On the Practical Detection of Hierarchical Heavy Hitters

Jalil Moraney, Danny Raz
Computer Science Department
Technion - Israel Institute of Technology
{jalilm, danny}@cs.technion.ac.il

***Abstract*—Finding the network's heaviest flows is an important and challenging network monitoring task and a critical building block for many other applications. In the hierarchical heavy hitters (HHH) problem, one needs to identify the most frequent network IP-prefixes hierarchically. This is a challenging task since the number of relevant IP-prefixes of flows in a busy router is much higher than the number of counters. To address this point, many streaming algorithms were recently developed, but they use complex data-structures and usually have non-constant per-packet update-time, preventing them from being deployed in line-speed. A randomized constant-time algorithm was proposed recently; however, it is only applicable to extremely large streams.**

In this paper, we propose a constant-time algorithm for detecting the HHH that does not have any convergence requirements and achieves comparable results to state of the art. Furthermore, our algorithm uses only efficient built-in counters available in current network devices, making it deployable on commercially off-the-shelf network gear. We provide an analytical study of the problem and show, using emulation over real traffic, that our algorithm performs at least as well as the best-known streaming algorithms without performing expensive per-packet operations or requiring convergence periods.

I. Introduction

The increasing popularity of cloud based systems and the shift towards Infrastructure as a Service (IaaS) as the preferred solution for many organizations [1], requires new approaches and novel solutions in the area of system management. Efficient management of such infrastructure heavily rely on efficient monitoring of the system resources and the workload. Despite many previous works considering efficiency aspects of many network management systems [2], [3], [4], very few tackled the practicality of deploying efficient monitoring tasks.

A practical efficient algorithm for any network monitoring task requires: (1) to be deployable on off the shelf network nodes, (2) to cope with current line rates, and finally (3) to use a limited amount of network resources which is much smaller than the ever increasing number of active flows.

An important monitoring task that was in the spotlight of recent research [5], [6], [7], [8], [9], [10] is the detection of Heavy Hitters (HH) and Hierarchical Heavy Hitters (HHH). A Heavy Hitter is a flow that is responsible for a considerable portion of the overall traffic (i.e., flows with traffic that exceeds a certain threshold) and a Hierarchical Heavy Hitter is an aggregation of non-HH flows that share a common property and is responsible, as a whole, for traffic above the threshold.

This paper takes advantage of the ability to reconfigure counters over time in network nodes. These reconfigurations, take place at statically provisioned periods of times, called rounds, depending on the given monitoring interval. During the rounds, the allocated counters simply measure the traffic of the assigned aggregated prefix. Thus, the per-packet operation can be handled in line rate with no additional support of special data structures or specialized hardware. Furthermore, we exploit the fact that these counters can measure both number of packets and total byte count with no additional cost to also solve the weighted version of the problem.

We present four algorithms, following the steps of [11], [12], [13]. In the first algorithm we assign to each counter an aggregated set of flows starting from some level of the hierarchy. At the end of each round, the algorithm seeks to zoom in on interesting flows down the hierarchy, thus evaluates the counters' values and reassigns the counters to the top half of the counters for the new round. The second algorithm aims at improving the performance by increasing the length of each round and decreasing the number of rounds. The motivation is that longer rounds leads to more stable HH and the zooming on process is more accurate.

Our third algorithm remedies a major limitation of the first two algorithms, the lack of "reconsidering" mechanism. I.e., the ability of the algorithm's to back off from the zooming process if the found flows turned out to be uninteresting and allocate the counter to previously discarded flow. This is achieved by keeping the frontier of the hierarchy disjointly monitored by counters over different levels. The fourth algorithm, waives the request of disjointly monitoring the frontier by monitored the highest shared ancestor that does not surpass the threshold. The motivation is to reduce the number of counters needed to cover the frontier and to enable deploying the algorithm when less counters are available.

The result is a family of practical efficient monitoring algorithms to the HHH problem which are deployable on off the shelf network nodes (see [12] for practicality of the approach) and can operate in line speed due to its constant time per-packet update operation. Furthermore, the algorithms use a configurable constant number of counters and guarantee not to use more than the allocated counters.

We evaluate the expected performance of our algorithms on real network traffic through an extensive simulation study using CAIDA's traces [14], [15]. The results indicate that our algorithms can detect around 90% of the true HHH while reporting only 3% non-HHH flows. These results can be achieved in line rate, while not requiring any convergence interval. Also, these results are comparable to the state of the art algorithms: (1)"MST" [16] - an accurate Space Saving [17] based algorithm that can not be deployed in line rate due to non-constant per-packet update operation, and (2)"RHHH" [7] a probabilistic constant update time improvement of "MST"

ISBN 978-3-903176-28-7

that requires a large convergence interval of 100M packets.

II. Related Work

Various papers had addressed the efficient detection of Hierarchical Heavy Hitters either in the field of streaming data or in network monitoring. These papers dealt with many aspects of the problem, including but not limited to (1) Number of dimensions of the hierarchy, (2) Relaxation of the problem by approximation, (3) Space requirements of the algorithms, (4) per item (packet) update time, (5) Lower bound on required Space, and (6) Convergence requirements of the algorithms.

The single dimension variant of the problem was introduced and approximated by a streaming algorithm in [18]. Later, [19] introduced an algorithm that requires $O(H^2/\epsilon)$ space, where H is the depth of the hierarchy and ϵ is the allowed relative estimation error for each single flow frequency.

Many algorithms to solve the multiple dimensions variant of the problem were proposed, each with its own properties [9], [16], [20], [21], [22], [23]. The common property of these algorithms is that the depth of the multi-dimensional hierarchy is the product of each dimension.

In [9], trie based algorithms were proposed that requires $O(H)$ update time, space of $O(H^2/\epsilon)$ and maintains a special data structure to hold the trie and its dynamic expansion. More recently, another trie-based solution was introduced in [20], the Full Ancestry and Partial Ancestry, that use $O(Hlog(N\epsilon)/\epsilon)$ space and requires $O(Hlog(N\epsilon))$ time per update, where N is the stream length. Alternatively, [21] improve the required space and update time to $O(H^{3/2}/\epsilon)$ (regardless of N) for the two dimensional problem.

A more efficient sketch-based algorithm with strong error and space guarantees were proposed in [16]. In their approach, they utilized a copy of Space Saving Sketch [24] per level, and detected the hierarchical heavy hitters by detected heavy hitters throughout all levels of the hierarchy. The algorithm requires $O(H/\epsilon)$ space and $O(H)$ update time [1].

Recently, a probabilistic version of the hierarchical heavy hitters problem was described in [7]. Exploiting this definition, the algorithm improved the update time to $O(1)$ on the expense of requiring a convergence interval. That is, a minimal number of items has to be processed before the probabilistic guarantees of the algorithm hold. The authors argue that in practice the algorithm's output is of the same quality as the previous deterministic approaches, while the convergence interval is not a real limitation since modern networks route a continuously increasing number of packets.

DREAM [25] is a framework for dynamically scheduling network traffic monitoring jobs (as black boxes), by allocating the amount of resources needed to maintain the predetermined accuracy of each job. When a monitoring job with a given accuracy requires more resources than available in the switch, the framework either rejects the job or tries to perform an expensive multi-switch resource allocation to achieve the desired accuracy rates. Such framework effectively balances two of the most critical characteristics of monitoring jobs, resources' availability and monitoring accuracy, and it is a very important and useful tool to schedule monitoring jobs in a compatible manner.

Our paper describes a different approach to detect hierarchical heavy hitters, while still being usable as a black box monitoring job in scheduling frameworks such as DREAM. The main trait of the approach is the practicality of deploying our algorithms on over-the-shelf network nodes while using a given number of counters (space) to detect the hierarchical heavy hitters. Furthermore, the per-packet update time of proposed algorithm is $O(1)$ without requiring any special data-structure or any convergence interval.

III. Notations and Problem Definition

A *flow* is a set of packets that share a common property (usually IP header fields). For example, all packets that have the same IP_SRC address belong to the same flow. One can form a hierarchy of flows by considering the prefixes of the field that specify a flow. In our example, the prefixes of the IP_SRC addresses forms the hierarchy. More formally, the universe of flows, $\mathcal{U}$, forms a hierarchy where the *dimensions* of the hierarchy is the number of fields that defines the flow and the *hierarchy levels* are the various prefixes of the fields values. Using this definition the flows in our example are of a single dimension (we only consider one field in the header) and a flow at level l consists of all packets that share the same prefix of length l from the IP_SRC field.

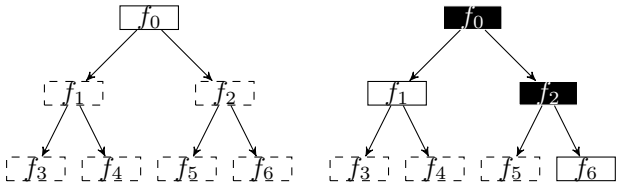
In order to formally define the hierarchy, we define the *Generalization Relation* between two hierarchy elements p, q, denoted by $p \prec q$, and say that p generalizes q, if p is a proper prefix of q with respect to a specific dimension (i.e., a field in the packet header). The *hierarchy depth*, H, is the length of the maximal path of prefixes that confirms with this generalization relation. In our example, the depth of the hierarchy is 32 as there are 32 bits in an IP address.

Given a *stream* of packets, $\mathcal{S}$ (sometimes referred to also as a trace) one can ask what is the frequency of a given flow in that stream. That is, how many packets that belong to this specific flow appear in the stream. In the case of streams of packets we also consider the weighted version of the frequency of a flow, which is the total amount of bytes carried by packets that belong to this flow. We use T to donate the total frequencies of all elements, N to denote the overall number of packets in the stream, and B to denote the total amount of bytes in the stream. Usually $T = N$, however when we consider the weighted frequencies of flows $T = B$.

Given a threshold ϕ, an element of the hierarchy (either a prefix or a specific flow) x, is a *Heavy Hitter (HH)* if its frequency is at least ϕ of the total frequencies of all elements, i.e., $f_x \geq \phi T$. The *Conditioned frequency* of prefix p with respect to a set of prefixes P, $cf_p(P)$, is the sum of frequencies of all elements that p generalizes without those of P. That is, we subtract from the frequency of p the sum of frequencies of the flows that are generalized by elements of the set P.

Given a threshold ϕ, an element of the hierarchy (either a prefix or a specific flow) is a *Hierarchical Heavy Hitter (HHH)*

[1] In the weighed version the update time is $O(Hlog(1/\epsilon))$



(a) The hierarchical flowset $f_3 \cup f_4 \cup f_5 \cup f_6$ can be measured by measuring f_0.

(b) The non Hierarchical flowset $f_3 \cup f_4 \cup f_6$, requires at least measuring f_1 and f_6.

Fig. 1: Example of hierarchical flowsets and non hierarchical flowsets, for the hierarchy covered by $f_0 = 39.128.128.128/30$.

if its conditioned frequency with respect to the set of HHH is at least ϕT. Another way to define the set of HHH for a given a stream of packets and a threshold ϕ, is to build it recursively. The set HHH_H is the set of HH at the lowest level of the hierarchy, i.e., $HHH_H = \{x \in \mathcal{U} | f_x \geq \phi T\}$. The set HHH_l, is the set of all prefixes at level l that their conditional frequency with respect to HHH_{l+1} is at least ϕT, i.e., $HHH_l = \{p \text{ at level } l | cf_p(HHH_{l+1}) \geq \phi T\}$. Then, the set of all HHH from all the levels of the hierarchy is $HHH = \bigcup_{l=0}^{H} HHH_l$. Table I contains the list of symbols and notations we use throughout this paper.

Symbol	Meaning
$\mathcal{U}$	The universe of flow identifiers
$\mathcal{S}$	The stream of packets
T	The total frequencies in $\mathcal{S}$
N	The overall number of packets in $\mathcal{S}$
B	The total byte count of packets in $\mathcal{S}$
f_x	The frequency of flow $x \in \mathcal{U}$
$cf_p(P)$	The conditional frequency of p in respect to P
ϕ	Heavy Hitter threshold
H	The depth of the hierarchy
$p \prec q$	p generalize q in the hierarchy
C	The number of available counters

TABLE I: A list of symbols and notations

IV. The "Simple Split" Algorithm

In general, the term *flowset* [11] is used to describe the stream formed from an aggregation of flows. In the context of this paper we are interested only in flowsets that cover a full hierarchy. Furthermore, it is very practical to assign counters to measure flowsets that cover a full hierarchy, using wildcards matching techniques widely available in any traditional network gear and over the shelf SDN switches [26], [27].

The main concept of this algorithm and our general approach is to facilitate the widely available easy measuring of such flowsets to calculate a set of suspect Heavy Hitter Flows and Hierarchical Heavy Hitter Flows. These sets are created by following paths of suspect heavy hitters flowsets down a *prefix trie*, where in each step the suspect flowset is broken into several disjoint flowsets to be measured independently. This follows the steps of [11], [12], [13].

In all of our algorithms, we identify each flow by a unique *string* over some alphabet and each flowset by a regular expression over the same alphabet, such that all flows contained in the flowset are the flows represented by the strings matching the flowset's regular expression. The motivation behind this approach is to identify each flow by an IP address and each flowset by a CIDR mask, such that a flowset is the group of all flows that their corresponding binary representation of the IP address is included in the flowset's CIDR mask.

This definition also allows tracking multidimensional flows (e.g., pairs of <IP_SRC,IP_DST>) by extending the string representing the flow to 64 bits consisted from contacting both strings of the IP source address and IP destination address. Also, one might consider five tuple flows <IP_SRC,IP_DST,SRC_PORT,DST_PORT,PROTOCOL> by contacting the binary strings of these fields from the IP header.

A main property of our general approach is that we limit the number of counters used by the algorithms a priori. That is, the number of counter available for monitoring is given as input (we usually use C to indicate this number see Table I) and the monitoring algorithm can not use more than this number of counters. With this limitation in hand, the main observation that motivates the algorithm, is that each HH flow in any level of the hierarchy mandates a HH flow in the upper levels of the hierarchy up to the root.

We note that given a threshold ϕ there can be no more than $\left\lfloor \frac{1}{\phi} \right\rfloor$ HH in a given level and no more than $\left\lfloor \frac{1}{\phi} \right\rfloor$ HHH in all the levels together. Furthermore, a HHH flow in a given level must be a HH flow in that level since the conditional frequency of the a flowset is at most its frequency. Thus, the algorithm tries to zoom in on HH flows by splitting the flowsets for which the overall traffic is greater than the threshold down to the lower level of the hierarchy. After detecting the suspect HH in all of the levels, the algorithm builds in a bottom-up approach the set of suspect HHH by calculating their conditional frequency.

A critical aspect of the algorithm is deciding when to investigate further the suspect flowsets, i.e., when to split the flowsets. Given the length of trace, the number of counters (C) and the depth of the hierarchy (H), the algorithm partitions the trace into $H + 1 - log_2(C)$ parts and performs a round of monitoring for each part. These parts, unless specifically stated, are usually equal. It is possible to partition the trace either by number of packets, by byte count or by time. Partitioning the trace by time is the most straightforward approach and only requires knowing the length of the monitoring period and could be user specific.

For each packet the algorithm updates exactly one counter, however, the algorithm performs $H + 1 - log_2(C)$ "heavy" steps, at the end of each round. In these steps the algorithm decides which of the currently monitored flowsets to split into two smaller flowsets. Such step requires $O(C)$ operations and happens $O(H)$ times regardless of the number of packets. Thus, the algorithm has a constant per-packet operation.

The static nature of the splitting and the observations about the nature of HH and HHH flows in the hierarchy form the base for the "Simple Split" Algorithm. The algorithm receives as input the number of counters, the depth of the hierarchy,

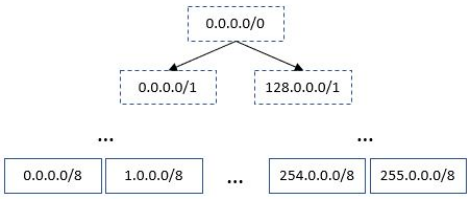


Fig. 2: Given $C = 256$ counters, the algorithm partitions the IP_SRC hierarchy ($H = 32$) at level 8 into disjoint sets and assigns a counter per flowset.

the monitoring period and the threshold. As an initialization step, the algorithm partitions the hierarchy into C flowsets and allocates a counter per each flowset. This means that the algorithm assigns a counter for each flowset in the level $log_2(C)$ of the hierarchy, as shown in figure 2.

During each round, where packets with respective values arrive, their keys are extracted and the counter of the matching flowset is updated with the respective value. The keys of the packets depend on the flow definition the user is using, usually a single dimension of the IP addresses. Furthermore, the respective values depends on which version of the problem we are tackling, the unweighted or the weighted version. In the unweighted version, we count the number of packets per flow, thus the values attached to each packets are 1. In the weighted version, we count the byte count of each flow, thus the values are the byte count field of the IP header.

At the end of each round, the algorithm takes decision regarding splitting flows. This means that the algorithm exams the aggregate values of the monitored flowsets and now should filter out "uninteresting" flowsets to allow allocating counters to finer (more specific) flowsets in order to move down the hierarchy. The most obvious candidates to refine are flowsets that had more than ϕ of the total frequency of the current period. However, since we refining each single flowsets into two flowsets, we have enough counter to refine the top $\frac{C}{2}$ flowsets. Thus, the algorithm sorts the monitored flowsets according to their counters values, refines the top $\frac{C}{2}$ and reassign the C counter to monitor these new flowsets.

This refinement step happens $H - log_2(C)$ times, until the algorithm reaches that last level of the hierarchy. Then, a bottom-up process takes place to calculate the set of candidate hierarchical heavy hitters. This process is straightforward calculation of the conditional frequency of each previously monitored flowset and output those that exceed the threshold.

This calculation does not require any estimation of the frequencies between the rounds since the rounds are equal. This approach works extremely well if the streams are stable over time. However, fluctuations in the flow's frequencies among the rounds, might cause missing some of the HH and thus some of the HHH flows. This is, of course, the main drawback of the "Simple Split" Algorithm.

One approach to tackle this drawback is splitting the trace into a smaller constant number of rounds (e.g. 4). This assumes, that the longer each round the more stable is the flows distribution among these rounds. This is the motivation behind the Multiple Split Algorithm presented below.

Another drawback of this algorithm is the lack of a "reconsidering" mechanism. That is, if the algorithm does not split a given flowset in an early round it will never reconsider it, even if there is some flow in this flowsets that became later responsible for a large amount of the traffic in the trace.

V. The "Multiple Split" Algorithm

The "Multiple Split" Algorithm follows that same motivation of the "Simple Split" Algorithm, while trying to relax its stability assumptions. In order to achieve that, we facilitate the ability to refine a given flowset not into just two disjoint flowsets, but rather into several 2^l disjoint flowsets.

The motivation behind this larger refinement process is that it results in smaller number of more stable rounds. The larger the round compared to the trace's size, the more we expect that it's flows distribution is closer to the entire trace distribution. Furthermore, this way we have a lower probability of deviating from the real HH flows down the hierarchy, due to bursts of other non HH flows or fluctuations in the real HH flows.

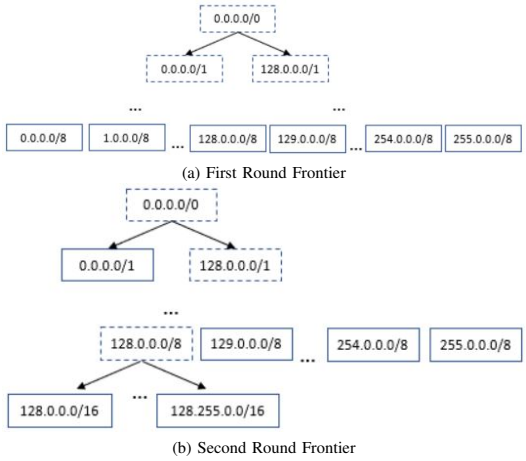
This larger refinement process allows the algorithm to advance several levels down the hierarchy at once. Since the algorithm is still limited to C counters, this comes at a cost that the algorithm can refine less flowsets than "Simple Split" Algorithm at each round. If at a given round the algorithm advances l levels, then to keep the limit of using C counters, it should refine each of the top $\lfloor \frac{C}{2^l} \rfloor$ flowsets into 2^l disjoint flowsets.

The initialization step is the same as in the "Simple Split" Algorithm with the addition of calculating the levels step. In each step the algorithm should decide at which levels of the hierarchy to monitor. It is clear that the first level should be $\lfloor log_2(C) \rfloor$ level in order to cover the entire frontier of the trie. Furthermore, the last level should be the depth of the hierarchy H, in order to be able to calculate the set of candidate HH and HHH correctly.

We calculate the levels to be monitored as part of the algorithm initialization. This is needed in order for the algorithm to know how many rounds of monitoring are expected so as to partition the trace into an adequate number of parts. One might consider a dynamic approach of calculating the next level "as we go", however this approach has no clear advantage in performance and in addition it imposes a new complication. Since in such a dynamic approach not all rounds are equal, the algorithm must use estimation when calculating the conditional frequencies of suspect HHH flows, possibly as proposed in [28]. Any such estimation step adds additional error to the final result with no clear advantage.

The rest of the algorithm is similar to the "Simple Split" Algorithm, except the refining of the flowsets, which replaces each candidate flowsets into 2^l disjoint flowsets rather than simply two. In fact, if the calculating levels step returns all levels (starting form $log_2(C)$ up to H) then the algorithm converges to the "Simple Split" Algorithm.

It is common to use the full byte levels $(8, 16, 24, 32$ as the monitored levels [29], [30]. For example, if we consider the IP_SRC hierarchy $H = 32$ with $2^{10} = 1024$ counters.



(a) First Round Frontier

(b) Second Round Frontier

Fig. 3: The retraction mechanism in "Holding the Frontier" Algorithm.

The initial step would be to assign the counters at level 10 of the hierarchy. Then, at the end of the respective rounds, the algorithm will advance to levels 16, 24 and 32. Previous works that reported HH and HHH in non 1-bit granularity limited their monitored to that specific granularity. Which means, they can not reconstruct from the non 1-bit data any insights on finer granularity HH and HHH. In the contrary, the "Multiple Split" Algorithm does not suffer from this limitation. Despite constructing the monitoring levels in non 1-bit granularity, the algorithm reconstructs the frequencies of all the inner nodes of the monitored trie.

VI. "Holding the Frontier" Algorithm

One of the main drawbacks of the "Simple Split" and "Multiple Split" Algorithm is the lack of a retracting mechanism. That is, once a flowset did not have enough frequency and was split further, we will not reconsider it again. The "Holding the Frontier" Algorithm remedies the lack of this mechanism.

We note that at the first round, the previous algorithms partition the space of flows into disjoint flowsets depending on the number of counters. At this initial step, all possible flows are covered by a single flowset. However, once a splitting decision is taken, only suspect flowsets will be partitioned and other flowset will stop being monitored. This usually keeps a vast part of the flow space uncovered.

In order to overcome this, the "Holding the Frontier" Algorithm tries to retract up the hierarchy the part of the flowsets that previously were not be monitored. This retraction will come at a cost of not freeing those counters to measure more refined flowsets as before. Thus, the retraction should be to the highest level possible in the hierarchy.

When considering up to which point to retract these flowsets, one should consider two main aspects. The first one is retracting to the highest level while still not having a flowsets which is above the threshold. Such flowsets are likely to split once again at the next round. The other consideration is to monitor flowsets which consist a frontier of the whole flow space and still maintain a disjoint partition.

Figure 3 depicts an example of the retraction mechanism in action for an IP_SRC hierarchy. In this figure, the CIDR mask inside the nodes represents the flowset of this node. Furthermore, nodes with full outline are the monitored nodes, while nodes with dashed outline are non monitored nodes.

At the first round, the algorithm starts monitoring the whole flow space at level 8. As a result of the monitoring in the first round, the algorithm decided to split only the flowset $128.0.0.0/8$ because it is the only flowset that surpassed the threshold. Instead of simply abandoning the other flowsets, the algorithm tries to recursively retract to the highest level.

This recursive retracting start by classifying all flowsets at the current level into several categories: refine, keep and retract. flowsets that are classified as refine are flowsets that surpasses the threshold, these flowsets are interesting and the algorithm partitions them in a lower level at the next round. Flowsets that are siblings of refine flowsets will be classified as keep, these flowsets did not surpass the threshold but can not be retracted to higher level since their siblings are refined. All other flowsets are classified as retract, and are aggregated into their common ancestor at a higher level.

We note that once two siblings are classified as retract at level l, then their common ancestor is added at level $l-1$. When processing the level $l-1$, if this ancestor surpassed the threshold then these two siblings will be added back to the frontier at level l as refine of level $l-1$.

However, when the algorithm decides on the refine set at the currently monitored level, it might partition these flowsets more than one level down the hierarchy as the case with the "Multiple Split" Algorithm, depending on the configurations.

Subfigure 3b depicts the frontier after the retraction process. The flowset $128.0.0.0/8$ was in the refine set of level 8 and thus was refined into the flowsets $128.0.0.0/16$ up to $128.255.0.0/16$. The flowset $129.0.0.0/8$ was classified as a keep in level 8 since its sibling was classified as a refine in the same level. The flowsets in the range of $0.0.0.0/8$-$127.0.0.0/8$ were not active during the first round, and thus were retracted up to level 1 and will be monitored by a single counter at the flowset $0.0.0.0/1$. The flowset $0.0.0.0/1$ was classified as keep at level 1 since its sibling was not present when level 1 was process, meaning it was partitioned at lower levels.

The flowsets $254.0.0.0/8, 255.0.0.0/8$ did not surpass the threshold thus were not classified as refine at level 8 and thus should be retracted, but assuming that their common ancestor $254.0.0.0/7$ did surpass the threshold and were classified as refine, they were added back to the frontier at level 8.

Up to this point we overlooked and important detail of the retracting mechanism, the affect of retracting on the number of available counters. In the best case scenario, the retracting mechanism manages to retract vast parts of the frontier to higher levels and saving enough counters, more than needed for refining flowsets at the refine set of the current level. In this scenario, the algorithm still managed to keep the frontier intact, covering the whole space, and to refine the interesting flowsets as needed. If additional counters are available, the algorithm unretracts higher levels by refining their flowsets.

This unretacting step, happens only if we saved enough counters, and the motivation behind is to set the algorithm

in better position for the next round. That is, since we have spare counters after all the required refinements, we partition flowsets at the higher position of the hierarchy into a lower level since these flowsets are the most aggregated and are the most prone to surpass the threshold without any actual child that surpasses the threshold.

On the other hand, in the worst scenario the retracting mechanism does not save enough counters to enable full refinement of the frontier. This scenario might happen in a balanced trace over the hierarchy where the algorithm does not manages to retract the frontier up to the higher levels. While the most common possibility of this scenario is that the number of counters is in the order of the number of HHH in the trace. In such scenario, the algorithm does not have enough counter for full refinement and for keeping the frontier intact since the set of suspect HH are in the order of the number of the available counters and the algorithm track each suspect HH by a single counter at each point.

The complexity of the retracting and unretracting step of this algorithm is still $O(HC)$, since at most we perform recursive retraction over (H) levels and unretracting over the same $O(H)$ levels. The most important thing to note when considering the complexity, is that at any given level we perform at most $O(C)$ operation and not $O(2^l)$ operations, since at each level the algorithm monitors at most $O(C)$ flowsets and when performing the classification of each level we consider only monitored flowsets.

Thus, despite complicating the per round operation the algorithm still performs a constant per-packet operation and a constant number of times a heavier operation that is still linear in the number of counters and $O(1)$ when considering the number of packets or even the number of active flows.

Algorithm "Holding the Frontier":

Input : A stream of packets S, a threshold ϕ, number of counters C and the depth of the hierarchy H
Output: set of HH and HHH in S

$F = init\_flowsets(C)$;
$levels = calculate\_levels(C, H)$;
$current\_level = levels[0]$;
$number\_rounds = |levels|$;
foreach r *in* $\{1..number\_rounds\}$ **do**
  $counters = assign\_counters(F)$;
  $P = get\_round\_packets(r, number\_of\_rounds)$;
  foreach *counter in counters* **do**
    counter.value= $\sum_{\{p \in P : flow(p) \in counter.flowset\}} value(p)$;
  end
  if $r < number\_rounds$ **then**
    $l = levels[r] - current\_level$;
    $to\_refine, retracted = retract\_frontier(counters, F, current\_level)$;
    $refined = refine\_flowsets(to\_refine, l)$;
    $needed\_counters = |refined \cup retraced|$;
    if $needed\_counters > C$ **then**
      $abort()$;
    end
    else
      $retracted = unretract\_frontier(C - needed\_counters, retracted)$;
    end
    $F = refined \cup retracted$;
  end
end
return $calculate\_hhh\_bottom\_up(\phi)$;

VII. "Shared Ancestor" Algorithm

One of the main limitations of the "Holding the Frontier" Algorithm is that when the number of counters is in the same order of the number of HH flows, even after retracting the frontier, the algorithm can not fully refine the suspect flowsets. In order to explain how to overcome this limitation, we examine the state of the frontier when a retraction happens in a sub-trie with a single refine flowset.

Figure 4 represents the frontier after retracting and refining when the flowset $128.0.0.0/8$ was the only active flow of this sub-trie and surpassed the threshold. Since it surpassed the threshold, it was refined into the flowset in the range $128.0.0.0/16$ up to $128.255.0.0/16$. Since it was the only flowset that surpassed the threshold, the retraction step tries to free counters however it keeps a path form the highest root that did not surpass the threshold down to the interesting flowset. This "wastes" several counters in the following round. None of the flowsets $129.0.0.0/8, 130.0.0./7, ..., 192.0.0.0/2$ could be retracted more since their siblings are not present in the respective level, even if they had 0 frequency in the round.

The "Shared Ancestor" Algorithm tries to save counters by replacing the allocation for these non-interesting flowsets by a single counter on their common ancestor. this breaks the promise to keep a disjoint partition of the frontier. Instead of allocating a counter per level, the algorithm allocates a counter to the highest shared ancestor that did not surpass the threshold. This shared ancestor counts the traffic all of its sub-trie and then we subtract the "interesting" flowsets for which we have separate counters.

The ellipses node in Figure 4 represents this shared ancestor and the monitored flowsets in the example are $128.0.0.0/1$ and all of the flowsets at level 16. If we compare this to the monitored set in "Holding the Frontier" Algorithm, that consists of the solid outline nodes, it requires 7 less counters. While these savings seems low, they might happen at any level of the hierarchy and in any part of the flow space, with a compound effect of freeing enough counters to make the algorithm deployable even with small number of counters.

One must note that in this algorithm, flowsets from various levels are monitored together and one must be very careful when comparing their frequencies since they set of flows from different sizes. One might be tempted to normalize the frequencies of the larger flowsets in order to be able to compare them to the smaller flowsets and use the same flowset, but in fact this might turn to be counter constructive.

The main motivation behind keeping a frontier is the ability to retract to abandoned flows, however, if we normalize the frequencies of large flowsets by dividing them by the number of flows in that flowset, then this flowset will surpass the threshold if and only if most of its flows exceed the the

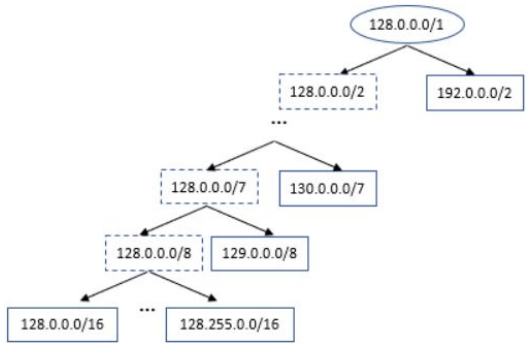


Fig. 4: The frontier after retraction in "Holding the Frontier" and "Shared Ancestor" with a single interesting flowset.

threshold. Usually, this means that this ancestor flowset will never be refined again, missing the point of retracting into abandoned flows. Details regarding the actual steps of the algorithm are provided in Algorithm "Shared Ancestor".

Algorithm "Shared Ancestor":

Input : A stream of packets S, a threshold ϕ, number of counters C and the depth of the hierarchy H
Output: set of HH and HHH in S

$F = init\_flowsets(C)$;
$levels = calculate\_levels(C, H)$;
$current\_level = levels[0]$;
$number\_rounds = |levels|$;
foreach r *in* $\{1..number\_rounds\}$ **do**
  $counters = assign\_counters(F)$;
  $P = get\_round\_packets(r, number\_of\_rounds)$;
  foreach *counter in counters* **do**
    counter.value= $\sum_{\{p \in P : flow(p) \in counter.flowset\}} value(p)$;
  end
  if $r < number\_rounds$ **then**
    $l = levels[r] - current\_level$;
    $to\_refine = calculate\_to\_refine(counters, F, current\_level)$;
    $refined = refine\_flowsets(to\_refine, l)$;
    $ancestors = \{\}$;
    foreach f *in* $refined$ **do**
      $sa = calculate\_shared\_ancestor(f, F)$;
      $ancestors.add(sa)$;
    end
    $F = refined \cup ancestors$;
  end
end
return $calculate\_hhh\_bottom\_up(\phi)$;

VIII. Evaluation

We evaluated our algorithms using the following real life traces: (1) CAIDA'16: CAIDA Internet Traces from "Equinix-Chicago" in 2016 [14], (2) CAIDA'18: CAIDA Internet Traces from "Equinix-NewYork" in 2018 [15]. We considered IP source hierarchies in a single bit granularities, such hierarchies were also used in [7], [16], and considered the unweighted frequencies of items (i.e., the number of packets). Each data point is the average of 10 runs, where each run started from randomly selected point in the given trace.

We use the *Recall* and *Precision* metrics proposed in [31] to evaluate the performance of the suggested algorithms. In order to compute these metrics we calculated the true set of HH and HHH using an space intensive algorithm that allocates a counter per flow for exact measurements. Recall is the number of true HHHs detected by the algorithm divided by the number of true HHHs. This metric is equivalent to the Detection Rate of the algorithm, i.e., what is the percentage of HHHs the algorithm detects. Precision is the number of true HHHs detected by the algorithm divided by the number of reported suspect HHH. This metric complements the false positive rate of the algorithm and tries to grasp on how many of the suspect HHHs the algorithm was mistaken.

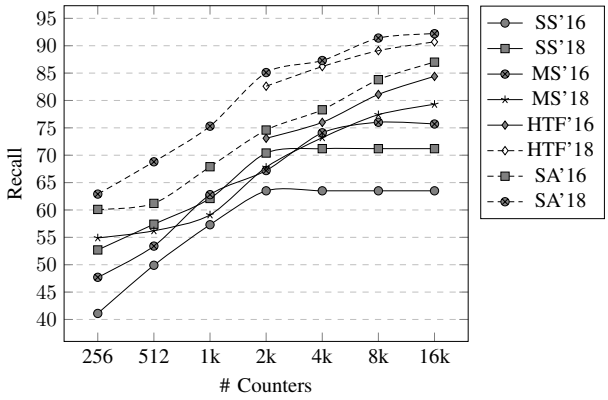


Fig. 5: The Recall of the algorithms for various number of counters on CAIDA'16 and CAIDA'18 traces.

We denote our proposed algorithms with "SS" for "Simple Split" , "MS" for "Multiple Split" , "HTF" for "Holding the Frontier" and "SA" for "Shared Ancestor". Furthermore, we use "RHHH" to denote the algorithm proposed in [7] and "MST" to denote the algorithm proposed in [16]. These algorithms solve an approximate version of the problem with an accuracy parameter ϵ, when comparing with them we expand the set of true HHH by a slack of ϵT.

Figure 5 shows the *Recall* of the various algorithms as a function of the number of available counters. Each line describes a single algorithm and the CAIDA trace, the runs are on random parts of the trace of 2^{25} packets (about one minute of traffic) with threshold $\phi = 0.001$. It is worthy to note that the performance on CAIDA'18 trace are usually better than those of CAIDA'16 trace, this is since in CAIDA'16 trace the heaviest flows are not as stable as in CAIDA'18. Also note that the number of HH in these traces at a given level of the hierarchy is at most 200. This explains the overall poor performance when using less counters than 512.

When considering the Recall of "Simple Split" and "Multiple Split" algorithms, one can notice an improvement as the number of counters increase up to a point where adding more counters does not help anymore. This limitation is explained by the lack of retracting mechanism in these algorithms, thus missing a portion of the HHH that start late in the monitoring interval and more specifically after the first round.

The "Holding the Frontier" Algorithm can not be ran properly for a very low number of counters, that is up to $1k$ counters. That is due to the fact that sometimes the attempt to retract the frontier does not free enough counters to perform

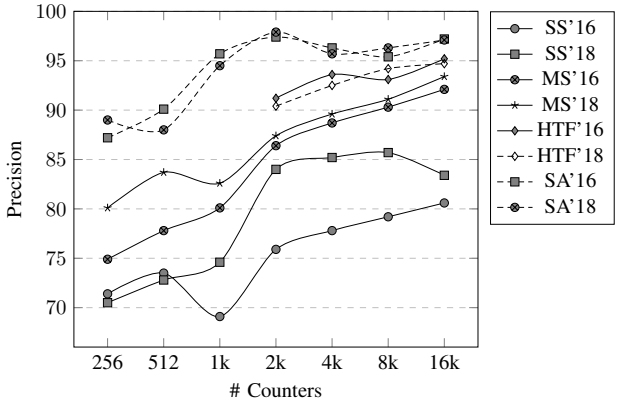


Fig. 6: The Precision of the algorithms for various number of counters on CAIDA'16 and CAIDA'18 traces.

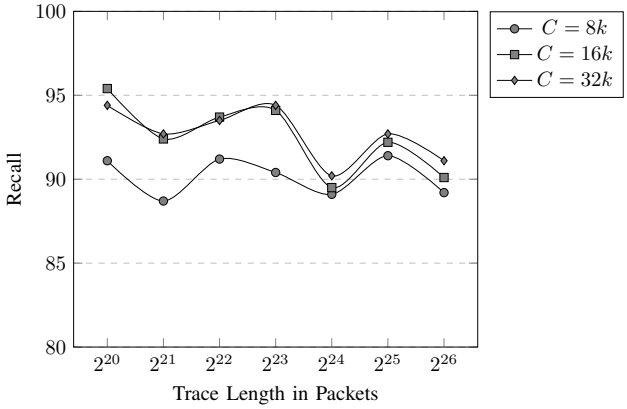


Fig. 7: The recall of "Shared Ancestor" Algorithm for various length of trace on CAIDA'18.

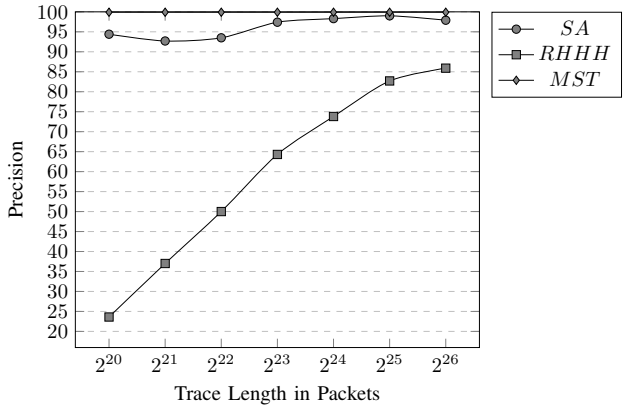


Fig. 8: The precision of "Shared Ancestor" Algorithm compared to "MST" and "RHHH" on CAIDA'16 trace, for $\phi = 0.01, \epsilon = 0.001, C = \frac{H}{\epsilon} = \frac{32}{0.001} = 32k$ on single bit IP source hierarchy for various lengths the trace.

full refinement of the interesting flowsets. Thus, we did not report the results for number of counters where at least one run of the algorithm did not finish due to this reason. For higher number of counters (starting from $2k$) the algorithm manages to breach the observed limitation at "Simple Split" and "Multiple Split" algorithms due to its retracting mechanism.

The main advantage of "Shared Ancestor" algorithm compared to "Holding the Frontier" algorithm lies on the ability to deploy it using a smaller number of counters. The difference in the recall of the algorithms is not statistically significant, where both reach around 90% recall on the CAIDA'18 traces.

Figure 6 depicts the *Precision* of the various algorithms as a function of the number of available counters in the same settings as Figure 5. More specifically, the figure shows how many flows reported by the algorithm as HHH were actually a true HHH, i.e. how precise was the algorithm in its reports. All of our algorithm tend to report most of their false positives, reported flows that are not HHH, in the higher levels of the hierarchy. Due to the parts of the hierarchy that are not monitored but still see some traffic, however, in lower levels the algorithms have a better estimation of the flow's frequencies and whether they are HHH or not. Furthermore, the higher in the hierarchy the calculation of HHH happens, the more its prone to errors due to errors in the lower levels.

We note that the difference in precision between the two traces in a given algorithm is not that noticeable. This is explained by the fact that if even in CAIDA'16 the heaviest flows are not consistent compared to CAIDA'18, the algorithms manage to filter out flows that lead to splitting at higher levels but did not remain suspect HH throughout the rounds.

Furthermore, the trend of better performance with more counter we observed in figure 5 is less clear especially in "Simple Split" and "Multiple Split" algorithms. That is, sometimes more counters lead to a small decrease in the precision of the algorithms. This might be explained by the fact that with more counters these simple algorithms focus on more unimportant parts down the hierarchy.

The precision of "Shared Ancestor" Algorithm reaches more than 95% for $1k$ counters and even around 97% for more than that. This means, that given enough counters this algorithm detects around 90% of the true HHH and report no more than 3% non-HHH flows. For the same reasons mentioned before, the results of "Holding the Frontier" Algorithm were not reported for counters less than $2k$.

Figure 7 depicts the recall of "Shared Ancestor" Algorithm (under same settings) as function of the trace length (number of packets processed) for a given number of counters. It easy to see that the algorithm does not require any convergence period in order to achieve high recall. Furthermore, there is no consistent trend in the recall of the algorithm as function of the trace length, besides slight variations that can be explained as fluctuations in the different parts of the trace.

Figure 8 depicts the precision of the "Shared Ancestor" algorithm as function of the trace length (number of packets processed) compared to "MST" and "RHHH". As expected, "RHHH" suffers from a convergence interval, only then it starts to keep its probabilistic guarantees of low false positive. "MST", which requires $O(H)$ update time per-packet, achieves almost perfect precision (no false positive) as guaranteed by ϵ. Our "Shared Ancestor" algorithm achieves high precision rate averaged around 95% regardless of the trace's length, while holding $O(1)$ per-packet update time. The

algorithms converges similarly on CAIDA'18 traces.

IX. Conclusions

In this paper we presented several practical algorithms for Hierarchical Heavy Hitters detection. These algorithms can be deployed on off-the-shelf network nodes (or software devices) and can operate in line speed due to their $O(1)$ per-packet operation. The current state of the art algorithms, either require $O(H)$ per-packet operation that makes them unfeasible to be deployed in line rate or requires a convergence interval before reporting satisfactory results which makes them less relevant in many practical settings. In contrary, our algorithms perform in line-speed with $O(1)$ update per-packet without requiring any convergence interval. Furthermore, no complex data structures are needed and our algorithms only require using built-in fast counters available in any network node.

We evaluated the algorithms on two recent real Internet packets traces and showed that they yield a comparable results to the state of the art without their limitations. The evaluation showed that the best algorithm can detect up to 90% of the HHH in a trace and report no more than 5% non HHH flows.

These algorithms could be easily extend to the case of multi-dimensional HHH while keeping the depth of the hierarchy linear in the number of dimensions without modifying the $O(1)$ update time. Also, they allow practical detection of the weighted set of HHH flows with minimal modification of the update operations while keeping all of the algorithms promises. In future work, we plan to study the control mechanism of the algorithms and their fine deployment issues. Furthermore, we plan to adjust the algorithms to detect DDoS attacks by facilitating the already needed calculation of HH at the lower level of the hierarchy.

References

[1] S. Goyal, "Software as a Service, Platform as a Service, Infrastructure as a Service - A Review," *International journal of Computer Science & Network Solutions*, vol. 1, no. 3, pp. 53–67, 2013.

[2] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 254–265.

[3] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the seventh Co-NEXT*. ACM, 2011, p. 8.

[4] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *computers & security*, vol. 28, no. 1-2, pp. 18–28, 2009.

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

[6] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner, "Optimal elephant flow detection," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[7] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 127–140.

[8] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 164–176.

[9] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. ACM, 2004, pp. 101–114.

[10] D. Tong and V. Prasanna, "High throughput hierarchical heavy hitter detection in data streams," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE, 2015, pp. 224–233.

[11] L. Yuan, C.-N. Chuah, and P. Mohapatra, "ProgME: towards programmable network measurement," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM '07*. NYC, NY, USA: ACM Press, 2007, pp. 97–108.

[12] J. Moraney and D. Raz, "Efficient detection of flow anomalies with limited monitoring resources," in *2016 12th International Conference on Network and Service Management*. IEEE, oct 2016, pp. 55–63.

[13] J. Moraney and D. Raz, "On the practical detection of the top-k flows," in *2018 14th International Conference on Network and Service Management*. IEEE, 2018, pp. 81–89.

[14] "The CAIDA UCSD Anonymized Internet Traces 2016 - equinix-chigaco January. 21st, Direction A." [Online]. Available: http://www.caida.org/data/passive/passive_2016_dataset.xml

[15] "The CAIDA UCSD Anonymized Internet Traces 2018 - equinix-nyc 2018-03-15, Direction A." https://www.caida.org/data/monitors/passive-equinix-nyc.xml.

[16] M. Mitzenmacher, T. Steinke, and J. Thaler, "Hierarchical heavy hitters with the space saving algorithm," in *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2012, pp. 160–174.

[17] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Lecture Notes in Computer Science*, ser. Lecture Notes in Computer Science, T. Eiter and L. Libkin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3363 LNCS, pp. 398–412.

[18] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in data streams," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 464–475.

[19] Y. Lin and H. Liu, "Separator: sifting hierarchical heavy hitters accurately from data streams," in *International Conference on Advanced Data Mining and Applications*. Springer, 2007, pp. 170–182.

[20] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in streaming data," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 4, p. 2, 2008.

[21] P. Truong and F. Guillemin, "Identification of heavyweight address prefix pairs in ip traffic," in *2009 21st International Teletraffic Congress*. IEEE, 2009, pp. 1–8.

[22] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 155–166.

[23] J. Hershberger, N. Shrivastava, S. Suri, and C. D. Tóth, "Space complexity of hierarchical heavy hitters in multi-dimensional data streams," in *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2005, pp. 338–347.

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

[25] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: Dynamic Resource Allocation for Software-defined Measurement," in *Proceedings of the 2014 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM '14*, pp. 419–430.

[26] OpenvSwitch, "Production Quality, Multilayer Open Virtual Switch."

[27] "OpenFlow Switch Specification 1.5.1," Tech. Rep., 2015.

[28] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications," in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '04. New York, NY, USA: ACM, 2004, pp. 101–114.

[29] M. Mitzenmacher, T. Steinke, and J. Thaler, "Hierarchical heavy hitters with the space saving algorithm," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2012, pp. 160–174.

[30] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. NYC, NY, USA: ACM, 2017, pp. 127–140.

[31] G. Cormode and M. Hadjieleftheriou, "Methods for finding frequent items in data streams," *VLDB Journal*, vol. 19, no. 1, pp. 3–20, 2010.