Sketchovsky: Enabling Ensembles of Sketches on Programmable Switches

Hun Namkung, *Carnegie Mellon University;* Zaoxing Liu, *Boston University;* Daehyeok Kim, *Microsoft Research;* Vyas Sekar and Peter Steenkiste, *Carnegie Mellon University*

https://www.usenix.org/conference/nsdi23/presentation/namkung

This paper is included in the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Sketchovsky: Enabling Ensembles of Sketches on Programmable Switches

Hun Namkung[⋆], Zaoxing Liu[†], Daehyeok Kim[§], Vyas Sekar[⋆], Peter Steenkiste[⋆]
[⋆]*Carnegie Mellon University,* [†]*Boston University,* [§]*Microsoft Research*

Abstract

Network operators need to run diverse measurement tasks on programmable switches to support management decisions (e.g., traffic engineering or anomaly detection). While prior work has shown the viability of running a single sketch instance, they largely ignore the problem of running *an ensemble of sketch instances* for a collection of measurement tasks. As such, existing efforts fall short of efficiently supporting a general ensemble of sketch instances. In this work, we present the design and implementation of Sketchovsky, a novel *cross-sketch optimization and composition* framework. We identify five new *cross*-sketch optimization building blocks to reduce critical switch hardware resources. We design efficient heuristics to select and apply these building blocks for arbitrary ensembles. To simplify developer effort, Sketchovsky automatically generates the composed code to be input to the hardware compiler. Our evaluation shows that Sketchovsky makes ensembles with up to 18 sketch instances become feasible and can reduce up to 45% of the critical hardware resources.

1 Introduction

Network operators need to concurrently run diverse measurement tasks for network management tasks such as traffic engineering, anomaly detection, load balancing, and resource provisioning [6, 10, 24, 35, 46]. A flow-level measurement task computes a desired statistic (e.g., heavy flow size or the distinct number of flows) based on the definition of a flowkey (e.g., srcIP or 5-tuple), a flow size (e.g., packet counts or bytes), and a measurement epoch (e.g., 1 minute).

Given resource constraints, *sketching algorithms* or *sketches* appear as a promising avenue to support measurement tasks on data collected from programmable switches (e.g., [4, 5, 7, 17, 21, 32, 40, 45, 49]). To support one measurement task, a sketch instance on a programmable switch is instantiated based on a sketching algorithm with configuration on parameters (e.g., flowkey or resource allocation). In practice, supporting the collection of measurement and management tasks will require simultaneously running an ensemble of sketch instances on the programmable switches.

Opts is optimizations and insts is instances

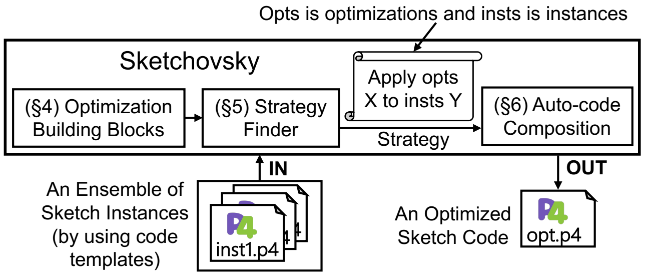

Sketchovsky
(§4) Optimization Building Blocks
(§5) Strategy Finder
Apply opts X to insts Y
Strategy
(§6) Auto-code Composition
IN
OUT
An Ensemble of Sketch Instances (by using code templates)
inst1.p4
An Optimized Sketch Code
opt.p4

Figure 1: Sketchovsky Overview.

However, existing sketch-based telemetry efforts largely focus on running a *single* sketch instance on programmable switches and they cannot effectively support *an ensemble of sketch instances*. Naively extending a single sketch instance to support an ensemble of measurement tasks will require at least a *linear* increase in hardware resources (e.g., counters, hash units, pipeline stages), which is intractable as more measurement tasks are needed. While there have been some recent efforts on improving per-sketch efficiency (e.g., [4]), supporting P4 code composition [22, 23, 30, 42, 51], elastically trading of resource vs. accuracy (e.g., [3, 36]), and on improving the generality of sketches (e.g., [9, 15, 32, 49]), we find that these fundamentally fall short of efficiently supporting a general ensemble of sketch instances without sacrificing accuracy.

Given the limitations of current methods for running sketch ensembles, we present Sketchovsky (Sketch + Tchaikovsky), a composition framework that takes the input of sketch codes for the ensemble and outputs an optimized sketch code by leveraging cross-sketch optimizations (Fig. 1). Sketchovsky is complementary to the earlier work on implementing single-sketch algorithm more efficiently, developing more general-purpose sketches, and research that explicitly trades off accuracy reduction for resource reduction (§2). Indeed, using Sketchovsky can amplify their benefits by running expressive sketches (e.g., [32]) or extending per-sketch optimizations (e.g., [4]).

The design of Sketchovsky makes three key contributions:

```
packetStream
.map(p => (p.sIP, p.pktlen))
.reduce(keys=(sIP, ), f=sum)
.filter((sIP, count) => count > Th)
```

Query 1: Heavy hitters detection of srcIP written in Sonata [25].

```
packetStream
.map(p => (p.sIP, p.dIP, p.sPort, p.dPort, p.Proto))
.distinct()
```

Query 2: Distinct number of 5-tuple flows written in Sonata [25].

Optimization building blocks (§4): We observe that sketching algorithms have three common workflow steps: hash computations, counter updates, and heavy flowkey storage. We identify and formalize five novel cross-sketch optimization building blocks to *reuse* key hardware resources *across* sketch instances in each step. For *hash computations*, we show how and when (1) hash results can be reused across sketch instances or (2) can be reconstructed by cheap XOR operations. In the *counter updates* step, we discuss how (3) counter arrays can be reused or (4) can be co-located to reduce memory accesses. In *heavy flowkey storage*, we discuss (5) a novel mechanism to reuse the heavy flowkey storage by using the union of all flowkeys for sketch instances in the ensemble. Each optimization guarantees no accuracy loss.

Strategy finder (§5): Given an arbitrary sketch ensemble, there are many possible ways to use and combine the above building blocks. Naively solving this problem is intractable due to the challenges in modeling resource usage, identifying conflicts for combining optimizations, and the combinatorially large search space (e.g., it takes more than a day to solve). We identify practical relaxations to the problem and a greedy heuristic to make the problem tractable to solve. We show that our approach can quickly identify (e.g., in less than 1 second) an effective strategy that yields significant benefits.

Auto-code composition (§6): To simplify developer and operator effort, we design a simple-yet-effective switch-code generation process that realizes the selected strategy obtained above. We provide code templates of sketching algorithms to create sketch codes for an ensemble of sketch instances (Fig. 1) to enable this automatic code composition.[1]

We demonstrate the utility and benefits of Sketchovsky over a wide range of settings and a library of sketching algorithms that measure various statistics of interest [11, 14, 17, 19, 20, 21, 27, 28, 29, 32, 44]. Given this basic library, we built an ensemble generator that can create diverse ensembles using a wide range of parameters. We then used the generator to generate tens of thousands of ensembles that we used to measure the resource reduction benefits of Sketchovsky. We measure accuracy results by running four representative ensembles on the Tofino switch processing various inter-ISP packet traces [2]. Compared to the baseline of SketchLib, Sketchovsky reduces the use of hash units by 7~40%, SALUs by 9~45%, and SRAM by 0~7% for the ensembles that have the same flowkey for all sketch instances. Even for the ensembles with randomly picked sketching algorithms and parameters, resource reduction is 3~21% for the hash units, 4~26% for SALUs, and 0~0.4% for SRAM. For the accuracy experiment, we report no accuracy loss for any sketch instances.

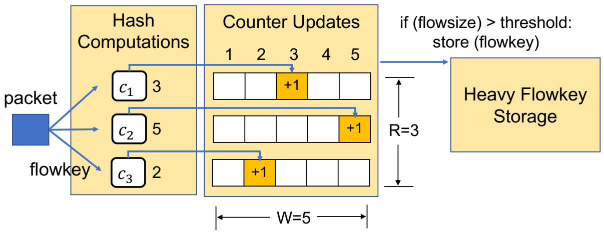


Figure 2: Sketching algorithms have three common workflow steps; hash computations, counter updates, and heavy flowkey storage.

2 Background and Related Work

We begin with background on sketches and their hardware footprints. Then we motivate the need for running an ensemble in practice. We end by explaining why state-of-the-art solutions fall short of effectively supporting an ensemble.

2.1 Sketches and Programmable Switches

Sketches or sketching algorithms on programmable switches are promising to support diverse measurement tasks due to resource efficiency and high accuracy. Sketching algorithms are randomized approximation algorithms designed to measure different statistics with a theoretical guarantee of high accuracy and sub-linear memory space in relation to the number of flows. Thus, sketching algorithms fit well for programmable switches with tight resource constraints. There are sketching algorithms to support diverse statistics for measurement tasks. For example, count-min sketch (CM) [17] can identify heavy hitters, and it can be configured with flowkey definition of srcIP to run Query. 1. HyperLogLog (HLL) [21] can estimate the distinct number of flows, and it can answer Query. 2 with flowkey definition of 5-tuple. There are other sketching algorithms, such as K-ary sketch (KARY) [27] for heavy change detection or UnivMon (UM) [32] for multiple statistics.

Sketching algorithms follow three common steps. We explain these steps with a canonical sketching algorithm count-min sketch [17] (Fig. 2). First, sketching algorithms perform *hash computations*. As each packet arrives, the count-min sketch extracts a flowkey (e.g., 5-tuple) from the packet header. On this key, the count-min sketch computes independent hash functions c_i. Second, using these hash results, sketching algorithms perform *counter updates*. They typically maintain 2D counter arrays, R independent counter arrays with the size of W, thus $R \times W$ counters in total. The hash result of c_i selects a specific column for counter update per row. The count-min sketch is a single-level sketching algorithm meaning that it maintains 2D counter arrays. There is a notion of a multi-level sketching algorithm, which uses multiple levels of 2D counter

[1] Sketchovsky is publicly available at https://github.com/Sketchovsky.

arrays. Third, sketching algorithms need to maintain *heavy flowkey storage*. Sketching algorithms use threshold values to compare against flow size estimates to detect and store heavy flowkeys. While these steps run on the data plane, the control plane periodically reads and resets counter arrays and heavy flowkey storage to compute desired metrics [37].

While the ideas of Sketchovsky can be applied to other programmable switch architectures (as stated in §9), we showcase the effectiveness and benefits of Sketchovsky using Intel's Tofino switch [1]. Tofino is a commercially available programmable switch built on the RMT architecture [13] and the P4 language [12]. Its pipeline stage architecture is equipped with equal resources per stage, and running sketching algorithms on programmable switches requires the use of four key hardware resources. **Hash units** execute hash functions and there are a certain number of hash units per pipeline stage. The hash results of hash units can be used to select a specific column for counter updates or other purposes (§A.1). Each pipeline stage has a fixed amount of **SRAM** that can be used to maintain the state. SRAM is used by counter arrays and heavy flowkey storage. **Stateful ALU (SALU)** is the essential hardware resource that allows one read and one write operation to a register array in SRAM. A counter array for a sketch instance is mapped to a register array. Thus, a sketch instance using R counter arrays requires R SALUs. Storing heavy flowkey also requires SALUs. More **pipeline stages** are needed for more usage of the above three hardware resources. In addition, the notion of dependencies among workflow steps (e.g., *counter updates* must be executed earlier than *heavy flowkey storage*) can contribute to even more pipeline stage usage due to the imbalanced resource allocation.

2.2 Need for Ensemble of Sketch Instances

Network operators need to concurrently run diverse flow-level measurement tasks on programmable switches because the more information operators can get about the network, the more they can make the right management decisions [15, 25, 34, 38, 47, 48, 49, 52]. As concrete examples of measurement tasks, we show two network queries written by Sonata [25], a state-of-the-art query language on programmable switches. Query. 1 can detect heavy hitters based on the sum of packet length in bytes aggregated on flowkey of srcIP. Query. 2 measures the distinct number of 5-tuple flows. Network operators want to concurrently run these measurement tasks as many as possible.

Each such measurement task would entail creating a sketch instance based on a base sketching algorithm (SA) with four configurable parameters (Table 1): (1) **Flowkey** is any combination of packet header fields (e.g., srcIP or 5-tuple); (2) **Flowsize** defines whether the sketch instance keeps track of packet counts or packet bytes; (3) **Epoch** is the measurement time interval; and (4) **Resource Parameters** configure the memory size (e.g., W and R of 2D counter arrays). The network operator should choose resource parameters carefully due to a trade-off between resource use and accuracy.

SI	Base SA	Configurable Parameters			
		Flowkey	Flowsize	Epoch	Res. Param.
s_1	CM	(srcIP)	counts	10s	(3, 1024)
s_2	CM	(dstIP)	bytes	10s	(5, 2048)
s_3	KARY	(srcIP, dstIP)	counts	10s	(4, 4096)
s_4	HLL	(srcIP, srcPort)	-	5min	(1, 2048)
s_5	UM	(5-tuple)	counts	5min	(3, 2048, 16)

Table 1: An example of an ensemble of sketch instances. For resource parameters, (R, W) for single-level and (R, W, level) for multi-level sketching algorithms.

Solution	General	Resource	Accuracy
P4 Composition [22, 23, 30, 42, 51]	✓	X	✓
Per-sketch optimizations [4]	✓	X	✓
Expressive sketches [9, 15, 32, 49]	X	✓	✓
Dynamic resource allocation [3, 36, 50]	✓	✓	X
Sketchovsky (Our system)	✓	✓	✓

Table 2: Existing efforts cannot support a general ensemble of measurement tasks with low resource footprint and high accuracy.

For instance, given Query. 1 above, we can use a count-min sketch instance on the srcIP as flowkey and for Query. 2 we may use HyperLogLog on the 5-tuple. More generally, given the collection of measurement tasks with different configurable parameters (flowkey, flowsize, epoch) and statistics, we will need to concurrently run *an ensemble of sketch instances* in practice. For our work, we assume that the ensemble of sketch instances is given as input; the problem of finding the best ensemble of sketch instances given a collection of measurement tasks is outside the scope of this paper (§9).

2.3 Prior Work and Limitations

We discuss some existing efforts in sketch-based telemetry on programmable switches and why they are insufficient to tackle the ensemble of sketch instances problem (Table 2).

Composing P4 programs. Many P4 code composition works have been recently published for resource optimizations [22, 23, 30, 42, 51]. However, none of them can optimize the sketch ensemble because they did not consider stateful processing, which is at the core functionality of sketching algorithms (e.g., *counter update* step). P4visor [51], Lyra [22], and Cetus [30] focus on optimizations for match-action tables, but they did not consider optimizations for stateful processing, including MicroP4 [42]. Thus, they cannot be used to optimize an ensemble of sketch instances.

Chipmunk [23] seems to be a promising candidate for providing cross-sketch optimizations at first glance because it compiles a program written by Domino language into optimized P4 code with stateful processing optimizations. However, Chipmunk can not compile a full single sketch implementation due to its limited scope. It only supports the update part of the stateful value but does not include the addressing part (e.g., computing hash functions to address the column

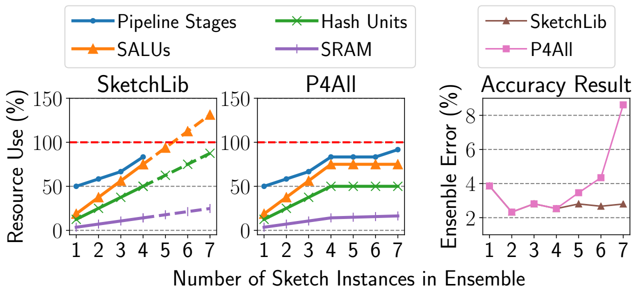


Figure 3: Existing efforts cannot efficiently run the ensemble.

index of counter arrays), which is critical for sketch implementations.

Per-sketch optimizations [4] can be used to implement the sketch ensemble. However, this approach cannot achieve low resource footprints due to *linearly* increasing resource consumption as we run multiple sketch instances.

More expressive sketches. To make improvements, recent advances in sketching theory empower a single sketch instance to support multiple measurement tasks [9, 15, 32, 49]. However, their coverage of the measurement tasks is still far from general (e.g., none of them can support the entropy estimation tasks for two different flowkey definitions).

Dynamic resource allocation. Earlier work has reduced resource use for the ensemble of sketch instances [3, 36, 50]. SCREAM [36] dynamically reduces resource parameters of sketch instances to meet specified minimum accuracy when there are variations in traffic. P4All [3] can be used to reduce the resource parameters of some sketch instances in the ensemble by identifying lower-prioritized sketch instances. FlyMon [50] enables dynamic parameter configuration at runtime (e.g., flowkeys and resource parameters). It essentially offers a time-sharing capability to run a sketch ensemble by switching out active sketch instances. However, all these techniques reduce resource use at the expense of accuracy. In contrast, our work proposes optimizations that reduce resources while maintaining accuracy for all sketch instances in the ensemble.

Quantitative results for existing efforts. We quantitatively show why existing efforts are insufficient. Fig. 3 shows the resource footprint and accuracy results for two approaches, per-sketch optimization (SketchLib) and dynamic resource allocation (P4All). To create the ensembles, we only use the count-min sketch with $(R,W) = (5,8K)$, flowkey of 4-tuple, different measurement epochs, and different flowsize definitions. We use CAIDA traces [2]. For the P4All experiment, we fix the width of counter arrays and reduce the number of rows if necessary to fit the maximum number of SALUs on the Tofino switch. We treat all sketch instances equally in the objective function.

The results in Fig. 3 show that SketchLib cannot support more than four sketch instances. While P4All can support more than four sketch instances by reducing hardware resources, this also reduces the accuracy. In summary, we find that existing techniques cannot achieve both low resource footprint and high accuracy.

3 Sketchovsky Overview

Given that prior work is insufficient, we explore a *complementary* approach to identify and exploit *cross*-sketch optimizations to run an ensemble of sketch instances $\mathcal{S} = \{s_i\}_{i=1}^{N}$. To this end, we present Sketchovsky (Fig. 1), a novel cross-sketch optimization and composition framework. Sketchovsky identifies five cross-sketch optimization building blocks so that resource consumption increases *sub-linearly* in the number of sketch instances with guarantees of no accuracy loss. Sketchovsky uses efficient heuristics to find an effective strategy to combine these building blocks for a given ensemble and implements a module to automatically generates an optimized switch code.

Optimization building blocks (§4). We find that key hardware resources used in each workflow step of sketching algorithms can be *reused across* multiple sketch instances. We identify five optimization building blocks to reduce resource footprint while maintaining accuracy. $\mathrm{O}_{\mathrm{Hash1}}$ and $\mathrm{O}_{\mathrm{Hash2}}$ optimize the first step of hash computations; $\mathrm{O}_{\mathrm{Ctr1}}$ and $\mathrm{O}_{\mathrm{Ctr2}}$ optimize the second step of counter updates, and $\mathrm{O}_{\mathrm{Key}}$ optimizes the third step of heavy flowkey storage. Note that optimizations can be generalized to other hardware (§9). Each O_j has applicable conditions to determine whether O_j can be applied to a subset of sketch instances $\mathbf{S} \subset \mathcal{S}$. Applicable conditions are expressed by *configurable parameters* introduced in (§2.2) (e.g., all $s_i \in \mathbf{S}$ have the same flowkey) and *sketch features*. The notion of sketch features captures the differences among different sketching algorithms in algorithm designs or data structures (e.g., counter array type, counter update type, or whether maintaining heavy flowkeys or not).

Strategy finder (§5). Among many valid strategies for applying five optimization building blocks to different subsets of sketch instances in the ensemble, it is challenging to quickly find the most effective strategy due to the intractably large search space. To solve this problem, we formulate an optimization problem by defining the objective function to minimize hardware resources. Next, we propose an idea of problem decomposition. We show that one large problem can be decomposed into small sub-problems, and separate solutions for sub-problems together produce the overall solution. To detect the validity of a strategy, the strategy finder takes inputs of sketch features (e.g., base sketching algorithm) and configurable parameters (e.g., flowkey and flowsize) for $\mathcal{S}$ as in Table 1. Optimization building blocks can be applied to a subset $\mathbf{S} \subset \mathcal{S}$ only if $\mathbf{S}$ satisfies the applicable conditions.

Auto-code composition (§6). Manually translating a strategy into an optimized code is challenging because the strategy contains information about the complicated interplay among multiple optimization building blocks and a set of sketch

Workflow Step	Optimizations	Reduction	Overhead
Hash Computations	HASHREUSE (O_{Hash1})	Hash unit	-
	HASHXOR (O_{Hash2})	Hash unit	Pipe Stages
Counter Updates	SALUREUSE (O_{Ctr1})	SALU,SRAM	-
	SALUMERGE (O_{Ctr2})	SALU	SRAM
Heavy Flowkey Storage	HFSREUSE (O_{Key})	SALU	Pipe Stages CP Comp

Table 3: Relationships among workflow steps, optimizations and resource reductions. CP Comp means Control Plane Computation, and Pipe Stages means Pipeline Stages.

Conditions	O_{Hash1}	O_{Hash2}	O_{Ctr1}	O_{Ctr2}	O_{Key}
Sketch Features					
C1. Same counter array type			✓	✓	
C2. Same counter update type			✓		
C3. Track heavy flowkey					✓
Configurable Parameters					
C4. Same flowkey definition	✓	*	✓	✓	
C5. Same flowsize definition			✓		
C6. Same measurement epoch			✓		

Table 4: Applicable conditions for five optimization building blocks.

instances. We build an auto-code composition that automatically translates a given strategy into optimized sketch code. This relieves the burden of manual work of network operators. Given the output of the strategy finder and a set of sketch P4 codes for $\mathcal{S}$, we generate a unified and optimized P4 program.

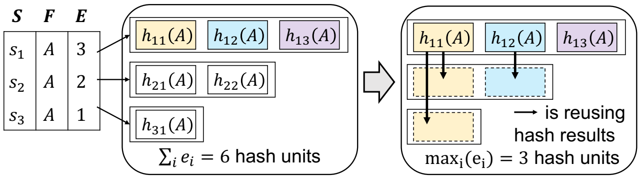
4 Optimization Building Blocks

Given $\mathcal{S} = \{s_i\}_{i=1}^N$, we identify five cross-sketch optimization building blocks (two for hash, two for counters, and one for flowkey storage) that can apply to a given set of sketch instances $\mathbf{S} = \{s_i\}_{i=1}^n \subset \mathcal{S}$. Table 3 summarizes relationships among workflow steps, optimizations and resource reductions. For each optimization, we explain the key idea, the conditions under which it applies, and its implications for resource use and accuracy. Table 4 summarizes applicable conditions to validate whether each optimization can be applied to $\mathbf{S} \subset \mathcal{S}$.

4.1 Hash Computations

To optimize the workflow step of *hash computations* (§2.1), we have two optimizations HASHREUSE (O_{Hash1}) and HASHXOR (O_{Hash2}). Hash unit refers to the hardware resource on programmable switches to execute hash functions. Hash result is the outcome hash value of the hash unit.

HASHREUSE (O_{Hash1}) Reusing hash results. If a set of sketch instances use the same definition of flowkey (e.g., srcIP), we can reuse hash results to reduce the usage of hash units. We explain this optimization using an example in Fig. 4. Assume we have a set of sketch instances $\mathbf{S} = \{s_i\}_{i=1}^n$ with a required set of independent hash results $\mathbf{E} = \{e_i\}_{i=1}^n$ and flowkey definition $\mathbf{F} = \{f_i\}_{i=1}^n$, which means that a sketch instance s_i needs e_i number of hash results based on flowkey f_i. Without optimization, $\sum_i e_i$ hash units are used. However, we can reuse hash results, and we can reduce the allocation of hash units to $\max_i(e_i)$ on the hardware as in Fig. 4.



S F E
s_1 A 3
s_2 A 2
s_3 A 1
$h_{11}(A)$ $h_{12}(A)$ $h_{13}(A)$
$h_{21}(A)$ $h_{22}(A)$
$h_{31}(A)$
$\sum_i e_i$ = 6 hash units
$h_{11}(A)$ $h_{12}(A)$ $h_{13}(A)$
⟶ is reusing hash results
$\max_i(e_i)$ = 3 hash units

Figure 4: HASHREUSE (O_{Hash1}) reduces hash units by reusing hash results. A small box with $h_{seed}(flowkey)$ indicates one hash unit allocation.

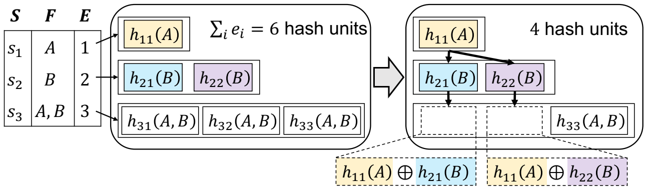


Figure 5: HASHXOR (O_{Hash2}) reduces hash units by using XOR.

Applicability: Regardless of any sketching algorithms, we can apply this optimization as long as $\mathbf{S}$ uses the same flowkeys. We denote this as (C4) in Table 4.

Implication: Allocation of $\max_i(e_i)$ hash units is sufficient to preserve the accuracy of $\mathbf{S} = \{s_i\}_{i=1}^n$. The accuracy of sketch instances is closely related to hash independence among hash results. To implement hash independence in practice, randomly picked hash seeds are used; $h_{seed1}(A)$ and $h_{seed2}(A)$ are independent if $seed1 \neq seed2$. For a single sketch instance, hash independence among hash results is required. A key insight here is that hash independence is not required *across* sketch instances. Thus we can reuse $\max_i(e_i)$ hash results across sketch instances in a way that all hash results within any single sketch instance s_i are independent (e.g., in Fig. 4).

HASHXOR (O_{Hash2}) Less hashing, same performance with XOR-based reconstruction. We can reduce hash units even for a set of sketch instances with different flowkeys by leveraging XOR operations. We explain this optimization using an example in Fig. 5 where $\mathbf{S} = \{s_1, s_2, s_3\}$ and $\mathbf{F} = \{\{A\}, \{B\}, \{A, B\}\}$ and $\mathbf{E} = \{1, 2, 3\}$. A and B are different packet headers, such as A = srcIP and B = dstIP. We can reduce allocation of hash units by reconstructing independent hash results for s_3 as follows because $\{A, B\} = \{A\} \cup \{B\}$.

$$h_{31}(A,B) = h_{11}(A) \oplus h_{21}(B) \tag{1}$$

$$h_{32}(A,B) = h_{11}(A) \oplus h_{22}(B) \tag{2}$$

Note that XOR-based reconstructed hash results $h_{31}(A,B)$ and $h_{32}(A,B)$ are independent because $h_{21}(B)$ and $h_{22}(B)$ are independent. For arbitrary e_1 and e_2, we can reconstruct $e_1 \times e_2$ independent hash results for s_3.

Applicability: This optimization HASHXOR (O_{Hash2}) can be applied if $\mathbf{S}$ and $\mathbf{F}$ meet the following condition.

$$\text{For } \mathbf{S} \in \mathcal{S}, |\mathbf{S}| = 3 \text{ and } f_1 \cup f_2 = f_3 \tag{3}$$

This optimization can be applied as long as a set of sketch instances satisfies (3). Thus, we mark (*) at (C4) in Table 4 for O_{Hash2}.

Implications: This idea of XOR-based hash reconstruction is proven pairwise independent and has already been used in other contexts [26, 43]. Thus, accuracy will not be compromised, and our evaluation result confirms this. As a minor side effect, more pipeline stages might be needed by adding XOR operations in the sketch workflow. However, we will see in the evaluation that the impact of this overhead is small.

4.2 Counter Update

To optimize the second workflow step of *counter updates*, we have SALUREUSE (O_{Ctr1}) and SALUMERGE (O_{Ctr2}).

SALUREUSE (O_{Ctr1}) Reusing counter arrays (rows) across sketch instances. If all sketch instances in **S** meet certain applicable conditions, we can reuse counter arrays to reduce both SALUs and SRAM. We first see how this optimization works by looking at an example in Fig. 6, and we will describe applicable conditions later. Suppose $\mathbf{S} = \{s_1, s_2, s_3\}$ satisfies applicable conditions of O_{Ctr1} and $\mathbf{C} = \{(r_i, w_i)\}_{i=1}^{n}$ represent that s_i has r_i number of counter arrays with width w_i. Then, instead of updating three different sets of counter arrays for three sketch instances in the data plane, we can update only one set of counter arrays. Then, in the control plane, one set of counter arrays can be used to compute statistics for all three sketch instances. The way we compute the row and width of counter arrays for reuse is represented by **W**:

$$\mathbf{W} = \{w_j^*\}_{j=1}^{\max_i(r_i)} \text{ where } w_j^* = \max_i \{w_i | r_i \geq j\} \tag{4}$$

W represents width w_j^* per j-th counter array for reuse. Note that **W** can have different widths across counter arrays, and it does not affect the functionality of sketching algorithms. We can see that **W** has $\max_i(r_i)$ rows. Thus, we can reduce SALUs from $\sum_i r_i$ to $\max_i(r_i)$. Moreover, SRAM usage is reduced from $\sum_i r_i w_i$ to $\sum_j w_j^*$ and we show $\sum_i r_i w_i - \sum_j w_j^* \geq 0$ in §B.1. While the discussion focused on single-level sketch instances, the same idea also applies to multi-level sketch instances.

Implication: If we compare resource parameters (r_i, w_i) of any sketch instance s_i to counter arrays for reusing **W**, **W** has the same or larger width. As a result, all sketch instances are guaranteed to achieve the same or improved accuracy.

Applicability: Applicable conditions for this optimization use two sketch features. The first sketch feature is **counter array type**. Sketching algorithms have different types of counter arrays; the single-level (SL) type has 2D counter arrays, and the multi-level (ML) type has multiple levels of 2D counter arrays. The second sketch feature is **counter update type**. Sketching algorithms have different ways of updating counters. It can be bitmaps (BITMAP) or integer counters that only add values (COUNTER). Refer §A.1 for a full list of five counter update types. $\mathbf{S} \subset \mathcal{S}$ must satisfy five conditions to apply this optimization (Table 4): the same counter array type, the same counter update type, the same flowkey, the same flowsize, and the same epoch (C1, C2, C4, C5, C6).

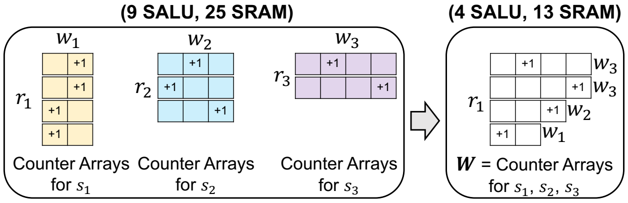


Figure 6: SALUREUSE (O_{Ctr1}) reuses counter arrays.

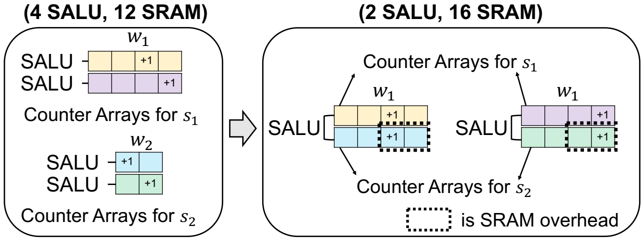


Figure 7: SALUMERGE (O_{Ctr2}) reduces SALUs by making SALUs update two counter arrays simultaneously.

SALUMERGE (O_{Ctr2}). Combining two counter updates into one SALU allocation. Leveraging the full capability of the underlying hardware resources can help resource reduction of **S**. We observe that SALU can update two registers addressed in the same index and we can leverage this feature to update two counter arrays simultaneously. As a result, we can reduce SALUs by up to 2x. We explain this optimization by using an example in Fig. 7. We have two sketch instances with two counter arrays each, and we originally needed four SALUs. Then, we can make SALUs update two counter arrays simultaneously and reduce SALUs from 4 to 2.

We find two rules in the Tofino switch for a SALU to update two counter arrays. (R1) derives applicable conditions, and (R2) incurs SRAM overhead.

- (R1) Column indexes for counter updates are the same
- (R2) Two counter arrays have the same width

Applicability: (R1) derives two applicable conditions (C1, C4). If sketch instances use the same counter array type (C1) (e.g., sketch instances are either all single-level or all multi-level) and use the same definition of flowkey (C4), we can apply this optimization. Because flowkeys are the same, we can leverage HASHREUSE (O_{Hash1}), and SALU can update two counter arrays using the reused hash result for the column index. If flowkeys are different, then column indexes for the counter update can not be the same, which will violate (R1). Note that we should not let SALU update two counter arrays within a sketch instance because updating the same column index will lose hash independence and degrade accuracy.

Implication: This optimization can incur the additional SRAM overhead due to (R2). Suppose we have two sketch instances $\{s_1, s_2\}$ with two counter arrays each as in Fig. 7 with width of $\{w_1, w_2\}$ s.t. $w_1 > w_2$. Suppose we can apply the optimization to $\{s_1, s_2\}$. Then, we should pick the longer width w_1 for counter arrays to preserve the accuracy of both $\{s_1, s_2\}$. As a result, the accuracy will be maintained (e.g., for s_1) or improved (e.g., for s_2). However, this will incur an SRAM overhead of $w_2 - w_1$ for s_2, as marked in Fig. 7. Despite the SRAM overhead, we argue that this optimization is still effective and practical for three reasons. First, increased SRAM is not wasted but will improve accuracy. Second, the overall SRAM overhead is bounded by 2x ($\frac{2\,max(w_1,w_2)}{w_1+w_2} \leq 2$). Third, SRAM is not the imminent bottleneck as we will see in the evaluation.

4.3 Heavy Flowkey Storage

To optimize the third workflow step of *heavy flowkey storage*, we have one optimization HfsReuse (O_{Key}).

HfsReuse ($\mathbf{O_{Key}}$). Reusing heavy flowkey storage across sketch instances. A large portion of sketching algorithms store heavy flowkeys in the switch data plane [9, 14, 16, 17, 27, 32, 41]. We can reduce the usage of SALUs (the hardware used for memory accesses) by reusing heavy flowkey storage across sketch instances. If multiple sketch instances have the same definition of flowkey (e.g., srcIP), we can store heavy flowkey in one heavy flowkey storage to save SALUs. We can generalize this idea to sketch instances with different definitions of flowkey using the notion of *union-key*. Suppose we have two sketch instances $\mathbf{S} = \{s_1, s_2\}$ with two different flowkey definitions $\mathbf{F}$ ={{srcIP}, {dstIP}}. Then, instead of maintaining two heavy flowkey storage, we use one flowkey storage using union-key of {srcIP, dstIP} where union-key can be computed by ($UK = \cup_i f_i$). Then, for a given packet, if *either* {srcIP} is identified as a heavy flowkey for s_1 *or* {dstIP} is identified as a heavy flowkey for s_2. We store {srcIP, dstIP} of the given packet in the heavy flowkey storage.

We can do a further optimization to reduce memory usage of heavy flowkey storage. Suppose $\mathbf{S} = \{s_1, s_2\}$ and $\mathbf{F}$ ={{srcIP}, {dstIP}}. For a given packet, if {srcIP} is identified as a heavy flowkey whereas {dstIP} is not, we store {srcIP, 0} so that the control plane knows this flowkey is only for s_1. To generalize this idea to multiple sketch instances, we can compute a *conditional union-key* $UK_C = \cup_j f_j$where (flow size estimate)$_j$ > $threshold_j$ and set 0 to ($UK - UK_C$) when we store heavy flowkey into the storage.

Applicability: We can apply this optimization to a set of sketch instances $\mathbf{S}$ if all sketch instances in $\mathbf{S}$ tracks heavy flowkeys (C3) as in Table 4. For different measurement epochs, we can compute the greatest common denominator (GCD) among all epochs, and the control plane can retrieve heavy flowkeys every time period of GCD. For example, if there are sketch instances with 10s, 20s, and 30s measurement epochs, the control plane retrieves heavy flowkeys for every 10s, and we can reconstruct heavy flowkeys for sketch instances of 20s and 30s.

Implication: By storing fine-grained heavy flowkeys by union-key, the control plane can retrieve heavy flowkeys for individual sketch instances by aggregation without missing any heavy flowkeys. This optimization incurs small additional computations on the switch control plane. However, this overhead does not affect the overall performance because this control plane computation is not on the critical path to provide measurement results. While the switch data plane updates the counter arrays, the switch control plane can independently execute heavy flowkey aggregation on the CPU. Another small overhead of the pipeline stage can occur, but we will see in the evaluation that the impact is small.

5 Strategy Finder

In the previous section, we proposed five optimizations $\{\text{O}_j\}_{j \in \{\text{Hash1, Hash2, Ctr1, Ctr2, Key}\}}$ and their applicable conditions to a subset of sketch instances $\mathbf{S} = \{s_i\}_{i=1}^{n} \subset \mathcal{S}$. In this section, we aim to develop a strategy finder that partitions $\mathcal{S}$ into the best applicable subsets so that five optimization building blocks can produce the maximum benefit for any given ensemble $\mathcal{S}$.

5.1 Problem Formulation

We formulate an optimization problem to find the optimal strategy. We consider partitions of $\mathcal{S}$ because each optimization O_j is applied to disjoint subsets of $\mathcal{S}$. Suppose $\mathcal{P_S} = \{P_k | P_k$ is kth partition of the set $\mathcal{S}\}$ is a set containing all partitions of the set $\mathcal{S}$ where $P_k = \{\mathbf{S}_l \subset \mathcal{S} | \dot{\bigcup}_l \mathbf{S}_l = \mathcal{S}\}$. The goal is to find the optimal strategy X^*, which minimizes hardware resources while satisfying the applicable conditions:

$$\min_{X} HwResource(X) \tag{5}$$

$$\text{s.t.} \sum_{k=1}^{|\mathcal{P_S}|} x_{jk} = 1, \forall j \in \{\text{Hash1, Hash2, Ctr1, Ctr2, Key}\} \tag{6}$$

$$Valid(X) = 1 \tag{7}$$

The decision variable is $X = \{X_j\}_{j \in \{\text{Hash1, Hash2, Ctr1, Ctr2, Key}\}}$. X_j selects a partition $P_k \in \mathcal{P_S}$ for O_j so that O_j is applied to all subsets $\in P_k$. To express this, we define $X_j = \{x_{jk} | x_{jk} \in \{0,1\}\}_{k \in \{1,\ldots,|\mathcal{P_S}|\}}$ and $x_{jk} = 1$ if P_k is selected. Note that (6) makes X_j pick only one partition P_k for O_j. About constraint (7), we use $Valid(X) \in \{0,1\}$ to denote whether strategy X is *valid* or not. X is valid if all subsets $\in P_k$ satisfy the applicable conditions of O_j for $\forall$ j. It is assumed that applicable conditions are met for the subset $\mathbf{S} \subset \mathcal{S}$ containing a single sketch instance s.t. $|\mathbf{S}| = 1$. For objective function (5), we aim to find a strategy X^* that minimizes hardware resource among all valid strategies X. To model this objective function $HwResource(X)$, we use the linear combination of four key resource usage:

$$LinearComb(X,R) = \sum_{r \in R} a_r \cdot resource_r(X) \quad (8)$$

$$R = \{\text{SALU, HashUnit, SRAM, PipelineStage}\} \quad (9)$$

Network operators can use (8) and customize the objective function by choosing different coefficient sets $\{a_r\}_{r \in R}$ for their preference. For example, suppose network operators desire to reduce SRAM more than other resources to run the ensemble with memory-intensive applications, they can increase the weight for a_{SRAM} in (8).

5.2 Challenges

We face three challenges in formulating and solving the optimization problem.

C-1. Large search space. We have a large search space for enumeration because the number of possible partition $|\mathcal{P}_{\mathcal{S}}|$ increases exponentially as the number of sketch instances $|\mathcal{S}|$ increases [8]. Even after we consider constraint (6), the decision variable X has $|\mathcal{P}_{\mathcal{S}}|^5$ combinations because X selects five partitions among $\mathcal{P}_{\mathcal{S}}$. This large search space makes finding the optimal solution X^* become intractable. In practice, operators often need to reconfigure sketch ensembles, and waiting for the optimization to complete may end up being on the critical path [50].

C-2. Modeling the valid function. It is hard to define $Valid(X)$ due to the dependencies among optimizations. Specifically, there exist dependencies between O_{w1} and O_{w2} for $w \in \{\text{Hash, Ctr}\}$ because they are applied to the same workflow steps. Thus, it is unclear whether a sketch instance s_i can be benefited from O_{w1} and O_{w2} at the same time. Further, if they can, then it is also unclear how to figure out the relationship between X_{w1} and X_{w2} to detect the validity of X to define $Valid(X)$.

C-3. Modeling the objective function. We find that computing $LinearComb(X,R)$ takes a long time because accurately measuring pipeline stage usage requires the compilation of an optimized P4 code by applying strategy X. The execution time for $resource_{pipeline\_stage}(X)$ takes several minutes. This delay will significantly impede the search process, and finding a solution X^* can become even more intractable.

5.3 Our Approach

Next, we reformulate the problem and show that finding the optimal strategies for each O_j will create the overall solution X^*. This reduces search space significantly and makes the problem tractable.

Excluding pipeline stage from the objective function. To handle (C-3), we make a pragmatic choice of excluding the pipeline stage from the objective function. We use $LinearComb(X,R')$ as objective function where $R' = \{\text{SALU, HashUnit, SRAM}\}$. $resource_r(X)$ for $r \in R'$ can be quickly computed because X contains information about the number of reused or XOR-reconstructed hash units, reused or co-located counter arrays, and reused heavy flowkey storage. The impact of this decision cannot be measured because the optimal objective function is unknown and difficult to define. However, we can still identify effective solutions that can yield significant benefits in practice because there is a correlation between the resource reduction on R' and the pipeline stage reduction (§8).

Search space decomposition *across* workflow steps. To overcome the challenge of large search space (C-1), we can decompose the optimization problem into three sub-problems, and solution X^* can be achieved by solving sub-problems separately. Specifically, we decompose the decision variable X into three groups corresponding to three workflow steps:

$$X_{Hash} = \{X_{Hash1}, X_{Hash2}\}, X_{Ctr} = \{X_{Ctr1}, X_{Ctr2}\}, X_{KEY} = \{X_{Key}\}$$

Then, we can also decompose the valid function and the objective function as follows:

$$X = \cup_{w \in \{\text{Hash, Ctr, KEY}\}} X_w \quad (10)$$

$$Valid(X) = \prod_{w \in \{\text{Hash, Ctr, KEY}\}} Valid(X_w) \quad (11)$$

$$HwResource(X) = \sum_{w \in \{\text{Hash, Ctr, KEY}\}} HwResource(X_w) \quad (12)$$

This problem decomposition is possible for two reasons. First, although there are dependencies in terms of applicability within X_w, there are no dependencies *across* X_w because optimizations are independently applied to different workflow steps. Thus, $Valid(X)$ can be achieved by multiplication of decomposed $Valid(X_w)$ as in (11). Second, $HwResource(X)$ can be achieved by summation of decomposed $HwResource(X_w)$ as in (12). Without the idea of excluding pipeline stage usage, this linearity property (12) does not hold because measuring pipeline stage usage must consider the overall table dependency graph (TDG) among workflow steps (X_w). As a result, a solution X^* can be achieved by $X^* = \{X^*_{Hash}, X^*_{Ctr}, X^*_{KEY}\}$ where X^*_w is a solution of each sub-problem for $w \in \{\text{Hash, Ctr, KEY}\}$ as follows:

$$\min_{X_w} HwResource(X_w) \text{ s.t. } Valid(X_w) = 1 \quad (13)$$

Two-step enumeration for X_{Hash} and X_{Ctr}. Although we can decompose $Valid(X)$ into three $Valid(X_w)$ as in (11), it is still unclear how to realize $Valid(X_w)$ for $w \in \{\text{Hash, Ctr}\}$ because there exist dependencies between O_{w1} and O_{w2}. We can solve this problem using an enumeration technique that efficiently explores the search space. Suppose the enumeration does not miss out on *valid* X_w (s.t. $Valid(X_w) = 1$) while efficiently skips *invalid* X_w. In that case, it will help to solve not only the challenge (C-2) of modeling an valid function but also the challenge (C-1) by reducing search space. To achieve this, we develop a two-step enumeration technique as in Alg. 1.

Algorithm 1 TwoStepEnumeration

```
1:  procedure TwoStepEnumeration($\mathcal{S}, w$)
2:      $\mathcal{P}_\mathcal{S} = \{P_k | P_k$ is k$^{th}$ partition of the set $\mathcal{S}\}$
3:      $min \leftarrow INTMAX$
4:      for $X_{w1}$ s.t. selected $P_{w1} \in \mathcal{P}_\mathcal{S}$ is valid do
5:          for $X_{w2}$ s.t. $P_{w1} \le P_{w2} \in \mathcal{P}_\mathcal{S}$ do
6:              $X_w \leftarrow \{X_{w1}, X_{w2}\}$
7:              $P_{w12}$ = NestedPartition($P_{w1}, P_{w2}$)
8:              if $P_{w12}$ is valid then
9:                  if $min > HwResource(X_w)$ then
10:                     $min \leftarrow HwResource(X_w)$
11:                     $X_w^* \leftarrow X_w$
12:     return $X_w^*$
```

We explain this algorithm by both cases of $w \in \{\text{Hash, Ctr}\}$. Let's first see an example for w = Hash, HashReuse (O_{Hash1}) and HashXOR (O_{Hash2}). Suppose we have five sketch instances $\mathcal{S} = \{s_i\}_{i=1}^5$ with flowkey definition $\mathbf{F}$ ={{srcIP}, {srcIP}, {dstIP}, {srcIP, dstIP}, {srcPort}}. Then the algorithm enumerates all valid P_{w1} at line 4 in Alg. 1. $P_{w1} = \{\{s_1,s_2\},\{s_3\},\{s_4\},\{s_5\}\}$ can be picked because $\{s_1,s_2\}$ have the same flowkey so that we can reuse hash results to reduce hash units. Next, given picked P_{w1}, it enumerates P_{w2} s.t. $P_{w1} \le P_{w2}$[2] to create *nested partition* P_{w12} using P_{w1} and P_{w2}. If $P_{w2} = \{\{s_1,s_2,s_3,s_4\},\{s_5\}\}$ is picked, then the nested partition is $P_{w12} = \{\{\{s_1,s_2\},\{s_3\},\{s_4\}\},\{\{s_5\}\}\}$. To see the validity of P_{w12}, check whether all subsets in P_{w12} satisfy applicable conditions of O_{Hash2} at line 8. Picked P_{w12} is valid because subset $\mathbf{S} = \{\{s_1,s_2\},\{s_3\},\{s_4\}\} \in P_{w12}$ satisfies applicable conditions of $|\mathbf{S}| = 3$ and {srcIP} ∪ {dstIP} = {srcIP, dstIP} as in (3) at §4.1. Note that $\{s_1,s_2\}$ is handled as if it is a single sketch instance with a flowkey of {srcIP}.

Interestingly, the same algorithm works for w = Ctr, SaluReuse (O_{Ctr1}) and SaluMerge (O_{Ctr2}). First, the algorithm enumerates P_{w1} where all subsets in P_{w1} satisfy applicable conditions (C1, C2, C4, C5, C6) of O_{Ctr1}. For picked P_{w1}, O_{Ctr1} is applied to all subsets $\in P_{w1}$, meaning that each subset has one set of counter arrays for reuse $\mathbf{W}$ as discussed as in (3) at §4.2. Then each subset can be handled as if it is a single sketch instance with counter arrays configured with $\mathbf{W}$. Next, we can detect the validity of nested partition P_{w12} using the applicable conditions (C1, C4) of O_{Ctr2} at line 8 in Alg. 1.

HFSReuse (O_{Key}) does not need this two-step enumeration. The solution for O_{Key} is one subset $\mathbf{S}$ containing all sketch instances that track heavy flowkey because this will minimize the hardware resource usage.

Search space decomposition *within* workflow steps. Although two-step enumeration reduces search space by picking P_{w1} first and then P_{w2} such that $P_{w1} \le P_{w2}$, this enumeration technique still takes a long time to finish (e.g., more than a day). To this end, we come up with an idea to decompose X_{w1} and X_{w2} by using a greedy heuristic algorithm. Instead of running a nested loop (lines 4-5 in Alg. 1) for finding P_{w1} and P_{w2}, we can first find the optimal P_{w1}^* given $\mathcal{S}$ and then finds P_{w2}^* based on already fixed P_{w1}^*. This greedy heuristic algorithm decomposes the search space of $\{X_{w1}, X_{w2}\}$ into separate $\{X_{w1}\}$ and $\{X_{w2}\}$.

The insight behind this greedy heuristic algorithm comes from the applicability-benefit trade-off between O_{w1} and O_{w2}. O_{w1} is more difficult to apply but has a high resource reduction benefit. O_{w2} is easier to apply but has a low resource benefit. Thus, it makes sense that the algorithm first applies O_{w1} as much as possible, then next applies O_{w2}. We can not prove whether this greedy heuristic algorithm can find the same or close solution compared to the two-step enumeration. However, we empirically show that the overhead of objective function increase is small (e.g., less than 2%) while solving time of the greedy heuristic algorithm is more than three orders of magnitude faster (§D.3).

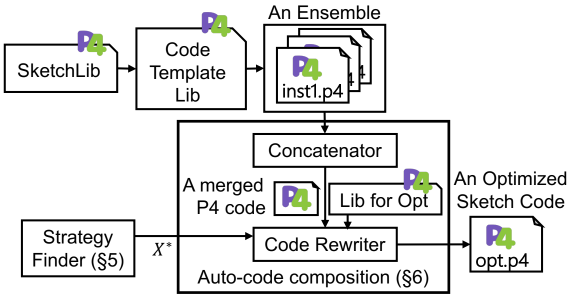


Figure 8: Overview of auto-code composition.

```
/* 1. hash computation step */
s#_h = HashUnit(seed1, FLOWKEY);
s#_value = TCAM_LPM(s#_h);
/* 2. counter update step */
CounterUpdate(seed2, FLOWKEY, WIDTH,
                     SL, PCSA, s1_value);
/* 3. heavy flowkey storage step - no code */
```

Figure 9: Code template example for PCSA.

6 Auto-Code Composition

Using solution X^* from the strategy finder, we need two steps to generate an optimized P4 code for $\mathcal{S}$ as in Fig. 8.

6.1 Sketch P4 Codes and Concatenation

The first step requires network operators to provide N sketch P4 codes that should match with sketch features and configurable parameters for the ensemble $\mathcal{S} = \{s_i\}_{i=1}^N$ (e.g., Table 1).

Code template library. Writing N sketch P4 codes from scratch is a cumbersome task for network operators. An effective way is to provide code templates of the sketching algorithm with which P4 code for a sketch instance can be created. We build code templates for sketching algorithms so that network operators can configure the template with tunable parameters. Fig. 9 shows a code template example of

[2] If every element of partition P_{w1} is a subset of some element of partition P_{w2}, then $P_{w1} \le P_{w2}$. In other words, P_{w1} is finer and P_{w2} is coarser.

one sketching algorithm PCSA [20]. Network operators can fill out placeholders using configurable parameters.

SketchLib. To further simplify the code template, we consider using a common library to write codes for sketch instances, such as SketchLib [4]. The idea of using API calls makes code templates simple and concise. We extend SketchLib to enable the flexible configuration of various sketch features and configurable parameters. For example, API call `CounterUpdate()` in Fig. 9 at line 5 gets any definition of flowkey and any counter update type (e.g., `PCSA` type is used in this example). A full list of extended API calls of SketchLib is in §C.1.

Code concatenation. Finally, the concatenator in Fig. 8 gets the input of N sketch P4 codes created from code templates and concatenates N sketch P4 codes into one merged P4 code.

6.2 Code Rewriting

The second step is code rewriting to translate the selected strategy X^* into optimized code. Code rewriter in Fig. 8 gets three inputs; a merged P4 code from the first step, strategy X^* from the strategy finder, and Library for Optimization (Lib for Opt) that is used to apply SALUMERGE (O_{Ctr2}). Using $X^* = \{X^*_{Hash}, X^*_{Ctr}, X^*_{KEY}\}$, the code rewriter sequentially translates X^*_w to each workflow step in a merged P4 code by rewriting short lines of code. Leveraging the code templates makes the code rewriting process a lot easier. First, a merged sketch P4 code is structured in a way that the code rewriter can easily parse and apply optimizations. Second, the amount of code rewrite is minimized because sketch code templates are concise by using API calls in SketchLib and Lib for Opt.

7 Implementation

Auto-code composition. We use two examples, O_{Hash1} and O_{Hash2} to illustrate how we auto-generate an optimized code. Fig. 10 is the code snippet without optimization and we call it before code. Fig. 11 is the code snippet after applying X^*_{Hash} and we call it after code. We have $\mathcal{S} = \{s_i\}_{i=1}^4$, $\mathbf{F} = \{\{\text{srcIP}\}, \{\text{srcIP}\}, \{\text{dstIP}\}, \{\text{srcIP, dstIP}\}\}$. The before code allocates hash units to generate hash results for each flowkey (lines 3, 7, 10, 13 in Fig. 10). To emulate different logical hash seeds for independence, we configure the hash units with different CRC polynomials in practice. Then, we apply optimizations using a given solution $X^*_{Hash} = \{\{\{s_1, s_2\}, \{s_3\}, \{s_4\}\}\}$, which means the code should reuse {srcIP} for $\{s_1, s_2\}$ and use XOR operation to create a hash result for {srcIP, dstIP} = {srcIP} ⊕ {dstIP}. If we look at line 4 in Fig. 11, the hash result of s_2 reuses the hash result of s_1. Line 6 in Fig. 11 shows XOR-based hash result reconstruction. As a result, the usage of the hash unit is reduced from 4 to 2.

Applying SALUMERGE (O_{Ctr2}) requires new codes for implementing two counter arrays to share one SALU that the before code does not have. Thus, we build a new library (Lib for Opt) shown in Fig. 8 to implement O_{Ctr2} and the code rewriter can use this library for applying O_{Ctr2}. The definition of the API call for Lib for Opt is in §C.1. For other optimizations $\{O_j\}_{j \in \{\text{Hash1, Hash2, Ctr1, Key}\}}$, we do not need new API calls because a simple rewrite is enough for implementing reusing resources (O_{Hash1}, O_{Hash2}, O_{Ctr1}) or XOR operation (O_{Hash2}) (e.g., at line 6 in Fig. 11). As a result, the code rewriter can translate all five optimizations into an optimized code. We show examples of before and after code snippets for O_{Hash1}, O_{Ctr1}, O_{Key} in §C.2.

```
01: // code for s1
02: /* 1. hash computation step */
03: s1_h = HashUnit(seed1, srcIP);
04: ... /* 2. counter update step */
05: ... /* 3. heavy flowkey storage step */
06: // code for s2
07: s2_h = HashUnit(seed2, srcIP);
08: ...
09: // code for s3
10: s3_h = HashUnit(seed3, dstIP);
11: ...
12: // code for s4
13: s4_h = HashUnit(seed4, srcIP, dstIP);
14: ...
```

Figure 10: [Before] HASHREUSE (O_{Hash1}) and HASHXOR (O_{Hash2}).

```
01: // code for s1, s2, s3
02: /* 1. hash computation step */
03: s1_h = HashUnit(seed1, srcIP);
04: s2_h = s1_h;
05: s3_h = HashUnit(seed3, dstIP);
06: s4_h = s1_h ^ s3_h;
07: ...
```

Figure 11: [After] HASHREUSE (O_{Hash1}) to $\{s_1, s_2\}$ and HASHXOR (O_{Hash2}) to $\{\{s_1, s_2\}, s_3, s_4\}$.

Strategy finder. One minor issue here is that while implementing SALUMERGE (O_{Ctr2}) on the Tofino switch, we face a known problem of sketch inaccuracy caused by the counter read and reset delays [37]. To address this, we add one more applicable condition of the same epoch (C6) to O_{Ctr2}.

8 Evaluation

Our extensive set of experiments shows that Sketchovsky can achieve both a low resource footprint and high accuracy simultaneously.

8.1 Experimental Setup

Testbed. We evaluate Sketchovsky on a local testbed with an Edgecore Wedge 100BF Tofino-based programmable switch and a server equipped with dual Intel Xeon Silver 4110 CPUs, 128GB RAM, and a 100Gbps Mellanox CX-4 NIC connected to the switch. We use the P4-16 version with the Tofino SDE version of 9.5.1.

Sketching algorithms. We use eleven sketch algorithms that measure six different statistics.[3] Although Bloom filter (BF) is not a sketching algorithm, we include BF because it also follows the workflow steps of sketching algorithms, and Sketchovsky can optimize it.

Four ensemble types. We use four types of ensembles that network operators would practically consider using. In ensembles of (Type 1. Same Sketch), (Type 2. Same Flowkey), and (Type 3. Same Epoch), all sketch instances in the ensemble use the same sketching algorithm, flowkey, and epoch, respectively. For (Type 4. Random), sketch instances in an ensemble are picked randomly.

Ensemble Generator. To create four types of ensembles, we build an ensemble generator that takes two inputs; (1) the ensemble type and (2) the number of sketch instances for the ensemble. Using these two inputs, the ensemble generator randomly picks sketching algorithms and assigns configurable parameters from a large pool of candidates. A full list of candidates for parameters is in §D.1. The ensemble generator does not allow any two sketch instances in an ensemble to have the same statistic, flowkey, flowsize, and epoch.

Metrics. We use three metrics for accuracy: (1) Relative Error (RE): $\frac{|True-Estimate|}{True}$, where *True* is the ground truth value and *Estimate* is the estimated value. We use this metric for sketching algorithms for cardinality and entropy. (2) Average Relative Error (ARE): $\frac{1}{k}\sum_{i=1}^{k}\frac{|f_i-\hat{f}_i|}{f_i}$, where k means the top k heavy flows. f_i is the actual flow size for flow i, and $\hat{f}_i$ is the estimated flow size from the sketch instances. This metric is used to evaluate the accuracy of the heavy hitter and heavy change detection. We use k=50. (3) Weighted Mean Relative Difference (WMRD) is used for MRAC [28].

For resource reduction, we use two metrics: (1) Resource Usage (RU): $\frac{\text{Used}}{\text{Available}}$, where Used is the amount of resource used for the ensemble and Available is the total amount of available resource on the switch; and (2) Resource Reduction (RR): $\frac{\text{RU (before) - RU (after)}}{\text{RU (before)}}$, where RU (before) is the amount of used resource before applying optimizations of Sketchovsky and RU (after) is the amount of used resource after optimization.

8.2 Accuracy

We show that Sketchovsky does not degrade accuracy and sometimes improves accuracy. For this experiment, we picked four ensembles from each ensemble type. A full list of base sketch algorithms and configurable parameters for picked ensembles is in §D.2. Given each ensemble as input, we generate sketch P4 codes for the Tofino switch both before and after we use Sketchovsky for optimization. All five optimizations are enabled. We then run sketch P4 codes for (four picked ensembles) × (before and after optimizations) on the Tofino switch and compare the accuracy of the sketch instances. We use five traffic workloads of inter-ISP packet traces collected on different dates.[4] For each traffic workload, we send ten 60s packet traces from a directly connected server to the Tofino switch using tcpreplay at full speed.

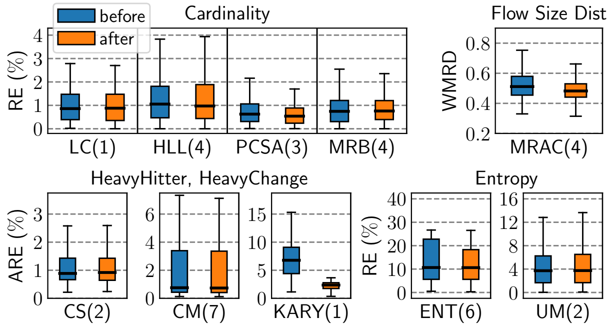


Figure 12: Overall accuracy evaluation.

Fig. 12 shows the overall accuracy results. We grouped sketch instances into four different statistics based on the sketching algorithm used. The X-axis in Fig. 12 shows the number of sketch instances with the same sketching algorithm (e.g., HLL(4) means there are four sketch instances using the sketching algorithm of HLL). The Y-axis shows the quartiles of errors for sketch instances. We see that none of the sketch instances lose accuracy after optimization. In fact, we observe some accuracy improvements. Thanks to O_{Ctr1}, counter arrays of KARY are increased from 1 to 3, and the error is reduced. In addition, we do not miss any heavy flowkeys both before and after optimization. Because BF does not produce measurement results, the accuracy result for two BF sketch instances is not shown.

8.3 Resource Reduction

Sketchovsky makes infeasible ensembles feasible. We show the resource reduction benefits of using Sketchovsky. For this experiment, we generate a total of 400 ensembles of sketch instances; (four ensemble types) × (10 different numbers of sketch instances from 2, 4, ..., 20) × (10 different ensembles). Then, we run Sketchovsky to produce 400 sketch P4 codes both before and after optimization.[5] Next, we compile the codes using the Tofino compiler to check the feasibility. To make the experiment more realistic, we append codes for L2 switching, L3 routing, and access control list (ACL) to all of the before and after optimization codes. L2 and ACL consume 55% of on-switch SRAM in total, and L3 uses 63% of TCAM.

[3] Linear counting (LC) [44], HyperLogLog (HLL) [21], PCSA [20], multi-resolution bitmap (MRB) [19] measure cardinality. Count-sketch (CS) [14], count-min sketch (CM) [17] can detect heavy hitters, and K-ary sketch (KARY) [27] can detect heavy change. Entropy sketch (ENT) [29] measures entropy, MRAC [28] measures flow size distribution (FSD). UnivMon (UM) [32] can measure general statistics. Bloom filter (BF) [11] can do the membership test. A full list of sketching algorithms that we used for our experiments with sketch features is in §D.1.

[4] We use five CAIDA backbone traces captured in 3/20/14 Sanjose, 6/19/14 Sanjose, 1/21/16 Chicago, 5/17/18 NYC, and 8/16/18 NYC [2]

[5] We use count-min sketches to create ensembles for (Ensemble Type 1) because it is one of the most popular and widely-used sketching algorithms.

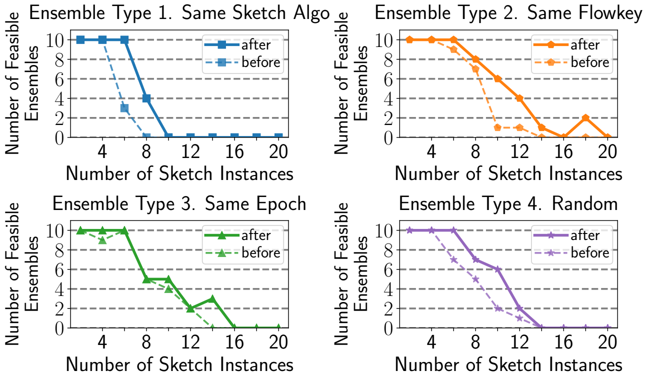


Figure 13: Feasibility comparison of ensembles before vs after.

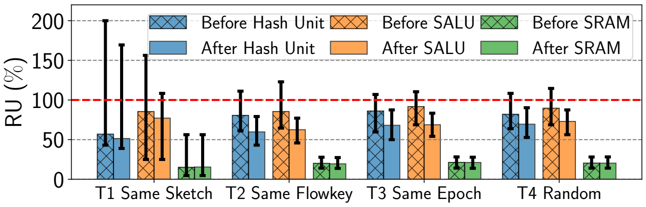


Figure 14: Resource usage comparison before vs after for the number of sketch instances = 12.

The X-axis in Fig. 13 is the number of sketch instances in the ensemble. The Y-axis is the number of feasible ensembles among ten ensembles per different number of sketch instances. The result shows that 42 ensembles that were previously infeasible become feasible with Sketchovsky. For example, if we look at "Ensemble Type 1" and "6 sketch instances", only 3 out of 10 ensembles were feasible before optimization. However, all ten ensembles become feasible after optimization. Note that the pipeline stage overhead of O_{Hash2} and O_{Key} does not negatively impact feasibility after applying them.

Resource usage before and after optimization. Fig. 14 shows the use of individual resources before and after optimization. Using the ensemble generator, we generated 1200 ensembles; (four ensemble types) × (300 different ensembles). Each ensemble has 12 sketch instances. Because some ensembles are not feasible on the Tofino switch because of the limited number of stages, we calculated resource use using the strategy finder so we are not limited by, and do not show, pipeline stages. We cross-checked the resource use between the strategy finder and the Tofino compiler for feasible ensembles. Each bar in Fig. 14 shows the median value, and the error line shows the 10% and 90% percentile among 300 ensembles. The red-dotted line shows the total available resources on the switch, so values above the red line represent infeasible ensembles. Fig. 14 visually shows how previously infeasible ensembles become feasible. SRAM usage of heavy flowkey storage before and after optimization is similar because heavy flowkeys overlap across sketch instances.

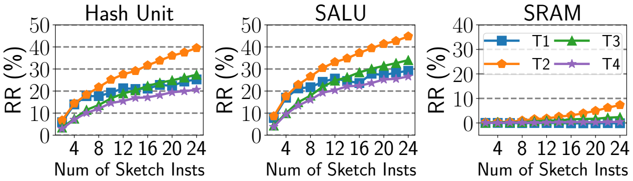


Figure 15: Resource reduction result.

	Resources	**Total**	O_{Hash1}	O_{Hash2}	O_{Ctr1}	O_{Ctr2}	O_{Key}
Type 1	Hash Unit	21.3	3.1	0.1			18.1
Same	SALU	25.7			-	0.8	24.9
Sketch	SRAM	-0.02			-	-0.02	
Type 2	Hash Unit	27.6	10.4	-			17.2
Same	SALU	33.1			3.8	5.9	23.4
Flowkey	SRAM	1.8			2.3	-0.5	
Type 3	Hash Unit	18.9	5.5	0.04			13.4
Same	SALU	24.7			2.2	3.7	18.8
Epoch	SRAM	0.9			1.3	-0.4	
Type 4	Hash Unit	15.5	1.9	0.04			13.6
Random	SALU	20.4			0.5	1.0	18.9
	SRAM	0.2			0.3	-0.1	

Table 5: Breakdown of resource reduction by each optimization for the number of sketch instances = 12.

Sensitivity analysis on the number of sketch instances. We show a more detailed view of resource reduction by looking at ensembles with different numbers of sketch instances. We generate (four ensemble types) × (12 different numbers of sketch instances from 2, 4, ..., 24) × (300 different ensembles). The X-axis of Fig. 15 is the number of sketch instances, and the Y-axis is the average reduction for the three resource types of 300 ensembles between before and after optimization. We can see that hash unit reduction is up to 40%, SALU reduction is up to 45%, and SRAM reduction is up to 7%. As the ensemble has more sketch instances, we have more opportunities to apply optimizations, and resource reduction benefits increase. SRAM reduction is more limited, but we do observe SRAM reduction for type 2 due to O_{Ctr1} because reusing counter arrays can reduce SRAM.

Fig. 15 also shows that the resource reduction depends on the ensemble type. Ensemble type 2 has sketch instances with the same flowkey, which makes many optimizations easier to apply. Thus, ensemble type 2 has the highest resource reduction. On the other hand, type 4 has random sketch instances, so optimizations are the least likely to be applied, resulting in the smallest resource reduction. However, even for random ensemble type, the reduction of the hash unit is up to 20% and SALU is up to 26% because Sketchovsky offers five multiple building blocks for optimization.

Breakdown on individual optimizations. We zoom into ensembles with 12 sketch instances and show the breakdown of resource reduction in Table 5. HFSREUSE (O_{Key}) shows consistently high resource reduction for all four ensemble types (18% to 25% SALU reduction). Note that O_{Key} can also reduce hash units. This is because of the specific hardware architecture of Tofino; one hash unit must be allocated for one SALU (now we call this HashUnit-SALU coupling). HASHREUSE (O_{Hash1}) is the next impactful optimization. For type 2, O_{Hash1} reduces hash units by up to 10.4%. SALUREUSE (O_{Ctr1}) reduces both SALU and SRAM and SALUMERGE (O_{Ctr2}) reduces SALUs but increases small SRAM overhead (negative values such as -0.5%). Finally, HASHXOR (O_{Hash2}) has the least impact on Tofino because of HashUnit-SALU coupling. Note that the application of O_{Ctr1} and O_{Ctr2} enables O_{Hash1} automatically. Thus the impact of O_{Ctr1} and O_{Ctr2} is bigger than shown in Table 5.

9 Discussion

Measurement-sketch mapping. We currently assume the ensemble of sketch instances is given as input. An interesting direction for future work is to automatically generate the most efficient ensemble of sketch instances for a given set of measurement tasks. We posit that explicitly considering the characteristic of input workload and the resource-accuracy trade-off in an ensemble setting using Sketchovsky could be an interesting direction for future work [33, 47].

Generalizing to other hardware. While our prototype uses Tofino due to its open-source development API, we posit that our research contributions, such as optimization building blocks, strategy finder, and auto-code composition framework, can be adapted to other programmable switches and platforms as they have similar resource bottlenecks [31, 40].

Generalizing to other sketching algorithms. Today, some sketching algorithms are still infeasible in the data plane due to their complex data structures [39, 40]. We envision Sketchovsky to be useful for implementing other feasible sketching algorithms on programmable switches than the eleven sketching algorithms we demonstrated in §8. In this section, we elaborate on the applicability of Sketchovsky's main components, including optimization building blocks, strategy finder, and auto-code composition framework, to other sketches.

First, the optimization building blocks proposed in Sketchovsky are based on common compute and memory operations in sketching algorithms (e.g., hash computations, arithmetic counter updates, heavy flowkey storage). Since all sketching algorithms perform *hash computations*, O_{Hash1} and O_{Hash2} are generally applicable to current and future sketching algorithms. For *counter updates*, some sketching algorithms may have complicated counter update operations (e.g., threshold-based counter updates in ElasticSketch [45]). Sketchovsky cannot directly support these complicated counter updates and requires a case study to determine whether resource savings are possible. It is also possible that these complicated counter operations are fundamentally excluded from further optimizations. For *heavy flowkey storage*, the sketching algorithms that require storing heavy flowkeys [9, 16, 41] can benefit from O_{Key}. In summary, the individual optimizations introduced in Sketchovsky are broadly applicable to sketching algorithms.

Second, the strategy finder is a general optimizer to maximize resource saving. As long as the specifications of sketching algorithms such as counter array type and counter update type are given as input, the strategy finder will output an optimized strategy. For example, a user may define single-level (SL) or multi-level (ML) as the counter array type and COUNTER or SIGNCOUNTER as the counter update type. When optimizing counting bloom filters [11] as part of an ensemble, the user can define the same counter array and counter update types as the count-min sketch [17], and exclude the heavy flowkey storage part.

Finally, the auto-code composition framework can accommodate new sketching algorithms as long as their sketch templates are properly defined based on the specification of Sketchovsky. A sketch template follows three main steps, i.e., hash computation, counter updates, and flowkey storage. For the auto-code composition framework to work, the user needs to ensure that if several sketches share a common operation (e.g., signed counter update), the code provided for implementing the operation must be the same across the sketch templates. For example, implementing a counting bloom filter should reuse codes from a count-min sketch for the hash computation and counter update operations. In summary, Sketchovsky can be generalized to other sketching algorithms.

10 Conclusions

In this paper, we tackle an often ignored problem of running an ensemble of sketch instances to support a given portfolio of measurement tasks. To the best of our knowledge, Sketchovsky is the first end-to-end system that explores cross-sketch optimizations in practice. We showed that our novel cross-sketch optimization building blocks and efficient strategy finder make previously infeasible ensembles of sketch instances feasible on modern hardware.

Acknowledgment

We would like to thank the anonymous NSDI reviewers and our shepherd Anja Feldmann for their helpful comments. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF awards 1565343, 1700521, 2106946, 2107086, and 2132639. Liu was also supported by the Red Hat Collaboratory at Boston University.

References

[1] Barefoot Tofino Switch. https://barefootnetworks.com/products/brief-tofino/.

[2] CAIDA Anonymized Internet Traces. https://www.caida.org/data/passive/passive_dataset.xml.

[3] Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (Renton, WA, Apr. 2022), USENIX Association.

[4] SketchLib: Enabling efficient sketch-based monitoring on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (Renton, WA, Apr. 2022), USENIX Association.

[5] AGARWAL, A., LIU, Z., AND SESHAN, S. {HeteroSketch}: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 719–741.

[6] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., ET AL. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), pp. 503–514.

[7] BASAT, R. B., CHEN, X., EINZIGER, G., AND ROTTENSTREICH, O. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking 28*, 3 (2020), 1172–1185.

[8] BELL, E. T. Exponential polynomials. *Annals of Mathematics* (1934), 258–277.

[9] BEN BASAT, R., EINZIGER, G., FRIEDMAN, R., LUIZELLI, M. C., AND WAISBARD, E. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 127–140.

[10] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies* (2011), pp. 1–12.

[11] BONOMI, F., MITZENMACHER, M., PANIGRAHY, R., SINGH, S., AND VARGHESE, G. An improved construction for counting bloom filters. In *European Symposium on Algorithms* (2006), Springer, pp. 684–695.

[12] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.* (2014).

[13] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 99–110.

[14] CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming* (2002), Springer, pp. 693–703.

[15] CHEN, X., LANDAU-FEIBISH, S., BRAVERMAN, M., AND REXFORD, J. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 226–239.

[16] CORMODE, G., KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. Finding hierarchical heavy hitters in data streams. In *Proceedings 2003 VLDB Conference* (2003), Elsevier, pp. 464–475.

[17] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[18] DURAND, M., AND FLAJOLET, P. Loglog counting of large cardinalities. In *European Symposium on Algorithms* (2003), Springer, pp. 605–617.

[19] ESTAN, C., VARGHESE, G., AND FISK, M. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement* (2003), pp. 153–166.

[20] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences 31*, 2 (1985), 182–209.

[21] FLAJOLET, P., RIC FUSY, GANDOUET, O., AND ET AL. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA* (2007).

[22] GAO, J., ZHAI, E., LIU, H. H., MIAO, R., ZHOU, Y., TIAN, B., SUN, C., CAI, D., ZHANG, M., AND YU, M. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. 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), pp. 435–450.

[23] GAO, X., KIM, T., WONG, M. D., RAGHUNATHAN, D., VARMA, A. K., KANNAN, P. G., SIVARAMAN, A., NARAYANA, S., AND GUPTA, A. Switch code generation using program synthesis. 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), pp. 44–61.

[24] GARCIA-TEODORO, P., DIAZ-VERDEJO, J., MACIÁ-FERNÁNDEZ, G., AND VÁZQUEZ, E. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security 28*, 1-2 (2009), 18–28.

[25] GUPTA, A., HARRISON, R., CANINI, M., FEAMSTER, N., REXFORD, J., AND WILLINGER, W. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 357–371.

[26] KIRSCH, A., AND MITZENMACHER, M. Less hashing, same performance: building a better bloom filter. In *European Symposium on Algorithms* (2006), Springer, pp. 456–467.

[27] KRISHNAMURTHY, B., SEN, S., ZHANG, Y., AND CHEN, Y. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement* (2003), pp. 234–247.

[28] KUMAR, A., SUNG, M., XU, J., AND WANG, J. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review 32*, 1 (2004), 177–188.

[29] LALL, A., SEKAR, V., OGIHARA, M., XU, J., AND ZHANG, H. Data streaming algorithms for estimating entropy of network traffic. *ACM SIGMETRICS Performance Evaluation Review 34*, 1 (2006), 145–156.

[30] LI, Y., GAO, J., ZHAI, E., LIU, M., LIU, K., AND LIU, H. H. Cetus: Releasing p4 programmers from the chore of trial and error compiling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 371–385.

[31] LIU, Z., BEN-BASAT, R., EINZIGER, G., KASSNER, Y., BRAVERMAN, V., FRIEDMAN, R., AND SEKAR, V. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 334–350.

[32] Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., and Braverman, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 101–114.

[33] Liu, Z., Namkung, H., Agarwal, A., Manousis, A., Steenkiste, P., Seshan, S., and Sekar, V. Sketchy with a chance of adoption: Can sketch-based telemetry be ready for prime time? In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)* (2021), IEEE, pp. 9–16.

[34] Liu, Z., Namkung, H., Nikolaidis, G., Lee, J., Kim, C., Jin, X., Braverman, V., Yu, M., and Sekar, V. Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 3829–3846.

[35] Miao, R., Zeng, H., Kim, C., Lee, J., and Yu, M. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 15–28.

[36] Moshref, M., Yu, M., Govindan, R., and Vahdat, A. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies* (2015), pp. 1–13.

[37] Namkung, H., Kim, D., Liu, Z., Sekar, V., and Steenkiste, P. Telemetry retrieval inaccuracy in programmable switches: Analysis and recommendations. In *Proceedings of the Symposium on SDN Research* (2021).

[38] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 85–98.

[39] Schweller, R., Gupta, A., Parsons, E., and Chen, Y. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement* (2004), pp. 207–212.

[40] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research* (2017), pp. 164–176.

[41] Song, C. H., Kannan, P. G., Low, B. K. H., and Chan, M. C. Fcm-sketch: generic network measurements with data plane support. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (2020), pp. 78–92.

[42] Soni, H., Rifai, M., Kumar, P., Doenges, R., and Foster, N. Composing dataplane programs with μp4. 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), pp. 329–343.

[43] Thorup, M., and Zhang, Y. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM Journal on Computing 41*, 2 (2012), 293–331.

[44] Whang, K.-Y., Vander-Zanden, B. T., and Taylor, H. M. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS) 15*, 2 (1990), 208–229.

[45] Yang, T., Jiang, J., Liu, P., Huang, Q., Gong, J., Zhou, Y., Miao, R., Li, X., and Uhlig, S. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 561–575.

[46] Yu, D., Zhu, Y., Arzani, B., Fonseca, R., Zhang, T., Deng, K., and Yuan, L. dshark: a general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 207–220.

[47] Yu, M. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review 49*, 1 (2019), 11–17.

[48] Zhang, M., Li, G., Wang, S., Liu, C., Chen, A., Hu, H., Gu, G., Li, Q., Xu, M., and Wu, J. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *Proceedings of NDSS* (2020).

[49] Zhang, Y., Liu, Z., Wang, R., Yang, T., Li, J., Miao, R., Liu, P., Zhang, R., and Jiang, J. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 207–222.

[50] Zheng, H., Tian, C., Yang, T., Lin, H., Liu, C., Zhang, Z., Dou, W., and Chen, G. Flymon: enabling on-the-fly task reconfiguration for network measurement. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 486–502.

[51] Zheng, P., Benson, T., and Hu, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (2018), pp. 98–111.

[52] Zhou, Y., Zhang, D., Gao, K., Sun, C., Cao, J., Wang, Y., Xu, M., and Wu, J. Newton: intent-driven network traffic monitoring. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (2020), pp. 295–308.

A Supplement to Background

A.1 Counter Update Type

We introduce five counter update types as in Alg. 2.

1. **Bitmap** is just a bitmap.
2. **Counter** receives index and size for the counter update, then increase the counter depending on packet counts or packet bytes.
3. **SignCounter** receives one additional input of 1-bit hash result. Depending on this hash value, it will either increase or decrease the counter. The 1-bit hash value is computed by using flowkey.
4. **HLL** type can be used for loglog-variant sketches [18, 21]. It receives index and value as inputs and updates the counter if it is less than the value. Value can be computed by a function $\rho(hash)$ where $hash \in \{0,1\}^{32}, \rho(hash)$ is the position of the leftmost 1-bit (e.g., $\rho(0001\dots) = 4$) and *hash* is computed using flowkey [21]. This ρ function can be implemented efficiently by using TCAM in the switch data plane [4].
5. **PCSA** receives index and bitmask as inputs. Then it uses the bit-OR operation to update the counter using the

bitmask. Bitmask value can be computed by shift operation $(1 << \rho(hash))$.

Algorithm 2 Five Counter Update Types

```
function BITMAP($index$)
    A[$index$] = 1
function COUNTER($index$, $size$)
    A[$index$] = A[$index$] + $size$
function SIGNCOUNTER($hash$, $index$, $size$)
    if $hash == 0$ then
        A[$index$] = A[$index$] + $size$
    else if $hash == 1$ then
        A[$index$] = A[$index$] - $size$
function HLL($index$, $value$)
    if A[$index$] < $value$ then
        A[$index$] = $value$
function PCSA($index$, $bitmask$)
    A[$index$] = A[$index$] | $bitmask$
```

B Supplement to Optimizations

B.1 SRAM reduction of SALUREUSE (O_{Ctr1})

SALUREUSE (O_{Ctr1}) reduces not only SALUs but also SRAM. Suppose $\mathbf{S} = \{s_i\}_{i=1}^{n}$ is a set of sketch instances and $\mathbf{C} = \{(r_i, w_i)\}_{i=1}^{n}$ represent that s_i has r_i number of counter arrays with width w_i. $\mathbf{W}$ represents row and width of counter arrays for reuse after applying O_{Ctr1}.

$$\mathbf{W} = \{w_j^*\}_{j=1}^{\max_i(r_i)} \text{ where } w_j^* = \max_i\{w_i | r_i \geq j\} \tag{14}$$

Then, SRAM usage changes from $\sum_{i=1}^{n} r_i w_i$ to $\sum_{j=1}^{\max_i(r_i)} w_j^*$. O_{Ctr1} will always maintain or reduce SRAM usage because $\sum_{i=1}^{n} r_i w_i - \sum_{j=1}^{\max_i(r_i)} w_j^* \geq 0$. Suppose $comp(x,y) \in \{0,1\}$ where $x, y \in \mathbb{N}$. If $x \leq y \rightarrow comp(x,y) = 1$, otherwise $\rightarrow comp(x,y) = 0$.

$$\sum_{i=1}^{n} r_i w_i - \sum_{j=1}^{\max_i(r_i)} w_j^* = \sum_{j=1}^{\max_i(r_i)} \left(\left(\sum_{i=1}^{n} w_i \cdot comp(r_i, j)\right) - w_j^* \right)$$

$\left(\sum_{i=1}^{n} w_i \cdot comp(r_i, j)\right) - w_j^* \geq 0$ for $1 \leq j \leq max_i(r_i)$ due to (14)

Thus, $\sum_{i=1}^{n} r_i w_i - \sum_{j=1}^{\max_i(r_i)} w_j^* \geq 0$

C Supplement to Auto-code Composition

C.1 SketchLib and Lib for Optimization

SketchLib. We extended API calls from SketchLib as in Table 6 for the easier code-rewrite process.

- `TCAM_LPM (hash_result)` uses TCAM for the longest prefix match to compute the leftmost position of 1-bit in the hash result, which is used in many sketching algorithms. This API call is the same as `tcam_optimization()` in SketchLib.
- `CounterUpdate (seed, flowkey, width, CA_type, CU_type, ...)` does one counter update for configured flowkey, counter array type (`CA_type`) of whether single-level (SL) or multi-level (ML), counter update type (`CU_type`), width of counter array (`width`). `seed` is used for the hash unit to generate column index for the counter update. Depending on the different `CU_type`, it takes more parameters (e.g., packet length for COUNTER type or value out of `TCAM_LPM` for HLL/PCSA type). We extended `consolidate_update()` in SketchLib to build this API call.
- `AboveThreshold (LIST(estimate), threshold)` gets the threshold and a list of flow size estimates (these are return values after each counter update). This API call returns whether the overall flow size estimate is above the threshold or not[6]. This logic was part of `heavy_flowkey_storage()` in SketchLib and we separate the API call for the code rewrite process.
- `HFS (flowkey)` stores heavy flowkey. This API extends `heavy_flowkey_threshold()` in SketchLib by supporting any definition of flowkey.

```
01: /* 1. hash computation step - no code */
02: /* 2. counter update step */
03: s#_est1 = CounterUpdate(seed1, FLOWKEY, WIDTH,
04:                         SL, Counter, FLOWSIZE);
05: s#_est2 = CounterUpdate(seed2, FLOWKEY, WIDTH,
06:                         SL, Counter, FLOWSIZE);
07: s#_est3 = CounterUpdate(seed3, FLOWKEY, WIDTH,
08:                         SL, Counter, FLOWSIZE);
09: s#_th = AboveThreshold(s#_est1, s#_est2, s#_est3,
10:                                      THRESHOLD);
11: /* 3. counter update step */
12: if (s#_th) { HFS(FLOWKEY); }
```

Figure 16: Code template example for count-min sketch.

You can see how these API calls are used in Fig. 16, which is a code template example for count-min sketch. Network operators can put {srcIP, dstIP} to FLOWKEY, *hdr.ipv4.total_len* to FLOWSIZE, and *1024* to WIDTH. For different numbers of counter arrays (e.g., 3 counter arrays), network operators should write multiple lines of code for counter updates (e.g., lines 3-8 in Fig. 16).

Lib for Opt. Lib for Opt is used to implement SALUMERGE (O_{Ctr2}) as in Table 6.

- `CounterUpdate_2 (seed, flowkey, width, CA_type, CU_type1, CU_type2, ...)` This API looks similar to `CounterUpdate()` but the difference is that this API does two counter updates by using one SALU. Thus, parameters include two counter update types

[6] For count-min sketch, overall flow size estimate is min of List (estimate). For count-sketch, overall flow size estimate is median of List (estimate).

Two Libs	API Name	API Parameters	Explanation
SketchLib	`TCAM_LPM()`	`hash_result`	Same as `tcam_optimization()` in SketchLib
	`CounterUpdate()`	`seed, flowkey, width, CA_type, CU_type, ...`	Extends `consolidate_update()` in SketchLib
	`AboveThreshold()`	`LIST(estimate), threshold`	Extends `heavy_flowkey_storage()` in SketchLib
	`HFS()`	`flowkey`	Extends `heavy_flowkey_storage()` in SketchLib

Lib for Opt	`CounterUpdate_2()`	`seed, flowkey, width, CA_type, CU_type1,`
		`CU_type2, ...`
			New API for SALUMERGE (O_{Ctr2})

Table 6: API calls in extended SketchLib and Lib for optimization.

```
01: // code for s1
02: ... /* 1. hash computation step */
03: /* 2. counter update step */
04: s1_est1 = CounterUpdate(seed1, srcIP, 2K, SL,
05:                         Counter, pktlen);
06: s1_est2 = CounterUpdate(seed2, srcIP, 2K, SL,
07:                         Counter, pktlen);
08: s1_est3 = CounterUpdate(seed3, srcIP, 2K, SL,
09:                         Counter, pktlen);
10: s1_th = AboveThreshold(s1_est1, s1_est2, s1_est3,
11:                                              100);
12: ... /* 3. heavy flowkey storage step */
13:
14: // code for s2
15: ... /* 1. hash computation step */
16: /* 2. counter update step */
17: s2_est1 = CounterUpdate(seed4, srcIP, 4K, SL,
18:                         Counter, pktlen);
19: s2_est2 = CounterUpdate(seed5, srcIP, 4K, SL,
20:                         Counter, pktlen);
21: s2_th = AboveThreshold(s2_est1, s2_est2, 100);
22: ... /* 3. heavy flowkey storage step */
23:
24: // code for s3
25: ... /* 1. hash computation step */
26: /* 2. counter update step */
27: CounterUpdate(seed6, srcIP, 4K, SL, Counter,
28:                                 pktlen);
29: ... /* 3. heavy flowkey storage step */
```

Figure 17: [Before] SALUREUSE (O_{Ctr1}).

`CU_type1` and `CU_type2`. There are one flowkey, one width, and one counter array type because they should be the same due to applicable conditions of O_{Ctr2}.

C.2 Before and After Code Snippets for O_{Ctr1}, O_{Ctr2}, and O_{Key}

Code rewrite for counter update. Code rewriter uses $\{X^*_{Ctr1}, X^*_{Ctr2}\}$ to apply SALUREUSE (O_{Ctr1}) and SALUMERGE (O_{Ctr2}) to counter update step. Although O_{Ctr1} and O_{Ctr2} can be applied simultaneously, we explain code rewrite logic separately for better readability. Code rewrite for O_{Ctr1} to $\mathcal{S}$ requires code changes with lines using `CounterUpdate()` in the extended SketchLib. Code rewrite for O_{Ctr2} uses a new API call, `CounterUpdate_2()`.

We first look at how to apply O_{Ctr1} using X^*_{Ctr1} by looking at the before (Fig. 17) and after (Fig. 18) code snippets. Three sketch instances $\{s_1, s_2, s_3\}$ in Fig. 17 are count-min sketch, K-ary sketch, and entropy sketch respectively and they have different resource parameters $\mathbf{C} = \{(r_i, w_i)\}^3_{i=1} = \{(3,2K),(2,4K),(1,8K)\}$. $\{s_1, s_2\}$ tracks heavy flowkey and they check whether flow size estimate is above threshold at line 10 and 21 in Fig. 17. X^*_{Ctr1} specifies that code rewriter should apply O_{Ctr1} to $\{s_1, s_2, s_3\}$, meaning that they satisfy applicable conditions for O_{Ctr1}. Then, the code rewriter computes row and width of counter arrays for reuse $\mathbf{W}$ as discussed as in (3), §4.2. As a result, $\mathbf{W} = \{8K, 4K, 2K\}$ is computed in this example and the code rewriter applies this as in lines 4-9 in code snippet Fig. 18.

```
01: // optimized code for s1, s2, s3
02: ... /* 1. hash computation step */
03: /* 2. counter update step */
04: s_est1 = CounterUpdate(seed1, srcIP, 8K, SL,
05:                        Counter, pktlen);
06: s_est2 = CounterUpdate(seed2, srcIP, 4K, SL,
07:                        Counter, pktlen);
08: s_est3 = CounterUpdate(seed3, srcIP, 2K, SL,
09:                        Counter, pktlen);
10: s1_th = AboveThreshold(s_est1, s_est2, s_est3,
11:                                           100);
12: s2_th = AboveThreshold(s_est1, s_est2, s_est3,
13:                                           200);
14: ... /* 3. counter update step */
```

Figure 18: [After] SALUREUSE (O_{Ctr1}) to $\{s_1, s_2, s_3\}$.

Next, we look at how the code rewriter applies O_{Ctr2} by using X^*_{Ctr2}. Fig. 19 is the before code snippet and Fig. 20 is the after code snippet. $\{s_1, s_2, s_3\}$ in Fig. 19 are count-min sketch, entropy sketch, and PCSA sketch respectively and $\mathbf{C} = \{(3,2K),(2,4K),(1,8K)\}$. We cannot apply O_{Ctr1} to $\{s_1, s_2, s_3\}$ for this example because flowsize definitions are different between s_1 and s_2 (s_1 tracks packet bytes if we look at lines 5, 7, 9 in Fig. 19 whereas s_2 tracks packet counts at lines 17-18 in Fig. 19). Counter update types are also different between $\{s_1, s_2\}$ and $\{s_3\}$. $\{s_1, s_2\}$ uses COUNTER type whereas $\{s_3\}$ uses PCSA type.

Instead of O_{Ctr1}, we can apply O_{Ctr2} and X^*_{Ctr2} specifies that the code rewriter can apply O_{Ctr2} to $\{s_1, s_2, s_3\}$. Using the information in X^*_{Ctr2}, the code rewriter knows that the first two counter arrays of s_1 can share SALUs with s_2, and the last counter array of s_1 can share a SALU with s_3. We use the new API call `CounterUpdate_2()` to apply this optimization at lines 4-9 in Fig. 20. For the first two counter arrays (lines 4-7), both counter update types are COUNTER type. Thus, the API call takes two additional parameters of flowsize definitions of packet bytes and packet counts. For the third counter array (lines 8-9), counter update types are COUNTER and PCSA.

```
01: // code for s1
02: ... /* 1. hash computation step */
03: /* 2. counter update step */
04: s1_est1 = CounterUpdate(seed1, srcIP, 2K, SL,
05:                             Counter, pktlen);
06: s1_est2 = CounterUpdate(seed2, srcIP, 2K, SL,
07:                             Counter, pktlen);
08: s1_est3 = CounterUpdate(seed3, srcIP, 2K, SL,
09:                             Counter, pktlen);
10: s1_th = AboveThreshold(s1_est1, s1_est2, s1_est3,
11:                                             100);
12: ... /* 3. heavy flowkey storage step */
13:
14: // code for s2
15: ... /* 1. hash computation step */
16: /* 2. counter update step */
17: CounterUpdate(seed4, srcIP, 4K, SL, Counter, 1);
18: CounterUpdate(seed5, srcIP, 4K, SL, Counter, 1);
19: ... /* 3. heavy flowkey storage step */
20:
21: // code for s3
22: ... /* 1. hash computation step */
23: /* 2. counter update step */
24: CounterUpdate(seed6, srcIP, 8K, SL, PCSA,
25:                                     s3_value);
26: ... /* 3. heavy flowkey storage step */
```

Figure 19: [Before] SALUMERGE (O_{Ctr2}).

```
01: // optimized code for s1, s2, s3
02: ... /* 1. hash computation step */
03: /* 2. counter update step */
04: s_est1 = CounterUpdate_2(seed1, srcIP, 8K, SL,
05:                   Counter, Counter, pktlen, 1);
06: s_est2 = CounterUpdate_2(seed2, srcIP, 4K, SL,
07:                   Counter, Counter, pktlen, 1);
08: s_est3 = CounterUpdate(seed3, srcIP, 2K, SL,
09:                 Counter, PCSA, pktlen, s3_value);
10: s1_th = AboveThreshold(s_est1, s_est2, s_est3,
11:                                           100);
12: ... /* 3. counter update step */
```

Figure 20: [After] SALUMERGE (O_{Ctr2}) to $\{s_1, s_2, s_3\}$.

Thus, two additional parameters are flowsize definition of packet bytes and an output value of TCAM_LPM written as s3_value at line 9 in Fig. 20.

Code rewrite for heavy flowkey storage. Code rewriter uses X^*_{Key} to apply HFSREUSE (O_{Key}) to the heavy flowkey storage step. Fig. 21 is the before code snippet and Fig. 22 is the after code snippet. We have four sketch instances $\{s_1, s_2, s_3, s_4\}$ with different flowkeys F = {{srcIP}, {srcIP, dstIP}, {srcIP, srcPort}, {srcIP, dstIP, srcPort, dstPort}} and all sketch instances track heavy flowkey. O_{Key} uses union key $UK = \cup_i f_i$ for the heavy flowkey storage for reuse. In this example, UK ={srcIP, dstIP, srcPort, dstPort} is written at line 14 in Fig. 22. Recall that we have further optimization using conditional union-key $UK_C = \cup_j f_j$where (flow size estimate)$_j > threshold_j$ and set 0 to $(UK - UK_C)$. This optimization is written in the code at lines 6-11 in Fig. 22. For each packet header field (e.g., dstIP), it detects which sketch instances have this header field (e.g., s_2 and s_4 because f_2 and f_4 have dstIP). Then if any of those sketch instances is above the threshold (at line 9), those header fields are included in UK_C. If not, this header field is set to zero (at line 5). As a result, we can reduce 4 heavy flowkey storages to 1 heavy flowkey storage.

```
01: // code for s1
02: ... /* 1. hash computation step */
03: ... /* 2. counter update step */
04: /* 3. heavy flowkey storage step */
05: if (s1_th) { HFS(srcIP); }
06:
07: // code for s2
08: ...
09: /* 3. heavy flowkey storage step */
10: if (s2_th) { HFS(srcIP, dstIP); }
11:
12: // code for s3
13: ...
14: /* 3. heavy flowkey storage step */
15: if (s3_th) { HFS(srcIP, srcPort); }
16:
17: // code for s4
18: ...
19: /* 3. heavy flowkey storage step */
20: if (s4_th) { HFS(srcIP, dstIP, srcPort, dstPort); }
```

Figure 21: [Before] HFSREUSE (O_{Key}).

```
01: // code for s1, s2, s3, s4
02: ... /* 1. hash computation step */
03: ... /* 2. counter update step */
04: /* 3. heavy flowkey storage step */
05: hf_srcIP = hf_dstIP = hf_srcPort = hf_dstPort = 0
06: if (s1_th || s2_th || s3_th || s4_th) {
07:   hf_srcIP = srcIP;
08: }
09: if (s2_th || s4_th) { hf_dstIP = dstIP; }
10: if (s3_th) { hf_srcPort = srcPort; }
11: if (s4_th) {hf_dstPort = dstPort; }
12:
13: if (s1_th || s2_th || s3_th || s4_th) {
14:   HFS(hf_srcIP, hf_dstIP, hf_srcPort, hf_dstPort);
15: }
```

Figure 22: [After] HFSREUSE (O_{Key}) to $\{s_1, s_2, s_3, s_4\}$.

D Supplement to Evaluation

D.1 Eleven Sketch Algorithms for Evaluation

We use eleven sketching algorithms for our evaluation as in Table 7. They have different sketch features. Counter array type can be single-level (SL) or multi-level (ML). We also show a pool of candidate configurable parameters per each sketching algorithm in Table 7. For entropy sketch, counter update type SIGNCOUNTER guarantees theoretically better accuracy due to F2 estimation. However, we found that the COUNTER type produces better accuracy in practice. Thus, we use this COUNTER type for entropy sketch in our evaluation.

Sketch Algorithms		Sketch Features			Configurable Parameters Candidates					
Statistic	Name	Counter Array	Counter Update	Heavy Flowkey	Flowkey	Flowsize	Epoch	Row	Width	Level
Membership	BF [11]	SL	BITMAP	N	{(srcIP), (dstIP), (srcIP, dstIP), (srcIP, srcPort), (dstIP, dstPort), (4-tuple), (5-tuple)}	{counts}	{10s, 20s, 30s, 40s}	{1}	{128K, 256K, 512K}	-
Cardinality	LC [44]	SL	BITMAP	N						
	MRB [19]	ML	BITMAP	N					{16K, 32K}	{8, 16}
	PCSA [20]	SL	PCSA	N					{4K, 8K, 16K}	-
	HLL [21]	SL	HLL	N						
HH/HC	CS [14]	SL	SIGNCOUNTER	Y		{counts, bytes}		{1, 2, 3, 4, 5}		
	CM [17]	SL	COUNTER	Y						
	KARY [27]	SL	COUNTER	Y						
Entropy	ENT [29]	SL	COUNTER	N		{counts}				
General	UM [32]	ML	SIGNCOUNTER	Y				{3,4,5}	{2K}	{16}
FSD	MRAC [28]	ML	COUNTER	N				{1}		{8, 16}

Table 7: Eleven sketch algorithms with sketch features and possible configurable parameters. (4-tuple) = (srcIP, dstIP, srcPort, dstPort). (5-tuple) = (srcIP, dstIP, srcPort, dstPort, proto).

D.2 Four Ensembles for Accuracy Evaluation

Table 8 - Table 11 shows four picked ensembles for four ensemble types. All five optimizations are found in four picked ensembles.

Ensemble Type 1.

- HASHREUSE (O_{Hash1}): none
- HASHXOR (O_{Hash2}): none
- SALUREUSE (O_{Ctr1}): none
- SALUMERGE (O_{Ctr2}): none
- HFSREUSE (O_{Key}): $\{s_1, s_2, s_3, s_4, s_5, s_6\}$

Ensemble Type 2.

- HASHREUSE (O_{Hash1}): $\{s_3, s_4, s_6, s_{10}\}$
- HASHXOR (O_{Hash2}): none
- SALUREUSE (O_{Ctr1}): $\{s_8, s_9\}$
- SALUMERGE (O_{Ctr2}): $\{\{s_1\}, \{s_2\}\}$, $\{s_7, \{s_8, s_9\}\}$
- HFSREUSE (O_{Key}): $\{s_2, s_8, s_9\}$

Ensemble Type 3.

- HASHREUSE (O_{Hash1}): $\{s_3, s_4\}$
- HASHXOR (O_{Hash2}): none
- SALUREUSE (O_{Ctr1}): $\{s_8, s_9\}$
- SALUMERGE (O_{Ctr2}): $\{\{s_3\}, \{s_4\}\}$, $\{\{s_6\}, \{s_7\}\}$
- HFSREUSE (O_{Key}): $\{s_4, s_5\}$

Ensemble Type 4.

- HASHREUSE (O_{Hash1}): none
- HASHXOR (O_{Hash2}): $\{\{s_1\}, \{s_2\}, \{s_3\}\}, \{\{s_4\}, \{s_5\}, \{s_9\}\}$
- SALUREUSE (O_{Ctr1}): none
- SALUMERGE (O_{Ctr2}): $\{\{s_7\}, \{s_8\}\}$
- HFSREUSE (O_{Key}): none

SI	Base SA (*)	Configurable Parameters			
		Flowkey	Flowsize	Epoch	Resource
s_1	CM	(srcIP)	counts	40s	(1, 16K)
s_2	CM	(srcIP)	bytes	10s	(5, 4K)
s_3	CM	(srcIP, dstIP)	bytes	30s	(2, 16K)
s_4	CM	(srcIP, srcPort)	bytes	30s	(5, 8K)
s_5	CM	(dstIP, dstPort)	bytes	20s	(2, 4K)
s_6	CM	(5-tuple)	counts	40s	(5, 8K)

Table 8: Ensemble Type 1. Same Sketch Algorithm

SI	Base SA	Configurable Parameters			
		Flowkey(*)	Flowsize	Epoch	Resource
s_1	ENT	(dstIP, dstPort)	counts	10s	(3, 16K)
s_2	CS	(dstIP, dstPort)	counts	10s	(3, 16K)
s_3	MRB	(dstIP, dstPort)	-	20s	(1, 16K, 8)
s_4	MRAC	(dstIP, dstPort)	counts	20s	(1, 2K, 8)
s_5	BF	(dstIP, dstPort)	-	30s	(3, 128K)
s_6	MRB	(dstIP, dstPort)	-	30s	(1, 16K, 16)
s_7	ENT	(dstIP, dstPort)	counts	30s	(4, 4K)
s_8	CM	(dstIP, dstPort)	bytes	30s	(3, 4K)
s_9	KARY	(dstIP, dstPort)	bytes	30s	(1, 4K)
s_{10}	MRAC	(dstIP, dstPort)	counts	40s	(1, 2K, 8)

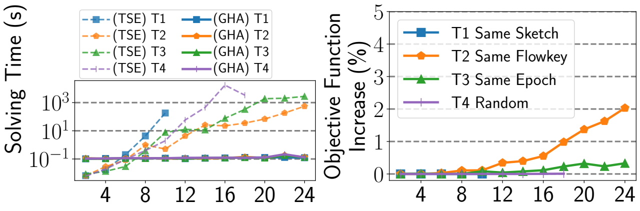
Table 9: Ensemble Type 2. Same Flowkey

SI	Base SA	Configurable Parameters			
		Flowkey	Flowsize	Epoch(*)	Resource
s_1	HLL	(srcIP)	-	30s	(1, 16K)
s_2	HLL	(dstIP)	-	30s	(1, 4K)
s_3	MRAC	(srcIP, dstIP)	counts	30s	(1, 2K, 8)
s_4	UM	(srcIP, dstIP)	counts	30s	(3, 2K, 16)
s_5	UM	(srcIP, srcPort)	counts	30s	(4, 2K, 16)
s_6	PCSA	(dstIP, dstPort)	-	30s	(1, 8K)
s_7	ENT	(dstIP, dstPort)	counts	30s	(2, 16K)
s_8	BF	(4-tuple)	-	30s	(3, 128K)
s_9	LC	(4-tuple)	-	30s	(1, 128K)
s_{10}	ENT	(5-tuple)	counts	30s	(5, 4K)

Table 10: Ensemble Type 3. Same Epoch

SI	Base	Configurable Parameters			
	SA	Flowkey	Flowsize	Epoch	Resource
s_1	MRAC	(srcIP)	counts	20s	(1, 2K, 16)
s_2	MRB	(dstIP)	-	30s	(1, 16K, 8)
s_3	MRB	(srcIP, dstIP)	-	20s	(1, 32K, 8)
s_4	HLL	(srcIP, srcPort)	-	10s	(1, 4K)
s_5	PCSA	(dstIP, dstPort)	-	20s	(1, 16K)
s_6	ENT	(dstIP, dstPort)	counts	30s	(3, 8K)
s_7	ENT	(4-tuple)	counts	30s	(5, 4K)
s_8	CS	(4-tuple)	counts	30s	(3, 8K)
s_9	PCSA	(4-tuple)	-	40s	(1, 16K)
s_{10}	HLL	(5-tuple)	-	30s	(1, 8K)

Table 11: Ensemble Type 4. Random



(a) Solving Time **(b)** Performance on Reduction

Figure 23: Two-step enumeration (TSE) vs greedy heuristic algorithm (GHA).

D.3 Experiment for Greedy Heuristic Algorithm

In the strategy finder section (§5), we propose the greedy heuristic algorithm to tackle the problem of large search space. Here we show that the performance loss of the greedy heuristic algorithm is small while solving time is three orders of magnitude faster.

Metric. We introduce two metrics for this experiment.

- Solving Time: time to find the solution.
- Objective Function Increase: $\frac{HwResource(X_G)}{HwResource(X_T)}$ where X_T is a found solution using the two-step enumeration and X_G is from the greedy heuristic algorithm.

We can see in Fig. 23a that the greedy heuristic algorithm is three orders of magnitude faster than two-step enumeration. However, the objective function increase is less than 2% (Fig. 23b). For solving time, we measure time for 300 ensembles per data point in Fig. 23a and show the worst solving time. Data points that take more than 24 hours are not shown.