Finding Global Top-K Flows in the Data Plane

Eitan Stein
The Open University of Israel
Israel

Lior Zeno
Technion
Israel

Shir Landau Feibish
The Open University of Israel
Israel

***Abstract*—Monitoring network traffic is crucial for tasks like attack detection, pinpointing failures, and traffic engineering. A key metric is identifying the top-k *heaviest* flows. While programmable networks enable efficient top-k detection within a single switch, identifying global top-k flows remains a challenge. Current solutions rely on a centralized controller, introducing delays that hinder responsiveness to short-lived events. We present NODE, a fully in-data-plane algorithm for global top-k detection. NODE allows *switches* to aggregate network-wide flow data, while ensuring all switches hold the same global top-k table. It achieves over 95% recall on real and synthetic traces using under 300KB per switch.**

I. Introduction

Real-time network monitoring is essential for tasks like traffic engineering, failure detection, and threat mitigation [1]–[6]. A key task is identifying top-k flows, which can support better routing decisions to alleviate congestion and help to mitigate volumetric attacks such as DDoS [2], [7], [8]. Furthermore, with the advancement of programmable networks, solutions have been developed for monitoring the network inside the data plane. Finding top-k flows in a programmable switch is challenging due to the harsh constraints on memory and processing. There are existing solutions for finding top-k flows [9] within a single programmable switch, however some tasks require finding the *network-wide* top-k flows.

Existing solutions find network-wide top-k flows by first processing traffic at each switch, followed by collective processing. Sampling based methods, either on-switch [10] or using a collector [11], [12], but high overhead limits sampling rates (e.g., 1 in 30K [13]), reducing accuracy. Others perform local heavy flow detection in the data plane [14]–[17] and send results to a controller for aggregation. However, controller delays make this unsuitable for short bursts, and local top-k misses distributed heavy flows, requiring costly resources or communication. In order to find top-k flows only within the data plane, switches need a mechanism for sharing information. Swish [18] is a shared state management system in the data plane. Swish enables detection of network-wide heavy hitters fully in the data plane by distributing and merging local Count-Min Sketches (CMS) [19] across switches. This approach is an order of magnitude faster than using a centralized controller to obtain the combined sketches. However, CMS tracks only counters, not flow IDs, which are needed to extract flow estimates. Therefore, CMS is unsuitable for top-k, and a different approach is needed.

ISBN 978-3-903176-72-0 © 2025 IFIP

The NODE framework. We present NODE (Network wide tOp-k in the Data planE) a system for finding network-wide top-k flows completely in the data plane. NODE operates in three steps: (1) NODE creates a local top-k table in each switch independently; (2) using Swish, NODE exchanges the local top-k table from each switch to all other switches, in order to find the *global* counters for *locally heavy* flows; (3) NODE merges the aggregated information in each switch to build an identical global top-k table in every switch in the network. We implement NODE on a simulated testbed and in P4 code for the Intel Tofino switch [20], and evaluate its performance under various parameters, comparing it to controller-based approaches.

II. Background and Related Work

We provide the background needed for understanding NODE's design and functionality. We describe the processing restrictions of the data plane, and give a short overview of the existing solutions for finding top-k flows both in and out of the data plane, as well as methods for information sharing in the data plane.

Data Plane Restrictions. To process packets at Tbps speeds, network hardware imposes strict limits on memory and computation. Programmable switches (like PISA [21]) use a feed-forward pipeline with a few stages, each with limited local memory accessible only during that stage. Packets can't access memory from other stages, and memory access per stage is minimal. If additional processing on the packet is required, the packet may be be recirculated, but excessive recirculation can hurt throughput.

Finding Top-K Flows in the Data Plane. Several solutions have been proposed for finding top-k flows and heavy hitters. We describe some of them as they will help in understanding NODE's design choices.

Space Saving. Space Saving [22] is a well known technique that maintains a subset of the items in the stream and a counter for each item. When a new packet with ID x is processed, if x is in the subset, its associated counter is incremented by 1. If x is not saved in the subset, the algorithm finds the member with the *smallest* counter (i.e. $MinCount$) in the subset and replaces that member with x and increments its counter by 1.

RAP. Random Admission Policy [23] expands on this idea but, upon seeing an ID that is not in the subset, instead of *always* replacing the entry with the lowest counter, it will replace it with a certain probability (the exact probability is $1/(MinCount + 1)$). This cancels excess counter increases by low frequency flows and thus reduces the flow count overestimation.

HashPipe. Space Saving and RAP can find top-k flows but aren't suited for the data plane, as they require sorting or scanning all counters to find the minimum, which are operations that exceed available memory access limits. HashPipe [24] was the first data-plane-friendly solution, avoiding global minimum searches by using d vectors with separate hash tables across stages to maintain a rolling minimum. When each packet reaches the hash table in the j-th vector, it is hashed to check if its ID matches the ID in the vector. If the IDs match, the associated counter is incremented. Otherwise, it compares counters and if the packet's counter is larger, it replaces the packet's values with the vector values. However, HashPipe does not meet the requirements to run on programmable switches [9]. Since it requires both accessing the ID before the counter to decide whether to increment the counter, but also requires to access counters before IDs to compare the counters to decide whether to switch the packet ID and the saved ID.

Precision. In order to create a top-k algorithm that can run on programmable switches, Precision [9] builds on both RAP and HashPipe by using a similar structure to HashPipe's table while handling packets similarly to RAP. When each packet reaches the hash table in the j-th vector, it is hashed (just like in HashPipe) to check if its ID matches the ID in the vector. If the IDs match, the associated counter is incremented. If a match is not found in any of the vectors it will recirculate to replace the smallest observed counter with probability $1/(MinCount + 1)$. This way there are no excessive recirculations. Note that Precision can run on programmable switches since it always first checks the ID and only then handles the counter.

Controller based network-wide top-k. Several solutions have been proposed for finding network-wide top-k flows [10], [14]–[16]. Some of them include data plane analytics for finding top-k flows, however these solutions depend on either pulling or pushing data to a centralized controller which builds the network-wide top-k list and sends this list to all switches. FlowRadar [17] is a known controller based algorithm for finding network-wide heavy hitters. FlowRadar encodes per flow information in the switch and sends this array frequently to the controller which decodes the flows and sends back network-wide information. Due to FlowRadar's frequent controller updates, it is likely to find all flows, and especially heavy flows as long as it has enough memory and may pose substantial communication overhead.

Information Sharing in the Data Plane. Many scenarios require sharing data between switches in the network. Most solutions either share only limited amount of data [25], or use the control plane for assistance [14]–[17], [26] as we have seen before. Sharing large amounts of data between switches *completely* in the data plane is more challenging as it requires both communicating the information in the face of errors and processing the information with the limited resources of the data plane.

Using Swish to exchange data between switches in the data plane. Swish [18] mechanism enables full data-plane shared state management across switches. It allows replicating and sending data between switches without having to rely on a central controller. Swish guarantees that all packets will be delivered successfully without duplicates. Swish can also handle failures, and in order to do so, it must send the data from a static source, so it can re-send the information if necessary. We wish to find network-wide top-k completely in the data plane in order to create the table faster as seen in the comparison between Swish and a centralized controller [18].

III. Challenges

In order to create a global top-k detection algorithm in the data plane, each switch in the network needs to first identify the locally heavy flows, and then merge this information with the heavy flows identified by each of the other switches in the network. However there are significant challenges in achieving this in the data plane:

Memory restrictions. A straightforward solution for merging the network-wide information, would be to have each switch broadcast its local top-k table to all other switches. Each switch would then store all of the tables and then process them all together. However, memory and computational abilities are very limited, such that each switch cannot simultaneously hold *all* of the received data from *all* the other switches and process all of the data at once. Instead, each switch needs to process the information from other switches, as it is received, in a streaming fashion.

Split heavy flows. To address this issue, let's consider another simple method where each switch maintains a local top-k flow table and shares it with others. Upon receiving a flow ID and counter from another switch, a switch checks for a match in its own table. If found, it adds the counter; if not and the counter is larger than the smallest in its table (within memory constraints), it replaces the smallest entry with the new one to retain heavier flows. Yet, this solution is inherently flawed. A heavy flow could be split across the network into small pieces, which may be missed with this solution. When attempting to merge the global information, if the flow isn't saved in the local switch table, each small piece of the flow could be considered as a small flow and be discarded in favor of flows that appear to be heavier.

Memory access dependencies. Another issue with the above solution is that, similarly to HashPipe [24], it requires both checking the ID first to find out if the ID is in the table, and checking the counter first to check if its larger. That is, on the one hand we wish to first compare IDs in order to see if the IDs match and then aggregate their counts, which requires accessing the IDs earlier in the pipeline, before handling the counters. But on the other hand, we might want to remove flows with a low count from the table (a packet with counter 1000 should be able to replace a saved flow with counter 100), but that requires to first compare counts and only then handle the IDs. However, if the IDs come before the counters in the pipeline, once we reach the counters the IDs can't be accessed without re-circulation.

Using recirculation to solve this can be problematic, as table entries may change during the process. For instance, a packet might decide to replace flow x (counter 100) with flow y (counter 500). But while it recirculates, another update could raise x's counter to 1100. The packet would still overwrite x, unaware of the change. Precision [9] avoids this issue since it increments counters by at most 1, so the smallest entry is unlikely to change significantly during recirculation.

IV. The NODE Framework

To find network-wide top-k flows in the data plane, we designed NODE—a fully in-data-plane solution that shares the needed information between switches to create an identical global top-k table in every switch. We now overview NODE followed by a detailed description.

Overview. To solve these challenges, NODE takes a deterministic approach and splits the process into 3:
1) Creating a local top-k table. Each switch processes the incoming packets to create a *local* top-k table.
2) Global counter aggregation. Each switch sends the contents of its local top-k table to all other switches. Each switch then aggregates the counters *only* for flow IDs that are *already found* in the *local* top-k table to find their *global* counters. In this way, the switch finds the global counts of the top-k flows in its *own* table.
3) Consolidation of all tables. Then, each switch sends the contents of its local top-k table, with the aggregated counts to all other switches. Each switch filters the flows with smaller counters, such that only the flows with the higher counters remain in the table. Once this process is completed, each switch holds a global top-k table.

Information sharing between switches is done completely in the data plane, using the Swish framework [18], which does not require any controller interaction. Swish guarantees that all information is sent exactly once in a single Swish pass (assuming no failures). We address the challenges described above as follows:

Memory constraints. NODE doesn't require saving all the information from every switch. Instead, NODE processes each packet that arrives from each switch in a streaming fashion (i.e., in a single pass).

Handling split flows. By performing global counter aggregation and then consolidating the tables, NODE uses the *global* information to calculate the global counter of each flow in order to identify the top-k flows. That is, when NODE performs the consolidation round between the switches (step 3 above), each packet will hold a flow ID and its *global* counter. Therefore, when NODE consolidates the tables it considers the entire flow count and therefore it doesn't need to worry about making decisions with only partial information.

Order of accessing IDs and counters. NODE uses different table layouts for each stage of information sharing (steps 2 and 3): for global counter aggregation, IDs precede counters; for table consolidation, counters are placed before the IDs. This lets NODE first match IDs, then update counters when calculating global counters for local flows. Later, during table consolidation, NODE first compares counters and only then modifies the ID in the table if the stored counter is smaller than the packet's counter. NODE avoids recirculation entirely for inter-switch packets, sidestepping its associated issues.

Detecting Global Top-K. As shown in Fig. 1 NODE maintains five tables in each switch. All of the tables use the same hash functions - that way a packet with ID x will get hashed to the same locations in every table. We will now describe how NODE uses these tables to find the global top-k flows, using the example shown in Fig. 1. Note that Fig. 1 has a single vector per table for simplicity, in a normal setting each table will have multiple vectors, maintaining pairs of IDs and counters.

NODE takes two parameters: 1) the number of vectors d in *each* of NODE's tables (which is also the number of hash functions used). Note that d is identical across all NODE tables, with the same hash function for each vector; 2) the size s of each vector, indicating the number of $(ID, count)$ pairs stored per vector, which is consistent across all vectors and tables.

Creating a local top-k table. NODE first finds the local top-k flows. Every packet that traverses the switch is processed by the local top-k algorithm (e.g. Precision as it can run on programmable switches and creates the same table format that NODE uses), and the flow ID is inserted into the Local top-k table accordingly. This table *continuously* maintains the top-k flows. In Fig. 1, switch 1 has flows f_1,f_2,f_5 in its local top-k table, while switch 2 has f_1,f_3,f_4 in its local top-k table.

Global counter aggregation. During each switch's Aggregation round, NODE shares its local top-k data with all other switches. To ensure consistency, it first copies this data into a static *Snapshot* table. This table is sent in parts, and thus remains *unchanged* throughout the

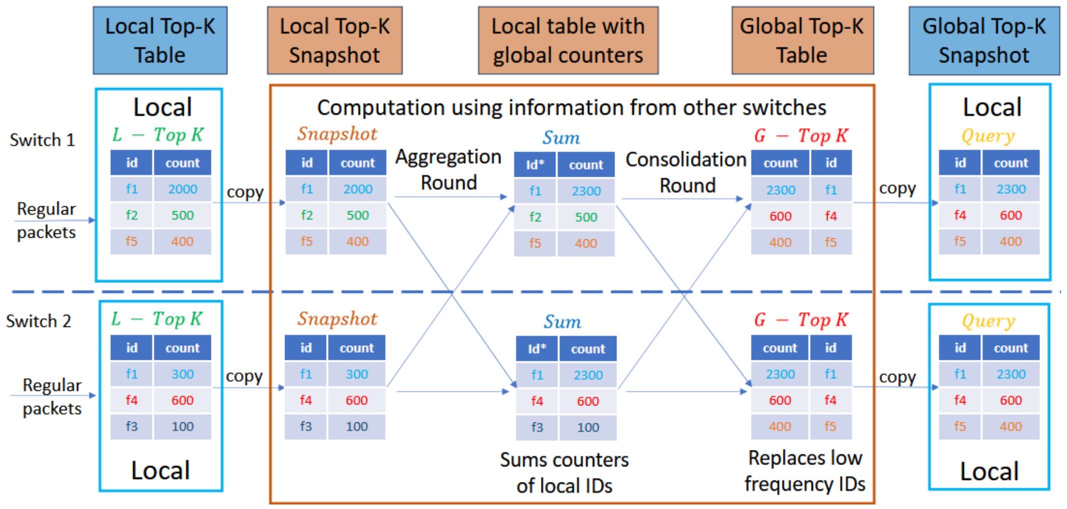


Fig. 1: An example of NODE flow. The Sum table uses the same IDs as the Snapshot table.

global top-k computation to avoid inconsistencies caused by the partial transmissions and potential packet loss. Since the local top-k table is continuously updated with incoming packets, a static copy is necessary for reliable data sharing. As shown in Fig. 1, the Snapshot table mirrors the local table just before the Aggregation round begins. In order to sum up the global counters of local flows, NODE uses another table called Sum, since the local top-k table is constantly updated and the Snapshot must remain unchanged for possible retransmissions. As shown in Fig. 1, the Sum table is separate for clarity but reuses the same flow IDs as the Snapshot, while maintaining its own counts for the global totals. To aggregate global counts, the Sum table is initialized as a copy of Snapshot. When a switch receives an ID and count from another switch, it checks if the ID exists in its *Snapshot* table. If it does, the count is added to the corresponding entry in the Sum table. For example, if Switch 1 sends flow f_1 with a count of 2000 to Switch 2, which also has f_1, it adds 2000 to its own count, updating Sum to 2300. If the ID isn't found, it's ignored (e.g., f_4 sent from Switch 2 to Switch 1 is discarded). After all Aggregation packets are received, Sum holds the global counts for each local flow. It becomes static, allowing it to be shared without creating a copy.

Consolidation of all tables. Once all counters of each of the flows have been aggregated, NODE is ready to consolidate the tables and determine which flows are the global heavy flows. In each switch, NODE sends the data from the *Sum* table to all other switches. We call this the Consolidation round. NODE uses a fourth table, $G - TopK$, which stores counters *before* IDs to compare values first. It starts empty and treats entries from the Sum table, as incoming packets. For each packet, it hashes the flow ID to a location, compares counters, and replaces the entry (both count and ID) if the packet's counter is larger. The replaced ID and counter are then used to probe the next slot, continuing this process through the table. As shown in Fig. 1, assume each switch starts *G-TopK* with the values from its own Sum table. For example, in switch 1, f_4 and f_2 hash to the same slot, but f_4 has a higher count, so it replaces f_2, and the packet continues with $(f_2, 500)$ to the next vector (if $d > 1$). Similarly, in switch 2, f_5 replaces f_3 in their shared slot, and the packet proceeds with $(f_3, 100)$. If the incoming counter is smaller, no replacement occurs—e.g., f_3 doesn't replace f_5 in switch 1, and f_2 doesn't replace f_6 in switch 2.

To ensure identical global top-k results across switches, NODE compares IDs when counters match. If the packet's ID is larger, it replaces the stored ID; if equal, processing stops to avoid duplicates. As shown in Fig. 1, both switches send f_1 with the same count, but only one copy is kept. After the Consolidation round, each switch holds the global top-k flows in *G-TopK*.

Note that *G-TopK* filters low-frequency flows based on their hashed locations. In Fig. 1, f_2 has a higher count than f_5 but is filtered because they hash to different slots and compete with different flows. With larger tables and more vectors, heavy flows are less likely to be filtered. At the end of the Consolidation round, all switches hold an identical global top-k table (see Fig. 1). This holds because packets with the same ID always carry the same count, and each (ID, count) pair can only be placed in one vector. Full proof omitted for space.

Finally, NODE uses a fifth table, *Query*, which snapshots *G-TopK* at the end of each cycle for querying. After completing one global top-k iteration, NODE starts a new one to stay updated. Like Swish, it detects when each round ends and transitions between Aggregation, Consolidation, and the next cycle accordingly.

V. Evaluation

We evaluate NODE for various performance metrics, and compare NODE to controller based approaches. We show that NODE achieves a recall rate of over 95% of the top-k flows with at most 288KB of memory for NODE's tables. Furthermore, we implemented NODE in P4 code for the Intel Tofino Wedge-100 programmable switch [20], using $\approx$ 2000 lines of code, and show the resource usage of the switch.

System Setup. We simulated networks of up to 100 switches using Python and C++ to model NODE. All evaluations used the Precision [9] algorithm for local top-k, implemented within NODE. Each table has two vectors ($d = 2$) with a uniform size s across all switches and identical hash functions per table. We chose $d = 2$ based on Precision's strong performance with two vectors and for memory efficiency. All tests used $K = 128$, with other parameters varied, and results are averaged over at least five runs.

Datasets. We used two types of traces: 1) Synthesized traces with different Zipfian distributions, each containing 100M packets and 10M unique flows. The distributions used to generate the traces were a=0.6, 0.8 and 1.0. 2) Real traffic from the CAIDA UCSD Anonymized Internet Traces (2018, 2019) [27]. To simulate large network traces we merged consecutive CAIDA traces.

Splitting the Stream. To challenge NODE , we split the stream so that the top 128 flows are evenly distributed across all switches, while the rest are assigned to specific switches. This makes it harder for NODE , as top flows may be missed in local top-k tables. We also tested partial affinities (e.g., 50% or 80%) and found that higher affinity improves accuracy, as local tables better estimate counts and aggregation matters less.

NODE Performance. Our key observation is that larger network sizes generally have worse results than smaller network sizes, however, given a large enough table size (larger s) NODE performs well. That said, in some cases the larger network actually performs better, we suspect that this is due to the fact that the overall saved information across the entire network is larger, enabling NODE to monitor more flows and thus get better results. Fig.2 shows results across traces. On a single switch, NODE behaves like Precision [9], since there is no network-wide information to merge. In comparison, even with a lot more switches, despite harsher flow splits, NODE performs well, likely because more switches create a larger effective memory. Additionally, as we observe results with increasing Zipfian distributions we can see NODE doesn't need a large table to reach a high recall with Zipfian distribution of 1.0. The figure also includes results that use clustering, where NODE runs within sub-networks and again across cluster reps; full analysis is omitted for space.

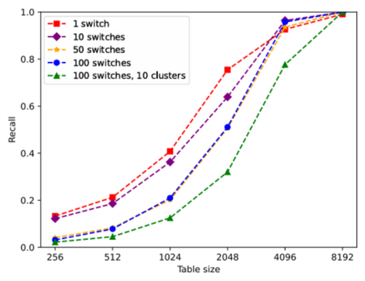
Comparing Memory Usage. We compared NODE with a basic approach, where each switch maintains a local top-k table, and periodically exchanges this data with the controller, which aggregates it to compute a network-wide top-k table that it then sends back to each switch. The controller implementation is not limited in memory or computation and may thus maintain all of the collected information and process it in a non-streaming manner. The comparison of how well they identify global top-k is shown in Fig. 2e, 2f. As shown, NODE achieves a recall that is very close to that achieved by the controller despite the fact that it functions within the confined resources and processing capabilities of the data plane. We also compare NODE to the controller based solution of FlowRadar [17]. Instead of attempting to emulate FlowRadar and potentially making a weaker version of it, we compare NODE to the results shown in the paper [17]. FlowRadar shows that even with perfect hash tables with no collisions it requires more than 2MB in each switch to support 100k unique flows, and over 20MB in each switch to support 1M unique flows.

In our evaluation, the largest table in NODE uses 8192 cells (two vectors of 4096), with 8-bytes for each cell (4 bytes for ID, 4 for count), totaling 64KB. So 4 tables use 64KB each, and Sum uses 32KB (since IDs are shared with Snapshot), for a total of 288KB per switch. In the synthetic streams we used 10M different flows and in the merged Caida dataset there are almost 6M different flow IDs. In both, NODE was able to achieve good results with only 288KB memory in each switch, which is significantly less than that required by FlowRadar.

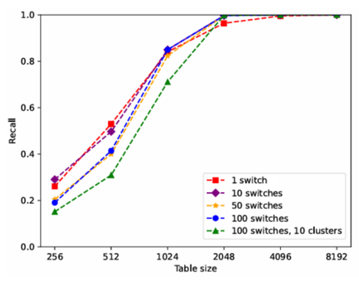
NODE's Resource Usage. We implemented NODE in P4 for the Intel Tofino Wedge-100 programmable switch [20]. Our P4 prototype uses $d = 2$ vectors per table with 8192 cells total ($s = 4096$ per vector), matching simulation settings. NODE uses all 12 switch stages, with tables ordered to support its logic (e.g., Snapshot after Local Top-k). Meter ALUs averaged 64.6% usage, SRAM averaged 12.1% (rising to 28.1% with $s = 32768$), and Hash Distance Unit usage averaged 33.3%. Shared hash functions across tables help minimize hash unit demand. Overall, as can be seen Meter ALUs are the most heavily used, yet for other switch resources no more than 34% is used. Note that this resource usage includes all of the NODE functionality, including any relevant components of Precision or Swish.

VI. Conclusion

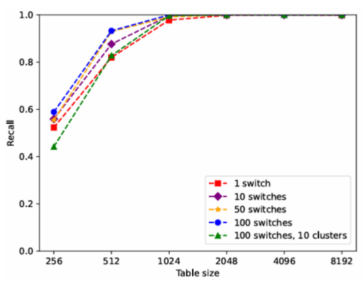
In conclusion, we present NODE, an algorithm for efficiently finding the network-wide top-k flows completely in the data plane. In the future we plan to study how clustering can further improve performance and perform evaluation on additional distributed hardware to show the performance improvement achieved by NODE compared to the centralized approach.



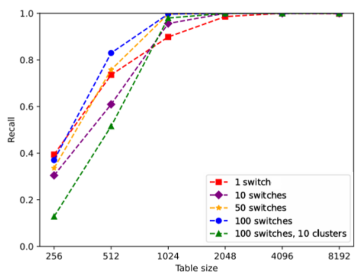
(a) Synth Zipf 0.6



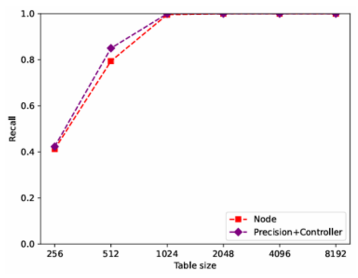
(b) Synth Zipf 0.8



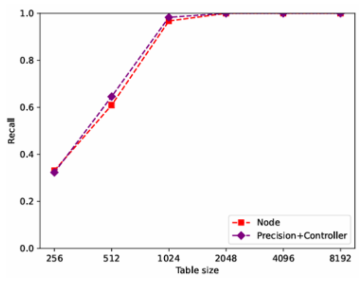
(c) Synth Zipf 1.0



(d) CAIDA



(e) Algo. comparison 100 switches



(f) Algo. comparison 10 switches

Fig. 2: NODE's recall with different datasets.

Acknowledgments

Supported by the Israel Science Foundation no. 980/21.

References

[1] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *CoNEXT*, 2011.
[2] V. Demianiuk, S. Gorinsky, S. I. Nikolenko, and K. Kogan, "Robust distributed monitoring of traffic flows," *IEEE/ACM Transactions on Networking*, vol. 29, no. 1, pp. 275–288, 2020.
[3] T. Benson and B. Chandrasekaran, "Sounding the bell for improving internet (of things) security," in *Workshop on Internet of Things Security and Privacy*, 2017, pp. 77–82.
[4] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic engineering in software defined networks," in *INFOCOM*, 2013.
[5] Y. Chen, R. Griffit, D. Zats, and R. H. Katz, "Understanding tcp incast and its implications for big data workloads," *University of California at Berkeley, Tech. Rep*, 2012.
[6] R. B. Basat, G. Einziger, S. L. Feibish, J. Moraney, and D. Raz, "Network-wide routing-oblivious heavy hitters," in *ANCS*, 2018, pp. 66–73.
[7] A. Shaikh, J. Rexford, and K. G. Shin, "Load-sensitive routing of long-lived ip flows," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4, pp. 215–226, 1999.
[8] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches," in *USENIX Security*, 2021, pp. 3829–3846.
[9] R. B. Basat, X. Chen, G. Einziger, R. Friedman, and Y. Kassner, "Randomized admission policy for efficient top-k, frequency, and volume estimation," *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, pp. 1432–1445, 2019.
[10] R. B. Basat, X. Chen, G. Einziger, S. L. Feibish, D. Raz, and M. Yu, "Routing oblivious measurement analytics," in *IFIP Networking*, 2020, pp. 449–457.
[11] "Netflow," https://www.ietf.org/rfc/rfc3954.txt.
[12] M. Wang, B. Li, and Z. Li, "sflow: Towards resource-efficient and agile service federation in service overlay networks," in *ICDCS*, 2004, pp. 628–635.
[13] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," 2015.
[14] R. Harrison, S. L. Feibish, A. Gupta, R. Teixeira, S. Muthukrishnan, and J. Rexford, "Carpe elephants: Seize the global heavy hitters," in *SPIN@SIGCOMM*, 2020, pp. 15–21.
[15] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *SOSR*, 2018.
[16] D. Ding, M. Savi, G. Antichi, and D. Siracusa, "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.
[17] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in $USENIX\ NSDI)$, 2016, pp. 311–324.
[18] L. Zeno, D. R. Ports, J. Nelson, D. Kim, S. Landau-Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, and M. Silberstein, "{SwiSh}: Distributed shared state abstractions for programmable switches," in *NSDI*, 2022, pp. 171–191.
[19] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
[20] "Intel Tofino," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html/, 6666.
[21] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.
[22] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Database Theory-ICDT 2005*, 2005, pp. 398–412.
[23] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Randomized admission policy for efficient top-k and frequency estimation," in *IEEE INFOCOM*, 2017, pp. 1–9.
[24] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *SoSR*, 2017, pp. 164–176.
[25] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "{NetChain}:{Scale-Free}{Sub-RTT} coordination," in *USENIX NSDI*, 2018, pp. 35–49.
[26] L. Tang, Q. Huang, and P. P. C. Lee, "A fast and compact invertible sketch for network-wide heavy flow detection," *Transactions on Networking*, vol. 28, no. 5, pp. 2350–2363, 2020.
[27] "The caida anonymized internet traces dataset," https://www.caida.org/catalog/datasets/passive_dataset, 2018, 2019.