On the Practical Detection of Heavy Hitter Flows

Jalil Moraney
Computer Science Department, Technion
Haifa, Israel
jalilm@cs.technion.ac.il

Danny Raz
Computer Science Department, Technion
Haifa, Israel
danny@cs.technion.ac.il

***Abstract*—The detection of of Heavy Hitter (HH) flows in a network device is a critical building block in many control and management tasks. A flow is considered a Heavy Hitter flow if its portion from the total traffic surpasses a given threshold. One of the most important aspect of this detection is its practicality; i.e., being able to work in line rate using the available scarce local memory in the device.**

In this paper, we present a practical heavy hitters detection algorithm that requires a constant amount of memory (not related to the number of flows or the number of packets) and performs at most $O(1)$ operation per packet to keep with line rate. We present an analysis of errors for our algorithm and compare it to state-of-the-art monitoring solutions, showing a superior performance where the allocated memory is less than $1MB$. In particular, we are able to detect more HH flows with less false positive without increasing the per-packet processing time.

***Index Terms*—heavy hitter flows, memory constrained monitoring**

I. Introduction

Modern Infrastructure as a Service (IaaS) providers rely on various network protocols to provide efficient service management capabilities. These protocols, as well, rely heavily on the ability to provide correct and efficient network monitoring that generates the needed statistics. Such protocols belong to various service management sub-domains such as traffic engineering [1], anomaly detection [2], load balancing [3], NFV placements [4] and many more.

The ability to provide accurate per flow statistics, while greatly important for service management protocols, is often impractical due to the high number of active flows and the limited on-chip memory needed to keep such counters. Furthermore, no remediation is expected in the near future; more flows are flowing through networks than before, the line rate is ever increasing requiring counters with more width and more monitoring applications are running, requiring their own portion of the available memory.

Given these limitations, it is important to consider designing and implementing efficient practical algorithms for various monitoring tasks [5], [6]. We consider an algorithm to be an efficient practical monitoring algorithm if (1) it provides important monitoring data, (2) performs $O(1)$ operations per packet to cope with line rate, (3) uses a constant limited amount of memory that is much smaller than and not related to the number of active flows, and finally (4) implementable on memory schemes available in off-the-shelf network gear.

978-3-903176-32-4

Many of the current state-of-the-art and most notable monitoring schemes require a non-constant amount of memory depending either on the traces size or in inverse relation to the guaranteed accuracy [7], [8], [9], [10]. In such schemes, the required memory is usually of size $o(\frac{1}{\epsilon})$ where ϵ is the guaranteed accuracy. This notation hides the fact that in many practical use-cases their memory requirement is higher than what is available in the network device.

With the increasing amount of various monitoring applications running concurrently on network devices, the approach of setting accuracy parameters per application and only then count for their required memory is not operational. This approach makes it hard for network operators to meet the memory constraint by prioritizing applications and optimizing their accuracy parameters. The correct approach to manage on-device monitoring applications is to set hard constraints on their memory footprint and only then set the desired accuracy parameters while slightly compromise the quality of the monitoring results according to the available resources.

The quality of the monitoring results is commonly measured via two metrics, *Recall* and *False Positive Rate* (FPR). Recall (sometimes also called Detection Rate) measures the proportion of positives items that are correctly identified while FPR measures the proportion of the none-positive items that are reported as positive (i.e., misidentified) from the set of reported items. Describing the quality of the monitoring data without reporting both metrics is not complete, since, while high recall algorithms are preferred, this should not come at a price of a very high FPR. An extreme example of such an algorithm is the one that reports all flows as positive, this algorithm has a recall of 100% but also almost a 100% FPR, hindering its output useless.

Having on-device monitoring algorithms that support line rate is of high importance, no network operator in the right mind would consider reducing network capacity due to the deployment of monitoring algorithms on devices. The guarantee of line rate performance usually manifests in having $O(1)$ operations per packet, avoiding any "heavy" operations that can not be perform in line rate.

Many monitoring algorithms maintain complex schemes and data structures that require once in a while perform a "heavy" operation (for example a re-ordering of the data scheme). Algorithms of this type claim to support line rate while having an amortized $O(1)$ operation by proving that this "heavy" operation happens once in many packets and when its cost

is split among all the relevant packets it would add a constant negligible effect.

This point of view on line rate operation is in many cases impractical. While truly it is an amortized $O(1)$ operation, during such operations, arriving packets must be stored to be later processed and this could affect the devices' available buffers and affects their performance. Thus, one of our motivations is to design a scheme that performs worst-case $O(1)$ operation.

A. Related Work

While the problem of detecting Heavy Hitter Flows, is a classical monitoring problem that is easily solvable given enough memory and had been addressed heavily in the literature [11], [12], [13], [14], [7], [15], [16], not enough thought was given to the practicality of the deployment of the proposed solutions.

The state-of-the-art algorithm to detect HH flows was introduced in [15]. In this paper the authors introduced, $IM-SUM$, an algorithm which solves an ϵ-approximation of the *frequency estimation* problem and a (ϕ, ϵ)-approximation of the HH problem. The algorithm holds two monitoring tables, a passive and an active tables, then periodically replaces their roles. The motivation is that the passive table holds the "biggest" flows and the active holds the most "recent" flows. When the passive table gets full a "heavy" maintenance operation is performed, that calculates the $\lfloor \frac{1}{\epsilon} \rfloor^{th}$ largest value in the passive table, drops all entries smaller than it, and moves larger entries to the active table. Furthermore, a variable q holds the value of the last calculated $\lfloor \frac{1}{\epsilon} \rfloor^{th}$ largest value and it functions as the estimation of flows not in any of the tables. The authors provide an evaluation that indicates that their algorithm preforms better than several previously proposed algorithms including Count Min Sketch [17], Hierarchical CM-sketch [18], Count Sketch with Adaptive Group Testing [19] and heap-based implementation of Space Saving [20].

The main drawbacks of this algorithm are: (1) it performs a "heavy" maintenance operation that makes its performance an amortized $O(1)$ operation, (2) it requires an amount of memory that is proportional to the inverse of the accuracy parameter $O(\frac{1}{\epsilon})$ and (3) its approximation is of the form $N(\phi-\epsilon)$ where N is the number of packets, ϕ is the threshold and ϵ is the accuracy parameter. This means that slack given to the approximation is ϵN which is proportional to the overall amount of traffic processed by the algorithm.

The authors of [21] presented an algorithm called "Sample & Hold" which holds an exact counter for each sampled flow. This way, it assures that the estimation of the sampled flow is quite accurate since after the first sampled packet of the flow the algorithm measures all its packets. This means that the only error in the estimation of the flow's frequency comes from packets that were not sampled prior to the first sampled packet. Then they showed that when oversampling by a factor of O, i.e., sampling with the probability of $p = O/(\phi N)$, the probability of missing a flow of size ϕN is approximately e^{-O} and that the relative error in estimating such flow is $\sqrt{2-p}/O$. The main drawbacks of "Sample & Hold", are that (i) it requires oversampling which yields in higher memory usage, and (ii) it performs a per-packet operation for "held" flows.

In [22], the authors introduced the *CEDAR* algorithm that decouples the stored flows identifiers from their estimation values by using a constant size table to store shared estimators. The algorithm holds two tables, the first one has an entry per each flow which holds the flow's identifier and an integer. This integer is used as an index to the second table pointing to the flow's current estimated value. Thus, the second table is a shared estimators table, where estimators' values are constructed in a way the provides unbiased estimations while achieving minimal maximal relative estimation error. When a new flow arrives, it enters the flows table with an index of 0, meaning its initial estimation is $v_0 = 0$. On each arrival of a packet from a previously seen flow, the algorithm extracts its current estimation, v_i, and the value of the next estimator v_{i+1} and increases the estimation of the flow to be v_{i+1} with probability $\frac{1}{v_{i+1}-v_i}$. The main drawback of [22] is that it maintains an entry per each active flow in the system and thus unpractical as a HH algorithm.

B. This paper

In this paper, we devise a practical algorithm for the problem of detecting Heavy Hitters Flows, by combining ideas from [21] and [22]. The algorithm builds on top of the "Sample & Hold" concept without being limited to the weighted sampling requirement, the per-packet operation, and the oversampling requirement. The algorithm also takes advantage of the constant amount of estimators in the CEDAR algorithm while keeping the needed memory also constant. This can be achieved since we only care about the size of large flows (HH candidate) and we do not need to maintain an estimator for each small flow. The main difficulty is identifying the flows that need to be monitored; this is done by looking at flows that have frequent appearances. We keep all recent flows in a table called Suspect Flows Array and when a new packet from one of these flows arrives, before the flow is evicted from the table, we mark this flow as a HH candidate and advance it to the Candidate Set where its frequency is being measured by the shared estimators.

The main contribution of our work is the presentation of a new practical heavy hitters detection algorithm that requires a constant amount of memory (not related to the number of flows, the number of packets, or the oversampling parameter) and performs at most $O(1)$ operation per sampled packet to keep with line rate. Furthermore, we compare it to state-of-the-art monitoring solutions on real internet traces, showing a superior performance where the allocated memory is less than $1MB$.

In Section II we describe the architecture and the data structures of our algorithm, the Candidate Set algorithm, and then in Section III, we analyze the possible source of errors. In Section IV we present a less robust variant of our algorithm, the Exact Bank algorithm, that performs better when the trace is heavy tailed. Then in Section V we evaluate the Candidate

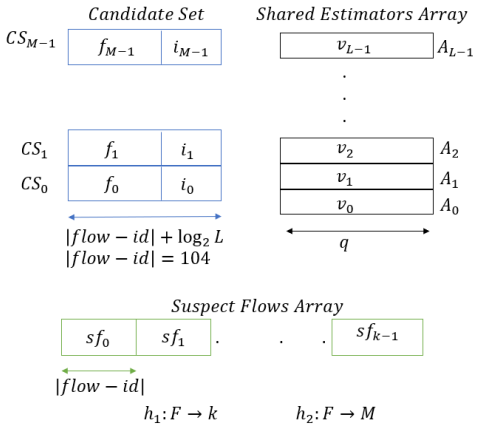


Fig. 1: The basic architecture of the Candidate Set algorithm

Set algorithm performances and compare it to state-of-art-algorithm, and finally we conclude in Section VI.

II. The Candidate Set Algorithm

The Candidate Set algorithm is based on three data structure: (1) the *Suspect Flows Array*, (2) the *Candidate Set*, and (3) the *Shared Estimators Array*. Figure 1 illustrates these data structures. The algorithm uses the first two to store identifiers of flows suspected as HH flows with different levels of confidence. In the Suspect Flows Array the algorithm stores identifiers of flows that were recently encountered in order to track their potential growth. When this growth is fulfilled, these flows are propagated to the Candidate Set. In the Candidate Set the algorithm stores identifiers of "heavy" flows, i.e., flows that already showed some evidence for a large amount of traffic.

This way, the algorithm is able to track "recent" and "heavy" flows in order to estimate their frequency using the Shared Estimators Array. Next, we describe how a HH flow is handled by the algorithm. First, it is added into the Suspect Flows Array. Then, on the second arrival of its packets, if it was not evicted in the meantime, it is propagated to the Candidate Set. Afterward, it is tracked in the Candidate Set and its frequency is estimated via the Shared Estimators Array.

Non HH flows that are of a negligible size usually reaches only the Suspect Flows Array and are not propagated to the Candidate Set since packets from other flows evict the entry before a second packet from the flow arrives.

Other non HH flows that are not negligible but still are not HH could propagate to the Candidate Set from the Suspect Flows Array. However, it is not enough to propagate to the Candidate Set in order to be considered as a HH. A flow size estimation in the Shared Estimators Array needs to be above the threshold. The Shared Estimators Array mechanism is based on the estimation table introduced in the CEDAR algorithm [22] and it guarantees a maximal relative error of δ for any flow in the Candidate Set, thus our algorithm might produce a false positive only on flows with frequencies in the range$[(1-\delta)\phi N, \phi N]$.

The first step of initializing the data structures is to partition the total memory available to the algorithm to the three data

TABLE I: A summary of the used notations

Symbol	Meaning
m	total available memory
m_{CS}	share of Candidate Set memory out of m
m_{SFA}	share of Suspect Flows Array memory out of m
m_{SEA}	share of Shared Estimators Array memory out of m
ϕ	heavy hitter threshold
N	number of packets
F	5-tuple flows Universe
δ	guaranteed relative error
A_i	i^{th} shared estimator
L	number of shared estimators
q	width of an estimator
v_0	the minimal estimator
v	the "propagation" parameter
M	the number of entries in the Candidate Set
CS_i	i^{th} flow in the Candidate Set
k	the number of entries in the Suspect Flows Array
sf_i	the i^{th} entry in the Suspect Flows Array

structures, $m = m_{CS} + m_{SFA} + m_{SEA}$. The algorithm performance strongly depends on this partition of the memory.

Some partitions are more logical than others. For example, it is intuitive that we are interested in having the Shared Estimators Array much smaller than the Candidate Set ($m_{SEA} << m_{CS}$). Otherwise, one can hold an exact counter for every flow in the Candidate Set without paying the penalty of the relative error introduced by the estimators. The partition between m_{CS} and m_{SFA} is one of the tuning parameters of the algorithm.

1) Shared Estimators Array Initialization: In [22], the initialization of the shared estimators' values is based on the parameters δ and L, and starts from $v_0 = 0$. The rest of the values are calculated in a way that guarantees a maximal relative error of δ. Since we are interested only in estimating the HH flows, we can start from $v_0 = v\phi N$, where N is the number of packets, ϕ is the threshold of HH flows, and v is the "propagation parameter". The motivation behind this initialization is that in the Candidate Set we monitor only large flows, thus there is no need to waste space for allocating smaller estimators. The "propagation parameter" is a key tuning parameter, which affects the initial estimated value of flows that were just propagated to the Candidate Set and controlling its value is important to bound some of the error types as we discuss later.

We show that we can generalize the CEDAR scheme to start from a minimal estimator greater than 0 without affecting its properties or incurring additional memory and processing time costs. For that, we show how to build such a generalized scheme on top of the original scheme with the aid of an additional variable z. We denote by v'_i the estimator values of the generalized scheme and by v_i the values of the original scheme.

We initialize the original scheme and set $z = v'_0$ and $v_0 = 0$. When querying a flow with estimation of v_i, we return $v'_i = v_i + z$. When probabilistically incrementing a flow's estimator, we do so in the same manner as the original scheme, i.e., by incrementing with probability of $\frac{1}{v_{i+1}-v_i}$ which is equal to $\frac{1}{v'_{i+1}-v'_i} = \frac{1}{v_{i+1}-z-v_i+z}$. The costs of this

modification with respect to the original scheme are very small: (1) holding the additional variable z, (2) a single assignment in the initialization, and (3) a single read of the variable z in each estimation query.

Given m_{SEA} memory for the Shared Estimators Array, we calculate the width of a single estimator to be $q = \lceil \log_2(10\phi N) \rceil$ and $L = \lfloor \frac{m_{SEA}}{q} \rfloor$. This assumes that the maximal value ever achieved by a flow is 10 times the threshold, while this is usually true and this value is never achieved, we note that the cost of increasing it is logarithmic. This means that doubling the maximal value increases q by one and reduces the number of estimators, L, by a negligible value of $\lceil \frac{M_{SEA}}{q*(q+1)} \rceil$.

2) Suspect Flows Array Initialization: The Suspect Flows Array is initialized as an empty array without any stored flow id. Furthermore, we initialize a hash function $h_1 : F \rightarrow k$, which maps every flow id to its entry in the Suspect Flows Array.

Given m_{SFA} memory for the Suspect Flows Array, we calculate the size of the array using $k = \lfloor \frac{m_{SFA}}{|flow\_id|} \rfloor$. Where $|flow\_id|$ is the number of bits needed to store the flow identifier and in the case of IPv4 5-tuple flows <IP_SRC, IP_DST, SRC_PORT, DST_PORT, PROTOCOL> it is 104.

3) Candidate Set Initialization: The Candidate Set is initialized as an empty array that does not hold any flow id and all indexes are initialized to 0. That is, there is no flow that the algorithm thinks is a candidate to be a HH flow and all of the initial estimations are v_0. Furthermore, we initialize a hash function $h_2 : F \rightarrow M$, which maps every flow id to its entry in the Candidate Set.

Given m_{CS} memory for the Candidate Set, we calculate the size of the set using $M = \lfloor \frac{m_{CS}}{|flow\_id| + \lceil \log_2 L \rceil} \rfloor$.

The insertion operation is triggered when a packet of a 5-tuple flow is encountered, thus it is important to keep this operation an $O(1)$ internal steps. For a packet from flow f, the insertion goes as follows:

with probability $\frac{1}{2v_0}$ perform the following:

1) if $CS_{h_2(f)} = (f, i)$ for some i, perform the following $2v_0$ times:
   a) increase i by 1 with probability $\frac{1}{v_{i+1} - v_i}$.
   b) if i reaches $L-1$ perform $ShiftUpEstimators()$ and stop.
2) if not, calculate the index $h_1(f)$:
   a) if $sf_{h_1(f)} \neq f$ or $sf_{h_1(f)}$ is $Null$, assign the flow identifier of f to $sf_{h_1(f)}$.
   b) if $sf_{h_1(f)} = f$, then:
      i) evacuate the entry from the array, $sf_{h_1(f)} = Null$.
      ii) if $CS_{h_2(f)}$ is empty then put $(f, 0)$.
      iii) else perform $EvictFromCandidateSet(f)$.

The main idea behind the insertion step is to first check if the current flow is already present in the Candidate Set; if it is present then we should increase its estimator. Since we consider one out of $2v_0$ packets, then when increasing the estimation we perform it $2v_0$ times.

If i reaches $L-1$ then this flow was advanced up to the highest estimator in Shared Estimators Array. Thus, there is a need to update the Shared Estimators Array to include more estimators and this is performed by $ShiftUpEstimators()$ as follows:

1) calculate v_L based on the original CEDAR scheme.
2) append v_L and pop v_0 from the array.
3) set v_0 to the value of the new minimal estimator.
4) go over the Candidate Set entries and remove flows pointing to the previous minimal value.

In the current architecture step 4 is not an $O(1)$ operation, however, we can simply modify the insertion operation to allow step 4 to be an $O(1)$ operation. Instead of searching for and removing flows from the Candidate Set, we store an additional variable, y that stores how many shifts were already performed. Then when checking if a flow is in the Candidate Set, we also check that its i is bigger than the stored number of shifts. If not, then we consider the entry to be empty and simply overwrite it. This is correct, since if $i < y$ then this flow should have been evicted in the i^{th} $ShiftUpEstimators$ operation.

If the flow we are interested in is not in the Candidate Set then we check if it is in the Suspect Flows Array. If the flow is not in the Suspect Flows Array then we add it to the appropriate entry, $sf_{h_1(f)}$. This way we indicate for future appearances of packets from this flow that this flow is "recent" enough. We note that adding a flow to the Suspect Flows Array might evict other flows that share the same hash entry.

If it is already in the Suspect Flows Array then we propagate it to the Candidate Set if its entry in the Candidate Set is indeed empty. When propagating we set the initial estimation to be $v_0 \neq 0$. The reason for this estimation is that we insert it after two packets, that did not have a hash collision in between their appearances, and all packets are sampled with a probability of $\frac{1}{2v_0}$.

However, it is possible that such a propagated flow will be hashed to an already allocated entry, in this case we should decide if to evict the older flow. The algorithm decides on the eviction with probability of $\frac{v_0}{v_i}$ where v_i is the estimation of the older flow. This is described in $EvictFromCandidateSet(f)$ as follows:

1) get $(g, i) = CS_{h_2(f)}$
2) with probability $\frac{v_0}{v_i}$, set $CS_{h_2(f)} = (f, i)$

The motivation behind this probabilistic eviction is that the potential to-be evicted flow had already gained some significant amount of packets previously, and simply evicting it might be the wrong decision. Thus, we evict it only if the new flow has a comparable amount of packets. That is, in expectation it requires $v_i = \frac{1}{2\frac{1}{2v_0}\frac{v_0}{v_i}}$ packets to replace the old flow, and that is why we insert the new flow with the same current estimation of the old flow and not with the minimal estimation.

A. Query a specific flow

When querying a specific flow, the algorithm calculates $h_2(f)$ and checks if $CS_{h_2(f)} = (f, i)$ for some i, if yes it returns v_i.

For flows that are not present in the Candidate Set the algorithm returns zero since it provides no guarantee on the relative error of non HH flows.

B. Query all Heavy Hitter Flows

When querying the data structure for all the HH flows, the algorithm searches the Shared Estimators Array for the index of the minimal estimator of value higher than $(1-\delta)\phi N$. Then, it extracts all tuples (f, i) with $i's$ higher than that index. Each tuple means that the flow f has an estimation of v_i.

We note that the step of finding the index of the minimal estimator with a value higher than $(1-\delta)\phi N$ is not an $O(1)$ operation. However, since this query usually happens offline, it is not significant. To support an $O(1)$ version of this query, it is possible to have an extra variable holding the current index of the minimal estimator with a value higher than $(1-\delta)\phi N$ at the initialization of the Shared Estimators Array and update it in every $ShiftUpEstimators()$ operation.

III. Analysis of Errors

In this section we study the types of errors introduced by our algorithm and provide an analysis for each of them. While each one of the papers [21], [22] provides guarantees regarding the performance, their combination by our algorithm introduces new aspects that we identify and analyze. In order to do so we denote by $F_{h_1(f)}$ the set of flows that are mapped to the same entry by h_1 as f, i.e., $F_{h_1(f)} = \{g \in F | h_1(g) = h_1(f)\}$, and by γ_f the portion of flow f from the whole trace, i.e., $\gamma_f N$ is the number of packets of flow f. Furthermore, we denote by Γ_f the sum of portions of flows mapped to the same entry as f by h_1, i.e., $\Gamma_f = \sum_{g \in F_{h_1(f)}} \gamma_g$.

The total error in the frequency estimation of a HH flow might originate from the various parts of the algorithm. The first type of error is the *estimation error* which originates from the usage of shared estimators in the Shared Estimators Array. This relative error is bounded by δ as guaranteed from the original CEDAR scheme.

The second error type, which we denote by *propagation error*, is the error introduced to the estimation of HH flows for not propagating early enough from the Suspect Flows Array to the Candidate Set. While indeed, the algorithm will give a frequency estimation within the relative error of the true frequency for any flow in the Candidate Set, however, this estimation holds from the moment the flow was propagated to the Candidate Set, and this propagation does not happen on the first packet of the flow. The propagation error captures the number of times the flow appeared in the trace before propagating to Candidate Set.

When examining the i^{th} and the $i+1^{th}$ packets of a HH flow f (for $i \geq 2$), the probability of f being evicted from the Suspect Flows Array before propagating to the Candidate Set is $\frac{\Gamma_f - \gamma_f}{\Gamma_f}$. This is true, since $\Gamma_F - \gamma_f$ is the portion of the flows but f that are mapped to the same entry as f. Furthermore, the probability of a HH flow f not to propagate to the Candidate Set after $i+1$ packets is $\left(\frac{\Gamma_f - \gamma_f}{\gamma_g}\right)^i$, which approaches 0 as more packets for flow f arrive.

The last type of error, which we denote by *eviction error*, is the error introduced to the estimation of HH flows due to the eviction of such flows from the Candidate Set. These evictions happen when there is a collision in h_2, then the algorithm evicts the currently tracked flow with a low probability of $\frac{v_0}{v_i}$. The motivation of evicting with this probability is to replace the current flow, only if the new flow had arrived many times. However, evictions of HH flows could still occur due to the possible collisions in h_2 with other HH flows.

Since h_2 is a uniform hash function, the expected number of collisions between HH flows is given by $\frac{h(h-1)}{2M}$ where h is the number of HH flows and M is the number of entries in the Candidate Set. It is clear that the more memory used for the Candidate Set the less expected collisions we will have in h_2 and the smaller the eviction error.

IV. Exact Bank Algorithm

In this section we present a simplification of the Candidate Set algorithm that has better performance when the trace is heavy tailed, meaning that most of the traffic is due to a small number of flows.

This algorithm holds a similar Suspect Flows Array as the Candidate Set algorithm, however, it lacks the Candidate Set itself and the Shared Estimators Array. Instead, it holds a constant size bank of exact counters. The main motivation is that in heavy tailed traces, most of the traffic is produced by the HH flows, thus it is possible to allocate about $\frac{1}{\phi}$ exact counters to measure these flows.

The usage of the Suspect Flows Array assures that we are not propagating too many flows in the Exact Bank and only measure "heavy" flows. The cost of this simple setup manifests in the insertion operation that requires searching the $\frac{1}{\phi}$ entries to find the appropriate entry.

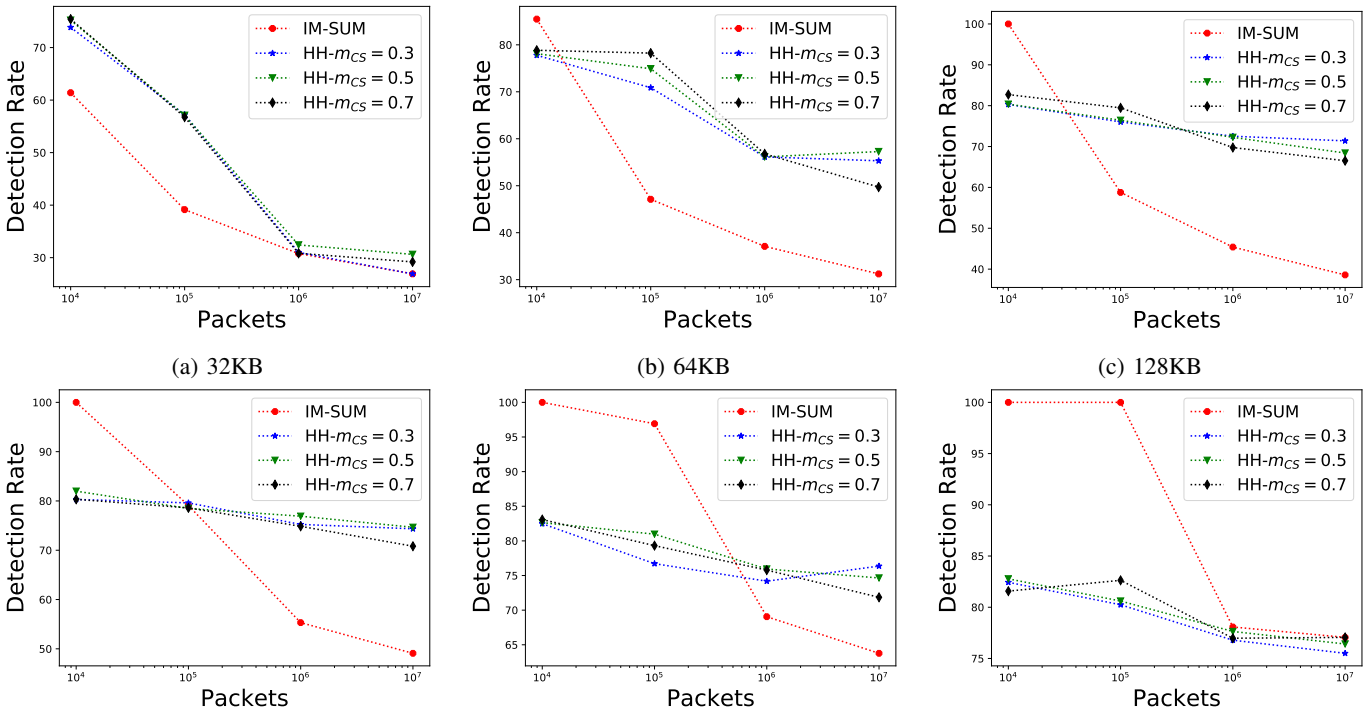
V. Evaluation

A. Settings

In order to evaluate the performance of our algorithms, we implemented them and a version of the IM-SUM algorithm from [15]. We evaluated the algorithms on the New-York 2019 dataset of CAIDA Anonymized Internet Traces [23]. This dataset was chosen as a recent trace of real traffic recorded on a high throughput 1Gbps backbone link of a Tier1 ISP (between New York, NY and Sao Paulo, Brazil).

For all of the evaluation, unless stated otherwise, we used the following: (1) IPv4 5-tuple flows with $|flow\_id| = 104$, (2) HH threshold, $\phi = 0.001$[1], (3) $\delta = 0.05$, (4) each datapoint is the average of 10 runs, each starting in a random packet in the trace, (5) the Shared Estimators Array memory is 1% of

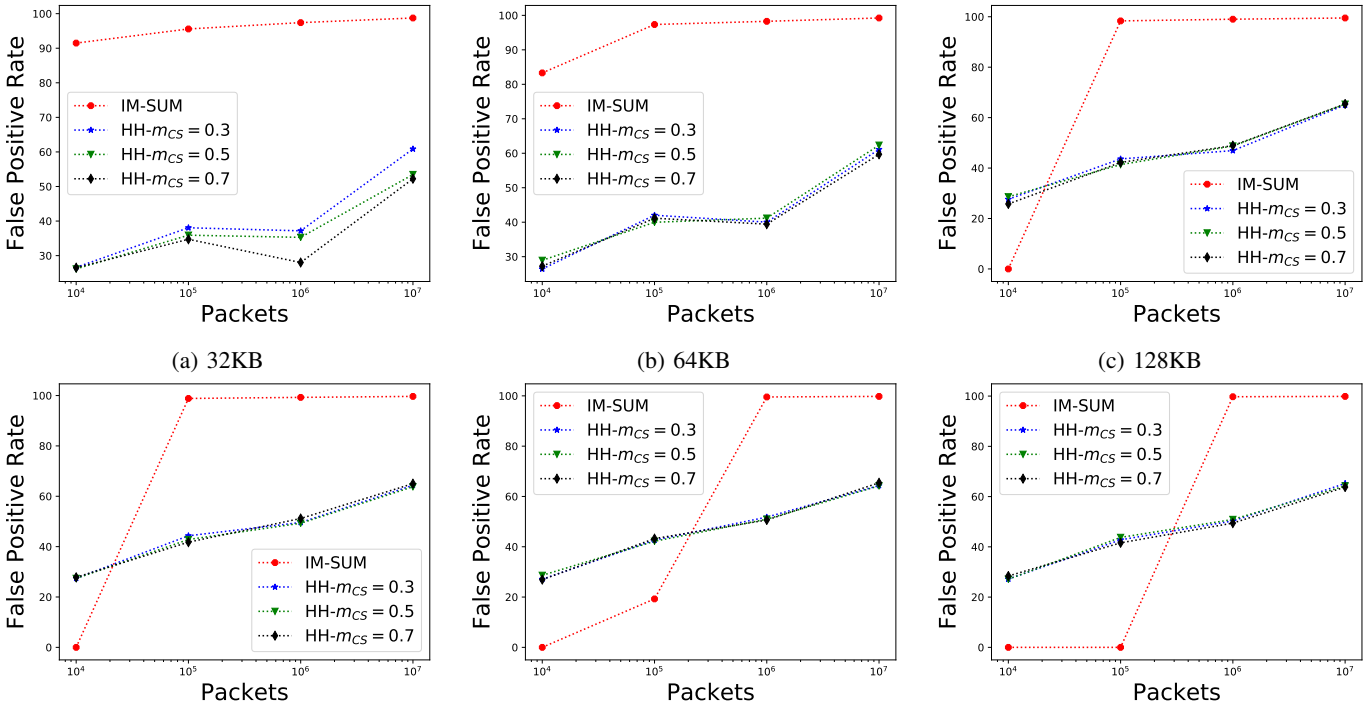
[1]This threshold is commonly used in the literature, on our data it provides a good trade-off between the number of HH (in the hundreds range) and their importance.



(a) 32KB (b) 64KB (c) 128KB

(d) 0.25MB (e) 0.5MB (f) 1MB

Fig. 2: The Average Detection Rate as function of number of packets, comparing our algorithm in three different settings vs. the Elephants algorithm for $\phi = 0.001, \delta = 0.05$



(a) 32KB (b) 64KB (c) 128KB

(d) 0.25MB (e) 0.5MB (f) 1MB

Fig. 3: The Average False Positive Rate as function of number of packets, comparing our algorithm in three different settings vs. the Elephants algorithm for $\phi = 0.001, \delta = 0.05$

the available memory ($m_{SEA} = 0.01$) and (6) a propagation parameter, $v = 0.35$ (our analysis showed that this value balances performance and Throughput best).

To measure the performance of the algorithms we used the following metrics:

- Detection Rate (DR), the same as the previously mentioned Recall - which is the ratio of HH flows that the algorithm detected.
- False Positive Ratio (FPR) - which is the ratio of non HH flows reported by the algorithm as HH from the total number of reported flows.
- Throughput - the number of insertions the algorithm can perform per a single millisecond.

Using these metrics, we compared our algorithm against the IM-SUM algorithm which in its turn performs better than the following previously suggested streaming algorithms [17], [18], [19], [20].

To be able to compare the algorithms we set the total memory available to use and the accuracy parameter. The memory usage of the IM-SUM algorithm is determined by the accuracy parameter, ϵ, and its performance parameter, γ, which tunes the algorithm's memory-speed trade off. In each run of IM-SUM, the amount of memory and the accuracy parameter was set, and we calculated the appropriate γ for this run. For the Candidate Set algorithm, the only tuning parameter is the partition of the left 99% of the memory between the Candidate Set and the Suspect Flows Array, i.e., setting m_{CS} and $m_{SFA} = 0.99 - m_{CS}$.

The Candidate Set algorithm and the IM-SUM algorithm provide different accuracy guarantees on the reported set of flows. The IM-SUM algorithm ensures accuracy in the range $[N(\phi - \epsilon), N\phi]$ while the Candidate Set algorithm ensures accuracy in the range $[(1-\delta)\phi N, \phi N]$. The effective accuracy slack of the IM-SUM algorithm is $N\epsilon$, which is much larger than Candidate Set's slack of $N\delta\phi$. Therefore, comparing the algorithms with the same accuracy parameter, $\epsilon = \delta$, gives an edge to the IM-SUM algorithm, allowing it to possibly report more flows as HH than the Candidate Set algorithm.

B. Results

Figure 2 shows the average DR of the algorithms as a function of the number of packets processed. Intuitively, the DR relies heavily on the amount of available memory and one can see that the more memory allocated to the algorithms, the higher the DR rate is. When evaluating the Candidate Set algorithm, we tested several variants with the different partition of the memory between the Candidate Set and the Suspect Flows Array. These variants allocates 30%,50%,70% of the memory to the Candidate Set and 69%,49%,29% to the Suspect Flows Array accordingly.

A priory It is not clear how this trade off between the size of the Candidate Set and the size of the Suspect Flows Array affects the performance of the algorithm. When the Candidate Set is large the algorithm can hold on to "big" flows while when Suspect Flows Array is large the algorithm can hold on to the more "recent" flows. One can see that for balanced partitions, not too few entries in the Suspect Flows Array nor in the Candidate Set the effect on DR is small.

Sub-figures 2a and 2b show that the Candidate Set performs better than IM-SUM algorithm for amounts of memory smaller than 64KB and in some settings even finds 50% more HH flows. However, it is worthy to note that both algorithm's DR degrades when processing more packets in this range of memories, which is unexpected since none of the algorithms guarantees relies on N. Sub-figures 2c to 2f show that this phenomenon of DR degradation when the number of packets increase is not present in the Candidate Set algorithm compared to the IM-SUM algorithm.

However, this is not the case for the IM-SUM algorithm. The more memory the algorithm has the higher the DR, while the more packets processed the lower the DR. That is, for a given amount of memory our algorithm can process more packets without compromising its DR.

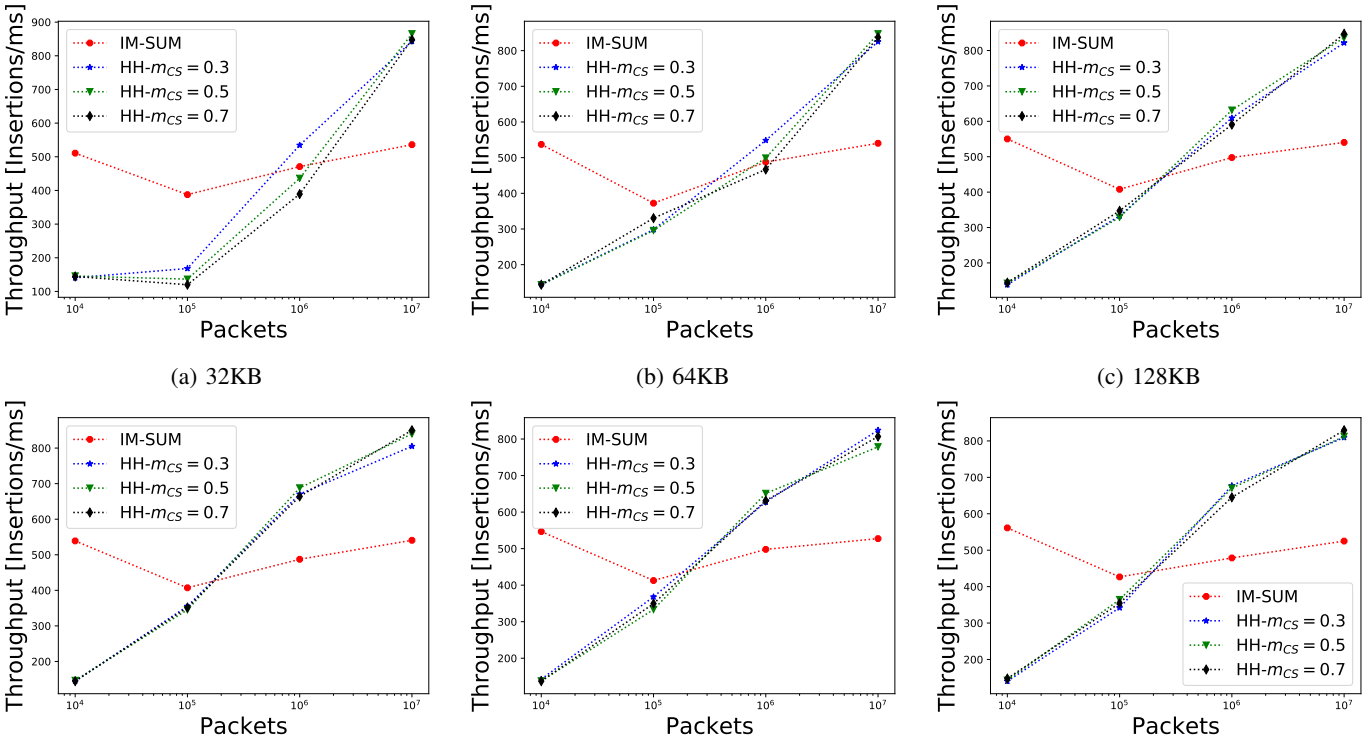
Figure 3 shows the average FPR of the algorithms as a function of the number of packets processed. It is worthy to note that the FPR of the Candidate Set algorithm does not depend on the amount of memory, that is, using more memory does not affect the FPR. This is true since the algorithm's FPR depends directly on the estimation given by the Shared Estimators Array, and this estimation is always accurate up to $1 \pm \delta$, regardless of the amount of memory. This also explains the lack of effect of the value of m_{CS} on the FPR.

Interestingly, when the IM-SUM algorithm is given enough memory and not many packets, it will not report false positive flows. However, for each memory settings there is a number of packets where the algorithm starts reporting almost 100% false positive flows, this means that the vast majority of the reported set of flows is non HH flows.

When considering the comparison of the algorithms in terms of FPR, it is evident that the Candidate Set algorithm performs better. For less than $64KB$ it is always less than IM-SUM's FPR, while for larger amounts of memory it stays pretty constant in the range 40%-60% while IM-SUM's FPR soars quickly to around 100%. It is worthy to note that while these values of FPR are unacceptable in security applications, increasing the memory of the Shared Estimators Arrayand decreasing its maximal error should lower the FPR.

Figure 4 shows the throughput in terms of insertions per millisecond of the algorithms as a function of the number of packets processed. For both algorithms, the more packets processed for a given amount of memory, the higher the throughput. In the case of IM-SUM, it seems that for $N = 10^4$ we get the highest throughput. However, this datapoint is an anomaly since, with that few packets, the algorithm does not perform the heavy maintenance operation.

In the case of the Candidate Set algorithm, the more packets we process the higher the throughput due to the fact that more packets yields higher v_0 and higher values in the Shared Estimators Array. This is translated to fewer operations per packet since its insertion is probabilistic with an inverse relation to the values in the Shared Estimators Array. This is also supported by the fact that the Candidate Set algorithm performs $O(1)$



(a) 32KB (b) 64KB (c) 128KB

(d) 0.25MB (e) 0.5MB (f) 1MB

Fig. 4: The Average Throughput as function of number of packets, comparing our algorithm in three different settings vs. the Elephants algorithm for $\phi = 0.001, \delta = 0.05$

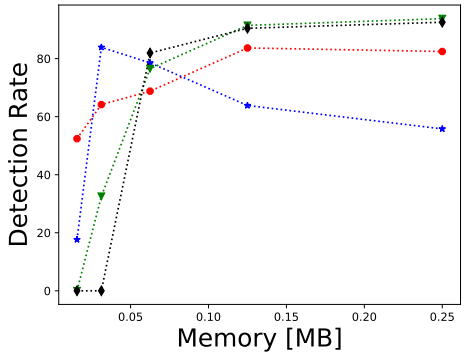


Fig. 5: The Average Detection Rate as function of memory, comparing our the Candidate Set algorithm and the Exact Bank algorithm with different sizes of banks ($\phi = 0.001, \delta = 0.05, N = 10^7, m_{CS} = 0.5, m_{SEA} = 0.01$)

"maintenance" operations compared to IM-SUM amortized maintenance operation. It is worth to note that the throughput of our algorithm is not affected by the amount of memory, that is true since all of the accesses to the data structures are $O(1)$ and does not rely on the sizes of these data structures.

Figure 5 compares the DR of the Candidate Set algorithm with the Exact Bank algorithm with three sizes of banks 1000, 2000, and 3000. Since every 1000 entries in the Exact Bank algorithm require around $16KB$, the algorithm suffers from poor DR in the smaller ranges of the memory. However, once the memory allows a decent size of the Suspect Flows Array, the Exact Bank algorithm starts to perform better since many more HH flows are propagating to the bank and have now an exact estimation up to the propagation error introduced by the Suspect Flows Array. One should note, that for Exact Bank with size 1000 the DR becomes poor once again the larger the Suspect Flows Array, that is since larger Suspect Flows Array is propagating too many flows to the Exact Bank and without an eviction. It is important to note, this higher DR comes with a cost of lower throughput and higher FPR since it performs a linear search in the size of the bank (which is constant) and it returns all flows in the Exact Bank as potential HH.

VI. Conclusions

In this paper, we presented a new practical algorithm for detecting Heavy Hitter flows. Our algorithm is deployable on off-the-shelf network gear, and is guaranteed to preform at most a constant number of operation per packet. Furthermore, we evaluated our algorithm on real internet traces and showing that it performs better, in terms of Detection Rate, False Positive Ratio and Throughput, than best existing algorithms in various settings. More specifically, when processing 10^7 packets our algorithm performs better for any amount of memory less than $1MB$. This also holds for $10^6, 10^5$ packets with memories of $0.5MB, 0.25MB$ respectively.

REFERENCES

[1] 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.

[2] 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.

[3] G. Dittmann and A. Herkersdorf, "Network processor load balancing for high-speed links," in *Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, vol. 735, 2002.

[4] Y. Fairstein, D. Harris, J. Naor, and D. Raz, "Nfv placement in resource-scarce edge nodes," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 51–60.

[5] 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.

[6] J. Moraney and D. Raz, "On the practical detection of hierarchical heavy hitters," in *2020 IFIP Networking Conference (Networking)*. IEEE, 2020, pp. 37–45.

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

[8] 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.

[9] 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.

[10] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Randomized Admission Policy for Efficient Top-k and Frequency Estimation," in *The 36th Annual IEEE International Conference on Computer Communications*, 2017.

[11] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, "Computing iceberg queries efficiently." in *Internaational Conference on Very Large Databases (VLDB'98), New York, August 1998*. Stanford InfoLab, 1999.

[12] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Surfing wavelets on streams: One-pass summaries for approximate aggregate queries," in *Vldb*, vol. 1, 2001, pp. 79–88.

[13] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Transactions on Database Systems (TODS)*, vol. 28, no. 1, pp. 51–55, 2003.

[14] E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *Proceedings of the 10th Annual European Symposium on Algorithms*. Springer-Verlag, 2002, pp. 348–360. [Online]. Available: http://dl.acm.org/citation.cfm?id=647912.740658

[15] 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.

[16] M. Zadnik and M. Canini, "Evolution of cache replacement policies to track heavy-hitter flows," in *International Conference on Passive and Active Network Measurement*. Springer, 2011, pp. 21–31.

[17] 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.

[18] P. Cheung-Mon-Chan and F. Clérot, "Finding hierarchical heavy hitters with the count min sketch," in *Proceedings of 4th International Workshop on Internet Performance, Simulation, Monitoring and Measurement, IPS-MOME*, 2006.

[19] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

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

[21] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proceedings of the 2nd Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM 2002. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2002, pp. 323—-336. [Online]. Available: https://doi.org/10.1145/633025.633056

[22] E. Tsidon, I. Hanniel, and I. Keslassy, "Estimators also need shared values to grow together," in *2012 Proceedings IEEE INFOCOM*. IEEE, 2012, pp. 1889–1897.

[23] "The CAIDA UCSD Anonymized Internet Traces 2019 - equinix-nyc 2019-01-17, Direction A." https://www.caida.org/data/monitors/passive-equinix-nyc.xml.