CocoSketch: High-Performance Sketch-based Measurement over Arbitrary Partial Key Query

Yinda Zhang[1,3], Zaoxing Liu[2], Ruixin Wang[1], Tong Yang[1], Jizhou Li[1],
Ruijie Miao[1], Peng Liu[1], Ruwen Zhang[1], Junchen Jiang[3]
[1]Peking University, [2]Boston University, [3]University of Chicago

ABSTRACT

Sketch-based measurement has emerged as a promising alternative to the traditional sampling-based network measurement approaches due to its high accuracy and resource efficiency. While there have been various designs around sketches, they focus on measuring one particular flow key, and it is infeasible to support many keys based on these sketches. In this work, we take a significant step towards supporting *arbitrary partial key queries*, where we only need to specify a full range of possible flow keys that are of interest before measurement starts, and in query time, we can extract the information of any key in that range. We design *CocoSketch*, which casts arbitrary partial key queries to the subset sum estimation problem and makes the theoretical tools for subset sum estimation practical. To realize desirable resource-accuracy tradeoffs in software and hardware platforms, we propose two techniques: (1) stochastic variance minimization to significantly reduce per-packet update delay, and (2) removing circular dependencies in the per-packet update logic to make the implementation hardware-friendly. We implement CocoSketch on four popular platforms (CPU, Open vSwitch, P4, and FPGA) and show that compared to baselines that use traditional single-key sketches, CocoSketch improves average packet processing throughput by 27.2× and accuracy by 10.4× when measuring six flow keys.

CCS CONCEPTS

• **Networks** → **Network monitoring**; **Network measurement**;

KEYWORDS

Sketch; Arbitrary Partial Key Query; P4; FPGA

ACM Reference Format:
Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, Junchen Jiang. 2021. CocoSketch: High-Performance Sketch-based Measurement over Arbitrary Partial Key Query. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–28, 2021, Virtual Event, USA*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3452296.3472892

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM '21, August 23–28, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8383-7/21/08…$15.00
https://doi.org/10.1145/3452296.3472892

1 INTRODUCTION

Network monitoring and measurement have been critical to various network management tasks, such as traffic engineering [1–7], accounting [8–12], load balancing [13–16], flow scheduling [17–19], and anomaly detection [20–22]. These tasks often require timely and accurate estimates of the network flow metrics, *e.g.*, heavy hitters [23–26], flow size distribution [27], or heavy changes [28, 29]. In response, recent efforts have demonstrated that sketching algorithms (sketches) can estimate these metrics with high fidelity at a high throughput using only small amounts of resources [30, 31].

At a high level, existing sketch-based designs commonly focus on estimating statistics defined over a *single* flow key. A flow key can be a specific header field (*e.g.*, SrcIP, DstIP), a combination of fields (*e.g.*, 5-tuple), or a subset of bits in a field (*e.g.*, any prefix in SrcIP). For instance, flow scheduling needs to track heavy hitters defined on the 5-tuple [17], while SYN flood detection needs to count distinct SrcIPs [32].

While recent efforts on single-key sketches have made significant progress [30, 33–38], it is impractical to use these sketches to measure *multiple* flow keys simultaneously. First, existing sketches [33, 39, 40] keep one independent sketch for each key, making it hard to scale to even a handful of keys given the limited compute/memory resources in commercial switches [31, 41] (as shown in §2.3). Second, they require operators to pre-define the set of flow keys before the measurement starts. However, in many use cases, such as network diagnosis and security, it might be difficult to enumerate a few keys that must be measured ahead of time [21, 42–48]. For instance, DDoS detection may track large flows defined on tens of flow keys, including SrcIP/DstIP, the 5-tuple, and arbitrary prefixes of them [21]. On a Tofino switch (*e.g.*, 48 ALUs) [49], a Count-Min sketch for each key requires eight ALUs, making it infeasible to run more than six sketches.

We define a new class of problem called *arbitrary partial key query*, which "late binds" what keys a sketch should support. Specifically, operators only need to pre-define a broad key range beforehand (called the *full key* k_F), and during query time, they can still query the flow size of any key that is a part of k_F (called *partial key*). For instance, if the full key k_F is the 5-tuple, the system should estimate the flow size of any partial keys of the 5-tuple, such as SrcIP and any prefix of SrcIP.

An ideal system for arbitrary partial key queries should meet three requirements: (1) *fidelity* (provable accuracy guarantee on any partial keys), (2) *resource efficiency* (high throughput using minimal memory), and (3) *compatibility* (on various software and hardware platforms, *e.g.*, Open vSwitch [50], PISA [49], and FPGA [51]).

Unfortunately, existing solutions that might support arbitrary partial key queries fall short on at least one requirement, as summarized in Table 1. R-HHH [39] reduces the overhead of updating

Solutions	Fidelity	Resource	Compatibility
Sketch per key (R-HHH)	✓		
Full-key sketch (§2.3)		✓	✓
Unbiased SpaceSaving	✓	✓	
CocoSketch (ours)	✓	✓	✓

Table 1: Our work v.s. prior solutions.

multiple sketches (one for each partial key) by selectively updating only $O(1)$ sketches per packet, but this technique will significantly increase the memory usage needed to achieve the same error bound. For instance, to find the hierarchical heavy hitters of SrcIP (*i.e.*, 32 prefixes), it will use 32 single-key sketches and 32MB total memory space (each sketch needs 1MB memory space to achieve a 95% F1 Score [52]), which already exceeds the 9MB memory available on Xilinx Alveo U280 FPGA [51]. Alternatively, we can use a single-key sketch to measure full-key flow sizes and recover partial-key flow sizes by aggregating full-key flows. However, prior work [53] has shown that this approach might have large estimation errors, which also corroborates our empirical evaluation (§7.5).

In this work, we present **CocoSketch** (**Co**rnu**co**pia Sketch), a sketch-based flow measurement system that supports arbitrary partial key queries. In contrast to the baselines that maintain multiple single-key sketches, CocoSketch achieves provable accuracy guarantees for arbitrary partial key queries but drastically reduces memory usage and update delay by maintaining only one sketch. Moreover, CocoSketch can be efficiently implemented on both software and hardware platforms.

CocoSketch shares the theoretical basis with Unbiased SpaceSaving (USS) [53], a recent technique for subset sum estimation [54]. Given a set of items, each with a weight, the subset sum estimation problem estimates the total weight of any subset of items. The problem of arbitrary partial key queries can be cast as the subset sum estimation problem: the size of a partial-key flow e equals the total size of a subset of full-key flows that match on the partial key with e. For instance, the size of a flow e defined by the fields of SrcIP and DstIP equals the total size of all 5-tuple flows that share the SrcIP and DstIP with e. The key idea behind USS is the *variance minimization* technique, which minimizes the variance of its subset-sum estimation. Unfortunately, the update delay of USS grows proportionally with more flows recorded in the system (on a scale of 10^4), so a straightforward implementation would have low throughput on CPUs (§7.5), and it cannot be supported by some resource-constraint hardware [32].

The challenge of CocoSketch lies in how to practically apply the theory of subset sum estimation to the partial key query problem. We propose two main techniques. (a) Inspired by USS, we introduce a technique called *stochastic variance minimization*. It harnesses "power-of-d choices" to drastically reduce the per-packet update delay while still maintaining a low total variance of size estimates on all flows. Our analysis in §5 shows that, like USS, our size estimates on any partial keys are unbiased and have bounded variances. (b) Due to *circular dependencies* among the per-packet update operations, naively implementing stochastic variance minimization on programmable switches can be infeasible (even when it runs on FPGA, the throughput is low). To make it runnable on hardware platforms, we further remove circular dependencies by parallelizing the operations of stochastic variance minimization in a way that incurs only minor increases in estimation errors. §7.5 empirically shows that the F1 Scores drop by less than 10% after removing the circular dependencies.

We implement CocoSketch prototypes on representative software (*e.g.*, CPU and Open vSwitch (OVS)) and hardware platforms (*e.g.*, programmable ASIC and FPGA). Our evaluation shows that to handle multiple partial keys under three measurement tasks (heavy hitter detection, heavy changes, and hierarchical heavy hitters (HHHs)), CocoSketch achieves 27.2× higher packet processing throughput than baselines such as UnivMon [33], Elastic Sketch [30], R-HHH [39], and USS [53], while reducing estimation error by 10.4× (and almost 40k× in HHHs).

2 BACKGROUND AND MOTIVATION

In this section, we begin with the background on single-key sketches and contrast them with the new problem of arbitrary partial key queries. Then we discuss the potential applications of arbitrary partial key queries and elaborate why existing solutions fall short.

2.1 Sketches for Network Measurement

Sketching algorithms (sketches) process data streams to estimate various statistics in an online fashion. Compared to traditional sampling-based techniques [55–58], sketches [24, 25, 29, 30, 33–35, 59] are particularly attractive for network measurement because of their provable and tunable accuracy-memory tradeoffs, allowing sketches to fit in network devices with diverse resource constraints.

Sketches for network measurement have mainly followed a **single-key** paradigm: each packet is identified as a <*key, value*> pair to be inserted into the sketch, where the key is a flow identifier defined by *one* combination of packet-header fields selected by the operator *before* the measurement starts, and the value is the packet count or the byte count of this flow. For instance, operators can set up a heavy-hitter sketch that extracts each packet's 5-tuple instance as the key and the packet size/count as the value to update the sketch. Periodically, the sketch will report the 5-tuple instances with the largest flow sizes. In network measurement, single-key sketches are widely used to count distinct flows [32, 60] and detect heavy hitters [29, 59], significant changes of traffic patterns [28, 61], and anomalies (*e.g.*, entropy estimation) [62–64], among others. Recent efforts [30, 31] have also improved the fidelity, resource efficiency, or hardware compatibility of single-key sketches.

2.2 Arbitrary Partial Key Problem

In contrast to the single-key paradigm, we define a new class of problems called *arbitrary partial key query*, which supports queries on multiple keys without the need to pre-define which keys to measure. Instead, operators only need to specify a *full key* that incorporates all *partial keys* that might be queried in the future. We formally define the problem as follows.

Definition 1 (Partial Key). *A key k_P is a* partial key *of key k_F (denoted by $k_P \prec k_F$), if there is a mapping $g(\cdot) : k_F \to k_P$, and for any flow $e \in k_P$ defined on key k_P, we have $f(e) = \sum_{e' \in k_F, g(e')=e} f(e')$, where $f(e)$ is a statistic (e.g., size) of flow e.*

For example, the size of a flow e of a partial key (*e.g.*, (56.49.82.∗)) equals the sum of the size of full-key flows $\{e'|g(e') = e\}$ (*e.g.*, {(56.49.82.0), ..., (56.49.82.255)}). Note that a partial key can be any subset of fields in a full key, *e.g.*, (SrcIP, DstIP) is a partial key of the 5-tuple full key.

Definition 2 (Arbitrary Partial Key Query). *Given a full key k_F and a metric function f, return the $f(e)$ of any flow $e \in k_P$ for any partial key $k_P \prec k_F$.*

In this paper, we assume that f is a flow size function. The problem of arbitrary partial key query enables a more flexible way of querying flow statistics without specifying which keys to query beforehand. We can first define the full key as the union of all keys that might be needed and deploy one sketch of the full key, and in query time, operators can recover the size of any partial key.

Use cases of arbitrary partial key query: The ability to answer arbitrary partial key queries enables a broad spectrum of potential use cases. In Trumpet [65], applications, such as guiding rule placement [66], coflow scheduling [67], and multi-key rate limiting [68], require estimation results over many different keys. For instance, there are often thousands of rules in rule management [66], which require measurement on different combinations of fields/prefixes in the 5-tuple (*i.e.*, tens to hundreds of different keys). Moreover, in security and diagnosis scenarios [21, 42–48], it is often hard to predict which keys are relevant to future security incidents unless we exhaustively track all possible keys. For example, DDoS detection needs various metrics (*e.g.*, heavy hitters, distinct flows) over potentially many flow keys, including SrcIP/DstIP, the 5-tuple, and any arbitrary prefixes of them [21].

2.3 Existing Solutions and Limitations

In this section, we show why existing single-key sketches are ill-suited to arbitrary partial key queries, whereas the theory literature offers a promising yet impractical approach.

One single-key sketch per key: One strawman to realize arbitrary partial key queries is by creating one single-key sketch (*e.g.*, [33, 40]) for each possible partial key. This method does not scale to many keys because deploying and updating many sketches simultaneously can cause significant storage and update overheads. Recent work R-HHH [39] can reduce the per-sketch operation overhead on multiple sketches (by randomly selecting $O(1)$ sketches to be updated per packet). While this sampling-based approach improves the sketch throughput in software, it will significantly increase resource usage to reach the same error bound or lower the accuracy given the same amount of memory space [39]. In hardware switches such as Barefoot Tofino [49], its resource usage (summarized in Table 2) will grow linearly with more sketches, so this approach cannot support more than a handful of keys.

Full-key sketch with post recovery: Alternatively, we can deploy a single-key sketch for the full key and use the two following ways to recover the size of a partial-key flow from the full-key flow information, though neither is ideal. (i) One way is to recover the size of each partial-key flow by querying and aggregating the sizes of all possible full-key flows that belong to the partial-key flow, but the number of such full-key flows can be prohibitively large: *e.g.*, with the 32-bit SrcIP as the partial key and the 104-bit 5-tuple as the full key, one needs to query $(2^{104}/2^{32})=2^{72}$ full-key flows to estimate merely one partial-key flow. (ii) An alternative way is that, instead of aggregating the estimates of all full-key flows, we aggregate only the full-key flows that are explicitly logged in the sketch. Prior analysis [53], however, suggests that aggregating such a subset of flows can yield high estimation bias and variance, and our evaluation in §7.5 indeed shows that it has higher estimation errors on partial keys than on the full key.

Resource Name	Count-Min	R-HHH
Hash Distribution Unit	20.83%	22.22%
Stateful ALU	16.67%	16.67%
Gateway	7.81%	8.33%
Map RAM	7.11%	7.11%
SRAM	4.27%	4.27%

Table 2: Resource usage breakdown of one single-key sketch (same configuration as §7.1) on a Tofino switch. The resource bottleneck is the hash distribution unit (in bold). A Tofino switch cannot support more than four single-key sketches.

Theory	Target
Single-key sketches	minimize $\max_e \left(f(e) - \widehat{f}(e)\right)^2$
Subset sum estimation	minimize $\sum_e \left(f(e) - \widehat{f}(e)\right)^2$

Table 3: In contrast to single-key sketches, subset sum estimation optimizes a different accuracy objective that is more suitable for arbitrary partial key queries. $f(e)$ is the real size of flow e. $\widehat{f}(e)$ is the estimated size of flow e.

Subset sum estimation: We advocate for a more promising approach – casting the arbitrary partial key query problem to the *subset sum estimation* problem. As summarized in Table 3, unlike single-key sketches that minimize the maximum estimation error on *individual* keys, subset sum estimation offers an unbiased estimate on the *sum* of all (and any *subset* of) items with minimum variance. It fits nicely with our goal since each partial-key flow size equals the total size of a subset of full-key flows.

Unfortunately, existing work on subset sum estimation, notably Unbiased SpaceSaving (USS) [53], is impractical for network measurement. As will be elaborated in §3.2, USS performs $O(n)$ memory accesses on every arriving packet, where n is the number of flows currently maintained in the system and can be on the scale of 10^4. Such prohibitive per-packet update overhead makes USS hard to keep up with the line rate requirements on software platforms and infeasible to run on some hardware platforms such as Barefoot Tofino [49]. Without changing the algorithm, it might be hard to speed it up with better implementation to achieve desirable performance. §7.2 shows that when accelerated by a hash table and a double linked list, USS still only achieves less than 1/3 the throughput of a single-key sketch.

In summary, re-using single-key sketches is a fundamental mismatch with the accuracy requirements of arbitrary partial key queries, whereas USS fits the accuracy goal of arbitrary partial key queries but falls short on system performance. The following sections will provide more details of USS and how we make it practical for hardware/software-based network measurements.

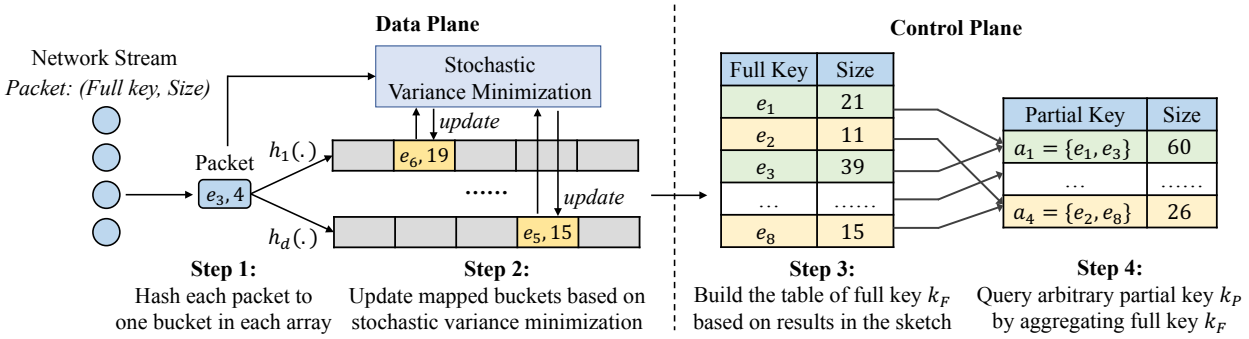


Figure 1: CocoSketch architecture.

3 OVERVIEW

We now give an overview of our solution, CocoSketch, and its two key ideas: stochastic variance minimization (§3.2) and removal of circular dependencies (§3.3).

3.1 Problem Scope

Requirements: CocoSketch has three design requirements:

- **[R1]** Accuracy guarantees over partial keys: CocoSketch should provide accuracy guarantees for all partial key queries. This paper focuses on the accuracy of flow size-related queries (*e.g.*, heavy hitter detection, heavy change detection).
- **[R2]** Compute and memory resource efficiency: CocoSketch should achieve high throughput using small memory footprints, on both software and hardware platforms.
- **[R3]** Compatibility with diverse platforms: CocoSketch should work on both software (*e.g.*, CPU and OVS [50]) and hardware (*e.g.*, FPGA [51] and reconfigurable ASIC [49]) platforms [69–73].

CocoSketch workflow: Before the measurement starts, the operator defines a full key k_F, of which any key that might be queried will be a partial key. k_F can be a large range of packet header fields such as 5-tuple or application-layer headers. Figure 1 shows the workflow of CocoSketch. CocoSketch maintains a single sketch with $d \cdot l$ buckets (where d and l are configurable parameters). On each arriving packet, CocoSketch's data plane updates the sketch in two logical steps:

Step 1: Extract the full key value e of the flow and use d hash functions to map e to d buckets, each from an array of l buckets.

Step 2: Update the counters of the mapped buckets with the packet size using *stochastic variance minimization* (explained shortly).

At the end of each measurement window, CocoSketch's control plane will answer flow size queries defined on any partial key $k_P \prec k_F$, with two logical steps:

Step 3: Based on the sketch maintained by the data plane, first recover the size of each recorded full-key flow.

Step 4: Aggregate the sizes of only the *recorded* full-key flows to infer the size of the flows defined by the queried partial key k_P.[1]

Next, we discuss the two technical ideas of CocoSketch. Figure 2 illustrates CocoSketch's performance advantages over the baselines.

[1]Note that this step would have had no accuracy guarantee, if the sketch were a full-key sketch updated by the traditional single-key sketch algorithm (as discussed in §2.3).

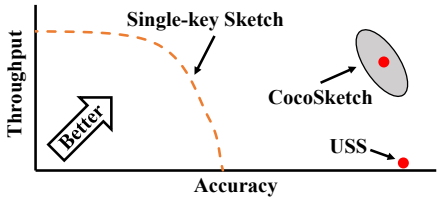


Figure 2: Accuracy-throughput analysis.

3.2 Stochastic Variance Minimization

Variance minimization in Unbiased SpaceSaving: Before describing our approach, we first explain how Unbiased SpaceSaving (USS) [53] minimizes its variance of flow size estimates and why it has a high update delay. For each incoming packet with full-key flow e and packet size w, (1) if e is already tracked in a bucket, then USS increments the counter of e in this bucket by w, so that variance is not increased; (2) otherwise, USS scans *all* buckets to find the min-sized bucket counter C_{min}, increments it by w, and then replaces the flow key associated with the bucket with e with probability $\frac{w}{C_{min}}$. As the number of memory accesses per update is the same as the number of buckets (on a scale of 10^4), USS violates [R2] and [R3]. How to reduce the update cost of USS while still maintaining the high accuracy guarantees?

Reducing update delay: We propose *Stochastic Variance Minimization*: for each packet whose flow is not currently tracked by the sketch, CocoSketch finds the smallest bucket among the d hash-indexed buckets (instead of all buckets), increments the counter, and replaces the flow in the same way as USS (see details in §4.1). In other words, if d is the total number of buckets, CocoSketch would be equivalent to USS. However, CocoSketch sets d to be much smaller (*e.g.*, 2 to 4) than the number of buckets (*e.g.*, 10^4), thus drastically reducing the update delay.

Now, the key question is why updating the bucket among only d buckets (instead of all buckets) per packet still yields unbiased size estimation with small variance?

Preserving estimation accuracy: The intuition is two-fold. Here, we assume that the flow sizes follow a heavy-tailed distribution (*i.e.*, most flows have small sizes).

- First, for a large flow, the counter of the bucket where the flow maps is a quite accurate estimate of its real flow size. This is because, like in USS, its counter is mostly incremented by the same large flow with a small chance of collision.

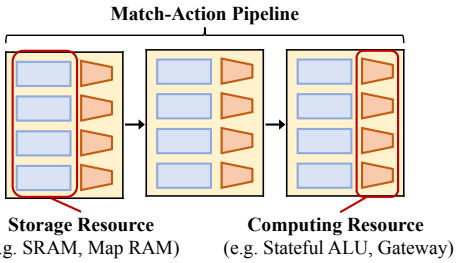


Figure 3: Reconfigurable Match-Action pipeline.

- Second, for small flows, our technique, like USS, spreads out the small flows among the mapped buckets (like a "load balancing" process) to control the per-flow variance. Intuitively, by always incrementing the minimum bucket among d stochastically selected buckets, it enjoys the benefit of "power-of-d choices". Though these buckets' values do not converge as fast as USS, the maximum collisions in all buckets are still bounded with a high probability, after a sufficiently large number of small flows arrive.

Even if the workload is not heavy-tailed, our theoretical analysis (§5.2 and §A.2) shows that CocoSketch can still achieve the same accuracy guarantee as that of USS by adding more buckets to increase the hash space and reduce collisions. In the worst case, we need $o\left((1/\delta)^{1/d}\right)$ times more space than USS, where δ is the probability that a given error threshold is violated. In practice, when $d = 2, \delta = 0.01$, only 1.6× more buckets are needed to achieve accuracy on par with USS. As a result, our evaluation §7.5 shows that compared to USS, CocoSketch improves throughput by 100× with only a marginal drop in accuracy (less than 3% drop in F1 Score).

3.3 Circular Dependency Removal

While stochastic variance minimization allows CocoSketch to achieve high performance in software platforms, it cannot be efficiently implemented in hardware platforms due to inherent *circular dependencies* in its update operations. We show this issue using the Tofino architecture and propose an effective solution to address it.

Constraints in RMT switches: Figure 3 shows the pipeline architecture of a Tofino switch as an example of RMT (reconfigurable match-action table) switches [74]. Each pipeline consists of multiple stages, and importantly, each stage cannot access the memory of any prior stages. Therefore, any sketch update algorithm must follow a *unidirectional* workflow, *i.e.*, data flow strictly from the first stage to the last stage. Moreover, each pipeline has a limited number (*e.g.*, 12) of stages, and each stage has limited memory (*e.g.*, SRAM and TCAM) and computing (*e.g.*, ALU) resources. Thus, any sketch algorithm must fit in a small memory space and perform a small number of memory accesses per packet.

Circular dependencies and their removal: Unfortunately, stochastic variance minimization introduces two forms of circular dependency as illustrated in Figure 4, making it incompatible with the unidirectional workflow in RMT switches. We design a hardware-friendly algorithm (see details in §4.2) to remove these dependencies for better performance and resource efficiency in hardware.

- First, we need to remove the circular dependency across buckets. Recall that there are $d > 1$ corresponding buckets per packet update. The updates to these d buckets depend on each other, *i.e.*, whether a bucket needs an update depends on the key/value in each of the remaining buckets, causing a circular dependency. To address this, we update each bucket independently and in parallel. Instead of running one instance of stochastic variance minimization on d buckets, we run d instances of stochastic variance minimization, each performed on only one bucket. Clearly, stochastic variance minimization on one bucket may lead to larger errors. To control the errors, we use the *median* value among the d buckets as the final result.

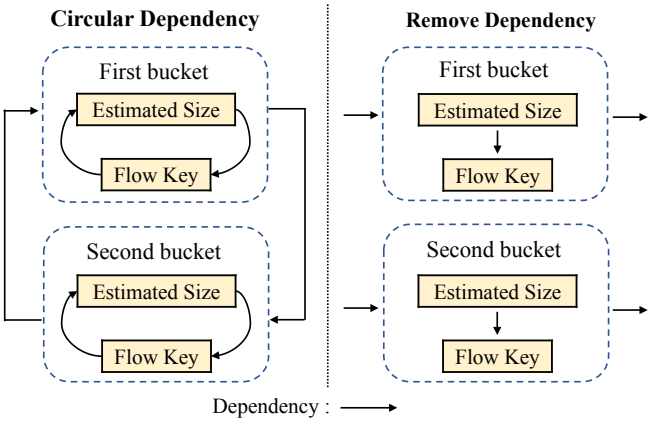


Figure 4: Removing circular dependency

- Second, we further remove the circular dependency between the flow key and its estimated size within each bucket. This dependency comes from the algorithmic design where (i) every counter update depends on the key in the bucket (because the counter should be incremented only if the recorded key matches that of the arriving packet), but (ii) every key update depends on the counter in the bucket (because the probability to replace the key in the bucket depends on its estimated size). To address this, we simplify the update logic, and put the flow key and the estimated size into separate stages. Thus, the update process in one bucket can be pipelined.

By eliminating the circular dependencies both across and within buckets, CocoSketch can be implemented efficiently in hardware (*e.g.*, in FPGA, throughput is improved by 5× compared to a naive implementation with circular dependencies). While removing the circular dependencies might weaken the accuracy guarantee, our evaluation in §7.5 demonstrates that the accuracy drop is not significant (*e.g.*, <10%). Besides RMT and FPGA, we expect that this technique might apply to other pipelined hardware platforms.

4 DETAILED DESIGN

Next, we formally describe the update and query mechanisms of CocoSketch, including the *basic CocoSketch* (§4.1) designed for CPU and OVS to use stochastic variance minimization, and the *hardware-friendly CocoSketch* (§4.2), which optimizes the basic CocoSketch for hardware platforms like programmable ASIC and FPGA. We list the frequently used symbols of this paper in Table 4.

4.1 Basic CocoSketch

Data structure: As shown in Figure 5, the sketch maintains d arrays of l (key, value) pairs. Each (key, value) pair (or called bucket) records a particular full key and its estimated flow size (counter).

Symbol	Description
k_F	the full key
k_P	the partial key
e	the flow
$f(e)$	the real size of flow e
$\widehat{f}(e)$	the estimated size of flow e
d	the number of arrays in CocoSketch
l	the number of buckets in one array
$h_i(.)$	the hash function corresponding to the i^{th} array
$B_i[j]$	the j^{th} bucket in the i^{th} array
$B_i[j].K$	the key field (full key) in $B_i[j]$
$B_i[j].V$	the value field (estimated size) in $B_i[j]$

Table 4: Symbols and notations.

Let $B_i[j]$ $(1 \leqslant i \leqslant d, 1 \leqslant j \leqslant l)$ be the j^{th} bucket of the i^{th} array, and $B_i[j].K$ and $B_i[j].V$ be its key and value. The d arrays are associated with d independent hash functions $h_1(.), ..., h_d(.)$.

Basic CocoSketch insertion: We denote each incoming packet as a pair of (e, w), where e is a particular full key, and w is its increment size. To insert (e, w), we first map e to d buckets (each from one of the d arrays) and use *stochastic variance minimization* to select which bucket to update. There are two cases: (1) if e matches the key in any of the d buckets, increment the value of that bucket by w and return; (2) otherwise, find the bucket with the smallest value (*e.g.*, the bucket in the k^{th} array, $B_k[h_k(e)]$) and update it as follows. We increase $B_k[h_k(e)].V$ by w. And then with probability $\frac{w}{B_k[h_k(e)].V}$, we replace $B_k[h_k(e)].K$ with e. If multiple buckets share the same smallest size value, randomly select one to update. §5 will formally analyze the fidelity of stochastic variance minimization over arbitrary partial keys. Note that for each incoming packet, our insertion logic guarantees that it only updates the value of only one bucket and the key of at most one bucket.

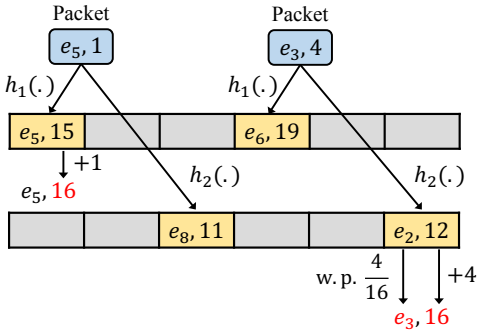


Figure 5: Insertion example in basic CocoSketch (with $d = 2$).

Example (Figure 5): We use $d = 2$ as an example. To insert packet $(e_5, 1)$, we first use two hash functions to map it to two buckets with content $(e_5, 15)$ and $(e_8, 11)$. Because e_5 is already recorded in one of the buckets, we simply increment the corresponding value by 1 (from 15 to 16). To insert packet $(e_3, 4)$, we first map it to the two buckets with content $(e_6, 19)$ and $(e_2, 12)$. Since e_3 is not recorded in either bucket, we identify the bucket with the smallest counter (*i.e.*, $(e_2, 12)$), increment the value by 4 (from 12 to 16), and finally, with probability $\frac{w}{B_k[h_k(e)].V} = \frac{4}{16}$, we replace the key e_2 with e_3.

4.2 Hardware-friendly CocoSketch

We now extend the basic CocoSketch to optimize the resource efficiency in hardware by *removing circular dependencies* in insertions.

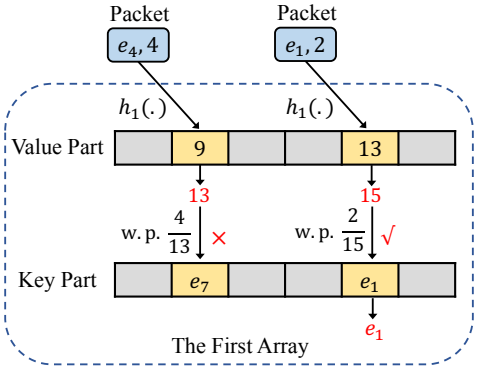


Figure 6: The insertion step in the hardware-friendly CocoSketch. Here we only show the insertion to the first array; the insertion process in each array is the same and independent.

Hardware-friendly insertion: The insertion step of each array is independent of each other in hardware. The reason is that the architecture of network hardware (*e.g.*, FPGA) is usually designed with diverse logical parts running in parallel, and a hardware-friendly algorithm design should leverage the parallelism to better utilize the resources. For each packet, instead of proceeding stochastic variance minimization over d buckets together, we update each bucket independently, as if $d = 1$ in stochastic variance minimization: we always increment the value of the mapped bucket $B_i[h_i(e)]$ by w and replace the key $B_i[h_i(e)].K$ with probability $\frac{w}{B_i[h_i(e)].V}$.

Example (Figure 6): We show an example where each packet triggers an independent update on each of its mapped buckets (the figure only shows the buckets in the first of d arrays). Moreover, the value part and the key part are updated sequentially: since $d = 1$, the value part is always updated and does not depend on the key part.

4.3 Query for Arbitrary Partial Key

Query front-end: We provide a front-end to query arbitrary partial key $k_P \prec k_F$. We first build a table with two columns *(Full Key, Size)* (*i.e.*, a table of estimated size of each recorded flow), by querying the sketch on the recorded full-key flows. In the hardware-friendly CocoSketch, since one flow may appear in multiple arrays, we will take the median estimated size in different arrays as its final estimated size.

The following SQL statement is the interface to query the measurement result of partial key k_P, where g is the mapping from a full key to a partial key, as defined in Definition 1.

```
SELECT g(k_F), SUM(Size)
FROM table
GROUP BY g(k_F)
```

Examples of partial key query (Figure 7): Suppose that the full key is (SrcIP, SrcPort), and we want to query the partial key SrcIP. We first get the full key result (left). Then, we aggregate the result based on the SrcIP fields to get the partial key result (right). There

are two full-key flows which share the SrcIP 19.98.10.26, so we add up their sizes and get the estimated size 1041 (520 + 521) of partial-key flow 19.98.10.26. In contrast, there is only one full-key flow with SrcIP 34.52.73.17, so the estimated size for the partial-key flow 34.52.73.17 is 856.

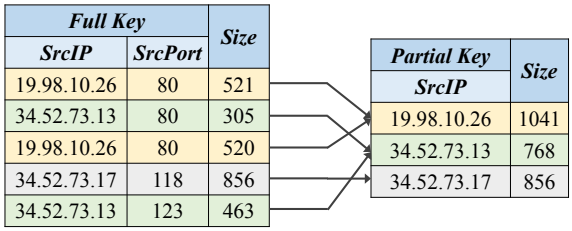


Figure 7: Example queries on partial keys.

5 ANALYSIS

In this section, we provide mathematical analysis for CocoSketch. Due to space constraints, we provide the interpretation of each theorem but defer the detailed proofs to Appendix A.

5.1 Stochastic Variance Minimization

We first analyze how our main technique minimizes the variance.

Variance minimization for subset sum estimation: Let $f(e)$ be the real size of the full key flow e, and $\widehat{f}(e)$ be its estimated size. As shown in §2.3, the target of subset sum estimation is

$$\text{minimize} \quad \sum_e \left(f(e) - \widehat{f}(e)\right)^2 \tag{1}$$

Because USS processes one packet at a time, it minimizes the increment on the sum of variance caused by each insertion, which is shown as follows.

$$\text{minimize} \quad \sum_e \Delta\left(f(e) - \widehat{f}(e)\right)^2 \tag{2}$$

Stochastic variance minimization for $d = 1$: We first discuss the simplest case when CocoSketch has only one array and one associated hash function ($d = 1$). Suppose that the incoming packet is (e_i, w), and it is mapped to the bucket whose recorded key and value are e_j and f_j. To optimize Eq. (2), we need to update the mapped bucket to (e', f') in a way that minimizes the increment of variance for each insertion.

Theorem 1. *The solution to optimize Eq. (2) is*

$$(e', f') = \begin{cases} (e_i, f_j + w), & \text{w.p. } \dfrac{w}{f_j + w} \\ (e_j, f_j + w), & \text{w.p. } \dfrac{f_j}{f_j + w} \end{cases} \tag{3}$$

The proof is in Appendix A.1. Note that regardless of whether e_i matches e_j, the value in the mapped bucket will always be incremented to $f_j + w$. Thus, the update of the value does not depend on the key when $d = 1$. Based on the Eq. (3), we can derive that

Theorem 2. *The minimum increment of variance sum to update the bucket (e_j, f_j) is*

$$\sum_e \Delta\left(f(e) - \widehat{f}(e)\right)^2 = \begin{cases} 2wf_j, & e_i \neq e_j \\ 0, & e_i = e_j \end{cases} \tag{4}$$

Stochastic variance minimization for $d > 1$: Second, we discuss the general case of $d > 1$. CocoSketch updates only one of the mapped buckets. According to Eq. (4), we can derive the variance increment if we will update the i^{th} mapped bucket, and then we can compare their variance increment and choose one whose increment is the least to update. If $e_i = e_j$, the increment of variance is 0. Therefore, we should first update the bucket recording the same flow of full key. If $e_i \neq e_j$, the increment of variance is $2wf_j$. Therefore, if there is no bucket recording the same flow, we should find the bucket with the smallest value field and update it based on Eq. (3).

5.2 Error Bound

Next, we derive the estimation errors of CocoSketch. Let $M = d \cdot l$ be the number of buckets in CocoSketch, where d is the number of arrays and l is the number of buckets in each array. We define $R(e)$ to be the relative error of e, *i.e.*, $R(e) = \left|\frac{\widehat{f}(e) - f(e)}{f(e)}\right|$. Theorem 3 (see Appendix A.2 for the proof) shows the bound of $R(e)$ for the hardware-friendly CocoSketch. Here, $f(\overline{e}) = \sum_{e_i \neq e} f(e_i)$.

Theorem 3. *Let $l = 3 \cdot \epsilon^{-2}$ and $d = O(\log \delta^{-1})$. For any flow e of arbitrary partial key $k_P \prec k_F$,*

$$\mathbb{P}\left[R(e) \geqslant \epsilon \cdot \sqrt{\frac{f(\overline{e})}{f(e)}}\right] \leqslant \delta \tag{5}$$

Interpretation: The same bound on relative errors holds for any partial key, including the full key. On the other hand, Theorem 3 shows that the distribution of error $R(e)$ varies with d and l. For instance, with a larger d (*i.e.*, a smaller δ), the error will be bounded (by $\epsilon \cdot \sqrt{f(\overline{e})/f(e)}$) with a greater probability, which matches our experiments in §7.5.

5.3 Recall Rate

Finally, we derive the recall rate (*i.e.*, how likely a flow is recorded) of the hardware-friendly CocoSketch (see Appendix A.3 for the proof). Let $Z(e)$ be a 0-1 function, with $Z(e) = 1$ if and only if flow e is recorded in the CocoSketch.

Theorem 4. *For any flow e of full key k_F,*

$$\mathbb{P}\left[Z(e) = 1\right] \geqslant 1 - \left(1 + l \cdot \frac{f(e)}{f(\overline{e})}\right)^{-d} \tag{6}$$

Interpretation: According to this theorem, the lower bound of the recall rate will increase as the flow size $f(e)$ increases. In other words, larger flows are more likely to be recorded. Moreover, the lower bound will raise as d increases. To put the theorem in practice, if we want to achieve a 99% recall rate on the heavy hitter that constitutes at least 1% of the whole traffic, (*i.e.*, $f(e)/f(\overline{e}) = 1/99$), we can set $d = 2$ and $l = 900$ (*i.e.*, in total, 1,800 buckets) in the sketch.

6 IMPLEMENTATION

We have implemented CocoSketch on four network platforms: x86 CPU, Open vSwitch (OVS) [50], Xilinx FPGA [51], and Barefoot Tofino [49]. In this section, we describe the implementation of hardware-friendly CocoSketch on FPGA and Barefoot Tofino and defer the implementation of the basic CocoSketch on CPU and OVS to Appendix B. We have open-sourced the artifact on GitHub [75].

6.1 FPGA Platform

FPGA background: FPGAs [51] are based on a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. The main resources of FPGA include Slice LUTs, Slice Registers, and Block RAM Tile. Slice LUTs are lookup tables, which are used to implement combinational logic. Slice Registers are mainly used as cache resources. Block RAM Tile is on-chip block storage, which is the main storage resource.

FPGA implementation: We have implemented the hardware-friendly version (§4.2) on a Xilinx Alveo U280 [51] with full pipelining. We divide our algorithm into four main parts: hash computation, accessing arrays of value, replacement probability calculation, and accessing arrays of key. In FPGA, accessing one BRAM Tile in FPGA needs two cycles while other operations such as hash computation and probability calculation take one cycle. We pipeline all the key/value memory accesses to improve the clock rate. To replace the key in a bucket with some probability $p \in (0, 1]$, we first generate a 32-bit random number $rand$, then replace the key recorded only if $rand \times \frac{1}{p} < 2^{32}$.

6.2 RMT Platform

P4 background: In RMT-based programmable switches [74], each incoming packet will undergo a packet header parser, several pipeline stages, and a deparser. Each stage has a Match-Action Table, where the corresponding actions are performed according to which entry the packets match. Moreover, a small amount of physical resource is allocated to each stage, including SRAM, TCAM, Map RAM, and stateful ALUs. The Map RAM can be used to convert ordinary SRAMs into counters/meters/registers, and the stateful ALUs are used to execute arithmetic operations on the stateful memory.

P4 implementation: We have implemented a P4 prototype of the hardware-friendly CocoSketch on the Tofino switch [49]. We find that the difficulty of implementing the hardware-friendly CocoSketch on the Tofino switch is the calculation of probability. Because the multiplication operation between two variables is not supported, we have to use a different way to calculate the probability. In P4, to replace the key recorded with probability $\frac{1}{value}$, we first generate a 32-bit random number $rand$ and then replace the key recorded only if $rand < \frac{2^{32}}{value}$. Note that the math unit provided by the current Tofino switch only supports approximate division between a constant and a variable. It does the approximate division based on the highest 4 bits of the variable. Given the real replacement probability $p = \frac{1}{value}$, the difference between the real probability and the calculated probability is usually below $0.1p$. For example, if the real replacement probability is $\frac{1}{17} = 5.9\%$, the difference will be only 0.37%. Thus, the approximate division can still calculate the probability with high accuracy.

7 EVALUATION

We conduct extensive experiments to compare CocoSketch with the latest single-key sketches and USS, and demonstrate that:

- CocoSketch achieves significantly higher accuracy when estimating multiple flow keys using the same amount of memory.
- Basic CocoSketch tracks multiple flow keys without accuracy degradation and is up to several orders of magnitude faster than other single-key sketches.
- Hardware-friendly CocoSketch achieves line rate on Tofino switch and FPGA with low resource usage.

7.1 Experimental Setup

Traces: We use two real-world traces in our experiments. (1) *CAIDA:* The traces collected in the Equinix-Chicago monitor from CAIDA in 2018 [76]. We use the trace with a monitoring interval of 60s, which contains around 27M packets. (2) *MAWI:* The traffic traces collected by MAWI [77]. We use the trace with a monitoring interval of 15min, which contains around 13M packets.

Metrics: We evaluate the following six performance metrics. The resource numbers are reported in hardware deployments.

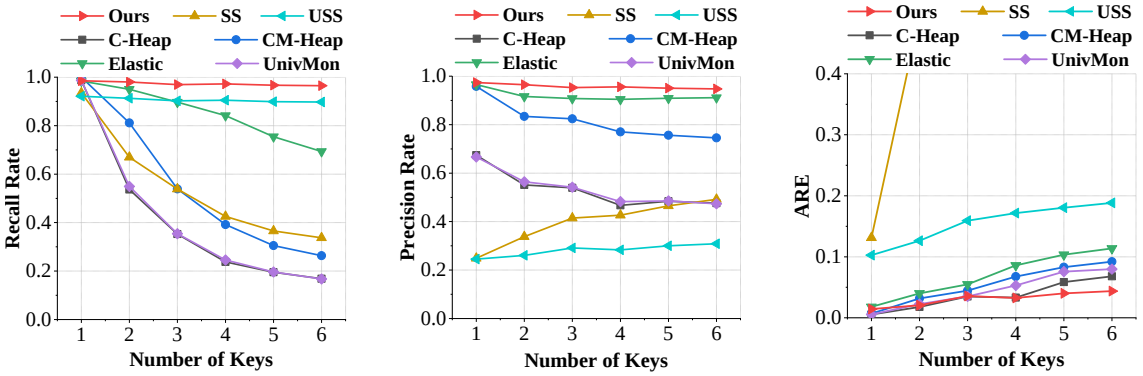
- *Recall Rate (RR):* The ratio of the number of correctly reported flows to the number of correct flows.[2]
- *Precision Rate (PR):* The ratio of the number of correctly reported flows to the number of reported flows.
- *F1 Score:* F1 Score is $\frac{2 \cdot RR \cdot PR}{RR+PR}$.
- *Average Relative Error (ARE):* $\frac{1}{|\Psi|} \sum_{e \in \Psi} \frac{|f(e) - \widehat{f}(e)|}{f(e)}$, where $f(e)$ is the real size, $\widehat{f}(e)$ is the estimated size, and Ψ is the query set.
- *Throughput*: Million packets per second (Mpps). The throughput numbers are the median value among 5 independent trials.
- 95^{th} *percentile CPU cycles:* 95^{th} percentile CPU cycles of per-packet processing.

Setting: By default, we set $d = 2$ in CocoSketch, measure 6 different partial keys (5-tuple, (SrcIP, DstIP) pair, (SrcIP, SrcPort) pair, (DstIP, DstPort) pair, SrcIP, and DstIP) on the CAIDA traces, set the threshold to be 10^{-4} of the total size of traffic (*e.g.*, a heavy hitter is a flow whose size is larger than 10^{-4} of the total size of traffic), and set the total memory at 500KB. We report the average metrics on these keys. For the CocoSketch and USS, we will use one sketch with 500KB memory to measure the full key (5-tuple) and get the result of other keys by aggregation. Other single-key algorithms uses one sketch for each key, as in prior work [33, 39, 40].

7.2 Accuracy

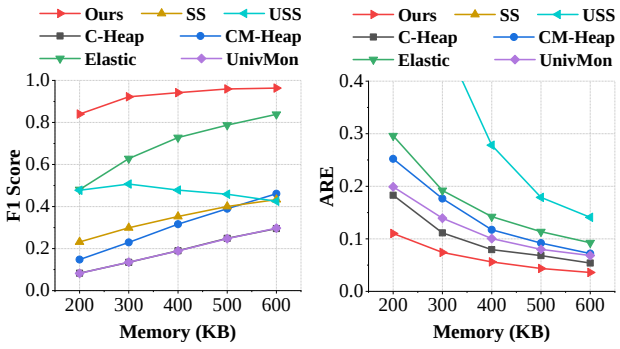
We compare the basic CocoSketch ("Ours" in the figures) with other sketches in three tasks (Heavy Hitters, Heavy Changes, and HHH) with the six partial keys described in §7.1. The baselines include Count sketch [25] with a min-heap (C-Heap), Count-Min sketch [24] with a min-heap (CM-Heap), SpaceSaving (SS) [23], the software version of the Elastic sketch [30], UnivMon [33], and Unbiased SpaceSaving (USS) [53]. In particular, we evaluate USS using an optimized implementation whose update process is enhanced by a hash table and a double-linked list. (Throughput of a naive

[2]For example, when querying heavy hitters, correct flows are the real heavy hitters in the traffic, and correctly reported flows are the real heavy hitters in the reported ones.



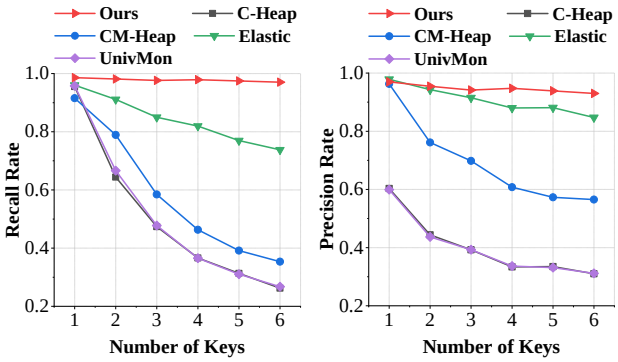
(a) Recall Rate (b) Precision Rate (c) ARE

Figure 8: Performance of heavy hitter detection under different numbers of partial keys.



(a) F1 Score (b) ARE

Figure 9: Performance of heavy hitter detection under different memory constraints.



(a) Recall Rate (b) Precision Rate

Figure 10: Performance of heavy change detection under different numbers of partial keys.

USS implementation is <0.1 Mpps.) We use a hash table to check whether a flow is already tracked in the sketch; and we maintain a double-linked list to rank buckets by their counters so that the minimal bucket can be found quickly. In contrast, CocoSketch does not require extra memory for the hash table or double-linked list.

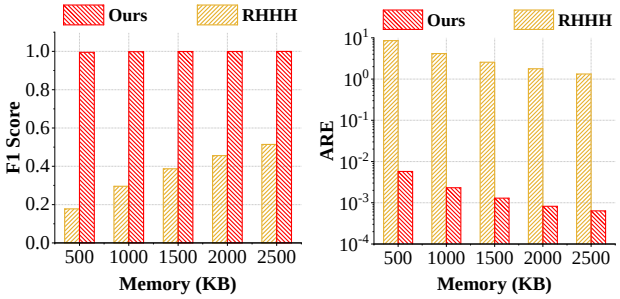
Heavy hitter detection with different numbers of keys (Figures 8(a) -8(c)): CocoSketch achieves the best overall accuracy. Even if only one partial key is measured, CocoSketch performs no worse than other algorithms. When the number of keys grows, CocoSketch always maintains a higher accuracy than the baseline algorithms. Both the recall rate and the precision rate of CocoSketch are above 95% regardless of the number of tracked partial keys. Compared to all baseline algorithms, the ARE of CocoSketch is 9.59× better on average. The precision rate of USS is 64% lower than that of CocoSketch. This is because USS's auxiliary data structures (hash table + a variant of double-linked list) occupy up to 4× memory space.

Heavy hitter detection under different memory configurations (Figures 9(a) -9(b)): CocoSketch also achieves higher accuracy with smaller memory footprints when measuring the 6 keys. With only 300KB memory, the F1 Score of CocoSketch is above 90%, while others are usually below 65%. The ARE of CocoSketch is around 10.43× better than the baseline algorithms. Note that SS is not shown in Figure 9(b) because its ARE is too large (> 0.4).

Heavy change detection with different number of keys (Figures 10(a) -10(b)): Similar to that of heavy hitter detection, with an increasing number of keys, the CocoSketch maintains its high fidelity, while the accuracy of other algorithms drops significantly. Both the recall rate and the precision rate of CocoSketch are higher than 95%, regardless of the number of tracked partial keys. When measuring 6 keys, the recall rate of the CocoSketch is around 71%, 62%, 23%, and 70% higher than that of C-Heap, CM-Heap, Elastic Sketch, and UnivMon, respectively.

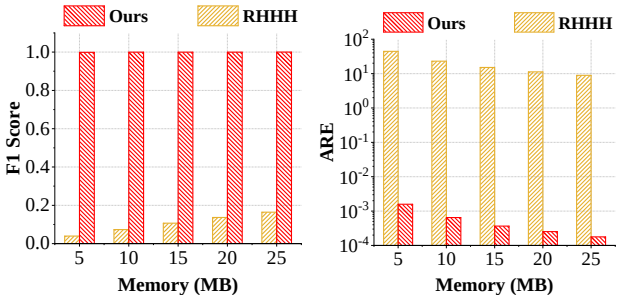
1-d HHH detection with different memory (Figure 11): We consider the source IP hierarchy in bit granularity (32 prefixes + 1 empty key) in 1-d HHH detection. We compare the basic CocoSketch with R-HHH [39] only, because the throughput of other baselines is too low to measure these many keys. With only 500KB memory, the F1 Score of CocoSketch is higher than 99.5%. For R-HHH, even with 2.5MB memory, its F1 Score stays around 50%. The ARE of CocoSketch is about 1902× smaller than that of R-HHH.

2-d HHH detection with different memory (Figure 12): We consider source/destination IP hierarchies in bit granularity (33 × 33 = 1089 keys) in 2-d HHH detection. With 5MB memory, the F1



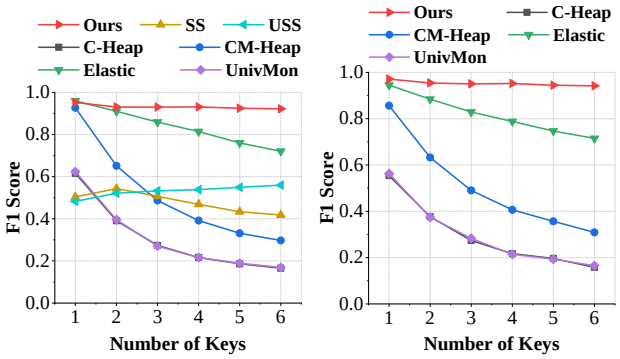
(a) F1 Score (b) ARE

Figure 11: 1-d HHH with different memory constraints.



(a) F1 Score (b) ARE

Figure 12: 2-d HHH with different memory constraints.

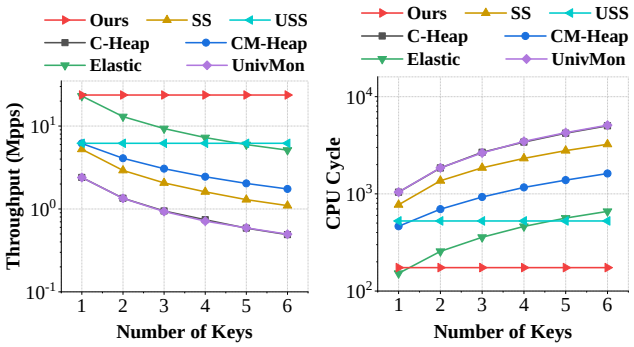


(a) F1 Score on Heavy Hitter (b) F1 Score on Heavy Change

Figure 13: Experiment results on the MAWI dataset.

Score of CocoSketch is higher than 99.8%. We use more than 5MB memory for R-HHH in this experiment, since it cannot work with smaller memory. Even with 25MB memory, its F1 Score is about 16%. The ARE of CocoSketch is about 39843× smaller than that of R-HHH.

Experiments on MAWI traces (Figure 13(a)-13(b)): We also run heavy hitters detection and heavy changes detection on MAWI traces. We find that CocoSketch also maintains high accuracy. When tracking more than two partial keys, CocoSketch achieves over 90% F1 Score and is better than all baselines.



(a) Throughput in CPU (b) 95^{th} percentile CPU cycles

Figure 14: Processing speed in CPU platform.

7.3 Software Platforms

In this section, we will compare the throughput of the basic CocoSketch ("Ours" in the figures) with other baseline algorithms.

Throughput in CPU (Figure 14(a)): Our memory configuration in this experiment is the same as that in the heavy hitter detection (§7.2). We compare single-thread packet processing throughput. The throughput of both CocoSketch and USS are not affected by the number of partial keys measured, while the throughput of other algorithms decreases with the number of partial keys increases. The throughput of CocoSketch is around 23.7 Mpps/core. When measuring 6 partial keys, its throughput is around 27.2 times higher than others.

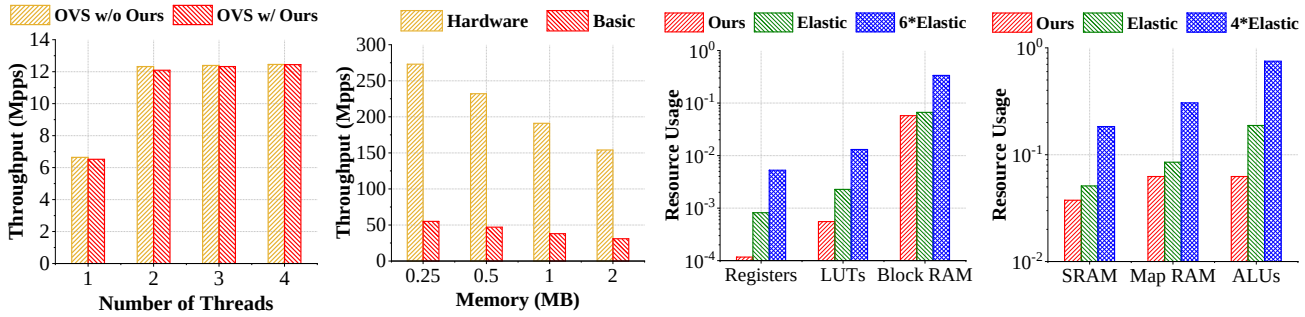
95^{th} percentile CPU cycle (Figure 14(b)): Similar to the throughput in CPU, the CPU cycle of other algorithms increases with the number of partial keys increasing. When measuring 6 partial keys, the number of CPU 95^{th} percentile cycles of CocoSketch is around 18.6, 3.8, 29.2, and 3.0 times smaller than that of SS, Elastic Sketch, UnivMon, and USS, respectively. Although the throughput of USS is also not affected by the number of partial keys measured, its throughput is lower because the auxiliary data structures (hash table + a variant of double-linked list) still need many memory accesses.

Throughput in OVS (Figure 15(a)): We find that the throughput of the CocoSketch increases with the number of threads. With two or more threads, CocoSketch reaches the speed limit of the evaluated 40Gbps NIC. We observe that CocoSketch incurs a small CPU overhead (< 1.8%).

7.4 Hardware Platforms

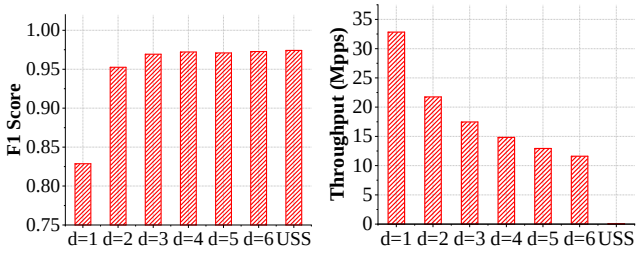
In this section, we compare the hardware-friendly CocoSketch (Ours) with Elastic Sketch [78]. Elastic Sketch has multiple versions designed for different platforms [30], each has a different performance. We configure the memory of evaluated sketches to guarantee 90% F1 Scores in heavy hitters detection (via accuracy experiments).

Throughput in FPGA platform (Figure 15(b)): We show the throughput of both the hardware-friendly CocoSketch and basic CocoSketch with $d = 2$ on the FPGA platform, reported by Vivado [79]. After removing the circular dependencies, hardware-friendly CocoSketch achieves about 5 times higher throughput



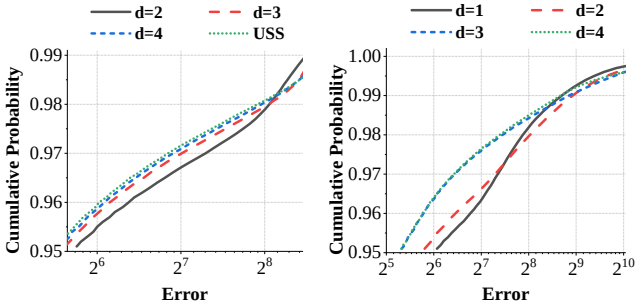
(a) Throughput in OVS Platform (b) Throughput in FPGA Platform (c) Resource Usage in FPGA Platform (d) Resource Usage in P4 Platform

Figure 15: Resource usage and throughput on different platforms



(a) F1 Score (b) Throughput

Figure 16: Varying d's in the basic CocoSketch.



(a) Basic CocoSketch (b) Hardware-friendly CocoSketch

Figure 17: CDF of absolute error under different d values.

than basic CocoSketch. With 2MB memory, the hardware-friendly CocoSketch is expected to achieve 150 Mpps, while the basic CocoSketch only reaches around 30 Mpps with a significantly lower clock frequency. This is because too many operations are performed in one stage in the basic CocoSketch.

Resource usage in FPGA platform (Figure 15(c)): We show the ratio of the resources used by algorithms to the total on-chip resources, which is reported by Vivado [79]. In the figure, "Elastic" indicates the resources used by Elastic Sketch when measuring 1 partial key, and "6*Elastic" indicates the resources used by Elastic Sketch when measuring 6 partial keys. CocoSketch uses fewer resources than that of Elastic Sketch. When measuring 6 partial keys, the *slice registers* that the CocoSketch needs are around 45 times smaller than Elastic Sketch. On FPGA platform, the bottleneck of multiple Elastic Sketches lies in the Block RAM Tile. When measuring 6 partial keys, the Block RAM Tile usage in Elastic Sketch is 34%, while CocoSketch only needs 5.8%.

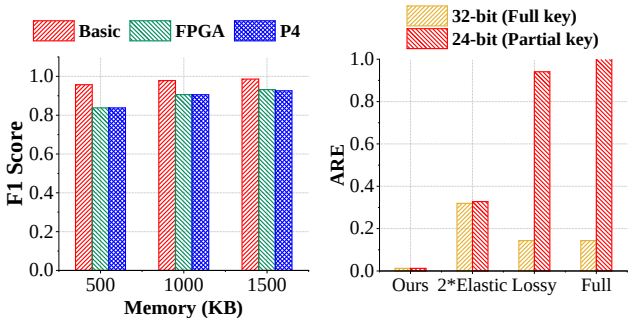
Resource usage in P4 platform (Figure 15(d)): We show the ratio of the resources used by algorithms to the total resources of 12 stages in the Tofino switch. Due to the logic of the algorithm, it is hard to utilize all resources in every stage, *i.e.*, we cannot achieve 100% utilization. In the figure, "4*Elastic" indicates the resources used by Elastic Sketch when measuring 4 keys. We should note that a Tofino switch data plane can implement at most 4 Elastic sketches at the same time due to the resource constraint. We find that CocoSketch uses fewer resources than Elastic Sketch. When measuring 6 partial keys, CocoSketch only needs 6.25% Stateful ALUs and 6.25% Map RAM. On P4 platform, the bottleneck of deploying multiple Elastic sketches lies in the Stateful ALUs. Elastic Sketch needs 18.75% Stateful ALUs in measuring 1 partial key and thus can measure up to 4 partial keys (75% Stateful ALUs and 30.56% Map RAM) in the device.

7.5 Microbenchmark

In this section, we show the performance under different parameter settings and different versions of CocoSketch.

Varying d in the basic CocoSketch (Figures 16 -17(a)): We fix the memory size at 500KB, and use the application of heavy hitters detection to show the performance under different d. We see that as we decrease the value of d from the maximum (the total number of buckets), the F1 Score decreases only marginally: 95.3% ($d = 2$) and 96.9% ($d = 3$). On the other hand, the throughput at $d = 2$ is 23.7 Mpps and at $d = 3$ is 17.5 Mpps, whereas when d is the total number of buckets, the throughput drops to below 0.1 Mpps. Note that CocoSketch becomes USS, when d is the total number of buckets, so the figures use "USS" to denote CocoSketch with the maximum d value.

Varying d in the hardware-friendly CocoSketch (Figure 17(b)): Since in hardware platforms different arrays run independently and in parallel, the value of d in hardware-friendly CocoSketch will not affect the throughput of CocoSketch. To show the performance difference, we fix the memory size at 500KB and show the CDF of error under different d, *i.e.*, for each distinct flow e, we calculate



(a) Different versions (b) Full-key sketch

Figure 18: (a) Different versions of CocoSketch, and (b) CocoSketch vs. full-key sketch baselines

its error $|\hat{f}(e) - f(e)|$ and get the distribution of error. We find that, with a larger d, CocoSketch has a small error with a higher probability, while its worst case is worse than others. Specifically, we find that the probability that the error is smaller than 70 for $d = 1$ is 95.1%, while it is 96.5% for $d = 3$. However, the worst 0.1% error for $d = 1$ is 1873, while it is 2358 for $d = 3$. Such results match the error bound derived in Theorem 3.

Different versions of CocoSketch (Figures 18(a)): We evaluate the heavy hitter detection task to compare the F1 Score of three versions of the CocoSketch: the basic CocoSketch used in software platforms, the hardware-friendly CocoSketch used in FPGA (without approximation on probability calculation), and the hardware-friendly CocoSketch used in P4 (with approximation on probability calculation). We find that the basic CocoSketch performs better than the hardware-friendly CocoSketch, though the accuracy gap between them is less than 10%. With 1MB memory, the hardware-friendly CocoSketch also achieves F1 Score higher than 90%. We also observe that the accuracy gap between the hardware implementations in FPGA and P4 is smaller than 1%, which indicates that our approximate division technique used in the P4 implementation (§6.2) has negligible impact on the accuracy.

Comparison with full-key sketch (Figure 18(b)): To compare CocoSketch to different strawman solutions shown in §2.3, we measure two keys, SrcIP (full key) and its 24-bit prefix (partial key), and show their ARE respectively. We fix the total memory at 6MB and calculate the ARE based on all distinct flows. CocoSketch achieves high accuracy on the full key and partial keys, where the ARE is smaller than 0.02. For "2*Elastic" (where we build one Elastic Sketch for each key), the ARE of both full key and partial key are around 0.3. For "Lossy" (where we recover the partial key only based on the recorded flows in the heavy part), the ARE of full key is around 0.14, while the ARE of partial key is around 0.94. This is because the heavy part of single-key sketch loses too much information to recover the partial key. For "Full" (where we recover the partial key by querying all full keys in the corresponding set), the ARE of full key is around 0.14, while the ARE of partial key larger than 1. This is because, in the single-key sketches, the error increases as they aggregate many full keys. Therefore, although "Lossy" and "Full" achieve desirable accuracy on the full key, neither achieves high accuracy on the arbitrary partial keys.

8 RELATED WORK

Sketch-based telemetry: In addition to the efforts described in §2, a number of techniques have been proposed to improve the fidelity, generality, performance, etc., of sketches [30, 31, 33, 38, 59, 80]. For example, UnivMon [33] introduces a general sketch to estimate a range of traffic statistics. WavingSketch [38] extends the Count Sketch [25] to find persistent items and super-spreaders. However, these sketches still focus on single-key measurements. While these sketch techniques are fundamentally limited in measuring arbitrary partial keys, some of them may bring additional benefits to CocoSketch, such as the sampling approach used in NitroSketch [31] can further improve the throughput, and the merge technique used in Elastic Sketch [30] can adapt to dynamic workloads with varying bandwidths. We leave this for future work.

Telemetry resource management: To deploy multiple sketches in a network, we need efficient resource management, and there are some recent efforts in this space [40, 65, 81, 82]. For example, Trumpet [65] uses event triggers at end-hosts to detect some telemetry events within high timeliness requirements. Their solutions address a different set of telemetry events than arbitrary partial key queries, which are orthogonal to our proposal. Furthermore, DREAM [81] and SCREAM [40] dynamically allocate resources for different measurement tasks to achieve a uniform accuracy target. Using their solutions in the arbitrary partial key problem still requires a single sketch per possible key, leading to the same resource inefficiency described in §2.3.

Other measurement tasks over multiple flow keys: Recent work BeauCoup [32] aims to support distinct counting queries over multiple keys simultaneously. Essentially, they need to maintain a separate data structure for each key, similar to single-key sketches. Furthermore, they cannot recover the partial key information from a full key. We leave the exploration of extending CocoSketch to support distinct counting for future work.

9 CONCLUSIONS

Sketching algorithms are extensively studied in network measurements. However, sketching over multiple flow keys is far from ideal for serving as a viable solution for software and hardware network platforms. In this paper, we present CocoSketch, a sketch-based measurement approach that accurately answers arbitrary partial key queries. Leveraging stochastic variance minimization, the data plane algorithms in CocoSketch run at high speed regardless of the number of partial keys measured, significantly outperforming existing sketches in terms of CPU performance and memory efficiency. By further removing circular dependencies, CocoSketch becomes hardware-friendly for programmable switches and FPGA. Our experiments demonstrate the performance of CocoSketch by comparing it with a variety of sketches under real-world traces. We have open-sourced code of CocoSketch and other baseline algorithms on GitHub [75].

Acknowledgments: We thank the anonymous reviewers and our shepherd Kate Lin for their valuable suggestions. This work was supported in part by National Natural Science Foundation of China (NSFC) (No. U20A20179). Junchen Jiang is supported by a Google Faculty Research Award. Tong Yang is the corresponding author.

Ethics: This work does not raise any ethical issues.

REFERENCES

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

[2] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *NSDI 2019*. USENIX Association, 2019.

[3] Anja Feldmann, Albert Greenberg, and et al. Deriving traffic demands for operational ip networks: Methodology and experience. In *ACM SIGCOMM*, 2000.

[4] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review*. ACM, 2015.

[5] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Simon: A simple and scalable method for sensing, inference and measurement in data center networks. In *NSDI 2019*, pages 549–564, 2019.

[6] Sam Burnett, Lily Chen, Douglas A Creager, Misha Efimov, Ilya Grigorik, Ben Jones, Harsha V Madhyastha, Pavlos Papageorge, Brian Rogan, Charles Stahl, et al. Network error logging: Client-side measurement of end-to-end web service reliability. In *NSDI 2020*, pages 985–998, 2020.

[7] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[8] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM 2011*. ACM, 2011.

[9] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: query-driven streaming network telemetry. In *SIGCOMM 2018*. ACM, 2018.

[10] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. csamp: A system for network-wide flow monitoring. In *NSDI 2008*. USENIX Association, 2008.

[11] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. 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*, 2020.

[12] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[13] 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 *SIGCOMM 2017*. ACM, 2017.

[14] 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 *17th USENIX Conference on File and Storage Technologies, FAST 2019*. USENIX Association, 2019.

[15] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[16] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *NSDI 17*, 2017.

[17] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *NSDI 2018*. USENIX Association, 2018.

[18] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. 2019.

[19] William M Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *NSDI 20*, 2020.

[20] Xin Li, Fang Bian, Mark Crovella, Christophe Diot, Ramesh Govindan, Gianluca Iannaccone, and Anukool Lakhina. Detection and identification of network anomalies using sketch subspaces. In *IMC 2006*. ACM, 2006.

[21] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick G. Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *IMC 2004*. ACM, 2004.

[22] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *ACM IMC*, 2004.

[23] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In Thomas Eiter and Leonid Libkin, editors, *ICDT 2005*, Lecture Notes in Computer Science. Springer, 2005.

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

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

[26] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[27] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *SIGMETRICS 2004*. ACM, 2004.

[28] Robert T. Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In Alfio Lombardo and James F. Kurose, editors, *IMC 2004*. ACM, 2004.

[29] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI 2016*. USENIX Association, 2016.

[30] 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*. ACM, 2018.

[31] 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.

[32] Xiaoqi Chen, Shir Landau Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *SIGCOMM '20*. ACM, 2020.

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

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

[35] 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*. ACM, 2017.

[36] Yinda Zhang, Jinyang Li, Yutian Lei, Tong Yang, Zhetao Li, Gong Zhang, and Bin Cui. On-off sketch: A fast and accurate sketch on persistence. *Proc. VLDB Endow.*, 2021.

[37] Xiangyang Gou, Long He, Yinda Zhang, Ke Wang, Xilai Liu, Tong Yang, Yi Wang, and Bin Cui. Sliding sketches: A framework using time zones for data stream processing in sliding windows. In *KDD '20*. ACM, 2020.

[38] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *KDD '20*, pages 1574–1584. ACM, 2020.

[39] Ran Ben-Basat, Gil Einziger, Roy Friedman, Marcelo Caggiani Luizelli, and Erez Waisbard. Constant time updates in hierarchical heavy hitters. In *SIGCOMM 2017*. ACM, 2017.

[40] 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.

[41] Omid Alipourfard, Masoud Moshref, and Minlan Yu. Re-evaluating measurement algorithms in software. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, 2015.

[42] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford-Chen, and Nicholas Weaver. Inside the slammer worm. *IEEE Secur. Priv.*, 2003.

[43] Vyas Sekar, Nick G Duffield, Oliver Spatscheck, Jacobus E van der Merwe, and Hui Zhang. Lads: Large-scale automated ddos detection system. In *USENIX Annual Technical Conference, General Track*, 2006.

[44] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *NSDI 18*, 2018.

[45] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, 2011.

[46] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI 14*, 2014.

[47] Philipp Richter, Ramakrishna Padmanabhan, Neil Spring, Arthur Berger, and David Clark. Advancing the art of internet edge outage detection. In *Proceedings of the Internet Measurement Conference 2018*, pages 350–363, 2018.

[48] Thomas Holterbach, Emile Aben, Cristel Pelsser, Randy Bush, and Laurent Vanbever. Measurement vantage point selection using a similarity metric. In *Proceedings of the Applied Networking Research Workshop*, pages 1–3, 2017.

[49] Barefoot tofino: World's fastest p4-programmable ethernet switch asics. https://barefootnetworks.com/products/brief-tofino/.

[50] 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*. USENIX Association, 2015.

[51] Alveo u280 data center accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.
[52] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In Nick Feamster and Jeffrey C. Mogul, editors, *NSDI 2013*. USENIX Association, 2013.
[53] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *SIGMOD 2018*. ACM, 2018.
[54] Nick G. Duffield, Carsten Lund, and Mikkel Thorup. Priority sampling for estimation of arbitrary subset sums. *J. ACM*, 2007.
[55] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, and Hui Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 165–176, 2006.
[56] Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. https://tools.ietf.org/html/rfc3176.
[57] Pavlos Nikolopoulos, Christos Pappas, Katerina Argyraki, and Adrian Perrig. Retroactive packet sampling for traffic receipts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2019.
[58] Sajad Shirali-Shahreza and Yashar Ganjali. Flexam: flexible sampling extension for monitoring and security applications in openflow. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.
[59] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *SOSR 2017*. ACM, 2017.
[60] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, 2003.
[61] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, 2003.
[62] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun (Jim) Xu, and Hui Zhang. Data streaming algorithms for estimating entropy of network traffic. In *SIGMETRICS 2006*, pages 145–156. ACM, 2006.
[63] Arno Wagner and Bernhard Plattner. Entropy based worm and anomaly detection in fast ip networks. In *14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*, 2005.
[64] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining anomalies using traffic feature distributions. In *ACM SIGCOMM*, 2005.
[65] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *SIGCOMM 2016*, pages 129–143. ACM, 2016.
[66] Masoud Moshref, Minlan Yu, Abhishek B. Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *NSDI 2013*, pages 157–170. USENIX Association, 2013.
[67] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM 2015*, pages 393–406. ACM, 2015.
[68] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM 2015*, pages 1–14. ACM, 2015.
[69] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. 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*, 2020.
[70] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. Sketchlib: Enabling efficient sketch-based monitoring on programmable switches. In *NSDI*, 2022.
[71] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *USENIX Security*, 2021.
[72] Mu He, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. Toward consistent state management of adaptive programmable networks based on p4. In *Proceedings of the ACM SIGCOMM 2019 Workshop on Networking for Emerging Applications and Technologies*, pages 29–35, 2019.
[73] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *SIGCOMM 2019*. ACM, 2019.
[74] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.
[75] Source code related to CocoSketch. https://github.com/yindazhang/CocoSketch.
[76] The caida anonymized 2016 internet traces. http://www.caida.org/data/overview/.
[77] MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/.
[78] Source code related to elastic sketch. https://github.com/BlockLiu/ElasticSketchCode.
[79] Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html.
[80] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. *SIAM APOCS*, 2020.
[81] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. DREAM: dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM 2014*, pages 419–430. ACM, 2014.
[82] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. Heterosketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *NSDI*, 2022.
[83] Hash website. http://burtleburtle.net/bob/hash/evahash.html.

A PROOFS

A.1 Stochastic Variance Minimization

In this section, we prove Theorem 1 and 2 shown in §5.1.

Theorem 1. *The solution to optimize Eq. (2) is*

$$(e', f') = \begin{cases} (e_i, f_j + w), & \text{w.p. } \dfrac{w}{f_j + w} \\ (e_j, f_j + w), & \text{w.p. } \dfrac{f_j}{f_j + w} \end{cases}$$

Proof. Remind that the incoming packet is (e_i, w), and it is mapped to the bucket recording key value pair (e_j, f_j). Then we should update the mapped bucket to (e', f') to optimize Eq. (2). Note that we only change the estimated size of full key e_i and e_j, so the variance increments of all other full keys are 0. If a full key is not recorded, its estimated size is 0. Otherwise, its estimated size is the corresponding value in the bucket. Obviously, if $e_i = e_j$, we can directly update the mapped bucket to $(e_j, f_j + w)$, and there is no increment of variance. If $e_i \neq e_j$, to keep unbiasedness, suppose that we will set $(e', f') = (e_i, w/p)$ with probability p, and set $(e', f') = (e_j, f_j/(1-p))$ with probability $1-p$. The increment of variance is that

$$\begin{aligned} \sum_e \Delta \left(f(e) - \widehat{f}(e)\right)^2 &= p \cdot \left(\left(\frac{w}{p} - w\right)^2 + f_j^2\right) \\ &\quad + (1-p) \cdot \left(w^2 + \left(\frac{f_j}{1-p} - f_j\right)^2\right) \\ &= \frac{w^2}{p} - w^2 + \frac{f_j^2}{1-p} - f_j^2 \end{aligned}$$

Then, we can get that we achieve the minimum when $p = w/(f_j+w)$. Based on the formula of p, we can get the result Eq. (3). □

Theorem 2. *The minimum increment of variance sum to update the bucket (e_j, f_j) is*

$$\sum_e \Delta \left(f(e) - \widehat{f}(e)\right)^2 = \begin{cases} 2wf_j, & e_i \neq e_j \\ 0, & e_i = e_j \end{cases}$$

Proof. Based on the proof of Theorem 1, we can get that if $e_i = e_j$, the variance increment is 0. If $e_i \neq e_j$,

$$\begin{aligned} \sum_e \Delta \left(f(e) - \widehat{f}(e)\right)^2 &= \frac{w^2}{p} - w^2 + \frac{f_j^2}{1-p} - f_j^2 \\ &= 2wf_j \end{aligned}$$

□

A.2 Error Bound

In this section, we first prove the unbiasedness of both basic CocoSketch and hardware-friendly CocoSketch. Then, we prove the error bound of Theorem 3 shown in §5.2.

Lemma 3. *For any flow e of any key $k \prec k_F$, in the basic CocoSketch,*

$$\mathbb{E}\left[\widehat{f}(e)\right] = f(e)$$

Proof. We first prove that, for any flow e of full key k_F, in the basic CocoSketch, $\mathbb{E}\left[\widehat{f}(e)\right] = f(e)$. Let $\widehat{f}^t(e)$ be the estimated size of e before t^{th} insertion. Suppose that the incoming packet is (e_i, w) for the t^{th} insertion. We prove the unbiasedness by showing that the expected increment to $\widehat{f}^t(e)$ is w if $e = e_i$ and 0 otherwise.

If $e = e_i$, there are two cases. *Case 1:* If e is recorded, the estimated size will be increased by w. *Case 2:* If e is not recorded, suppose that the mapped bucket whose value is the smallest is in the k^{th} array. The expected increment is

$$(B_k[h_k(e)].V + w) \cdot \frac{w}{(B_k[h_k(e)].V + w)} = w$$

Therefore, the expected increment to $\widehat{f}^t(e)$ is w if $e = e_i$.

If $e \neq e_i$, there are two cases. *Case 1:* If e is recorded and the corresponding bucket will be updated, the expected increment is

$$\left(\widehat{f}^t(e) + w\right) \cdot \frac{\widehat{f}^t(e)}{\left(\widehat{f}^t(e) + w\right)} - \widehat{f}^t(e) = 0$$

Case 2: Otherwise, the estimated size does not change. Therefore, the expected increment to $\widehat{f}^t(e)$ is 0 if $e \neq e_i$.

As a result, the basic CocoSketch achieves unbiasedness for the full key. Then, for any flow e of any key $k \prec k_F$, we have

$$\mathbb{E}\left[\widehat{f}(e)\right] = \mathbb{E}\left[\sum_{k(a)=e} \widehat{f}(a)\right] = \sum_{k(a)=e} f(a) = f(e)$$

□

Let $\widehat{f_i}(e)$ be the estimated size of flow e in the i^{th} array of the hardware-friendly CocoSketch.

Lemma 4. *For any flow e of any key $k \prec k_F$, in the hardware-friendly CocoSketch,*

$$\mathbb{E}\left[\widehat{f_i}(e)\right] = f(e)$$

Proof. Note that in a bucket, the probability of occupying the bucket is proportional to the size of each flow. Therefore, after the insertion process,

$$\mathbb{P}\left[B_i[h_i(e)].K = e\right] = \frac{f(e)}{B_i[h_i(e)].V}$$

Based on the probability, we can get the expectation of the estimated size in each array.

$$\mathbb{E}\left[\widehat{f_i}(e)\right] = \frac{f(e)}{B_i[h_i(e)].V} \cdot B_i[h_i(e)].V = f(e)$$

□

Lemma 5. *For any flow e of any key $k \prec k_F$, in the hardware-friendly CocoSketch,*

$$\mathrm{Var}\left[\widehat{f_i}(e)\right] = \frac{f(e) \cdot f(\overline{e})}{l}$$

Proof. In the i^{th} array, let $I_{i,j}(e)$ be 1 if $k(B_i[j].K) = e$ and 0 otherwise. We define

$$C_{i,j}(e) = \sum_{\substack{k(a)=e \\ h_i(a)=j}} f(a), \quad \widehat{C_{i,j}}(e) = I_{i,j}(e) \cdot B_i[j].V$$

We have

$$\mathrm{Var}\left[\widehat{C_{i,j}}(e)\right] = C_{i,j}(e) \cdot \mathbb{E}\left[B_i[j].V - C_{i,j}(e)\right] = C_{i,j}(e) \cdot \frac{f(\overline{e})}{l}$$

$$\mathrm{Cov}\left[\widehat{C_{i,j}}(e), \widehat{C_{i,k}}(e)\right] = 0, j \neq k$$

Then, we can get the variance for the i^{th} array is that

$$\begin{aligned}\mathrm{Var}\left[\widehat{f_i}(e)\right] &= \mathrm{Var}\left[\sum_{j=1}^{l} \widehat{C_{i,j}}(e)\right] \\ &= \sum_{j=1}^{l} C_{i,j}(e) \cdot \frac{f(\overline{e})}{l} \\ &= \frac{f(e) \cdot f(\overline{e})}{l}\end{aligned}$$

□

Theorem 3. *Let $l = 3 \cdot \epsilon^{-2}$ and $d = O(\log \delta^{-1})$. For any flow e of arbitrary partial key $k_P \prec k_F$,*

$$\mathbb{P}\left[R(e) \geqslant \epsilon \cdot \sqrt{\frac{f(\overline{e})}{f(e)}}\right] \leqslant \delta$$

Proof. Let $R_i(e)$ be the relative error of flow e based on its estimated size $\widehat{f_i}(e)$ in the i^{th} array of the hardware-friendly CocoSketch. According to the variance and Chebyshev's inequality, we have

$$\begin{aligned}\mathbb{P}\left[R_i(e) \geqslant \epsilon \cdot \sqrt{\frac{f(\overline{e})}{f(e)}}\right] &= \mathbb{P}\left[\left|\widehat{f_i}(e) - f(e)\right| \geqslant \epsilon \cdot \sqrt{f(\overline{e}) \cdot f(e)}\right] \\ &\leqslant \frac{\mathrm{Var}\left[\widehat{f_i}(e)\right]}{\epsilon^2 \cdot f(\overline{e}) \cdot f(e)} \\ &= \epsilon^{-2} \cdot l^{-1}\end{aligned}$$

By setting $l = 3 \cdot \epsilon^{-2}$, we have

$$\mathbb{P}\left[R_i(e) \geqslant \epsilon \cdot \sqrt{\frac{f(\overline{e})}{f(e)}}\right] \leqslant \frac{1}{3}$$

Because the final estimated size is the median result, if the $R(e) \geqslant \epsilon \cdot \sqrt{f(\overline{e})/f(e)}$, at least $d/2$ $R_i(e)$ must be larger than $\epsilon \cdot \sqrt{f(\overline{e})/f(e)}$. Based on the Chernoff's inequality, setting $d = O(\log \delta^{-1})$ can make such probability reduce to δ. □

Let $M = d \cdot l$. Then we analyze M needed for different d to achieve given ϵ and δ. Based on the proof above, we have

$$\delta = \left(\frac{d}{\epsilon^2 \cdot M}\right)^{O(d)}$$

If two configuration M_1, d_1 and M_2, d_2 achieve the same error bound, we have

$$M_2 \approx \frac{d_2 \cdot \delta^{-\frac{1}{d_2}}}{d_1 \cdot \delta^{-\frac{1}{d_1}}} \cdot M_1$$

We can get that, $d \approx \ln \delta^{-1}$ can achieve the smallest M for given δ. For other d, we need around $\frac{d \cdot (1/\delta)^{1/d}}{e \cdot \ln(1/\delta)}$ times more memory, where e is the Euler's number.

A.3 Recall Rate

In this section, we prove Theorem 4 shown in §5.3.

Theorem 4. *For any flow e of full key k_F,*

$$\mathbb{P}\left[Z(e) = 1\right] \geqslant 1 - \left(1 + l \cdot \frac{f(e)}{f(\overline{e})}\right)^{-d}$$

Proof. In the i^{th} array, let $Z_i(e)$ be a 0-1 function. $Z_i(e) = 1$ if and only if e is recorded in the i^{th} array of the CocoSketch. According to the Jensen's inequality, we have

$$\begin{aligned}\mathbb{P}\left[Z_i(e) = 1\right] &= \sum_m \mathbb{P}\left[B_i[h_i(e)].V = m\right] \frac{f(e)}{B_i[h_i(e)].V} \\ &\geqslant \frac{f(e)}{\mathbb{E}\left[B_i[h_i(e)].V\right]} = \frac{l \cdot f(e)}{f(\overline{e}) + l \cdot f(e)}\end{aligned}$$

Because the hash functions are independent,

$$\begin{aligned}\mathbb{P}\left[Z(e) = 1\right] &= 1 - \prod_{i=1}^{d} \left(1 - \mathbb{P}\left[Z_i(e) = 1\right]\right) \\ &\geqslant 1 - \left(1 - \frac{l \cdot f(e)}{f(\overline{e}) + l \cdot f(e)}\right)^{d} \\ &= 1 - \left(1 + l \cdot \frac{f(e)}{f(\overline{e})}\right)^{-d}\end{aligned}$$

□

B BASIC COCOSKETCH IMPLEMENTATION

CPU Implementation: We implement the basic CocoSketch (§4.1) using C++. The hash functions are implemented using the 32-bit Bob Hash [83] with different hash seeds. We implement and evaluate them on a machine with one 4-core processor (8 threads, Intel(R) Core(TM) i5-8259U CPU @ 2.30GHz) and 16 GB DRAM memory. The processor has 64KB L1 cache, 256KB L2 cache for each core, and 6MB L3 cache shared by all cores.

OVS Implementation: We implement CocoSketch on OVS v2.12.1 with DPDK 18.11.10. We use ring buffers as the shared memory to connect the *datapath* in OVS and the measurement process of the CocoSketch. When a packet enters the datapath, its packet header will be written into ring buffers. The measurement process continuously reads packet header information from ring buffers by polling. Our testbed has two servers that are directly connected. One server runs OVS, and another server generates high-speed TCP traffic using pktgen-dpdk (version 3.7.2). Each server is equipped with a Mellanox ConnectX-3 40G NIC, an Intel Core i5-8400@2.80GHz CPU, and 16GB DRAM. To accelerate the process, we assign multiple (*e.g.*, 4) Rx queues for the DPDK receive port in OVS. Different Rx queues are pinned to different cores and are polled by different Poll Mode Driver threads.