Dynamic Sketch: Efficient and Adjustable Heavy Hitter Detection for Software Packet Processing

Yipeng Wang*, Tong Yang†, Ren Wang*, Charlie Tai*
*Intel Labs
{yipeng1.wang, ren.wang, charlie.tai}@intel.com
†Peking University
yangtongemail@gmail.com

***Abstract*—Heavy hitter detection is a key task for networking traffic profiling, which can be used for various purposes such as Denial of Service (DoS) attack detection, Quality of Service (QoS) scheduling, load balancing, and flow size based routing, etc. Over the years, many efforts have been made on designing data structures and algorithms to achieve fast and memory-efficient inline profiling in cloud networks. Traditional heavy hitter detection methods, however, yield an innate and nonadjustable profiling accuracy (i.e., false positive or false negative) once the data structure is initialized. Users have no runtime feedback information nor control on the profiling accuracy, which could be an important factor for their usages.**

In this paper, we propose and evaluate a novel dynamic and memory-efficient heavy hitter detection algorithm, called Dynamic sketch. Dynamic sketch performs runtime accuracy monitoring and provides feedback to users via a sampling based method. It also self-adjusts the accuracy at runtime to satisfy the target given by the user. We implemented Dynamic sketch and our evaluations show that Dynamic sketch is able to report profiling accuracy with only a minimal 2% performance overhead. In addition, Dynamic sketch is 2.35× faster than the state-of-the-art hash table based heavy hitter detector and achieves more than 2× memory efficiency than the state-of-the-art sketch based implementation.

I. INTRODUCTION

The rapid growth of Software Defined Networking (SDN) and Network Virtualization deployed in cloud poses great challenges on high-speed software packet processing. Along with networking functions such as switching and load balancing, monitoring the traffic characteristics at real time is critical for network management and performance optimization. One common and important aspect of traffic monitoring in the cloud is heavy hitter detection, with the goal to identify the set of "heavy" (or elephant) flows which have much higher packet count or bandwidth consumption comparing to other lighter (or mice) flows. The applications of heavy hitter detection include DoS attack detection, flow-size based routing and planning, billing and charging, and QoS for flow scheduling [1]–[7]. Figure 1 shows an example heavy hitter detector that co-locates with the virtual switch to profile traffic destined to multiple services on a single platform.

Data streaming algorithms [8] are a class of algorithms that can be used for heavy hitter detection. These algorithms, however, may not be directly applicable to inline networking traffic profiling tasks. Inline traffic profiling poses stringent speed and memory efficiency constraints to achieve runtime and low overhead monitoring of huge number of flows. Additional processing of traffic in the middle of datapath may trigger undesirable overhead thus negatively impact the performance. Hence, heavy hitter detection algorithms need to be extremely memory efficient and fast.

978-1-7281-4832-8/19/$31.00

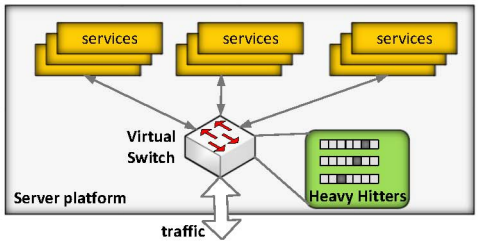


Fig. 1. Heavy hitter detector that co-locates with virtual switch to profile traffic destined to multiple services on a single platform.

Many research efforts have been put forth on the area of fast and efficient heavy hitter detection. From earlier counter array based data structure [9] to more recent sketch based data structure [10]. Most recently, researchers propose to combine various data structures to support more features [11]–[13], or apply specialized data structure for hardware switches [7]. However, current solutions have two drawbacks. First, these algorithms do not provide runtime accuracy estimation to the user thus the users have no information on the quality of currently reported results. Second, these algorithms yield innate and nonadjustable accuracy rates once data structure is initialized while users may have different accuracy targets. For example, with different usages, users may prefer either higher precision rate (less false positives) or higher recall rate (less false negatives).

Motivated by the need to provide high performance monitoring with adjustable accuracy for the cloud networks, in this paper, we propose a new heavy hitter detection algorithm inspired by Elastic sketch [12], called Dynamic sketch. Specifically, 1) Dynamic sketch is a new and enhanced sketch data structure which is much more memory efficient comparing to traditional sketch designs. 2) We propose and implement a novel mechanism to efficiently estimate the profiling accuracy at runtime so users understand the quality of the profiling

results. 3) Dynamic sketch is able to adjust the profiling accuracy for different usages during runtime to meet the user-defined accuracy. To the best of our knowledge, Dynamic sketch is the first heavy hitter detector that has such capabilities. Moreover, we also propose a more comprehensive metric to evaluate the quality of heavy hitter detection algorithms.

We implement and evaluate our design with Intel Xeon processor and real traffic traces. The experiment results show that Dynamic sketch can self-monitor its accuracy with only a minimal 2% performance overhead. Meanwhile, it is 2.35× faster than the state-of-the-art hash table based heavy hitter detector and achieves more than 2× memory efficiency than the state-of-the-art sketch based implementation.

II. Design of Dynamic sketch

In this section, we describe the detailed design of Dynamic sketch algorithm, including: 1) overview of the design of Dynamic sketch, 2) the *Door keeper* mechanism to realize high memory efficiency, 3) the sampling based mechanism for runtime accuracy estimation, and 4) a closed-loop feedback control mechanism using both the Door keeper and sampling method to dynamically achieve the user-defined adjustable accuracy. We use the terms of precision rate (PR), recall rate (RR), false positives, and false negatives frequently throughout the paper when we discuss the rationale behind our design choices. PR refers to the percentage of heavy hitter flows that are reported by the algorithm are real heavy hitters, and RR refers to the percentage of real heavy hitters are indeed reported by the algorithm. False positive means a light flow gets mis-classified as a heavy flow, while false negative means the opposite. Thus, more false positives means lower PR, and more false negatives means lower RR. These are critical measurements to evaluate the quality of heavy hitter detection algorithms.

A. Overview of Dynamic sketch

Figure 2 shows the high level scheme of Dynamic sketch algorithm. The data structure is based on Elastic sketch [12] and composed of three major parts: 1) a bucket-based hash table that contains information of heavy flows, 2) an additional table containing a series of sampling buckets for online accuracy monitoring, and 3) a count-min sketch as fall-back data structure to capture heavy flows that temporarily fall out of the hash table. For profiling heavy hitters, packet headers go through the data structure and necessary information gets recorded. After every profiling window, keys in the hash table will be read out for analysis. If the estimated count of a key is above certain threshold, it is reported as heavy flow.

The hash table data structure is similar to the one proposed in Elastic sketch [12]. It is supposed to keep all the heavy flow keys and their counts, thus it is also called the "heavy part". It is composed of an array of buckets with each bucket containing multiple keys. Each key in the hash table has a private counter field called $vote+$, and there is a shared counter called $vote-$ per each bucket. For every incoming packet, the packet header (i.e. flow key) will be hashed first to find the index of the bucket. If this key already exists in the bucket (i.e. a table hit), the corresponding key will increment its $vote+$ by 1. If not hit but there is an empty entry in the bucket, the key will be inserted there and its $vote+$ counter is initialized to be 1. If all the entries in this bucket are occupied by existing keys, one of the keys that is not heavy enough will be replaced by the new key. To find the key to be replaced, $\frac{vote-}{vote+}$ is calculated for each existing key. If the result is greater than a certain ratio, this key is deemed "light" and voted out by the new incoming key. If none of the existing keys is light, the new key will not replace any of the existing keys, however, the $vote-$ will be incremented by 1. Intuitively, a key has to be heavy enough to stay in the hash table.

The keys that are either voted out of the hash table or missed the hash table, will be fed into a count-min sketch (CMS) [10], which is also the "light part". A count-min sketch is a 2-D array of counters that count the frequency of keys. Flow keys are hashed and corresponding counter in each row of the counter arrays is incremented. The final count of the key that ends up in the CMS is the minimum of all corresponding counters. For example in Figure 2, based on three hash values of the key, counter 4 in row one, counter 3 in row two, and counter 7 in row three are incremented. In this example, the estimated count of the key will be 11 since it is the minimum among all the three counters. As a sketch data structure, CMS could over-estimate the flow count due to hash collisions, which contributes to the false positives.

Many studies use CMS alone to implement a heavy hitter detector. However, comparing to a hash table, CMS needs to access multiple cache lines for each update, and requires an extra sorted data structure to store the heavy keys. Here, the CMS is used as the fall-back data structure to capture heavy hitters that fall out of the hash table. The reason is that the voting mechanism described above is not perfect. Heavy flows can be voted out of the hash table temporarily by highly bursty light flows which leads to false negatives. Although those light flows are light across the whole profiling window, they could be heavy in a short period of time with bursty behavior. They accumulate $vote-$ quickly enough to evict potential heavy flows. This is especially true for short connections that happen frequently in mobile network and data centers. Without the CMS, the false negative rate could be high. While with the CMS, the heavy flows that are occasionally voted out of the hash table will not lose any information. When the heavy flow gets back to the hash table, all the history count of this flow is still recorded in the CMS, thus the false negative rate is reduced. At the end of each profiling window, the total count of the key will be estimated by adding the key's $vote+$ from the hash table to the count getting from the CMS.

Besides the hash table and the CMS, sampling buckets shown in the middle of the figure are used to monitor the profiling accuracy. Each sampling bucket has the same data structure as the regular bucket of the hash table but of larger size. Every key that hits the sampled buckets in the hash table will also be fed into the sampling buckets. The sampling buckets are responsible for estimating the accuracy of the

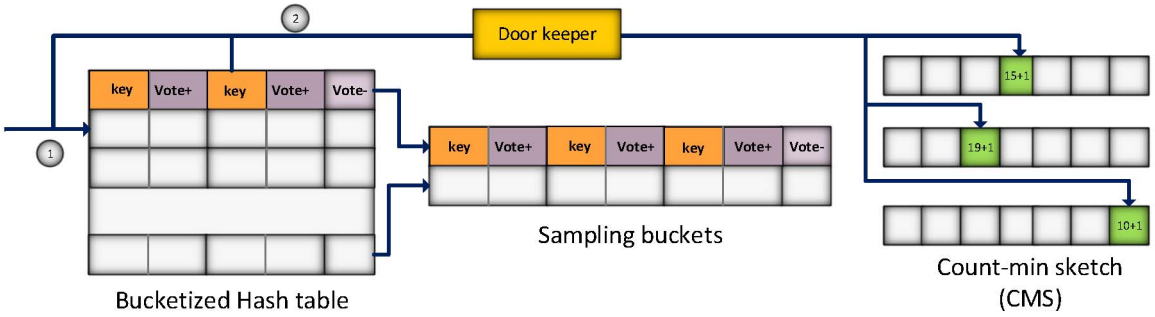


Fig. 2. The data structure of Dynamic sketch. ①: flow is input into the data structure; certain buckets are sampled for accuracy monitoring. Certain flows that go to those sampled buckets are also added into the sampling buckets. ② flow that misses the hash table or flows voted out of the hash table will be fed into the CMS, corresponding counters are incremented.

current profiling window to provide useful feedback to the user. We will discuss more details of the sampling buckets later.

B. Door Keeper to Adjust Accuracy

One issue of the above described baseline algorithm is that the CMS has to be very large to not over-estimate the packet count too much. Over-estimation in the CMS could cause huge false positive rate. In Elastic sketch with similar design, the CMS has to use $2\times$ more memory space than the hash table to keep a relatively good false positive rate.

To reduce the size of the CMS to be more memory efficient, we propose the *Door keeper* mechanism. The purpose of Door keeper is to keep light flows out of the CMS as much as possible while still receive heavy flows that are accidentally voted out of the hash table. With Door keeper, the CMS only accepts flows that are: 1) evicted from the hash table with a $vote+$ larger than the door keeping threshold, and 2) miss the hash table but hit CMS with non-zero counter. The first condition is to limit the flows allowed into CMS to be heavy flows only. We observe that most light flows never enter the hash table. The flows that get evicted from hash table with a large count are more likely to be heavy flows. The second condition is to guarantee that heavy flows that have already been evicted out of the hash table keep getting updates during its temporary stay in the CMS.

Adjustable Accuracy One major benefit of the Door keeper algorithm is that the door keeping threshold directly trades off between the false positive and false negative rates, and it can be tuned during runtime. When the threshold is high, the data structure effectively becomes a hash-table only algorithm since very few flows can make it into the CMS. In this case, the false positive rate is minimal since the over-estimation caused by CMS is negligible. On the other hand, with a low threshold, we conservatively allow many flows to enter the CMS. In such case, Dynamic sketch regresses to the baseline algorithm without Door keeper, thus the false negative rate will be smaller (but false positive rate becomes higher).

With Door keeper, users can adjust false positive and false negative rates w.r.t. different use cases even during runtime. For example, for DoS detection, low false negative is much more critical than low false positive. In such case, the threshold should set to be low. On the other hand, for flow-size based routing and planning, both false negatives and false positives are important. Thus the threshold should be moderate. Later we will describe a closed-loop feedback mechanism to adjust the door keeping threshold automatically during runtime, to achieve accuracy targets specified by the users.

C. Bucket Sampling for Accuracy Monitoring

Besides Door keeper, we design a sampling method to monitor the profiling accuracy. None of the existing heavy hitter detection algorithms can report profiling accuracy to users. There is no estimation on false positive and false negative rates of current profiling results and users have no means to flexibly balance between memory cost and accuracy in real-time, which could be important for high level decision making. Many traditional streaming algorithms provide certain mathematical error bound but it is difficult to relate the bound to false-positive and false-negative rate of a running sketch, which could be very different from the theoretical error bound. For example, many sketch-based algorithms provide a so-called (ϵ, δ) error bound. It means that the estimated packet count will fall into the error bound of ϵ with a probability of δ. With fixed memory space, one could choose to either have tighter ϵ or higher δ. It is hard to decide which yields better accuracy for different workloads. Thus, it is ideal for a monitoring based mechanism to provide users with more insight instead of only providing the loose mathematical bound.

Hence, we propose a sampling based mechanism on the hash table to estimate the false positive and false negative rates of the current sketch at runtime. The proposed method is very light-weight and only requires a small amount of additional memory space. We sample a small percentage of buckets in the hash table, augmenting them with larger sampling buckets, to emulate a larger hash table. We treat the content in those larger sampling buckets as the ground truth. At the end of each sampling window, we compare the content of the sampled buckets in the hash table with the content of the sampling buckets to calculate the false positive and false negative rate.

We report this as the estimated accuracy. In Section III, we show that this sampling mechanism tracks the real accuracy closely with only sampling 5% of the total buckets.

D. Dynamic Sketch: Closed-loop accuracy control

With the sampling method and the Door keeping algorithm, Dynamic sketch can dynamically monitor and adjust itself to achieve certain accuracy targets. Specifically, users can specify their precision and recall rate targets to Dynamic sketch, and Dynamic sketch aims to reach the targets via adjusting the door keeping threshold dynamically based on the feedback from the sampling mechanism.

As discussed, the door keeping threshold trades off between the false positives and false negatives (precision vs. recall rate) which means higher precision rate (PR) leads to lower recall rate (RR) and vice versa. Thus, we design the algorithm to achieve PR target meanwhile to maintain RR as high as possible. Both PR and RR are reported to users so that they can adjust the target if necessary in the following profiling windows. During the self-adjusting phase, the algorithm starts with a low door keeping threshold. This is to start with RR as high as possible. It might be necessary to trade RR for PR later on. Every 100k packets (or specified by user), the sketch reads the sampling sets and estimates PR and RR. If PR drops below the specified target value and RR is still higher than the target, the sketch performs the following two steps: 1) increase door keeping threshold for next sampling window, and 2) reset certain percentage of counters in the CMS. The first action is to improve PR for the next sampling window as we described in the Door keeper section. However, many mice flows may have already sneaked into the CMS due to the initial low threshold. Thus, the second action of resetting certain amount of counters in the CMS effectively removes those flows. We choose to reset only certain percentage of the total counters (e.g. 20%) due to the following considerations: 1) to trade off between the false positive and false negative with finer granularity, and 2) to reduce the overhead of the counter resetting. To further reduce the cost of resetting counters, a more passive resetting process may be applied. For example, in the next sampling window, when a new packet comes to the CMS, we reset the corresponding counter once and mark it as reset done. The process can be repeated until certain percentage of counters are marked.

If the monitored RR is lower than the target, the algorithm lowers the current door keeping threshold. If neither RR nor PR target can be reached after the adjusting phase finishes, a warning is sent to the user. User can then choose to either enlarge the data structure or lower the target in the next phase.

E. Heavy Hitter Detection Metric

After describing the design of Dynamic sketch, we now introduce a new metric to evaluate heavy hitter detection algorithm. With the traditional binary classification used by many previous studies, a flow would be classified as either heavy or light with a crisp boundary (e.g. 500 packets), without considering the distance of this flow's packet count to the boundary. We believe the classification of flows close to the boundary should carry different weight comparing to the flows far off from the boundary during accuracy evaluation. For example, if we wrongly classified a flow of 490 packets as a heavy flow, it should be much better than mis-classifying a flow of only 100 packets. Thus, we propose an improved false positive and false negative definition to take consideration of such factor.

We use a linear weight algorithm to weigh false predictions differently depending on their deviation from the boundary. As indicated by the following equation, we treat a mis-classified flow as a fraction of one false-positive or false-negative if its packet count x is within a range of the boundary.

$$f(x) = \begin{cases} 1 & : (|x - boundary|) > range \\ \frac{|x-boundary|}{range} & : (|x - boundary|) \leq range \end{cases}$$

For example, a flow with 490 packets that is mis-classified as a heavy flow (i.e. above 500 packets) is only counted as 0.2 false positives with a range of 10% around the boundary. We believe the new metric could more practically reflect the quality of the heavy hitter detector. We use such improved metric throughout the evaluation section. However, it's worth noting that even with the original simple cut-off definitions, the conclusion of the evaluation section still holds.

TABLE I
TRAFFIC TRACES USED FOR EVALUATIONS

trace	flow info.
CAIDA	10 traces, 2.5m packets each, 110k flows, 850 heavy flows
Facebook	9.1m packets, 16k flows, 739 heavy flows
UCLA	10m packets, 30k flows, 740 heavy flows

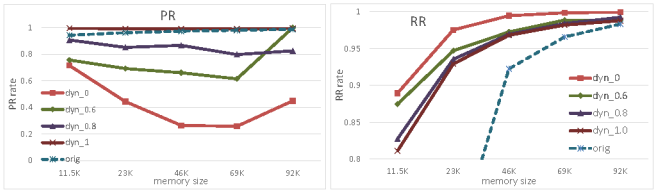
III. PERFORMANCE EVALUATION

A. Platform and Data Set Configuration

Platform: We use Intel® Xeon® Platinum 8160 CPU (code name Skylake SP) on a two-socket platform with each CPU running at 2.1GHz. Each CPU chip has 24 cores and there are in total of 48 cores. We run the software implementation on a single core for all the evaluations. We have in total of 88GB DDR4-2666 DRAM on board.

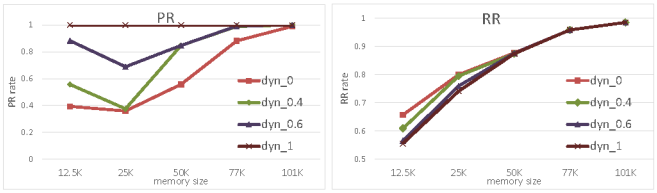
Dataset: For comparison, we use the data traces provided by Elastic sketch [14], which consist of multiple datasets based on CAIDA traffic traces. We also use two longer traces, from Facebook [15] and UCLA [16]. They have very distinct characteristics than the CAIDA traces w.r.t. flow and packet count, which allows us to evaluate our design with various traffic scenarios. We use the Source IP as the flow ID in all the evaluations, 0.02% packet count as the heavy flow threshold for CAIDA and Facebook traces, and 0.002% for UCLA traces. The number of heavy flows is roughly under 1000 for all traces, thus we can use similar memory size for comparison. More details on the dataset are shown in Table I.

Software Parameters: For all the tests, 5% of the hash table size of memory is allocated to the count-min sketch,



(a) (b)

Fig. 3. Precision rate (a) and recall rate (b) comparisons among various configurations.



(a) (b)

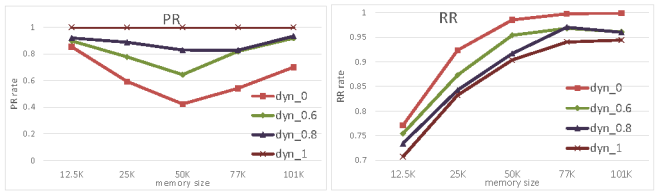
Fig. 4. Precision rate (a) and recall rate (b) comparisons among various configurations with Facebook trace.

comparing to 2× in the Elastic sketch paper [12]. For comparison with other algorithms, we fixed the total memory size for fairness. We use the same 1-row count-min sketch structure as Elastic sketch, and the hash table is a 8-entry bucketized hash table structure. The baseline door-keeper threshold is the average count of the counters of the bucket, and it is increased or decreased depending on the control loop.

B. Accuracy of Heavy Hitter Detection

We run all CAIDA traces and plot the average recall and precision rates for comparison with various parameters in Figure 3. The memory size of the x-axis is the total memory size including the sampling buckets for Dynamic sketch. In these tests, we apply different preset target PRs, and allow the algorithm to freely trade off RR and maintain targeted PR. We sample 5% of total buckets with each sampling bucket consisting of 16 entries. The hash table itself has 8 entries per bucket. Intuitively larger sampling buckets can be used for higher accuracy estimation, but we found 16-entry is sufficient to effectively predict PR for closed-loop control. As a result, only 10% additional memory is allocated for sampling purpose.

Figure 3(a) shows that PR are kept above the target for all the tests. As expected, higher PR usually trades off for lower RR. Note that original Elastic sketch (i.e. the "orig" curve in the figure) without Door keeper and uses a large CMS needs significant more memory and cache resource (which may negatively impact the performance of co-running workloads), produces much lower RR and provides no self-adjusting capability.



(a) (b)

Fig. 5. Precision rate (a) and recall rate (b) comparisons among various configurations with UCLA trace.

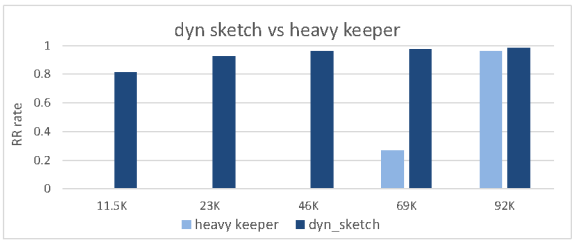


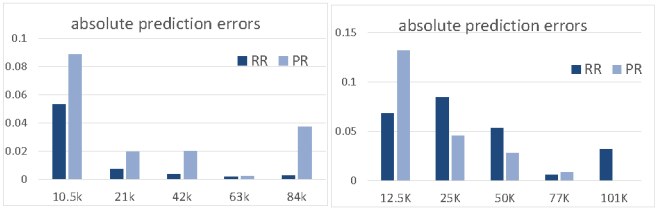
Fig. 6. Recall rate comparison to heavy keeper

Facebook and UCLA traces Figure 4 and Figure 5 show the evaluation results of the Facebook and UCLA traces. Due to the much higher packet count and more uniform distribution than the CAIDA traces, we sample 5% buckets with 32 entries per bucket in these tests. Both sets of results show similar effectiveness comparing to that of the CAIDA traces. For the Facebook trace, we observe that RR does not change notably when PR target is adjusted, meaning that RR is more tightly bounded by the size of total memory rather than the CMS in this case. This is because the Facebook trace exhibits more uniform distribution than the other two traces.

Comparing to Heavy Keeper We also compare the accuracy with the state-of-the-art hash table based heavy hitter detection algorithm, Heavy keeper [17]. We allocate various memory sizes and configure Heavy keeper to identify top-900 heavy flows. A major issue of Heavy keeper lies in the extra memory required to maintain the heap for heavy flows. From the open source implementation [18], we found that 54 extra bytes are needed for each heavy flow, comparing to only 8 bytes in Dynamic sketch. Figure 6 shows that Heavy keeper cannot start with smaller memory sizes, while for larger memory sizes, Dynamic sketch achieves better accuracy. Figure 8 shows that the heap consumes significant higher overhead to maintain. It costs 2-3x more cycles for processing each packet for Heavy keeper. Other counter and heap based algorithms have even worse accuracy and performance [17]. It is also not straightforward on how to monitor the accuracy during runtime for heap based algorithms.

C. Set Sampling Evaluation

Figure 7 evaluates how accurate the sampling method tracks runtime PR and RR. As we can see, for most cases the absolute error is within 0.05 with the exceptions for very low memory size cases. When smaller memory size is allocated to the hash table, the absolute number of sampling buckets are also small which leads to lower accuracy due to inherent sampling variations. Another observation is that the Facebook trace yields lower accuracy than the CAIDA traces even if we give 32 entries per sampling buckets (comparing to 16-entry



(a) (b)

Fig. 7. Absolute error comparison: CAIDA trace estimation errors with 5% 16-entry sampling (a) and Facebook trace estimation errors with 5% 32-entry sampling (b) reported by the sampling technique.

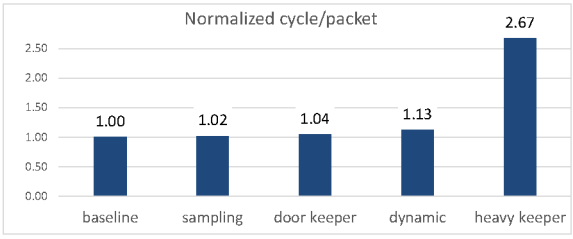


Fig. 8. Performance comparison

for CAIDA traces). This is again due to the more uniform distribution of the Facebook trace.

D. Processing Speed Evaluation

The performance comparison is based on the 5% sampling rate with 16-entry sampling buckets. Figure 8 shows that during steady state operations, the Sampling and Door keeper algorithms only incur a small overhead of 2-4%. Meanwhile, during active dynamic control phase (resetting counters and calculating RR and PR every 100k packets), the processing overhead is more pronounced, at 13%. In practise, when the profiling accuracy reached the target, the algorithm exits the dynamic control phase and operates at steady state. During the steady state operation, the sampling window is much larger which reduces overhead significantly.

IV. Related Works

Various algorithms have been proposed for heavy hitter detection. These algorithms in general fall into two categories. The first category is counter-based algorithms. Examples in this category include Frequent [19], [20], Lossy Counting [21] and Space Saving [9], summarized in [22]. With counter-based algorithms, candidate heavy flows are stored in a linear table of counters, which requires O(n) overhead for key lookups. Periodically, the table also needs to be swept to evict keys with smaller counters. More advanced algorithms could achieve O(1) for lookup and update, however they require much more memory [23]. A recent study [7] proposed a variant of counter-based algorithm which works better for hardware pipeline implementation, but not for software packet processing. The second major category is sketch-based algorithms [1], [10], [11], [24]–[26]. Traditionally sketch-based algorithm uses sketch with a sorted data structure such as min-heap to record the key of the heavy flows. Time consuming operations are required to maintain the heap for example in the Heavy keeper [17].

A most recent work on traffic profiling is Elastic sketch [12]. In this study, the authors propose to combine a hash table with a count-min sketch (CMS) [10] to accurately report heavy flows. Previous studies also use a hash table as the key storage for a CMS [27], [28]. The difference is that Elastic sketch puts the hash table in front of the CMS. Dynamic sketch is based on Elastic sketch. However, with the algorithms we proposed, Dynamic sketch is much more memory efficient than Elastic sketch and able to adjust itself for accuracy targets which the Elastic sketch cannot do.

V. Conclusion

In this paper, we propose Dynamic sketch which provides users profiling accuracy information at runtime, and adjusts accuracy toward user specified target via a dynamic closed-loop feedback control. None of existing algorithms have such capability. Comparing to traditional algorithms which give a rough mathematical error bound, Dynamic sketch provides more practical information in real networking deployment. Evaluation shows that Dynamic sketch successfully tracks user specified accuracy targets and reports PR and RR accurately with minimal performance overhead. Meanwhile, Dynamic sketch is much more memory efficient than the state-of-the-art heavy hitter algorithms including Elastic sketch and Heavy keeper. This is critical for networking services running on the same platform with many cloud applications, who are competing for hardware resources.

References

[1] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'13, 2013.

[2] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '11, 2011.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, 2014.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10, 2010.

[5] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding tcp incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, ser. WREN '09, 2009.

[6] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18, 2018.

[7] 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.

[8] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96, 1996.

[9] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proceedings of the 10th International Conference on Database Theory*, 2005.

[10] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, 2005.
[11] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, 2016.
[12] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018.
[13] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19, 2019.
[14] *Elastic sketch GitHub repo*, 2019 (accessed April 10, 2019), https://github.com/BlockLiu/ElasticSketchCode.
[15] *Facebook networking trace*, 2019 (accessed April 10, 2019), https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/.
[16] *UCLA networking trace*, 2019 (accessed April 10, 2019), https://lasr.cs.ucla.edu/ddos/traces/.
[17] J. Gong, T. Yang, H. Zhang, H. Li, S. Uhlig, S. Chen, L. Uden, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18, 2018.
[18] *Heavy keeper GitHub repo*, 2019 (accessed April 10, 2019), https://github.com/papergitkeeper/heavy-keeper-project.
[19] 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*, 2003.
[20] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *Proceedings of the 10th Annual European Symposium on Algorithms*, ser. ESA '02, 2002.
[21] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02, 2002.
[22] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, 2002.
[23] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," *Proceedings of he 35th Annual IEEE International Conference on Computer Communications*, 2016.
[24] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, 1996.
[25] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '04, 2004.
[26] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in streaming data," *ACM Transaction on Knowledge Discovery from Data*, 2008.
[27] O. Alipourfard, M. Moshref, and M. Yu, "Re-evaluating measurement algorithms in software," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIV, 2015.
[28] O. Alipourfard, M. Moshref, Y. Zhou, T. Yang, and M. Yu, "A comparison of performance and accuracy of measurement algorithms in software," in *Proceedings of the Symposium on SDN Research*, 2018.