Switch-Aid: Accurately Identifying Heavy Hitter Flows

Shubham Pandey[1], Satya Sai Bharath Vemula[2], Rwitam Bandyopadhyay[3], Ali Nadim Alhaj [4] and Charan Ramtej Kodi[5]

[1,2,3]Department of Computer Science, Purdue University, West Lafayette, United States.
[4,5] School of Computer and Information Sciences, University of Hyderabad, Telangana, India - 500046.
[1]shubhambeethoven@gmail.com, [2]vemula.sai8@gmail.com, [3]rwitam@gmail.com, [4]ali.n.alhag@gmail.com, [5]21mcpc10@uohyd.ac.in

Abstract **Effective detection of heavy-hitter flows in data centers is essential for maintaining fair bandwidth allocation and identifying anomalous network behavior. Traditional data-plane heavy-hitter detection methods, designed for memory- and compute-constrained environments, rely on lightweight per-packet heuristics running on commodity switches. While these techniques achieve high processing speeds and low memory overhead, they often compromise accuracy. Recent advances in integrating machine learning (ML) models directly into the data plane enable line-rate packet processing without sacrificing precision. This paper proposes a set of ML-driven optimizations that emulate commonly used hash functions in sketch-based algorithms. These optimizations dynamically resolve hash collisions and cache invalidations through real-time inference. We evaluated our approach on the CAIDA dataset, demonstrating significant improvements in accuracy over traditional sketch-based methods. Our implementation is written in P4 and fully deployed in the data plane, supporting straightforward portability to programmable and bare-metal switching platforms.**

***Index Terms*— Programmable switches, Sketch-Based Algorithms, Data-Plane, Machine Learning, Heavy Hitters.**

I. Introduction

Due to the vast and complex nature of network infrastructure, anomaly detection and telemetry systems have assumed a significant role in ensuring the security and optimal performance of data centers and the ISP's backend [8]. With the advent of programmable switches [12], the network community is offloading several anomaly detection logics to the programmable data plane. This is done to increase the accuracy of anomaly detection and reduce latency in detecting anomalies. Also, this will reduce the controller overhead [19]. One such anomaly is the detection of heavy hitters [9, 10]. Heavy-hitter detection systems aim to identify flows that occupy significantly more bandwidth than other flows using the same link. This is useful for applications such as warning heavy network users, detecting super-spreaders [11], monitoring traffic, and balancing traffic load [13].

It is difficult to accurately detect heavy-hitting flows. Previously, data centers used a counter-based approach (for each flow, one counter) [14]. However, the number of concurrent flows is becoming so large that it's no longer feasible to deal with them in this manner. In the current state-of-the-art programmable switches, algorithms such as count-mean-sketch, sampling, and stream processing have become prominent for identifying heavy-hitter flows. The sampling base method [7, 15], on the other hand, regularly samples the network traffic and sends a copy of the sampled packet to the monitoring system. The monitoring server processes the sampled packets to detect heavy hitters. To achieve higher accuracy, the sampling method has to sample a large number of packets. This is not possible due to the high storage requirements to store packets before processing them.

On the other hand, to bypass storing packets and analyzing all packets, streaming-based systems [10] offload heavy-hitter detection logic to the data plane of a switch. These systems assume that network traffic is invariant to time and therefore use simple counters and thresholds to detect heavy hitters. As packets arrive, counters are updated to maintain a count of packets. If the count crosses a threshold, the flow is reported as a heavy hitter flow. After one time slot (eq., 3 seconds), the counters are reset. The thresholds are calculated during the initial setup, which can lead to inaccurate detection of heavy hitters as network traffic changes rapidly.

Finally, Sketch-based algorithms [16] utilize hash maps and counters to keep count of packets in a flow. Such systems overcome the need for thresholds but lead to high rates of false positives due to hash collisions. Our goal is to achieve high accuracy and reduce the latency of detecting heavy-hitting flows. In this report, we analyze a set of optimizations utilizing sketch methods to enhance accuracy. We hypothesize that by intelligently clearing counters in sketch algorithms, we can reduce the false positive rate of sketch algorithms. As time passes, this allows us to maintain an accurate count of the floor and avoid previous flows from creating bias in detecting future heavy-hitter flaws. We also look over the possibility of using machine learning for heavy hitter detection. Systems like Taurus have been out, which proposed machine learning blocks in the switch. We evaluate

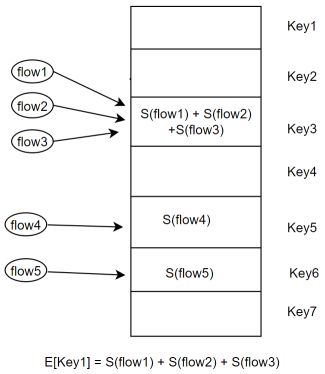

Fig. 1: Illustration of one hash counter usage.

the effects of our proposed optimizations and notice a 20 percent increase in accuracy in detecting heavy-hitter flows. The main contributions of this paper are summarized as follows:

- A machine learning-enhanced approach is introduced to improve the accuracy of sketch-based heavy hitter detection by reducing errors caused by hash collisions in Count-Min Sketch structures.
- A set of optimizations, including regression-based estimation and adaptive counter management, is designed to better handle dynamic network traffic patterns in the data plane.
- A fully data-plane implementation using P4 is developed, demonstrating practical feasibility on programmable switches with minimal overhead.
- Extensive evaluation on real-world CAIDA traces shows significant accuracy improvements over baseline methods, achieving up to 99% accuracy at higher percentiles.

II. Background and Related Work

In this section, we present several algorithms that can be used to detect heavy hitters. There are two possible definitions for heavy hitters:

- First, heavy hitters can be defined as the flows for which the size in bytes exceeds a fixed threshold.
- Second, heavy hitters can be defined as flows whose share exceeds a prescribed fraction of the traffic observed.

The two definitions are equivalent if the size of the total traffic is known beforehand. Studies have found that a small percentage of flows account for a large percentage of traffic. In this case, the heavy hitter identification problem in the context of network monitoring is essentially the frequent item identification problem with weighted input, as seen in the context of data stream mining.

A. One hash

In the context of network switches, a few billion packets pass through a switch every second, comprising at least a few million unique flows (identified by five tuples: source IP, source port, destination IP, destination port, and transport layer protocol). Considering the memory and computing resources available, it is impossible to store the information of each flow that passes through the switch. To address this resource constraint, sketches have been designed and proven effective for computing network metrics, such as calculating the top percentile flows. We introduce the most preliminary sketch using a limited-size hash table to compute the approximate size of the flows.

The method utilizes a limited-size hash set data structure (hash-based sketch) [1], where each flow is mapped to an entry in the hash set. The algorithm to map the packet is as follows: When a packet comes in, the algorithm takes the identity of the flow, which is the 5-tuples, computes a hash of the 5-tuples, builds a range from (0 to size_of_hash_table - 1), and uses this index value to map the corresponding flow to the index. Once it matches the index, it increments the value at the index by 1.

Theoretically, if the number of flows is equal to the size of the hash table, the size of each flow could be calculated accurately. However, since the number of flows that pass through a switch can be dynamic, the results of using this method are approximate rather than deterministic, as multiple flows may be mapped to the same index, and the estimated size of the flows may deviate from the actual ground truth. For example, in Figure 1, we show that for the hash-based sketches $flow1$, $flow2$, and $flow3$ get mapped to the same slot on the hash table, and the estimated size of each of these 3 becomes $size(flow1)$ + $size(flow2)$ + $size(flow3)$.

B. Sampling based methods

Sampling-based telemetry systems, such as Sflow [7], utilize counter-based sampling over incoming network traffic. This is done to prevent analysis of the majority of network traffic for anomalies, as it is prohibitively expensive in terms of memory and network bandwidth to do so at high speeds. The counters use statistical packet-based or time-based sampling to generate counter records, which are sent to the monitoring system for further analysis. The quality of such sampling systems depends on the sampling rate of the system. The higher sampling rate, the higher memory and network bandwidth required for analysis. Due to the absence of adjacent high-storage systems, a bottleneck appears, preventing these systems from providing accurate results. Due to this limitation, such systems are often referred to as approximate measurement systems. These systems are used to calculate the approximate number of packets/bytes of network traffic for a particular time window.

C. Streaming based methods

On the other hand, streaming-based telemetry systems [6] are capable of analyzing every packet at high speeds for anomaly detection. Their processing is often close

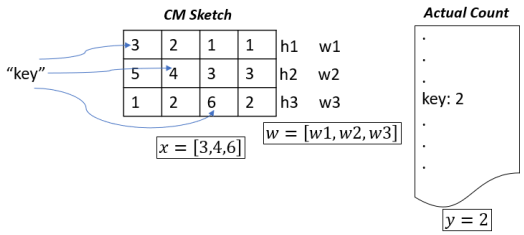

Fig. 2: Illustration of CM Sketch usage.

to the data plane as they try to offload processing onto the data plane itself. Even then, systems like Sonata [17] cannot mitigate the memory and bandwidth usage of switch-side operators without sacrificing accuracy. To keep up with the high speeds of network traffic, a query in a query plan (set of queries) operates on a streaming window of packets, the output of which is then fed into the next query that operates on the next available window of packets (ex, q_1 operates on W_1, q_2 on W_2, and so on. Almost all modern systems, whether server-based or switch-accelerated, employ this scheme. In server-based streaming systems (ex, Apache Stream), it reduces the storage overhead as packets are looked up only once, and it lowers the overall query plan completion times. Switch-accelerated systems further accelerate query plan execution speeds by offloading CPU work tasks to the switch, thereby improving throughput and latency by orders of magnitude. However, all these systems make an underlying assumption that executing queries of a query plan on distinct windows of packets in a network does not lead to an increase in inaccurate results. Considering as such is necessary, as it simplifies the interface and allows systems to naturally map onto the underlying streaming abstraction, naturally. However, assuming that each window is a statistically similar representation of another window does not hold in today's networks — microsecond-level bursts and intermittent failures can result in vastly different traffic distributions, even between two neighboring windows.

D. Count Min sketch

The Count-Min (CM) Sketch [2, 3, 4] is a compact data structure capable of representing high-dimensional vectors and answering queries on these vectors. In particular, point queries and dot product queries are supported with strong accuracy guarantees. Such queries are at the core of many computations, so the structure can be used to answer a variety of other queries, including frequent items (also known as heavy hitters), quantile finding, join size estimation, and more. Since the data structure can easily process updates in the form of additions or subtractions to vector dimensions (which may correspond to insertions, deletions, or other transactions), it efficiently handles such operations. It is therefore capable of processing high-rate streams of updates. The data structure maintains the linear projection of the vector with several other random vectors. These vectors are defined implicitly by simple hash functions. Increasing the range of the hash functions increases the accuracy of the summary, and increasing the number of hash functions decreases the probability of a bad estimate. These tradeoffs are quantified precisely below. Because of this linearity, CM sketches can be scaled, added, and subtracted to produce summaries of the corresponding scaled and combined vectors.

Figure 2 illustrates how a Count-Min (CM) Sketch works to estimate the frequency of a key in a stream of data. The CM-Sketch is composed of several hash functions (h_1, h_2, h_3) and their corresponding hash tables, w_1, w_2, w_3. Every incoming key is hashed into one cell per row; the counters at these positions are incremented correspondingly. The key, in this example, maps to positions $[3, 4, 6]$ across the three hash tables to provide corresponding counter values $x = [3, 4, 6]$. To estimate the frequency of the key, the algorithm takes the minimum of these counter values, i.e. $y = \min(x) = 2$, which approximates the actual count shown on the right. This mechanism enables the CM Sketch to efficiently estimate frequencies using limited memory while tolerating a bounded error due to hash collisions.

III. Optimizations

We believe that by using a set of optimizations, we can increase the accuracy of sketch-based solutions. One of the challenges of Count-Min Sketch is evicting older values. It can either use a hard reset after processing a set of packets, the least recently used implementation, or machine learning methods to reset counter values.

A. LRU Method

The Least-Recently-Used (LRU) [18] is one of the popular caching strategies. It defines the policy to discard the least recently used items first from the cache and make room for new elements when the cache is full. It is used to organize items in order of their use, allowing for the identification of items that have not been used for a long time. Our first implementation of the LRU algorithm uses LRU with a fixed cache size. This implementation resembles a simple eviction of older values after a fixed number of packets have been processed. Our second implementation involves checking the total benefits of LRU when the cache size constraint is removed. The goal of the second implementation is to assess the benefits of utilizing large cache sizes and dynamically adjusting the cache size.

Algorithm 1 gives the implementation details of the LRU policy in case of a fixed cache size. When a flow arrives (line 3), the algorithm first checks whether it is already in the cache. If the flow is found-(lines 4–10), its counter and timestamp are updated based on the packet threshold condition (lines 6–9). If the flow is not present and the cache is full - lines 11–13, it picks the least recently used flow and evicts it to allow the new

Algorithm 1 LRU with fixed cache size

```
1:  procedure LRU(flow, pckId, pckTheshold, cacheSize) ▷ flow is the 5-tuple, pckId is the timestamp or packet id in which the flow arrives, pckTheshold is the threshold that refreshes the LRU cache, cacheSize is the size of the LRU cache
2:      if flow in lruCache then
3:          values = lruCache[flow]
4:          count, prevPckId = values[0], values[1]
5:          if pkdId − prevPckId ≥ pckTheshold then
6:              lruCache[flow] =[1,pckId]
7:          else
8:              count = count +1
9:              lruCache[flow] =[count, pckId]
10:         end if
11:     else
12:         if len(lruCache) ≤ cacheSize then
13:             Select the least used flow flowLru
14:             delete flowLRU from LRUCache
15:             lruCache[flow] =[1,pckId]
16:         else
17:             lruCache[flow]=[1, pckId]
18:         end if
19:     end if
20: end procedure
```

one to enter. Otherwise, if space is available (lines 14–16), the new flow is inserted directly into the cache.

This approach maintains temporal locality but can easily lead to frequent evictions if the cache size is small.

Algorithm 2 LRU with expandable cache

```
    procedure LRU(flow, pckId, pckTheshold)  ▷ flow is the 5-tuple, pckId is the timestamp or packet id in which the flow arrives, pckTheshold is the threshold that refreshes the LRU cache
2:      if flow in lruCache then
            values = lruCache[flow]
4:          count, prevPckId = values[0], values[1]
            if pkdId − prevPckId ≥ pckTheshold then
6:              lruCache[flow] =[1,pckId]
            else
8:              count = count +1
                lruCache[flow] =[count, pckId]
10:         end if
        else
12:         lruCache[flow]=[1, pckId]
        end if
14: end procedure
```

Algorithm 2 enhances the basic LRU policy with dynamic cache expansions. Upon arrival (line 2), the algorithm checks if the flow is already cached. If the flow exists in the cache (lines 3–9), then it updates its counter and timestamp according to whether the packet threshold condition is met (lines 5–8). Otherwise, if this is a new flow, on line 10, one more entry is added to the cache without evicting any prior entries. This flexible design relaxes the stringent cache size constraint, thereby avoiding unnecessary eviction and better fitting the variability in traffic.

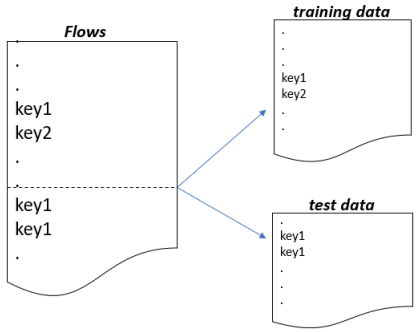

Fig. 3: Train-Test Split of Flows

B. Machine learning Method

We explore the possibility of using ML to enhance the performance of CM Sketch [5]. We formulate the output produced by CM-Sketch as a Linear Regression problem. There are two different ways to apply the weights. We discuss those two approaches in the following paragraphs.

The first approach is to populate the table by running the CM-Sketch algorithm for all records. While doing that, we also maintain the actual count of all the unique flows in a HashMap. After that, we formulate the ML problem as shown before. We go across each unique key and find the values in buckets where it gets hashed for each function. The collection of these values forms the input x, and the actual output y is taken from the hashmap containing the actual count. We associate weights w with each of the hash functions and want to learn them in a way that the product of x and w becomes closer to the actual value y. The first approach appears to be a solution to the actual problem we want to solve. However, it would fail to work in a real-world scenario where we have to deal with the flows dynamically, and there is no notion of a pre-computed CM Sketch, to begin with.

We address this issue with the second approach. In this case, we would sample only a subset of flows and build a CM Sketch from this limited information. Once we have built this table, we will apply the same ML algorithm as discussed earlier and get the corresponding weights for each hash function. This means that after the ML model has been trained, we create a new CM Sketch on the values from the test set to independently evaluate the learned model. By doing this, we ensure that test data does not impact the model's learning. In the first case, the ML model was trained on the complete dataset. This allows us to dynamically approximate the count of newer flows by simply obtaining the value of x and multiplying it by the calculated weights.

Methods	Storage footprint(MB)	Accuracy at 95 Percentile(%)	Accuracy at 99 Percentile(%)
Baseline			
One hash	256 MB	6.67	4.42
CM Sketch	256 MB	95.12	97.76
Optimization			
LRU	512 MB	66.33	97.67
Machine Learning	256 MB	99.56	99.12

TABLE I: Accuracies of baselines and optimizations.

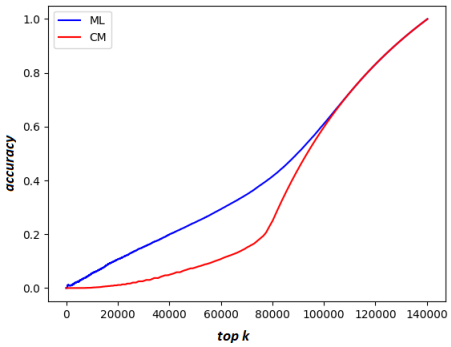

Fig. 4: Accuracy of the two models

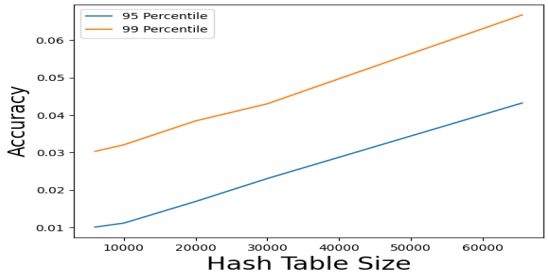

Fig. 5: CRC32 accuracies at different Hash Table Size at 95 and 99 percentile

The idea of sampling is effectively the same as dividing the original dataset into training and testing sets, as shown in Figure 3. We train the ML model using the training data and evaluate its performance on the test set. Figure 4 shows the plot of accuracy for both CM Sketch (CM) and ML approach (ML). x axis represents k in the top k flow, whereas y axis represents the percentage of flows that were correctly classified as top k by the model. It can be observed that, initially, for smaller values of k, the CM sketch is unable to perform well. However, when the information in CM Sketch is augmented with an ML model, we achieve better performance for small values. Eventually, both models start to perform equally well because the size of the *top k* list keeps increasing, and it eventually becomes 1 when all items are considered. We argue that the higher accuracy is more important for smaller values of k, where the CM sketch performs extremely badly, and the accuracy is very sensitive to the precision with which we compute the flow count. Hence, the use of a linear regression model helps achieve much better accuracy, even for small values of k.

IV. Preliminary Evaluations

For our quantitative study, we used a CAIDA PCAP File containing 85 million packets. Our global parameters of evaluation are the accuracy of heavy hitter detection, where thresholds are set at the 95th and 99th percentiles of the flow size and storage space needed for different configurations.

We calculate the accuracies I of the baselines using a Python implementation of the algorithms, and we have also implemented the algorithms in P4, utilizing the Tofino architecture. The ground truth was first calculated by analyzing all packets sequentially and calculating the packet count for each unique flow. Once that data was available, we ran One hash, CM Sketch, LRU, and machine learning simulations for querying Heavy Hitters across a wide variety of depth and width configurations. We also tested Heavy Hitter queries for a range of top-k percentile values.

A. One hash

We completed our experiment using 3 different hash functions (sha512, crc32, and md5) with percentile being 95% and 99%, and hash table size ranging from 6000 to 65536. We expect that three different hash functions should not show a drastic difference in accuracy. Table II shows the accuracies comparison of our experiment results.

Table II presents the accuracy for one hash algorithm for a table size of 65536. This highlights that general hash functions do not exhibit significant advantages over one another. One point of interest is the lower accuracy of the 99th percentile compared to the 95th percentile. This is because of a large number of false positives in the 99th percentile heavy-hitter list.

Figure 5 shows that CRC32 accuracies steadily increase as hash table sizes increase from 6000 to 65536 at both 95 and 99 percentiles (similar trends can also be observed for SHA512 and MD5 in our experiment results).

B. Count min sketch

The tests were run for the following configurations. The depth was varied as $2, 4, 8, 16$ consecutively. The width was varied as $6000, 10000, 20000, 30000, 65536$ respectively. The top-k percentile Heavy Hitter query was completely captured from 1 to 100 percentile, for the most granular details.

Hash Functions	Hash Table Size	95 Percentile Accuracy	99 Percentile Accuracy
sha512	65536	6.65036%	4.41515%
crc32	65536	6.67456%	4.32372%
md5	65536	6.60873%	4.39721%

TABLE II: Accuracy table of different hash functions

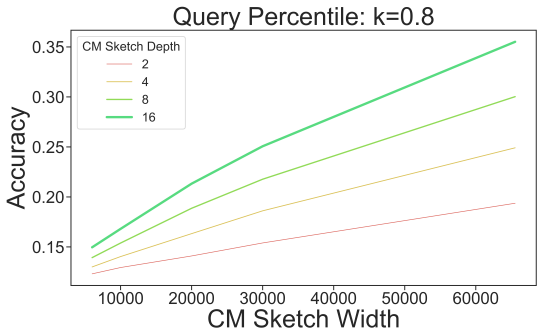

Fig. 6: CMS Width vs Accuracy

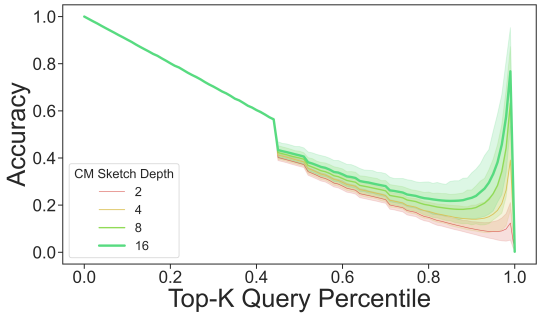

Fig. 7: CMS Top-K Query Percentile vs Accuracy

1) Evaluation 1 - Correlating Accuracy With CMS Depth, Width: We can see that query accuracy improves with increasing width and depth configurations of the CM Sketch, as it can capture more data.

However, some interesting findings were that lower percentile queries (< 60 percentile) resulted in very noisy trends, and performance varied significantly with increasing width, before finally reaching 100% accuracy (we assume there are no hash collisions at this point). The higher percentile queries (> 60th percentile) yielded smoother performance trends, characterized by increased width and depth. **Figure 6** shows the trends for the Top 80 percentile Heavy Hitter query.

2) Evaluation 2 - Correlating Top-K Query Percentile With CMS Depth, Width: In general, lower percentile queries are more accurate, while higher percentile queries are less accurate. However, to see exactly how they vary in granular detail, we captured top-k query percentiles ranging from 1 to 100 in increments of 1 and noted the accuracy for each. The data is shown in **Figure 7**. The solid lines represent the CM Sketch depths of 2, 4, 8, and 16. The spread around the line represents the width variations of the CM Sketch.

We see that the accuracy consistently drops to the 80th percentile mark, and then it shoots up drastically to the 99th percentile mark. However, proceeding to the 100th percentile query, we found that the CMS accuracy dipped very steeply. The reasoning for this would be that since the CM Sketch is a probabilistic data structure, querying 100% percentile Heavy Hitters from it would inevitably be less accurate.

3) Evaluation 3 - Correlating Accuracy With Memory Footprint: In this evaluation, we tried another exciting idea. Given a specific memory footprint, we can have multiple $depth \times width$ combinations for the CM Sketch. The motivation behind this experiment was to find out what combination gives us optimal accuracy. The observation here is that lower depths and higher widths provide more accuracy for a specific memory footprint.

C. LRU

Hash Set Size	95 Percentile Accuracy	99 Percentile Accuracy
2	16.81%	29.97%
4	34.18%	50.7%
8	51.7%	96.73%
16	66.3%	97.67%

TABLE III: Accuracies table of sha512, crc32 and md5 with hash table size 65536

We performed a parameter sweep to analyze the changes in accuracy. The study found that the LRU cache is not suitable for traffic measurement. This is due to constant changes in network traffic statistics. It is very difficult to use a fixed threshold for counter clearing, as it must balance clearing older flow counts while keeping heavy-hitting flow counts. Since we have no way to receive an alert when flows end, we have a highly variable threshold for packet counter clearing. Another reason for low accuracy was that the LRU cache had to remove large flows frequently to make room. This resulted in an increased false negative ratio.

Table III shows how the size of the hash set affects the accuracy of the LRU-based optimization. It is observed that accuracy increases with the size of the hash set, ranging from 2 to 16, and obtains up to 66.3% at the 95th percentile and 97.67% at the 99th percentile. This indicates that larger cache sizes retain more flow information and reduce errors caused by evictions.

However, even with increased cache capacity, the overall accuracy remains suboptimal compared to sketch-based and ML-enhanced methods. This arises because LRU caching struggles to adapt well to highly dynamic network traffic, where flow statistics are constantly changing. Since explicit detection of flow termination is not possible, using a fixed threshold for counter reset results in either early eviction of active flows or retention of outdated entries.

V. Conclusions

This paper implements the current state-of-the-art method, CMS, and a set of optimizations to emulate the hash functions typically used in Sketches and to utilize machine-learning inference to dynamically handle collisions/ cache revocations. We compared our proposed optimizations with the current state-of-the-art models in terms of accuracy. We show a 2-3 % increase in accuracy when we use a simple machine-learning model. Our second optimization of LRU showed a 1-2% decrease in accuracy due to rapid changes in network traffic statistics. For rapid prototyping and greater flexibility, we implemented the algorithms in Tofino p4 code, which runs on programmable switches and can be ported as-is to bare-metal systems.

References

[1] Yang, Tong, Zhang, Haowei, Li, Jinyang, Gong, Junzhi, Uhlig, Steve, Chen, Shigang, Li, Xiaoming. "HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[2] Yang, Tong, Wang, Lun, Shen, Yulong, Shahzad, Muhammad, Huang, Qun, Jiang, Xiaohong, Tan, Kun, Li, Xiaoming. "Empowering sketches with machine learning for network measurements," *Proceedings of the 2018 Workshop on Network Meets AI & ML*, pp. 15–20, 2018.

[3] Graham Cormode. "Count-Min Sketch," *Encyclopedia of Algorithms*, 2016.

[4] Cormode, Graham, Muthukrishnan, Shan. "An improved data stream summary: The count-min sketch and its applications," *latin american symposium on theoretical informatics*, pp. 29–38, 2004.

[5] Sapio, Amedeo, Canini, Marco, Ho, Chen-Yu, Nelson, Jacob, Kalnis, Panos, Kim, Changhoon, Krishnamurthy, Arvind, Moshref, Masoud, Ports, Dan. "Scaling distributed machine learning with {In-Network} aggregation," *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 785–808, 2021.

[6] Gupta, Arpit, Harrison, Rob, Canini, Marco, Feamster, Nick, Rexford, Jennifer, Willinger, Walter. "Sonata: Query-driven streaming network telemetry," *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pp. 357–371, 2018.

[7] Wang, Mea, Li, Baochun, Li, Zongpeng. "sFlow: Towards resource-efficient and agile service federation in service overlay networks," *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pp. 628–635, 2004.

[8] Alhaj, Ali Nadim, Dutta, Nitul. "Analysis of security attacks in SDN network: A comprehensive survey," *Contemporary Issues in Communication, Cloud and Big Data Analytics: Proceedings of CCB 2020*, pp. 27–37, 2021.

[9] Alhaj, Ali Nadim, Bhukya, Wilson Naik, Lal, Rajendra Prasad. "Advanced Method for Flow Frequency Estimation and Top-k Heavy Hitters Detection in SDN Leveraging Programmable Switch," *Proceedings of the 26th International Conference on Distributed Computing and Networking*, pp. 66–72, 2025.

[10] Alhaj, Ali, Bhukya, Wilson, Lal, Rajendra. "A Novel Space-Efficient Method for Detecting Network-Wide Heavy Hitters in Software-Defined Networking Using P4-Switch.," *International Arab Journal of Information Technology (IAJIT)*, vol. 22, no. 1, 2025.

[11] Alhaj, Ali Nadim, Bhukya, Wilson Naik, Lal, Rajendra Prasad. "A Space-Saving Technique in SDN for Identifying Top-k Super-Spreaders Using P4-Enabled Switches," *International Conference on Computing and Communication Networks*, pp. 659–673, 2024.

[12] Alhaj, Ali Nadim, Bhukya, Wilson Naik, Lal, Rajendra Prasad. "Adaptive network-wide superspreader detection using programmable switches," *AEU-International Journal of Electronics and Communications*, pp. 156041, 2025.

[13] Ke, Chih-Heng, Hsu, Shih-Jung. "Load balancing using P4 in software-defined networks," *Journal of Internet Technology*, vol. 21, no. 6, pp. 1671–1679, 2020.

[14] Cherian, Mimi, Varma, Satishkumar L. "Secure SDN–IoT framework for DDoS attack detection using deep learning and counter based approach," *Journal of Network and Systems Management*, vol. 31, no. 3, pp. 54, 2023.

[15] Li, Yuliang, Miao, Rui, Kim, Changhoon, Yu, Minlan. "FlowRadar: A Better NetFlow for Data Centers," *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 311–324, 2016.

[16] Liu, Lei, Ding, Tong, Feng, Hui, Yan, Zhongmin, Lu, Xudong. "Tree sketch: An accurate and memory-efficient sketch for network-wide measurement," *Computer Communications*, vol. 194, pp. 148–155, 2022.

[17] Dai, Kael, Hernando, Juan, Billeh, Yazan N, Gratiy, Sergey L, Planas, Judit, Davison, Andrew P, Dura-Bernal, Salvador, Gleeson, Padraig, Devresse, Adrien, Dichter, Benjamin K, others. "The SONATA data format for efficient description of large-scale network models," *PLoS computational biology*, vol. 16, no. 2, pp. e1007696, 2020.

[18] Zhang, Jinbei, Chen, Chunpeng, Cai, Kechao, Lui, John CS. "Incremental least-recently-used algorithm: Good, robust, and predictable performance," *IEEE Transactions on Mobile Computing*, 2025.

[19] Alhaj, Ali Nadim, Bhukya, Wilson Naik, Lal, Rajendra Prasad. "Efficient deployment of programmable switches for scalable network monitoring," *Computer Communications*, pp. 108470, 2026.