OctoSketch: Enabling Real-Time, Continuous Network Monitoring over Multiple Cores

Yinda Zhang[†], Peiqing Chen[⋆], Zaoxing Liu[⋆]
[⋆]*University of Maryland,* [†]*University of Pennsylvania*

Abstract

Sketching algorithms (sketches) have emerged as a resource-efficient and accurate solution for software-based network monitoring. However, existing sketch-based monitoring makes sacrifices in online accuracy (query time accuracy) and performance (handling line rate traffic with low latency) when dealing with distributed traffic across multiple cores. In this work, we present OctoSketch, a software monitoring framework that can scale a wide spectrum of sketches to many cores with high online accuracy and performance. In contrast to previous systems that adopt straightforward sketch merges from individual cores to obtain the aggregated result, we devise a continuous, change-based mechanism that can generally be applied to sketches to perform the aggregation. This design ensures high online accuracy of the aggregated result at any query time and reduces computation costs to achieve high throughput. We apply OctoSketch to nine representative sketches on three software platforms (CPU, DPDK, and eBPF XDP). Our results demonstrate that OctoSketch achieves about 15.6× lower errors and up to 4.5× higher throughput than the state-of-the-art.

1 Introduction

While telemetry on dedicated switching hardware continues to be important, the deployment of monitoring capabilities in software platforms is increasing with the transition towards virtualized deployments and "white-box" capabilities [1–3]. The ability to monitor network traffic within software platforms has become a key enabler for many network management applications, including load balancing, anomaly detection, and performance diagnosis [4, 5]. Over the years, the volume of traffic processed by each server has seen a substantial increase (*e.g.*, 10G to 100G). As a result, efficient monitoring of traffic across multiple cores emerges as a pressing demand.

When monitoring traffic distributed across cores, downstream applications often entail high-fidelity *aggregated* measurement results. For instance, in-network caching systems [6, 7] require real-time measurement of the *hot* objects to determine what to cache. Similarly, a load balancer needs aggregated flow statistics (*e.g.*, heavy hitters, top-k flows) to decide where to assign flows (to cores or nodes) [7, 8]. In this scenario, sketch algorithms naturally emerge as a promising solution. This is because recent theoretical advances [9] have shown that many sketches, such as Count-Min [10], Count Sketch [11], and UnivMon [12], have intrinsic *mergeability*: Independent sketches monitoring different traffic partitions can be merged in a way such as *sum* and *max* to obtain aggregated results without losing accuracy guarantee. With this property, a natural design for sketch-based multicore monitoring is to run an independent sketch per core and periodically merge the entire sketches from all cores for aggregation.

While we have seen significant recent progress in sketch algorithms [12–17] and their implementations [18, 19], we argue that sketch-based multicore monitoring systems remain impractical. Many existing systems, such as Elastic Sketch [13], NitroSketch [14], and HeteroSketch [20], adopt the above "sketch-merge" design to scale their systems to distributed settings. However, these solutions raise two significant issues in practice. First, this approach does not retain accuracy at any given time – if the *aggregator* responsible for answering the query has not yet obtained the most recent sketches from cores, the aggregated result is stale, missing the measurement over the current traffic. A recent study [21] shows that an inaccurate and stale result can lead to a significant load imbalance in a load balancer. Second, if we consider an undesirable extreme to frequently merge sketches for fresh results, the packet performance will be significantly reduced, and it is difficult for each core to achieve high line rates (§2.2).

Ideally, we want a multicore monitoring system to meet three requirements. First, we need high accuracy whenever applications query a measurement result. This is essentially *online accuracy*, which enables real-time telemetry support for applications. Second, we expect *resource efficiency* to achieve line rates with low resource overhead since CPU and memory resources are shared among other services [22]. Finally, we need *generality*, covering a broad range of traffic metrics for various application requirements [12, 13, 23].

In this paper, we present **OctoSketch**, a software framework to perform high-performance and real-time monitoring on multiple CPU cores that meets the above three requirements. In OctoSketch, we argue for a continuous, change-based design to aggregate the sketch results among multiple cores. In contrast to periodically merging entire sketches,

OctoSketch keeps track of individual counter changes in the sketch for every packet and sends the counter difference-Δ (from the last update) when the change is sufficiently large (*e.g.*, over a threshold to meet the accuracy requirement). By this simple-yet-effective design, each worker still maintains a sketch but intends to send the Δ that could affect the overall accuracy of the aggregated result. In this way, communication between the worker and the aggregator becomes a "continuous" stream of worker-aggregator messages, which are tiny in space but carry the most critical and timely information. Compared to previous efforts [9, 12, 13, 20], OctoSketch eliminates potentially wasted counter updates when merging the sketches (*e.g.*, many counters do not change significantly since the last update) to improve online accuracy and optimize resource footprints.

With the continuous, change-based message passing scheme, OctoSketch is able to realize dynamic resource allocation policies toward different systems and user objectives. For example, within a CPU resource budget, OctoSketch can be configured to achieve the best (possible) online accuracy; or if the accuracy requirement is loose (*e.g.*, $\ll$ than best possible accuracy), OctoSketch can be configured to save CPU cycles instead of using all the aggregator CPU trying to reach an unnecessary accuracy. OctoSketch realizes these policies by (1) maintaining a *shared buffer* between workers and the aggregator, (2) adjusting worker message frequency via dynamic counter update thresholds, and (3) letting the aggregator learn the sending rate of the workers via the queue occupancy in the shared buffer, as detailed in §4.3.

To the best of our knowledge, OctoSketch is the first work to: (a) propose the integration of a continuous, change-based update mechanism with various sketches to enable the practical adoption of sketches in the multi-core scenario. Our goal is to improve the accuracy and performance of distributed sketches across cores to meet various measurement objectives; (b) analytically prove that OctoSketch can retain the same asymptotic error bounds as in the ideal case in which traffic is not distributed. That is, when querying OctoSketch, the aggregator can provide a result that represents the aggregation of traffic from all cores (by a bounded error) at any given time; and (c) provide practical end-to-end design and implementation of this idea on three popular software platforms. We further show that, compared to existing sketch-merge approaches, OctoSketch reduces message passing overhead by up to four orders of magnitude for the same accuracy.

We apply OctoSketch to a wide range of state-of-the-art sketches (*e.g.*, Count-Min Sketch [10], UnivMon [12], Elastic Sketch [13], and CocoSketch [16]) and demonstrate its performance on the CPU, Intel DPDK library [24], and eBPF XDP [25]. Our experiments show that OctoSketch achieves around 15.6× smaller errors than previous sketch-merge techniques at query time. Moreover, OctoSketch also reaches up to 4.5× higher throughput and up to 1.9× reduction in CPU utilization in DPDK. We apply OctoSketch to two common use cases: load balancer and key-value cache. Our results show that OctoSketch achieves a 3.15× lower imbalance rate and a 13% higher cache hit rate than the sketch-merge baseline. We have open-sourced OctoSketch and other baseline algorithms on https://github.com/Froot-NetSys/OctoSketch.

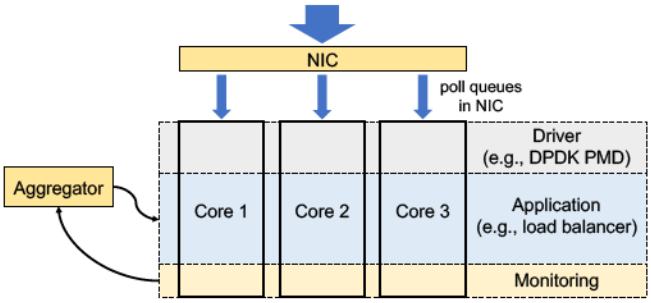


Figure 1: Multicore monitoring problem in a server

Ethics: This work does not raise ethical issues.

2 Background and Motivation

In this section, we first describe how distributed traffic over multiple cores brings new challenges to online monitoring. We then discuss existing efforts and their limitations in scaling monitoring capabilities to distributed settings.

2.1 Multicore monitoring problem

Since the latest NIC designs [26] for data centers have already reached 100Gbps/port, network applications supporting such a high speed require multiple CPU cores. For instance, a DPDK-enabled Open vSwitch [27] requires 6 to 10 CPU cores to reach 100Gbps. Hence, we call such a network application using multiple cores a *multicore application*.

Multicore applications often require timely and accurate traffic measurement results, such as heavy hitters, distinct flows, and entropy [4, 5]. Figure 1 shows a typical workflow for monitoring traffic distributed over multiple cores. The NIC will distribute the incoming packets into multiple Rx queues, and each CPU core will poll packets from one or more queues via driver (*e.g.*, DPDK polling mode driver [24]). The resource in each core is shared between drivers, applications, and monitoring programs.

In contrast to a non-distributed monitoring setting, this multicore workflow for a single server poses three significant challenges:

- ***Online accuracy:*** Many applications need to obtain accurate aggregated statistics in an online fashion. For example, cache-based load balancers rely on real-time large flow measurements to determine hot items to cache [6, 7]. We define the *online accuracy* according to the statistics computed over the traffic from the end of the last measurement to the current query time. Due to the distributed nature of the traffic, there can be a significant error or delay to query aggregated statistics at any given time for two reasons: (1) the current aggregated result is stale, and (2) frequent ag-

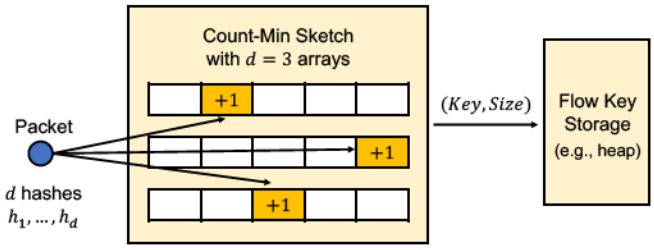


Figure 2: Example of the Count-Min sketch

gregation from all the cores is expensive to compute.

- ***Resource efficiency:*** The resources (*e.g.*, CPU and memory space) used by monitoring should keep up with line-rate requirements and leave ample room for applications.
- ***Generality in metrics of interest:*** Similar to other network monitoring systems [12,13], a multicore monitoring system shall measure multiple traffic metrics to support a range of applications (*e.g.*, DDoS detection [5], load balancing [4], and traffic engineering [28]).

2.2 Prior Solutions and Limitations

Background of Sketches. Sketches have emerged as a promising network monitoring solution due to their high resource efficiency and accuracy guarantees. At a high level, sketches are a kind of approximate data structure that can estimate various statistics online. For example, there are (1) Count-Min sketch [10] and Count sketch [11] for detecting heavy hitters (flows with large sizes), (2) Locher sketch [29] for detecting superspreaders (Source IPs that connect to many Destination IPs), (3) CocoSketch [16] for querying flow size on multiple keys, and (4) UnivMon [12] and Elastic sketch [13] for supporting a range of these tasks instead of a specialized sketch per task. As shown in prior work [12–14, 30–32], these sketches can often provide better accuracy-memory tradeoffs than traditional sampling-based techniques [33, 34].

To illustrate the insertion process of sketches, we use the Count-Min sketch for heavy-hitter detection as an example. As shown in Figure 2, the Count-Min sketch typically consists of multiple arrays of counters. For the insertion of each packet, it computes a set of independent hash values based on the flow key (*e.g.*, Source IP) of the packet and updates the corresponding counter in each array. Moreover, Count-Min Sketch needs additional data structures (*e.g.*, heap) to store flow keys whose estimated sizes are large. We further discuss the background of sketches and their applications in §A.

Key-based partition. One way to apply sketches in the multicore scenario is to divide different keys to different cores based on hashing. For each core, we can maintain a sketch for tracking the keys hashed to it and the keys in different sketches never overlap. However, strictly pinning a key to a core can lead to degraded packet performance. For example, when the skewness of the network traffic varies, key-based partition often leads to high load imbalance among cores.

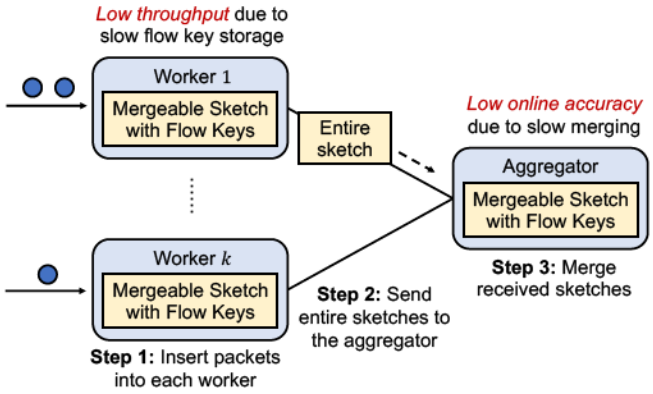


Figure 3: Merging sketches from multiple cores to an aggregator (sketch-merge)

Splitting large flows into multiple cores will help balance the load, as shown in existing load-balancing algorithms [35]. Moreover, depending on the definition of the key, it is sometimes infeasible to pin a key to a core. For instance, a NIC may distribute packets based on the 5-tuple while the application may want to measure the keys based on user-level information (*e.g.*, the key-value pair). In this case, it is hard to guarantee all packets associated with a key will reach the same core. Moreover, some multiple applications require measuring results based on multiple key definitions, and we cannot guarantee key distribution across different keys, as shown in [16]. As a result, OctoSketch does not make any assumptions about the way of distributing keys to accommodate various network applications.

Merging sketches from multiple cores (sketch-merge). Another natural solution is to merge multiple sketches from different cores into one for query periodically, where the merged sketch preserves the same accuracy guarantee [9]. For instance, Sketch 1 measuring flow set A can be merged with Sketch 2 measuring flow set B (*e.g.*, *sum* or *max* of the two counter arrays) to obtain statistics about a combined flow set $A \cup B$, as long as Sketches 1 and 2 share the same configuration. Most existing solutions leverage this mergeability to scale to multi-site (*e.g.*, multiple cores or servers) in diverse domains, including UnivMon [12], HeteroSketch [20], and FetchSGD [36].

However, periodically merging entire sketches from multiple cores brings large penalties on online accuracy and overall throughput. We use Figure 3 as an example to illustrate the workflow and its issues: Given k *workers*, each worker maintains one sketch to handle its own traffic. If one worker has received enough packets or a certain time interval is reached, it will send its sketch with heavy flow keys to the *aggregator* and recreate a new one. Once the aggregator receives a new sketch from a worker, the aggregator merges the sketch to its own (aggregated) sketch, and reports aggregated results such as heavy hitters and distinct counts. Unfortunately, the merging process can bring significant overheads to both the

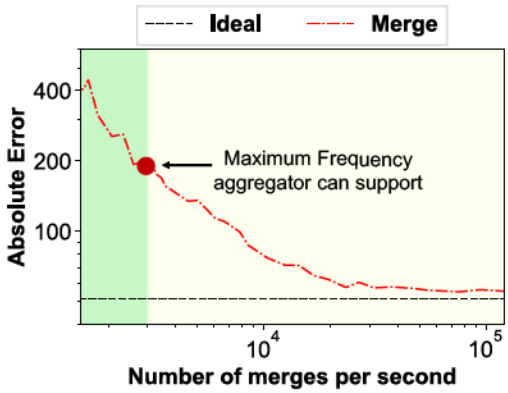


Figure 4: Frequency of sketch-merge vs. Error

aggregator and the workers as follows:

- **Bottleneck in the aggregator:** In practice, the aggregator is stuck in a fundamental dilemma: To achieve high online accuracy, the aggregator has to frequently merge multiple sketches (*e.g.*, merging at every millisecond), which can be prohibitively expensive to compute; otherwise, while infrequent and fixed-time merges save resources, the queried result at the aggregator may be stale by up to a fixed time window. To understand this problem, we consider a hypothetical non-merging design where all packets are processed by a single core with unlimited CPU cycles and cache, and use its accuracy as a reference of the ideal accuracy. As depicted in Figure 4, the aggregator needs to merge around 3×10^4 sketches per second to achieve comparable accuracy to the ideal case under 150 million packets per second throughput. However, a 2.35GHz CPU core can only support up to 3×10^3 merges per second. [1]
- **Bottleneck in the worker:** Each worker core keeps an entire sketch instance, including its counter structure and flow key storage. Prior efforts (*e.g.*, [13, 14]) have demonstrated that flow key storage is one of the performance bottlenecks in sketches. Maintaining such flow key storage per core is computation-heavy considering the high-volume traffic.

3 System Overview

We now describe the high-level design of OctoSketch and highlight the key ideas to achieve the multicore requirements.

3.1 OctoSketch Workflow

As illustrated in Figure 5, OctoSketch has two main components: multiple monitoring *workers* for ingesting the traffic

[1] The setting is the same as in §6 and §7. We use Count-Min sketch and CAIDA dataset [37], and show the absolute errors of heavy hitter detection. In this experiment, we have 16 workers and 1 aggregator. The frequency of merging is set at each worker. Since the aggregator is receiving and aggregating sketches from all workers, the aggregator is the bottleneck and can only accommodate 3×10^3 merges per second, while each worker can still send more merges. When increasing the merge frequency beyond the maximum rate the aggregator can support, we use a buffer to store sketches and aggregate the results later for evaluating the errors.

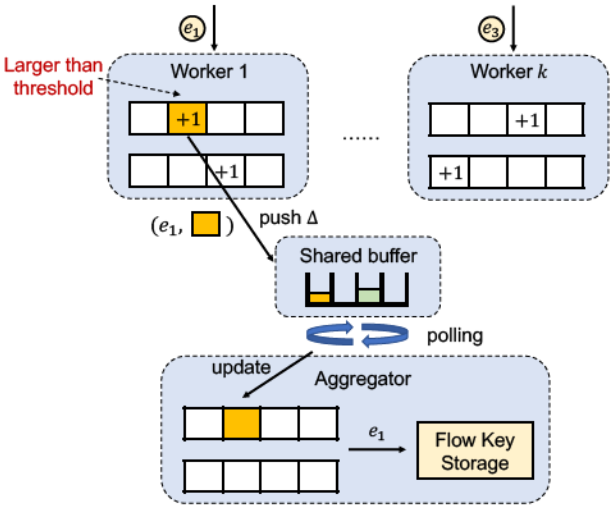


Figure 5: Workflow of OctoSketch

Query Type	Sketch	Acc.	Thp.
Flow Size	Count-Min [10]	9.32×	3.85×
	Count Sketch [11]	9.04×	3.22×
Cardinality	LogLog [38]	54.93×	1.29×
	HyperLogLog [39]	38.97×	1.29×
Super-Spreader	Locher Sketch [29]	4.06×	4.51×
Quantile	DDSketch [40]	4.29×	0.92×
Multi-Key	CocoSketch [16]	37.25×	1.01×
General	UnivMon [12]	13.55×	2.63×
	ElasticSketch [13]	14.03×	0.93×

Table 1: Applicability of the OctoSketch on latest sketches

and one *aggregator* for answering queries and overall control. By default, we allocate each worker and aggregator a separate core/thread. Thanks to the lightweight design of OctoSketch, one aggregator is sufficient to achieve online accuracy (§7). There is also a *shared buffer* for workers and aggregators to prevent inter-thread message losses from bursts.

Data ingestion in the workers: Each worker is a monitoring program running in a CPU core that is responsible for processing its own portion of the traffic from a NIC Rx queue and maintaining relevant sketch data structures (*e.g.*, hash-indexed counter arrays) for later aggregation. In OctoSketch, the goal of the workers is to provide timely and accurate counter updates to the aggregator to answer queries, while minimizing the CPU and memory footprints. When the worker sends an update to the aggregator, it will insert the update into a shared buffer. Unlike prior work that sends the entire sketch as an update, OctoSketch adapts to actual workloads and sends only lightweight, change-based counter updates (Δs). We describe this distributed update mechanism in detail in §4.1.

Query estimation in the aggregator: The aggregator fetches updates from all workers via the shared buffer and updates its own aggregated data structures to compute the statistics of interest. We describe the aggregation procedure in detail

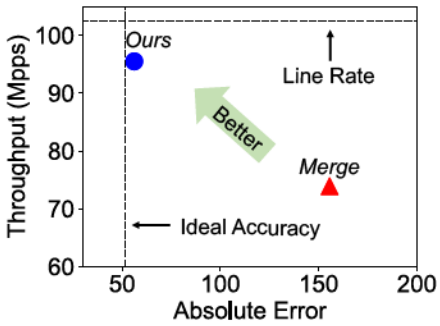


Figure 6: Online accuracy vs. throughput

in §4.2. The query results will then be shared with other software processing libraries or online applications for their management purposes.

Supported sketches (Table 1): OctoSketch is a general framework that support all *linearly mergeable* sketches and thus can answer a broad spectrum of sketch-based queries. To demonstrate the performance of OctoSketch, we apply OctoSketch to 9 popular sketches as shown in Table 1. We note that OctoSketch cannot make sketches that are fundamentally not mergeable (*e.g.*, RAP [41] for detecting heavy hitters) work in the multicore scenario.

3.2 Key Ideas

To realize the OctoSketch workflow, we have three key ideas. By leveraging these ideas, OctoSketch can achieve near-ideal online accuracy and line-rate throughput, outperforming the baseline solution as summarized in Figure 6.

Idea 1: ***Adopting a continuous, change-based mechanism where each worker only sends "sufficiently changed" counters to the aggregator.***

This idea achieves significantly better online accuracy compared to merging sketches (*e.g.*, all the sketch counters and the corresponding flow key storage). The fundamental reason behind the inefficiency of sketch-merge is that if we look at the value of each counter, not all counters are equally important to query accuracy. Due to the skewness of the network workload, the counter values in the sketch are also heavy-tailed. As shown in Figure 7, if we use the Count-Min sketch to find heavy hitters, more than 80% of the counter values do not change in the aggregator, while there are also counters whose changes are larger than 10^3. In other words, the aggregator and the workers waste most computations to merge counters which are not important and delay the updates from important counters.

Inspired by prior change-based solutions [42–45], OctoSketch essentially "amortizes" a large sketch merging operation into a series of individual counter change notifications to achieve online accuracy. Each notification only contains the counters that changed above a threshold. In this way, we can save the computation of merging small counters and use the resources for merging large counters more frequently. In other words, we only send counters with the most critical *informational value* for online accuracy. Our analysis in §5 shows that estimation accuracy is bounded by a certain range from ideal accuracy.

Compared to prior change-based designs with hash tables or other deterministic structures, OctoSketch aims to enable the use of diverse sketches for efficient multicore monitoring by addressing several additional challenges. First, sketches introduce extra errors on top of the errors caused by the change-based updates. The change thresholds must be determined based on the unified error bounds. Second, there are several key parameters regarding the sketch data structures and communication between different parties. We need to dynamically fine-tune the parameters to allocate the resources among workers and aggregators for various objectives. Third, for different sketches, we need to analyze and reconstruct the data structures for better performance.

Idea 2: ***Adaptive resource allocation between workers and aggregator under various objectives.***

With Idea 1, OctoSketch provides an improved tradeoff between communication and online accuracy. As a multicore system, a key question is how OctoSketch can allocate resources among workers and aggregator to meet various system objectives. For instance, an example objective is to achieve the best possible online accuracy with a fixed CPU budget (e.g., 50%). Users can also specify an accuracy requirement (e.g., 95%) while the objective is to minimize total CPU utilization. To meet these objectives, we adaptively allocate resources using dynamic counter change thresholds based on traffic rates. In particular, we set a universal threshold for all workers to adjust the resources: (1) When the packet arrival rate on a worker is high, the counter changes are significant, and the worker needs to send updates via the work-aggregator channel. (2) When the packet arrival rate is low, the worker does not send updates until sufficient counter changes are made, saving worker/aggregator resources for other worker-aggregator channels.

Therefore, to achieve the best possible online accuracy, the aggregator just needs to poll counter updates from workers (via concurrent queues) as much as possible within a CPU limit. When the queue length is small, the aggregator will decrease the threshold to receive more frequent counter updates. If we only want to fulfill an accuracy requirement that is lower than the best possible accuracy, we can derive a fixed threshold to meet the error bound based on the analysis in §5.

Idea 3: ***Reconstructing workers and aggregator to remove redundant data structures.***

To scale to multiple cores, prior solutions need to maintain flow key storage in each worker and aggregator, which is compute-heavy (as shown in §2.2). We observe that Idea 1 and 2 enable the opportunity to reconstruct sketches and remove flow key storage in the workers. Specifically, since

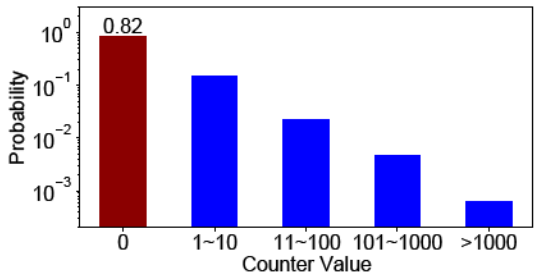


Figure 7: Counter value distribution when sketch-merge

Symbol	Description
k	the number of workers
τ	the threshold shared by all workers
e	the flow key of a packet
$f(e)$	the real size of flow e
$\widehat{f}(e)$	the estimated size of flow e
d	the number of arrays in the sketch
l	the number of counters in each array
$h_i(.)$	the hash function for the i^{th} array
$C_i[j]$	the j^{th} counter in the i^{th} array
q_j	the shared queue between worker j and aggregator

Table 2: Symbols and notations

the traffic statistics are calculated based on aggregated flows, only the aggregator needs to keep a flow key storage and the workers can remove their flow key storage. Therefore, each worker eliminates the performance bottleneck of maintaining complex data structures (*e.g.*, heap) and only sends potential heavy flow keys to the aggregator. Due to the efficient continuous, change-based mechanism, the counter and key updates will not create additional burdens on the aggregator.

4 Detailed Design

In this section, we first use the Count-Min sketch [10] as an example to describe how OctoSketch works in both workers (§4.1) and the aggregator (§4.2). Then, we extend the design of OctoSketch to other sketches. We summarize the frequently used symbols and notations in Table 2.

4.1 Worker

Data structure: In each worker, OctoSketch maintains a Count-Min sketch with d arrays of l counters without flow key storage. Since OctoSketch guarantees that all counters in every worker are always smaller than a given threshold τ (*e.g.*, 128), we only need $\lceil \log \tau \rceil$ bits (*e.g.*, 8) for each counter instead of using 32-bit counters. Thus, we can effectively save 4× memory without losing accuracy. In our experiments, we find that setting τ to 128 is often sufficient to handle the highest achievable throughput ($\approx$ 1400Mpps) for 16 workers.

Sketch insertion: Algorithm 1 describes how to insert a packet into each worker. The original operations of the Count-Min sketch are shown in black color, while the newly added operations by OctoSketch are shown in green color. In Line 3-5, when the Count-Min sketch updates a counter in each of its arrays, the worker will check whether the counter is larger than the threshold τ. If the counter value is above τ in a packet insertion, the worker generates a message containing a vector of <the flow key of the packet, the row and column indices of the counter, and the counter value> and pushes it into a concurrent queue between worker and aggregator. Each worker has a corresponding queue to avoid competition among workers, and the counter will be cleared out.

Algorithm 1: Sketch insertions on worker j.

```
Input: A packet with flow key e
1 foreach i (1 ⩽ i ⩽ d) do
2     C_i[h_i(e)] ← C_i[h_i(e)] + 1;
3     if C_i[h_i(e)] ⩾ τ then
4         q_j.enqueue(e, i, h_i(e), C_i[h_i(e)]);
5         C_i[h_i(e)] ← 0;
6 end
```

Figure 8 illustrates an example of the insertion step. To insert packet e_1, the worker first updates a counter in each array. Because the accessed counter in the second array is larger than the threshold 10, OctoSketch pushes the tuple <the flow key (e_1), the 2D counter index (the 2^{nd} array, the 4^{th} counter), counter value (10)> into the shared buffer and sets the counter to 0. While the aggregator can get the counter index via hashing the flow key, we choose to send the index directly to reduce the hash computation cost for the aggregator.

4.2 Aggregator

Data structure: The aggregator needs to maintain a $d \times l$ Count-Min sketch that is same-sized as other sketches in the workers, and a min-heap to record heavy flow keys.

Algorithm 2: Sketch insertions on the aggregator.

```
1 foreach i (1 ⩽ i ⩽ k) do
2     (e, j, p, V) ← q_i.dequeue();
3     C_j[p] ← C_j[p] + V;
4     if C_j[p] > heap.min then
5         f̂(e) ← min_{k∈[d]} C_k[h_k(e)];
6         heap.insert(e, f̂(e));
7 end
```

Insertion to the aggregated sketch: As shown in Algorithm 2, the aggregator keeps polling the items (the 4-tuple) from the k shared (concurrent) queues. For each item from the queue, the aggregator first updates the corresponding counter based on the index and value, and then updates the flow key heap. If the new counter is larger than the minimum value recorded in the heap, the aggregator will obtain the estimated

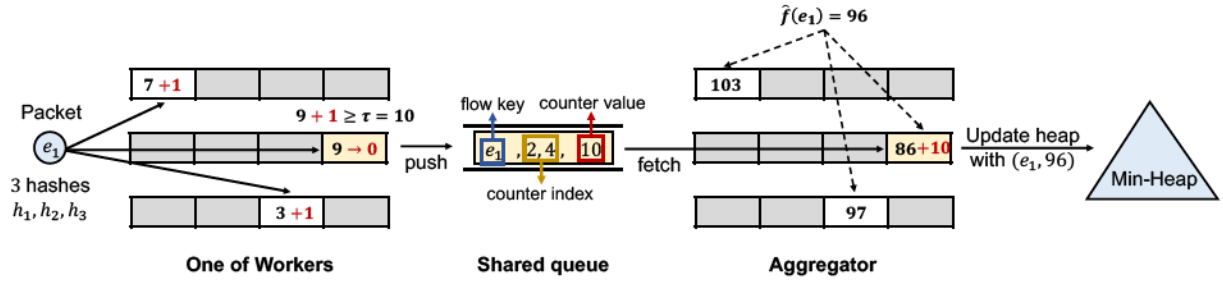


Figure 8: The insertion step in OctoSketch

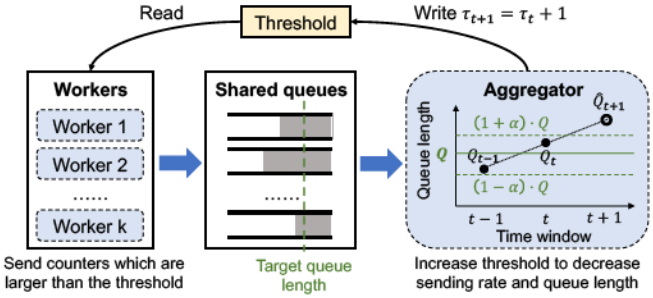


Figure 9: Adaptive thresholds

flow size e based on the query method of the Count-Min sketch (minimum counter value in the d accessed counters) and insert e into the heap. For example, as shown in Figure 8, if a tuple $< e_1, 2, 4, 10 >$ is in the shared queue, the aggregator will increase the 4^{th} counter in the 2^{nd} array by 10. Note if the counter after updating is larger than the minimum value record in the heap, the aggregator will update the heap with the tuple $(e_1, 96)$, where 96 is the estimated size of e_1.

4.3 Resource Allocation

We conceptually consolidate k workers as a single worker and k shared queues as a single queue to address the resource allocation problem. All k workers share the same threshold τ. The threshold is an atomic value. Ideally, the aggregator should adjust the threshold τ to match its receive rate with the total sending rate of k workers. Different from a distributed system where servers are interconnected by the network, the aggregator in a single server can quickly access the variables of other cores and obtain a global view to adjust the threshold.

The aggregator periodically modifies the threshold (*e.g.*, every 100μs). To ensure that the total queue length is small, the aggregator should set a target queue length Q (*e.g.*, 10). [2] For each time window t, the aggregator measures the total queue length Q_t as a reference mark for adjusting the threshold. We assume the sending rate is stable in that short period, so we can predict the queue length for the next time window based on the previous one. Specifically, as shown in Equation 1, according to Q_{t-1} and Q_t, we calculate the expected queue length $\widehat{Q}_{t+1}$ for the next time window $t+1$ corresponding to the threshold τ.

$$\widehat{Q}_{t+1} = Q_t + (Q_t - Q_{t-1}) \tag{1}$$

As shown in Equation 2, if the expected queue length $\widehat{Q}_{t+1}$ is too small, the aggregator will decrease the threshold to increase the send rate and the queue length; if $\widehat{Q}_{t+1}$ is too large, the aggregator will increase the threshold to decrease the queue length.

$$\tau_{t+1} = \begin{cases} \tau_t - 1, & \widehat{Q}_{t+1} < (1-\alpha)\cdot Q \\ \tau_t + 1, & \widehat{Q}_{t+1} > (1+\alpha)\cdot Q \\ \tau_t, & \text{Otherwise} \end{cases} \tag{2}$$

Policies for resource allocation:

- *Best possible accuracy:* The policy we describe above can make OctoSketch quickly adapt to dynamic packet arrival rates. However, it will use up 100% of CPU resources in the aggregator to improve accuracy as much as possible, which may not be ideal. Alternatively, we can consider two additional policies to reduce the aggregator CPU utilization.
- *Optimizing CPU usage for an accuracy target:* As we will prove in §5, the error bound of OctoSketch depends on the threshold τ. Based on an accuracy target, we can calculate the corresponding threshold τ' based on Equation 3 and set it as the lower bound of the threshold. In particular, the threshold will not decrease if it is smaller than τ'. In this policy, we can free up some extra computation resources when there is not much traffic.
- *Setting CPU usage limit:* We can also add a scheduling policy that forces the aggregator to use at most p% (*e.g.*, 50%) of the CPU. We leverage the Linux program CPU limit [46] to monitor and control CPU usage within p%.

4.4 Extension to Other Sketches

With some additional considerations, we can apply OctoSketch to a broad spectrum of sketches. In this work, we consider 8 additional sketches and scale them into multiple cores with OctoSketch. We summarize the key point of applying OctoSketch to these sketches and defer the details of each sketch to Appendix C.

[2] The target queue length should be larger than 0. Otherwise, the aggregator tends to keep increasing the threshold to ensure that the sending rate is always lower than the receiving rate.

Supporting sketches for estimating Cardinality: The merging process of cardinality-related sketches (*e.g.*, HyperLogLog [39], Locher sketch [29]) is different from the Count-Min sketch. Given the received counter A and the corresponding counter B in the aggregator, instead of summing up the counters, these sketches should set $B = \text{Max}(A, B)$. To apply OctoSketch to these kinds of sketches, every time the counter is updated, the worker never zeros it out but only sends the counter whose value change is large enough to the aggregator. Take HyperLogLog as an example. If the counter before updating is C and is C' after updating, the worker will send the counter if $\left|2^C - 2^{C'}\right| \geqslant 2^\tau$. We use 2^C instead of C, because the estimation provided by the HyperLogLog is based on the 2^C, which is the same as that of LogLog and Locher sketch. In Appendix B, we prove that, after applying OctoSketch, the cardinality-related sketches can achieve the same accuracy as the ideal one if the cardinality is sufficiently large.

Handling counters with flow keys: Unlike Count-Min sketch where the sketch structure only contains counters, some complex sketches (*e.g.*, CocoSketch [16] and Elastic sketch [13]) has a flow key corresponding to every counter. For these sketches, OctoSketch will send both the key and the counter to the aggregator and set the counter to zero if the counter is large enough. For each <key, counter> pair, the aggregator inserts the key into the sketch using the same insertion logic as the original sketch.

Handling negative counter values: For sketches such as the Count Sketch [11] and UnivMon [12], each counter value can be positive or negative. For these sketches, when OctoSketch checks whether the counter is large enough, we use the absolute value instead.

Benefits of applying OctoSketch (Table 1): The actual benefits vary among sketches as shown in Table 1. The throughput benefit of OctoSketch often comes from the heavy key storage. Because some sketches (*e.g.*, ElasticSketch [13]) proposed their own heavy key storage to speed up, OctoSketch cannot further optimize their throughput. The throughput may decrease due to the additional overhead of the concurrent queue. However, such a gap is often small (< 10%), and OctoSketch can still improve their accuracy.

5 Analysis

In this section, we first show that, after applying OctoSketch, sketches can still achieve the same error bounds as that of the ideal accuracy after receiving enough packets. Then, we analyze the tradeoff between online accuracy and communication cost. We prove that OctoSketch requires up to four orders of magnitude fewer worker-aggregator messages than prior sketch-merge approach for the same accuracy.

5.1 Error Bound

We show the error bound of OctoSketch for the Count-Min sketch in this section. The detailed proofs and analysis of OctoSketch for Count sketch and HyperLogLog are deferred to Appendix B. These three sketches provide three typical kinds of accuracy guarantees which are also used by other sketches (*e.g.*, Elastic sketch, Locher sketch, and UnivMon). To analyze the worst-case guarantee, we assume that the threshold is always τ_{max} and denote $\tau \equiv \tau_{max}$.

Let $\widehat{f}(e)$ be the estimated flow size of flow e for OctoSketch, $f(e)$ be the real flow size of flow e, k' be the maximum number of workers that a flow may pass by, and $L_1 = \sum_e f(e)$ which is the total number of packets in the traffic.

Theorem 1. *For OctoSketch for Count-Min sketch, let* $d = \log_2 \delta^{-1}$ *and* $l = 2\varepsilon^{-1}$. *For any flow e and any traffics whose* $L_1 > \varepsilon^{-1} k' \tau$,

$$\Pr\left[\left|\widehat{f}(e) - f(e)\right| > \varepsilon L_1\right] < \delta \tag{3}$$

Proof sketch: Suppose that $\widehat{f}'(e)$ is the estimated size of the Count-Min sketch working in a single core. Note that $\left|\widehat{f}(e) - f(e)\right| \leqslant \left|\widehat{f}'(e) - f(e)\right| + \left|\widehat{f}(e) - \widehat{f}'(e)\right|$, where the first part $\left|\widehat{f}'(e) - f(e)\right|$ is the original error from the Count-Min, while the second part $\left|\widehat{f}(e) - \widehat{f}'(e)\right|$ is the additional error brought by OctoSketch. For the first part, we can reuse the result in the paper of Count-Min sketch [10]. The analysis of the additional error caused by OctoSketch is shown in Appendix B. We show that such an additional error is finally marginal compared with that of the Count-Min sketch.

Interpretation: This theorem shows that OctoSketch can still achieve the same asymptotic error bounds as perform sketches in a single location after receiving enough packets. The number of packets depends on the maximum number of workers a flow may pass by (k') and the threshold of each worker (τ). In our experiments, setting τ to 2^7 is enough to achieve around 1400Mpps for 16 workers. In that case, after receiving 1M packets, OctoSketch can guarantee the same asymptotic error bounds for $\varepsilon > 2^{-9} \approx 0.2\%$. Note that flows may not appear on all 16 workers. If each flow is distributed to most two workers (*i.e.*, $k' = 2$), after 1M packets, OctoSketch can guarantee the same error bounds of $\varepsilon > 2^{-12} \approx 0.024\%$. Moreover, using this theorem, we can also compute the required threshold τ based on the accuracy requirement ε, trace length L_1, and k'. Specifically, we can set the threshold as $\tau = \frac{\varepsilon L_1}{k'}$ (4).

5.2 Communication and Accuracy tradeoff

We want to guarantee that the accuracy of the aggregated result over multiple cores is close to the ideal accuracy. We first formally define this accuracy goal in the following.

Definition 1 (Accuracy Goal)**.** *Given a sketch* S_1 *which works in a single core and a sketch* S_2 *which is the aggregated result over multiple cores, suppose that* $\widehat{f}_i(e)$ *is the estimated flow size of e for sketch* S_i. *We want to ensure that, at any time,*

for any flow e, $\left|\widehat{f}_1(e) - \widehat{f}_2(e)\right| < \Delta$*, where* Δ *is a predefined parameter.*

Note that the cost to achieve the accuracy goal varies for different sketches. In this section, we use the Count-Min sketch [10] and Count sketch [11] as a case study to show the superiority of OctoSketch over sketch-merge. To calculate the cost, we consider the number of counters the aggregator should process to achieve the accuracy goal. Suppose that for the Count-Min/Count sketch in each worker and the aggregator, there are d arrays, each with l counters.

Theorem 2. *To achieve the accuracy goal, sketch-merge needs to send* $O\left(\Delta^{-1}k \cdot N \cdot dl\right)$ *counters, while OctoSketch needs to send* $O\left(\Delta^{-1}k \cdot N \cdot d\right)$ *counters.*

Interpretation: We can see that OctoSketch needs to send l times fewer counters to achieve the accuracy goal compared to sketch-merge. In prior work [12,13], there are often more than thousands of counters in each array, *i.e.*, $l > 10^3$. Therefore, OctoSketch can send much fewer counters to achieve the same accuracy guarantee. Note that the computation cost needed in the aggregator is proportional to the number of counters sent. It also validates our experimental results that sketch-merge needs too much computation to achieve the similar accuracy. In experiments, we set $d = 3$ and $l = 2^{16}$. If we set Δ to 100, for 16 workers ($k = 16$), OctoSketch needs to send at most $0.48N$ counters to achieve the accuracy goal, *i.e.*, OctoSketch only needs to send at most 1 counter to the aggregator per 2 packets. Meanwhile, sketch-merge needs around 3×10^4 counters per packet to achieve the same accuracy.

6 Implementation

We implement OctoSketch using C++ and use xxHash [47] as the hashing library. All experiments are run on CloudLab [48]. We have open-sourced the artifact on GitHub [49].

Shared buffer: We use the concurrent queue [50] to be the shared buffers between workers and the aggregator. The queue is lock-free and uses atomic operations to achieve high throughput. In addition, it can dynamically allocate memory according to the number of items in the queue.

CPU: We implement OctoSketch on a machine with an AMD EPYC 7452 32-Core Processor at 2.35GHz and 128 GB ECC memory. We pre-process the input traces and distribute them to different workers. Each worker only needs to read the flow keys in memory and insert them into the sketch. In that way, we can focus on measuring the performance of sketches without the impacts from other applications.

DPDK: We also integrate OctoSketch with DPDK (version 21.11) [24]. Each OctoSketch worker is integrated with the polling mode thread in DPDK. Our testbed has two servers that are the same as the CPU implementation. Each server is equipped with a Mellanox ConnectX-5 Ex 100G NIC [26]. One server generates high-speed TCP traffic using pktgen-dpdk, while another server runs DPDK to receive packets and process them using OctoSketch. We use multiple cores to receive packets. Each worker corresponds to one core and one Rx queue, and it should extract packets from the Rx queue and insert them into the sketch. Thus, we can measure the overhead of sketches compared with DPDK.

eBPF XDP: We also integrate OctoSketch with XDP [25] in Linux kernel 5.15.0 using *SKB* mode. Our testbed has two servers that are the same as the DPDK implementation. We load the OctoSketch worker into the kernel to process packets, and the aggregator works in the user space. We use the *bpf_ringbuf* provided in the XDP library to send data from kernel space to user space. Note that in our setting, we cannot implement sketch-merge as it needs to send too much data (the whole sketch and heavy flow keys) at a time.

7 Evaluation

Our experiments compare the OctoSketch to the baseline sketch-merge approach and demonstrate that:

- OctoSketch can always maintain high online accuracy at any query time.
- OctoSketch can achieve high throughputs on all tested platforms, and it scales linearly with the number of workers.
- OctoSketch can achieve high resource efficiency on a representative high-performance packet processing library (Intel DPDK).

7.1 Experimental Methodology

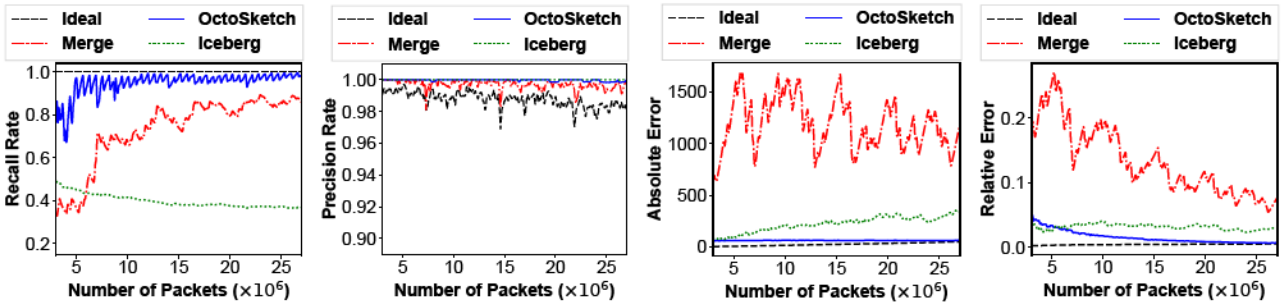
Traces: We mainly use two datasets:

- We use CAIDA traces [37] collected in the Equinix-Chicago monitor in 2018 in for our experiments by default. We use <Source IP, Destination IP> as the flow key.
- We generated 11 datasets that follows the Zipf [51] distribution with various skewness using Web Polygraph [52].

Metrics: We evaluate the following six performance metrics.

- *Absolute Error:* $\frac{1}{|Q|}\sum_{e\in Q}\left|f(e) - \widehat{f}(e)\right|$, where $f(e)$ is the real size, $\widehat{f}(e)$ is the estimated size, and Q is the query set.
- *Relative Error:* $\frac{1}{|Q|}\sum_{e\in Q}\frac{|f(e)-\widehat{f}(e)|}{f(e)}$, where $f(e)$ is the real value, $\widehat{f}(e)$ is the estimated value, and Q is the query set.
- *Recall Rate:* The ratio of the number of correctly reported flows to the number of correct flows.[3]
- *Miss Rate:* The ratio of the number of missing correct flows to the number of correct flows, which is $1 - RR$.
- *Precision Rate:* The ratio of the number of correctly reported flows to the number of reported flows.
- *F1 Score:* F1 Score is $2 \cdot (RR \cdot PR)/(RR + PR)$, where RR is the recall rate, and PR is the precision rate.
- *Throughput*: For CPU, we use million insertions per second (Mips). For DPDK and XDP, we use million packets per second (Mpps). The throughput numbers are the average value among 100 trials.

[3] Correct flows are the real heavy hitters in the traffic, and correctly reported flows are the real heavy hitters in the reported ones.



(a) Recall Rate (b) Precision Rate (c) Absolute Error (d) Relative Error

Figure 10: Accuracy of Count-Min Sketch at different query time.

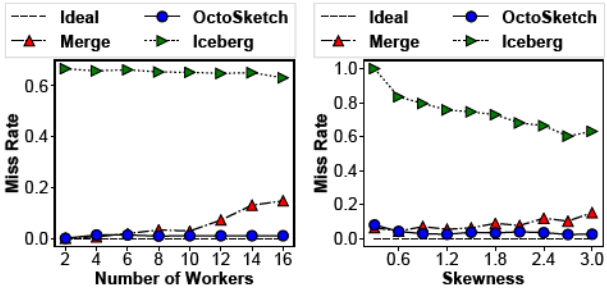
Sketches: We apply OctoSketch to all 9 sketches in Table 1. Due to space limitations, we show 4 representative sketches in this section: Count-Min Sketch [10] for detecting heavy hitters, Locher Sketch [29] for measuring super-spreaders, CocoSketch [16] for arbitrary partial key queries, and UnivMon [12] for generic flow monitoring.

Baselines: (1) *Ideal accuracy* is the accuracy of the sketch that works in a single core and measures the whole traffic. (2) *Sketch-merge* is merging the entire sketches as shown in §2.2. (3) *Iceberg* [53] uses the Count-Min sketch to find global heavy hitters. Instead of utilizing the sketch mergeability, Iceberg only uses the Count-Min sketch as a local heavy hitter estimator. Each worker in the Iceberg only sends the local heavy flow keys to the aggregator. The aggregator will ask every other worker for the statistics of these flow keys and aggregate the result. Once all worker answers, the aggregator can decide on the global heavy hitters. As shown in its paper [53], such a multi-round communication can reduce communication overhead.

Parameters: We refer to the individual paper to configure the sketch parameters [10, 12, 13, 16, 29]. For instance, in Count-Min sketch and Locher sketch, we use 3 arrays of 2^{16} counters per array. For CocoSketch, we use 2 arrays as suggested in their paper, and there are 2^{16} buckets per array. We describe the detailed parameters and configurations in Appendix D.1.

7.2 Online Accuracy

F1 scores at different query times (Figure 10a-10b&20): The F1 Score of OctoSketch is often close to the ideal accuracy and is 36.5% higher than sketch-merge. The sketch-merge needs to wait longer to get the sketches from all workers and converge to a relatively accurate result. Therefore, its F1 score is often lower than 0.5 before processing 5 million packets. Then we look at the recall (Figure 10a) and precision (Figure 10b) of the F1 score, respectively. The recall of OctoSketch is usually lower than the ideal accuracy, while the precision is often higher. It is because the counter in the aggregator is often smaller than that of the ideal accuracy since there is some information left in each worker. As a result, OctoSketch tends to underestimate compared to the ideal accuracy, resulting in a higher precision but a lower recall. The



(a) Accuracy vs. # of workers (b) Accuracy vs. Skewness

Figure 11: Accuracy with different parameters

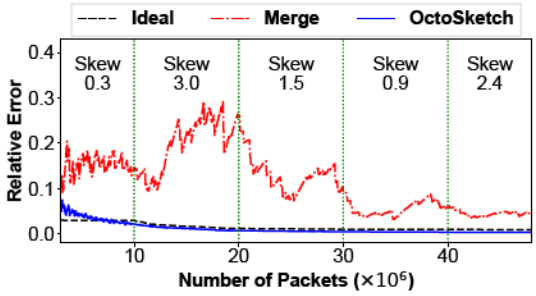


Figure 12: Dynamically changing workloads

recall rate of Iceberg is around 54% lower than OctoSketch. Its recall rate is low because the multi-round communication to all workers increases the delay (staleness) in the aggregator, which makes the aggregator miss many heavy flows.

Errors at different query times (Figure 10c-10d&21): OctoSketch can keep low errors in any query time. The gap between the relative error of OctoSketch and the ideal accuracy is often less than 0.05 and continues to decrease with more packets received, which matches our theoretical analysis in §5.1. The relative error of sketch-merge is 15.6× larger than OctoSketch, and its absolute error is 41.9× larger than OctoSketch. In addition, the accuracy of sketch-merge changes significantly over time: error drops abruptly once the aggregator merges a sketch, and keeps increasing until the aggregator merges another one. In contrast, OctoSketch's continuous mechanism maintains stable accuracy. The relative error of Iceberg is 2.6× larger than OctoSketch.

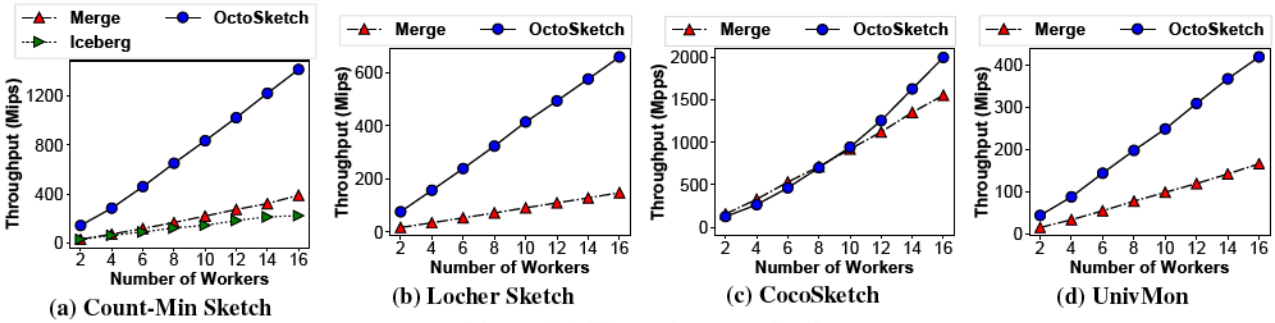


Figure 13: Throughput on CPU

Accuracy with different numbers of workers (Figure 11a): The miss rate of sketch-merge on the Count-Min sketch increases when scaling to more workers. It is because the time it takes to merge sketches in the aggregator has to increase as the number of workers increases. With 16 workers, the miss rate of sketch-merge for the Count-Min sketch is around 14.7× larger than that of OctoSketch, while the difference between the miss rate of OctoSketch and the ideal one is less than 2%. The Iceberg algorithm uses multi-round communication, which further increases the complexity and waiting time of the aggregator. As a result, its miss rate is around 12.0× larger than sketch-merge and 70.3× larger than OctoSketch.

Accuracy with different skewness (Figure 11b): OctoSketch for the Count-Min sketch achieves low miss rates on different skewnesses, while the error of sketch-merge increases with increasing skewness. It also verifies our statement shown in §3.2 that sketch-merge is not friendly to heavy-tailed workloads. The miss rate of sketch-merge is around 9.1× larger than OctoSketch, while the miss rate of Iceberg is around 21.0× larger.

Dynamically changing workloads (Figure 12): We simulate dynamic workloads by combing five datasets with different skewness ($0.3 \rightarrow 3.0 \rightarrow 1.5 \rightarrow 0.9 \rightarrow 2.4$). We find that OctoSketch for the Count-Min sketch is robust to dynamically changing workloads and maintains similar accuracy to the ideal. However, we note that OctoSketch does not address the inherent accuracy issues of the sketches. If the original sketch used is not capable of handling dynamically changing workloads, OctoSketch does not mitigate this limitation, but does not worsen the accuracy. Nevertheless, we observe that many sketches [10, 11, 13, 39] tend to exhibit a certain level of robustness to such dynamic workloads.

7.3 Throughput

CPU Throughput (Figure 13): OctoSketch achieves high CPU throughput and scales well with more workers. In the Count-Min sketch, with 16 workers, OctoSketch can achieve around 1400Mips. The multi-round communication in Iceberg increases the computation overhead of each worker and thus decreases the throughput. The throughput of Iceberg is around 1.4× lower than sketch-merge and 5.3× lower than OctoSketch. Compared to sketch-merge, OctoSketch can achieve 3.85×, 4.5×, and 2.63× higher throughput for Count-Min sketch, Locher sketch, and UnivMon, respectively. It is worth noting that the throughput of OctoSketch can sometimes increase faster than linear scaling with more workers. This is because the counter update threshold may also increase with more workers. Therefore, each worker costs less to send counters and the total throughput increases faster. For CocoSketch, due to the extra overhead of OctoSketch to send counters, the throughput sometimes is slightly lower than sketch merge, but the gap between them is less than 10%.

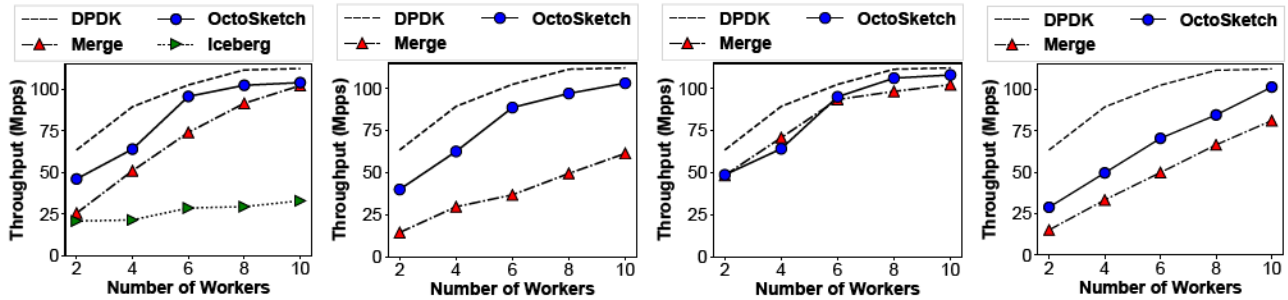
DPDK Throughput (Figure 14): OctoSketch achieves high throughput in DPDK. For the six tested sketches, OctoSketch can often reach ≈100Mpps with 10 workers. Sketch-merge often needs at least 2 more workers to achieve similar throughput as OctoSketch for Count-Min sketch, Locher sketch, and UnivMon. Specifically, OctoSketch accelerates the Locher sketch to achieve 97.0Mpps in 8 workers, while sketch-merge only achieves 61.3Mpps using 10 workers. One exception is CocoSketch: the throughput of OctoSketch is similar to that of the sketch-merge. This is because CocoSketch does not need additional heavy key storage, and thus OctoSketch does not further optimize throughput. Iceberg reaches only 32.9Mpps with 10 workers, because Iceberg needs multiround communication between the aggregator and all workers and additional workers can bring communication overheads.

eBPF XDP Throughput (Figure 15): As we cannot implement sketch-merge in XDP as shown in §6, we compare the throughput of OctoSketch to that of the XDP with sketches. The throughput of OctoSketch scales linearly with the number of workers. The throughput of OctoSketch with the Count-Min sketch is about 85% of the XDP, and the throughput of OctoSketch with the CocoSketch achieves 92% of the XDP.

7.4 CPU Utilization and Stability

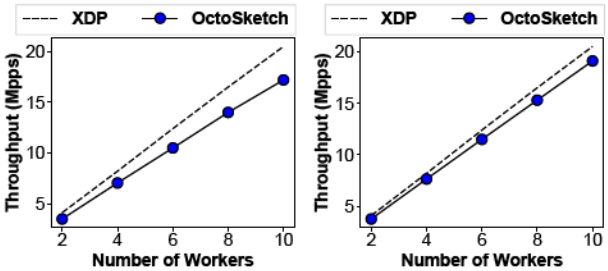
In this experiment, we show how OctoSketch can quickly adapt to varying packet arrival rates in DPDK.

CPU usage (Figure 16): The sketch CPU utilization is often reduced after applying OctoSketch. For the Count-Min sketch, the CPU usage of OctoSketch is around 1.75 and 3.34 times lower than sketch-merge and Iceberg, respectively. As shown in §7.3, OctoSketch incurs more CPU cycles when applied to CocoSketch due to the cost of sending counters. But the



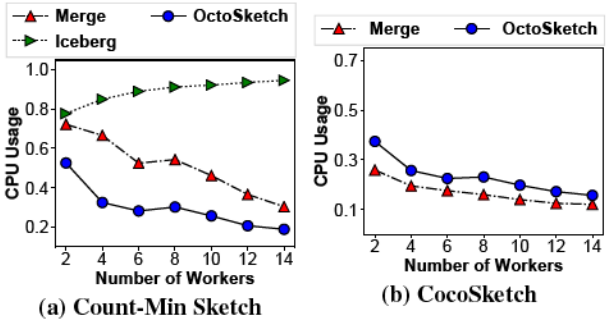
(a) Count-Min Sketch (b) Locher Sketch (c) CocoSketch (d) UnivMon

Figure 14: Throughput on DPDK



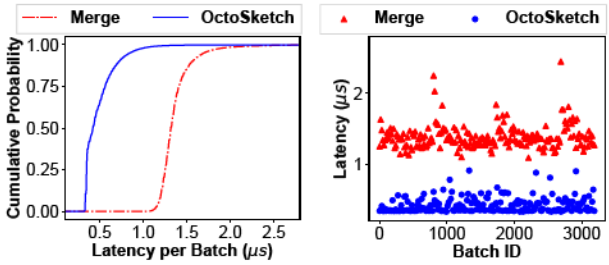
(a) Count-Min Sketch (b) CocoSketch

Figure 15: Throughput on XDP



(a) Count-Min Sketch (b) CocoSketch

Figure 16: CPU usage on DPDK



(a) CDF of Latency (b) Mini-batch Latency

Figure 17: Latency on DPDK

difference in CPU usage is usually less than 7%. Note that the CPU usage of sketches often decreases with more workers because the overhead of performing packet processing under DPDK will be increased when dealing with large volumes of traffic. The CPU usage of Iceberg increases with more workers due to the extra multi-round communication between workers and the aggregator.

Packet processing latency (Figure 17): We measure the latency needed for the Count-Min sketch in mini-batches (32 packets). We do not show per-packet latency as it is too fine-grained (≈ 10ns) for servers to measure. As shown in Figure 17a, for OctoSketch, more than 65% of the batches can be finished in 0.5μs, and over 98% of batches can be finished in 1μs. Moreover, the median latency of sketch-merge is around 3 times larger than that of the OctoSketch. Figure 17b shows that sketch-merge has larger jitters than OctoSketch. Every time sketch-merge creates a new sketch, there will be frequent sketch and flow key operations in the next few batches, leading to large latency. The jitters in OctoSketch are mainly caused by the *atomic* operations in the concurrent queue. Overall, the latencies and jitters in OctoSketch are significantly smaller than those of sketch-merge.

Adaptive thresholds (Figure 18): To illustrate the adaptiveness of OctoSketch, we use 10 workers to run Count-Min sketches with DPDK, set the lower bound of the thresholds to 8, and dynamically change the packet arrival rate. The upper figure shows that OctoSketch can quickly adapt to the varying arrival rates of packets. Moreover, we notice that the highest threshold used is still smaller than 20, *i.e.*, 8-bit is enough for each counter. The bottom figure shows the total queue length of the 10 shared queues. We find that the large queue length is often caused by the threshold decrease. Even if the packet arrival rate is stable, OctoSketch will still dynamically adjust the threshold. Sometimes, the threshold is too low, leading to a longer queue. However, OctoSketch can quickly adjust the threshold to decrease the queue length. We can see the queue length is relatively stable around the target length and is usually well below 400. Therefore, the size of the shared buffer needed by OctoSketch is often smaller than 10KB. Such a size is much smaller than that of the sketch-merge. For sketch-merge, we need at least 768KB (the size of a Count-Min sketch) more memory for merging.

7.5 Case Study: Load balancer

In this section, we show the benefit of OctoSketch for the Intel dynamic load balancer [35]. We also conduct a case study on the key-value cache and defer the results to Appendix §D.3.

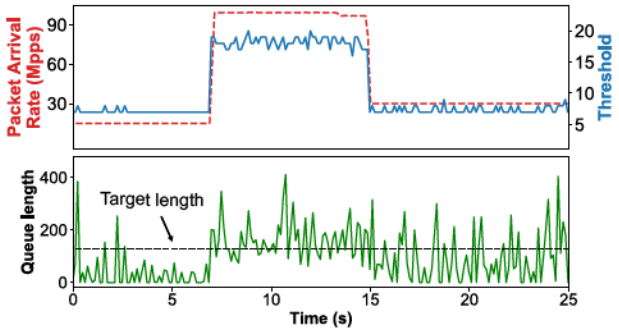
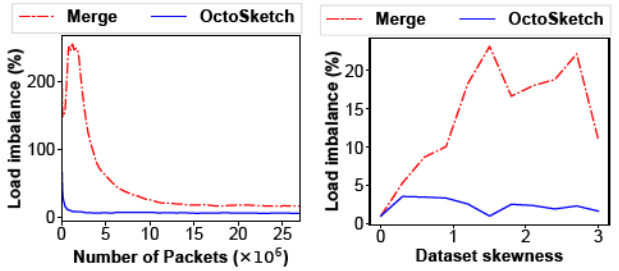


Figure 18: Adaptive thresholds



(a) Over time **(b) Over skewness**

Figure 19: Load balancer with different monitoring tools.

Initially, all traffic is distributed through RSS (based on hash). The aggregator aggregates the statistics over multiple cores and detects large flows. Once a flow is detected as a large flow, the aggregator sets a flow director rule for this flow in the network adaptor. Then, the large flow will be redirected and redistributed to multiple cores with the help of Intel DLB [54] which maintains the order of packets. In this experiment, we run simulations on the CPU due to the lack of Intel DLB hardware support in our testbed and measure load imbalance rate as the ratio of the number of packets processed by the most loaded worker relative to the least loaded worker [55]. The ideal imbalance rate should be 0.

Load balance over time (Figure 19a): OctoSketch helps Intel load balancer to achieve low imbalance rates. With fewer than 1M packets, the imbalance rate from OctoSketch is lower than 10%. However, sketch-merge needs more than 10M packets to reach a low imbalance rate and its imbalance rate can be large ($> 100\%$) before that. This is because sketch-merge spends more time detecting new large flows than OctoSketch, resulting in more packet losses. Moreover, after processing more than 20M packets, the imbalance rate of OctoSketch is still $3.15\times$ lower than that of sketch-merge.

Load balance over skewness (Figure 19b): We also show the imbalance rates on processing different packet traces with varying skewness. OctoSketch maintains low imbalance rates regardless of the skewness, while sketch-merge's imbalance rate is about $8.89\times$ larger when the skewness is larger than 1.2. Note that larger skewness does not always lead to larger imbalances. The imbalance rate also depends on how fast the algorithm can detect large flows and redistribute them.

8 Other Related work

Mergeability of sketches: As discussed in §2.2, most existing solutions (*e.g.*, CocoSketch [56], Beaucoup [15], HeteroSketch [20], and FetchSGD [36]) leverage the mergeability to apply sketches to distributed systems. These works often send the whole sketch to the aggregator, and the aggregator can get the aggregated result based on the merged sketch. There are also some works [57] that try to explore the operations that can be supported by merge. Specifically, prior works mainly use merge to get the statistics over union ($A \cup B$), while [57] discusses how to use merge to support other operations like intersection, Jaccard similarity, and relative complement.

Continuous monitoring model: In addition to the linear mergeability [9] discussed in §2.2, there are also theoretical works about continuous monitoring for distributed systems. However, some of these efforts [58] assume that every worker has a lot of resources to record all flow sizes, while some of them [59, 60] only focus on a specific sketch of a task. In addition, these works [53] all aim to minimize the total communication cost, and failed to consider the computation cost. As shown in §2.2, the main bottleneck in the multicore scenario is the computation cost.

Real-time telemetry: Timeliness is an important property in distributed monitoring. Trumpet [61] is an event monitoring system in which users define network-wide events. There is a centralized controller which installs triggers at end-hosts where triggers test for local conditions, and the controller aggregates these signals and tests for the presence of specified network-wide events. They use a hash table in each end-host and assume that the end-host can process all packets with full accuracy. However, in our high-volume, multicore context, each core in a server does not have enough computation and memory resources to do so.

9 Conclusions

Today's networked applications require multicore packet processing and distributed flow monitoring. While sketches have emerged as a resource-efficient and highly accurate measurement primitive, fundamental limitations remain. Prior distributed solutions merge sketches from all cores and bring significant accuracy degradation and resource overhead. In this paper, we propose OctoSketch to scale sketches to multicore scenarios using a continuous, change-based mechanism to aggregate multicore measurements. OctoSketch can be applied to a broad range of sketches and answer various sketch-based queries. Our evaluation shows that OctoSketch can achieve significantly higher accuracy and throughput than prior sketch-merge techniques.

Acknowledgments: We thank the anonymous reviewers and our shepherd Behnaz Arzani for their thorough comments and feedback. This work was supported in part by NSF grants CNS-2107086, SaTC-2132643, CNS-2106946, and the Red Hat Collaboratory award.

References

[1] Georgios P. Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *NSDI 2018*, pages 171–186. USENIX Association, 2018.

[2] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for NFV applications. In *SOSP 2015*, pages 121–136. ACM, 2015.

[3] Brocade vyatta 5400 vrouter. http://www.brocade.com/products/all/networkfunctions-virtualization/product-details/5400-vrouter/index.page.

[4] DPDK-Based Load Balancer. https://dpdksummitapac2021.sched.com/event/hdLm.

[5] Seyed Kaveh Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *USENIX Security 2015*, pages 817–832. USENIX Association, 2015.

[6] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *SOSP 2017*, pages 121–136. ACM, 2017.

[7] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *FAST 2019*, pages 143–157. USENIX Association, 2019.

[8] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. of SIGCOMM*, 2017.

[9] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. In *PODS 2012*, pages 23–34. ACM, 2012.

[10] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.

[11] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 2004.

[12] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM*, 2016.

[13] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: adaptive and fast network-wide measurements. In *SIGCOMM 2018*, pages 561–575. ACM, 2018.

[14] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: robust and general sketch-based monitoring in software switches. In *SIGCOMM 2019*. ACM, 2019.

[15] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 226–239, 2020.

[16] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *SIGCOMM 2021*, pages 207–222. ACM, 2021.

[17] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *ICNP 2018*, pages 313–323. IEEE Computer Society, 2018.

[18] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic switch programming with p4all. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 168–174, 2020.

[19] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. SketchLib: Enabling efficient sketch-based monitoring on programmable switches. In *USENIX NSDI*, 2022.

[20] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. {HeteroSketch}: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *In USENIX NSDI)*, pages 719–741, 2022.

[21] Peiqing Chen, Yuhan Wu, Tong Yang, Junchen Jiang, and Zaoxing Liu. Precise error estimation for sketch-based flow measurement. In *ACM Internet Measurement Conference (IMC)*, 2021.

[22] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. SCREAM: sketch resource allocation for software-defined measurement. In *CoNEXT 2015*, pages 14:1–14:13. ACM, 2015.

[23] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. Flymon: enabling on-the-fly task reconfiguration for network measurement. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 486–502, 2022.

[24] Data plane development kit. https://www.dpdk.org/.

[25] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In *CoNEXT 2018*, pages 54–66. ACM, 2018.

[26] ConnectX-5 Ex. https://www.mellanox.com/related-docs/oem/dell/PB_ConnectX-5_Ex_Card_dell.pdf.

[27] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of open vswitch. In *NSDI 15*, pages 117–130. USENIX Association, 2015.

[28] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: fine grained traffic engineering for data centers. In *Co-NEXT '11*. ACM, 2011.

[29] Thomas Locher. Finding heavy distinct hitters in data streams. In *SPAA 2011*, pages 299–308. ACM, 2011.

[30] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *SIGCOMM 2018*, pages 576–590. ACM, 2018.

[31] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *SIGCOMM 2017*, pages 113–126. ACM, 2017.

[32] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI 2013*. USENIX Association, 2013.

[33] Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. https://tools.ietf.org/html/rfc3176.

[34] Sajad Shirali-Shahreza and Yashar Ganjali. Flexam: flexible sampling extension for monitoring and security applications in openflow. In *HotSDN 2013*, 2013.

[35] Intel dynamic load balancer (intel dlb) - accelerating elephant flow. https://builders.intel.com/docs/networkbuilders/intel-dynamic-load-balancer-intel-dlb-accelerating-elephant-flow-technology-guide-1677672283.pdf.

[36] Daniel Rothchild, Ashwinee Panda, Enayat Ullah, Nikita Ivkin, Ion Stoica, Vladimir Braverman, Joseph Gonzalez, and Raman Arora. Fetchsgd: Communication-efficient federated learning with sketching. In *ICML 2020*, volume 119, pages 8253–8265. PMLR, 2020.

[37] The CAIDA UCSD Anonymized Internet Traces. http://www.caida.org/data/passive/passive_dataset.xml.

[38] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). In *ESA 2003*, volume 2832 of *Lecture Notes in Computer Science*, pages 605–617. Springer, 2003.

[39] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156, 2007.

[40] Charles Masson, Jee E. Rim, and Homin K. Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proc. VLDB Endow.*, 12(12):2195–2205, 2019.

[41] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *INFOCOM 2017*, pages 1–9. IEEE, 2017.

[42] Abhinandan Das, Sumit Ganguly, Minos N. Garofalakis, and Rajeev Rastogi. Distributed set expression cardinality estimation. In *VLDB 2004*, pages 312–323. Morgan Kaufmann, 2004.

[43] Minos N. Garofalakis. Distributed data streams. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 883–890. Springer US, 2009.

[44] Brian Babcock and Chris Olston. Distributed top-k monitoring. In *SIGMOD 2003*, pages 28–39. ACM, 2003.

[45] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *SOSR 2018*, pages 8:1–8:7. ACM, 2018.

[46] cpulimit. https://github.com/opsengine/cpulimit.

[47] xxHash Library. http://www.xxhash.com/.

[48] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *ATC*, pages 1–14, July 2019.

[49] Source code related to OctoSketch. https://github.com/Froot-NetSys/OctoSketch.

[50] Fast Concurrent Queue. https://github.com/cameron314/readerwriterqueue.

[51] David MW Powers. Applications and explanations of Zipf's law. In *Proc. EMNLP-CoNLL*. Association for Computational Linguistics, 1998.

[52] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.

[53] Emmanuelle Anceaume, Yann Busnel, Nicolo Rivetti, and Bruno Sericola. Identifying global icebergs in distributed streams. In *SRDS 2015*, pages 266–275. IEEE Computer Society, 2015.

[54] Queue management and load balancing on intel architecture. https://builders.intel.com/docs/networkbuilders/SKU-343247-001US-queue-management-and-load-balancing-on-intel-architecture.pdf.

[55] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostic. RSS++: load and state-aware receive side scaling. In *CoNEXT 2019*, pages 318–333. ACM, 2019.

[56] Ruijie Miao, Yinda Zhang, Zihao Zheng, Ruixin Wang, Ruwen Zhang, Tong Yang, Zaoxing Liu, and Junchen Jiang. High-performance sketch-based measurement over arbitrary partial key query. *IEEE/ACM Transactions on Networking*, 2023.

[57] Jakub Lemiesz. On the algebra of data sketches. *Proc. VLDB Endow.*, 14(9):1655–1667, 2021.

[58] Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD 2006*, pages 289–300. ACM, 2006.

[59] Ke Yi and Qin Zhang. Optimal tracking of distributed heavy hitters and quantiles. In *PODS 2009*, pages 167–174. ACM, 2009.

[60] Zengfeng Huang, Ke Yi, and Qin Zhang. Randomized algorithms for tracking distributed count, frequencies, and ranks. In *PODS 2012*, pages 295–306. ACM, 2012.

[61] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM 2016*, pages 129–143. ACM, 2016.

[62] Sketch in DPDK. https://github.com/DPDK/dpdk/commit/db354bd2e1f82294ddeb667a1dbca27a325d1eb4.

A Background of sketches

Here we give a background of sketching algorithms and the applications of sketches on software platforms.

Sketch example: Count-Min sketch. The Count-Min sketch [10] maintains d arrays of counters, with w counters in each array. When a packet arrives, the Count-Min sketch calculates d independent hash values from the *flow identity* of the packet (*e.g.*, 5-tuple, source IP, and destination IP). Each hash function offers an independent counter position in each array, and the Count-Min sketch subsequently increases the associated counter per array. When querying the size of a flow, its estimation is given by the minimum value among the associated counters for that flow.

Applications of sketches: Performing telemetry under high-performance packet processing is an important application for sketches. As an example, DPDK has included sketching algorithms in their library to provide an efficient way to profile the traffic for heavy hitters [62]. In addition, prior works [4] apply sketches to a DPDK-based Load Balancer. Specifically, they use sketches to detect heavy hitters and distribute them to multiple idle worker cores for parallel packet processing. Moreover, prior system work also applies sketches to super-spreader/DDoS detection [32] and key-value cache [7, 21].

B Analysis

Theorem 1. *For OctoSketch for Count-Min sketch, let* $d = \log_2 \delta^{-1}$ *and* $l = 2\varepsilon^{-1}$. *For any flow* e *and any traffic whose* $L_1 > \varepsilon^{-1} k' \tau$,

$$\Pr\left[\left|\widehat{f}(e) - f(e)\right| > \varepsilon L_1\right] < \delta \tag{4}$$

Proof. Suppose that $\widehat{f}'(e)$ is the estimated size of the Count-Min sketch that works in a single core. Note that $\left|\widehat{f}(e) - f(e)\right| \leqslant \left|\widehat{f}'(e) - f(e)\right| + \left|\widehat{f}(e) - \widehat{f}'(e)\right|$, where the first part $\left|\widehat{f}'(e) - f(e)\right|$ is the original error from the Count-Min sketch, while the second part $\left|\widehat{f}(e) - \widehat{f}'(e)\right|$ is the additional error brought by the OctoSketch.

For the first part, given $d = \log_2 \delta^{-1}$ and $l = 2\varepsilon^{-1}$, the error bound of the Count-Min sketch [10] as shown in its paper is that

$$\Pr\left[\left|\widehat{f}'(e) - f(e)\right| > \varepsilon L_1\right] < \delta. \tag{5}$$

Then, we consider the second part, *i.e.*, the gap of estimated flow size between OctoSketch for Count-Min sketch and the Count-Min sketch that works in a single core. Note that for each worker, there is at most τ flow size information left in the worker's sketch at any time. The total gap between each counter in the aggregator's sketch and the single core baseline is at most $k' \cdot \tau$. Because the gap between the estimation is not larger than the gap between each counter, we can get that $\left|\widehat{f}'(e) - \widehat{f}(e)\right| < k'\tau$. Given $L_1 > \varepsilon^{-1}k'\tau$, we can ensure that $\left|\widehat{f}'(e) - \widehat{f}(e)\right| < \varepsilon L_1$.

Because the Count-Min sketch only overestimates the flow size, we can get that $\widehat{f}'(e) \geqslant f(e)$. Because OctoSketch will always have some flow size information in the worker not updated to the aggregator, we have the query result from the aggregator $\widehat{f}(e) \leqslant \widehat{f}'(e)$. Based on 5 and $\widehat{f}(e) \leqslant \widehat{f}'(e)$, we have

$$\Pr\left[\widehat{f}(e) - f(e) > \varepsilon L_1\right] < \delta. \tag{6}$$

Based on $\widehat{f}'(e) \geqslant f(e)$, $\widehat{f}'(e) \geqslant \widehat{f}(e)$, and $\left|\widehat{f}'(e) - \widehat{f}(e)\right| < \varepsilon L_1$, we have $f(e) - \widehat{f}(e) < \varepsilon L_1$. Based on 6 and $f(e) - \widehat{f}(e) < \varepsilon L_1$, we can get the result. □

Theorem 2. *To achieve the accuracy goal, sketch-merge needs to send* $O\left(\Delta^{-1}k \cdot N \cdot dl\right)$ *counters, while OctoSketch needs to send* $O\left(\Delta^{-1}k \cdot N \cdot d\right)$ *counters.*

Proof. We first consider the number of counters needed by the sketch-merge technique. There are $d \cdot l$ counters for each sketch. To guarantee that the gap of estimated size is smaller than Δ, sketch-merge needs to send the whole sketch ($d \cdot l$ counters) to the aggregator for every $\frac{\Delta}{k}$ packets. Therefore, sketch-merge needs to send the sketch at least $\frac{k \cdot N}{\Delta}$ times. As a result, sketch-merge needs to send $O\left(\frac{k \cdot N}{\Delta} \cdot d \cdot l\right)$ counters.

For OctoSketch, to guarantee the error, OctoSketch needs to set the threshold as $\frac{\Delta}{k}$. Note that each packet only accesses one counter in the array. Because there are N packets in the traffic, each array sends at most $\frac{k \cdot N}{\Delta}$ counters. As a result, OctoSketch needs to send $O\left(\frac{k \cdot N}{\Delta} \cdot d\right)$ counters. □

Let $L_k = \sqrt[k]{\sum_e f^k(e)}$ be the k-th norm of the frequency vector of the traffic. Then, we show the error bound of the OctoSketch for the Count sketch.

Theorem 3. *For OctoSketch for the Count sketch, let* $d = O(\log_2 \delta^{-1})$ *and* $l = 8\varepsilon^{-2}$. *For any flow* e *and any traffics whose* $L_2 > 2\varepsilon^{-1}k'\tau$,

$$\Pr\left[\left|\widehat{f}(e) - f(e)\right| > \varepsilon L_2\right] < \delta \tag{7}$$

Proof. Similar to that of the Count-Min sketch, we first analyze the error brought by the Count sketch. Suppose that $\widehat{f}'(e)$ is the estimated size of the Count sketch that works in a single core. Given $d = O(\log_2 \delta^{-1})$ and $l = 8\varepsilon^{-2}$, based on the error bound of the Count sketch [11], we have $\Pr\left[\left|\widehat{f}'(e) - f(e)\right| > \frac{\varepsilon}{2}L_2\right] < \delta$. Then we analyze the error brought by OctoSketch. Given the threshold τ of each worker and $L_2 > 2\varepsilon^{-1}k'\tau$, we can make sure that

$$\left|\widehat{f}'(e) - \widehat{f}(e)\right| < k'\tau < \frac{\varepsilon}{2}L_2$$

Based on the triangle inequality, we have

$$\Pr\left[\left|\widehat{f}(e) - f(e)\right| > \varepsilon L_2\right] < \delta$$

□

Then, we show the error bound of the OctoSketch for HyperLogLog. Because HyperLogLog works for different query tasks from the Count-Min sketch and the Count sketch, we define different symbols for HyperLogLog. Let $\widehat{Z}$ be the estimated cardinality of HyperLogLog, $\widehat{Z}'$ be the estimated cardinality of the OctoSketch for HyperLogLog, m be the number of counters used in the HyperLogLog, α_m is the constant used by HyperLogLog given m, $C[i]$ be the i^{th} counter in the HyperLogLog, and $C'[i]$ be the i^{th} counter in the OctoSketch for HyperLogLog. As shown in the paper [39], α_m is always smaller than 0.73 regardless of m, and

$$\widehat{Z} = \alpha_m m^2 \left(\sum_{i=1}^{m} 2^{-C[i]}\right)^{-1}$$

Theorem 4. *If* $\widehat{Z} > 2\alpha_m m^2 2^{\tau-2}$, *we have*

$$\widehat{Z} = \widehat{Z}'$$

Proof. OctoSketch can guarantee that, for any i, $\left|2^{C[i]} - 2^{C'[i]}\right| < 2^{\tau}$, and $C'[i] = \tau - 1$ by default. As all counters are integers, we can get that

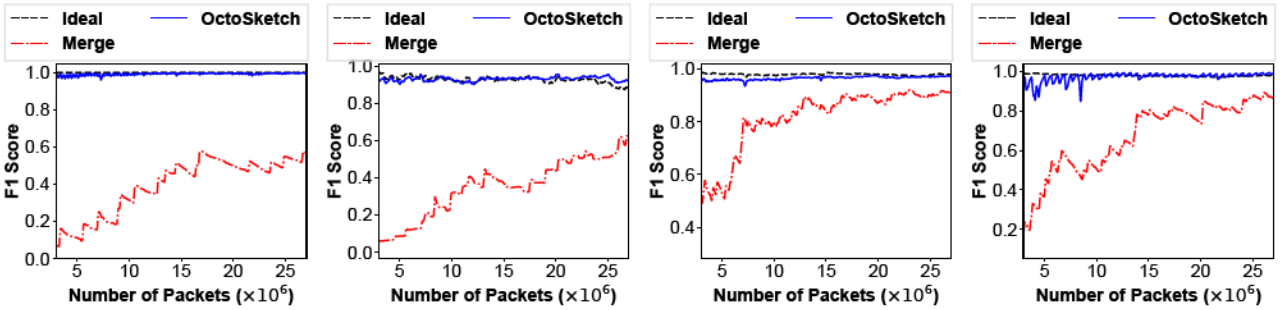
$$C'[i] = \begin{cases} 0, & C[i] < \tau - 1 \\ C[i], & C[i] \geqslant \tau - 1 \end{cases}$$

Therefore, if all $C[i] \geqslant \tau - 1$, we can guarantee that $\widehat{Z} = \widehat{Z}'$. If there is any $C[i] < \tau - 1$,

$$\widehat{Z} \leqslant \alpha_m m^2 \left(2^{-(\tau-2)}\right)^{-1} = 2\alpha_m m^2 2^{\tau-2}$$

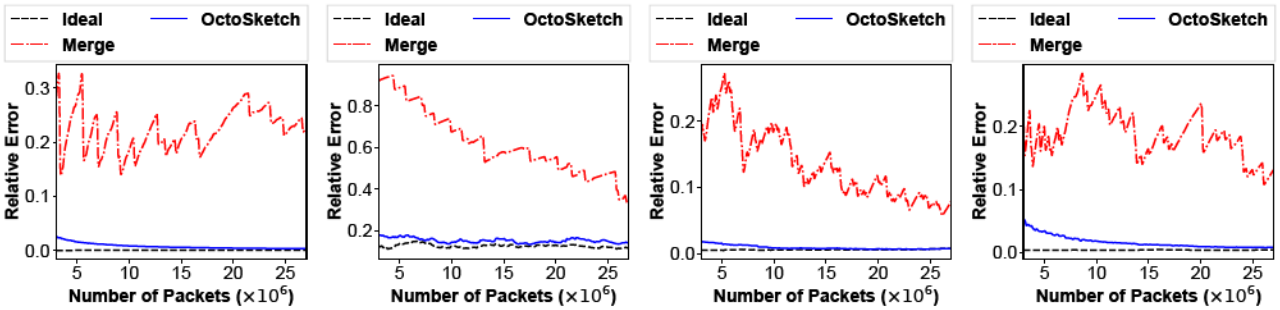
□

Different from the Count-Min sketch, not all packets can change the value of the accessed HyperLogLog. As the property of HyperLogLog, if there are N distinct flow keys mapped to the counter, the counter will only be updated $O(\log N)$ times. In other words, the number of updates is much smaller than the number of packets in the workloads. For example, there are about 30 million packets in 1 minute in CAIDA traces [37], while there are only around 1 million distinct flow keys, and only part of them can update the accessed counter. Therefore, in our experiments, the threshold $\tau = 2$ is often enough for the aggregator to process all updates.



(a) CocoSketch (b) Locher Sketch (c) Elastic Sketch (d) UnivMon

Figure 20: F1 scores on other sketches



(a) CocoSketch (b) Locher Sketch (c) Elastic Sketch (d) UnivMon

Figure 21: Relative errors on other sketches

C Extensions to other sketches

In this section, we show how we apply OctoSketch to the other eight sketches in detail.

Count sketch [11] and UnivMon [12]: There are also multiple arrays of counters in the Count sketch. Count sketch offers unbiased size estimation to each flow and thus each counter value in the Count sketch can be positive or negative. Therefore, as discussed in §4.4, OctoSketch uses the absolute value to check whether the counter is large enough. Moreover, UnivMon is composed of multiple Count sketches, and the insertion process is the same for each Count sketch.

LogLog [38], HyperLogLog [39], and Locher sketch [29]: There is only one array of small counters (*e.g.*, 4-bit counter) for LogLog and HyperLogLog. Note that the insertion logics of LogLog and HyperLogLog are the same, and they mainly differ in the query method. Therefore, we apply OctoSketch to both of them in the same way. The details of applying OctoSketch to HyperLogLog are discussed in §4.4. Locher sketch is comprised of multiple arrays of HyperLogLog [39] estimators, and the insertion process is the same for each HyperLogLog.

DDSketch [40]: There is one array of counters in the DDSketch. As the insertion logic of the DDSketch is similar to that of the Count-Min sketch, we apply OctoSketch to the DDSketch in the same way.

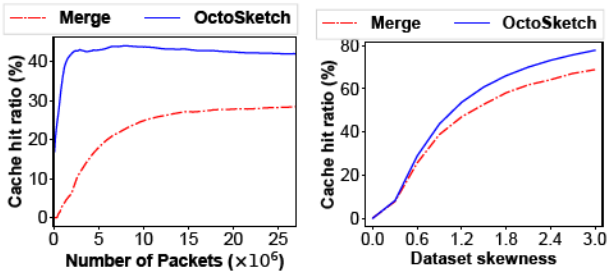
CocoSketch [16]: CocoSketch is composed of a number of (*e.g.*, 2) arrays of buckets, and each bucket stores a key and a counter. CocoSketch does not need an additional heap to record flow keys. Therefore, OctoSketch for CocoSketch also does not need the heap in the aggregator. As discussed in §4.4, OctoSketch will send the whole bucket (key and counter) to the aggregator, if the counter is large enough. In the aggregator, OctoSketch uses the same insertion logic as CocoSketch to update its own aggregated sketch.

Elastic sketch [13]: There are two parts in the Elastic sketch: a *heavy part* and a *light part*. The light part is a Count-Min sketch. The heavy part is an array of buckets to record heavy flow keys, where each bucket also contains a key and a counter similar to that of the CocoSketch. Elastic sketch also does not need an additional heap to record heavy flow keys. OctoSketch for Elastic sketch should keep both the heavy part and the light part in both workers and aggregator and does not need the heap in the aggregator. To apply OctoSketch, the light part's insertion logic is the same as that of the OctoSketch-optimized Count-Min sketch. For the heavy part, OctoSketch will send the flow key and the counter to the aggregator if the counter in the bucket is sufficiently large as shown in §4.4. In the aggregator, OctoSketch uses the same insertion logic as Elastic sketch to update its own aggregated sketch.

D Evaluation

D.1 Parameters

Sketch parameters: For Count-Min sketch and Locher sketch, we use 3 arrays of 2^{16} counters per array. For CocoSketch, we use 2 arrays as suggested in their paper, and there are 2^{16} buckets per array. For Elastic sketch, only one Count-Min sketch array is used (3×2^{16} counters) based on



(a) Over time (b) Over skewness

Figure 22: Cache hit ratio

the paper. For UnivMon, we use 6 Count sketches. We ensure that sketch-merge and OctoSketch use the same configuration (and memory) in the aggregator. As shown in §4, for OctoSketch, the sketch in each worker can use a smaller counter size without accuracy loss. In experiments, each counter in the worker of OctoSketch is 8-bit, while each counter in the worker of sketch-merge is 32-bit. Specifically, the Count-Min sketch for sketch-merge in workers needs 768KB memory, while it only needs 192KB memory for the workers of OctoSketch. For the resource allocation policy in OctoSketch, we set $\alpha = 0.25$. We set the cycle of sketch-merge based on the maximum frequency the aggregator can support.

D.2 Figures

Due to space limitations, we move the Figure 20 and 21 for §7.2 to the appendix.

D.3 Case Study: Key-Value Cache

In this section, we show the benefit of OctoSketch for a key-value cache. Similar to the [21], we run experiments on the simulator of DistCache [7]. Specifically, the traffic is distributed over multiple workers, and the aggregator aggregates the statistics over multiple workers. Once an object is detected as a hot object on the aggregator, it is cached. If an object is no longer recognized as a hot object, it will be offloaded from the cache. We calculate the cache hit ratio while using OctoSketch and sketch-merge for hot object detection.

Cache hit ratio over time (Figure 22a): OctoSketch can achieve fast detection and a high cache hit ratio. With less than 2M packets, the cache hit ratio of OctoSketch is higher than 40%. However, sketch-merge needs more than 10M packets to converge. It indicates that sketch-merge spends much time detecting new hot objects which may lead to many cache misses. In addition, due to the different detection rates on hot objects, the cache hit ratio of OctoSketch is still 13% higher than sketch-merge after 20M packets.

Cache hit ratio over skewness (Figure 22b): We also show the final cache hit ratio on datasets with different skewness. The cache hit rate is higher with larger skewness. The cache hit ratio of OctoSketch is around 9% higher than that of the sketch-merge when the skewness is larger than 1.5.