CocoSketch: High-Performance Sketch-Based Measurement Over Arbitrary Partial Key Query

Ruijie Miao, Yinda Zhang, Zihao Zheng, Ruixin Wang, Ruwen Zhang, Tong Yang, *Member, IEEE*, Zaoxing Liu, and Junchen Jiang

***Abstract*— Sketch-based measurement has emerged as a promising solutions due to its high accuracy and resource efficiency. Prior sketches focus on measuring single flow keys and cannot support measurement on multiple keys. This work takes a significant step towards supporting *arbitrary partial key queries*, which aims to provide information for any key in the predefined range of possible flow keys. The designed system, *CocoSketch*, casts arbitrary partial key queries to the subset sum estimation problem and makes the theoretical tools for subset sum estimation practical. CocoSketch utilizes 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. This paper extends the conference version by discussing how CocoSketch adapts to new measurement requirements, including: (1) collecting the exact information of specified flow keys, and (2) distributed measurement. CocoSketch is implemented on five popular platforms (CPU, Open vSwitch, Redis, P4, and FPGA). Experiment results show that compared to baselines that use traditional single-key sketches, CocoSketch improves average packet processing throughput by $27.2\times$ and accuracy by $10.4\times$ when measuring six flow keys.**

***Index Terms*— Sketch, arbitrary partial key query, P4, FPGA.**

I. Introduction

NETWORK monitoring and measurement have been critical to various network management tasks, such as traffic engineering [2], [3], accounting [4], [5], [6], load balancing [7], [8], [9], flow scheduling [10], [11], and anomaly detection [12], [13], [14]. These tasks often require timely and accurate estimates of the network flow metrics, *e.g.*, heavy hitters [15], [16], [17], [18], flow size distribution [19], or heavy changes [20], [21]. 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 [22], [23].

At a high level, existing sketches 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). While recent efforts on single-key sketches have made significant progress [22], [24], [25], [26], [27], it is impractical to use these sketches to measure *multiple* flow keys simultaneously. First, existing sketches [24], [28] 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 [23], [29]. Second, they require operators to pre-define the set of flow keys before the measurement starts, which is impractical in diverse scenarios such as network diagnosis and security [14], [30], [31], [32].

This paper defines 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 following 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), Besides, it would be better to be *extensible*, which will make the measurement system adapt to new measurement requirements easily.

Unfortunately, existing solutions that might support arbitrary partial key queries fall short on at least one requirement. R-HHH [28] reduces the overhead of updating 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. An alternative solution is to use a single-key sketch to measure full-key flow sizes and recover partial-key flow sizes

Manuscript received 8 November 2022; revised 21 February 2023; accepted 6 March 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Caesar. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B0101390001 and in part by the National Natural Science Foundation of China (NSFC) under Grant U20A20179. The work of Junchen Jiang was supported by the Google Faculty Research Award. The preliminary version of this paper has been published in ACM Special Interest Group on Data Communication (SIGCOMM), 2021 [DOI: 10.1145/3452296.3472892]. *(Ruijie Miao, Yinda Zhang, and Zihao Zheng contributed equally to this work.) (Corresponding author: Tong Yang.)*

Ruijie Miao, Zihao Zheng, Ruixin Wang, and Ruwen Zhang are with the National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing 100871, China (e-mail: miaoruijie@pku.edu.cn).

Yinda Zhang is with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104 USA (e-mail: hgdkgszyd@gmail.com).

Tong Yang is with the National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing 100871, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: yangtongemail@gmail.com).

Zaoxing Liu is with the Electrical and Computer Engineering Department, and the Department of Computer Science, Boston University, Boston, MA 02215 USA (e-mail: zaoxing@bu.edu).

Junchen Jiang is with the Department of Computer Science, The University of Chicago, Chicago, IL 60637 USA (e-mail: junchenj@uchicago.edu).

Digital Object Identifier 10.1109/TNET.2023.3257226

by aggregating full-key flows. However, prior work [33] has shown that this approach might have large estimation errors. Unbiased SpaceSaving (USS), a recent technique for subset sum estimation [34], can be applied for arbitrary partial key query. 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. 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 (§VIII-E), and it cannot be supported by some resource-constraint hardware [35].

This work presents **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 (*e.g.*, Open vSwitch [36], Redis [37], PISA [38], and FPGA [39]), and can adapt to new measurement requirements easily.

CocoSketch share the theoretical basis with USS, and the challenge of CocoSketch lies in how to practically apply the theory of subset sum estimation to the partial key query problem. CocoSketch utilizes two main techniques. (a) Inspired by USS, the technique, *stochastic variance minimization* is introduced to CocoSketch. 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. Theoretical analysis in §VI shows that, like USS, estimation of CocoSketch 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 address the problem, CocoSketch further removes circular dependencies by parallelizing the operations of stochastic variance minimization in a way that incurs only minor increases in estimation errors (less than 10% drop in F1 score).

Due to the simplicity of the data plane design, CocoSketch can easily adapt to new measurement requirements. This paper discusses two common requirements. First, in addition to arbitrary partial key query, a measurement system may be required to track exact information of specified full flow keys for debugging or performance analysis [40], [41], [42]. Second, in large-scale data centers the network measurement system often consists of multiple measurement nodes, and therefore should support distributed measurement. For the former, a variant of CocoSketch named *D-CocoSketch* is presented, which can track exact information for specified full keys while still maintaining high accuracy for arbitrary partial key query. For the latter, this work shows that CocoSketch can be applied in a distributed manner and achieve arbitrary partial key query for the whole network topology.

II. Background and Motivation

A. Sketches for Network Measurement

Sketching algorithms [16], [17], [21], [22], [24], [25] process data streams to estimate various statistics in an online fashion. With provable and tunable accuracy-memory tradeoffs, sketches can fit in network devices with diverse resource constraints. Typically, research in sketches follows 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. In network measurement, single-key sketches are widely used in diverse applications [20], [21], [35], [43], [44], [45], [46], [47], [48].

B. Arbitrary Partial Key Problem

In contrast to the single-key paradigm, this paper defines 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. The formal definition is presented 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 $g(\cdot) : k_F \rightarrow k_P$, and for any flow $e \in k_P$ defined on key k_P, there is*

$$f(e) = \sum_{e' \in k_F, g(e')=e} f(e')$$

where $f(e)$ is a statistic (e.g., size) of flow e.

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 $k_P \prec k_F$.

This paper assumes f is a flow size function. For example, if the full key is SrcIP, any of its prefix (*e.g.*, /24 prefix) will be a partial key. The size of a flow e for a partial key of /24 SrcIP prefix (*e.g.*, $(56.49.82.*)$) equals the sum of the size of full-key flows in the prefix (*e.g.*, $\{(56.49.82.0), \ldots, (56.49.82.255)\}$). The problem of arbitrary partial key query enables a more flexible way of querying flow statistics without specifying which keys to query beforehand.

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 [49], applications, such as guiding rule placement [50], coflow scheduling [51], and multi-key rate limiting [52], require estimation results over many different keys. In security and diagnosis scenarios, 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 [14].

TABLE I
OPTIMIZATION TARGETS FOR DIFFERENT SOLUTIONS

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

C. Existing Solutions and Limitations

One single-key sketch per key: One strawman to realize arbitrary partial key queries is by creating one single-key sketch (*e.g.*, [24]) 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 [28] 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 [28]. In hardware switches such as Barefoot Tofino [38], its resource usage will grow linearly with more sketches, so this approach cannot support more than a handful of keys.

Besides, such solutions requires to pre-define the set of flow keys before measurement starts. A single measurement task, DDoS detection, may track large flows defined on tens of flow keys, including SrcIP/DstIP, the 5-tuple, and arbitrary prefixes of them [14]. Therefore, it is almost impossible to cover all partial keys that are required by different tasks. Even if it could, it would lead to unacceptable errors given limited memory in hardware switches.

Full-key sketch with post recovery: An alternative solution is to 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) The other way is, instead of aggregating the estimates of all full-key flows, aggregating only the full-key flows that are explicitly logged in the sketch. Prior analysis [33], however, suggests that aggregating such a subset of flows can yield high estimation bias and variance, which is confirmed by evaluations in §VIII-E.

Subset sum estimation: A more promising approach is to cast the arbitrary partial key query problem to the *subset sum estimation* problem. As summarized in Table I, 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 the design 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) [33], is impractical for network measurement. As will be elaborated in §III-A, 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 [38].

D. New Measurement Requirements

Exact information of specified full flow keys. For network measurement, it is often necessary to provide operators with exact information for some full flow keys, which can be used to examine the performance metrics and locate problems of the upper-layer applications. For instance, with the knowledge of the packet number sent and received by one application, the operators can decide exactly how much bandwidth is consumed and whether the network transmission becomes the bottleneck. Existing widely-used network measurement solutions, such as NetFlow [42] and sFlow [41], provides two forms for measuring exact information: (1) users specify a set of flow match rules (*e.g.*, SrcIP = 10.0.0.1, DstIP = 10.0.0.*), and the exact information of matched flows are measured; (2) users specify a sample rate, and exact information of sampled flows are measured.

Distributed measurement. In large-scale data center network, it is impractical to build a measurement system with only one measurement node. Directly selecting one server or one forwarding device as the measurement node cannot cover network-wide flows. Figure 2 shows a Fat-tree topology that is widely used in data centers. Deploying CocoSketch on any single node cannot cover flow A and B at the same time, as they have different forwarding paths and pass different switches. Mirroring all packets to one measurement node is also impractical, as a single node cannot process all packets in the data center given the huge volume of the network traffic. As reported in [40], the volume of network traffic has grown to hundreds of Terabytes per second and is still growing rapidly. Therefore, a measurement system usually deploys multiple measurement nodes, which work collectively to provide consistent measurement results with full coverage.

III. OVERVIEW

In this section, an overview of CocoSketch, and its two key ideas: stochastic variance minimization (§III-A) and removal of circular dependencies (§III-B), is presented.

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 e of the flow and use d hash functions to map e to d buckets, each from an array of l buckets.

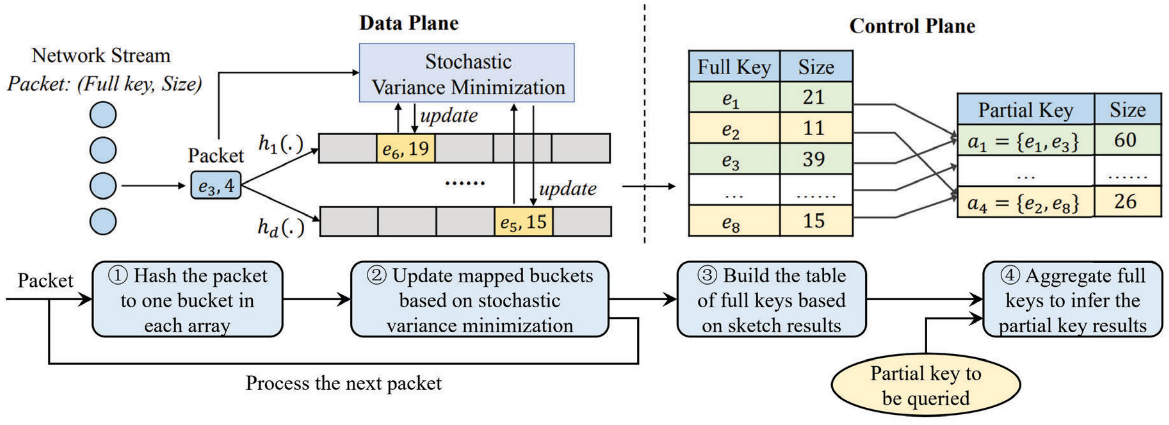

Data Plane
Control Plane
Network Stream
Packet: (Full key, Size)
Stochastic Variance Minimization
update
Packet
$h_1(.)$
$h_d(.)$
$e_3, 4$
$e_6, 19$
$e_5, 15$
Full Key
Size
e_1 21
e_2 11
e_3 39
e_8 15
Partial Key
Size
$a_1 = \{e_1, e_3\}$ 60
$a_4 = \{e_2, e_8\}$ 26
① Hash the packet to one bucket in each array
② Update mapped buckets based on stochastic variance minimization
③ Build the table of full keys based on sketch results
④ Aggregate full keys to infer the partial key results
Process the next packet
Partial key to be queried

Fig. 1. Overview of the CocoSketch architecture, which composes of data plane and control plane.

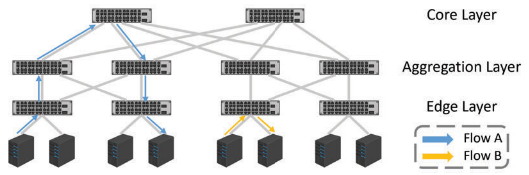


Fig. 2. An example of fat-tree topology with 2 core switches, 4 aggregation switches, 4 edge switches and 8 servers.

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 the *recorded* full-key flows to infer the size of flows defined by the queried partial key k_P.

The two technical ideas are discussed as follows.

A. Stochastic Variance Minimization

Variance minimization in Unbiased SpaceSaving: Before describing the technique in CocoSketch, it is necessary to first explain how Unbiased SpaceSaving (USS) [33] 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 is slow and has difficulty to be deployed on hardware. How to reduce the update cost of USS while still maintaining the high accuracy guarantees?

Reducing update delay: 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. 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.

Preserving estimation accuracy: Updating the bucket among only d buckets per packet is still accurate under heavy-tailed flow distribution, which is common in real world. The reasons are twofold.

- First, for a large flow, the mapped counter 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.
- Second, for small flows, like USS, CocoSketch spreads out the small flows among the mapped buckets (like a "load balancing" process) to control the per-flow variance.

B. 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. A typical example is the Tofino architecture, and its constraints and corresponding solutions are discussed as follows.

Constraints in RMT switches: As typical examples of RMT (reconfigurable match-action table) switches, Tofino switches use pipeline architecture. 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.

Circular dependencies and their removal: Unfortunately, stochastic variance minimization introduces two forms of circular dependency as illustrated in Figure 3, making it incompatible with the unidirectional workflow in RMT switches. This work designs a hardware-friendly algorithm (see details in §IV-B) to remove these dependencies for better performance and resource efficiency in hardware.

- First, remove the circular dependency across buckets. Instead of running one instance of stochastic variance minimization on d buckets, the hardware-friendly CocoSketch runs d instances of stochastic variance minimization, each performed on only one bucket. To control the errors, it uses the *median* value among the d buckets as the final result.

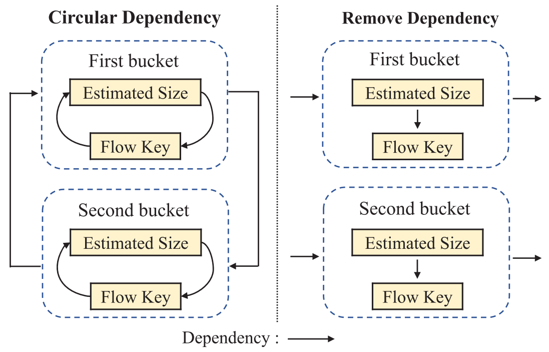


Fig. 3. Removing circular dependency for hardware-friendly design.

- Second, remove the circular dependency between the flow key and its estimated size within each bucket. The hardware-friendly CocoSketch simplifies the update logic, and puts the flow key and the estimated size into separate stages. Thus, the update process in one bucket can be pipelined.

While removing the circular dependencies might weaken the accuracy guarantee, evaluations in §VIII-E demonstrates that the accuracy drop is not significant (*e.g.*, <10%).

IV. Detailed Design

A. Basic CocoSketch

Data structure: As shown in Figure 4, 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). 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 $h_1(.), \ldots, h_d(.)$.

Basic CocoSketch insertion: Each incoming packet is denoted as a pair of (e, w), where e is a particular full key, and w is its increment size. To insert (e, w), CocoSketch first maps 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. CocoSketch increases $B_k[h_k(e)].V$ by w. And then with probability $\frac{w}{B_k[h_k(e)].V}$, CocoSketch replaces $B_k[h_k(e)].K$ with e. If multiple buckets share the same smallest size value, randomly select one to update. §VI will formally analyze the fidelity of stochastic variance minimization over arbitrary partial keys. Note that for each incoming packet, the insertion logic guarantees that it only updates the value of only one bucket and the key of at most one bucket.

Example (Figure 4): To insert packet $(e_5, 1)$, CocoSketch first uses 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, CocoSketch simply increments the corresponding value by 1 (from 15 to 16). To insert packet $(e_3, 4)$, CocoSketch first maps it to the two buckets with content $(e_6, 19)$ and $(e_2, 12)$. Since e_3 is not recorded in either bucket, CocoSketch identifies the bucket with the smallest counter (*i.e.*, $(e_2, 12)$), increments the value by 4,

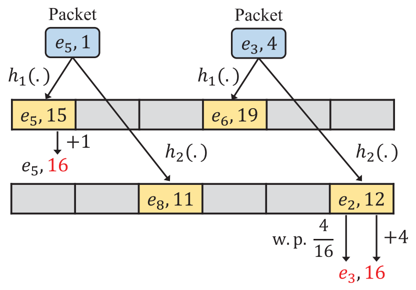


Fig. 4. Insertion example in basic CocoSketch (with $d = 2$).

and finally, with probability $\frac{w}{B_k[h_k(e)].V} = \frac{4}{16}$, replaces the key e_2 with e_3.

B. Hardware-friendly CocoSketch

To optimize the resource efficiency in hardware, the hardware-friendly CocoSketch *removes circular dependencies* in insertions.

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, the hardware-friendly CocoSketch updates each bucket independently, as if $d = 1$ in stochastic variance minimization: hardware-friendly CocoSketch always increments the value of the mapped bucket $B_i[h_i(e)]$ by w and replaces the key $B_i[h_i(e)].K$ with probability $\frac{w}{B_i[h_i(e)].V}$.

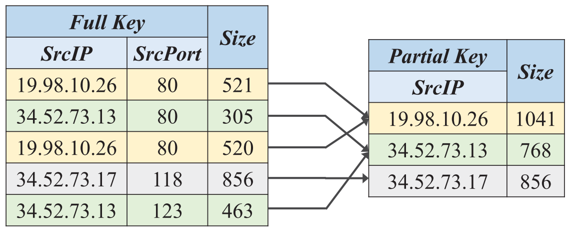
C. Query for Arbitrary Partial Key

Query front-end: CocoSketch provides a front-end for querying arbitrary partial key $k_P \prec k_F$. CocoSketch first builds 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, it 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 5): Suppose that the full key is (SrcIP, SrcPort), and the queried partial key is SrcIP. CocoSketch first gets the full key result (left). Then, CocoSketch aggregates 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 CocoSketch adds 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.



Full Key		Size
SrcIP	SrcPort	
19.98.10.26	80	521
34.52.73.13	80	305
19.98.10.26	80	520
34.52.73.17	118	856
34.52.73.13	123	463

Partial Key	Size
SrcIP	
19.98.10.26	1041
34.52.73.13	768
34.52.73.17	856

Fig. 5. Example queries on partial keys.

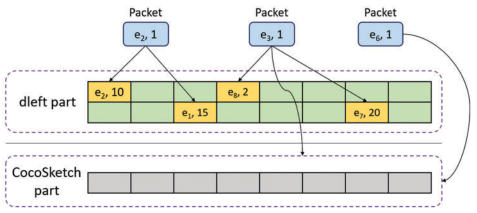


Fig. 6. The data structure of D-CocoSketch and running examples.

D. Qualitative Analysis

Compared with the baseline that builds one single-key sketch for each partial key, CocoSketch has the advantage in terms of memory efficiency and insertion throughput. To provide the same error bound, such baseline consumes $O(K)$ memory, where K is the number of partial keys, while the memory consumption of CocoSketch is independent of K. For insertion, the baseline requires $O(K)$ hash computations and $O(K)$ memory accesses, while CocoSketch requires $O(1)$ hash computation and memory access.

Compared with the baseline that builds a single-key sketch with post recovery, CocoSketch is more efficient in memory and query time. For single-key sketch such as CM sketches [16], to provide the same error bound, the required memory of such baseline is $O(S)$, where S is the number of full keys contained in the partial key. For CocoSketch, the memory consumption is independent of S (see §VI-B). The query time of such baseline is also $O(S)$, while that of CocoSketch is independent of S.

Compared with USS, as analyzed in §III-A, the time complexity of insertion is $O(n)$, where n is the number of buckets, while for CocoSketch it is $O(d)$, and d is much smaller.

V. CocoSketch Can Do More

CocoSketch can support addition requirements beside arbitrary partial key query: (1) providing exact information of specified full flow keys; (2) distributed measurement.

A. Requirements of Exact Information for Specified Full Keys

While the demand of exact information is common, CocoSketch itself fails to meet the requirement. Therefore, this work designs a variant of CocoSketch that can support both partial key query and providing exact information of full keys.

Data structure: D-CocoSketch, a variant of CocoSketch, utilizes a d-left hash table to record full keys. As shown in Figure 6, D-CocoSketch consists of two parts: the d-left part and the CocoSketch part. The d-left part is composed of h sub-tables, $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_h$, and each sub-table has m buckets. The i-th sub-table is associated with a hash function $g_i(\cdot)$, which hashes a full key to $\{0, 1, \cdots, m-1\}$. The buckets record exact information about flows that operators are interested in, including the full keys and corresponding frequencies. The CocoSketch part is a one-array hardware-friendly CocoSketch, which consists of a bucket array $\mathcal{B}$. Here the hardware-friendly CocoSketch is chosen to support hardware deployment, because deploying measurement functionality in programmable switches has shown huge benefits of high packet processing speed [53].

Insertion: D-CocoSketch supports two modes: the sample mode and the rule match mode. In the sample mode, users specify a sample rate p, and D-CocoSketch samples flows according to the sample rate p. For the incoming packet that belongs to the sampled flows, D-CocoSketch first tries to insert it in the d-left part. If the insertion fails, it will then be inserted into the CocoSketch part. For unsampled packets, they will be directly inserted in the CocoSketch part. In the rule match mode, users specify the flow match rules. For the incoming packets that match the rules, their insertion behavior is the same as the sampled packets in the sample mode. And the unmatched packets are the same as the unsampled packets.

For the incoming packet that will be tried to insert to the d-left part, it is denoted as (e, w), where e is the full key and w is its incremental frequency size. D-CocoSketch sequentially tries to insert (e, w) to the d-left hash tables. For the i-th hash table $\mathcal{D}_i$, if the hashed bucket, $\mathcal{D}_{\rangle}[g_i(e)]$, is empty or has the same full key of e, then the frequency counter will be updated. Otherwise, D-CocoSketch tries to insert it to the next $(i+1)$-th hash table. This process stops once it is successfully inserted to one of the hash tables.

Examples: As shown in Figure 6, when inserting packet $(e_2, 1)$, it is sampled or matches the rules, so D-CocoSketch tries to insert it in the d-left part. The hashed bucket in the first sub-table has the same key of e_2, so the insertion in the d-left part succeeds, and the bucket is updated from $(e_2, 10)$ to $(e_2, 11)$. When inserting packet $(e_3, 1)$, it is sampled or matches the rules, so D-CocoSketch also tries to insert it in the d-left part. However, the two hashed buckets in the sub-tables are not empty and record different keys, so the insertion fails. It will then be inserted in the CocoSketch part, following the insertion of the hardware-friendly CocoSketch. When inserting packet $(e_6, 1)$, it is not sampled or does not match the rules, so D-CocoSketch directly inserts it in the CocoSketch part.

Query front-end: The query front-end of arbitrary partial key for D-CocoSketch is similar to that of the CocoSketch. The first step is to build a two-column table of *(Full Key, Size)*. As the flows that are inserted in the d-left hash tables will not appear in the CocoSketch part, the hash tables and the CocoSketch part are simply stitched together. Note that D-CocoSketch uses one-array hardware-friendly CocoSketch, so calculating median value is unnecessary. The query interface on the built table is the same in §IV-C.

B. Requirements of Distributed Deployment

To achieve full coverage in the network measurement, CocoSketch should be deployed in a distributed manner. The challenge lies in how to let multiple CocoSketchs provide

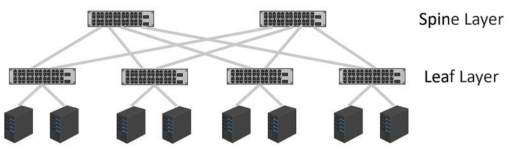


Fig. 7. An example of spine-leaf topology with 2 spine switches, 4 leaf switches and 8 servers.

unbiased estimation collectively. The key idea is to utilize the hierarchy of data center network topology to ensure unbiased estimation for all flows.

Solution: Ideally, CocoSketches should be deployed on multiple switches with a collective query method, which should provide unbiased estimation for any flows in order to support arbitrary partial key query. Formally, suppose there are n CocoSketches deployed on n different switches, and suppose the set of flows to be measured is F. Assume for any flow f and any switch, either all of its packets pass through the switch or none of them. Suppose for i-th CocoSketch, the set of flows that pass through its corresponding switch and are inserted in the sketch is F_i. Although it is generally difficult to obtain an unbiased estimation from multiple CocoSketches, it can be obtained when F_i are a *partition* of F, that is, the following two conditions are met: (1) $F_i \cap F_j =, \forall i \neq j$; (2) $\cup_{i=1}^{n} F_i = F$.

The collective query front-end is to simply merge the tables from all CocoSketches, and query results on the merged table in the same way as single-node CocoSketch. As a result, multiple CocoSketches make up "one big CocoSketch". We will provide formal proof in §VI-C.

Thanks to the hierarchy in the data center network, the distributed deployment of CocoSketch that satisfies the flow partition can be easily found. The topology of data center network is usually a multi-rooted tree, and the switches are located in different hierarchies. This work assumes the data center uses ECMP as the routing protocol, which force one flow to choose only one switch as the entry of the higher layers. CocoSketch can be deployed on the lowest layer where all of the flows will pass through. In this way, the packets of each flow will be inserted in exactly one of the switches, and each packet is inserted exactly once. For instance, in a two-level spine-leaf topology shown in Figure 7, CocoSketches can be deployed on the switches of the leaf layer. Similarly, in a three level Fat-Tree topology, CocoSketches can be deployed on the switches of the edge layer.

An intuitive way to understand the distributed deployment of CocoSketch is that, it can be thought as a "one big logical CocoSketch" that is physically stored on multiple switches. There are special hash functions that hash packets to one specific position in one specific block, which is determined by both the forward path of the packets and the hash functions in the passed switches. The insertion and query operations remain the same, so it can work in a distributed manner.

VI. Analysis

A. Stochastic Variance Minimization

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. 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.

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

Stochastic variance minimization for $d = 1$: First, consider 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. (1), the key point is how 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. (1) is

$$(e', f') = \begin{cases} (e_i, f_j + w), \text{ w.p. } \dfrac{w}{f_j + w} \\ (e_j, f_j + w), \text{ w.p. } \dfrac{f_j}{f_j + w} \end{cases} \tag{2}$$

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 CocoSketch updates the mapped bucket to (e', f') to optimize Eq. (1). Note that CocoSketch only changes 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 $e_i = e_j$, CocoSketch directly updates 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 the bucket is set to $(e', f') = (e_i, w/p)$ with probability p, and set $(e', f') = (e_j, f_j/(1-p))$ with probability $1 - p$. The increment of variance is that

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

Therefore, CocoSketch achieves the minimum variance when $p = w/(f_j + w)$. Based on the formula of p, Eq. (2) is proved. □

Based on the Eq. (2), the following theorem holds,

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

$$\sum_{e} \Delta \left(f(e) - \widehat{f}(e) \right)^2 = \begin{cases} 2\ w f_j, & e_i \neq e_j \\ 0, & e_i = e_j \end{cases} \tag{3}$$

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

$$\sum_{e} \Delta \left(f(e) - \widehat{f}(e) \right)^2 = 2\ w f_j$$

□

Stochastic variance minimization for $d > 1$: For $d > 1$, consider the i^{th} mapped bucket. If $e_i = e_j$, the increment of variance is 0. If $e_i \neq e_j$, the increment of variance is $2wf_j$.

Therefore, CocoSketch should first update the bucket with the same full key, or otherwise update the bucket with smallest value.

B. Error Bound

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: The first step is to 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. The unbiasedness is proved 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

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

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.

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

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

□

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, the expectation of the estimated size in each array is,

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

□

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

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

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

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

There is

$$\begin{aligned} \text{Var}\left[\widehat{C_{i,j}}(e)\right] &= C_{i,j}(e) \cdot \mathbb{E}\left[B_i[j].V - C_{i,j}(e)\right] \\ &= C_{i,j}(e) \cdot \frac{f(\overline{e})}{l} \\ \text{Cov}\left[\widehat{C_{i,j}}(e), \widehat{C_{i,k}}(e)\right] &= 0, j \neq k \end{aligned}$$

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

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

□

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,

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

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

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

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. □

C. Extensions of CocoSketch

This section gives the proof of the unbiasedness for the D-CocoSketch and the distributed CocoSketch.

Theorem 4: For any flow e of any key $k \prec k_F$, in the D-CocoSketch,

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

Proof: For any flow e, it will be inserted to either the dleft part or the CocoSketch part. If the flow e is inserted to the dleft part, as the dleft part is a hash table, the recorded frequency is fully accurate. If the flow e is inserted to the CocoSketch part, according to Lemma 3, the result is also unbiased. Therefore, the estimation result of D-CocoSketch is unbiased. □

Theorem 5: For any flow e of any key $k \prec k_F$, in the distributed CocoSketch,

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

Proof: For the distributed deployment of CocoSketchs, the flow e will appear and only appear in one of the CocoSketchs. Suppose the packets of flow e are inserted to CocoSketch C, and the estimated frequency of flow e on sketch C is $\widehat{f_C}(e)$. As the flow e is not inserted in any other sketches, the key e will not appear in the table of query front-end for other sketches. Therefore, the estimated frequency is completely dependent on the sketch C. As CocoSketch provides unbiased estimation,

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

□

VII. Deployment Platforms

CocoSketch has been implemented on five platforms: x86 CPU, Redis [37], Open vSwitch (OVS) [36], Xilinx FPGA [39], and Barefoot Tofino [38]. In this section, the implementation of basic CocoSketch on CPU, Redis and OVS and hardware-friendly CocoSketch on FPGA and Barefoot Tofino is described. In addition, the deployment of CocoSketch on the wireless devices is discussed.

A. Basic CocoSketch Implementation

CPU Implementation: Basic CocoSketch is implemented (§IV-A) using C++. The hash functions are implemented using the 32-bit Bob Hash [54] with different hash seeds. Its implementation and evaluation is 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.

Redis implementation: Basic CocoSketch is implemented as a module in the Redis, where user can use the provided API to create CocoSketch, insert full keys and query for aggregate sum of partial keys. MurmurHash [55] is used as the hash function. The Redis implementation is evaluated on a machine with dual 18-core CPUs (36 threads, Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz) and 125GB DRAM memory.

OVS Implementation: CocoSketch is implemented on OVS v2.12.1 with DPDK 18.11.10. Ring buffers are used 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. The 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, multiple (*e.g.*, 4) Rx queues are assigned for the DPDK receive port in OVS. Different Rx queues are pinned to different cores and are polled by different Poll Mode Driver threads.

B. FPGA Platform

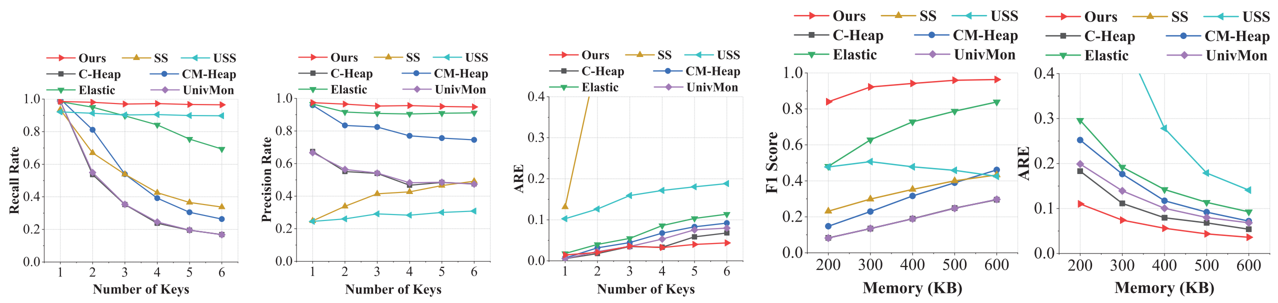
FPGA background: FPGAs [39] are based on a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. The main resources of FPGA include Slice LUTs, Slice Registers, and Block RAM Tile. Slice LUTs are lookup tables, which are used to implement combinational logic. Slice Registers are mainly used as cache resources. Block RAM Tile is on-chip block storage, which is the main storage resource.

FPGA implementation: The hardware-friendly version (§IV-B) is implemented on a Xilinx Alveo U280 [39] with full pipelining. The algorithm is divided 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. The implementation pipelines all the key/value memory accesses to improve the clock rate. To replace the key in a bucket with some probability $p \in (0, 1]$, the implementation first generates a 32-bit random number $rand$, then replaces the key recorded only if $rand \times \frac{1}{p} < 2^{32}$.

C. RMT Platform

P4 background: In RMT-based programmable switches [56], 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: A P4 prototype of the hardware-friendly CocoSketch is implemented on the Tofino switch [38]. The difficulty of implementing the hardware-friendly CocoSketch on the Tofino switch lies in the calculation of probability. Because the multiplication operation between two variables is not supported, the implementation uses a different way to calculate the probability. In P4, to replace the key recorded with probability $\frac{1}{value}$, the implementation first generates a 32-bit random number $rand$ and then replaces 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 $0.1\ p$. 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.



(a) Recall Rate (b) Precision Rate (c) ARE (d) F1 Score (e) ARE

Fig. 8. Performance of heavy hitter detection under different numbers of partial keys and different memory.

TABLE II
METRICS FOR EVALUATIONS

Metric	Description
Recall Rate (RR)	$\frac{\text{\# correctly reported flows}}{\text{\# correct flows}}$
Precision Rate (PR)	$\frac{\text{\# correctly reported flows}}{\text{\# reported flows}}$
F1 Score	$\frac{2 \cdot \text{RR} \cdot \text{PR}}{\text{RR} + \text{PR}}$
Average Relative Error (ARE)	$\frac{1}{n}\sum_{i=1}^{n}\frac{\lvert f(e_i)-\widehat{f}(e_i)\rvert}{f(e_i)}$, where $f, \widehat{f}$ are real and estimated sizes
Throughput	Million packets per second (Mpps)
95^{th} percentile CPU cycles	95^{th} percentile CPU cycles of per-packet processing

D. Wireless Platform

Prior works [57], [58], [59] have explored the application of sketching algorithms on wireless platforms. Specifically, Joltik [59] deploys sketching algorithms on wireless sensors and base stations by implementing sketches on the STM32L476RGT6U MCU in the NUCLEO-L476RG board. Their code is implemented in C using Mbed compiler, which supports both C and C++. As CocoSketch can be implemented in C++, it can also be deployed on wireless platform in the same method.

VIII. EVALUATION

A. Experimental Setup

Traces: The experiments use two real-world traces. (1) *CAIDA:* The traces collected in the Equinix-Chicago monitor from CAIDA in 2018 [60]. In the evaluation, the trace with a monitoring interval of 60s is used, which contains around 27M packets. (2) *MAWI:* The traffic traces collected by MAWI [61]. In the evaluation, the trace with a monitoring interval of 15min is used, which contains around 13M packets.

Metrics: The six performance metrics in Table II are evaluated, which are well acknowledged as the metrics of the accuracy and packet processing efficiency for sketching algorithms by the research community [21], [22], [23], [24], [25], [26], [27]. The resource numbers are reported in hardware deployments.

Setting: By default, the experiments 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 heavy hitter threshold to be 10^{-4} of the total size of traffic, and set the total memory at 500KB. The average metrics on these keys are reported. For the CocoSketch and USS, one sketch with 500KB memory is used to measure the full key (5-tuple) and results of other keys are got by aggregation. Other single-key algorithms uses one sketch for each key, as in prior work [24], [28], [62].

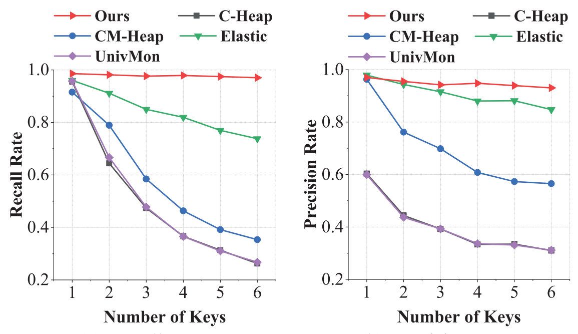
B. Accuracy

The basic CocoSketch ("Ours" in the figures) is compared with other sketches in three tasks (Heavy Hitters, Heavy Changes, and HHH) with the six partial keys described in §VIII-A. The baselines include Count sketch [17] with a min-heap (C-Heap), Count-Min sketch [16] with a min-heap (CM-Heap), SpaceSaving (SS) [15], the software version of the Elastic sketch [22], UnivMon [24], and Unbiased SpaceSaving (USS) [33]. In particular, USS uses an optimized implementation whose update process is enhanced by a hash table and a double-linked list. (Throughput of a naive USS implementation is < 0.1 Mpps.) A hash table is used to check whether a flow is already tracked in the sketch; and a double-linked list is maintained to rank buckets by their counters so that the minimal bucket can be found quickly. In contrast, CocoSketch does not require extra memory.

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.

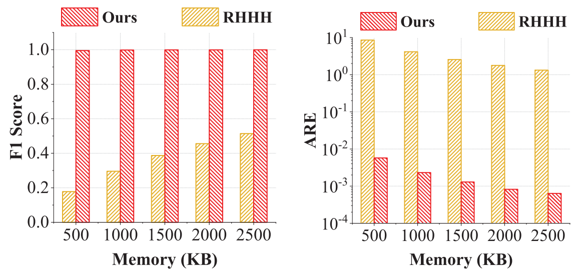
Heavy hitter detection under different memory configurations (Figures -8(d)-8 (e)): CocoSketch also achieves higher accuracy with smaller memory footprints when measuring the 6 keys. With only 300KB memory, the F1 Score of CocoSketch is above 90%, while others are usually below 65%. The ARE of CocoSketch is around 10.43× better than the baseline algorithms. Note that SS is not shown in Figure 8(e) because its ARE is too large (> 0.4).

Heavy change detection with different number of keys (Figures 9(a)-9(b)): Similar to that of heavy hitter detection,



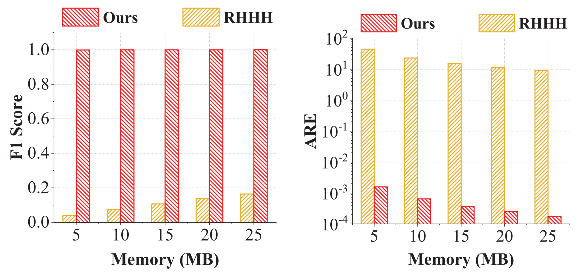
(a) Recall Rate (b) Precision Rate

Fig. 9. Heavy change detection under different numbers of partial keys.



(a) F1 Score (b) ARE

Fig. 10. 1-d HHH with different memory constraints.



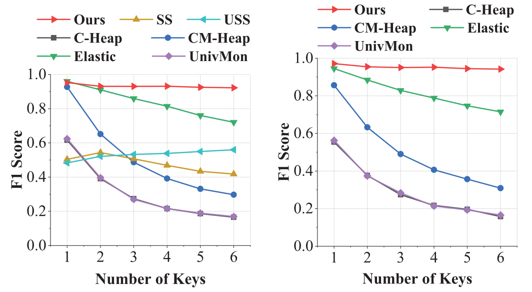
(a) F1 Score (b) ARE

Fig. 11. 2-d HHH with different memory constraints.

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.

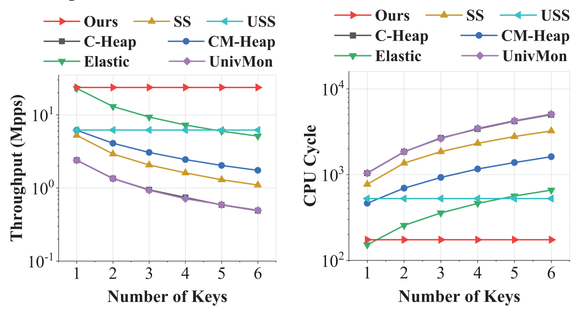
1-d HHH detection with different memory (Figure 10): 1-d HHH detection evaluates the source IP hierarchy in bit granularity (32 prefixes + 1 empty key). The basic CocoSketch is compared with R-HHH [28] only, because the throughput of other baselines is too low to measure these many keys. With only 500KB 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 11): 2-d HHH detection evaluates source/destination IP hierarchies in bit granularity ($33 \times 33 = 1089$ keys). With 5MB memory, the F1 Score of CocoSketch is higher than 99.8%. R-HHH uses more than 5MB memory in this experiment, since it cannot work with smaller memory. Even with 25MB memory, its F1



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

Fig. 12. Experiment results on the MAWI dataset.



(a) Throughput in CPU (b) 95th percentile CPU cycles

Fig. 13. Processing speed in CPU platform.

Score is about 16%. The ARE of CocoSketch is about 39843× smaller than that of R-HHH.

Experiments on MAWI traces (Figure 12(a)-12(b)): The evaluation of heavy hitters detection and heavy changes detection is also conducted on MAWI traces. CocoSketch maintains high accuracy. When tracking more than two partial keys, CocoSketch achieves over 90% F1 Score and is better than all baselines.

C. Software Platforms

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

Throughput in CPU (Figure 13(a)): The memory configuration in this experiment is the same as that in the heavy hitter detection (§VIII-B). The evaluation uses 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.

95th percentile CPU cycle (Figure 13(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 95th 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 14(a)): The throughput of the CocoSketch increases with the number of threads. With

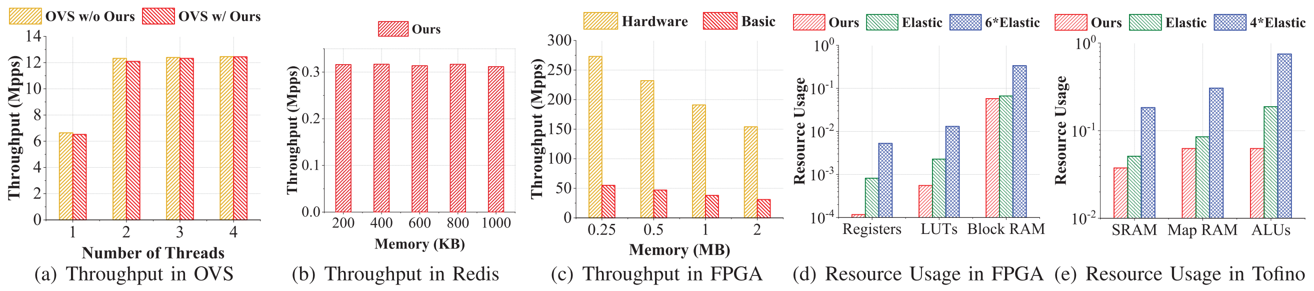


Fig. 14. Resource usage and throughput on different platforms.

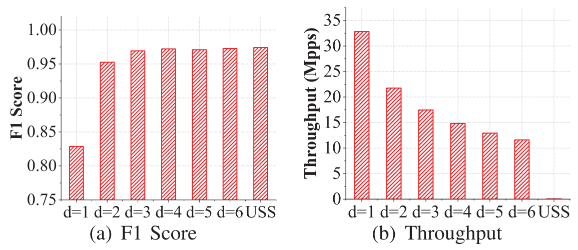


Fig. 15. Varying d's in the basic CocoSketch.

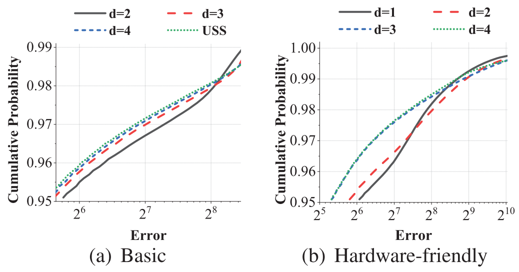


Fig. 16. CDF of absolute error under different d values.

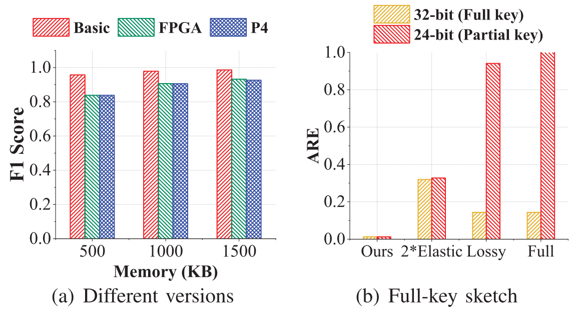


Fig. 17. (a) Different versions of CocoSketch, and (b) CocoSketch vs. full-key sketch baselines.

two or more threads, CocoSketch reaches the speed limit of the evaluated 40Gbps NIC. CocoSketch incurs a small CPU overhead ($< 1.8\%$).

Throughput in Redis (Figure 14(b)): The throughput of the CocoSketch in Redis is not affected by the memory consumption. The reason is that Redis requires to send requests to the Redis server, and the processing speed of CocoSketch will not be the bottleneck. Therefore, the cache utility of CocoSketch has little effect on the throughput. The experimental results show that CocoSketch in Redis has a throughput of around 0.3 Mpps.

D. Hardware Platforms

In this section, the hardware-friendly CocoSketch (Ours) is compared with Elastic Sketch [63]. Elastic Sketch has multiple versions designed for different platforms [22], each has a different performance. The memory configurations of evaluated sketches guarantee 90% F1 Scores in heavy hitters detection (via accuracy experiments).

Throughput in FPGA platform (Figure 14(c)): Hardware-friendly CocoSketch achieves about 5 times higher throughput 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.

Resource usage in FPGA platform (Figure 14(d)): 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 14(e)): The figure shows the ratio of the resources used by algorithms to the total resources of 12 stages in the Tofino switch. In the figure, "4*Elastic" indicates the resources used by Elastic Sketch when measuring 4 keys. Note that a Tofino switch data plane can implement at most 4 Elastic sketches at the same time due to the resource constraint. 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.

E. Microbenchmark

In this section, the performance under different parameter settings and different versions of CocoSketch is shown.

Varying d in the basic CocoSketch (Figures 15 -16(a)): The memory size is fixed at 500KB, and the performance under different d is shown by the accuracy of heavy hitters detection. When the value of d decreases 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,

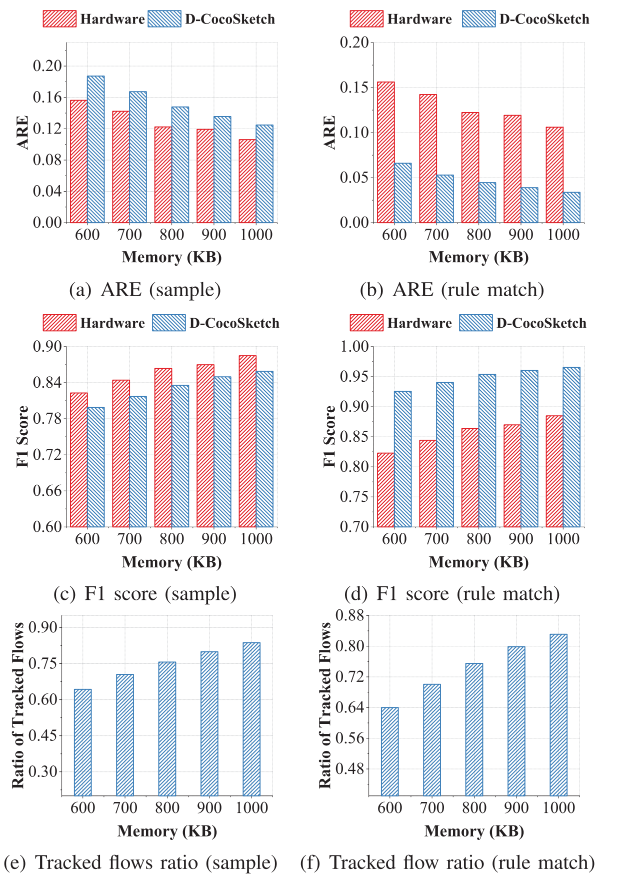


Fig. 18. Performance of D-CocoSketch under different memory consumption.

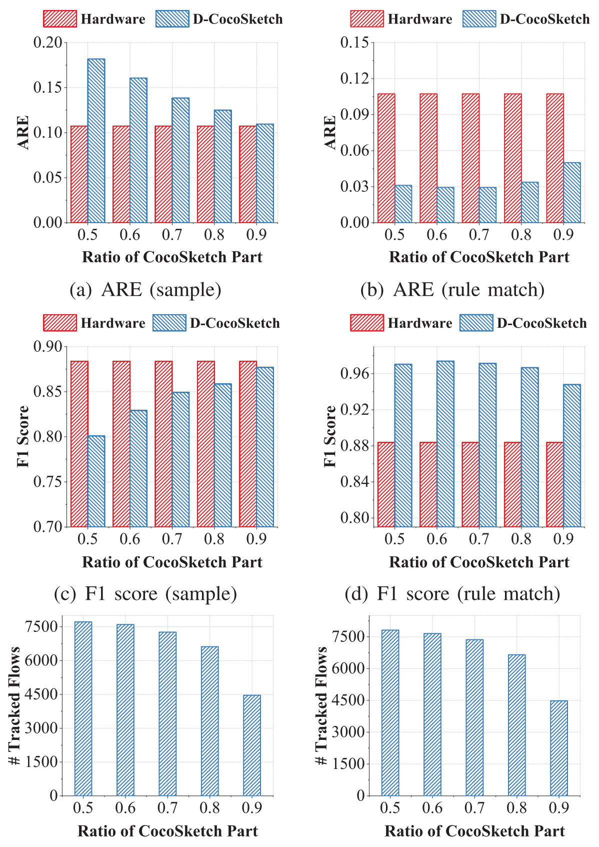


Fig. 19. The effect of the memory ratio r on D-CocoSketch.

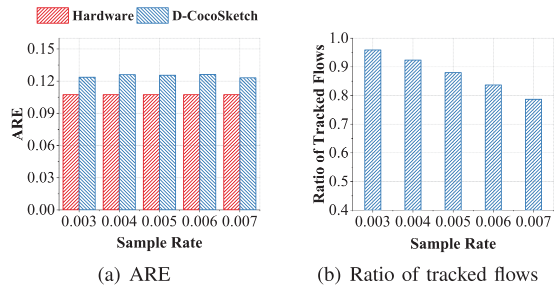


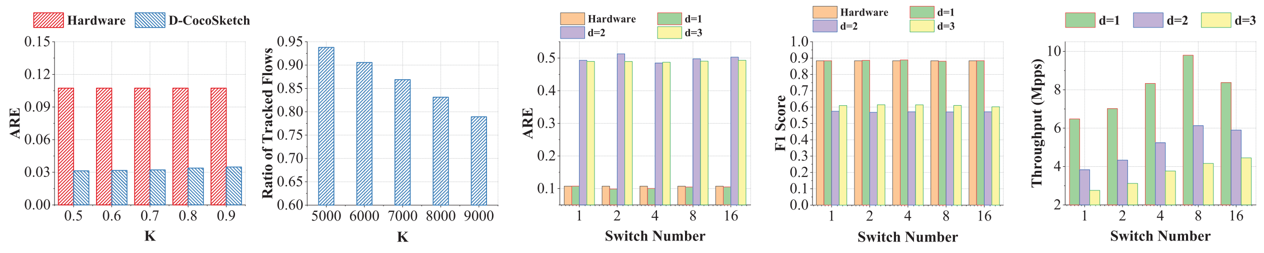
Fig. 20. The effect of sample rate p in the sample mode.

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 16(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, the experiment fixes the memory size at 500KB and shows the CDF of error under different d, *i.e.*, for each distinct flow e, calculating its error $|\hat{f}(e) - f(e)|$ and getting the distribution of error. The results show that, with a larger d, CocoSketch has a small error with a higher probability, while its worst case is worse than others. Specifically, 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 17(a)): The experiments compare the F1 Score of three versions of the CocoSketch on the heavy hitters detection: 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). 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%. The accuracy gap between the hardware implementations in FPGA and P4 is smaller than 1%, which indicates that the approximate division technique used in the P4 implementation (§VII-C) has negligible impact on the accuracy.

Comparison with full-key sketch (Figure 17(b)): To compare CocoSketch to different strawman solutions shown in §II-C, The experiment measures two keys, SrcIP (full key) and its 24-bit prefix (partial key), and shows their ARE respectively. The total memory is fixed at 6MB and the ARE is calculated 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" (one Elastic Sketch for each key), the ARE of both full key and partial key are around 0.3. For "Lossy" (recovering 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. For "Full" (recovering 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.



(a) ARE (b) Ratio of tracked flows (c) ARE (d) F1 score (e) Throughput

Fig. 21. (a)-(b): the effect of the matched top-k heavy-hitter number for D-CocoSketch; (c)-(d): evaluations of CocoSketch in the distributed scenario.

F. D-CocoSketch

This section shows the evaluation of the performance of D-CocoSketch under different memory consumption and compare it with hardware-friendly CocoSketch. The effect of different parameter settings is also evaluated, including the ratio of the CocoSketch part memory to the total memory r, and the number of sub-tables in the dleft part h, the sample rate p in the sample mode, and the number of matched flows in the rule match mode. All experiments use CAIDA traces and evaluate heavy hitter detection on six partial keys. For the rule match mode, top-k heavy hitters of the full keys are selected as the matched flows. Without specific discussion, the number of sub-table is set to 2 and the ratio of CocoSketch part is set to 0.8. In the sample mode, the sample rate is set to 0.006, and in rule match mode, the number of matched flows K is set to 8000. The total memory is set to 1000KB in default.

The performance of D-CocoSketch under different memory (Figure 18): The experimental results show that, D-CocoSketch can achieve comparable accuracy with hardware-friendly CocoSketch, while providing exact information for tracked flows. When the memory consumption grows from 600KB to 1000KB, compared with hardware-friendly CocoSketch that consumes the same memory, the increase of ARE is less than 22.52%, and the drop of F1 score is less than 3.37%. The ratio of tracked flows ranges between 0.64 and 0.84 in the sample mode, and between 0.64 and 0.83 in the rule match mode. The reason why D-CocoSketch in the rule match mode improves accuracy is that, top-k heavy hitters of full keys is used as matched flows, and in CAIDA traces the heavy hitters of full keys contribute most to the heavy hitters of partial keys.

The effect of r (Figure 19): The experiments vary the ratio r of the CocoSketch part from 0.5 to 0.9, and evaluate the D-CocoSketch on the accuracy of heavy hitter detection and its ability to track exact information. As shown in the figure, as the memory ratio of the CocoSketch part grows, the accuracy gap between D-CocoSketch and hardware-friendly CocoSketch reduces, while the number of tracked flows also reduces.

The effect of the sample rate p (Figure 20): The sample rate varies from 0.003 to 0.007. The experimental results show that, the sample rate has little effect on the accuracy, while the ratio of tracked flows decreases as the sample rate increases. When the sample rate increases from 0.003 to 0.007, the ratio of tracked flows has a 18% drop, and the ARE ranges between 0.123 and 0.126.

The effect of the matched top-k flow number (Figure 21(a) - 21(b)): In the rule match mode, the number of match flows varies from 5000 to 9000, and the experimental results are similar to those in the sample mode. The results show that, when the number of matched flows increases from 5000 to 9000, the ratio of tracked flows has a 16% drop, and the ARE ranges between 0.031 and 0.035.

G. Distributed Measurement

The evaluation simulates distributed measurement with a multi-threading program, where one thread simulates the insertion of CocoSketch in one switch. The experiment uses the CAIDA trace, and for each flow a random switch is selected as its measurement node, where all of its packets are inserted. The accuracy and throughput of CocoSketch is evaluated in the distributed measurement

Experimental results (Figure 21(c) - 21(e)): The experimental results show that, the accuracy of distributed deployment of CocoSketch is close to that of the single-node deployment. As the number of switches grows from 1 to 16, the fluctuation of ARE is less than 7.2%, and the fluctuation of F1 score is less than 0.5%. This is because, the distributed deployment strategy is equivalent to divide a big CocoSketch into blocks and use a special hash function to hash a subset of flows into one block. The distribution of flows in the CocoSketch is still uniform and thus has good performance. For throughput, as the number of switches grows from 1 to 8, the throughput increases by up to 1.6×. The throughput may drop when the number of switches grows from 8 to 16, possibly due to the schedule overhead.

IX. Conclusion

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. This paper presents 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. Moreover, CocoSketch can support additional requirements while maintaining high accuracy for partial key queries. Experiments demonstrate the performance of CocoSketch by comparing it with a variety of sketches under real-world traces.

Acknowledgment

The authors thank the anonymous reviewers for their valuable suggestions.

References

[1] Y. Zhang et al., "CocoSketch: High-performance sketch-based measurement over arbitrary partial key query," in *Proc. ACM SIGCOMM Conf.*, 2021, pp. 207–222.

[2] A. Feldmann and A. Greenberg, "Deriving traffic demands for operational IP networks: Methodology and experience," in *Proc. ACM SIGCOMM*, 2000, pp. 257–270.

[3] C. Guo et al., "Pingmesh: A large-scale system for data center network latency measurement and analysis," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 139–152, 2015.

[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM*, 2011, pp. 254–265.

[5] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. SIGCOMM*, 2018, pp. 357–371.

[6] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "cSamp: A system for network-wide flow monitoring," in *Proc. NSDI*, 2008, pp. 233–246.

[7] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proc. SIGCOMM*, 2017, pp. 15–28.

[8] Z. Liu et al., "DistCache: Provable load balancing for large-scale storage systems with distributed caching," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST)*, 2019, pp. 1–16.

[9] M. Alizadeh et al., "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 503–514.

[10] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *Proc. NSDI*, 2018, pp. 1–17.

[11] Y. Li et al., "HPCC: High precision congestion control," in *Proc. ACM Special Interest Group Data Commun.*, 2019, pp. 44–58.

[12] X. Li et al., "Detection and identification of network anomalies using sketch subspaces," in *Proc. IMC*, 2006, pp. 147–152.

[13] C.-Y. Hong, M. Caesar, N. Duffield, and J. Wang, "Tiresias: Online anomaly detection for hierarchical operational network data," in *Proc. IEEE 32nd Int. Conf. Distrib. Comput. Syst.*, Jun. 2012, pp. 173–182.

[14] Y. Zhang, S. Singh, S. Sen, N. G. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications," in *Proc. IMC*, 2004, pp. 101–114.

[15] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. ICDT* (Lecture Notes in Computer Science), T. Eiter and L. Libkin, Eds. Berlin, Germany: Springer, 2005, pp. 398–412.

[16] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.

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

[18] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *Proc. Symp. SDN Res.*, Mar. 2018, pp. 1–7.

[19] A. Kumar, M. Sung, J. Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," in *Proc. Joint Int. Conf. Meas. Modeling Comput. Syst.*, Jun. 2004, pp. 177–188.

[20] R. T. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *Proc. IMC*, A. Lombardo and J. F. Kurose, Eds. New York, NY, USA: ACM, 2004, pp. 207–212.

[21] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better netflow for data centers," in *Proc. NSDI*, 2016, pp. 1–15.

[22] T. Yang et al., "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. SIGCOMM*, 2018, pp. 561–575.

[23] Z. Liu et al., "NitroSketch: Robust and general sketch-based monitoring in software switches," in *Proc. SIGCOMM*, 2019, pp. 334–350.

[24] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proc. SIGCOMM*, 2016, pp. 101–114.

[25] Q. Huang, P. P. C. Lee, and Y. Bao, "SketchLearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proc. SIGCOMM*, 2018, pp. 576–590.

[26] Q. Huang et al., "SketchVisor: Robust network measurement for software packet processing," in *Proc. SIGCOMM*, 2017, pp. 113–126.

[27] J. Li et al., "WavingSketch: An unbiased and generic sketch for finding top-k items in data streams," in *Proc. KDD*, 2020, pp. 1574–1584.

[28] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 127–140.

[29] O. Alipourfard, M. Moshref, and M. Yu, "Re-evaluating measurement algorithms in software," in *Proc. 14th ACM Workshop Hot Topics Netw.*, Nov. 2015, pp. 1–7.

[30] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Secur. Privacy*, vol. 1, no. 4, pp. 33–39, Jul. 2003.

[31] V. Sekar, N. G. Duffield, O. Spatscheck, J. E. van der Merwe, and H. Zhang, "LADS: Large-scale automated DDoS detection system," in *Proc. USENIX Annu. Tech. Conf., Gen. Track*, 2006, pp. 171–184.

[32] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 350–361.

[33] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," in *Proc. SIGMOD*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. New York, NY, USA: ACM, 2018, pp. 1129–1140.

[34] N. Duffield, C. Lund, and M. Thorup, "Priority sampling for estimation of arbitrary subset sums," *J. ACM*, vol. 54, no. 6, p. 32, Dec. 2007.

[35] X. Chen, S. L. Feibish, M. Braverman, and J. Rexford, "BeauCoup: Answering many network traffic queries, one memory update at a time," in *Proc. SIGCOMM*, 2020, pp. 226–239.

[36] B. Pfaff et al., "The design and implementation of open vswitch," in *Proc. NSDI*, 2015, pp. 1–15.

[37] J. Carlson, *Redis Action*. New York, NY, USA: Simon & Schuster, 2013.

[38] *Barefoot Tofino: World's Fastest P4-Programmable Ethernet Switch ASICS*. Accessed: Oct. 2010. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[39] *ALVEO U280 Data Center Accelerator Card*. Accessed: Oct. 2010. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html

[40] Y. Zhu et al., "Packet-level telemetry in large datacenter networks," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 479–491.

[41] *Inmon Corporation's Sflow: A Method for Monitoring Traffic in Switched and Routed Networks*. Accessed: Sep. 2022. [Online]. Available: https://tools.ietf.org/html/rfc3176

[42] B. Claise, G. Sadasivan, V. Valluri, and M. Djernaes, *Cisco Systems Netflow Services Export Version 9*, IETF, document RFC 3954, 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3954.txt

[43] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proc. 3rd ACM SIGCOMM Conf. Internet Meas.*, 2003, pp. 153–166.

[44] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 164–176.

[45] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Proc. 3rd ACM SIGCOMM Conf. Internet Meas.*, 2003, pp. 234–247.

[46] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *Proc. Joint Int. Conf. Meas. Modeling Comput. Syst.*, Jun. 2006, pp. 145–156.

[47] A. Wagner, A. Wagner, B. Plattner, and B. Plattner, "Entropy based worm and anomaly detection in fast IP networks," in *Proc. 14th IEEE Int. Workshops Enabling Technol., Infrastruct. Collaborative Enterprise (WETICE)*, Jun. 2005, pp. 172–177.

[48] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *Proc. Conf. Appl., Technol., Architectures, Protocols Comput. Commun.*, Aug. 2005, pp. 1–12.

[49] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proc. SIGCOMM*, 2016, pp. 129–143.

[50] M. Moshref, M. Yu, A. B. Sharma, and R. Govindan, "Scalable rule management for data centers," in *Proc. NSDI*, 2013, pp. 157–170.

[51] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 393–406.

[52] A. Kumar et al., "BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing," in *Proc. SIGCOMM*, 2015, pp. 1–14.

[53] L. Zeno et al., "Packet-level telemetry in large datacenter networks," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 171–191.

[54] *Hash Website*. Accessed: Oct. 2010. [Online]. Available: http://burtleburtle.net/bob/hash/evahash.html

[55] *Murmurhash*. Accessed: Sep. 2022. [Online]. Available: https://github.com/aappleby/smhasher

[56] P. Bosshart et al., "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, 2013.

[57] G. Li and Y. Wang, "Sketch based anomaly detection scheme in wireless sensor networks," in *Proc. Int. Conf. Cyber-Enabled Distrib. Comput. Knowl. Discovery*, Oct. 2013, pp. 344–348.

[58] A. L. De Aquino et al., "Data stream based algorithms for wireless sensor network applications," in *Proc. Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, May 2007, pp. 869–876.

[59] M. Yang, J. Zhang, A. Gadre, Z. Liu, S. Kumar, and V. Sekar, "Joltik: Enabling energy-efficient 'future-proof' analytics on low-power wide-area networks," in *Proc. 26th Annu. Int. Conf. Mobile Comput. Netw.*, 2020, pp. 1–14.

[60] *The CAIDA Anonymized 2016 Internet Traces*. Accessed: Oct. 2010. [Online]. Available: http://www.caida.org/data/overview/

[61] *MAWI Working Group Traffic Archive*. Accessed: Oct. 2010. [Online]. Available: http://mawi.wide.ad.jp/mawi/

[62] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "SCREAM: Sketch resource allocation for software-defined measurement," in *Proc. CoNEXT*, 2015, p. 14.

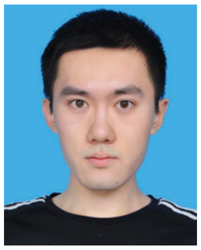
[63] *Source Code Related to Elastic Sketch*. Accessed: Oct. 2010. [Online]. Available: https://github.com/BlockLiu/ElasticSketchCode

Ruijie Miao is currently pursuing the Ph.D. degree with the School of Computer Science, Peking University, under the supervision of Tong Yang. His research interests include network measurement and streaming algorithms.

Yinda Zhang received the bachelor's degree from Peking University and the master's degree from The University of Chicago. He is currently pursuing the Ph.D. degree with the University of Pennsylvania. His research interests include network measurements and streaming algorithms.

Zihao Zheng is currently pursuing the bachelor's degree in computer science with Peking University. His research interests include network measurements, sketches, and network systems.

Ruixin Wang is currently pursuing the M.E. degree in software engineering with Peking University. His research interests include network measurements, sketches, and machine learning.



Ruwen Zhang received the B.S. degree in mathematics from Peking University in 2021, where he is currently pursuing the master's degree, under the supervision of Tong Yang. He has participated in several articles in the network area. His research interests include network and data stream processing.

Tong Yang (Member, IEEE) received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently an Associate Professor with the School of Computer Science, Peking University. He has published more than ten papers in SIGCOMM, SIGKDD, SIGMOD, and NSDI. His research interests include network measurements, sketches, IP lookups, Bloom filters, and KV stores.

Zaoxing Liu received the Ph.D. degree in computer science from Johns Hopkins University. He is currently an Assistant Professor of electrical and computer engineering and computer science with Boston University (BU). Prior to BU, he held a post-doctoral position with Carnegie Mellon University. His research has won interdisciplinary recognitions, including USENIX FAST Best Paper Award, USENIX ATC "Best of Rest," and ACM STOC "Best of Rest."

Junchen Jiang received the bachelor's degree from Tsinghua University in 2011 and the Ph.D. degree from CMU in 2017. He is currently an Assistant Professor of computer science with The University of Chicago. His research interests include networked systems, multimedia systems, and their intersections with machine learning. He was a recipient of the Google Faculty Research Award in 2020, the National Science Foundation CAREER Award in 2022, and the CMU Computer Science Doctoral Dissertation Award in 2017. He has served as an Area Chair for *Journal of Systems Research* (JSys) in 2021.