [43, 28]. While previous research has primarily focused on identifying prominent data streams within individual switches, there is a growing imperative for network administrators to extend their monitoring efforts to encompass the broader network landscape. This need becomes apparent in scenarios such as detecting port scanners [22] and super-spreaders [52, 55], where their activities could easily slip under the radar if traffic surveillance remains confined to a single location. Merely collecting the results from the nodes is insufficient because huge data flows can easily result in missed detections from multiple viewpoints. For example, one can reduce the detection thresholds at each switch. This will increase communication overheads. Additionally, sampling techniques are very often used, but accuracy is reduced, especially in high-traffic networks [36, 38].

This research paper introduces a novel method to deal with the NWHH problem in SDN based on two basic algorithms. The first is the Space-Efficient Algorithm (SEA), which uses the Top-k principle in the switch to collect local statistics, and the second works on the controller to merge local Top-k lists. Our approach leverages the capabilities of P4-based switches, which employ a programmable language meticulously tailored to define the operational characteristics of packet-forwarding devices. This unique attribute of P4 empowers us with precise control over the data plane. Building upon P4's programmability, we have developed an optimized and highly efficient algorithm to identify the most prominent HH within the network. Our innovative SEA algorithm extends the HashPipe framework [28, 43], enhancing memory efficiency and packet processing capabilities. The SEA achieves this by introducing an additional stage to the HashPipe structure. The initial stage utilizes Filtering to verify the presence of a pre-flow container to which an incoming data packet belongs. This modified algorithm ensures improved memory utilization by minimizing the duplicates and the ability to process packets quickly, thus optimizing overall performance.

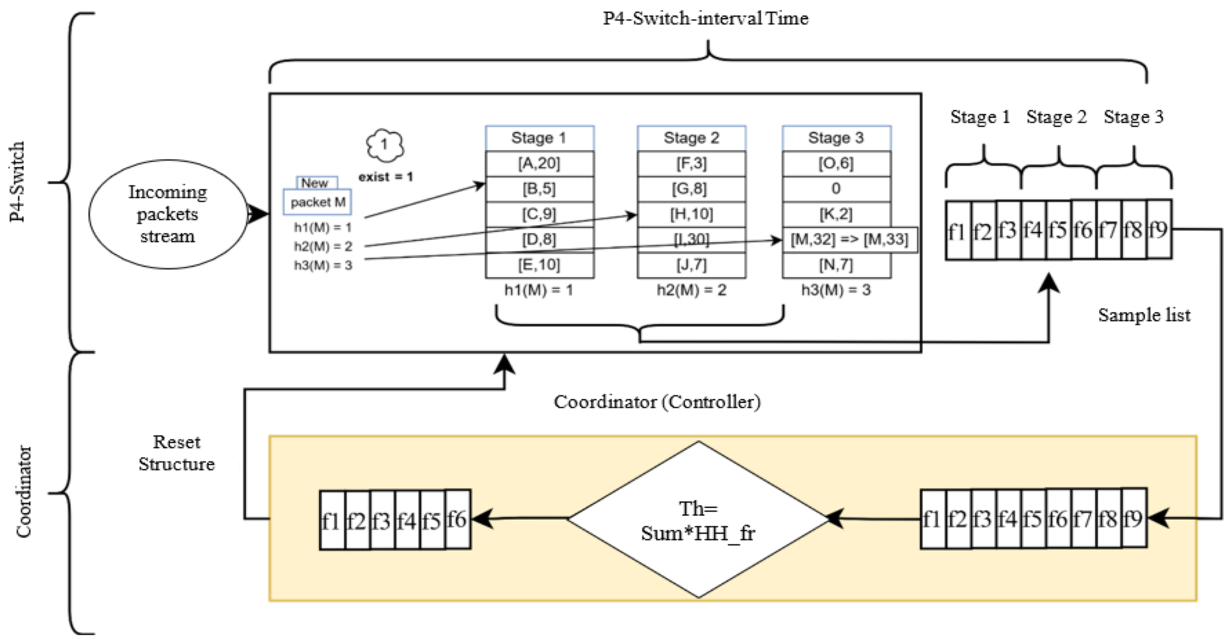


Figure 1. Proposed strategy for detecting NWHH.

Furthermore, we introduce another algorithm that facilitates the aggregation of Top-k HH lists from multiple switches. This algorithm operates at the coordinator level and dynamically adjusts the HH threshold in proportion to the sizes of the most critical flows within the network. Doing so effectively identifies the general HH, enhancing the network's visibility and adaptability. Our research significantly advances HH detection in SDN environments, improving memory efficiency, multi-switch coordination, and real-time responsiveness. These innovations pave the way for more efficient network management and the timely mitigation of network anomalies. Figure 1 illustrates a general scenario of the full proposed system. It implements the SEA algorithm on distributed programmable switches that act as monitors throughout the network. Each switch collects a local Top-k HH list and sends it to the central coordinator (SDN controller). The central coordinator uses a proposed merge algorithm for aggregating and determining the threshold value dynamically and derives a global HH list NWHH.

The previous research gaps can be summarized as follows:

1. The inability to detect NWHH effectively.
2. Inefficient utilization of limited programmable switch memory for accurate HH detection.
3. The need to manually determine the HH detection threshold. In contrast to existing state-of-the-art solutions, our approach excels with its ease of implementation, improved memory efficiency, and the ability to achieve higher accuracy with reduced memory usage.

The key contributions of research in this paper are as follows:

- SEA for programmable switches: we have devised an innovative algorithm tailored to programmable switches, an extension of the HashPipe algorithm. This advancement involves incorporating an additional stage within the HashPipe structure [43]. These additions serve to significantly enhance memory efficiency while effectively addressing the issue of data duplication.
- Multi-switch heavy flow collection and dynamic detection: we have introduced a sophisticated mechanism to collect heavy flows, representing the Top-k lists, from all switches within the network. Furthermore, we have implemented a dynamic detection threshold that aids in identifying flows that constitute the NWHH. This dynamic threshold adjustment enhances the adaptability and precision of our approach.
- Experimental validation and comparative analysis: our research is substantiated through a comprehensive implementation of P4 switches. Furthermore, we have conducted a thorough comparative analysis, benchmarking our solution against the most recent and relevant research in the field. This validation process underscores the effectiveness and competitiveness of our proposed methodology.

1.1. Background

In this section, we have defined the fundamental research problem, which we can divide into two primary areas: one related to determining the HH problem (within one network switch) and another associated with detecting the HH throughout the entire network NWHH.

1.1.1. Heavy Hitter in a Single Switch

Within the realm of data streams and the Top-k problem, a HH denotes an element that exhibits the highest occurrence within a specified stream window (interval time). The Top-k problem involves identifying K elements with the greatest frequencies or counts in a dataset.

- **Formal Definitions**
  - HH (threshold's version): within a data stream, let S represent the sequential arrival of elements(packets), and fi denote the frequency (flow size) of element i in the stream. An element i qualifies as a HH if its flow size surpasses a specified threshold, represented as $\phi \cdot n$, with n being the total number of packets. Mathematically $fi \geq \phi$ n, here, ϕ determines the threshold for HH [14].
  - HH (Top-k version): identifying the Top-k flows with the utmost flow size in the stream characterizes the Top-k HH problem. This challenge is commonly employed to pinpoint the most noteworthy or recurrent items within a defined context [29, 43]. The problem can be formulated as:

- **Given**
  - n: the total number of elements in the dataset.
  - fi: the frequency of occurrence of element i in the dataset.

- **Objective**

Identifying the Top-k flows by size (number of packets) passing on the link. Let H be the set of Top-k flows: $H=\{f1, f2, \ldots, fk\}$ where $f1 \geq f2 \geq \ldots \geq fK$.

The challenge of Top-k revolves around pinpointing K elements with the greatest frequency or count, and a HH is characterized as a noteworthy element based on its frequency within the specified context. Our objective is to devise an algorithm that addresses the Top-k problem by identifying HH or the K elements with the highest frequencies.

1.1.2. Network-Wide Heavy Hitters

NWHH is a networking and traffic analysis term that identifies the most notable or frequently occurring elements throughout the entire network. In the realm of network traffic, HH are commonly employed to denote sources, destinations, or communication patterns that substantially contribute to the overall traffic load on the network.

- **Formal Definitions**
  - Network traffics: pertains to the movement of data packets or messages within a computer network. It encompasses all digital communication and interactions between devices within a specified network, including computers, servers, routers, and other network-enabled devices.
  - NWHH: within the network, an element, be it a source, destination, or communication pattern, qualifies as a HH if its occurrence frequency in the network traffic surpasses a specific threshold [14, 45]. The problem can be formulated as:

- **Given**
  - Tint: Time interval.
  - θH: A HH fraction related to the network size.
  - P: The number of programmable switches distributed at several monitoring points in the network.
  - SLi: A Top-k list collected from programmable switch i. where $0<i<P$.

- **Objective**

Merge all local HH lists SLi to get a global NWHH list. The HH can be detected at the network level when the size of this flow exceeds a general threshold that is calculated dynamically. The grouping of similar flows that pass through more than one switch is considered, and these sizes will be combined. So, for each element (flow) i in the network denoted as fi, it is regarded as a NWHH if its frequency of occurrence exceeds the threshold $\phi*N$. Mathematically: $f_i \geq \phi*N$.

The goal of identifying NWHH is to identify sources, destinations, or communication patterns that substantially impact the overall traffic within a network. This data holds value for network administrators and analysts as it aids in optimizing network performance, detecting anomalies, and making well-informed decisions regarding network resource allocation.

1.2. Organization of the Paper

The remaining parts of this paper are organized as follows: Section 2 includes relevant research and the state-of-the-art. Section 3 talks about the research methodology, where the first part discusses the switch algorithm, and the second part addresses the coordinator (controller) algorithm. Section 4 includes the implementation of the switch algorithm, the coordinator algorithm, and our experiments with discussion. The last section, 5, consists of the conclusion and the most critical future proposals.

2. Related Work

In recent years, work has been done to find practical solutions for detecting heavy flows HH. Some primitive solutions relied on collecting statistics (samples of flows at a specific frequency rate), as in NetFlow [10] and sFlow [47], where the focus was on using limited resources to obtain somewhat acceptable accuracy. Numerous endeavors have been exerted to enhance the efficiency of network measurement by refining sketch algorithms [13, 16, 39]. In pursuing this objective, [12] introduced the Count-Min-Sketch (CMS), a technique to summarize data streams. The CMS mitigates hash collisions by selecting the smallest value among multiple counters to represent flow size. Regrettably, when an elephant flow and a mouse flow are mapped to the same bucket, the mouse flow tends to be significantly overestimated. Several algorithms have been devised to alleviate this issue by segregating elephant flows from mouse flows [20, 57]. Some research has focused on solutions to detect the HH within a single switch only locally, without taking into account the NWHH detection, as in algorithms based on Sketch alike [9, 49, 50, 53]. These sketch algorithms provide high accuracy using data structures with limited memory. Still, the problem with these sketch-based solutions is that they collect statistics on network flows. Still, we need additional cache memory for reverse recall and identifying the IDs (srcIP, dstIP) of flows that are likely to be HH. The problem increases with the possibility of duplicating and storing data of more than one flow in one entry or register, and this is what we solved by our structure, which maintains one entry for each flow with direct storage of the flow ID and its counter.

Other literature also focused on NWHH detection, such as MV-Sketch [45], which works on the Majority Vote Algorithm (MJRTY) principle to detect local heavy flows. MV-Sketch algorithm optimizes memory consumption by retaining heavy flows and evicting mouse flows. Additionally, it solved the sketch's lack of reversibility. In cases where one elephant flow's ID is dropped from the sketch (because of collision), they won't be re-evaluated, resulting in underestimation issues. MV-Sketch supports a NWHH detection mechanism based on collecting many local MV-sketch instances from several monitoring points. Then, form a final list that can be considered a global MV-Sketch of the NWHH. Research such as [19, 20, 50] and the Count-Min-Sketch Network-Wide Heavy-Hitters (CMS-NWHH) approach [14] proposed a solution to detect distributed HH using the CMS+cache algorithm to collect local flows in programmable switches. Here, we may need additional memory for backward retrieval but may lose some accuracy due to collision in the CMS structure. This CMS-NWHH research uses a NWHH detection mechanism by integrating several local HH lists and estimating a dynamic general threshold for wide-HH detection.

Our SEA algorithm architecture, which uses specific memory and maintains a lower collision rate, has addressed Sketch problems (collisions, backward retrieval). Inter-Packet Gap (IPG) [50] tried thinking out of the box using the IPG principle to detect heavy flows. IPG follows Heavy-Keeper's [51] strategy for detecting NWHH by using the mechanism of merging local heavy flow lists by the controller. Still, it depends on the type of stream flows used to obtain acceptable accuracy. Liu *et al*. [29], Li *et al*. [27], Zhou and Qian and [56] have tried to develop multi-tasking architectures, which store statistics about total flows in the network to be used at a later time to detect HH, DDoS, and super-spreader attacks.

3. Proposed Methodology

This section is separated into two main parts. The first deals with the programmable switch and the proposed algorithm for the switch, which solves the duplicate problem and uses the space-saving principle. The second part deals with the coordinator algorithm and the mechanism for collecting Top-k lists from all programmable switches. The coordinator uses a dynamic threshold mechanism to combine and show the final global HH list.

3.1. Single Switch Algorithm for Detecting HH

This part introduces an innovative real-time algorithm designed to gather the Top-k flows traversing a one P4-

switch. Our approach leverages the foundation of principles of the space-saving algorithm [11]. We extend the HashPipe [43] algorithm by incorporating a streamlined filtering mechanism to enhance its efficacy. This augmentation addresses issues related to duplicate data and significantly optimizes memory utilization.

3.1.1. Space Saving Technique

Space-saving entails an algorithm rooted in counter-based techniques, wherein merely k counters are deployed to monitor k-heavy flows. This innovative approach achieves the utmost economy in memory utilization while preserving a predetermined level of accuracy. This holds true in theoretical considerations [31], practical and real-world evaluations [11]. The space-saving algorithm, noteworthy for its ability to update just one counter per incoming packet, introduces the challenge of efficiently locating the item with the minimum counter value within the table. Regrettably, conventional methods such as exhaustive table scanning with each packet arrival or the swift identification of the minimum value in the table are not inherently supported by emerging programmable hardware. Furthermore, maintaining data structures like sorted linked lists [31] or priority queues [42] necessitates multiple memory accesses. This constraint must be managed within the confines of per-packet time constraints.

3.1.2. Multistage Hashpipe Algorithm

The algorithm offers a solution to minimize redundant packet processing within the switch pipeline, employing two fundamental concepts. Firstly, it employs a strategy of tracking a continuous minimum value. Packets progress by multi-pipeline; we keep tabs on the smallest counter value encountered thus far, along with its corresponding key. This information is seamlessly transmitted as packet metadata as it travels through the pipeline. Modern programmable switches permit the utilization of such metadata to convey processing outcomes between distinct pipeline stages. This metadata can be recorded at any given stage and subsequently employed for packet matching at a later point [43]. During the packet's journey through the pipeline, the switch performs hashing operations based on the carried key (metadata key) at each stage rather than hashing based on the key corresponding to the incoming packet. If a match occurs in the table or the matched stage slot is unoccupied, the slot or the slot's counter is updated directly, and the algorithm stops at this point. Conversely, if there is no match, the keys and count associated with the larger between the carried counter (metadata counter) and the one in the slot (stage input) are written back into the stage input, and the smaller is retained in the metadata. The algorithm leverages arithmetic and logical operations accessible in the match-action tables of emerging switches to execute the counter comparison. Depending on the packet's progress through the stages, the key may continue to the next stage or be entirely removed from the tables when it reaches the final stage.

Secondly, a consistent practice is adopted wherein new flows are invariably inserted in the first stage. If the incoming key is not found within the initial pipeline stage, there is no associated counter value for comparison with the key in that particular table. Consequently, the decision is made to consistently insert the new flow into the initial stage while simultaneously relocating the existing key and counter stored in the initial stage to the metadata. Following this stage, the packet can effectively monitor the rolling minimum value across subsequent stages using the above-mentioned conventional method.

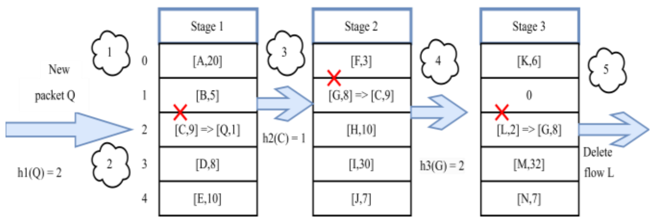


Figure 2. Multi-stages Hashpipe algorithm.

Figure 2 presents a straightforward illustration of the algorithm, featuring three stages (d=3). At each stage, an individual and autonomous hash function is applied, with each hash table containing a list of paired entries in the form of (key, count), each linked to a specific data flow. In this example:

1. Upon the arrival of a new packet denoted as Q, the H1 hash function is invoked.
2. This operation yields the retrieval of a record indexed at (2). The presence of a conflict in this record triggers the replacement of C with the incoming packet (Q, 1).
3. Flow C and its associated count are subsequently moved to the second stage, which is not empty in index 1. Consequently, a comparison ensues between C and G counters. The counter with the greater magnitude, C, is retained within the second stage's hash table. At the same time, the data flow with the smaller count is forwarded to the final stage.
4. This iterative process is reiterated in the third stage.
5. Finally, data flow L is purged from the data structure.

3.1.3. Space-Efficient Algorithm (SEA)

In reference to the Hashpipe algorithm, a significant result of consistently assigning incoming keys to the initial stage is the possibility of duplicate keys being spread across multiple tables in the pipeline. This situation arises because the key could reappear at the last stage within the pipeline. Acknowledging that such duplication is inevitable when packets follow a one-time passage through the pipeline is essential. Consequently, these duplicates might consume table space, reducing the available slots for high-volume flows. This could, in turn, lead to the eviction of these high-volume flows, with their counts distributed among the duplicate entries. Figure 3 explains the problem better, with the arrival of a packet belonging to flow L. 1-It will be inserted in the first stage in place of flow D. 2-then flow D will be transferred with the same algorithm mentioned previously. In the last stage, the flow will be divided into an empty space, and the algorithm will stop. The problem is that we will have two entries for the flow L in the first and third stages, thus wasting memory resources.

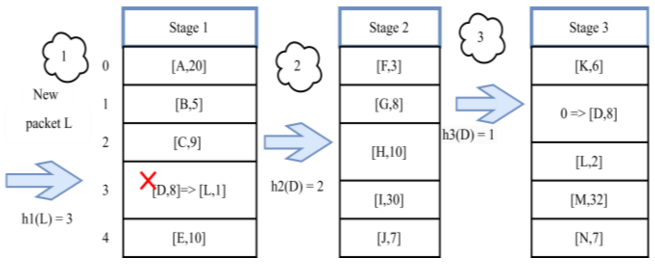


Figure 3. Multi-stages Hashpipe duplicates issue.

To address the persistent issue of duplicate data within the context of our research, we propose incorporating an additional stage, a filtering stage. This filtering stage draws inspiration from the Bloom-Filter algorithm [8, 44]. The filtering stage serves as the initial point of evaluation, where the primary objective is to ascertain whether an incoming packet corresponds to a flow already present within the multi-stage data structure. This process involves two distinct scenarios: Firstly, the verification process returns (exist=1) when a packet is linked with a flow already recorded at any stage. If we get (exist=1), an exhaustive search ensues within the following stages to pinpoint the entry of this particular flow, subsequently incrementing its respective counter. Secondly, if the incoming packet corresponds to a new flow not stored previously in the subsequent stages, the verification process returns (exist=0). In these instances, we implement the same Hashpipe algorithm previously outlined in section 3.1.2.

The main goal of our proposed space-efficient-algorithm is to store the identifiers of the Top-k heaviest flows and solve duplicates by ensuring one slot for each flow. The algorithm works in two different scenarios. The first scenario (exist=0) is shown in Figure 4:

1. When a packet with key L arrives. The filtering process is applied by applying the hash tables corresponding to each stage in our example (h1, h2, h3) for the flow key to which the incoming packet belongs, and upon detecting that there is no entry for this flow in
2. The multistage data structure (where each stage is linked to a separate register). Since this flow has no entry at any stage, the value exist=0 will be returned.
3. If the value exist=0, The first hash function will be calculated to give the value 3 in the first stage. Therefore, if record 3 is not empty, the current input [D, 8] will be replaced by the new flow input [L, 1] (The value of the corresponding entry in the first stage will always be replaced if exist=0).
4. The input [D, 8] will be moved to the second stage. The h2 function will be applied but obtain "non-empty" input. By comparing the "flow-D" counter and "flow-H" counter, we will keep the flow with the highest counter "flow-H" and move the smallest counter flow-D to stage 3.
5. By repeating the previous step 3 and getting an empty input, we will insert [D, 8] into this input.

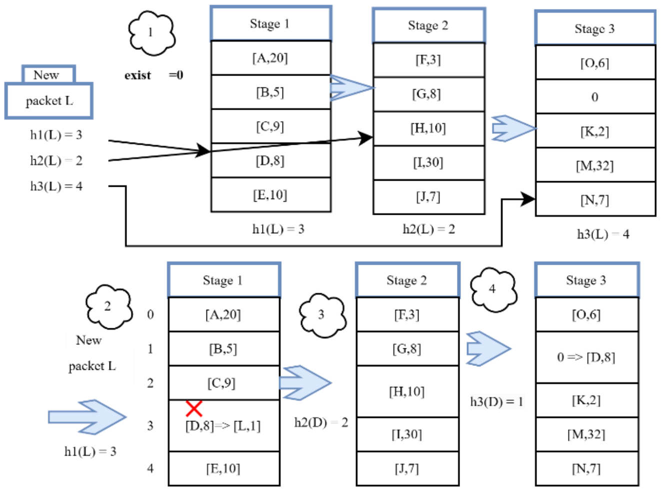


Figure 4. SEA when incoming packet doesn't belong to any existing flow.

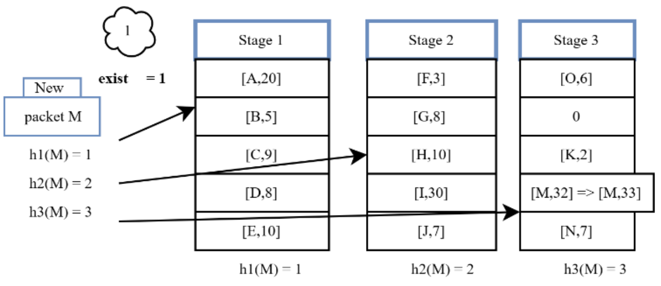


Figure 5. SEA when the incoming packet belongs to an existing flow.

The second scenario is shown in Figure 5:

1. We obtain the value exist=1 in the filtering stage, and therefore, the packet belongs to a flow previously stored in the multi-stage data structure.
2. In our example, we find one input for flow M in stage 3. The counter is updated only by adding a value of 1 to the counter.
3. The algorithm stops when an entry for the flow is found.

- **Proposed Algorithm for Programmable Switch**

Algorithm (1) shows the proposed syntax within the programmable switch. For each incoming packet, the flow key to which the packet belongs is calculated using a 5-tuple, and the counter is initialized with 1 (lines 3-4). After that, check whether the key was previously stored in the multi-stage data structure (line 5). For each stage in a multi-stage data structure, if exist=1, then the identifier has a previously stored input. This input will be searched in a stage, and the counter is incremented directly using the SEAFilter function, which will return the value 1 if there is a prior entry for the flow to which the incoming packet belongs and 0 if no entry for the flow is found in the multistage data structure (lines39-48). If exist=0, input will be added to the new flow using the insertToStage method (lines 7-13). The main function MultiStage returns the Top-k list where K is the size of the multi-stage data structure (stages.number ∗stage.length). The method insertToStage works for each stage in a multi-stage data structure. If this input is empty, the key for this flow with count=1 will be inserted directly into this input (lines 20-24). If the input is not empty, but the value of the following key equals the key in the input "S[index].key=m.key," then the counter will be incremented by 1 directly (lines 25-28). If the stage is the first stage and the corresponding input is not empty and does not contain the same flow key. The flow key will be inserted directly with the count=1 in the first stage (lines 29-32). If the input is not empty, it is not the first stage, and the key value does not match. The count for the portable flow will be compared with the count at the input, the flow with the larger count will be preserved, and the other flow Key will be transferred to the next stage (lines 33-36).

Algorithm 1: Space Efficient Algorithm.

Input: [p_1, p_2, .., p_r]-Packet stream within one interval time; S_i-Stage i Register; N-Number of stages; H[1..N]-List of hash functions
Output: List of the local Top-k flows

```
m[key, count] ← [0, 0]   ▷ Metadata carried values
Function MultiStage():
  for p in [$p_1$, $p_2$, ..$p_T$] do
    m.key ← 5− tuple(p); m.count ← 1
    exist ← SEAFilter(m.key)
     if !(exist) then
      for i ← 1 to N do
        index ← $H_i$(m.key)
        if insertToStage($S_i$, index, i) then
        Break;   ▷ End Process
      end
    i + +
   end
  end
end
 Top−k−list ← [$S_1$, $S_2$, ...$S_N$]
 return Top−k−list
end

Function InsertToStage(S, index, num):
  if S[index] is empty then
    S[index].key ← m.key
    S[index].count ← m.count
    return 1;   ▷ successfully insert
  end
  else if S[index].key = m.key then
    S[index].count ← S[index].count + m.count
    return 1:  ▷ successfully insert
  end
  else if (num = 1) then
    Swap(m, S[index])
    return 0;   ▷ move to next stage
  end
  else if S[index].count < m.count then
    Swap(m, S[index])
    return 0;   ▷ move to next stage
  end
 return 0 ▷ End process
end

Function SEAFilter():
  for i ← 1 to N do
      index ← $H_i$(m.key)
     if S[index].key = m.key then
        S[index].count ← S[index].count + m.count
       return 1     ▷ End process
    end
  end
 return 0    ▷ End process
end
```

- **Time Complexity**

We can analyze the time complexity as follows:

- *Function multistage*: the loop iterates T times (once for each packet). Inside another loop that iterates N times (number of stages). Inside nested loops, the function InsertToStage is called, which has a time complexity of O(1) since hash table operations are constant time. The time complexity of the loop is O(T.N).
- *Function InsertToStage*: the hash table operations (insertion, lookup, update) take constant time, O(1).
- Function SEAFilter: the function's time complexity is O(N) because iterates through the stages.

The algorithm's overall time complexity is O(T.N), Where N is the number of stages, and T is the number of incoming packets within one interval.

- **Space Complexity**

The primary data structure that contributes to the space complexity is the Multi-stage data structure, so the overall space complexity of the algorithm is $O(N \cdot L)$, where L is the length of the stage register (Si) and N is the total number of stages.

3.2. Algorithm for Detecting Network-Wide HH

This section introduces an innovative algorithm designed to aggregate Top-k lists from all programmable switches within the network over a designated time interval. Subsequently, these lists are merged using a dynamic threshold mechanism to derive the ultimate global HH flow list. This approach addresses a problem previously outlined in section 3.2. Each Top-k list can be viewed as a local HH list, representing significant flows within individual switches. The amalgamation of these lists, facilitated by our proposed algorithm, yields a comprehensive global HH list. The summation algorithm dynamically adjusts the global threshold based on the aggregate flow sizes within the local HH's lists. Collecting local HH lists from all programmable switches provides a holistic perspective of critical flows consuming network resources.

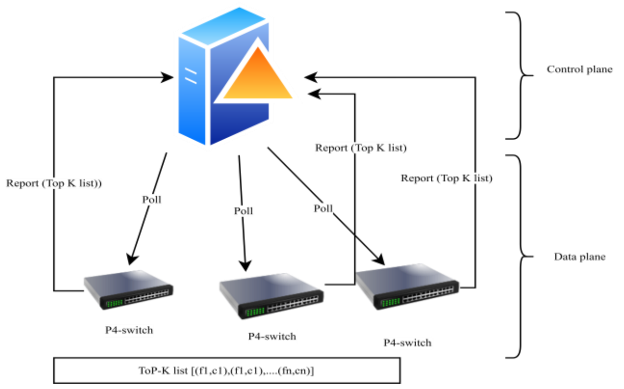


Figure 6. Interaction between P4-switches and coordinator for NWHH' detection.

In Figure 6, we illustrate the interaction between the central coordinator, also referred to as the controller, and network programmable switches distributed across multiple locations in the network. The time dimension is pivotal in this communication framework, with time being discretized into specific intervals. At the culmination of these time intervals, the controller initiates a poll signal, prompting all programmable switches to furnish their respective Top-k lists as in Equation (1).

$$SUM_{total} = \sum_{i=1}^{CL.size} fi \tag{1}$$

Where, Cl.size: global HH list Size, fi: flow i count which collected in Global HH list CL.

The algorithm will consider flows whose size will be estimated in more than one switch. A single flow can pass through more than one programmable switch. The controller estimates the global HH threshold *ThGlobal*, after collecting all the local HH lists using in Equation (2):

$$Th_{Global} = SUM_{total} * \theta_H \tag{2}$$

Where, θH: heavy hitter fraction.

After that, the controller will collect flows whose size exceeds the previous global threshold into the final HH's list and send this list as a report to the user (IT manager). After that, a new time period will be started, and all switch registers will be initialized.

3.2.1. Proposed Algorithm for Coordinator (Controller)

Algorithm (2) presents the syntax proposed for the coordinator. In this algorithm, the function getCollectionFlowList is responsible for gathering the Top-k lists from all programmable switches, assuming there are P programmable switches. If a prior entry exists for a flow, denoted as 'f' in the global HH list 'CL,' the new size will be aggregated with the previously stored count for 'f' as illustrated in lines (6-8). Should no prior entry be found for flow 'f,' a new entry is generated for that specific flow, as outlined in lines (9-12). The getFullSize method, detailed in lines (15-19), estimates the network size, which is defined as the sum of sizes for all local HH flows. The GetNHH function relies on the size of the Network SUMtotal to determine the threshold value dynamically. This method iterates through all the flows collected using the getCollectionFlowList function and adds flows whose sizes surpass the threshold to the final HH list, as depicted in lines (20-28).

Algorithm 2: Multi-Switch Heavy Flow Collection (Coordinator Algorithm).

```
Input: P - Number of p4-switches, $\theta_H$: Heavy Hitter fraction
  $SL_i$ : i(1..P ) - p4-switch Top-k list
Output: HHL-Final NWHH list

CL ← [0, 0, ...0]    ▷ List for collecting all SL
$SUM_{total}$ ← 0       ▷ To save the total traffic count.

Function getCollectionFlowList():
   for i ← 1 to P do
      for f in SLi do
        if f in CL then
          CL[f].count = Cl[f].count + f.count
        end
        else
          CL.add(f)
      end
    end
  end
end

Function getFullSize():
   for f in CL do
    $SUM_{total}$ + = f.count
  end
end

Function GetNHH():
  for f in CL do
     if f.count >= $\theta_H$ ∗ |$SUM_{total}$ | then
        HHL.add(f)
    end
  end
      ▷ return final NWHH
 return HHL;
end
```

The ultimate outcome of this algorithm is the comprehensive NWHH list. It is imperative to highlight that this algorithm operates at the end of each time interval, delivering a report on the identified heavy-hitter flows. This report serves as a crucial indicator, allowing network administrators to take requisite measures to mitigate the impact of these heavy flows on the network's overall performance. The potential actions may encompass load-balancing strategies or the selective dropping of suspicious flows designed to safeguard network resources from anticipated or ongoing attacks. By employing such a proactive approach, this algorithm effectively fortifies the network against potential threats, ensuring the robust management of network resources and the overall reliability of network operations.

- **Time Complexity**

We can assess the time complexity in the following manner:

- Function getCollectionFlowList: the nested loop runs P times for the outer loop and processes each flow in SLi once. The operations inside the loop have constant time (addition and updating counts in the list). Therefore, the time complexity is O(P.sizeof(SLi)).
- Function getFullSize: the loop runs over the Collecting List (CL), and the operations inside the loop are constant time. Therefore, the time complexity is O(sizeof(CL)).
- Function GetNHH: the loop runs over CL, and the operations inside the loop have constant time. Therefore, the time complexity is O(sizeof(CL)).
- The function getCollectionFlowList dominates the overall time complexity, so the algorithm's time complexity is O(P.size of SLi). The size of SLi (Top-k list) is K, so the final complexity is O(P.K).

- **Space Complexity**

The CL is the primary data structure contributing to space complexity. The space complexity is O(sizeof(CL)) for storing the CL. Hence, the overall space complexity of the algorithm is O(sizeof(CL)). Note that the maximum size of CL is the switch number multiplied by the length of SLi(Top-k list length).

4. Implementation and Evaluation

This section explains embedding our algorithm onto programmable switches and controllers (coordinator). Subsequently, we comprehensively elucidate the work environment and dataset utilized. Finally, we substantiate the efficacy of our algorithm by conducting a series of empirical experiments, comparing its performance with cutting-edge mechanisms in the field.

4.1. P4-Switch

Programmable switches in the data plane have been employed to implement our algorithm. We utilized P4 and harnessed the capabilities of the P4 behavioural model [34] to specify the functionality of P4 switches, encompassing aspects like parsers, tables, actions, ingress, and egress stages within the P4 pipeline. In P4-enabled switches, registers serve as stateful memory elements, allowing both read and write operations [58]. Broadly, our algorithm operates by implementing a match-action stage within the switch pipeline for each hash table. Within our algorithm, every match-action stage includes a default action representing the algorithm execution, which applies to every incoming packet. Each stage employs unique P4 register arrays tailored to the respective hash table. The register arrays preserve flow identifiers and associated counts for multi-stage data structures. Regarding hashing to sample locations, the P4 behavioural model [34] allows for the definition of custom hash functions. In our approach, we utilize hash functions CRC32 [3] of the form hi=(ai.x+bi)%p, where the selected ai and bi are chosen to be co-prime, ensuring the independence of hash functions across different stages. P is a large prime number sufficient to minimize collisions and ensure the effectiveness of the hi. To handle packet metadata for tracking the current minimum, we store the values retrieved from the registers within packet metadata. This is necessary because direct condition testing on register values isn't feasible in P4. This approach enables us to compute the minimum value between the carried key and the key in the table before writing it back into the registers. Additionally, packet metadata is vital in conveying state information, such as exist value (0 or 1) and the current minimum flow identifier and count, from one stage to another.

4.2. Coordinator Implementation

We developed an initial version of the centralized controller using Python. This controller can retrieve register data from switches through the straightforward switch CLI provided by the P4 behavioural model. The controller receives Top-k lists from P4 switches at the end of the time interval Tint. Subsequently, these Top-k lists from all p4-switches are consolidated into a comprehensive global CL. Ultimately, the controller can detect the NWHH's final list by applying the global threshold ThGlobal. In order to prepare for a fresh round of heavy-hitter detection, the controller resets all registers in P4 switches.

4.3. Experiments Setup

In this part, we provided a detailed explanation of the work environment, dataset, and benchmark.

4.3.1. Test-Bed and Environment

To carry out all experiments, we used a server with the specifications: Intel Core i5-8300H, 4GB NVIDIA GeForce GTX 1650, 8GB RAM, and (2Tera HDD-512GB SSD). This PC runs Ubuntu 22.04. We opted for Mininet [48] as our emulated network environment to implement P4 switches that execute NWHH strategies. To compile P4 code, we used the P4c tool [35], so the compiling result is a JavaScript Object Notation (JSON) document specifying the P4 switch's behaviour, including its parser, tables, and actions in the P4 pipeline. Subsequently, any P4 switch generated following the behavioural model [34] loads this JSON file. A Mininet topology is established to complete the setup, interconnecting the P4 switches defined by their behaviour. In our comparison, we used topology, which consists of three interconnected switches, each linked to a host, like the topology used by Ding *et al*. [15].

4.3.2. Datasets

Packet captures: We employ Center for Applied Internet Data Analysis (CAIDA)'s anonymous internet captures recorded on a 10-gigabit Ethernet link monitoring a USA city in January 2019 [7]. Each minute of this recorded data comprises roughly 64 million network packets. What concerns us about these flows is only the first minutes, as these flows will be divided according to different windows time in proportion to the applied experiments, as explained later. For example, windows time will be chosen to take different packet numbers (500, 10K, 100K, 150K, 300K, 400K, 500K...) within one interval.

4.3.3. Metrics

- **Precision** (*PR*): the ratio of correctly identified heavy flows to the total number of reported flows as in Equation (3).

$$PR = \frac{TP}{TP + FP} \quad (3)$$

Where True-Positive (*TP*) represents correctly identified HH flows. False-Positive (*FP*): refers to non-HH flows incorrectly detected as HH.

- **Recall** (*R*): the proportion of correctly identified heavy flows in relation to the total number of actual heavy flows as in Equation (4).

$$R = \frac{TP}{TP + FN} \quad (4)$$

Where FN (False Negative) indicates HH flows that are incorrectly identified as non-HH.

- **F1-score**: the *F*1-*score* represents the harmonic average of precision and recall. A higher *F*1-*score*, closer to 1, indicates a better overall result as Equation (5).

$$F1 - SCORE = \frac{2 * R * PR}{R + PR} \quad (5)$$

- **Throughput**: is defined as the rate at which packets are processed per second as Equation (6) measured in packets per second (pkts/s).

$$Throughput = \frac{Total\_N}{Time} \quad (6)$$

Where *Total_N*: total number of packets. *Time*: progressing time.

4.3.4. Benchmark

This paper compares our solution SEA-NWHH with several state-of-the-art solutions for detecting NWHH, such as CMS-NWHH [15], which uses a structure for switches. MV-Sketch [45] uses a new structure for switches based on (majority vote algorithm). IPG [41] is based on the principle of (inter-packet-gap). Hashpipe-NWHH, we merged the standard Hashplipe [43] algorithm with our coordinator algorithm. Table 1 shows our SEA algorithm's main simulation parameters and mechanisms compared to the state-of-the-art.

Table 1. The main simulation parameters and mechanisms for our SEA lgorit and the state-of-the-art.

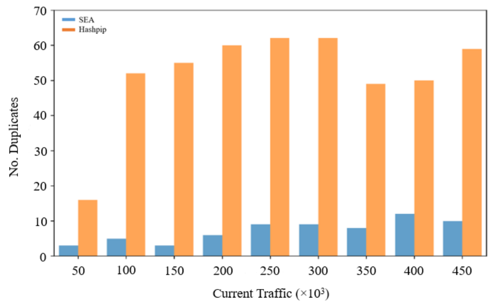
Mechanism	SEA-NWHH	CMS-NWHH	MV-Sketch	IPG
Switch algorithm	SEA (multi stage hashpipe +filtering)	Count-min-sketch+cash for local sample list	MV-Sketch-invertible sketch use (majority vote algorithm)	Heavy-Keepr+IPG mechanism
Memory	Rows∗columns∗96	CMS (Ns∗Nh∗32)+(sample list size∗32)	MV-Sketch size (R∗W∗(32+64+32))	(M∗72), M: Memory slots number
Coordinator mechanism	Merge multi-Top-k lists using dynamic threshold	Merge multi-sample lists using dynamic threshold	Merge muti MV-Sketches to large one (global list)	Using Heavy-Keeper mechanism [51]

4.4. Experiments and Results

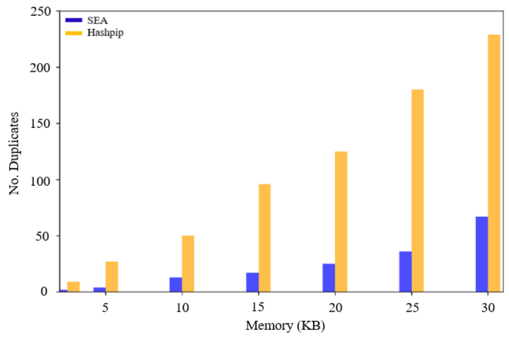
In this subsection, we completed a number of experiments, which first demonstrated the importance of our algorithm for minimizing duplicates. We then discussed the ability of our algorithm to detect the NWHH with high accuracy. The last experiments demonstrate the accuracy and effectiveness of our algorithm and how to improve this accuracy using different parameters.

4.4.1. Experiment 1 (Reduce Duplication Analysis)

Figure 7 illustrates the enhanced performance of our SEA algorithm in mitigating duplicate occurrences compared to the foundational HashPipe algorithm. As depicted in Figure 7-a), augmenting the flow count over time demonstrates minimal impact on the prevalence of duplicates within the multi-stage data structure. Notably, duplicates in HashPipe fluctuate between 50 and 70. In scenarios where the total memory allocation for the multi-stage data structure is 10KB, the duplicate-related overhead approaches 10% of the full memory capacity. Conversely, our refined SEA algorithm ensures that the memory loss attributable to duplicate occurrences remains below 1%. Furthermore, Figure 7-b) examines the ramifications of memory expansion, whereby an increase in memory entails augmenting the number of entries within each stage while maintaining a constant number of stages (i.e., 4). Such memory expansion substantially amplifies the incidence of duplicates within the HashPipe algorithm. Conversely, our SEA algorithm exhibits minimal susceptibility to this effect by increasing the number of entries allocated to statistics within each stage. This augmentation enhances accuracy and mitigates the occurrence of false negatives, as elaborated in the subsequent experiment.



a) Illustrates the impact of using fixed memory size on varying concurrent flow numbers (using multi-window values).

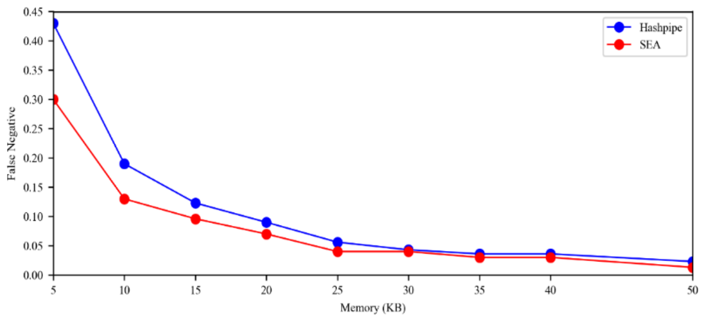


b) Examines the effect of a fixed number of flows (400k) with varying memory volumes.

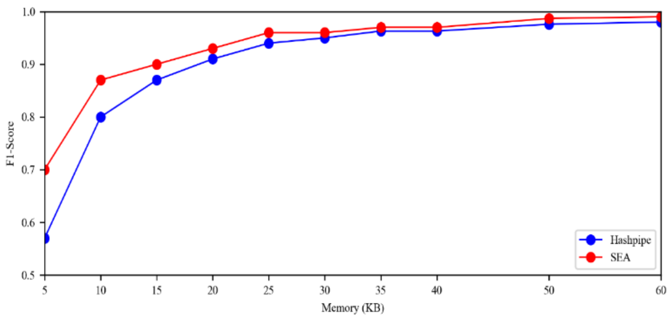
Figure 7. Comparison of duplicate occurrences between the standard Hashpipe algorithm and our SEA algorithm.

4.4.2. Experiment 2 (SEA Algorithm Efficiency-Comparing with Standard Hashpipe)

Figure 8 illustrates the efficacy of our SEA algorithm in reducing false negatives and improving the F1-Score compared to the HashPipe algorithm. As the memory allocation for both algorithms increases, the number of counters assigned to detect heavy flows also rises. In our experimental setup, we aimed to identify the top 300 heavy flows (k=300). Figure 8-a) demonstrates that increasing the memory allocation for our SEA algorithm reduces false negatives by less than 5% when the memory does not exceed 20K. Notably, our algorithm exhibits significant improvement over the HashPipe algorithm, which requires approximately 30K of memory to achieve a false negative rate of less than 5%. Furthermore, Figure 8-b) depicts the impact of memory expansion on the F1-Score. To attain F1-Score exceeding 95% with our SEA algorithm, a memory allocation between 15K and 20K is required, whereas the HashPipe algorithm necessitates over 25K of memory to achieve a comparable F1-Score exceeding 95%.



a) Compare false-negative using (k=300, trace contains 400k flows) with increasing memory.

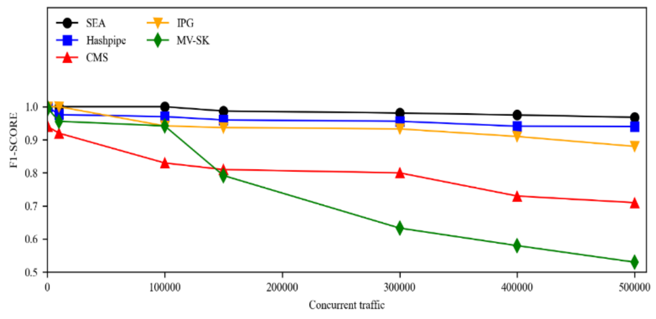


b) F1-score using (k=300, trace contains 400k flows) with increasing memory.

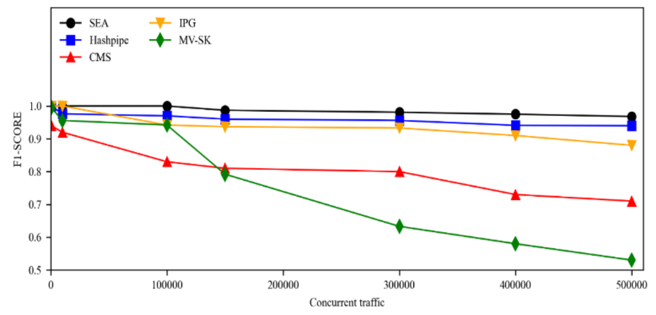
Figure 8. Compare accuracy between (our SEA, standard Hashpipe).

4.4.3. Experiment 3 (Accuracy for Network-Wide-HH Detection)

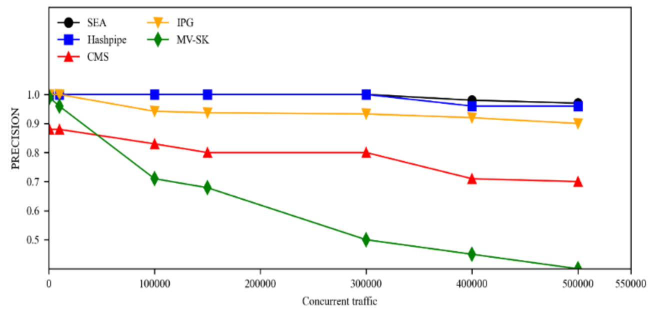
Figure 9 presents a comparative analysis of the accuracy achieved by five NWHH strategies: Our SEA-NWHH, Hashpipe-NWHH, IPG, CMS-NWHH, and MV-Sketch. The study considers various flow sizes while maintaining a fixed memory allocation of 10KB and a suitable threshold to detect 80 to 90 HH. The results depicted in Figure 9-a) indicate a consistent decline in accuracy across all five algorithms as the flow size increases within a given network interval. Specifically, Figure 9-a) illustrates that our algorithm achieves superior F1-score values compared to its counterparts, with scores ranging between (95%-100%). Notably, the discrepancy between our algorithm and the closest competitor, “HashPipe,” is evident. HashPipe demonstrates commendable F1-score values, remaining relatively unaffected by an increase in the number of flows. Conversely, other algorithms exhibit accuracy levels below 90% under the same memory constraint. This discrepancy can be attributed to the unique approach of our algorithm, which prioritizes the retention of high-size flows while disregarding smaller flows. Incorporating a filtering stage enhances memory utilization efficiency. In contrast, algorithms such as CMS and MV-Sketch maintain comprehensive network statistics, necessitating additional memory to accommodate sample lists. IPG, relying on probabilistic principles grounded in flow packet gap calculations, exhibits varying accuracy levels contingent upon traffic type. Further analysis reveals that our proposed network-wide strategy outperforms alternative algorithms in terms of Recall and Precision values, as evidenced in Figure 9-b) and (c). Our algorithm consistently achieves Recall values ranging between 94% and 100%. Noteworthy improvements are also observed in the modified HashPipe algorithm. By implementing a Top-k strategy within the switch, this algorithm effectively collects statistical data for local heavy flows while discarding less significant microflows. This is evident from the Precision values, which closely approach ideal levels for our strategy.



a) F1-score comparison.



b) Recall comparison.

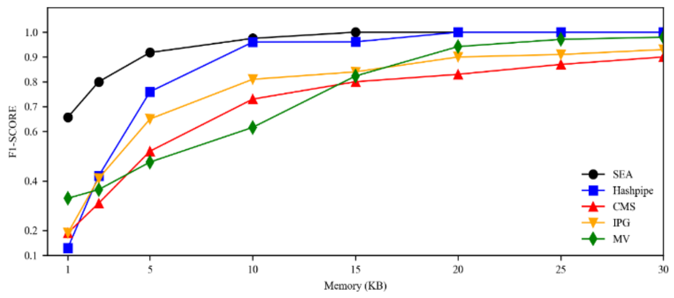


c) Precision comparison.

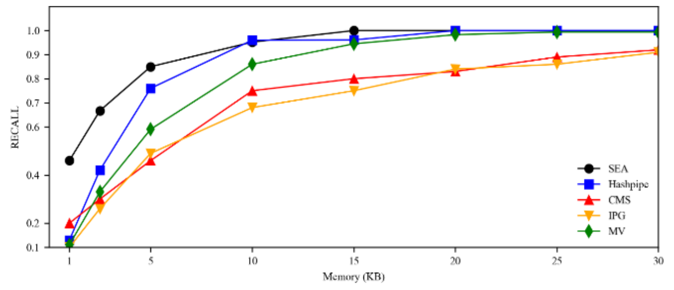
Figure 9. Accuracy comparison between (our Top-k NHH, MV-Sketch, IPG, CMS-NWHH) using multi concurrent traffic and fix-memory (10KB).

4.4.4. Experiment 4 (Memory Efficiency for Network-Wide HH Detection)

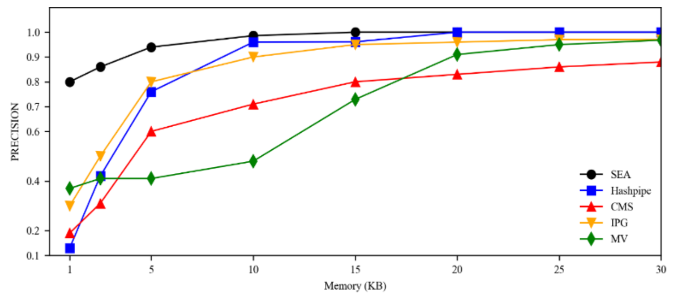
Figure 10 illustrates the impact of allocated memory on each switch for five algorithms, where increased memory size generally correlates with enhanced accuracy. Specifically, Figure 10-a) highlights the augmentation of F1-Score values with escalating memory allocation, underscoring the supremacy of our SEA-NWHH algorithm, particularly evident with memory exceeding 5KB. Notably, F1-Score values surpass the 98%-100% threshold following a memory allocation of 10KB. This enhancement is attributed to the augmented number of records per stage facilitated by increased memory, thereby bolstering the accuracy of hash tables and augmenting the capacity to accommodate additional flows. Consequently, the probability of retaining heavy flows within a multi-stage data structure is heightened. It is noteworthy that as mem-ory allocation surpasses 20KB, the accuracy of all five algorithms converges towards exceeding 95%. Figure 10-b) and (c) show the same results with slight differences for the Recall and Precision values, which approach 99% as memory increases. Our algorithm has some superiority over the CMS-NWHH algorithm with the value of Recall when the reserved memory size exceeds 15KB. The IPG algorithm is close to the results of our algorithm for Precision.



a) F1-score comparison.



b) Recall comparison.



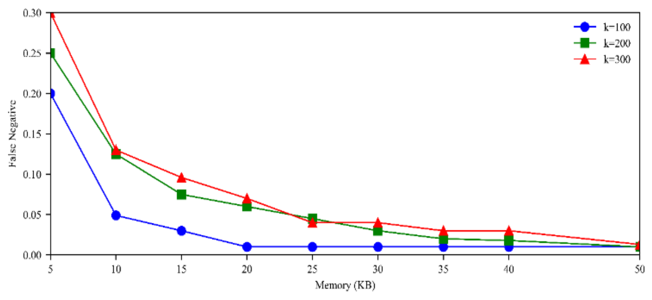
c) Precision comparison.

Figure 10. Accuracy comparison between (our Top-k NHH, MV-Sketch, IPG, CMS- NWHH) using fix traffic size and multi memory size.

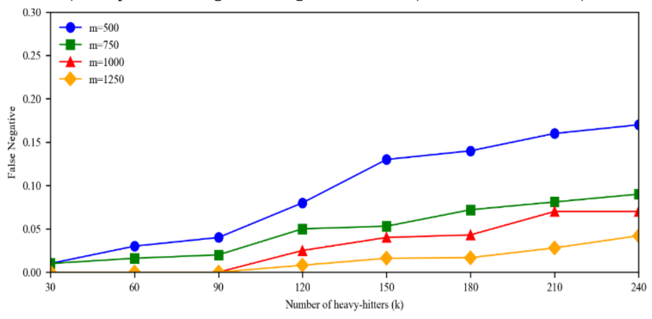
4.4.5. Experiment 5 (SEA Performance and Accuracy Analysis)

Figure 11-a) depicts the variation in false-negative rates concerning the number of heavier flows to be detected (K). We explored three K values (100, 200, 300) across incremental total memory allocations. It is evident that false negatives diminish with augmented memory size. However, to achieve high accuracy and maintain a false-negative rate below 5% for detecting the top 100 heavy flows, a total memory allocation of 10K proves sufficient. Detecting the top 200 or 300 HH necessitates a total memory allocation ranging between (30KB-35KB). Figure 11-b) elucidates how the negative rate is influenced by the number of inputs at each stage within a multistage data structure, with varying numbers of Top-k heavy flows targeted for detection. We examined four values for the number of inputs associated with each stage (m): 500, 750, 1000, and 1250. Notably, augmenting the number of inputs at each stage augments the accuracy of heavy flow detection. Therefore, the desired increase in heavy flows to be detected must align with the number of inputs at each stage to ensure high accuracy and a reduced false-negative rate.

Figure 12 presents the impact of augmenting the number of stages and the number of inputs in each stage on switch performance. Notably, it is observed that augmenting the number of inputs (m) exerts minimal influence on switch performance, evidenced by a convergence in values across a range of 10 to 1000 inputs per stage. Conversely, increasing the number of stages notably impacts switch performance and speed. As the number of stages increases, throughput values decrease, consequently reducing the number of packets processed within a specific interval. Furthermore, it is noted that maintaining high accuracy is achievable by augmenting the number of entries in each stage and adding a fixed, albeit small, number of stages. This approach ensures acceptable accuracy levels while adhering to memory constraints.



a) Compare false-negative using multi k value (k=100, k=200, k=300).



b) Compare false-negative using multi number of inputs per {500, 750, 1000, 1250}.

Figure 11. The effect of changing the number of detected HH (k) and the capacity of each stage (m).

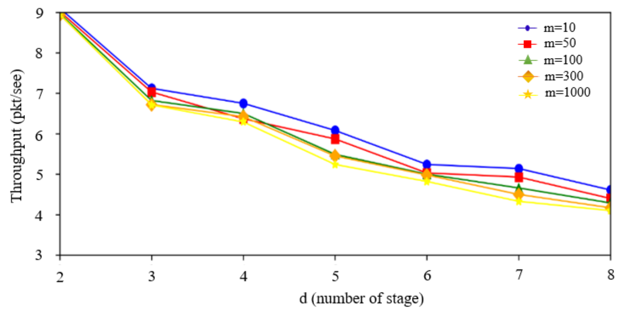


Figure 12. The effect of changing the number of stages [d] and the number of inputs per stages [m] on programmable switches throughput when applied SEA.

5. Conclusions and Future Work

This research proposes a new paradigm for NWHH detection in SDN environments. By leveraging P4-based switches, the proposed switch algorithm extends from the HashPipe framework, greatly enhancing memory efficiency and packet processing speed. This enhancement optimizes network performance while enabling real-time responsiveness in detecting Top-k HH.

Furthermore, this research developed a novel algorithm for coordinators that used a dynamic threshold to collect NWHH and achieve high network visibility and adaptiveness. The proposed approach differs in the ease of its deployment, much enhanced memory efficiency, and higher accuracy with lower requirements on memory resources compared to state-of-the-art solutions. This research contributes summarized in three key areas: developing a SEA for programmable switches, introducing a dynamic multi-switch coordination mechanism for heavy flow detection, and providing comprehensive experimental validation and comparative analysis. These advancements offer a more efficient and scalable approach to managing network traffic and mitigating potential anomalies in real time.

In future research, work can be done to suggest mechanisms to mitigate the communication overhead between switches and the central coordinator. Future work can also study programmable switch deployment at only strategic positions within the network instead of a full-scale deployment used in this work, further optimizing performance by doing better resource usage.

References

[1] Alhaj A. and Dutta N., *Contemporary Issues in Communication, Cloud and Big Data Analytics*, Springer, 2022. https://link.springer.com/chapter/10.1007/978-981-16-4244-9_3

[2] Alhaj A., Patel N., Singh A., Bondugula R., Dar M., and Ahamed J., "Design and Analysis of a Robust Security Layer for Software Defined Network Framework," *International Journal of Sensor Networks*, vol. 46, no. 1, pp. 1-14, 2024. https://doi.org/10.1504/IJSNET.2024.141613

[3] Bale A., Yadav K., Alam M., Shrivastava A., Varma R., Solanki R., and Savadatti M., "An Intelligent 64-bit Parallel CRC for High-Speed Communication System Applications," *International Journal of Intelligent Systems and Applications in Engineering*, vol. 11, no. 10s, pp. 543-551, 2023. https://www.ijisae.org/index.php/IJISAE/article/view/3310

[4] Basat R., Chen X., Einziger G., and Rottenstreich O., "Designing Heavy-Hitter Detection Algorithms for Programmable Switches," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1172-1185, 2020.

https://doi.org/10.1109/TNET.2020.2982739

[5] Benson T., Anand A., Akella A., and Zhang M., "MicroTE: Fine Grained Traffic Engineering for Data Centers," *in Proceedings of the 7th Conference on Emerging Networking Experiments and Technologies*, Tokyo, pp. 1-12, 2011. https://doi.org/10.1145/2079296.2079304

[6] Bosshart P., Gibb G., Kim H., Varghese G., McKeown N., Izzard M., Mujica F., and Horowitz M., "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99-110, 2013. https://doi.org/10.1145/2534169.2486011

[7] CAIDA: The CAIDA Anonymized Internet Traces Dataset, https://data.caida.org/datasets/passive-2019, Last Visited, 2024.

[8] Chabchoub Y., Fricker C., and Mohamed H., "Analysis of a Bloom Filter Algorithm Via the Supermarket Model," *in Proceedings of the 21st International Teletraffic Congress*, Paris, pp. 1-8, 2009. https://ieeexplore.ieee.org/document/5300252

[9] Cheng X., Jing X., Yan Z., Li X., Wang P., and Wu W., "Alsketch: An Adaptive Learning-Based Sketch for Accurate Network Measurement under Dynamic Traffic Distribution," *Journal of Network and Computer Applications*, vol. 216, pp. 103659, 2023. https://doi.org/10.1016/j.jnca.2023.103659

[10] Claise B., "Cisco Systems Netflow Services Export Version 9," Technical Report, 2004. https://www.rfc-editor.org/rfc/pdfrfc/rfc3954.txt.pdf

[11] Cormode G. and Hadjieleftheriou M., "Finding Frequent Items in Data Streams," *in Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1530-1541, 2008. https://doi.org/10.14778/1454159.1454225

[12] Cormode G. and Muthukrishnan S., "An Improved Data Stream Summary: The Count-Min Sketch and its Applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58-75, 2005. https://doi.org/10.1016/j.jalgor.2003.12.001

[13] Cormode G. and Muthukrishnan S., "What's New: Finding Significant Differences in Network Data Streams," *IEEE/ACM Transactions on Networking*, vol. 13, no. 6, pp. 1219-1232, 2005. DOI:10.1109/TNET.2005.860096

[14] Ding D., Savi M., Antichi G., and Siracusa D., "An Incrementally-Deployable P4-Enabled Architecture for Network-Wide Heavy-Hitter Detection," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 75-88, 2020. DOI:10.1109/TNSM.2020.2968979

[15] Ding D., Savi M., Pederzolli F., and Siracusa D., "Design and Development of Net-Work Monitoring Strategies in P4-Enabled Programmable Switches," *in Proceedings of the NOMS IEEE/IFIP Network Operations and Management Symposium*, Budapest, pp. 1-6, 2022. DOI:10.1109/NOMS54207.2022.9789848

[16] Fu Y., Li D., Shen S., Zhang Y., and Chen K., "Clustering-Preserving Net-Work Flow Sketching," *in Proceedings of the INFOCOM IEEE Conference on Computer Communications*, Toronto, pp. 1309-1318, 2020. DOI:10.1109/INFOCOM41043.2020.9155388

[17] Harrison R., Cai Q., Gupta A., and Rexford J., "Network-Wide Heavy Hitter Detection with Commodity Switches," *in Proceedings of the Symposium on SDN Research*, Los Angeles, pp. 1-7, 2018. https://doi.org/10.1145/3185467.318547

[18] Hu J., Min G., Jia W., and Woodward M., "Comprehensive QoS Analysis of Enhanced Distributed Channel Access in Wireless Local Area Networks," *Information Sciences*, vol. 214, pp. 20-34, 2012. https://doi.org/10.1016/j.ins.2012.05.013

[19] Huang Q., Jin X., Lee P., Li R., Tang L., Chen Y., and Zhang G., "Sketchvisor: Robust Network Measurement for Software Packet Processing," *in Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Los Angeles, pp. 113-126, 2017. https://doi.org/10.1145/3098822.3098831

[20] Huang Q., Lee P., and Bao Y., "Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference," *in Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Budapest, pp. 576-590, 2018. https://doi.org/10.1145/3230543.3230559

[21] Islam M., Al-Mukhtar M., Khan M., and Hossain M., "A Survey on SDN and SDCN Traffic Measurement: Existing Approaches and Research Challenges," *Eng*, vol. 4, no. 2, pp. 1071-1115, 2023. https://doi.org/10.3390/eng4020063

[22] Jung J., Paxson V., Berger A., and Balakrishnan H., "Fast Portscan Detection Using Sequential Hypothesis Testing," *in Proceedings of the IEEE Symposium on Security and Privacy, Proceedings*, Berkeley, pp. 211-225, 2004. DOI:10.1109/SECPRI.2004.1301325

[23] Kfoury E., Crichigno J., and Bou-Harb E., "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends," *IEEE Access*, vol. 9, pp. 87094-87155, 2021. DOI:10.1109/ACCESS.2021.3086704

[24] Kucera J., Popescu D., Wang H., Moore A., Korenek J., and Antichi G., "Enabling Event-Triggered Data Plane Monitoring," *in Proceedings of the Symposium on SDN Research*, San Jose, pp. 14-26, 2020.

https://doi.org/10.1145/3373360.3380830

[25] Lakhina A., Crovella M., and Diot C., "Diagnosing Network-Wide Traffic Anomalies," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 219-230, 2004. https://doi.org/10.1145/1030194.1015492

[26] Li H., Ota K., and Dong M., "LS-SDV: Virtual Network Management in Large-Scale Software-Defined IoT," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 8, pp. 1783-1793, 2019. DOI:10.1109/JSAC.2019.2927099

[27] Li L., Xie K., Pei S., Wen J., Liang W., and Xie G., "CS-Sketch: Compressive Sensing Enhanced Sketch for Full Traffic Measurement," *IEEE Transactions on Network Science and Engineering*, vol. 11, no. 3, pp. 2338-2352, 2024. DOI:10.1109/TNSE.2023.3305125

[28] Lin Y., Huang C., and Tsai S., "SDN Soft Computing Application for Detecting Heavy Hitters," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 10, pp. 5690-5699, 2019. DOI:10.1109/TII.2019.2909933

[29] Liu L., Ding T., Feng H., Yan Z., and Lu X., "Tree Sketch: An Accurate and Memory-Efficient Sketch for Network-Wide Measurement," *Computer Communications*, vol. 194, pp. 148-155, 2022. https://doi.org/10.1016/j.comcom.2022.07.009

[30] Liu Z., Manousis A., Vorsanger G., Sekar V., and Braverman V., "One Sketch to Rule them all: Rethinking Network Flow Monitoring with UnivMon," *in Proceedings of the ACM SIGCOMM Conference*, Florianopolis, pp. 101-114, 2016. https://doi.org/10.1145/2934872.2934906

[31] Metwally A., Agrawal D., and El Abbadi A., "Efficient Computation of Frequent and Top-k Elements in Data Streams," *in Proceedings of the 10th International Conference on Database Theory*, Edinburgh, pp. 398-412, 2005. https://link.springer.com/chapter/10.1007/978-3-540-30570-5_27

[32] Najm M., Patra M., and Tamarapalli V., "Cost-and-Delay Aware Dynamic Resource Allocation in Federated Vehicular Clouds," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 6, pp. 6159-6171, 2021. DOI:10.1109/TVT.2021.3079912

[33] Najm M., Tripathi R., Alhakeem M., and Tamarapalli V., "A Cost-Aware Management Framework for Placement of Data-Intensive Applications on Federated Cloud," *Journal of Network and Systems Management*, vol. 29, pp. 1-33, 2021. https://link.springer.com/article/10.1007/s10922-021-09594-9

[34] P4 Language Consortium: P4 Switch Behavioral Model, https://github.com/p4lang/behavioral-model, Last Visited, 2024.

[35] P4lang Consortium: P4C, https://github.com/p4lang/p4c, Last Visited, 2024.

[36] Phaal P. and Panchen S., Packet Sampling Basics, https://sflow.org/, Last Visited, 2024.

[37] Prabakaran S. and Ramar R., "Software Defined Network: Load Balancing Algorithm Design and Analysis," *The International Arab Journal of Information Technology*, vol. 18, no. 3, pp. 312-318, 2021. https://doi.org/10.34028/iajit/18/3/7

[38] Ran D., Chen X., and Song L., "ComPipe: A Novel Flow Placement and Measurement Algorithm for Programmable Composite Pipelines," *Electronics*, vol. 13, no. 6, pp. 1-19, 2024. https://doi.org/10.3390/electronics13061022

[39] Roy P., Khan A., and Alonso G., "Augmented Sketch: Faster and more Accurate Stream Processing," *in Proceedings of the International Conference on Management of Data*, San Francisco, pp. 1449-1463, 2016. https://doi.org/10.1145/2882903.2882948

[40] Singh A., "Machine Learning in OpenFlow Network: Comparative Analysis of DDoS Detection Techniques," *The International Arab Journal of Information Technology*, vol. 18, no. 2, pp. 221-226, 2021. https://doi.org/10.34028/iajit/18/2/11

[41] Singh S., Rothenberg C., Luizelli M., Antichi G., Gomes P., and Pongracz G., "HH-IPG: Leveraging Inter-Packet Gap Metrics in P4 Hardware for Heavy Hitter Detection," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 3536-3548, 2023. DOI:10.1109/TNSM.2022.3227065

[42] Sivaraman A., Subramanian S., Alizadeh M., Chole S., Chuang S., Agrawal A., Balakrishnan H., Edsall T., Katti S., and McKeown N., "Programmable Packet Scheduling at Line Rate," *in Proceedings of the ACM SIGCOMM Conference*, Florianopolis, pp. 44-57, 2016. https://doi.org/10.1145/2934872.2934899

[43] Sivaraman V., Narayana S., Rottenstreich O., Muthukrishnan S., and Rexford J., "Heavy-Hitter Detection Entirely in the Data Plane," *in Proceedings of the Symposium on SDN Research*, Santa Clara, pp. 164-176, 2017. https://doi.org/10.1145/3050220.3063772

[44] Song H., Dharmapurikar S., Turner J., and Lockwood J., "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 181-192, 2005. https://doi.org/10.1145/1090191.1080114

[45] Tang L., Huang Q., and Lee P., "A Fast and Compact Invertible Sketch for Network-Wide Heavy Flow Detection," *IEEE/ACM Transactions*

on Networking, vol. 28, no. 5, pp. 2350-2363, 2020. DOI:10.1109/TNET.2020.3011798

[46] Tang L., Huang Q., and Lee P., "SpreadSketch: Toward Invertible and Network-Wide Detection of Superspreaders," *in Proceedings of the IEEE INFOCOM IEEE Conference on Computer Communications*, Toronto, pp. 1608-1617, 2020. DOI:10.1109/INFOCOM41043.2020.9155541

[47] Wang M., Li B., and Li Z., "sFlow: Towards Resource-Efficient and Agile Service Federation in Service Overlay Networks," *in Proceedings of the 24th International Conference on Distributed Computing Systems, Proceedings*, Tokyo, pp. 628-635, 2004. DOI:10.1109/ICDCS.2004.1281630

[48] Xiang Z. and Seeling P., *Computing in Communication Networks, from Theory to Practice*, Elsevier, 2020. https://doi.org/10.1016/B978-0-12-820488-7.00025-6

[49] Xiao Q., Cai X., Qin Y., Tang Z., Chen S., and Liu Y., "Universal and Accurate Sketch for Estimating Heavy Hitters and Moments in Data Streams," *IEEE/ACM Transactions on Networking*, vol. 31, no. 5, pp. 1919-1934, 2023. DOI:10.1109/TNET.2022.3216025

[50] Yang T., Jiang J., Liu P., Huang Q., Gong J., Zhou Y., Miao R., Li X., and Uhlig S., "Elastic Sketch: Adaptive and Fast Network-Wide Measurements," *in Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Budapest, pp. 561-575, 2018. https://doi.org/10.1145/3230543.3230544

[51] Yang T., Zhang H., Li J., Gong J., Uhlig S., Chen S., and Li X., "HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845-1858, 2019. https://doi.org/10.1109/TNET.2019.2933868

[52] Yu M., Jose L., and Miao R., "Software {Defined}{Traffic} Measurement with {OpenSketch}," *in Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, Lombard, pp. 29-42, 2013. https://dl.acm.org/doi/10.5555/2482626.2482631

[53] Zhang Q., Xiao Q., and Cai Y., "A Generic Sketch for Estimating Super-Spreaders and Per-Flow Cardinality Distribution in High-Speed Data Streams," *Computer Networks*, vol. 237, pp. 110059, 2023. https://doi.org/10.1016/j.comnet.2023.110059

[54] Zhang X., Wang D., Ota K., Dong M., and Li H., "Exponential Stability of Mixed Time-Delay Neural Networks Based on Switching Approaches," *IEEE Transactions on Cybernetics*, vol. 52, no. 2, pp. 1125-1137, 2020. DOI:10.1109/TCYB.2020.2985777

[55] Zhang Z., Lu J., Ren Q., Li Z., Hu Y., and Chen H., "An Accurate and Invertible Sketch for Super Spread Detection," *Electronics*, vol. 13, no. 1, pp. 1-20, 2024. https://doi.org/10.3390/electronics13010222

[56] Zhou A. and Qian J., "An Adaptive Method for Identifying Super Nodes from Network-Wide View," *Journal of Network and Systems Management*, vol. 31, pp. 1-28, 2023. https://link.springer.com/article/10.1007/s10922-023-09745-0

[57] Zhou Y., Alipourfard O., Yu M., and Yang T., "Accelerating Network Measurement in Software," *ACM SIGCOMM Computer Communication Review*, vol. 48, no. 3, pp. 2-12, 2018. https://doi.org/10.1145/3276799.327680

[58] Zhu H., Wang T., Hong Y., Ports D., Sivaraman A., and Jin X., "{NetVRM}: Virtual Register Memory for Programmable Networks," *in Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*, Renton, pp. 155-170, 2022. https://nyuscholars.nyu.edu/en/publications/netvrm-virtual-register-memory-for-programmable-networks

Ali Alhaj is a dedicated PhD scholar at the University of Hyderabad from 2021. He holds an MTech in Cyber Security from Marwadi University, India, awarded in 2021, and a BTech in Systems and Networks from Tishreen University, Syria, conferred in 2018.

Wilson Bhukya is an Associate Professor at the School of Computer and Information Sciences, University of Hyderabad. Qualification: Ph.D. from University of Hyderabad. MTech in computer sciences, JNTU.

Rajendra Lal is an Assistant Professor at the School of Computer and Information Sciences, University of Hyderabad. Qualification: 2013 Ph.D. from Utkal University.