Enabling Practical and Performant Sketch-based Network Telemetry on Programmable Switches

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Hun Namkung

B.S., Computer Science, Korea Advanced Institute of Science and Technology (KAIST)

Carnegie Mellon University
Pittsburgh, PA

Jan 2023

Acknowledgements

I deeply appreciate the privilege that I could have received during a long journey to achieving my Ph.D. I feel very fortunate to meet two amazing advisors, incredibly talented colleagues, a wonderful research environment, and great friends here at CMU. Despite these extraordinary privileges, earning a Ph.D. has been a challenging task for me. I am proud of myself as I am about to complete this momentous milestone in my life.

First and foremost, I want to thank my two advisors Peter Steenkiste and Vyas Sekar. Besides their excellent intellectuality and keen insights, they give me tremendous support, guidance, and help whenever I need them. I sincerely appreciate their endurance because they give me endless opportunities to learn and improve. Under their great guidance, I was able to learn the rigorous thought process, the courage that any challenges can be overcome, and the great passion for finding and solving interesting problems that can benefit the world.

I would also like to extend my sincere thanks to Minlan Yu and Alan Liu who accepted my request to join the thesis committee. My entire research is inspired by many nominal works of Minlan and having her as a thesis committee member is a great honor for me. Thank you Minlan for your great feedback and comments that can hugely improve my thesis. I also thank Alan, who joined here CMU as a postdoc. Collaboration with him was truly enjoyable and I appreciate his intellectual and mental support when I experience the ups and downs during the journey.

I also want to thank my colleague and project collaborator, Daehyeok Kim. Besides his great ability for doing research, his attitude, commitment, and dedication to research are big inspirations to me. Whenever I have a hard time, he always gives me great pieces of advice on how to see things in positive ways and his positive energy helped me a lot.

I thank my fellow graduate students, group members, and friends for their support: Soojin Moon, Antonis Manousis, Sekar Kulandaivel, Aqsa Kashaf, Yucheng Yin, Milind Srivastava, Brian Singer, Ao Li, Maria Apostolaki, Arjun Singhvi, Kittipat Apicharttrisorn, Anup Agarwal, Zhou Cheng, Byungsoo Jeon, Kiwan Maeng, Dohyun Kim, Juyong Kim, Byeongjoo Ahn, Soyong Shin, Haejoon Lee, Jiin Woo, Yaejee Cho. Also, I want to thank Chad Dougherty for managing and helping with Beluga lab clusters.

Last but not least, I would like to express my deepest appreciation to my wife, Saerom Park, for her constant love and support throughout my journey to completing this thesis. Her unwavering emotional support has been a constant source of motivation and strength. I am forever grateful for her sacrifices and for being my rock during the toughest times. This achievement would not have been possible without her. Thank you from the bottom of my heart.

The work presented in this thesis has been 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, and 2107086.

Thesis Committee Members

Peter Steenkiste (Co-chair)
Vyas Sekar (Co-chair)
Minlan Yu, Harvard University
Zaoxing Liu, Boston University

Abstract

Network telemetry plays an essential role in managing network systems. Various flow-level traffic measurement results (e.g., identifying heavy flows) are needed by network operators to make the right management decisions. In this thesis, we envision performant and practical flow-level network telemetry that satisfies four requirements: (1) low resource footprint, (2) high measurement accuracy, (3) high packet processing speed, and (4) support for diverse measurement results. State-of-the-art techniques of packet sampling suffer from low measurement accuracy. Instead, an alternative type of technique called sketching algorithms, or sketches, has received considerable attention due to their high measurement accuracy and resource efficiency. With recent advances in programmable network hardware technology, programmable switches have become a promising platform to program and deploy sketches with Tbps scale of high packet processing speed.

Although running sketches on a programmable switch is a promising way to achieve the goal of satisfying the four requirements, there is a gap between sketches and programmable switches. While there have been continuous improvements on the theoretical side of sketching algorithms for better resource-accuracy tradeoffs, much less attention has been paid to how to efficiently run sketches on actual hardware switches. Specifically, there are three practical challenges: first, sketch implementations require excessive resources on the hardware switch, making rich sketch-based telemetry often infeasible. Second, it takes a long time for developers to implement sketches on programmable switches because of the complex underlying hardware architecture. Third, while packets update counters in the switch data plane, the switch control plane also reads and resets the counters, causing consistency problems that degrade measurement accuracy significantly.

In this thesis, we present techniques that enable performant and practical sketch-based network telemetry on programmable switches by proposing optimizations to the sketch implementations, as well as by providing APIs and the code composition framework that automatically generates optimized sketch codes. In particular, we designed four novel systems. SketchLib and Sketchovsky propose optimizations for a single sketch and multiple sketches to reduce the hardware resources in the data plane. Auto-code composition framework automatically generates optimized data plane sketch code. CounterFetchLib presents optimizations and APIs for minimizing read and reset delays in the control plane to address the counter contention problem. We show the feasibility and effectiveness of optimizations through extensive hands-on evaluations using the actual hardware. Using auto-code composition framework and APIs, developers can quickly implement accurate and performant sketches on the programmable switch without worrying about the complexities of the underlying hardware architecture.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Network telemetry is a key enabler for making the right network management decisions. Flow-level measurement results enable many network management applications, such as traffic engineering, anomaly detection, load balancing, and resource provisioning [19, 26, 50, 52, 79, 83, 101]. For example, heavy flow detection on source IP can detect DDoS attacks, and measuring the unique number of 5-tuple flows can be used to detect SYN flood attacks. The more information operators can get about the network, the more operators can make the right management and control decisions [102]. Therefore, it is important to collect diverse measurement results concurrently. Measurement results are produced by running measurement tasks on network switches.

In this thesis, we envision performant and practical flow-level network telemetry that satisfies four requirements; (1) low resource footprint, (2) high measurement accuracy, (3) high packet processing speed, and (4) support for diverse measurement results. Due to limited per-packet processing time on the network switches (e.g., the scale of ns [28]), packet sampling is the state-of-the-art technique widely used today to reduce the computation and memory overhead on the switch for running measurement tasks (sampling 1 in 1000 packets is common [88]). However, the sampling-based approach severely suffers from accuracy degradation of measurement results [40, 42, 87]. Moreover, packet sampling still requires linear memory space of $O(N)$, where N is the number of unique flows in the traffic.

To overcome these shortcomings, an alternative class of techniques called sketching algorithms or sketches has attracted extensive attention for running measurement tasks efficiently on network switches [27, 32, 37, 43, 46, 47, 65, 66, 67, 71, 96]. Sketching algorithms are promising because they only require only $log(N)$ memory space, and offer a theoretical guarantee of high accuracy for measurement results by processing every packet. In parallel, a recent innovation in programmable network hardware switches makes the switch data plane more programmable without sacrificing the performance of packet processing speed (e.g., the scale of Tbps). This advancement enables many useful network functions (NFs) on programmable switches such as network address translators (NAT), load-balancing [79], in-network cache [59], consensus [39, 68], machine learning (ML) [98], and in-network lock management [104].

Given ever-increasing traffic rates, we observe that the confluence of these two trends, sketching algorithm and programmable hardware switch, is a promising avenue for providing rich telemetry. Sketching algorithms can offer high accuracy and resource efficiency, while programmable switches are highly performant and flexible. Although this approach seems promising to achieve the goal of practical and performant flow-level network telemetry, there are still missing pieces to realize this goal. In this thesis, we explore challenges when we combine those two trends of running sketching algorithms on programmable switches. Then we propose a principled and viable path to address identified challenges to realize practical and performant sketch-based network telemetry on programmable switches. In this chapter, we look at the background of sketching algorithms and programmable switches, explore challenges, and see the contributions of this thesis.

1.1 Sketching algorithms are promising for network telemetry

Sketching algorithms or *sketches* are randomized approximation algorithms that are designed to compute different observed statistics on a given data stream during every measurement time interval, called *epoch*. Prior work has shown that sketches [22, 32, 37, 58, 71, 73, 78, 88, 100] offer better resource-accuracy trade-offs relative to traditional techniques that rely on sampling (e.g., NetFlow [35]). Many different sketching algorithms can support diverse statistics for measure-

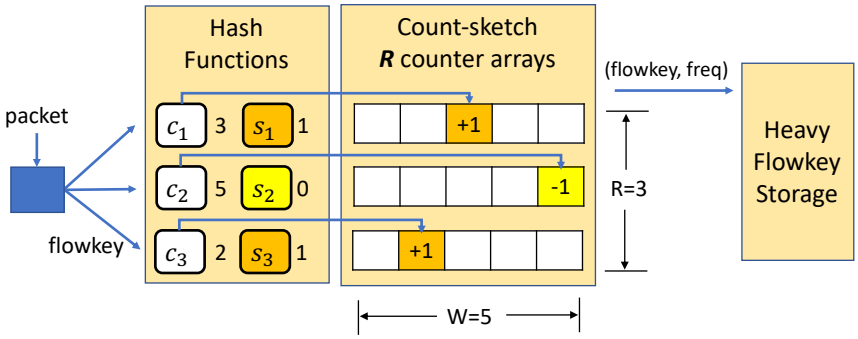


Figure 1.1: Count Sketch has three components - hash computations, multiple counter arrays, and heavy flowkey storage.

ment tasks. For example, count-min sketch (CM) [37] can identify heavy hitters, HyperLogLog (HLL) [47] can estimate the distinct number of flows, and K-ary sketch (KARY) [65] for heavy change detection. Recently, more expressive sketching algorithms [22, 58, 71, 100, 106] are developed to support general estimation capabilities (e.g., UnivMon [71]) and multidimensional analytics (e.g., R-HHH [22]). We classify prior sketching work into two categories:

1. Single-level sketches: As a canonical example, we consider the *count sketch* (CS) [32] for heavy hitter detection shown in Figure 1.1. Sketching algorithms follow three common steps. First, sketching algorithms perform *hash computations*. As each packet arrives, the count sketch extracts a flowkey (e.g., 5-tuple) from the packet header. On this key, count sketch computes two independent hash functions c_i and s_i, corresponding for each row i. Second, using these hash results, sketching algorithms perform *counter updates*. The count sketch is a single-level sketching algorithm, meaning that it maintains 2D counter arrays; R independent counter arrays with the size of W, thus $R \times W$ counters in total. Then, hash results c_i is used to select a specific column, and s_i is a 1-bit hash used to determine either to increase or decrease the counter for each row i. Third, sketching algorithms need to maintain *heavy flowkey storage*. Sketching algorithm uses threshold value to compare against flow size estimates to detect and store heavy flowkeys.

2. Multi-level sketches: Multi-level sketches consist of multiple single-level sketches to enable richer measurement tasks. For instance, R-HHH and UnivMon use multiple count sketches, called levels (e.g., L levels of $R \times W$ counters). R-HHH supports the detection of hierarchical heavy hitters, which detects heavy hitters based on different lengths of IP prefixes, and UnivMon provides

```
control ingress // R-HHH
{
  V = randomInt(1, L);

  if (V == 1) {
    key = srcIP/32;
    apply(CS_level_1,key);
  }
  if (V == 2) {
    key = srcIP/24;
    apply(CS_level_2,key);
  }
  if (V == 3) {
    key = srcIP/16;
    apply(CS_level_3,key);
  }
  ...
}
```

((a)) R-HHH

```
control ingress // UnivMon
{
  key = srcIP/32;

  apply(CS_level_1, key);
  apply(compute_hash_h1, key);

  if (h1 == 1) {  // 0 or 1
    apply(CS_level_2, key);
    apply(compute_hash_h2, key);

    if (h2 == 1) {
      apply(CS_level_3, key);
      apply(compute_hash_h3, key);

      if (h3 == 1) {
        ...
}
```

((b)) UnivMon

Figure 1.2: Simplified P4 code of existing multi-level sketches.

more general estimation capabilities. Other sketches like PCSA, MRAC, and multi-resolution bitmap (MRB) [43, 46, 66] use multiple 1D-array single-level sketches. Multi-level sketches typically perform counter updates for a few selected levels for a given flowkey. For instance, as shown in Figure 1.2(a), R-HHH randomly selects one level of count sketch using a level-specific key (e.g., IP prefix) to update per packet. In contrast, UnivMon uses an additional sampling stage using hash functions that return 0 or 1 to select levels for the update, as shown in Figure 1.2(b).

1.2 Programmable switches are high-performant and flexible

With recent innovations in programmable network hardware technology, programmable switches have emerged as an attractive platform to deploy various network functions with high packet processing speed (e.g., Tbps). Our focus in this thesis is programmable switch hardware based on the Reconfigurable Match-Action Tables (RMT) paradigm [28]. An example of a canonical commercial realization of this architecture is the Intel Tofino switch chip [9].

Hardware architecture. RMT-based programmable switches have a pipeline of reconfigurable match-action tables in the data plane, as shown in Figure 1.3. There are constraints in the packet processing pipeline to meet the line-rate processing requirement. For example, at each stage, a packet can access a limited amount of compute and memory resources. Each stage has an identical

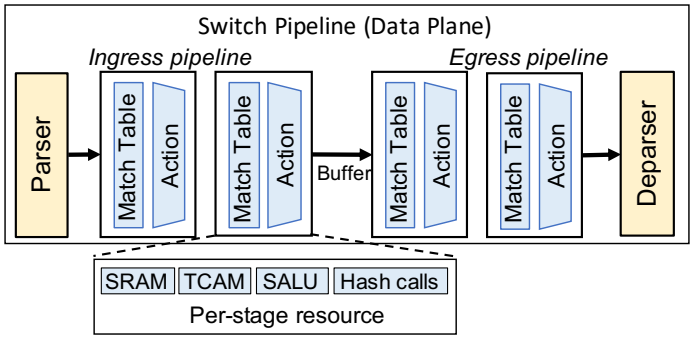


Figure 1.3: RMT switch architecture.

design with the same types of resources. To provide flexible match-action operations, each stage has a *match table* that matches packet headers to specific values followed by an action unit that executes a set of simple instructions, depending on the output of the matching unit.

Key hardware resources. We now briefly describe the key hardware resources available in each pipeline stage. First, there are a number of hardware *hash function calls* (hash calls) per pipeline stage. They are used to compute hash functions (e.g., CRC with user-defined polynomials) over packet header fields or metadata to support operations such as load balancing and table lookups. Each pipeline stage also has a fixed amount of SRAM that can be used to maintain state (e.g., counter arrays). *Stateful ALUs* (SALUs) are hardware resources that allow one read and one write operation to the stateful object in SRAM. Each SALU can be used for counter update operations, such as counter increment or decrement. Finally, each pipeline stage is also equipped with some amount of *ternary content-addressable memory* (TCAM) that can be used for wildcard matches over header fields. Overall, the amount of these limited resources is fixed at hardware design time. For example, a commercial programmable switch today is equipped with (at most) 10 SALUs, 10 hash calls, 10 MBs of SRAM and TCAM per pipeline stage with a total of 12 pipeline stages [28, 79, 110].[1]

The data plane can interact with the switch control plane for additional processing. However, the switch control plane is not designed for real-time processing, e.g., the bandwidth to the control plane is limited and the response time is high. It is therefore only useful for infrequent operations.

[1] The other absolute resource numbers are proprietary.

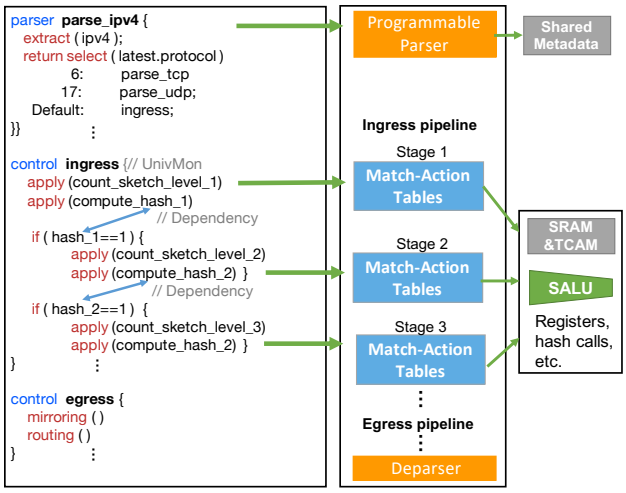


Figure 1.4: Mapping P4 code to switch resources.

1.3 Sketching algorithms on programmable switches

Programmable switches have become a promising platform for running sketches. As network traffic patterns change over time, network operators want the flexibility to run different sets of measurement tasks. Unlike traditional fixed-function hardware devices, network operators can program and deploy any existing or new sketching algorithms, at any time they want, with programmable switches. To support one measurement task (e.g., heavy hitter detection on source IP) on programmable switches, a sketch instance should be initiated based on a sketching algorithm with configuration on parameters (e.g., flowkey definition such as source IP and resource allocations for R and W). As one sketch instance can often support one measurement task, running multiple sketch instances concurrently on the programmable switch can support diverse measurement results. Sketch implementations for sketch instances are partitioned across the data and control plane.

1.3.1 Sketching Algorithms on Switch Data Plane

Sketch data plane implementations are developed to make packets to counter arrays as in Figure 1.5. Data plane programs for RMT switches are written in the P4 language [29] as illustrated in Figure 1.4. At a high level, a P4 program consists of the following components. First, a packet

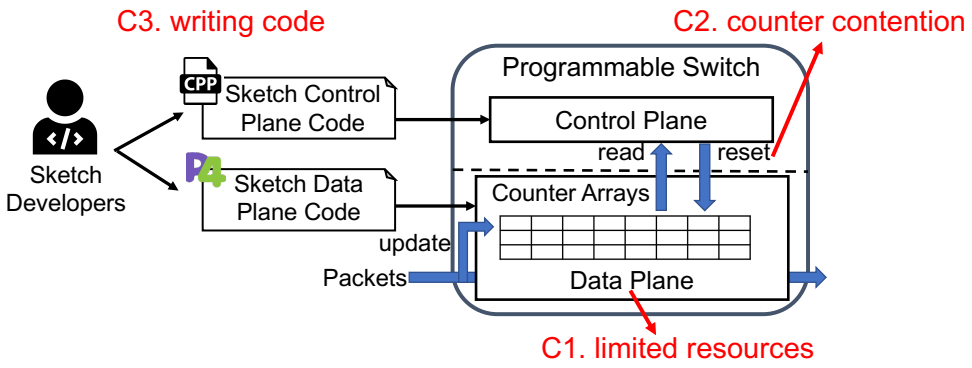


Figure 1.5: Three challenges for running sketches on programmable switches

parser parses the header fields of each packet and stores the extracted fields in metadata. Second, a series of match-action operations are executed based on the *match-action* abstraction, e.g., matching a specific header field and updating a register as an action. The action is specified by special functions that map operations to hardware resources, e.g., functions for hashing and accessing memory. Finally, the P4 program defines the packet forwarding behaviors, e.g., routing a packet to an egress port, recirculating it in the pipeline, or forwarding it to the switch control plane.

The P4 compiler maps the P4 program into a static pipeline realization. The compiler analyzes the *dependencies* between operations in the P4 program to map the program onto the pipeline stages. For instance, given the code snippet in Figure 1.4, the resolution of each `if`-clause depends on the previous hash result. Because of this dependency, two consecutive `if`-clauses cannot run in parallel, so the compiler has to map them to different pipeline stages in order for them to run sequentially. If a mapping of a whole program is possible considering hardware constraints, packets are guaranteed to be processed at line rate; otherwise, compilation fails. Note that vendor-specific compiler backends are typically proprietary.

1.3.2 Sketching Algorithms on Switch Control Plane

Figure 1.5 shows the common workflow for deploying sketching algorithms on the control plane of programmable switches. Traffic is chunked into time intervals or *epochs*. On the data plane, sketch implementations maintain counter arrays that are updated by processing packets, as we discussed

in the previous section. At the end of every epoch, the control plane periodically reads the counter arrays and resets them. Essentially, the counter arrays are *shared state* between the data plane and the control plane. Fetched counters in the control plane are used to compute measurement results.

1.4 Challenges of running sketches on programmable switches

While running multiple sketch instances on programmable switches seems like a promising way to achieve all four requirements, we find that there are still many missing pieces to enable sketches on programmable switches. Even though there have been continuous improvements on the theoretical side of sketching algorithms for better resource-accuracy trade-offs, little exploration has gone into figuring out the feasibility of sketches on programmable switches. It turns out that deploying such sketches on programmable switches leads to many practical problems, such as significant overhead on the data plane hardware resources, as well as a counter contention problem between the data plane and control plane. As a result, sketch implementations are often infeasible due to excessive resource usage on the data plane, and they suffer from significant accuracy degradation due to miscounting measurement information in the control plane. Further, sketch developers must put in a lot of effort to learn underlying hardware architecture, in order for them to write codes for data and control plane implementations. We articulate three main challenges to run sketches on programmable switches (Figure 1.5):

Challenge 1: Limited data plane resources. To maintain a high packet processing speed of Tbps, the per-packet processing time is limited, and there are only a small number of hardware resources on programmable switches (e.g., hardware resources for hash computations and for memory accesses to counter arrays). However, running sketch instances needs a lot more hardware resources than current programmable switches can offer. A single sketch instance is often infeasible due to the lack of hardware resources. Thus, it is even more challenging to run multiple sketch instances concurrently.

Challenge 2: Contention on counters between data and control plane. To compute measurement results, the switch control plane must periodically read and reset counters. However, since

there is no provided mechanism to preserve data consistency for counters, the data contention problem occurs when the data plane updates counters while the control plane tries to read and reset the same counters. As a result, retrieved counters in the control plane become inaccurate and increase the measurement error significantly (by up to 94x).

Challenge 3: Long development time. Learning the underlying architecture of programmable switches and attendant programming languages used for writing sketch implementations is a challenging task that takes a lot of time. To make it even worse, writing resource-efficient and optimized code for both the data plane and control plane needs even more training time, potentially amounting to multiple years of experience and effort.

1.5 Thesis overview

1.5.1 High-level approach

We address these challenges by (1) performing systematic bottleneck analysis, (2) proposing optimizations to the data and control plane sketch implementations based on the bottleneck analysis result, and (3) providing APIs and auto-code composition framework to reduce the developer's manual effort. Table 1.1 summarizes the high-level approach of this thesis to address the three challenges.

- To address the first challenge (C1) of limited resources in the data plane, we first identify data plane resource bottlenecks on programmable switches for running sketch instances. Based on this result, we propose both per-sketch and cross-sketch optimizations. Per-sketch optimizations reduce resource overhead *within* individual sketch instances (chapter 3). In opposition, cross-sketch optimizations reduce resource overhead *across* a set of sketch instances (chapter 4).

- To address the second challenge (C2) of the counter contention problem, we minimize execution time for read and reset operations to minimize error. To achieve these optimizations,

Contributions	Plane	C1. Limited Resources	C2. Counter Contention	C3. Writing Code
SketchLib (chapter 3)	Data Plane	✓		✓
Sketchovsky (chapter 4)		✓		
Auto-code Composition (chapter 5)				✓
CounterFetchLib (chapter 6)	Control Plane		✓	✓

Table 1.1: Contribution Summary

we decompose read and reset delays into smaller chunks and identify which of them are bottleneck delays. This bottleneck analysis result leads us to propose optimizations that reduce the total delay significantly (chapter 6).

- To address the third challenge of long development time (C3), we provide APIs for sketch developers to easily apply all of the proposed optimizations (§3.4, §6.4). Moreover, we build an auto-composition framework that automatically generates optimized data plane code for an ensemble of sketch instances (chapter 5).

1.5.2 Thesis Statement

We enable performant and practical sketch-based network telemetry on programmable switches by proposing optimizations to the sketch implementations and by providing APIs and code composition framework that automatically generates optimized codes.

1.5.3 Thesis contributions summary

We have four novel systems to tackle three challenges to achieve the goal of deploying multiple sketch instances on programmable switches concurrently.

SketchLib: Optimizing single sketch instance on programmable switches (chapter 3). SketchLib identifies four resource bottlenecks that run sketching algorithms on programmable switches. Then, it proposes six per-sketch optimizations that enable a single sketch instance on the switch data plane. Finally, SketchLib provides APIs to apply these optimizations. In SketchLib, we demonstrate:

- Optimizations reduce bottleneck resource usage by up to 9-96% so that many previously infeasible sketches become feasible on programmable switches while the accuracy is not affected by optimizations.
- We show that optimizations are applicable to a broad range of 15 sketching algorithms [22, 32, 37, 38, 41, 43, 46, 47, 58, 65, 66, 67, 71, 89, 92]. Using API reduces lines of code from 201-471 to 91-131.

Sketchovsky: Optimizing ensembles of sketch instances on programmable switches (chapter 4). While SketchLib enables a single sketch instance by per-sketch optimizations, Sketchovsky enables an ensemble of sketch instances by proposing cross-sketch optimizations. We identify five cross-sketch optimizations, and the key insight is that there are many opportunities to reuse hardware resources across sketch instances. Sketchovsky also proposes greedy heuristic algorithms to find the best way to use optimizations. In Sketchovsky, we demonstrate:

- Sketchovsky causes up to 18 sketch instances to become feasible by reducing bottleneck resource usage by up to 45%.
- Sketchovsky always maintains, and sometimes even improves, accuracy.

Auto-code Composition Framework: Automatically generates optimized sketch data plane code (chapter 5). An auto-code composition framework automatically writes optimized sketch data plane code for developers. We provide code templates for each sketching algorithm using SketchLib. For a single sketch instance, developers can configure flowkeys and resource parameters using code templates to build optimized code. For an ensemble of sketch instances, auto-code composition creates the optimized code by applying the best strategy to use optimizations from Sketchovsky using code rewrites.

CounterFetchLib: Optimizing sketch control plane on programmable switches for accurate measurement results (chapter 6). CounterFetchLib decomposes counter read and reset delays into smaller chunks and identifies bottleneck delays. Based on this analysis, we propose an optimization characterized by deferring a part of the read operation after the reset operation, where the

specific fraction of the read operation is not in the critical path to compute the measurement results. Another optimization additionally leverages bulk reset operation. Finally, CounterFetchLib provides APIs for developers to apply these optimizations. In CounterFetchLib, we demonstrate:

- These optimizations reduce delays by 95%.
- The error induced by the delay is reduced by 97%.

To summarize, SketchLib and Sketchovsky tackle the first challenge of limited data plane hardware resources by proposing per-sketch and cross-sketch optimizations. CounterFetchLib addresses the second challenge of counter contention by optimizing the control plane. All three works address the third challenge of long development time by providing APIs.

1.6 Scope of the thesis

This thesis focuses on programmable switch hardware based on the Reconfigurable Match-Action Tables (RMT) paradigm [28]. Specifically, we explore the feasibility and effectiveness of proposed optimizations and APIs using Intel Tofino switch chip [9], a canonical commercial realization of RMT architecture. Based on public documentation and conversations with vendors, we believe that while other programmable switches (e.g., Broadcom Trident [10]) may have different hardware limitations and resource allocation constraints, the architectural bottlenecks for sketches are likely similar. We leave it as future work to extend this thesis to other programmable switches and other hardware targets.

1.7 Outline

The rest of this thesis is organized as follows: chapter 2 discusses the taxonomy of network telemetry and prior work in this space. In chapter 3, we present SketchLib, a library of API calls of per-sketch optimizations for data plane resources to enable a single sketch instance. In chapter 4, we introduce Sketchovsky, which proposes many cross-sketch optimizations that enable ensembles

of sketch instances. Specifically, this chapter compiles optimization building blocks and greedy heuristic algorithms, in order to find the best strategy for using them. In chapter 5, we propose an auto-code composition framework that translates those strategies into optimized code. Next, chapter 6 discusses how to solve the counter contention problem by optimizing read and reset operations in the sketch control plane implementations. Finally, chapter 7 summarizes lessons learned, future research directions, and our conclusion.

Chapter 2

Related Work: A Taxonomy of Network Telemetry

We present a taxonomy of network telemetry in Table 2.1 to explain why existing approaches do not satisfy all four requirements of (1) low resource footprint, (2) high measurement accuracy, (3) high packet processing speed, and (4) support for diverse measurement results. This taxonomy helps us put our contributions in context; approaches that are taken in this thesis are emphasized by bold text.

<table>
<tr><td rowspan="7">Network Telemetry</td><td>Packet-level</td><td colspan="4">NetSight [53], EverFlow [111], dShark [24], INT [101]</td></tr>
<tr><td rowspan="6">Flow-level</td><td>Expressive Query Language</td><td colspan="3">Marple [83], Sonata [52], Newton [110]</td></tr>
<tr><td>Sampling-based</td><td colspan="3">NetFlow [35], sFlow [95]</td></tr>
<tr><td rowspan="4">Sketch-based</td><td>Software Switch</td><td colspan="2">NitroSketch [73], SketchVisor [57]</td></tr>
<tr><td rowspan="3">Programmable Switch</td><td rowspan="2">Single Sketch</td><td>UnivMon [71], R-HHH [22], FCM [89]
ElasticSketch [100], SketchLearn [58]
CocoSketch [106]</td></tr>
<tr><td colspan="2">Multiple Sketches (Our Approach)</td></tr>
</table>

Table 2.1: A Taxonomy of Network Telemetry

2.1 Packet-level telemetry

Packet-level telemetry is useful to diagnose network issues such as route error, route loop, or congestion link detection. The telemetry result is measured by tracking individual packets by in-network packet dump [24, 53, 111] or appending route information to packet header [24]. EverFlow [111] enables efficient in-network packet capture by tracking only a portion of traffic, and dShark [101] provides a programming model to support packet-level telemetry queries in a dis-

tributed manner based on packet capture fed by EverFlow. In-band network telemetry (INT) [24] makes switches attach information to each packet, such as switch ID and queue status, which can be collected at the endpoint where the packet arrives. Although these packet-level telemetry results help debug network issues, they cannot provide useful flow-level measurement results.

2.2 Expressive query language

Expressive query languages allow network operators to run diverse flow-level measurement tasks on hardware switches. Marple [83] proposes a programming language model for flow-level queries with basic primitives such as filter, map, groupby, and zip operations, along with novel design contributions for hardware switches to support these operations efficiently. Sonata [52] takes a further step to run this expressive query language even more efficiently by partitioning the computation of queries into the hardware switch data plane and stream processor running on nearby CPU servers. Newton [110] contributed to the dynamically changing queries to run on the programmable switch. Although these expressive query languages help generate diverse flow-level measurement results, they are based on precise measurement techniques that aim to produce 100% accurate results. Due to this reason, this approach requires too many hardware resources, and it is often not feasible for today's network switches. Thus, this approach violates the first requirement, (1) low resource footprint.

2.3 Sampling-based approach

Because the precise measurement approach incurs high resource overhead, the approximation-based approach is often used instead. Specifically, packet sampling is a state-of-the-art technique that processes only sampled packets on the network traffic. NetFlow [35] samples packets by a configurable ratio (e.g., sample 1 packet out of 1000) and maintains an active working set of flow information in a hash table in the data plane. On the other hand, sFlow [95] samples packets and mirror them to a separate collector where flow information is maintained. While packet sampling

effectively reduces resource overhead, sampling-based approaches suffer from low accuracy [40, 42, 87]. Thus, this approach does not satisfy the second requirement, (2) high accuracy.

2.4 Sketching algorithms on software switch

Due to the inaccuracy problem of packet sampling, an alternative class of techniques called sketching algorithms or sketches attracted considerable attention due to their resource efficiency and high accuracy. As software packet processing is an essential pillar of modern data center networks, several recent works optimize sketching algorithms to run on software switches efficiently. SketchVisor [73] reduces CPU overhead by activating the fast path under high traffic load with slight accuracy degradations. NitroSketch [73] takes a more fundamental approach by leveraging geometric sampling for updating counter arrays to reduce CPU overhead without sacrificing accuracy. While these innovations for running sketches on software switches continue to be important, software switches cannot keep up with hardware switches' fast processing speed. As a result, running sketches on software switches does not satisfy our third requirement of (3) high packet processing speed.

2.5 Single sketch instance on programmable switch

Instead of a software switch, a programmable hardware switch is more promising for running sketches, as it can process Tbps scale of high packet processing speed with the flexibility to program and run any sketching algorithms. However, previous work only focuses on running a single sketch instance, and this approach is fundamentally limited in terms of the coverage of measurement results. Researchers focused on developing new and general sketching algorithms to cover multiple measurement results. [22, 58, 71, 89, 100, 106]. For example, by running R-HHH [22] or CocoSketch [106], heavy hitter detection on multiple flowkeys is supported. By running UnivMon [71], ElasticSketch [100], SketchLearn [58], or FCM [89] on a predefined flowkey (e.g., source IP), then multiple statistics—such as identifying heavy hitters, entropy, and cardinality—can be

measured simultaneously. Despite these achievements, developing a new sketching algorithm to support both multi-flowkey and multi-statistics measurement results and making it hardware-friendly is challenging. Thus, running a single sketch instance cannot satisfy the final requirement, (4) support for diverse measurement results.

2.6 Multiple sketch instances on programmable switch (our approach)

This thesis presents a systematic approach to run multiple sketch instances concurrently on programmable switches as it has the potential to meet all four requirements: (1) high accuracy, (2) low resource footprint, (3) high packet processing speed, and (4) diverse measurement results. Instead of packet sampling, sketching algorithms can provide both accuracy and low resource usage. Programmable hardware switches are more effective than software switches at achieving high packet processing speed. By running multiple sketch instances, we can achieve the requirement of diverse measurement results.

This thesis focuses on a systematic approach for enabling efficient sketch realizations on programmable switches via optimizations and APIs, but it does not explore a theoretical approach to developing new and more general sketching algorithms. We argue that bridging the gap between sketches and programmable switches by this systematic approach is an effective and timely way to realize performant, rich, and practical sketch-based network telemetry, because we can take advantage of many already-existing sketching algorithms [36, 54, 103]. However, developing new sketching algorithms continues to be valuable and important. Our systematic approach is orthogonal but also complementary to this theoretical approach, as systematic optimizations can inspire new general sketching algorithms to be compatible with programmable switches.

Chapter 3

SketchLib: Optimizing A Single Sketch Instance on Programmable Switches

Implementing a single sketch instance on programmable switches remains an open challenge. For example, off-the-shelf sketch implementations often cannot run with the desired accuracy levels due to insufficient hardware resources (see §3.1). Indeed, some proposed sketches (e.g., [71]) are infeasible as implemented, and even those that are feasible consume too many significant resources.

Even if more hardware resources may become available, so too do operators' demands of in-switch applications, and the resources consumed by sketches will be unavailable for other switch functions. Thus, it is essential to explore if, and how, we can efficiently realize a single sketch instance on programmable switches. Specifically, we focus on programmable hardware switches based on the Reconfigurable Match-Action Tables (RMT) paradigm [9]. We identify and analyze four key resource bottlenecks for realizing sketches on RMT switch hardware:

- *Hash calls:* Sketches make a number of counter updates based on independent hash functions, requiring a large number of hash calls in hardware.
- *Memory accesses:* Sketches need to access on-chip memory (e.g., SRAM) for counter updates, but the number of memory accesses per packet is limited in hardware.

- *Pipeline stages:* Some sketches need to select a subset of counter arrays for counter updates [43, 66, 71]. However, implementing this naively can cause a long chain of sequential computation dependencies, which stresses the limited number of switch pipeline stages.
- *Resources for tracking heavy flowkeys:* Some sketches need to keep track of the flowkeys identifying the heavy hitters (e.g., 5-tuple, source IP, or destination IP) [22, 32, 37, 65, 71]. Common structures such as priority queues or heaps used in software are not supported on programmable switches, and existing solutions entail undesirable tradeoffs that affect miss rate, data plane memory, and control plane bandwidth.

Having identified these bottlenecks, our contribution is a careful synthesis of known and novel optimizations into a practical library for enabling efficient sketch implementations atop the RMT architecture. While some of these build on prior work in optimizing sketching for other targets such as software switches, FPGAs, and embedded platforms [73, 97, 99, 103], our main contribution is in realizing feasible and effective optimizations based on our bottleneck analysis and translating them into the switch hardware setting. For example, to reduce the number of hash calls, we identify opportunities to consolidate and reuse hash results across multiple counter updates [45, 64]. Similarly, we identify an opportunity to reduce the pipeline stages by eliminating code dependencies based on longest prefix matching using TCAM [103]. We reduce the memory accesses by refactoring sketch algorithms and removing unnecessary memory accesses. We also develop practical flowkey tracking mechanisms that are feasible in hardware. Note that all optimizations preserve correctness while reducing the resource footprint.

To make it easy for sketch developers to benefit from these optimizations with minimal effort, we implement *SketchLib*, an easy-to-use API using the P4 language [29]. These optimizations can be applied to a broad spectrum of classical sketches (e.g., [32, 37, 65]) and recent innovations (e.g., [22, 71]). We qualitatively evaluate the suitability of SketchLib for 19 published sketches and observe that 15 of them can be expressed and can benefit from one or more of our optimizations. We acknowledge that not all optimizations are applicable for every sketch, and we envision sketch developers using our API to adopt the relevant optimizations.

We quantitatively evaluate the utility of SketchLib in improving 7 of the 15 applicable sketches covering a diverse set of target measurement tasks: Count Sketch (CS) [32], PCSA [46], MRAC [66], Multi-resolution Bitmap [43], Hierarchical Heavy Hitters [22], and UnivMon [71]. Our evaluation using a range of packet traces empirically confirms that our optimizations provide similar accuracy ($\leq 1.9\%$) with substantially (up to 96%) reduced resource usage. Furthermore, some complex sketches (e.g., UnivMon) that were previously infeasible on current hardware become feasible.

Contributions and Roadmap. To summarize, we make the following contributions:

- **Bottleneck Analysis (§3.1)**: We identified four key resource bottlenecks for sketch implementations on the programmable switch data plane.
- **Optimizations (§3.2)**: We identify and synthesize practical correctness-preserving optimizations to address the bottlenecks for sketches on the hardware switch.
- **API Implementation (§3.3)**: We design a convenient API to make our optimizations easy to use for developers who implement sketches on RMT programmable switches. We verified significant resource benefits on a broad range of sketching algorithms. [1]

3.1 Motivation: Bottleneck Analysis

In this section, we consider three exemplar sketches (single-level: count sketch; multi-level: R-HHH and UnivMon) to quantify the resource bottlenecks of sketch implementations on programmable switches. We implement them in P4 based on the logic described in prior work [32, 43, 46, 47, 66, 71] similar to the structure presented in Figure 1.2.

3.1.1 Methodology and Setup

Configuring sketches. Running sketches entails picking parameters (e.g., the count (R) and size (W) of counter arrays) to trade-off the accuracy vs. resource use. We envision an operator configuring the sketches with some target accuracy goal, e.g., the median error should be less

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

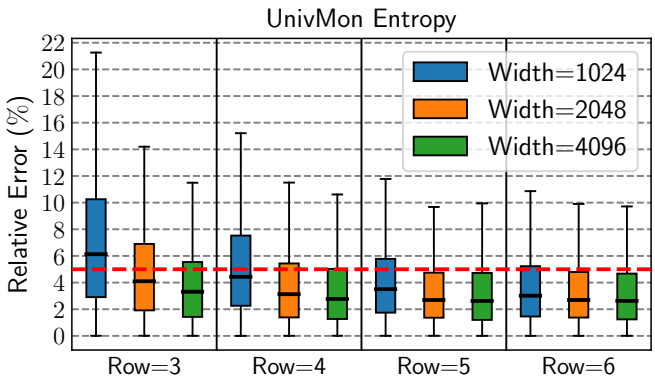


Figure 3.1: UnivMon entropy estimation error for different configurations. Dotted red line indicates target accuracy.

than 5%. Operators can use trace-driven analysis to pick reasonable operating regimes for these parameters.

As an example, Figure 3.1 illustrates this trade-off for entropy estimation using UnivMon. The figure shows the estimation accuracy using an hour-long inter-ISP packet trace captured on a OC-192 link [1] with different parameters R and W for count sketches and $L = 16$ levels. We see that the error decreases as we increase the number of rows (R) and width (W) for count sketches. Naturally, the higher accuracy configurations incur more hardware resources. For our bottleneck analysis, we target an accuracy of under 5% median error (dotted red line in Figure 3.1), which we achieve with minimal resource use with the configuration $R = 3$ and $W = 2048$. We repeat the analysis for count sketch and R-HHH and consider a similar operating regime for these sketches as well.

Estimating resource footprint. For a given set of sketch parameters, the most direct way to measure the required hardware resources is to compile the code and run it on the hardware. However, this limits our analysis to currently available platforms. In order to support "what if" analysis for hardware with different resources (e.g., more pipeline stages), we extended an existing open source tool for mapping P4 programs to the RMT hardware, which we will refer to as RMT resource mapper [60]. Specifically, we address three issues to extend RMT resource mapper for our analysis:

- *Inputs:* The input to Tofino compiler is P4-16 code with some hardware-specific primitives whereas RMT resource mapper accepts only P4-14 code [14]. Thus, we first convert our P4-16 code into equivalent P4-14 code. Then, we convert Tofino-specific primitives to equivalent ones specified in the language specification. For instance, we replace Tofino-specific primitives for accessing registers with *register_read* and *register_write*.
- *Resources:* First, RMT resource mapper does not model hash calls and SALUs in their original design. Thus, we extend RMT resource mapper to model hash calls and SALUs and added the corresponding optimization constraints for assignment of these new resources. Second, we observed that RMT resource mapper assigns memory even for tables without any entries and action data. To fix this disconnect, we decouple the memory/table assignment.
- *Objective:* RMT resource mapper supports different optimization objectives: minimizing latency, power, or pipeline stages. The objective of minimizing pipeline stages is the most suitable because it gives resource mappings that are closest to those generated by the Tofino compiler.

With these fixes in place, we validate our extensions by comparing the resource usages between RMT resource mapper and Tofino compiler for a wide range of sketches and configurations, for the cases that are feasible on current hardware. Based on the measurement results, we conclude that our modified RMT resource mapper is a good proxy of Tofino compiler, as it captures the relevant resource constraints, and its resource allocation results are close to that of Tofino compiler (see §A.1 for more details).

3.1.2 Identified Bottlenecks

Using the RMT resource mapper, we measure the usage of each type of resource based on the output of the compiler for three sketches: Count Sketch, R-HHH, and UnivMon. For the purpose of bottleneck analysis, we use a base configuration of: $W = 2048$, $R = 3$, and $L = 16$ for UnivMon and $L = 25$ for R-HHH [22], which provides an error of up to 15% when processing packets from an inter-ISP packet trace [1]. We choose the value for L from the original papers [22, 71].

Figure 3.2 illustrates how the use of four bottleneck resources depends on key sketch parameters. While the amount of available hardware resources can differ across hardware vendors and versions, we see that resource usage increases rapidly as we need more counters to meet higher accuracy requirements. While we cannot report exact resource usages due to proprietary reasons, we note that UnivMon and R-HHH are infeasible today on the hardware for many configurations. Perhaps more importantly, switches must also support tasks other than sketch-based telemetry (e.g., [59, 79]). Thus, it is critical to reduce the resource footprint of the sketches to ensure they can co-exist with other switch functions.

B1. Hash calls: Recall that count sketch needs $2R$ hash calls per packet, matching the results in Figure 3.2(a). UnivMon and R-HHH execute one count sketch per level L. As a result, R-HHH needs $L \cdot 2R$ hash calls. UnivMon needs to compute an additional L 1-bit hash calls in its sampling stage, adding up to $L \cdot (2R + 1)$ hash calls.

At first glance, it may seem that the number of hash calls is not a bottleneck as these are called on demand per packet. While this is true in a software setting, where only the required calls are performed on demand, hashing on hardware is different. On a hardware switch, all hash calls appearing in the code need to be *pre-allocated*, since execution at line rate must be guaranteed for all possible execution paths. This increases resource requirements, even if hash calls need not be executed. For example, even though UnivMon and R-HHH (Figure 1.2) may not update all levels of count sketches for all packets, all hash resources must be pre-allocated.

B2. Memory accesses: Count Sketch maintains R counter arrays and for each row it must read one counter from memory and update its value. This means that count sketch needs R counter updates per packet, requiring R Stateful ALUs (SALUs) as shown in Figure 3.2(b). When the compiler compiles the P4 code of UnivMon and R-HHH in Figure 1.2, it allocates separate memory regions and SALUs for each level of count sketches, thus SALU requirements are proportional to the number of levels L. Since we need R memory processes per packet for the count sketch at each level, we need a total $L \cdot R$ SALUs for R-HHH and UnivMon. This makes memory access hardware

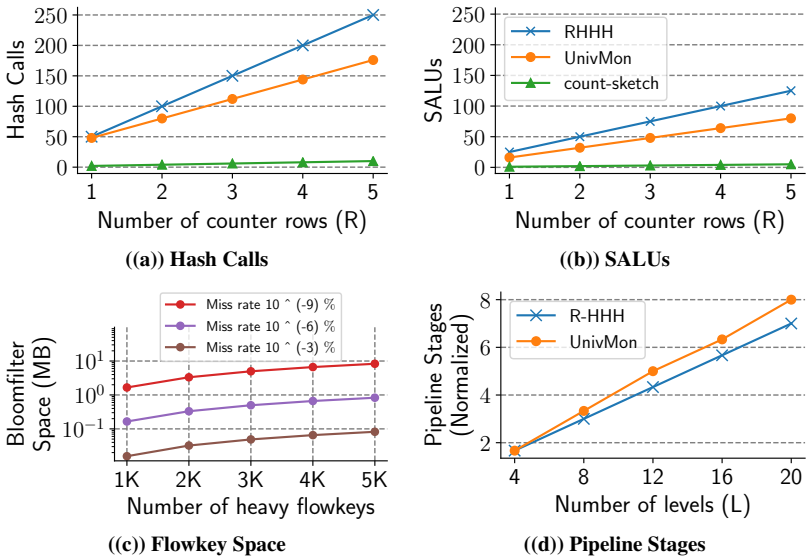


Figure 3.2: Resource bottlenecks for sketch implementations.

(SALU) a bottleneck (Figure 3.2(b)). Similar to hash resources, SALUs need to be *pre-allocated* at compile time, even if they may remain unused.

B3. Resources for tracking heavy flowkeys: Many sketches need to track heavy flowkeys to enable downstream analytics tasks. Typically, these sketches store heavy flowkeys in a separate data structure (e.g., heap or priority queue) [32, 71].

In practice, however, the exact details of if/how this can be realized on switch hardware are unclear. Specifically, a heap or priority queue, while feasible in software switches, is too complex to be implemented on the programmable hardware switch. Alternatively, the data plane can relay all flowkeys to the switch control processor or record all flowkeys in the data plane. However, these are not feasible: e.g., bandwidth between the data plane and the control plane is limited, and data plane memory is also limited. Some sketch constructions store heavy flowkeys together with the counters [23, 58, 88]. However, these are infeasible at line-rate on today's RMT switches.[2]

To reduce the memory use, prior work proposed an optimized baseline: when a packet arrives, it checks whether the frequency of a flowkey has exceeded the threshold by querying the sketch counter, and if so, it reports the key to the control plane [59, 71, 72]. Unfortunately, this still

[2]Specifically, HashPipe [88] cannot be directly implemented on RMT architecture due to complex memory access patterns (see [23] for more details). Precision [23] requires recirculation, which means some packets must go through entire pipeline again.

	Miss rate	CP bandwidth	DP resource
Recored every key in the DP	Zero	Low	Infeasible
Report every key to the CP	Zero	Infeasible	Low
Report heavy keys to the CP	Zero	Infeasible	Low
Report non-duplicate heavy keys	Low	Low	High

Table 3.1: Strawman solutions for tracking heavy flowkeys (CP: control plane, DP: data plane).

presents a problem, as "heavy" flowkeys may be reported redundantly every time a packet arrives and needs more control plane bandwidth. To avoid duplicate reporting to the control plane, we could use a Bloom filter to check if a heavy key has already been reported [59]. However, we need to configure the Bloom filter (i.e., bitmap size and number of hash functions) to have especially low false positives, since a false positive in the Bloom filter for the duplicate check is a potential miss of a heavy flowkey. Figure 3.2(c) confirms this trade-off; we can configure the Bloom filter depending on the target miss rate, and we find the memory footprint is correspondingly higher (We use 3 hash functions for Figure 3.2(c)). Although using a Bloom filter might be a valid approach if we allow some missing heavy flowkeys, we argue a design that targets a zero miss rate is more desirable.

We implement four possible strawman solutions to report heavy flowkeys and run microbenchmarks on a Tofino hardware switch, to understand a trade-off between the accuracy and the resource consumption. Table 3.1 summarizes our analysis and shows that we have an undesirable trade-off between the miss rate of heavy flowkeys, data plane resources (memory for keys and hash calls for Bloom filters), and the control plane bandwidth (for reporting keys).

B4. Pipeline stages: So far, we have implicitly assumed that the switch has a single pool of resources (i.e., SRAM/TCAM, SALUs, and hash calls) that can be allocated from the switch to the sketch operations. In reality, the resources are partitioned across the pipeline stages. This impacts resource use in two ways. First, before an operation can be assigned to a stage, all required resources need to be available on that stage. If that is not the case, it needs to be moved to the next stage. Second, if there is a dependency between two operations, e.g., $O_1 \rightarrow O_2$ in the code, then O_2 must be placed on a later stage than O_1, even if there are unused resources available on stages

earlier in the pipeline. For example, the sequential `if` clauses used by UnivMon (Figure 1.4) create sequential dependencies between the if clauses.

This means that, depending on resources required by operations and dependencies between them, the compiler will only be able to use a subset of the resources on the switch. To account for this, we consider pipeline stages as a separate resource. Figure 3.2(d) shows the number of pipeline stages needed as a function of level L if we respect this architectural constraint. We see that UnivMon requires similar or more pipeline stages than R-HHH with the same configuration parameters, and the gap increases along with the number of levels. This is a direct result of the sequential dependencies in UnivMon. The number of pipeline stages used is measured by running the RMT resource mapper.

3.2 Optimizations

Next, we present a series of optimizations to address the resource bottlenecks we identified earlier. For each optimization, we discuss the key idea, before then discussing the correctness and applicability constraints of that key idea. Some of these optimizations (e.g., O1, O3, O4) have appeared in earlier theoretical efforts and demonstrated in other settings (e.g., FPGA, software switch). Our contribution here is translating these ideas into hardware switches. Others (O2, O5, O6) are novel to the best of our knowledge. As summarized in Table 3.2, our optimizations can be applied to a broad spectrum of published sketches for telemetry and benefit 15 out of the 19 sketches listed. Some sketches that are outside of our scope cannot be supported, as they either use: (1) processing logic that is infeasible in hardware (i.e., Hashpipe); (2) counter data structures different from sketches (i.e., BeauCoup); or (3) complex processing patterns such as packet recirculation (i.e., Precision).

Sketch Type	Sketch Name	Feasible on HW?	Applicability of SketchLib
Frequency Estimation / Heavy Hitters	Count-Min [37]	Yes	O6
	Count Sketch [32]	Yes	O1, O6
	MRAC [66]	Yes	O3, O5
	Hashpipe [88]	No	N/A, due to infeasible logic
	Precision [23]	Yes	No, uses packet recirculation
Hierarchical Heavy Hitters	RHHH [22]	Yes	O1, O2, O5, O6
	HHH [38]	Yes	O1, O6
Cardinality	PCSA [46]	Yes	O3, O5
	MRB [43]	Yes	O3, O5
	LogLog [41]	Yes	O3
	HyperLogLog [47]	Yes	O3
Entropy	EntropySketch [67]	Yes	O1
Change Detection	K-ary [65]	Yes	O1, O2, O6
Super Spreaders	SpreadSketch [92]	Yes	O3, O5
	BeauCoup [34]	Yes	No, non-counter based sketch
General	UnivMon [71]	Yes	O1, O2, O3, O4, O5, O6
	FCM [89]	Yes	O6
	SketchLearn [58]	Yes	O2
	ElasticSketch [100]	Yes	Not applicable

Table 3.2: Applicability of SketchLib on existing sketches.

3.2.1 Optimizing Hash Calls

Both single- and multi-level sketches need to compute multiple hash functions, resulting in high hash call usage in the hardware pipeline. We describe two optimizations: consolidating short hash calls and reusing hash calls.

Optimization 1. Consolidate many short hash calls. We observe that many hash calls only need short-length (e.g., 1-bit) hash results. For instance, count sketch (Figure 1.1) computes a series of 1 bit hash calls, s_1 to s_R. Similarly, UnivMon (Figure 1.2 (b)) computes h_1 to h_L. We can reduce the number of hash calls by consolidating many short hash calls, as long as the inputs to the hash calls are the same.

Consider a count sketch with $R \times W = 3 \times 512$ counters. Per row, we need two hash results: a 9-bit (i.e., $\log_2 512 = 9$) hash to index into the counter array and a 1-bit hash for the "sign" of the counter. Instead of using $3 \cdot 2 = 6$ hash calls, we can instead use one hash call that returns a 30-bit result to provide the 6 hash calls as in Figure 3.3. Note that splitting a long hash result only needs

simple hardware shift and bit mask operations. R-HHH and UnivMon are also benefited as they use multiple count sketches. Further, UnivMon uses many 1-bit hash calls in its sampling stage.

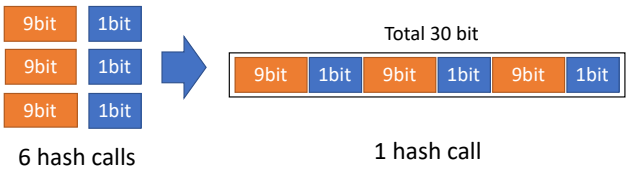


Figure 3.3: Optimization 1 reduces hash calls for count sketch.

Correctness and applicability: For this optimization to be valid, the short-bit hash results that are split from the longer hash result must use the same flowkey as the input and, if required by the sketching algorithm, be pairwise independent [32]. Independence is achieved by randomly picking (different) seeds in practice for hash calls [22, 71, 100]. Theoretical analysis in other contexts [45, 64] shows that using different bits from the same hash call can also provide independence. Empirically, recent work [97] shows no accuracy loss for Univmon, and our results (§3.4) confirm this with other sketches listed in Table 3.5. In addition, hash calls need to be short, so that the sum of hash bit length is less than the length of one call (e.g., 32 bits). Fortunately, many single- and multi-level sketches [22, 32, 43, 46, 47, 66, 71] satisfy this condition.

Optimization 2: Reuse the hash calls across levels for multi-level sketches. Our second insight is that we can reuse the hash calls if there are no independence requirements across them; i.e., they can use the same seed. Although hash independence is usually required across different counter arrays within a single level sketch, it is not required *across* levels [30]. Thus, we can use the *same* hash seed cross different levels for multi-level sketches.

Specifically, the original implementations of R-HHH and UnivMon (see Figure 1.2) use a *different* hash seed in each of the CS_level_i count sketch executions. We can modify the code to reuse the same hash seed and reuse hash results when independence is not needed. This optimization reduces the number of hash calls significantly. For example in Figure 1.2, R-HHH and UnivMon each have a set of hash calls F_i as $\{f_{i1}, f_{i2}, ...f_{i(2R)}\}$ at each level i of count sketch, resulting in $L \cdot 2R$ hash calls. By simply changing all of F_i to F_1, we reduce hash call usage from

Flowkey	Seed	Additional Condition	Opt
same	diff.	Sum of hash bit length is less than max capacity	O1
same	same	-	O2
diff.	same	One level of hash calls is executed	
diff.	diff.	-	-

Table 3.3: Conditions for optimization 1 and optimization 2.

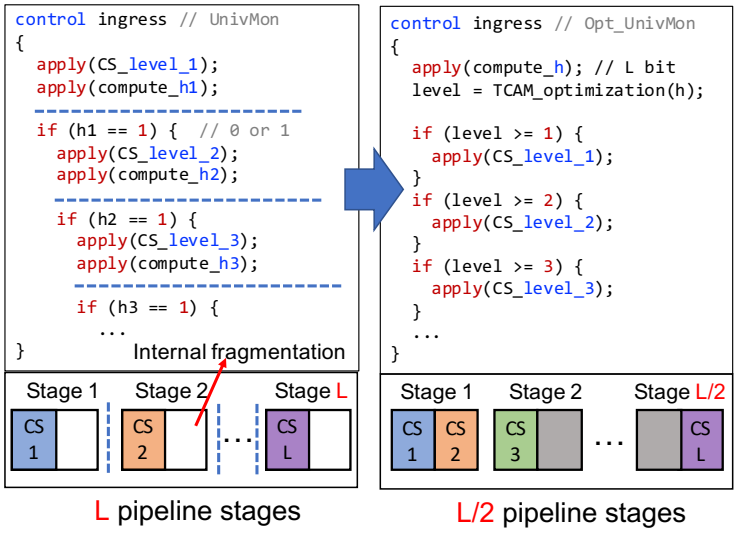


Figure 3.4: Optimization 3 removes the sequential computation dependency and reduces the usage of pipeline stages.

$L \cdot 2R$ to $2R$. For R-HHH, the result of F_1 is used to update one selected level of count sketch, and for UnivMon, result of F_1 can be used to update potentially multiple levels per packet.

Correctness and applicability: Reusing seed values across levels does not affect the theoretical independence requirements [30]. We empirically confirm in the evaluation that this optimization achieves similar accuracy (§3.4.1).

Table 3.3 summarizes the conditions under which the two hash optimizations are used. Note that for O2, if different levels' hashes have diverse output bit-length requirements, the hash call with the longest output bit-length will be used to supply hash results with various bit lengths. We also need to make sure that the hash seeds are either the same in the first place or can be set to be the same for O2 to apply.

3.2.2 Optimizing Pipeline Stages

The sequential `if` clauses are observed in both single and multi-level sketches. This creates sequential compute dependencies and entails high usage of pipeline stages.

Optimization 3: Avoiding the sequential `if` clauses using a longest prefix match. To explain this optimization, we use UnivMon (Figure 3.4) as an example. Deciding which levels to be updated for each flowkey creates a logical *dependency* between levels. Specifically, level i+1 needs to be updated only if the value of h_i returns 1 for hash functions $h_i : [n] \rightarrow \{0/1\}$. These L-level dependencies lead to an implementation as Figure 3.4-left using sequential `if` clauses with hash values (h_1,h_2,...,h_L).

To address this bottleneck, our insight is that the number of leading 1-bits in (h_1,h_2,...,h_L) represents the sequence of "true" conditions in the `if` clauses. We observe that this is equivalent to the *longest prefix match* (LPM), which can be computed efficiently in hardware. That is, we can compute L hash bits together using a single L bit hash and use LPM to identify which layers need to be updated. This LPM operation is realized via TCAM as shown in Figure 3.5. We insert rules with 1- and wildcard bits corresponding to each level and perform LPM to obtain the last level of UnivMon for each flowkey. LPM is relatively cheap—can be done in one pipeline stage using a small amount of TCAM. With this optimization, we can reduce the usage of pipeline stages by half if one count-sketch consumes half of the resources in one pipeline stage (Figure 3.4-right).

Correctness and applicability: Our refactored implementation has the same functionality, resulting in the same updates to the sketch arrays. This optimization applies to many single and multi-level sketches that build on the power-of-two choices observation [43, 46, 47, 66, 103].

3.2.3 Optimizing Memory Accesses

Sketches require memory accesses for their counter updates, leading to high SALU usage. This becomes especially significant for multi-level sketches.

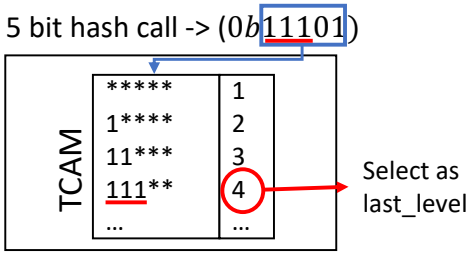


Figure 3.5: Replacing the sequential `if` clauses via TCAM.

Optimization 4: Refactor multi-level sketches to update one level per packet. We refactor multi-level sketching algorithms and their code to guarantee only one level is updated per flowkey. Recall that UnivMon needs to update one or more levels of count sketch (CS) for each packet with its flowkey. In Figure 3.6 (top), a flowkey of packet K_{green} updates three levels, while K_{gray} updates two levels, and K_{red} updates all levels of count sketch. Instead, our modified algorithm is guaranteed to update only the "last" level for each packet, as shown in Figure 3.6 (bottom). The modified algorithm becomes structurally similar to other multi-level sketches that natively update only one level [22, 43, 46, 66]. As a result, the processing overhead is significantly reduced.

This "update-last-level" idea was also proposed to optimize UnivMon for embedded platforms [99] and software switches [73, 97]. Our contribution here is: (1) to extend this to programmable switches and (2) to generalize the idea to support updating arbitrary levels. Based on the algorithmic design, different multi-level sketches may require different optimization strategies to update a level (e.g., RHHH [22] modifies HHH [38] by randomly selecting a level to update). To implement this optimization, we can insert user-defined ternary rules in TCAM (as O3) to classify packets into different levels in a multi-level sketch.

Correctness and applicability: By construction, our modified algorithm provides equivalent functionality as the original version. As shown on the right side of Figure 3.6 with K_{green} flowkey as an example, Levels 1 and 2 do not need to be updated anymore. Level 3 has the estimated flow count for this particular flow with the same or better accuracy since Level 3 only processes a smaller amount of traffic than Levels 1 and 2. Thus, the estimated count of K_{green} from Level 3 can be

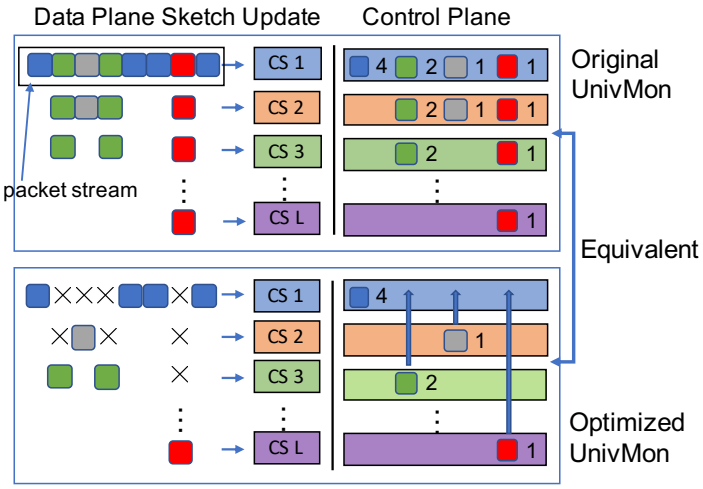


Figure 3.6: UnivMon updates only the last level per packet. CS stands for Count-Sketch.

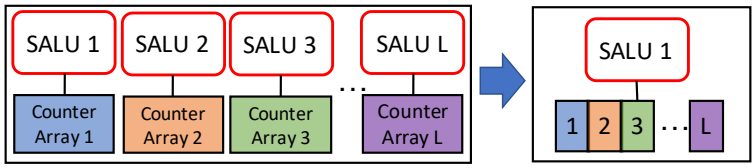


Figure 3.7: Optimization 5 removes unnecessary allocated SALUs by rewriting P4 code.

reused for Levels 1 and 2. This applies to all other flowkeys during the offline estimation in the network control plane.

To apply this optimization, a multi-level sketch should meet two conditions: (1) the original algorithm has multiple sketch updates per packet, and (2) it is algorithmically correct to reduce the multi-level updates to one per packet. That said, we acknowledge that there are scenarios where this optimization is not directly applicable. For instance, it is not obvious if/how we can refactor some multi-level sketches such as SketchLearn [58] to update only one level per flowkey (if possible). This requires future research.

Optimization 5: Remove unnecessary SALU operations. A multi-level sketch maintains multiple independent levels of sketches. For each counter at each level, the compiler statically allocates an SALU for memory access. This results in high SALU usage, even if only one level needs to be updated per packet; i.e., usage is the same as updating all levels.

We can remove unnecessary SALUs when only one update is needed per packet. The compiler inefficiently preallocates SALUs for all possible memory accesses because it is difficult to auto-

Resource Bottlenecks	Optimizations	API
Hash Calls	O1. Consolidate short-bit hash calls	`hash_consolidate_and_split()`
	O2. Reuse hash calls across levels	`select_key_and_hash()`
Pipeline Stages	O3. Remove sequential if clauses using TCAM	`tcam_optimization()`
SALUs	O4. Update only one level per flowkey	-
	O5. Rewrite P4 code to reduce memory accesses	`consolidate_update()`
Resources for tracking heavy flowkeys	O6. Use a hash table to remove duplicate flowkeys	`heavy_flowkey_storage()`

Table 3.4: The relationships among the bottlenecks, optimizations and API calls.

matically discern that only one update is needed at runtime. Our optimization restructures the P4 code to make this explicit for the compiler that only one count sketch update is needed per level. Instead of using separate counter arrays located in different switch memory regions, we consolidate the counter arrays of all levels in a single array located in one region of memory. This is possible because SALU can support random access, thus based on the selected level, we can compute the corresponding index value to access the consolidated register. Figure 3.7 illustrates this SALU optimization. This optimization reduces the SALU requirements for multi-level sketches by a factor of L (the number of levels, e.g., 25 for R-HHH [22]).

Correctness and applicability: This technique does not affect accuracy because the modified code has the same functionality as the original version. We can apply the optimization to multi-level sketches that have the property of updating only one level per flowkey. There are many multi-level sketches satisfying this property [22, 43, 46, 66, 71].

3.2.4 Optimizing Heavy Flowkey Reporting

Optimization 6: Use a hash table and an exact-match table for checking duplicate flowkeys. As discussed in §3.1.2 B3, prior efforts [59, 72] use Bloom filters as the duplicate checker but the false positives from the filters will cause misses of heavy flowkeys, unless a very large Bloom filter is used. To improve this tradeoff between miss rate and data plane resource, we use a hash table and an exact-match table to check duplicates. Specifically, the hash table stores heavy flowkeys and detects whether there is a collision. For each heavy flowkey, if it is already stored in the hash table or exact-match table, it will not be reported to the controller; otherwise, it will be inserted to the hash table. However, if this flowkey collides with another key in the hash table, then it will be

reported to the controller, which then inserts this flowkey to the exact-match table to filter future duplicate keys. In this way, we can ensure a zero miss rate on reporting heavy flowkeys.

Correctness and applicability: This optimization ensures a zero miss rate of heavy flowkeys because when collisions happen in the hash table, the flowkeys are reported to the control plane and inserted to the exact-match table (as a secondary duplicate checker). No unique heavy flowkeys are dropped in this mechanism. Compared to Bloom filters, this approach adds some additional control plane bandwidth when collisions happen in the hash table. As we evaluate in §3.4.5, this added bandwidth is small (e.g., 2% increase). This optimization can be applied to both single- and multi-level sketches requiring heavy flowkey tracking [22, 32, 71].

3.3 SketchLib API

In this section, we present our P4 API for helping sketch developers to use our optimizations. For each API call, we show the implementation for the macro and how the macro is used. SketchLib API supports both P4-14 and P4-16 [17]. Table 3.4 maps the optimizations to the API calls.

`hash_consolidate_and_split(Key,Seed,List(BitLen),BL_sum,List(Mask))` [3] reduces hash calls by consolidating small bit hash calls (O1). Figure 3.8 shows how a sequence of short hash calls is replaced by a macro that uses only a single hash call with length the sum of all `BitLen` of the shorter hashes. The resulting hash value is then partitioned in shorter hashes. For P4-14, we split the result using `modify_field_with_shift(dst, src, shift, mask)` primitive (i.e., `dst = (src » shift) & mask`) where `mask` is a series of 1's with `BitLen` as shown. For P4-16, the same principle is applied, but bit slice operation (e.g., `h[BL1:0]`) is used. Note that the macro specifies both the number of short hashes being merged (*List*) and the names of the short hashes, so multiple macros must be defined if O1 is applied multiple times.

[3] While there is no concept of `List` in P4, we use it to describe the type of parameters conceptually throughout this section. In our API implementations, it is converted to multiple parameters; e.g., `List(BitLen)` → `(BL1, BL2, BL3)` as shown in Figure 3.8.

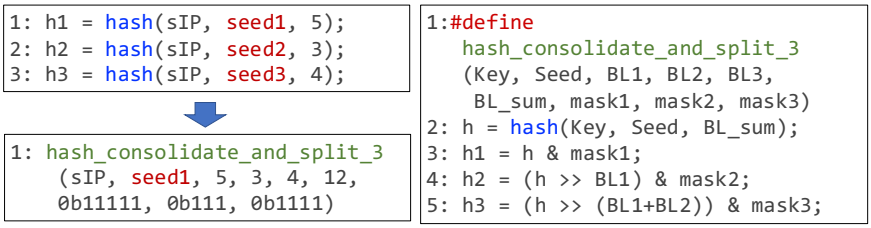


Figure 3.8: `hash_consolidate_and_split()`

`select_key_and_hash(List(Key),Level,Seed,BitLen)` implements O2 for the case one of the several hash calls with different `Key` and same `Seed` is selected for execution. Here, we can select the key in advance and use only one hash call to get the result as in Figure 3.9. For instance, R-HHH can be optimized by using this API call. The example shown is a single hash call, but if multiple hash calls are needed (e.g. sketch with R =5 needs 5 hash calls), the number of them can be increased. For the sketches that share the same `Key` and `Seed` (e.g., UnivMon), no separate API call is necessary since the hash value can simply be reused.

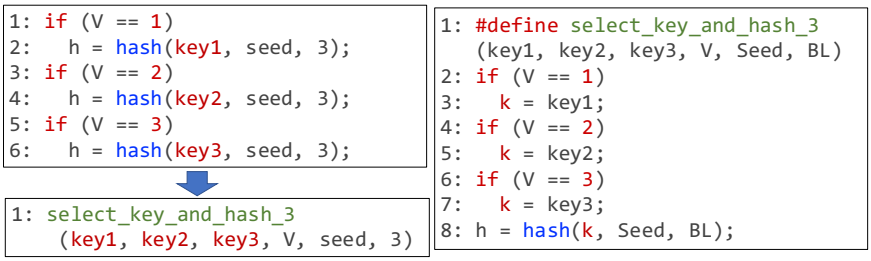


Figure 3.9: `select_key_and_hash()`

`tcam_optimization(Hash_Result)` implements O3 to remove sequential `if` clauses by applying an equivalent a LPM table which uses TCAM to which levels need to be updated. The macro implements the use of the TCAM to look up the level (see Figure 3.4).

`consolidate_update(Level,Index)` implements O5 to reduce memory accesses, as illustrated in Figure 3.10. `Level` indicates the selected counter array and `Index` references the location for the memory update within the counter array. The API call consolidates counter arrays and computes the new address for the consolidated array. `size` indicates the bit length (e.g., 10) of the width (e.g., 1024).

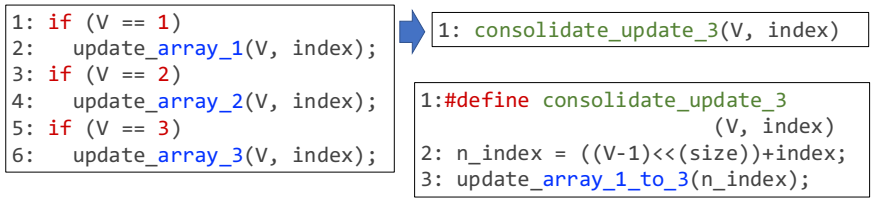


Figure 3.10: `consolidate_memory_update()`

`heavy_flowkey_storage(Key,List(Estimate),Threshold)` reduces the memory space for heavy flowkeys (O6). The challenge is checking whether the estimated flow count is above a threshold entirely in the data plane. Specifically, this entails computing the median value based on an estimated flow count from each row and comparing it to the threshold value. However, computing the median is not supported in the data plane. Instead, we leverage the fact that we can check whether the median of a set of values exceeds a threshold without computing the median as follows. We compare all of estimated flow count for all rows, as shown in lines 3-9 in Figure 3.11 which is for $R = 3$ case. Then, the condition (sum (s1, s2, s3) $\geq$ 2) at line 11 is equal to (median(est1, est2, est3) > T).[4] This can be generalized for different Rs. We implement the duplicate filter using a hash table and a exact-match table. If a flowkey collides with an entry in the hash table and the exact-match table does not have an entry for the flowkey, we report it to switch control plane via a PCIe channel. Upon receiving the reported key, the switch control plane CPU adds entries into the exact-match table.

3.4 Evaluation

In this section, we evaluate the benefits of SketchLib on seven sketches. Across a range of settings, we see that SketchLib can reduce the resource footprint of sketches on switch hardware (up to 96%) while achieving similar accuracy.

[4]For Count-Min sketch [37], we can use (sum (s1, s2, s3) $\geq$ 1).

```
01:#define heavy_flowkey_storage_3
                  (Key, Est1, Est2, Est3, T)
02:
03: s1, s2, s3 = 0;
04: if (Est1 > T)
05:   s1 = 1;
06: if (Est2 > T)
07:   s2 = 1;
08: if (Est3 > T)
09:   s3 = 1;
10:
// above threshold test
11: if (s1 + s2 + s3 >= 2) {
12:   if (HT[h(Key)] == empty) { // HashTable
13:     HT[h(Key)] = Key;
14:     send_to_cpu(Key);
15:   } else if(HT[h(Key)] != Key) {
16:     if (!(flowkey in MT)) { // MatchTable
17:       send_to_cpu(Key);
18:     }
19:   }
20: }
```

Figure 3.11: `heavy_flowkey_storage()`

3.4.1 Experimental Setup

Sketches. We implement all 15 sketches in Table 3.2 using SketchLib and source codes for sketches are available at [17]. Among 15 sketches, we pick seven representative sketches for our evaluation as in Table 3.5.

Testbed. We evaluate SketchLib 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 of SketchLib with Tofino SDE version of 9.1.1 in our experiments.

Traces. We use five CAIDA backbone traces capture at 3/20/14 to 6/19/14 Sanjose, 1/21/16 Chicago, 5/17/18 to 8/16/18 New York City [1]. We split one hour traces into 30 second epochs. Each epoch includes about 12M-23M packets, with 398K distinct source IPs, 280K distinct destination IPs, and 1.6M distinct 5 tuples.

Sketch parameters. Table 3.5 shows the configuration parameters for the sketches. Most sketches use 4 byte counters. The cardinality estimators (e.g., MRB and PCSA) use bitmap thus each counter is 1 bit.

	Level (L)	Row (R)	Width (W)	Space
CS [32]	-	5	4096	80KB
HLL [47]	-	-	2048	8KB
UnivMon [71]	16	5	2048	640KB
R-HHH [22]	25	3	2048	600KB
MRAC [66]	12	-	2048	96KB
MRB [43]	16	-	4096	8KB
PCSA [46]	32	-	20	0.125KB

Table 3.5: Sketch parameters for evaluation.

Metrics. Depending on the sketch and the measurement task, we report two error metrics. For each metric, we run the experiment 5 times independently with different hash parameters and report the 25%, 50%, 75% percentiles of the errors. For brevity, we report results using source IP as the flowkey except for R-HHH, noting that the results are qualitatively similar for other types of flowkeys. R-HHH uses (source IP, destination IP) pair as flowkey as presented in the original paper [22].

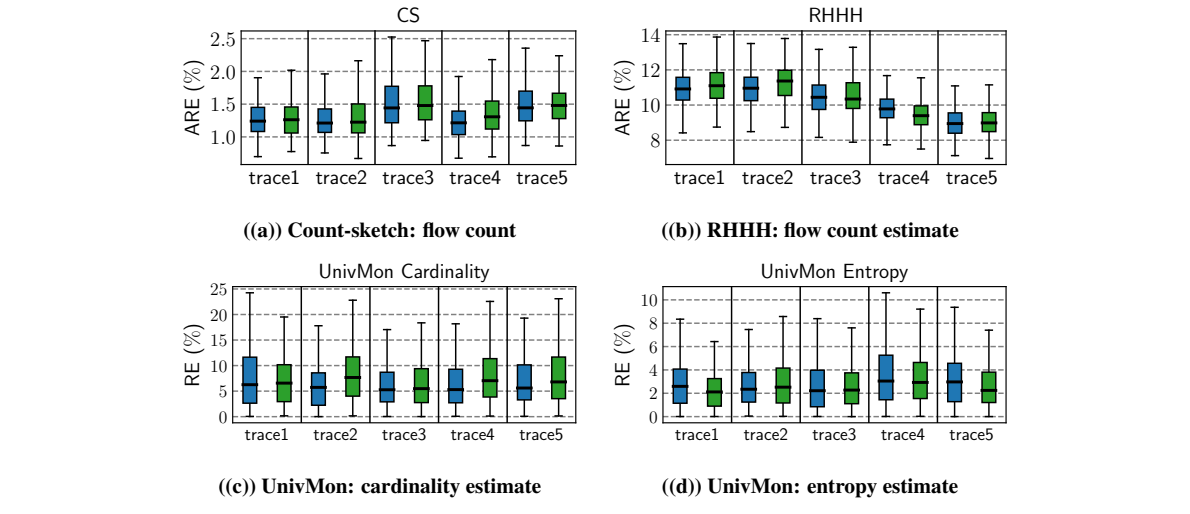
- Average Relative Error (ARE): $\frac{1}{k}\sum_{i=1}^{k}\frac{|f_i-\hat{f}_i|}{f_i}$, where k means the top k heavy flows. f_i is actual flow count for flow i and $\hat{f}_i$ is the estimated flow count from the sketch. $f_i \geq f_{i+1}$ for any i, thus it is sorted in descending order. We use k=50 for count sketch and R-HHH.
- Relative Error (RE): $\frac{|True-Estimate|}{True}$, where $True$ is ground truth value and $Estimate$ is estimated value. We use this metric for sketches that estimate cardinality and/or entropy.

3.4.2 Accuracy

We run the accuracy experiment of SketchLib in two ways. First, we show the accuracy is preserved between baseline software implementation and hardware implementation with SketchLib (§3.4.2). Second, we compare the accuracy of the hardware implementations with and without SketchLib (§3.4.2).

Comparison with the Software Baseline

Reporting methodology. We compare the accuracy of the sketch refactored with SketchLib (on hardware) against a baseline software implementation. The baseline software implementation runs sketches on the software. We run experiments over multiple traces with independent runs. After optimizing sketches with SketchLib, we run experiments on Tofino hardware with all five traces.

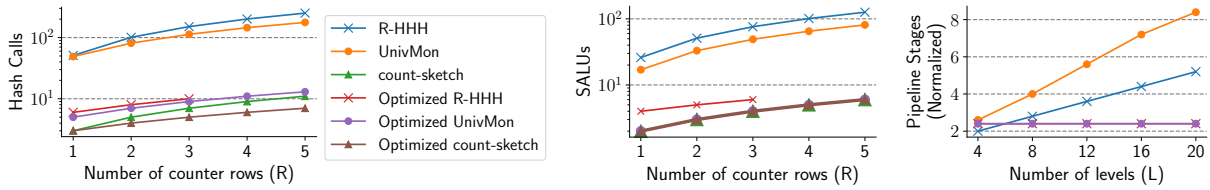


((a)) Count-sketch: flow count **((b)) RHHH: flow count estimate**

((c)) UnivMon: cardinality estimate **((d)) UnivMon: entropy estimate**

((e)) HLL **((f)) MRB** **((g)) PCSA**

Figure 3.12: Accuracy comparison of sketches between original and optimized sketches across traces. Left: original, Right: optimized.



((a)) Hash Calls **((b)) SALUs** **((c)) Pipeline Stages**

Figure 3.13: Resource consumption before/after optimizations.

For each one-hour trace, we randomly sample 40 30-second epochs and obtain 5 accuracy numbers per epoch with independent trials. The server replays traces to the switch using tcpreplay at a speed of 800K packets/second. Between epochs, we wait for switch control plane to pull counters and flowkeys from the data plane (see §3.5).

Result. Figure 3.12 empirically validates that SketchLib optimizations achieve similar accuracy. For every trace, the left blue bar represents the software baseline and the right green bar is the hardware reported result with SketchLib applied.[5] Figure 3.12(a) - Figure 3.12(d) shows the accuracy of sketches that need to track heavy flowkeys and the rest show sketches that need to maintain

[5]We do not show MRAC as the estimation logic for MRAC is not public.

UnivMon	Without SketchLib			With SketchLib
Level (L)	8	6	5	16
Row (R)	4	5	6	5
Width (W)	32768	32768	32768	2048
RE (%)	95.4%	98.8%	99.4%	9.5%

Table 3.6: Relative error in cardinality estimation with and without SketchLib.

only counter arrays. Figure 3.12(c) and Figure 3.12(d) show the errors of UnivMon for cardinality estimation and entropy estimation. We can visually confirm that the distributions of accuracy before and after optimizations are similar.

Accuracy Improvement with SketchLib

Reporting methodology. We want to compare the best accuracy between with and without SketchLib on the hardware. We use UnivMon for this experiment. To systematically sweep configuration parameters for the best accuracy without SketchLib, we exploit the property of UnivMon. Among three sketch parameters level (L), row (R), and width (W), L is the most critical parameter, thus we pick three highest feasible L. Then we find maximum R and lastly W. Given fixed L, we explored different parameter R other than maximum R but result was similar. We use the simulator with 40 samples of trace1. With SketchLib, we use the same configuration from the original UnivMon paper.

Result. Table 3.6 shows that all feasible configurations without SketchLib show high error rate more than 95%. On the other hand, UnivMon with SketchLib shows low error rate of 9.5%.

3.4.3 Switch Resource Consumption

Next, we report the resource usage improvements on the identified resource bottlenecks (Table 3.7). The sketch parameters used are reported in Table 3.5.

Reporting methodology. We measure resource usages from original implementation using RMT resource mapper and optimized implementation using Tofino compiler to measure resource reductions. To factor out resource reductions for different optimizations in Table 3.7, we wrote P4 code with individual optimizations applied using SketchLib APIs to measure the resource usages.

Sketches	Hash Calls (O1/O2)	SALUs	Pipeline Stages
CS	31%/0		9%
HLL	80%/0		86%
UnivMon	44%/47%	90%	65%
RHHH	32%/60%	92%	62%
MRAC	87%/0	91%	68%
MRB	90%/0	93%	76%
PCSA	92%/0	96%	86%

Table 3.7: Individual resource reductions by optimizations.

Hash calls. Table 3.7 shows that using O1 to consolidate the 1-bit hash calls is effective for both single and multi-level sketches. For example, the number of hash calls for count sketch is reduced by 31%. R-HHH and UnivMon benefit from O1 as they are composed of multiple count sketches. Further, PCSA, MRAC, MRB and HLL have a series of 1-bit hash calls which O1 improves. For UnivMon and R-HHH, we can apply both O1 and O2 by reusing hash calls across levels to further reduce hash calls by over 90%. We further investigate the sensitivity of the reduction of hash calls vs. sketch parameters in Figure 3.13(a).[6] Multi-level sketches UnivMon and R-HHH have significant reduction and the resource used is close to single-level count sketch.

Stateful ALUs. O5 applies only to multi-level sketches and reduces SALU usage significantly if there are many levels in the sketch. With 16-32 levels, O5 saves 92% to 96% of the SALUs. We can see in Figure 3.13(a) that O5 reduces SALU of UnivMon and R-HHH significantly across rows.

Counter Memory Space. Interestingly, O5, which we designed to reduce SALU usage, also reduces memory space. Investigating this further, we find that original sketch implementations have a memory region fragmentation problem. One counter array is smaller than a block of SRAM, causing additional (unused) memory overhead per each counter array. O5 has the added benefit of consolidating counter arrays and achieve 54%–96% of resource reduction in memory space for multi-level sketches (not shown).

Pipeline stages. The reduction of pipeline stages depends on a combination of factors — hash calls, SALUs, and code dependencies. Table 3.7 shows reduced pipeline stages from 9% to 86% across sketches. Sketches where O5 applies (HLL, UnivMon, MRAC, MRB, PCSA) have a large

[6] Missing point for R-HHH in Figure 3.13 means it is infeasible.

	FCM native	SketchLib-optimized		
Resource	FCM+topK	FCM(+O6)	CM	UnivMon
Pipe. Stage	8	8	7	12
SRAM	9.5%	10.8%	8.0%	7.3%
TCAM	0%	0%	0%	0.3%
SALUs	20.8%	14.6%	14.6%	12.5%
Hash Calls	13.9%	9.7%	11.1%	18.1%
Hash Bits	5.6%	4.0%	4.0%	4.9%

Table 3.8: Comparison of hardware resource utilization.

		SketchLib-optimized		
	FCM+topK	FCM(+O6)	CM	UnivMon
HH (ARE)	1.41%	0.01%	0.13%	0.73%

Table 3.9: ARE of heavy hitter detection.

reduction because it removes many sequential `if` clauses. Effectively, our optimization can make the footprint of multi-level sketches agnostic to number of levels (Figure 3.13(c)).

3.4.4 Comparison with FCM

FCM [89] is a recently published sketch with general capability, and it is feasible on the programmable switch. Thus, we compare FCM against sketches optimized with SketchLib in terms of resource usages and accuracy. Table 3.8 shows resource utilization comparison between FCM and SketchLib optimized sketches. We use the same configuration from public FCM code [4], and make SketchLib-optimized sketch use similar resources to FCM.

Heavy hitter detection. Table 3.9 shows the accuracy result of heavy hitter detection. We can see that FCM+topK suffers from a high error rate because of an inefficient mechanism for tracking heavy flowkey (approximate topK implementation of ElasticSketch [100]). Note that if FCM deploys one of our optimizations for tracking heavy flowkeys, FCM+O6 reduces the error rate significantly from 1.41% to 0.01%. We use the simulator with 40 samples of trace1 and report median ARE.

Entropy and cardinality. Table 3.10 and Table 3.11 compare entropy and cardinality estimation accuracy between FCM+topK and SketchLib-optimized UnivMon. In the experiments, UnivMon reports top-200 heavy hitters per level. For entropy, UnivMon shows a relatively stable error rate (2~3%) across workloads, whereas FCM is dependent on workloads and the error rate can

# of flows	500K	1M	5M	10M	30M
FCM+topK	0.35%	0.84%	3.60%	6.15%	17.0%
SketchLib UnivMon	2.59%	2.08%	2.21%	2.36%	2.96%

Table 3.10: Entropy error (RE), FCM vs. SketchLib-optimized UnivMon.

# of flows	500K	1M	5M	10M	30M
FCM+topK	0.004%	0.107%	0.519%	100%	100%
SketchLib UnivMon	21.9%	20.7%	31.7%	39.5%	73.8%

Table 3.11: Cardinality error (RE), FCM vs. SketchLib-optimized UnivMon.

go up to 17%. For cardinality, the error rate of UnivMon is moderately increasing [7], whereas FCM suddenly becomes unusable after 5M flows. This is because Linear Counting [96] is used to estimate cardinality in FCM.

3.4.5 Tracking Heavy Flowkeys

To evaluate the impact of O6, we consider three metrics: miss rate, control plane bandwidth, and data plane memory. We compare SketchLib-optimized approach vs. an "optimal" software solution. For this evaluation, we use two sketches (CS, UM) that track "heavy" flowkeys. For each 1-hr trace, we split it into epochs as before, and set a target threshold corresponding to the top 0.2 percentile of flow sizes (The results are independent of the threshold; this is to make the experiment concrete). Across different traces and sketches, SketchLib incurs zero miss rate, and at most 2% increase in control plane bandwidth (due to small number of duplicates), using less than 400KB of data plane memory overall (independent of the threshold, results not shown for brevity). To put this in context, a Bloom-filter based strawman for suppressing duplicates, as discussed in §3.1 and configured with the same memory use has a miss rate of 0.2%. Overall, this confirms that SketchLib offers a more practical alternative to the infeasible, inaccurate, and/or expensive strawman solutions from §3.1.

[7] We observe that, when UnivMon reports more heavy hitters per level, the cardinality error rate decreases (e.g., 17.58% in 10M flows with top-1000).

Resource	With SketchLib	
	UnivMon	UnivMon + NFs (L2, L3, LB, FW)
Pipe. Stage	12	12
SRAM	7.3%	38.6%
TCAM	0.3%	25.0%
SALUs	12.5%	12.5%
Hash Calls	18.1%	18.1%
Hash Bits	4.9%	11.2%

Table 3.12: Sketches are infeasible without SketchLib. With SketchLib, there are rooms for additional network functions (L2/L3 forwarding, L4 load balancer, and stateful firewall).

Sketch	CS	HLL	UM	RHHH	MRAC	MRB	PCSA
Before	201	290	460	471	261	317	305
After	131	112	127	128	91	94	93

Table 3.13: Lines of code simplification (UM stands for UnivMon).

3.4.6 Other Benchmarks

Additional Network Functions. After optimized with SketchLib, sketches can even coexist with additional network functions such as L2/L3 forwarding, L4 load balancer, and stateful firewall. Table 3.12 shows resource utilization for additional network functions.

Code simplification. In addition to the resource efficiency benefits, our optimizations also simplify the sketch implementations by reducing the lines of code, as shown in Table 6.7.

Compilation time. We also measured compilation time to see whether our modified code will add significant overhead to the compiler. Compile time is measured on the server specified in (§3.4.1). For most cases, there was a negligible (≤ 1 second) increase (not shown).

3.5 Related Work

Programmable switches. The programmable switch architecture was introduced by Bosshart and others [28]. Subsequent work proposed a programming framework [29], functional hardware [9], and also compilation workflows [60]. Other vendors have developed programmable pipelined architectures and compilation workflows from P4 or P4-like primitives [3, 6]. While our focus is on Tofino, our approach could be useful for other platforms as well.

Optimizing sketches. HashPipe [88] focused on heavy hitter detection, but it is not feasible in the current hardware. Other work has focused on the optimizing sketching algorithms in software

switches (e.g., [57, 73, 97]). However, some of their ideas do not translate into a hardware context. For instance, NitroSketch increases the memory footprint to reduce CPU consumption, but the key bottleneck in hardware is different. Similarly, other approaches split a sketch into a fast and slow path on the software switch (e.g., [57]). Unfortunately, this is not relevant in hardware since we need all operations to be in the fast path. Some recent work [97, 99] specifically focus on optimizing UnivMon for embedded platforms and software switches. We translate these insights to a switch hardware realization, and generalize beyond UnivMon.

Other work in network telemetry. Our focus in this paper is on sketch-based telemetry. There are other efforts for complementary monitoring capabilities (e.g., [52, 55, 91]) and performance-oriented objectives (e.g., [51, 83]).

3.6 Summary

Unfortunately, existing sketch implementations are not efficiently realizable, thereby limiting their effectiveness and coexistence with other switch functions. To this end, we systematically analyze the resource bottlenecks, suggest correct-by-construction optimizations, and design a practical library to help developers use these optimizations. Our evaluations show that the SketchLib library is broadly applicable to many sketches and reduces their resource footprint while also achieving similar accuracy.

Chapter 4

Sketchovsky: Optimizing Ensembles of Sketch Instances on Programmable Switches

Network operators need to concurrently run diverse measurement tasks for network management tasks, such as traffic engineering, anomaly detection, load balancing, and resource provisioning [19, 26, 50, 79, 101]. A given 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).

Sketching algorithms or *sketches* appear to be a promising avenue to support measurement tasks on data collected from programmable switches (e.g., [17, 18, 20, 37, 47, 71, 88, 106]). 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.

However, existing sketch-based telemetry efforts largely focus on running a *single* sketch instance on programmable switches; 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

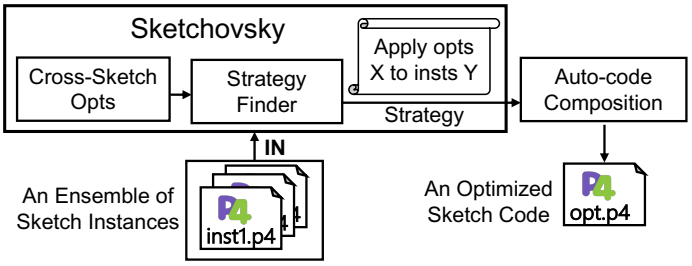


Figure 4.1: Sketchovsky. Opts is optimizations and insts is instances.

some recent attempts at improving per-sketch efficiency (e.g., [17]), supporting P4 code composition [48, 49, 70, 90, 109], elastically trading of resource vs. accuracy (e.g., [16, 80]), and improving the generality of sketches (e.g., [22, 34, 71, 106]), we find that these fundamentally fall short of efficiently supporting a general ensemble of sketch instances without sacrificing accuracy. (We elaborate on this in §4.1).

Given the limitations of existing approaches for the sketch ensemble, 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. 4.1). Sketchovsky is complementary to the above efforts to build more efficient single-sketch algorithms or developing more general-purpose sketches or explicitly trading off resource reduction for accuracy reduction. Indeed, using Sketchovsky can amplify their benefits to include more expressive sketches (e.g., [71]) or extend per-sketch optimizations (e.g., [17]).

The design of Sketchovsky makes three key contributions:

Optimization building blocks (§4.3): 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

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

Query 4.2: Heavy hitters detection of srcIP written in Sonata [52]

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 (§4.4): 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 challenges in modeling resource usage and in identifying conflicts for combining optimizations, as well as because of 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., less than 1 second) an effective strategy that yields significant benefit across various problem instances.

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 [27, 32, 37, 43, 46, 47, 65, 66, 67, 71, 96]. Given this basic library, we consider an ensemble generator that uses a large pool of candidate parameters to consider various types of ensembles. We use a combination of benchmarks and trace-driven results [1] to measure accuracy results for running 36 sketch instances grouped by four ensembles. Compared to the baseline of SketchLib, the resource reduction benefit is at most for ensembles that have the same flowkey for all sketch instances by reducing 7~40% of the hash calls, 9~45% of SALUs, and 0~7% of SRAM. The benefit is at worst for ensembles that sketching algorithms and parameters are picked randomly by reducing 3~21% of the hash calls, 4~26% of SALUs, and 0~0.4% of SRAM.

4.1 Motivation

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

Query 4.3: Distinct number of 5-tuple flows written in Sonata [52]

4.1.1 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 [34, 52, 76, 83, 102, 105, 106, 110]. As concrete examples of measurement tasks, we show two network queries written by Sonata [52], a state-of-the-art query language on programmable switches. Query. 4.2 can detect heavy hitters based on the sum of packet length in bytes aggregated on flowkey of srcIP. Query. 4.3 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: (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 because there is a trade-off between resource usage and accuracy.

For instance, given Query. 4.2 above, we can use a count-min sketch instance on the srcIP as flowkey and for Query. 4.3 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 (§4.7).

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 4.1: 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 [48, 49, 70, 90, 109]	✓	X	✓
Per-sketch optimizations [17]	✓	X	✓
More expressive sketches [22, 34, 71, 106]	X	✓	✓
Dynamic resource allocation at compile time [16, 80]	✓	✓	X
Dynamic resource allocation at run time [108]	X	X	✓
Sketchovsky (Our system)	✓	✓	✓

Table 4.2: Existing efforts cannot support a general ensemble of measurement tasks with low resource footprint and high accuracy

4.1.2 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.

Composing P4 programs. Many P4 code composition works have been recently published for resource optimizations [48, 49, 70, 90, 109]. 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 [109], Lyra [48], and Cetus [70] focus on optimizations for match-action tables, but they did not consider optimizations for stateful processing, including MicroP4 [90]. Thus, they cannot be used to optimize an ensemble of sketch instances.

Chipmunk [49] seems to be a promising candidate for providing cross-sketch optimizations at first glance, since it compiles a program written by Domino language into optimized P4 code with stateful processing optimizations. However, Chipmunk cannot 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 index of counter arrays), which is critical for sketch implementations.

Per-sketch optimizations. Per-sketch optimizations [17] can be used to implement the sketch ensemble. However, this approach cannot achieve low resource footprints because *linearly* increasing resource consumption is required 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 [22, 34, 71, 106]. However, their coverage of the measurement tasks is still far from general (e.g., none of them can support two entropy estimation tasks for two different flowkey definitions).

Dynamic resource allocation. SCREAM [80] dynamically reduces resource parameters of sketch instances to meet specified minimum accuracy when there are variations in traffic. However, SCREAM will not work when there is low traffic variance. P4All [16] can elastically reduce resource allocation of P4 applications with the table-like data structure. It can be used to reduce resource parameters of some sketch instances in the ensemble by identifying lower-prioritized sketch instances. However, P4All cannot handle cases where every sketch instance is equally important. Although these approaches of reducing resource parameters at compile time can achieve low resource footprints, it also degrades accuracy. FlyMon [108] 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. This approach is orthogonal to running as many sketch instances *concurrently* as possible.

Quantitative results for existing efforts. We quantitatively show why existing efforts are insufficient. As a representative for per-sketch optimization (including expressive sketches), we use SketchLib. As an exemplar for dynamic/elastic resource sharing, we use P4All. Fig. 4.4 shows the feasibility-accuracy trade-off between per-sketch optimization (SketchLib) and dynamic resource allocation (P4All). To create ensembles, we use sketch instances of the count-min sketch with $(R, W) = (5, 8K)$, flowkey of 4-tuple, different measurement epochs, and flowsize definitions. Then we measure resource footprint and accuracy results using CAIDA traces [1]. We can see that

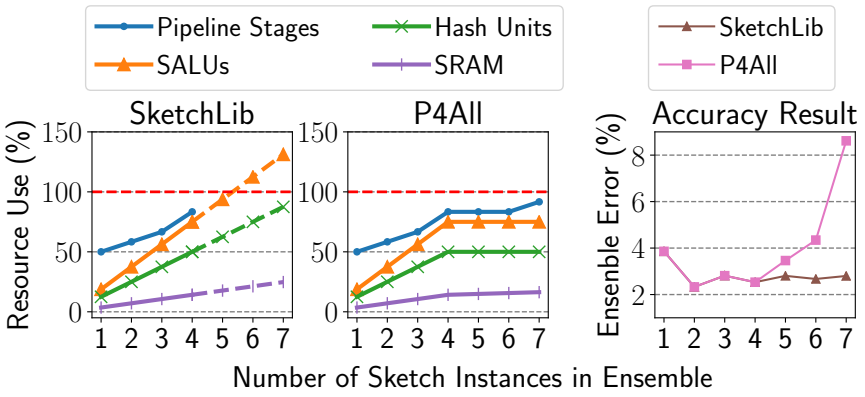


Figure 4.4: Existing efforts cannot efficiently run the ensemble

SketchLib cannot support more than four sketch instances. While P4All can support ≥ four sketch instances by reducing hardware resources, it can only do so by reducing accuracy. [1] In summary, we find that while existing techniques are valuable, they fall short of our goal.

4.2 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 $\mathcal{S} = \{s_i\}_{i=1}^{N}$. We envision that our framework can coexist and amplify these existing efforts, since the values on their own are insufficient.

To this end, we present Sketchovsky (Fig. 4.1), a novel cross-sketch optimization and composition framework. Sketchovsky identifies five cross-sketch optimization building blocks so that resource consumption increases *sub-linearly* with guarantees of no accuracy loss. Sketchovsky designs 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 (§4.3). 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. O_{Hash1} and O_{Hash2} optimize the first step of hash computations; O_{Ctr1} and O_{Ctr2} optimize the second

[1] P4All [7, 16] does not consider elastically reducing rows of counter arrays, which is directly related to feasibility due to a critical hardware resource SALU. Thus, we use an objective function that treats all sketch instances equally given a constraint of maximum SALU to emulate P4All.

step of counter updates, and O_{Key} optimizes the third step of heavy flowkey storage. Note that optimizations can be generalized to other hardware (§4.7). 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 (§4.1.1) (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 (§4.4). 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 of 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 4.1. Optimization building blocks can be applied to a subset $\mathbf{S} \subset \mathcal{S}$ only if $\mathbf{S}$ satisfies the applicable conditions.

4.3 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 4.3 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 4.4 summarizes applicable conditions to validate whether each optimization can be applied to $\mathbf{S} \subset \mathcal{S}$.

Workflow Step	Optimizations	Reduction	Overhead
Hash Computations	*Hash-Reuse* (O_{Hash1})	Hash call	-
	Hash-XOR (O_{Hash2})	Hash call	Pipe Stages
Counter Updates	*SALU-Reuse* (O_{Ctr1})	SALU,SRAM	-
	SALU-Merge (O_{Ctr2})	SALU	SRAM
Heavy Flowkey Storage	*HFS-Reuse* (O_{Key})	SALU	Pipe Stages CP Comp

Table 4.3: 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 4.4: Applicable conditions for five optimization building blocks

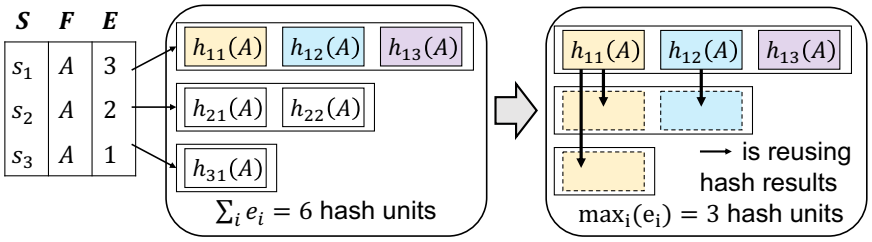


Figure 4.5: *Hash-Reuse* (O_{Hash1}) reduces hash calls by reusing hash results. A small box with $h_{seed}(flowkey)$ indicates one hash call allocation.

4.3.1 Hash Computations

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

***Hash-Reuse* (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 calls. We explain this optimization using an example in Fig. 4.5. 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

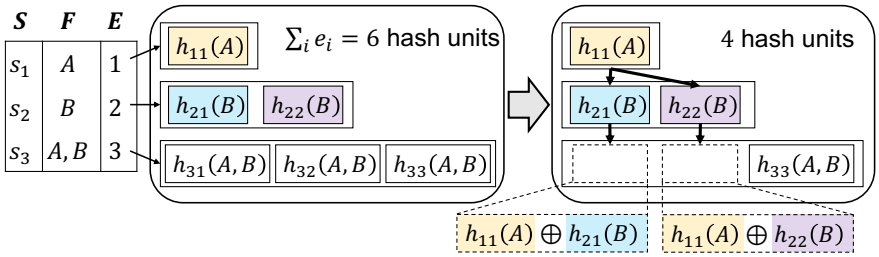


Figure 4.6: *Hash-XOR* (O_{Hash2}) reduces hash calls by using XOR

optimization, $\sum_i e_i$ hash calls are used. However, we can reuse hash results, and we can reduce the allocation of hash calls to $\max_i(e_i)$ on the hardware as in Fig. 4.5.

Applicability: Regardless of any sketching algorithms, we can apply this optimization as long as **S** uses the same flowkeys. We denote this as (C4) in Table 4.4.

Implication: Allocation of $\max_i(e_i)$ hash calls 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. 4.5).

***Hash-XOR* (O_{Hash2}) Less hashing, same performance with XOR-based reconstruction.** We can reduce hash calls even for a set of sketch instances with different flowkeys by leveraging XOR operations. We explain this optimization using an example in Fig. 4.6 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 calls by reconstructing independent hash results for s_3 as follows because $\{A, B\} = \{A\} \cup \{B\}$.

$$h_{31}(A, B) = h_{11}(A) \oplus h_{21}(B) \tag{4.1}$$

$$h_{32}(A, B) = h_{11}(A) \oplus h_{22}(B) \tag{4.2}$$

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 *Hash-XOR* (O_{Hash2}) can be applied if $\mathbf{S}$ and $\mathbf{F}$ meet the following condition.

$$\text{For } \mathbf{S} \in \mathcal{S}, |\mathbf{S}| = 3 \text{ and } f_1 \cup f_2 = f_3 \tag{4.3}$$

This optimization can be applied as long as a set of sketch instances satisfies (4.3). Thus, we mark (*) at (C4) in Table 4.4 for O_{Hash2}.

Implications: This idea of XOR-based hash reconstruction is proven pairwise independent and has already been used in other contexts [64, 93]. Thus, accuracy will not be compromised, which is confirmed by our evaluation result. 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.

4.3.2 Counter Update

To optimize the second workflow step of *counter updates*, we have *SALU-Reuse* (O_{Ctr1}) and *SALU-Merge* (O_{Ctr2}).

SALU-Reuse **(O_{Ctr1}) Reusing counter arrays (rows) across sketch instances.** If all sketch instances in $\mathbf{S}$ 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. 4.7, 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

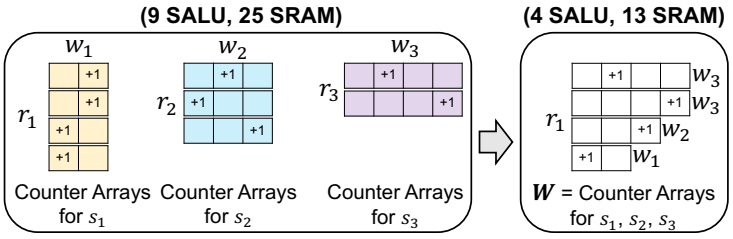


Figure 4.7: *SALU-Reuse* (O_{Ctr1}) reuses counter arrays

compute the row and width of counter arrays for reuse is represented by **W**:

$$\mathbf{W} = \{w_j^*\}_{j=1}^{\max_i(r_i)} \text{ where } w_j^* = \max_i\{w_i | r_i \geq j\} \tag{4.4}$$

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.2.1. 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 §B.1.1 for a full list of five counter update types. $\mathbf{S} \subset \mathcal{S}$ must satisfy five conditions to apply this optimization (Table 4.4): 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).

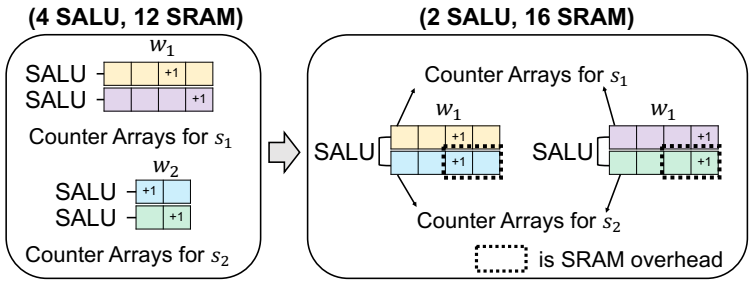


Figure 4.8: *SALU-Merge* (O_{Ctr2}) reduces SALUs by making SALUs update two counter arrays simultaneously

***SALU-Merge* (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. 4.8. 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 the flowkeys are the same, we can leverage *Hash-Reuse* (O_{Hash1}), and SALU can update two counter arrays using the reused hash result for the column index. However, if the 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. 4.8 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. 4.8. 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 SRAM overhead is bounded by 2x. Third, SRAM is not the imminent bottleneck as we will see in the evaluation.

4.3.3 Heavy Flowkey Storage

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

HFS-Reuse **(O_{Key}). Reusing heavy flowkey storage across sketch instances.** We can reduce SALU usage 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 4.4. 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.

4.4 Strategy Finder

In the previous section, we proposed five optimizations $\{O_j\}_{j\in\{Hash1,Hash2,Ctr1,Ctr2,Key\}}$ 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 $\mathcal{S}$.

4.4.1 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 $\mathcal{P}_\mathcal{S} = \{P_k | P_k$ is kth partition of the set $\mathcal{S}\}$ is a set containing all partitions of the set $\mathcal{S}$ where $P_k = \{\mathbf{S}_l \subset \mathcal{S} | \biguplus_l \mathbf{S}_l = \mathcal{S}\}$. The goal is to find the optimal strategy X^*, which minimizes hardware resources while satisfying

the applicable conditions:

$$\min_{X} \; HW\_Resource(X) \tag{4.5}$$

$$\text{s.t.} \sum_{k=1}^{|\mathcal{P}_\mathcal{S}|} x_{jk} = 1, \forall j \in \{Hash1, Hash2, Ctr1, Ctr2, Key\} \tag{4.6}$$

$$Applicable(X) = 1 \tag{4.7}$$

The decision variable is $X = \{X_j\}_{j \in \{Hash1, Hash2, Ctr1, Ctr2, Key\}}$. X_j selects a partition $P_k \in \mathcal{P}_\mathcal{S}$ for O_j so that O_j is applied to all subsets $\in P_k$. To express this, we define $X_j = \{x_{jk} | x_{jk} \in \{0,1\}\}_{k \in \{1,\ldots,|\mathcal{P}_\mathcal{S}|\}}$ and $x_{jk} = 1$ if P_k is selected. Note that (4.6) makes X_j pick only one partition P_k for O_j.

About constraint (4.7), we use $Applicable(X) \in \{0,1\}$ 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 (4.5), we aim to find a strategy X^* that minimizes hardware resource $HW\_Resource(X)$ among all valid strategies X. To model this objective function, we use the linear combination of four key resource usage:

$$LinearComb(X, R) = \sum_{r \in R} a_r \cdot resource_r(X) \tag{4.8}$$

$$R = \{\text{SALU, HashUnit, SRAM, PipelineStage}\} \tag{4.9}$$

Network operators can use (4.8) 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 the objective function.

4.4.2 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 [21].

Even after we consider constraint (4.6), 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.

C-2. Modeling the applicable function. It is hard to define $Applicable(X)$ due to the dependencies among optimizations. Specifically, there exist dependencies between O_{w1} and O_{w2} for $w \in \{Hash, Ctr\}$ 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 $Applicable(X)$.

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 $\textit{resource}_{pipeline\_stage}(X)$ can take several minutes. This delay will significantly impede the search process, and finding a solution X^* can become even more intractable.

4.4.3 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 tackle (C-3), we make a pragmatic choice of excluding the pipeline stage from the objective function. Specifically, we use $LinearComb(X, R')$ as objective function where R' = {SALU, HashUnit, SRAM}.

$LinearComb(X, R')$ can be easily realized because $\textit{resource}_r(X)$ for $r \in R'$ can be quickly computed because X contains information about the number of reused or XOR-reconstructed hash calls, reused or co-located counter arrays, and reused heavy flowkey storage. While this loses generality in the objective, we argue this is practical because there is a correlation between the resource reduction of R' and the pipeline stage reduction.

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 applicable function and the objective function as follows:

$$X = \cup_{w \in \{Hash,Ctr,KEY\}} X_w \tag{4.10}$$

$$Applicable(X) = \prod_{w \in \{Hash,Ctr,KEY\}} Applicable(X_w) \tag{4.11}$$

$$HW\_Resource(X) = \sum_{w \in \{Hash,Ctr,KEY\}} HW\_Resource(X_w) \tag{4.12}$$

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, $Applicable(X)$ can be achieved by multiplication of decomposed $Applicable(X_w)$ as in (4.11). Second, $HW\_Resource(X)$ can be achieved by summation of decomposed $HW\_Resource(X_w)$ as in (4.12). Without the idea of excluding pipeline stage usage, this linearity property (4.12) 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 $w \in \{Hash, Ctr, KEY\}$ as follows:

$$\min_{X_w} \; HW\_Resource(X_w) \text{ s.t. } Applicable(X_w) = 1 \tag{4.13}$$

Two-step enumeration for X_{Hash} and X_{Ctr}. Although we can decompose $Applicable(X)$ into three $Applicable(X_w)$ as in (4.11), it is still unclear how to realize $Applicable(X_w)$ for $w \in \{Hash, Ctr\}$ 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. $Applicable(X_w) = 1$) while efficiently skips *invalid* X_w. In that case, it will help to solve not only the challenge (C-2) of modeling an applicable

Algorithm 1 TwoStepEnumeration

```
1:  procedure TwoStepEnumeration(S, w)
2:      P_S = {P_k | P_k is kth partition of the set S}
3:      min ← INT_MAX
4:      for X_w1 s.t. selected P_w1 ∈ P_S is valid do
5:          for X_w2 s.t. P_w1 ≤ P_w2 ∈ P_S do
6:              X_w ← {X_w1, X_w2}
7:              P_w12 = NestedPartition(P_w1, P_w2)
8:              if P_w12 is valid then
9:                  if min > HW_Resource(X_w) then
10:                     min ← HW_Resource(X_w)
11:                     X*_w ← X_w
12:     return X*_w
```

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.

We explain this algorithm by both cases of $w \in \{Hash, Ctr\}$. Let's first see an example for $w = Hash$, *Hash-Reuse* (O_{Hash1}) and *Hash-XOR* (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 calls. Next, given picked P_{w1}, it enumerates P_{w2} s.t. $P_{w1} \leq P_{w2}$[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 $\{\text{srcIP}\} \cup \{\text{dstIP}\} = \{\text{srcIP, dstIP}\}$ as in (4.3) at §4.3.1. 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 $w = Ctr$, *SALU-Reuse* (O_{Ctr1}) and *SALU-Merge* (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 (4.3) at §4.3.2. Then each subset can be handled as if it is a single sketch instance with counter arrays configured with $\mathbf{W}$.

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

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.

HFS-Reuse (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 $P_{w1} \leq P_{w2}$, 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 picked 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 (§4.6.4).

4.5 Implementation

4.5.1 Strategy Finder

One minor issue here is that while implementing *SALU-Merge* (O_{Ctr2}) on the Tofino switch, we face the known problem of sketch inaccuracy, which is caused by the counter read and reset delays [82]. To address this, we add one more applicable condition of the same epoch (C6) to O_{Ctr2}.

4.6 Evaluation

We performed an extensive set of experiments that demonstrate that Sketchovsky can achieve low resource footprint and high accuracy at the same time. This evaluation is based on the auto-code composition framework, which will be introduced in the next chapter (chapter 5).

4.6.1 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 practically. 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 §B.3.1. The ensemble generator does not allow any two sketch instances in an ensemble to have the same statistic, flowkey, flowsize, and epoch.

[3] Linear counting (LC) [96], HyperLogLog (HLL) [47], PCSA [46], multi-resolution bitmap (MRB) [43] measure cardinality. Count-sketch (CS) [32], count-min sketch (CM) [37] can detect heavy hitters, and K-ary sketch (KARY) [65] can detect heavy change. Entropy sketch (ENT) [67] measures entropy, MRAC [66] measures flow size distribution (FSD). UnivMon (UM) [71] can measure general statistics. Bloom filter (BF) [27] can do the membership test. A full list of sketching algorithms with sketch features is in §B.3.1.

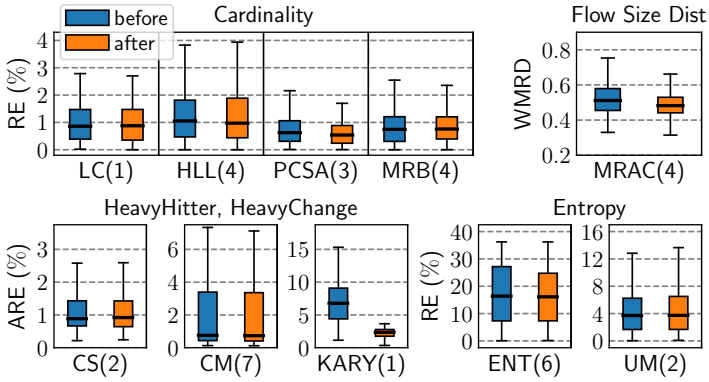


Figure 4.9: Overall accuracy evaluation

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 [66].

For resource reduction, we use two metrics: (1) Resource Usage (RU): $\frac{Used}{Available}$, 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): $\frac{RU(before)-RU(after)}{RU(before)}$, 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.

4.6.2 Accuracy

We show that Sketchovsky does not degrade, and actually somtimes 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 §B.3.2. 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

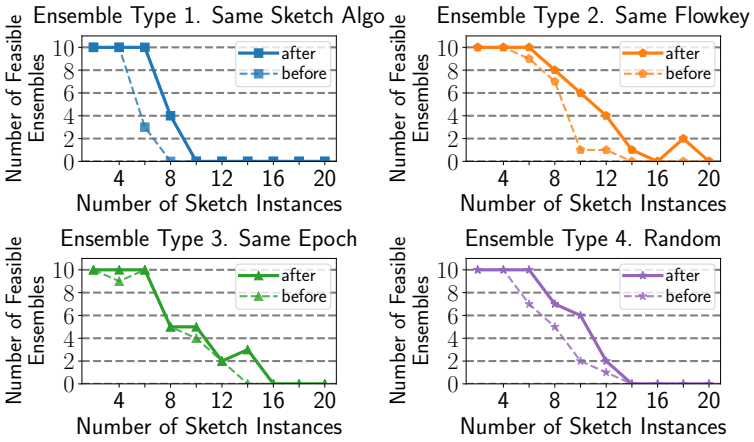


Figure 4.10: Feasibility comparison of ensembles before vs after

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.

Fig. 4.9 shows the overall accuracy results. We grouped sketch instances into four different statistics based on the sketching algorithm used. The X-axis in Fig. 4.9 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 significantly. O_{Ctr2} is applied to one of the PCSA, MRAC, and ENT sketch instances, and the error is also 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.

4.6.3 Resource Reduction

Sketchovsky makes infeasible ensembles feasible. We show that using Sketchovsky incurs several resource reduction benefits. For this experiment, we generate a total of 400 ensembles of

[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 [1]

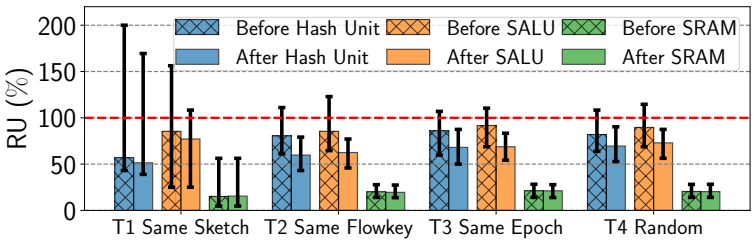


Figure 4.11: Resource usage comparison before vs after for the number of sketch instances = 12.

sketch instances; (four ensemble types) × (10 different numbers of sketch instances from 2, 4, ..., 20) × (10 different ensembles). 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. Then, we run Sketchovsky to produce 400 sketch P4 codes both before and after optimization. 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.

The X-axis in Fig. 4.10 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 (8 sketch instances), all ten ensembles were *in*feasible before optimization. However, 4 out of 10 ensembles become feasible after optimization. We can also see that the pipeline stage overhead that O_{Hash2} and O_{Key} can cause does not negatively impact feasibility after applying them.

Resource usage before and after optimization. Fig. 4.11 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. 4.11 shows the median value, and the error line shows the 10%

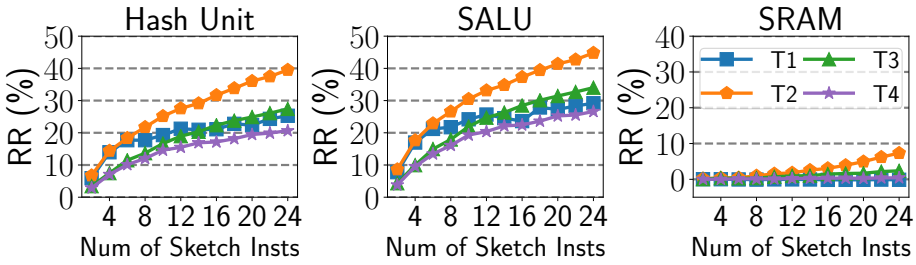


Figure 4.12: Resource reduction result

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. 4.11 visually shows how previously infeasible ensembles become feasible.

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. 4.12 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 call 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. 4.12 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 call is up to 20% and SALU is up to 26% because Sketchovsky offers five multiple building blocks for optimization.

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

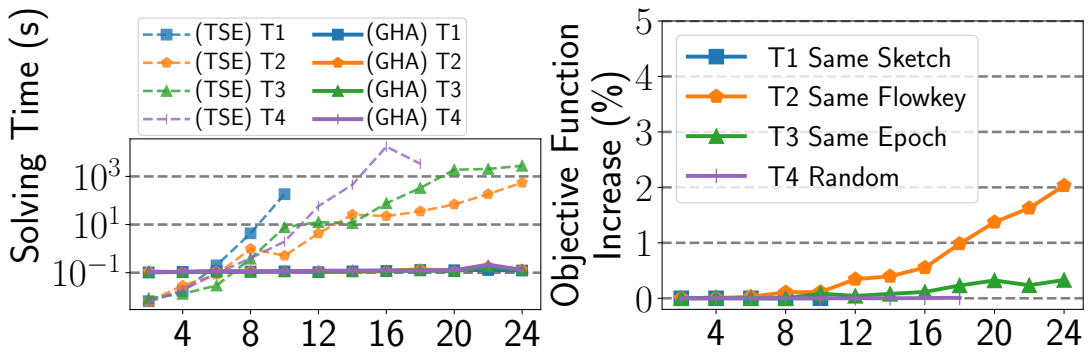
Breakdown on individual optimizations. We zoom into ensembles with 12 sketch instances and show the breakdown of resource reduction in Table 4.5. *HFS-Reuse* (O_{Key}) shows consistently high resource reduction for all four ensemble types (18% to 25% SALU reduction). Note that O_{Key} can also reduce hash calls. This is because of the specific hardware architecture of Tofino; one hash call must be allocated for one SALU (now we call this HashUnit-SALU coupling). *Hash-Reuse* (O_{Hash1}) is the next impactful optimization. For type 2, O_{Hash1} reduces hash calls by up to 10.4%. *SALU-Reuse* (O_{Ctr1}) reduces both SALU and SRAM and *SALU-Merge* (O_{Ctr2}) reduces SALUs but increases small SRAM overhead (negative values such as -0.5%). Finally, *Hash-XOR* (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 4.5.

4.6.4 Experiment for Greedy Heuristic Algorithm

In the strategy finder section (§4.4), we propose the use of 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.



((a)) Solving Time

((b)) Performance on Reduction

Figure 4.13: Two-step enumeration (TSE) vs greedy heuristic algorithm (GHA)

- Objective Function Increase: $\frac{HW\_Resource(X_G)}{HW\_Resource(X_T)}$ 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. 4.13(a) 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. 4.13(b)). For solving time, we measure time for 300 ensembles per data point in Fig. 4.13(a) and show the worst solving time. Data points that take more than 24 hours are not shown.

4.7 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 using Sketchovsky to explicitly consider the characteristic of input workload and the resource-accuracy trade-off in an ensemble setting could be an interesting direction for future work [75, 102].

Generalizing to other hardwares. While our prototype uses Tofino due to its open development API, we posit that our optimization building blocks and strategy algorithm can be generalized to other programmable switches and platforms.

4.8 Summary

In this paper, we tackled 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.

Chapter 5

Auto-code Composition Framework: Automatically Generates Optimized Sketch Data Plane Code

To simplify developer and operator effort, we design a simple-yet-effective switch-code generation process that realizes the selected strategy from Sketchovsky. 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 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. This auto-code composition framework can help build code for both a single sketch instance and ensembles of sketch instances.

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. Using solution X^* from the strategy finder in Sketchovsky, we need three steps to generate an optimized P4 code for $\mathcal{S}$ as in Fig. 5.1.

5.1 Step 1. Create Sketch P4 Codes

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 4.1).

Code template library. Writing N sketch P4 codes from scratch is a cumbersome task for network operators. An effective method is to provide code templates of the sketching algorithm

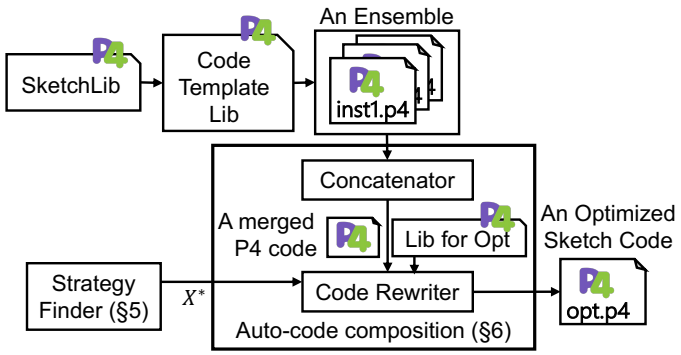


Figure 5.1: Overview of auto-code composition

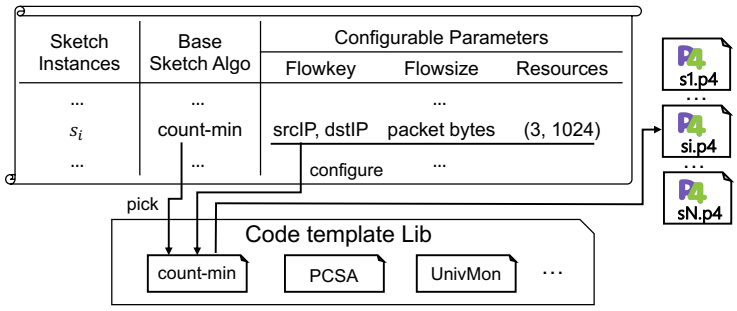


Figure 5.2: Code template library

```
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 5.3: Code template example for count-min sketch

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. 5.2 shows how to use the code template library. For each sketch instance s_i, it first picks a code template in the code template library using base sketching algorithm. Then using the configurable parameters, network operators can conveniently create sketch P4 code for s_i.

Next, we show examples of code templates. Fig. 5.3 and Fig. 5.4 are code templates for count-min sketch and PCSA. Code templates have placeholders with an underline for configurable

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

Figure 5.4: Code template example for PCSA

parameters. Network operators can fill out placeholders using configurable parameters. For example, they 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 update (e.g., line 3-8 in Fig. 5.3).

Code templates have five sections starting with the init section containing codes for setup such as header includes, and the parser. Next, there are three sections for three workflow steps; hash computation, counter update, and heavy flowkey storage. Lastly, there is the end section including codes for the the deparser. The init section and the end section are identical across all code templates. Thus, these sections are hidden in Fig. 5.3 and Fig. 5.4. Our focus is on three sections for three workflow steps.

SketchLib. To simplify the code template, we consider using a common library to write codes for sketch instances, such as SketchLib [17]. 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. 5.4 at line 5 gets any definition of flowkey and any counter update type (e.g., `PCSA` type is used in this example). We summarize extended API calls from SketchLib as in Table 5.1.

- `TCAM_LPM (hash_result)` uses TCAM for the longest prefix match to compute the left-most 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`).

Two Libs	API Name	API Parameters	Extended from SketchLib
SketchLib	`TCAM_LPM()`	hash_result	Same as `tcam_optimization()`
	`CounterUpdate()`	seed, flowkey, width, CA_type, CU_type, ...	Extends `consolidate_update()`
	`AboveThreshold()`	LIST(estimate), threshold	Extends `heavy_flowkey_storage()`
	`HFS()`	flowkey	Extends `heavy_flowkey_storage()`
Lib for Opt	`CounterUpdate_2()`	seed, flowkey, width, CA_type, CU_type1, CU_type2, ...	New API for *SALU-Merge* (O_{Ctr2})

Table 5.1: API calls extended from SketchLib and Lib for optimization

`seed` is used for the hash call 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[1]. 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.

5.2 Step 2. Code Concatenation

For the next step, the concatenator in Fig. 5.1 gets the input of N sketch P4 codes created from code templates and concatenates N sketch P4 codes into one merged P4 code. A list of sketch P4 codes created by instantiation of code templates is given as an input to the auto-code composition component as in Fig. 5.1. Ultimately, we need one merged P4 code to run on the switch. To do this, a module concatenator gets the input of N sketch P4 codes and concatenates them into one merged P4 code. It is depicted in Fig. 5.5 as the left side of the figure. Note that we need only one init section at the beginning and one end section at the end. In the middle, we have concatenated three sections for each sketch instance.

[1]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).

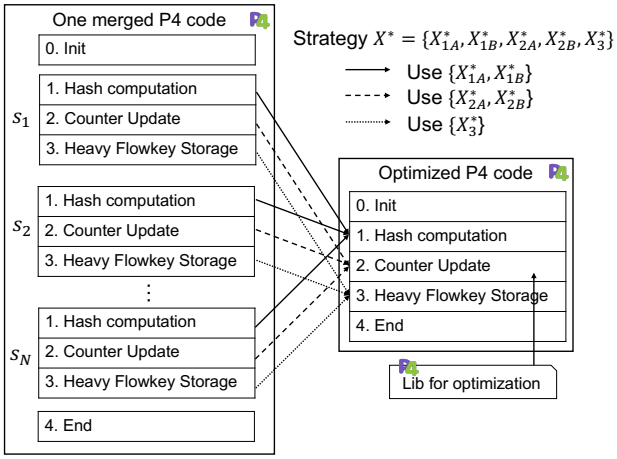


Figure 5.5: Code rewriter uses strategy X^* to create an optimized P4 code

5.3 Step 3. Code Rewrite using Strategy

The third step is code rewriting to translate the selected strategy X^* into optimized code. Code rewriter in Fig. 5.1 gets three inputs; a merged P4 code from the step 2, strategy X^* from the strategy finder, and Library for Optimization (Lib for Opt) that is used to apply *SALU-Merge* (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.

Applying *SALU-Merge* (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. 5.1 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 Table 5.1.

- `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

```
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 5.6: [Before] *Hash-Reuse* (O_{Hash1}) and *Hash-XOR* (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 5.7: [After] *Hash-Reuse* (O_{Hash1}) to $\{s_1, s_2\}$ and *Hash-XOR* (O_{Hash2}) to $\{\{s_1, s_2\}, s_3, s_4\}$

`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}.

For other optimizations $\{O_j\}_{j\in\{Hash1,Hash2,Ctr1,Key\}}$, 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. 5.7). As a result, the code rewriter can translate all five optimizations into an optimized code. Next, we show examples of before and after code snippets for all optimizations.

5.3.1 Before and After Code Snippets for Auto-code Composition

We start by looking at the before and after code snippets for hash computation optimizations (O_{Hash1} and O_{Hash2}) to illustrate how we auto-generate optimized codes. Then, we see the code snippets for counter update optimizations (O_{Ctr1}, O_{Ctr2}), and heavy flowkey optimization (O_{Key}).

Code rewrite for hash computation. Fig. 5.6 is the code snippet without optimization and we call it before code. Fig. 5.7 is the code snippet after applying X^*_{Hash} and we call it after code. We have

$\mathcal{S}$ = $\{s_i\}_{i=1}^{4}$, **F** = $\{\{srcIP\}, \{srcIP\}, \{dstIP\}, \{srcIP, dstIP\}\}$. The before code allocates hash calls to generate hash results for each flowkey (lines 3, 7, 10, 13 in Fig. 5.6). 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} $\oplus$ {dstIP}. If we look at line 4 in Fig. 5.7, the hash result of s_2 reuses the hash result of s_1. Line 6 in Fig. 5.7 shows XOR-based hash result reconstruction. As a result, the usage of the hash call is reduced from 4 to 2.

Code rewrite for counter update. Code rewriter uses $\{X^{*}_{Ctr1}, X^{*}_{Ctr2}\}$ to apply *SALU-Reuse* (O_{Ctr1}) and *SALU-Merge* (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. 5.8) and after (Fig. 5.9) code snippets. Three sketch instances $\{s_1, s_2, s_3\}$ in Fig. 5.8 are count-min sketch, K-ary sketch, and entropy sketch respectively and they have different resource parameters **C** = $\{(r_i, w_i)\}_{i=1}^{3}$ = $\{(3, 2K), (2, 4K), (1, 8K)\}$. $\{s_1, s_2\}$ tracks heavy flowkey and they check whether flow size estimate is above threshold at line 10 and 21 in Fig. 5.8. 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 **W** as discussed as in (4.3), §4.3.2. As a result, **W** = $\{8K, 4K, 2K\}$ is computed in this example and the code rewriter applies this as in lines 4-9 in code snippet Fig. 5.9.

Next, we look at how the code rewriter applies O_{Ctr2} by using X^{*}_{Ctr2}. Fig. 5.10 is the before code snippet and Fig. 5.11 is the after code snippet. $\{s_1, s_2, s_3\}$ in Fig. 5.10 are count-min sketch, entropy sketch, and PCSA sketch respectively and **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. 5.10 whereas s_2 tracks packet

```
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 5.8: [Before] *SALU-Reuse* (O_{Ctr1})

```
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 5.9: [After] *SALU-Reuse* (O_{Ctr1}) to $\{s_1, s_2, s_3\}$

counts at lines 17-18 in Fig. 5.10). 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. 5.11. 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. 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. 5.11.

```
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 5.10: [Before] *SALU-Merge* (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 5.11: [After] *SALU-Merge* (O_{Ctr2}) to $\{s_1, s_2, s_3\}$

Code rewrite for heavy flowkey storage. Code rewriter uses X^*_{Key} to apply *HFS-Reuse* (O_{Key}) to the heavy flowkey storage step. Fig. 5.12 is the before code snippet and Fig. 5.13 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. 5.13. 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. 5.13. 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 5.12: [Before] *HFS-Reuse* (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 5.13: [After] *HFS-Reuse* (O_{Key}) to $\{s_1, s_2, s_3, s_4\}$

Chapter 6

CounterFetchLib: Optimizing Sketch Control Plane on Programmable Switches for Accurate Measurement Results

Recent advances have made it possible to design and implement various telemetry capabilities, such as sketches [32, 37, 43], counting bloom filters [44], and others [34] in programmable switches [9, 11]. At a high level, these network measurement tasks maintain data structures with arrays of counters in the data plane for tracking traffic flows, which are then retrieved by the switch and network control plane.

Our specific focus in this work is on sketches. The typical workflow of sketch-based telemetry is as follows: for every (pre-defined) measurement *epoch* (i.e., a periodic time window), the switch control plane fetches the counter arrays to compute statistics of interest (e.g., heavy hitters, distinct flows, entropy [22, 32, 37, 43, 58, 71, 100]) and resets the counter arrays. Essentially, the counter arrays are *shared state* between the data plane and the control plane. The data plane updates the state when processing packets, and the control plane reads the state per epoch and resets the state for the next epoch.

While the fidelity of the sketches is backed by theoretical analysis [32, 37, 43], in practice when we implement and deploy sketches using the above workflow on programmable hardware switches (e.g., Intel Tofino-based switch), the empirical results are inaccurate (§6.1). For instance, there is

a significant accuracy drop (e.g., up to 94× error increase), when the epoch size is small (e.g., 5s to 1s). To the best of our knowledge, we are the first to document this counter retrieval problem and propose solutions.

We systematically investigate the state fetching and resetting process implemented on an Intel Tofino-based switch [9]. Our analysis shows that the time spent on pulling and resetting data plane states is non-trivial. We decompose delays into the control and data plane delays, identify a total of six potential delays, and quantify the impact of each component. Our analysis reveals that two control plane delays can cause significant impacts on the accuracy of counters (§6.2).

Having identified the key bottlenecks, we propose four correct-by-construction solution building blocks, within the scope of sketching algorithms, with different trade-offs:

- Duplicating sketch instances in the data plane, one of which is updated alternately in successive epochs.
- For sketches with a linearity property [44, 58, 65, 85], the control plane can offset the error by subtracting counter arrays between previous and current epoch.
- Deferring a control plane read operation after a reset operation to reduce the impact of the bottleneck delay.
- Using a faster bulk reset API.

We also propose guidelines on which building blocks are appropriate for different use cases (§6.3). We implement these building blocks for five sketches [32, 37, 43, 47, 71] and evaluate them on a Tofino-based programmable switch [9]. We demonstrate that delays are reduced by more than 95%, and error is reduced by more than 97% (§6.5). While our focus is on sketches, our findings and solutions are likely more broadly applicable to other measurement tasks, since they often share workflow interactions between the data and control plane.

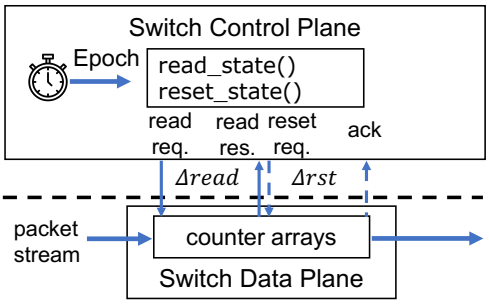


Figure 6.1: Workflow of sketches.

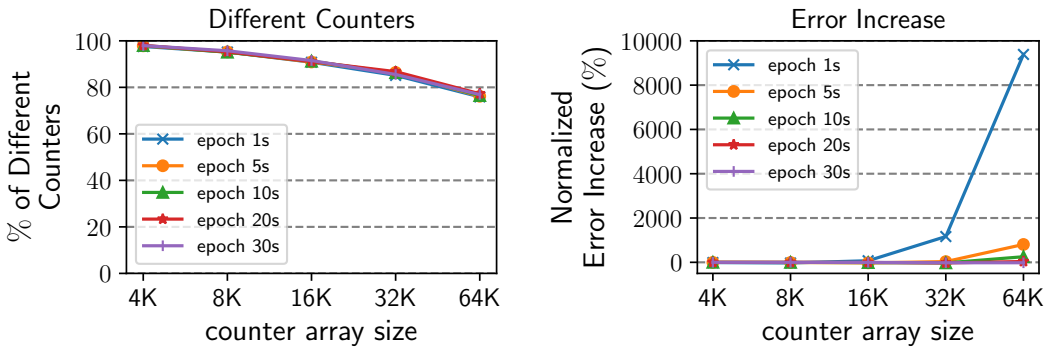
6.1 Motivation

We first describe the common workflow of sketches in Fig. 6.1. We then highlight the sketch accuracy drop in an actual hardware implementation compared to a software implementation and discuss the implications of this observation.

Typical Workflow. Fig. 6.1 shows the common workflow for deploying network measurement tasks on programmable switches. Traffic is chunked into time intervals or *epochs*. On the data plane, network measurement tasks maintain counter arrays that are updated by processing packets. At the end of every epoch, the control plane periodically reads the counter arrays and resets them. We implement five published sketches [32, 37, 43, 47, 71] on a Tofino-based programmable switch using the above workflow and observed a significant discrepancy in accuracy compared with a software implementation. We illustrate the problem using a simple sketch called count-min sketch (CM) [37]. The CM uses a 2D array of counters to detect heavy hitters from a packet stream for a given flowkey (e.g., srcIP). We use srcIP as the flowkey for our implementation.

Methodology. The sketch implementation on the hardware is partitioned across the data and control plane. In the data plane, we run the count-min sketch (CM) [37] written in P4 [29] and send the input packet stream (S) to the switch from a directly connected server using `tcpreplay` [8]. The control plane periodically reads and resets the counter arrays using the control plane API provided by the Tofino SDK [12]. The SDK supports Python and C++, and we present results using C++ API.[1] To obtain the theoretically expected accuracy, we use a software implementation of CM

[1]Based on the conversation with Barefoot, the Python API is not recommended for latency critical applications because Python API is a RPC wrapper for the C++ API.



((a)) Different Counters. **((b)) Error Increase.**

Figure 6.2: Different counters cause accuracy degradation.

sketch written in C++. We split an input packet stream S into multiple subsets S_i corresponding to an epoch with length L.[2] The software implementation pauses packet processing while it reads and resets the counter array. We measured the accuracy of the sketch using both the software and hardware implementations for different epoch lengths (L) and counter array sizes.

Findings. Fig. 6.2(a) shows the percentage of the number of different counters in the counter array between the software simulation and the hardware measurement. We see that up to 98% of counters are different. This discrepancy problem reduces accuracy as shown in Fig. 6.2(b). The normalized error increase is defined as $\frac{Error_{actual} - Error_{exp}}{Error_{exp}}$, where $Error_{exp}$ is the expected error using software sketch implementation relative to ground truth and $Error_{actual}$ is the actual error using the hardware implementation. The error increases up to 94× at an array size of 64K and epoch length of 1. We use an average relative error (§6.5.1) as the error of CM.

Implications. At a high level, the discrepancy arises due to the delays involved in the read and reset operation in Fig. 6.1, $\Delta read$ and Δrst. As we will see later (§6.2), they are not negligible. Note that the above results focus on a simple sketch with a small counter array, a relatively large epoch (1 to 30 seconds), and non-adversarial traffic conditions. In practice, the problem could be worse.

[2]For the input packet stream S, we sample ten one-minute packet traces from inter-ISP packet trace captured on an OC-192 link [1].

- First, richer network measurement tasks that use more data plane counters, such as R-HHH [22] and UnivMon [71], will be impacted more as the impact increases with the size of the counter array.
- Second, network measurement tasks with tighter timing deadlines (shorter epochs) will be impacted more, as the delay becomes more significant relative to the epoch length.
- Lastly, the worst case error can become unbounded; there can be bursts of packets (e.g., anomalies or attacks) that coincide with the $\Delta read$ or Δrst intervals.

6.2 Problem Diagnosis

We take a closer look at the read and reset delays to better understand the discrepancy problem.

6.2.1 A Closer Look at Sources of Error

We can logically decompose the read and reset delays into control and data plane delays, as shown in Fig. 6.3. The read delay at $Epoch_i$, $\Delta read_i$, consists of two control plane delays and one data plane delay: $\Delta read_i = \Delta read_i^{C1} + \Delta read_i^{C2} + \Delta read_i^{D}$, where $\Delta read_i^{D}$ represents the duration of read operation in the data plane. Similarly, we can represent the the reset delay as $\Delta rst_i = \Delta rst_i^{C1} + \Delta rst_i^{C2} + \Delta rst_i^{D}$, with similar control and data plane components.

Let $F(S)$ be the function of sketching algorithm computed on a given set of packets S. For $Epoch_i$, we want to measure $F(S_i)$. However, the above delays cause a different set of packets actually being monitored; they are marked as epoch packets and measured packets in Fig. 6.4.

More specifically (see Fig. 6.3), let $S_{\Delta read_i}$ and $S_{\Delta rst_i}$ denote the sets of packet streams during the read and reset operation in $Epoch_i$. Let $S_{\Delta read_i^{C1}}$, $S_{\Delta rst_i^{C1}}$ denote the sets of packets during $\Delta read_i^{C1}$ and Δrst_i^{C1}. $S_{\Delta read_i^{D}}$, $S_{\Delta rst_i^{D}}$ are more subtle because the control plane and data plane access the counter array simultaneously. We define $S_{\Delta read_i^{D}}$ (similarly $S_{\Delta rst_i^{D}}$) as the set of packets in the traffic stream during $\Delta read_i^{D}$ (Δrst_i^{D}) where packets in this set update the counter array *before* the read and reset operation is executed.

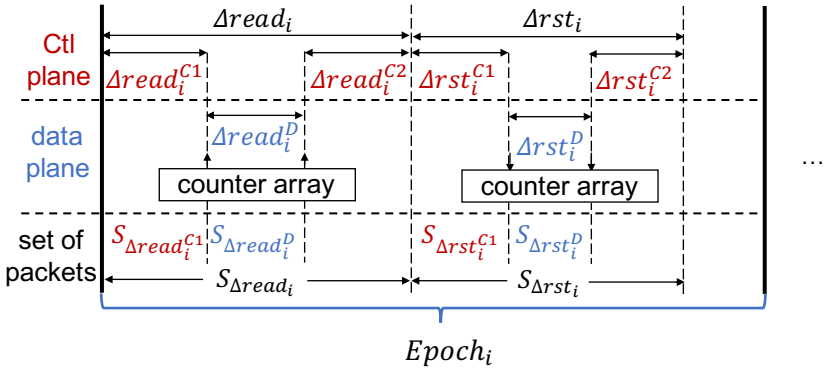


Figure 6.3: Decomposition of the read and reset delays into control plane and data plane delays at $Epoch_i$.

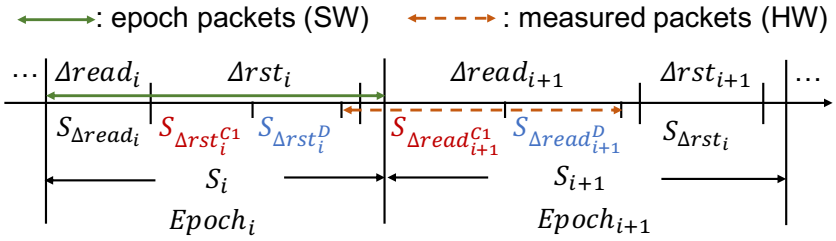


Figure 6.4: Different input packet sets between software and hardware create the discrepancy problem.

The measured packets (the dotted line in Fig. 6.4) is $F(\{S_i - (S_{\Delta read_i} \cup S_{\Delta rst_i^{C1}} \cup S_{\Delta rst_i^D})\} \cup \{ S_{\Delta read_{i+1}^{C1}} \cup S_{\Delta read_{i+1}^D} \})$. Note that the effect of $\Delta read_i^{C2}$ is included in $S_{\Delta read_i}$. Next, we quantify the delays that cause the loss in accuracy.

6.2.2 Quantifying Sources of Error

To understand the magnitude of impact from each source of delay, we measure and quantify each delay.

Methodology. We measure the delays by sending packets at a controlled rate to the switch data plane and reading the counter values into the control plane. We use custom benchmarking programs in addition to the sketch implementations—data plane program using P4 language and the control plane using C++ API. Our measurements use efficient control plane read and reset operations. For the read operation, we utilize *table sync operation* which uses bulk DMA transfer from data plane counter arrays into control plane buffer so that the control plane can read counters

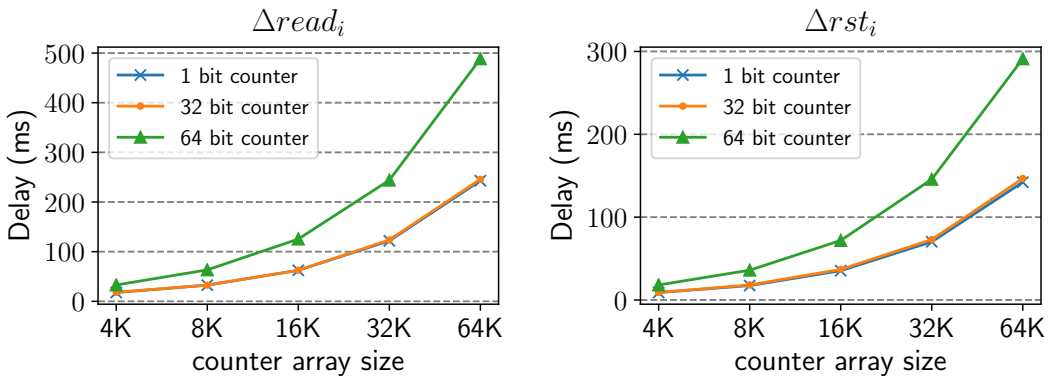


Figure 6.5: The read and reset delays (ms).

quickly. For the reset operation, we use the transaction API, which accelerates the individual write operations.

To measure $\Delta read_i^D$ we need to measure the time during the data reads from the first counter *counter[0]* to the last counter *counter[N-1]* of a counter array. This can be measured by synthesizing a packet stream that contains two packets every $100\mu s$ and using them to increment (+1) *counter[0]* and *counter[N-1]* respectively. In this way, $\Delta read_i^D$ can be measured by (*counter[N-1]* −*counter[0]*) $\times 100\mu s$ because the read operation is executed sequentially from 0 to $N-1$. The server uses `tcpreplay` to send this synthesized packet stream to the directly connected switch while the control plane executes the read operation.

We use the same setup to measure the duration of the data plane reset operation Δrst_i^D. However, since the reset operation resets counter values sequentially starting from the first one, the control plane executes the reset operation during `tcpreplay` and then executes the read operation after `tcpreplay` is finished. (*counter[0]* −*counter[N-1]*) $\times 100\mu s$ then represents Δrst_i^D. To measure $\Delta read_i^{C1}$ and $\Delta read_i^{C2}$, the control plane reads the first counter value before and after the read operation and we can then calculate those values using subtraction. We apply the same ideas for measuring Δrst_i^{C1} and Δrst_i^{C2}.

Result. Fig. 6.5 shows the $\Delta read_i$ and Δrst_i delays for different counter array sizes and counters (e.g., 1-bit, 32-bit). The read delay $\Delta read_i$ can be up to 488 ms and the reset delay Δrst_i can be up to 291 ms. Both delays increase linearly as the size of the counter array increases. For different

Delays		4K	16K	64K
$\Delta read_i$	$\Delta read_i^{C1}$	0.30	0.97	3.62
	$\Delta read_i^{D}$	0.01	0.07	0.31
	$\Delta read_i^{C2}$	22.21	66.70	244.49
	Total	**22.53**	**67.74**	**248.43**
Δrst_i	Δrst_i^{C1}	16.45	41.69	145.53
	Δrst_i^{D}	0.09	0.36	1.49
	Δrst_i^{C2}	0.02	0.02	0.03
	Total	**16.56**	**42.08**	**147.06**
$\Delta read_i$ + Δrst_i		**39.10**	**109.81**	**395.48**

Table 6.1: Six delay measurement (ms).

counter sizes, a 64-bit counter takes 1.97× more delay than a 32-bit counter because the switch maintains a 64-bit counter as a pair of 32-bit counters. However, the delay difference between 32-bit and 1-bit counter is marginal (1.01×).[3]

Next, we look at six decomposed delays for 32-bit counters in Table 6.1. Surprisingly, $\Delta read_i^D$ and Δrst_i^D take less than 0.1%, 0.4% of the total read and reset delays. Meanwhile, we can see that $\Delta read_i^{C2}$ and Δrst_i^{C1} are the dominant factors as they take up more than 98% of the sum of the read and reset delays.

Key takeaways. Out of six delays, two *control plane* delays $\Delta read_i^{C2}$ and Δrst_i^{C1} are dominant factors. For example, $\Delta read_i^{C2}$ (Δrst_i^{C1}) of 16K array size takes 61% (38%) of the total sum of delays. Across all sizes of counter arrays, both bottleneck delays together account for 99% of the total delay.

6.3 Building Blocks and Solution Guidelines

In this section, we propose four solution building blocks to mask or reduce the delays identified in the previous section. These have varying trade-offs regarding the epoch size they can support, resource usage, general applicability across tasks. Table 6.2 summarizes these trade-offs. We also provide some general guidelines for combining building blocks as solutions appropriate for different use cases.

[3]We cannot measure the delays for 1-bit counters with the described methodology because 1-bit counter can not store an integer value. Instead, we used a timer in the control plane program to measure delays for the 1-bit counter in Fig. 6.5.

6.3.1 Building Blocks

B1: Use duplicate counters. A simple idea is to duplicate sketch instances in the data plane and alternately use them for odd/even epochs. At $Epoch_i$, counter array in sketch instance 1 can be updated in the data plane while the control plane reads and resets sketch instance 2. Then at $Epoch_{i+1}$, counter array in sketch instance 2 can be updated in the data plane while the control plane reads and resets instance 1.

Trade-off. This idea masks all delays and the key bottleneck delays. However, this idea requires 2× the data plane memory. Realizing it also requires some data plane code (P4) change.

B2: Offset counter errors in the control plane. Some sketches have a *linearity property* [81]. That is, counter arrays can be combined in a mathematical sense by addition and subtraction of each counter. In such cases, the control plane can avoid explicitly resetting or duplicating the counters. Instead, it stores the counter arrays reported from the previous epoch (in the control plane) and obtains the counters for the current epoch by subtracting the previous counter arrays from the counter arrays reported at the current epoch.[4]

Trade-off. This idea masks the key bottleneck delays of the reset and does not incur any additional data plane resources. It only requires small control plane code updates in order to subtract counter arrays. However, this idea is only applicable to sketches satisfying the linearity property. Fortunately, we've seen a range of linear sketches such as [32, 44, 58, 65, 85] for various measurements. We do see one caveat that some sketches for tracking heavy hitters in the data plane [37] need to access per-epoch counters to identify heavy flowkeys. Since the data plane only stores accumulated values, we cannot obtain per-epoch values directly in the data plane.

B3: Defer control plane read operation. We observe that during $\Delta read_i^{C2}$, most of the time is spent on reading counter arrays from an internal buffer in the control plane. That is, data is already transferred from the data plane using bulk DMA transfer as in Fig. 6.6. Thus, we can defer this operation of reading data from the buffer after the reset operation. We can implement this idea

[4]The idea of not resetting the counters across epochs can bring up a concern of overflow. However, subtracting two counter array still works as long as there is at most 1 overflow per epoch. Empirically, the 32-bit counter is large enough to avoid two overflows.

Building Blocks	$\Delta read_i^{C2}$	Δrst_i^{C1}	Epoch	Gen.	Res.
B1	hide	hide	smallest	✓	2x
B2	hide	hide	small	×	1x
B3	hide	×	med	✓	1x
B4	×	reduce	med	✓	1x

Table 6.2: Tradeoffs for solution building blocks in different metrics such as hiding/reducing two bottleneck delays, epoch size it can support, generality, resource usages.

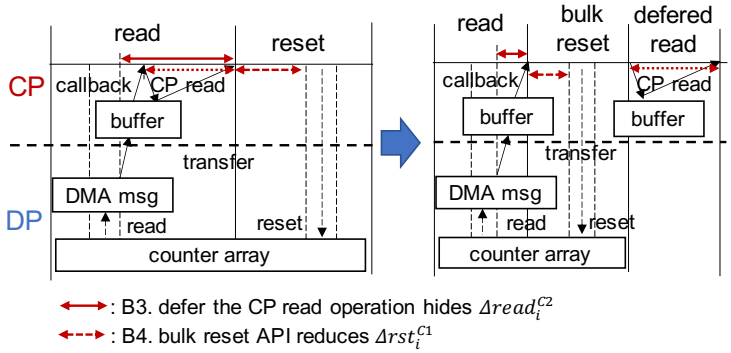


Figure 6.6: B3: Defer control plane read operation and B4: Use bulk reset API.

because 1) the reset operation does not reset the internal buffer and 2) the read operation can be divided into separate API calls: bulk DMA transfer and reading data from the internal buffer.

Trade-off. This idea does not require additional resources and it can be applied to sketches without linearity property. However, it only reduces the effect of $\Delta read_i^{C2}$.

B4: Use bulk reset API. This solution building block directly reduces Δrst_i^{C1} as in Fig. 6.6. We observed that the basic control plane support for reset updates counters one at a time. This is effectively a write operation and provides a more general capability to write an arbitrary value at a specific location. However, we note that there is also a `clear` API that suffices for our needs well since it resets all of the counter arrays to zero with much lower delay (18× faster).[5]

Trade-off. This idea only reduces the effect of Δrst_i^{C1}, thus it still can suffer accuracy degradation for a small epoch length.

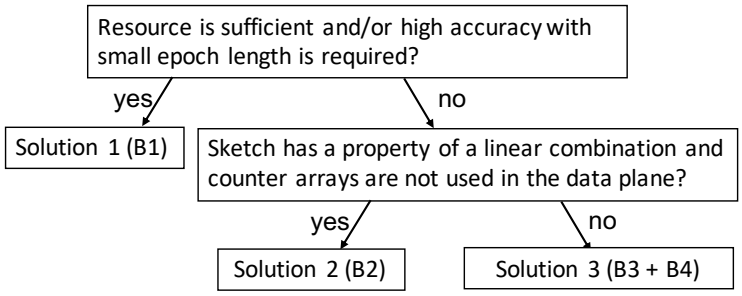


Figure 6.7: Decision tree for selecting solutions.

6.3.2 Guidelines for Sketch Developers

Based on the above building blocks, we suggest a guideline for sketch developers on which solution is appropriate for different use cases summarized in the decision tree (Fig. 6.7):

- **Solution1 (B1)** would fit for small sketches and/or resources are sufficient. B1 provides the highest fidelity, especially for small epoch length.
- **Solution2 (B2)** uses low resource footprint. It is a simple solution for sketches satisfying linearity when counter arrays are not used in the data plane.
- **Solution3 (Combine B3 and B4)**. These two building blocks can be combined to tackle two bottleneck delays. The combined solution requires some implementation effort but is general and is appropriate when resource overhead is critical.

6.4 API calls

We write C++ API calls to implement read and reset operations for three solutions as in Table 6.3.

- `int* read_counters (counter_name, row, width)` reads counter arrays and returns fetched counter arrays to compute measurement results. This API call can be used for solution 1. It gets parameters of counter_name to specify counter arrays to read. Row and width information of counter arrays should be specified.
- `int* read_reset_counters (counter_name, row, width)` both reads and resets counter arrays. It internally implements deferring control plane read (B3) operation after

[5] According to the conversation with Intel, Tofino2 supports an even faster bulk reset API.

API Name	API Parameters	Explanation
`int* read_counters()`	counter_name, row, width	For Solution 2
`int* read_reset_counters()`	counter_name, row, width	For Solution 1 and Solution 3

Table 6.3: API calls extended from SketchLib and Lib for optimization

bulk reset calls (B4). Thus this API call can be used for solution 3. This API call is also compatible with solution 1 as well.

6.5 Evaluation

Our evaluation demonstrates that (a) all solutions significantly reduce the error of the hardware implementation relative to the expected accuracy and (b) the implementation effort for the solutions is marginal in terms of additional lines of code.

6.5.1 Experimental Setup

Testbed. We use 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 Tofino SDE version 9.1.1 in our experiment. We send the trace to the switch from a directly connected server using `tcpreplay`.

Traces. We use sampled ten one-minute packet traces from CAIDA backbone traces capture at 1/21/16 Chicago [1].[6]

Sketches. We implement five sketches, MRB [43], HLL [47], count sketch (CS) [32], count-min sketch (CM) [37], and UnivMon (UM) [71] using P4 language. MRB uses 1-bit counters and the rest of the sketches use 32-bit counters. MRB and HLL use one counter array and CS, CM, UM use four counter arrays. MRB, HLL estimate cardinality, CS, CM estimate the average relative error of top-100 heavy hitter flow counts, and UM estimates entropy. Note that out of five sketches, CS, CM, UM satisfy the linearity property. We assume that we know all of the flowkeys for CS, CM, UM since identifying heavy flowkeys on the data plane is orthogonal to this work. We use P4 version of $P4_{16}$.

[6] We also run experiments with other traces such as data center traces [25] and attack traces [2]. Results are similar thus, they are not shown.

A.size	MRB 64K	HLL 4K	CS 64K	CM 64K	UM 128K
Unopt	1273/2%	91/1%	618K/73%	700K/76%	1030K/26%
Sol 1	0/0%	0/0%	0/0%	0/0%	0/0%
Sol 2	×	×	10K/7%	10K/7%	16K/5%
Sol 3	5/0%	3/0%	22K/12%	22K/13%	33K/8%

Table 6.4: Total counter value difference / relative counter difference for five sketches and three solutions using epoch=1s.

Array size		MRB 64K	HLL 4K	CS 64K	CM 64K	UM 128K
Expected Errors	**Ideal sketch**	1.6%	4.8%	0.7%	0.4%	2.8%
Actual Errors	**Unopt**	20.1%	6.2%	35.4%	34.8%	64.7%
	Sol 1	1.6%	4.8%	0.7%	0.4%	2.8%
	Sol 2	×	×	1.0%	0.7%	2.8%
	Sol 3	1.7%	4.8%	1.5%	1.1%	3.6%

Table 6.5: Expected errors vs. actual errors using epoch=1s.

Metrics of difference. We consider three types of metrics:

- *Raw counters:* We consider both the total counter value difference = $\sum_i |expected[i] - actual[i]|$ and the relative counter difference = $\frac{\sum_i (expected[i] != actual[i])}{array\_size}$.
- *Sketch Errors:* Average Relative Error (ARE) is $\frac{1}{k}\sum_{i=1}^{k}\frac{|f_i - \hat{f}_i|}{f_i}$, where k is 100. f_i is true flow count, $\hat{f}_i$ is flow count estimate, and $f_i \geq f_{i+1}$ for any i. This metric is used for CS and CM. Relative Error (RE) is $\frac{|\text{True} - \text{Estimate}|}{\text{True}}$, where $True$ is true statistic value and $Estimate$ is estimated value. This metric is used for MRB, HLL, UM.
- *Delay:* We measure the sum of delays that corresponds to union and subtraction components in §6.2.1: $\Delta read_i + \Delta rst_i^{C1} + \Delta rst_i^{D} + \Delta read_{i+1}^{C1} + \Delta read_{i+1}^{D}$.

6.5.2 Error and Delay Reduction

Counter difference reduction. We first look at the counter difference reduction in Table 6.4. We use a fixed epoch length of 1 second. We can see that all solutions reduce almost all of the total counter value difference compared to unoptimized hardware implementation. Specifically, Sol 1 incurs no counter difference, and Sol 2, Sol 3 incur negligible counter differences. Note that the total counter value difference has a more direct effect on sketch accuracy than the relative counter difference.

	4K	16K	64K
Unopt	**39.39**	**110.84**	**399.39**
Sol 1	0 (100%)	0 (100%)	0 (100%)
Sol 2	0.32 (99.20%)	1.04 (99.06%)	3.94 (99.01%)
Sol 3	1.53 (96.11%)	4.66 (95.79%)	16.67 (95.83%)

Table 6.6: The sum of delays after applying solutions in ms (% of reduction compared to unoptimized).

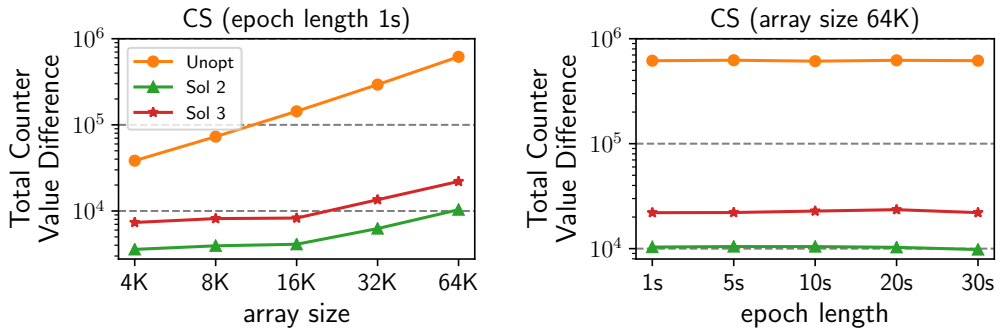


Figure 6.8: Total counter value difference for CS.

Error reduction. Next, we look at the error reduction in Table 6.5. Compared to errors on unoptimized implementation, actual errors on all solutions are almost close to expected errors measured on software implementation.

Delay reduction. Table 6.6 shows that all solutions reduce delays significantly. Sol 1 does not incur any delays. Sol 2 can reduce delays by 99% across all counter array sizes. Sol 3 also reduces delays by 95%. Note that the delays after applying solutions are still linear to the counter array size.

Detailed measurement. We observe reductions for fixed array size and epoch length. We pick one sketch (CS) and look at the counter differences and error reductions for different array sizes and epoch lengths. Fig. 6.8 shows that as array size increases, the total counter value difference increases linearly, but it is constant over epoch lengths. Note that Sol 1 does not incur any counter differences across all array sizes and epoch lengths. Fig. 6.9 shows that the error gap between expected and un-optimized measurement is increasing as array size increases and epoch length decreases. All solutions effectively reduce this gap, and they show similar errors as expected.

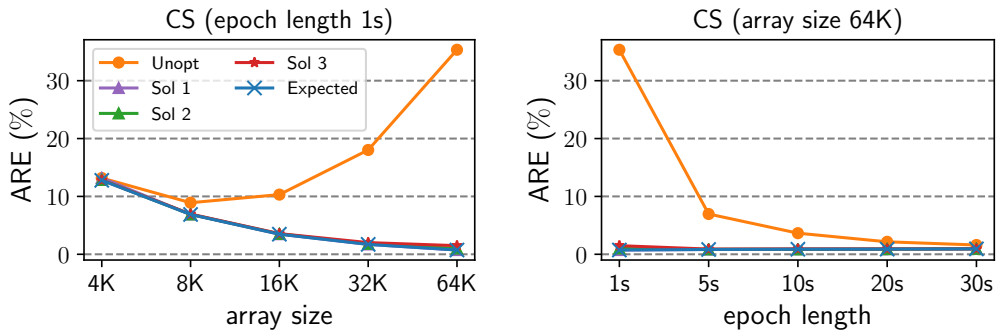


Figure 6.9: Average relative error for CS.

Additional Lines of code	Sol 1 B1	Sol 2 B2	Sol 3 B3	Sol 3 B4
Data Plane P4 Code	29	0	0	0
Control Plane Program (C++ API)	63	0	0	19
Offline Processing	0	9	0	0

Table 6.7: Additional lines of code for implementing solutions.

6.5.3 Implementation Effort

Table 6.7 shows additional lines of code for implementing solutions. Sol 1 requires P4 code change for duplicating instances and C++ control plane program change for reading instances alternatively. Code change for Sol 2 is in an offline processing program written in Python for subtracting counter arrays. B3 in Sol 3 does not incur any additional lines of code since it just swaps the order of the control plane read and reset operation. B4 in Sol 3 requires additional control plane program code for bulk reset API.

6.6 Related work

Sketch-based telemetry. Sketches have emerged as a promising telemetry solution for flow-level measurements, including heavy hitters [32, 37, 71, 88, 100], entropy estimation [71, 84, 86], change detection [65, 103], and distinct flows [47, 71]. While recent efforts [31, 107] propose to maintain more light-weight sketches per device, they still suffer from the incorrect counter retrieval issue in programmable switches and can benefit from our solutions.

Other work in network telemetry. There are complementary telemetry capabilities that focus on packet-level and path-level monitoring (e.g., INT [63] and PINT [24]), higher-order telemetry (e.g., performance statistics [33, 51, 74], application level monitoring [94]), diagnosis [56, 62], as well as network-wide adaptive telemetry [52, 55]. A not hard extension is to explore if these measurement tasks can suffer from a similar incorrect state retrieval and reset problem.

Other programmable platforms. In addition to the switches discussed in this paper, SmartNICs such as multicore SoC NICs [5, 6] and FPGA NICs [13] are platforms for telemetry. Recent work [61] in measuring the performances of various SmartNICs demonstrated a similar bottleneck between the data plane and the control plane. A future direction is to explore the telemetry retrieval inaccuracy problem in SmartNICs.

6.7 Summary

We identify and quantify the causes of an accuracy degradation in the switch control plane when we deploy sketching algorithms on programmable switches. Our solutions informed by our analysis can eliminate almost all the inaccuracy for five sketches. We believe our insights are more broadly applicable to other network measurement tasks with similar control-data plane interactions.

Chapter 7

Conclusions

This chapter summarizes our contributions (§7.1) and presents reflections and lessons learned while designing our systems (§7.2). Finally, it ends with future research direction (§7.3).

7.1 Summary of Contributions

In this thesis, we argue that we can enable performant and practical sketch-based network telemetry on programmable switches, by proposing optimizations to the sketch implementations, and by providing APIs and the code composition framework that automatically generates optimized codes.

With **SketchLib** (chapter 3), we propose six novel per-sketch optimizations to deploy a single sketch instance on the data plane of programmable switches. We also defined and implemented an API that realizes these optimizations, saving developers development time. SketchLib can reduce resource bottlenecks by up to 90%, making many previously infeasible sketches feasible. We show that SketchLib can apply to a broad range of sketching algorithms.

With **Sketchovsky** (chapter 4), we propose five cross-sketch optimizations to deploy an ensemble of sketch instances on the data plane of programmable switches. There are many opportunities to reuse hardware resources across sketch instances. However, searching for the optimal strategy is intractable and can take days, because there are many ways to use and combine the optimizations. We propose a set of greedy heuristic algorithms that allow us to fine a near-optimal strategy in less than one second. Using this strategy, we show that we can reduce up to 45% of resource bottle-

necks of ensembles of sketch instances. Sketchovsky makes many previously infeasible ensembles of sketch instances feasible.

Using our **Auto-code Composition Framework** (chapter 5), optimized P4 code for an ensemble of sketch instances can be automatically generated by simple code rewrites. Although Sketchlib and Sketchovsky can quickly find near-optimal strategies, manually applying them to an ensemble of sketch instances and writing optimized code is a demanding task. Our auto-code composition framework can reduce the development time significantly for translating picked strategy into the optimized code.

To help with the design of **CounterFetchLib** (chapter 6), we did bottleneck analysis to find the root cause of the measurement errors caused by the sharing of counters by the control and data plane and propose solutions to reduce execution time for read and reset operations. As a result, the delay is reduced by 95%, and measurement error is reduced by 97%. Further, we provide API calls for developers to reduce the development time of sketch control plane implementations.

7.1.1 Putting it all together

All four systems are integrated together to enable performant and practical sketch-based network telemetry on programmable switches. Given an ensemble of sketch instances, the auto-code composition framework receives the strategy from Sketchovsky to automatically generate optimized code. When generating code, each sketch instance is created based on sketch templates, and templates use API calls from SketchLib. Thus, templates become more simple and more concise while per-sketch optimizations are automatically applied. CounterFetchLib is used to optimize read and reset operation in the control plane for both a single sketch instance (SketchLib) and an ensemble of sketch instances (Sketchovsky and Auto-code Composition).

7.2 Lessons Learned

Reflection 1: RMT hardware is difficult to deal with. It takes a significant amount of time to understand the inner workings of the RMT programmable hardware switch. The switch utilizes

various hardware resources for implementing sketches, and figuring out constraints and dependencies among hardware resources is challenging. For example, different hardware resources are used together to perform hash computations (e.g., hash bits, hash calculation units, and hash distribution units). It is important to understand the function of each hardware component and how they are integrated together to identify resource bottlenecks (e.g., hash distribution unit was the bottleneck). The packet header vector (PHV) compilation error was also difficult to understand: the occurrence of error depends on the usage of other resources, but detailed information on these constraints is not publicly available. Understanding these constraints and dependencies requires many iterations of the compilation with trials and errors. Thus, the difficulty of learning the internal architecture and writing data plane and control plane programs is a big factor in hindering the deployment of programmable switches into industry settings. In this sense, our approach of providing APIs, auto-code composition framework, and open-source codes is a great step towards reaping the true potential of deploying sketching algorithms deployed on programmable switches.

Reflection 2: sketching algorithms on programmable switches are powerful. From my thesis, I learned that running sketching algorithms on programmable switches for flow-level network telemetry is indeed feasible and powerful. We made many contributions to realize this goal and showed its feasibility using actual hardware switches. We believe this approach has huge potential for better managing and debugging real-world problems. Many practitioners are still relying on packet sampling or NetFlow, and they may suffer from inflexibility and inaccuracy as a result. Conversations with industry and companies confirm the desire for rich flow-level network telemetry.

Reflection 3: we still have many missing pieces. Although this thesis achieves significant improvements, there are still many missing pieces to fully realize the four goals that motivated our research. The first of these is generality. We only focused on around 20 sketching algorithms on RMT-based hardware architecture. It would be beneficial to incorporate more sketching algorithms on diverse hardware devices. The second of these is a global view of the network system. We focused on implementing sketching efficiently on a single device, but what about multiple de-

vices with different network topologies with different devices? There are new challenges, such as how to distribute workload to each device correctly as if it is one big switch abstraction. The next section will discuss these limitations and promising future research directions.

7.3 Future Work

The ultimate vision for performant sketch-based network telemetry is to make network telemetry easier to use and manageable across various hardware devices in network-wide settings. Although our work makes solid contributions towards achieving this goal, there are still some missing pieces. To this end, we propose promising future research directions to fill those gaps.

Query - sketching mapping. The scope of this thesis assumes that the input is ensembles of sketch instances with fixed resource configuration. However, deciding on a list of sketch instances with resource parameters can be challenging for network operators. Ideally, network operators have a set of queries they want to run, and this set of queries should be the input of the problem. Each query has its own statistic (e.g., topk flows or cardinality) to compute, definitions of flowkey and flowsize, and desired accuracy level (e.g., 1% or 0.1% of error). Given this set of queries, finding a set of sketch instances with resource parameters is challenging. This mapping must consider resource-accuracy trade-offs for all sketching algorithms based on the knowledge that some sketching algorithms can support multiple statistics. This mapping process should also consider the objective function of which hardware resources must be minimized, because different sketching algorithms use different resource usage profiles, and network operators have different needs for objective functions depending on other network functions running in parallel with sketch instances. Moreover, Sketchovsky must be considered because resource usage varies due to our cross-sketch optimizations for different sets of sketch instances. Given these challenges and potential benefits, finding an optimal set of sketch instances given a set of queries is an interesting and promising direction for future work.

Optimization using compiler. Although our proposed per/cross-sketch optimizations are effective and easy to use by using API calls and auto-code composition, our scope is limited in the

sense that developers should only rely on our APIs, and the auto-code composition is based on sketch templates. Suppose developers can freely write programs for a single sketch or ensemble of sketching algorithms, and the compiler can detect inefficiency automatically and apply optimizations internally. In that case, it will be more powerful and beneficial to developers. The challenge of this approach is that compiler should be able to recognize all different ways to program sketching algorithms and apply appropriate optimizations.

Expanding generality on many dimensions. Sketch-based network telemetry can be much more improved by expanding generality in many dimensions.

- **Sketching algorithms.** In SketchLib and Sketchovsky, we considered around 20 different sketching algorithms. However, many more sketching algorithms exist and are being developed [69]. Thus, incorporating more sketching algorithms to show existing optimizations can be applicable or developing new optimizations is a promising direction for future research. Specifically, we can think of adding more sketch features. For example, Sketchovsky considered two counter array types; 2D counter arrays of single-level (SL) or 3D of multi-level (ML). However, the scope can be easily expanded by adding more types, such as 4D counter arrays [77, 92]. In this way, considering more sketch features to cover more sketching algorithms is a promising direction for future work.

- **Programmable switches.** While we focused on the Tofino programmable switch, there are other programmable switches as well (e.g., Broadcom Trident [10]). As those programmable switches have different architecture and different programming languages [15], expanding our approach of per/cross-sketch optimizations, control plane optimizations, auto-code composition and APIs to other programmable switches is an interesting direction.

- **Other programmable network data plane devices.** Besides programmable switches, the recent trend of in-network computing made other network devices more programmable. Network Interface Cards (NICs) with Network Processing Units (NPUs) have programmability (a.k.a. smartNICs). Field Programmable Gate Arrays (FPGAs) can also support data plane

programmability. A few sketching algorithms are optimized and deployed on these platforms, but there is no effort to provide API and code composition for a set of various sketching algorithms. Expanding our systems to these heterogeneous programmable network devices seems promising. We posit that our per/cross sketch optimizations and the notion of APIs and auto-code composition can be similarly applied to other hardware devices..

Network-wide deployment of sketching algorithms on heterogenous devices. We only looked at deploying sketching algorithms on a single hardware device. However, network systems are composed of many heterogeneous data plane devices with different network topologies. Given this, how to distribute a set of sketch instances to heterogeneous network devices in network-wide deployment is an open challenge. It is known that there are affinities between sketching algorithms and hardware devices [18] (e.g., specific sketching algorithms are more efficient on specific hardware devices). If specific sketch instances are running together on a single device, it can be more efficient depending on the applicable conditions of cross-sketch optimizations. Considering all these constraints make the problem more challenging.

Appendix A

SketchLib Appendix

A.1 Comparison of RMT resource mapper and Tofino compiler

To validate RMT resource mapper as a proxy for Tofino compiler, we conduct experiments to compare resource allocation results of RMT resource mapper and the Tofino compiler. We pick five different sketches (UnivMon, R-HHH, PCSA, HLL, and MRB). We vary one parameter of sketches while fixing other parameters and analyze the resource allocation results. We focus on five different resource types; pipeline stages, hash calls, SALU, SRAM, and TCAM.

Fig. A.1–Fig. A.5 illustrate the results. Note that all of the resource usages are normalized. We can see that for hash calls, SALU, SRAM, and TCAM usages are identical between RMT resource mapper and the Tofino compiler. For pipeline stages, results are the same for PCSA, HLL, and MRB. However, RMT resource mapper finds mapping which uses fewer pipeline stages than the Tofino compiler for UnivMon and R-HHH. RMT resource mapper minimizes stages while the Tofino compiler finds more sparse mapping (e.g., mapping a small number of tables per stage). We validate both of the mappings from RMT resource mapper and Tofino compiler are valid. We confirm with the vendor that the Tofino compiler uses complex heuristics and the cost function of power budget and compilation time, which are different from that of RMT resource mapper and can introduce the gap. Our extensions to the RMT resource mapper is available at [17].

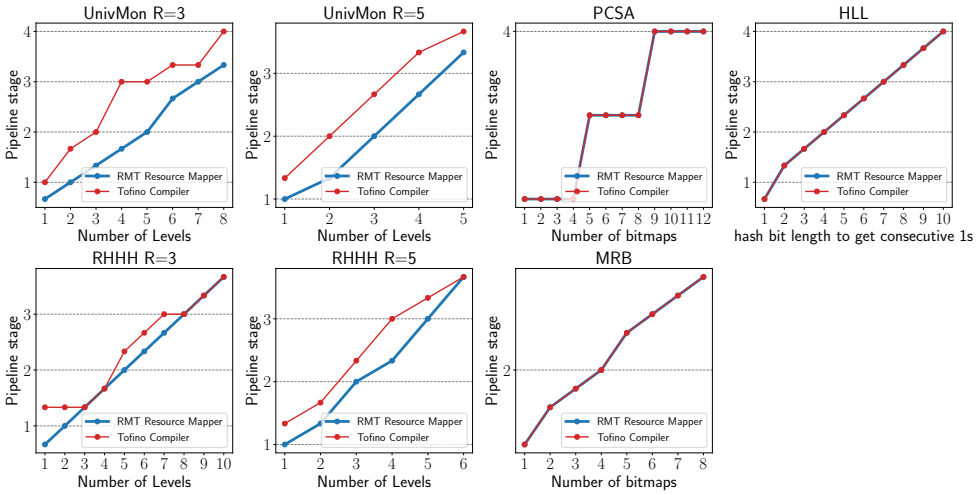


Figure A.1: RMT resource mapper vs. Tofino compiler: pipeline stages

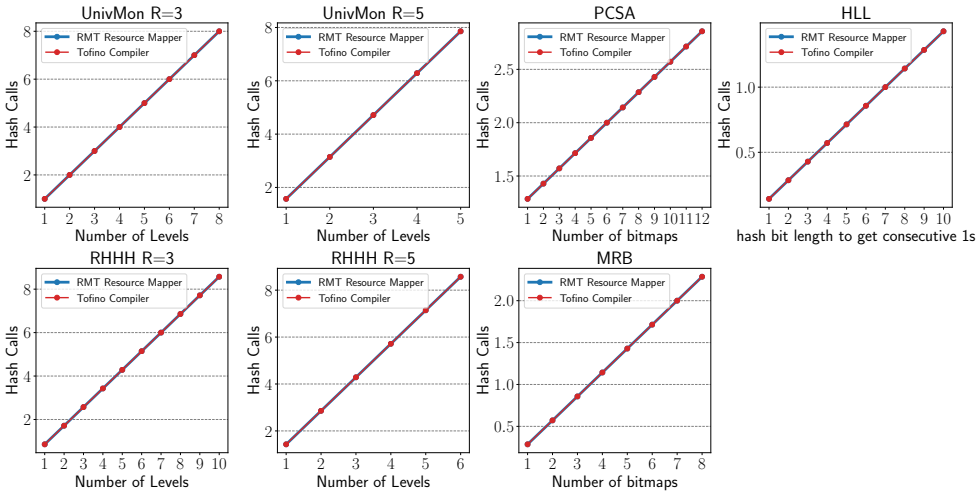


Figure A.2: RMT resource mapper vs. Tofino compiler: Hash Call

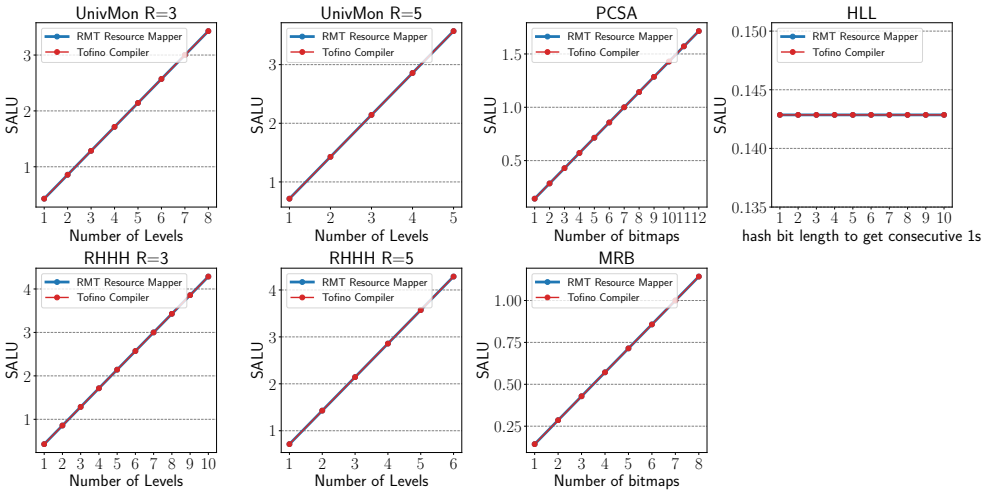


Figure A.3: RMT resource mapper vs. Tofino compiler: SALU

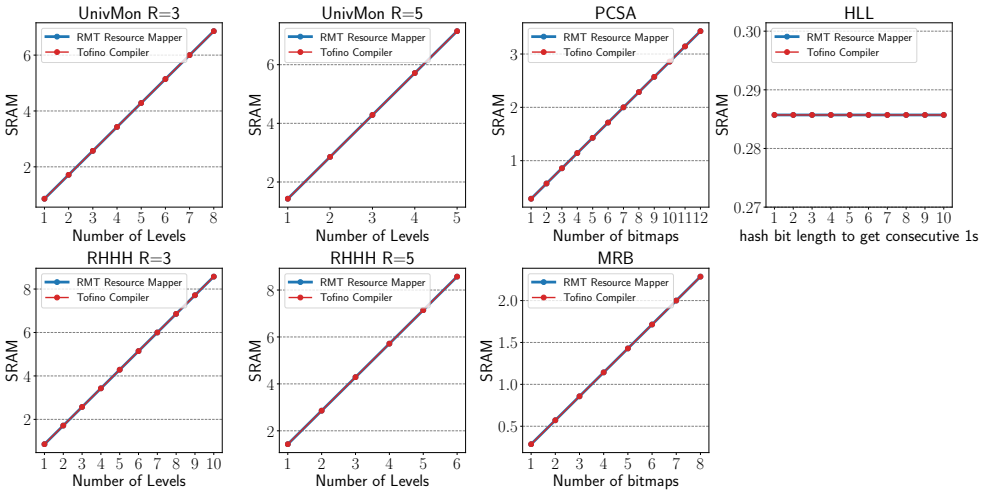


Figure A.4: RMT resource mapper vs. Tofino compiler: SRAM

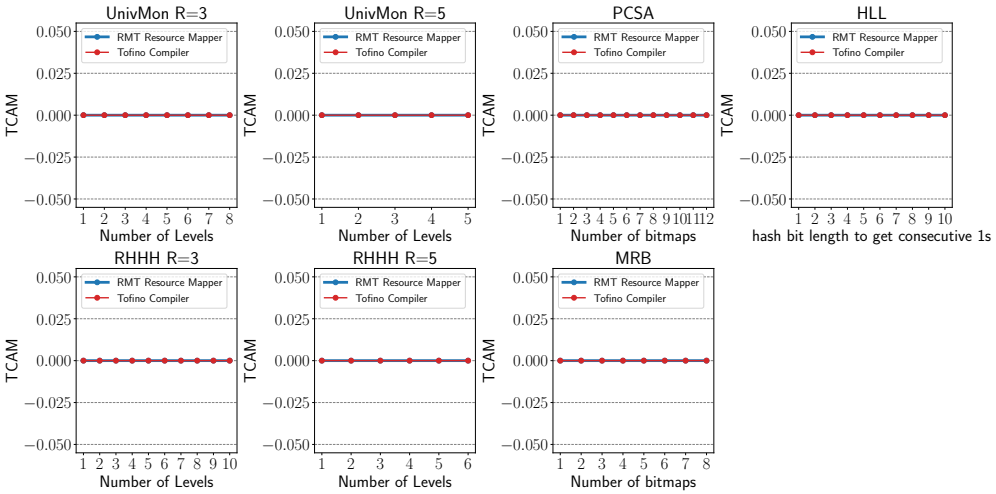


Figure A.5: RMT resource mapper vs. Tofino compiler: TCAM

Appendix B

Sketchovsky Appendix

B.1 Supplement to Background

B.1.1 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 [41, 47]. 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., $\rho(0001\ldots) = 4$) and $hash$ is computed using flowkey [47]. This ρ function can be implemented efficiently by using TCAM in the switch data plane [17].
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

```
1:  function BITMAP(index)
2:      A[index] = 1
3:  function COUNTER(index, size)
4:      A[index] = A[index] + size
5:  function SIGNCOUNTER(hash, index, size)
6:      if hash == 0 then
7:          A[index] = A[index] + size
8:      else if hash == 1 then
9:          A[index] = A[index] - size
10: function HLL(index, value)
11:     if A[index] < value then
12:         A[index] = value
13: function PCSA(index, bitmask)
14:     A[index] = A[index] | bitmask
```

B.2 Supplement to Optimizations

B.2.1 SRAM reduction of *SALU-Reuse* (O_{Ctr1})

SALU-Reuse (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}.

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

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 $\rightarrow comp(x, y) = 0$.

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

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

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

B.3 Supplement to Evaluation

B.3.1 Eleven Sketch Algorithms for Evaluation

We use eleven sketching algorithms for our evaluation as in Table B.1. 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 B.1.

Sketch Algorithms		**Sketch Features**			**Configurable Parameters Candidates**					
Statistic	Name	Counter Array	Counter Update	Heavy Flowkey	Flowkey	Flowsize	Epoch	Row	Width	Level
Membership	BF [27]	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 [96]	SL	BITMAP	N					{16K, 32K}	{8, 16}
	MRB [43]	ML	BITMAP	N						
	PCSA [46]	SL	PCSA	N					{4K, 8K, 16K}	-
	HLL [47]	SL	HLL	N						
HH/HC	CS [32]	SL	SIGNCOUNTER	Y		{counts, bytes}		{1, 2, 3, 4, 5}		
	CM [37]	SL	COUNTER	Y						
	KARY [65]	SL	COUNTER	Y						
Entropy	ENT [67]	SL	COUNTER	N		{counts}		{3,4,5}	{2K}	{16}
General	UM [71]	ML	SIGNCOUNTER	Y						
FSD	MRAC [66]	ML	COUNTER	N				{1}		{8, 16}

Table B.1: Eleven sketch algorithms with sketch features and possible configurable parameters. (4-tuple) = (srcIP, dstIP, srcPort, dstPort). (5-tuple) = (srcIP, dstIP, srcPort, dstPort, proto).

B.3.2 Four Ensembles for Accuracy Evaluation

Table B.2 - Table B.5 shows four picked ensembles for four ensemble types. All five optimizations are found in four picked ensembles.

Ensemble Type 1.

- *Hash-Reuse* (O_{Hash1}): none
- *Hash-XOR* (O_{Hash2}): none
- *SALU-Reuse* (O_{Ctr1}): none
- *SALU-Merge* (O_{Ctr2}): none
- *HFS-Reuse* (O_{Key}): $\{s_1, s_2, s_3, s_4, s_5, s_6\}$

Ensemble Type 2.

- *Hash-Reuse* (O_{Hash1}): $\{s_3, s_4, s_6, s_{10}\}$

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 B.2: Ensemble Type 1. Same Sketch Algorithm

- *Hash-XOR* (O_{Hash2}): none
- *SALU-Reuse* (O_{Ctr1}): $\{s_8, s_9\}$
- *SALU-Merge* (O_{Ctr2}): $\{\{s_1\}, \{s_2\}\}$, $\{s_7, \{s_8, s_9\}\}$
- *HFS-Reuse* (O_{Key}): $\{s_2, s_8, s_9\}$

Ensemble Type 3.

- *Hash-Reuse* (O_{Hash1}): $\{s_3, s_4\}$
- *Hash-XOR* (O_{Hash2}): none
- *SALU-Reuse* (O_{Ctr1}): $\{s_8, s_9\}$
- *SALU-Merge* (O_{Ctr2}): $\{\{s_3\}, \{s_4\}\}$, $\{\{s_6\}, \{s_7\}\}$
- *HFS-Reuse* (O_{Key}): $\{s_4, s_5\}$

Ensemble Type 4.

- *Hash-Reuse* (O_{Hash1}): none
- *Hash-XOR* (O_{Hash2}):$\{\{s_1\}, \{s_2\}, \{s_3\}\}$, $\{\{s_4\}, \{s_5\}, \{s_9\}\}$
- *SALU-Reuse* (O_{Ctr1}): none
- *SALU-Merge* (O_{Ctr2}): $\{\{s_7\}, \{s_8\}\}$
- *HFS-Reuse* (O_{Key}): none

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 B.3: 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 B.4: Ensemble Type 3. Same Epoch

SI	Base SA	Configurable Parameters			
		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 B.5: Ensemble Type 4. Random

Bibliography

[1] The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/data/passive/passive_dataset.xml. 21, 22, 37, 48, 51, 68, 88, 96

[2] The U.S. National CyberWatch Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC). https://www.netresec.com/?page=MACCDC. 96

[3] Marvell LiquidIO SmartNICs. https://www.marvell.com/products/ethernet-adapters-and-controllers.html. 44

[4] FCM-sketch source code. https://github.com/fcm-project/fcm_p4. 42

[5] Mellanox DPU. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. 100

[6] Netronome Agilio SmartNICs. https://www.netronome.com/products/nfe/. 44, 100

[7] Open Sourced P4All. https://github.com/mhogan26/P4All. 52

[8] tcpreplay. https://tcpreplay.appneta.com/wiki/tcpreplay-man.html. 87

[9] Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/. 4, 12, 18, 44, 85, 86

[10] Broadcom Trident 3. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series/, . 12, 105

[11] Broadcom Trident 4. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, . 85

[12] Barefoot P4 Studio. https://www.barefootnetworks.com/products/brief-p4-studio/. 87

[13] Xilinx FPGA. https://www.xilinx.com/products/silicon-devices/fpga.html. 100

[14] $P4_{14}$ Language Specification. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf, 2018. 22

[15] NPL Specifications . https://nplang.org/npl/specifications/, 2020. 105

[16] Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association. URL https://www.usenix.org/conference/nsdi22/presentation/hogan. 47, 50, 51, 52

[17] SketchLib: Enabling efficient sketch-based monitoring on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association. URL https://www.usenix.org/conference/nsdi22/presentation/namkung. 34, 37, 46, 47, 50, 51, 76, 107, 110

[18] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. {HeteroSketch}: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 719–741, 2022. 46, 106

[19] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 503–514, 2014. 1, 46

[20] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking*, 28(3):1172–1185, 2020. 46

[21] Eric Temple Bell. Exponential polynomials. *Annals of Mathematics*, pages 258–277, 1934. 61

[22] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C Luizelli, and Erez Waisbard. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 127–140, 2017. 2, 3, 11, 14, 16, 19, 20, 22, 27, 28, 31, 33, 34, 38, 47, 50, 51, 85, 89

[23] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 313–323. IEEE, 2018. 24, 27

[24] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 662–680, 2020. 14, 15, 100

[25] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010. 96

[26] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, pages 1–12, 2011. 1, 46

[27] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms*, pages 684–695. Springer, 2006. 2, 48, 66, 112

[28] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013. 1, 4, 5, 12, 44

[29] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 2014. 6, 19, 44, 87

[30] Vladimir Braverman and Rafail Ostrovsky. Zero-one frequency laws. In *Proc. of STOC*, 2010. 28, 29

[31] Valerio Bruschi, Ran Ben Basat, Zaoxing Liu, Gianni Antichi, Giuseppe Bianchi, and Michael Mitzenmacher. Discovering the heavy hitters with disaggregated sketches. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 536–537, 2020. 99

[32] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002. 2, 3, 11, 19, 20, 24, 27, 28, 34, 38, 48, 66, 85, 86, 87, 93, 96, 99, 112

[33] Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, and Jennifer Rexford. Measuring tcp round-trip time in the data plane. In *Proc. of SIGCOMM SPIN Workshop*, 2020. 100

[34] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 226–239, 2020. 27, 47, 49, 50, 51, 85

[35] B. Claise. Cisco systems NetFlow services export version 9. RFC 3954. URL `https://tools.ietf.org/html/rfc3954`. 2, 14, 15

[36] Graham Cormode and Marios Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal*, 19(1):3–20, 2010. 17

[37] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005. 2, 3, 11, 19, 27, 36, 46, 48, 66, 85, 86, 87, 93, 96, 99, 112

[38] Graham Cormode, Flip Korn, Shanmugavelayutham Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in data streams. In *Proceedings 2003 VLDB Conference*, pages 464–475. Elsevier, 2003. 11, 27, 31

[39] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738, 2020. 2

[40] Nick Duffield, Carsten Lund, and Mikkel Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 325–336, 2003. 1, 16

[41] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617. Springer, 2003. 11, 27, 110

[42] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 323–336, 2002. 1, 16

[43] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 153–166, 2003. 2, 4, 11, 19, 20, 27, 28, 30, 31, 33, 38, 48, 66, 85, 86, 87, 96, 112

[44] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293, 2000. 85, 86, 93

[45] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985. 19, 28

[46] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985. 2, 4, 11, 20, 27, 28, 30, 31, 33, 38, 48, 66, 112

[47] Philippe Flajolet, ric Fusy, Olivier Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, 2007. 2, 3, 11, 20, 27, 28, 30, 38, 46, 48, 66, 86, 87, 96, 99, 110, 112

[48] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 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*, pages 435–450, 2020. 47, 50

[49] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 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*, pages 44–61, 2020. 47, 50

[50] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1-2):18–28, 2009. 1, 46

[51] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research*, pages 61–74, 2017. 45, 100

[52] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 357–371, 2018. x, 1, 14, 15, 45, 48, 49, 100

[53] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 71–85, 2014. 14

[54] Hazar Harmouch and Felix Naumann. Cardinality estimation: An experimental survey. *Proceedings of the VLDB Endowment*, 11(4):499–512, 2017. 17

[55] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018. 45, 100

[56] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *Proc. of NSDI*, 2019. 100

[57] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126, 2017. 14, 45

[58] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590, 2018. 2, 3, 11, 14, 16, 24, 27, 32, 85, 86, 93

[59] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proc. of ACM SOSP*, 2017. 2, 23, 24, 25, 33

[60] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 103–115, 2015. 21, 44

[61] Georgios P Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostic, and Gerald Q Maguire Jr. What you need to know about (smart) network interface cards. In *PAM*, 2021. 100

[62] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *Proc. of USENIX NSDI*, 2019. 100

[63] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM Demo Session*, 2015. 100

[64] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: building a better bloom filter. In *European Symposium on Algorithms*, pages 456–467. Springer, 2006. 19, 28, 56

[65] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 234–247, 2003. 2, 3, 11, 19, 27, 48, 66, 86, 93, 99, 112

[66] Abhishek Kumar, Minho Sung, Jun Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):177–188, 2004. 2, 4, 11, 19, 20, 27, 28, 30, 31, 33, 38, 48, 66, 67, 112

[67] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. Data streaming algorithms for estimating entropy of network traffic. *ACM SIGMETRICS Performance Evaluation Review*, 34(1):145–156, 2006. 2, 11, 27, 48, 66, 112

[68] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say {NO} to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, 2016. 2

[69] Shangsen Li, Lailong Luo, Deke Guo, Qianzhen Zhang, and Pengtao Fu. A survey of sketches in traffic measurement: Design, optimization, application and implementation. *arXiv preprint arXiv:2012.07214*, 2020. 105

[70] Yifan Li, Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu. Cetus: Releasing p4 programmers from the chore of trial and error compiling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 371–385, 2022. 47, 50

[71] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016. 2, 3, 11, 14, 16, 18, 19, 20, 22, 24, 27, 28, 33, 34, 38, 46, 47, 48, 50, 51, 66, 85, 86, 87, 89, 96, 99, 112

[72] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *Proc. of USENIX FAST*, 2019. 24, 33

[73] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. 2019. 2, 14, 16, 19, 31, 45

[74] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Proc. of APoCS*. SIAM, 2020. 100

[75] Zaoxing Liu, Hun Namkung, Anup Agarwal, Antonis Manousis, Peter Steenkiste, Srinivasan Seshan, and Vyas Sekar. 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)*, pages 9–16. IEEE, 2021. 72

[76] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3829–3846, 2021. 49

[77] Antonis Manousis, Zhuo Cheng, Ran Ben Basat, Zaoxing Liu, and Vyas Sekar. Enabling efficient and general subpopulation analytics in multidimensional data streams. *arXiv preprint arXiv:2208.04927*, 2022. 105

[78] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005. 2

[79] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017. 1, 2, 5, 23, 46

[80] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015. 47, 50, 51

[81] Shanmugavelayutham Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005. 93

[82] Hun Namkung, Daehyeok Kim, Zaoxing Liu, Vyas Sekar, and Peter Steenkiste. Telemetry retrieval inaccuracy in programmable switches: Analysis and recommendations. In *Proceedings of the Symposium on SDN Research*, 2021. 65

[83] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017. 1, 14, 15, 45, 49

[84] George Nychis, Vyas Sekar, David G. Andersen, Hyong Kim, and Hui Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *ACM IMC, 2008*. 99

[85] George Nychis, Vyas Sekar, David G Andersen, Hyong Kim, and Hui Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 151–156, 2008. 86, 93

[86] George Nychis, Vyas Sekar, David G Andersen, Hyong Kim, and Hui Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 151–156, 2008. 99

[87] Anirudh Ramachandran, Srinivasan Seetharaman, Nick Feamster, and Vijay Vazirani. Fast monitoring of traffic subpopulations. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 257–270, 2008. 1, 16

[88] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017. 1, 2, 24, 27, 44, 46, 99

[89] Cha Hwan Song, Pravein Govindan Kannan, Bryan Kian Hsiang Low, and Mun Choon Chan. Fcm-sketch: generic network measurements with data plane support. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 78–92, 2020. 11, 14, 16, 27, 42

[90] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 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*, pages 329–343, 2020. 47, 50

[91] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 453–456, 2018. 45

[92] Lu Tang, Qun Huang, and Patrick PC Lee. Spreadsketch: Toward invertible and network-wide detection of superspreaders. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1608–1617. IEEE, 2020. 11, 27, 105

[93] Mikkel Thorup and Yin Zhang. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM Journal on Computing*, 41(2):293–331, 2012. 56

[94] Liang Wang, Hyojoon Kim, Prateek Mittal, and Jennifer Rexford. Programmable in-network obfuscation of dns traffic (work-in-progress). 100

[95] Mea Wang, Baochun Li, and Zongpeng Li. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *Proc. of IEEE ICDCS*, 2004. 14, 15

[96] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990. 2, 43, 48, 66, 112

[97] Qingjun Xiao, Zhiying Tang, and Shigang Chen. Universal online sketch for tracking heavy hitters and estimating moments of data streams. In *IEEE INFOCOM*, 2020. 19, 28, 31, 45

[98] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*, pages 25–33, 2019. 2

[99] Mingran Yang, Junbo Zhang, Akshay Gadre, Zaoxing Liu, Swarun Kumar, and Vyas Sekar. Joltik: enabling energy-efficient" future-proof" analytics on low-power wide-area networks. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020. 19, 31, 45

[100] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018. 2, 3, 14, 16, 27, 28, 42, 85, 99

[101] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dshark: a general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 207–220, 2019. 1, 14, 46

[102] Minlan Yu. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review*, 49(1):11–17, 2019. 1, 49, 72

[103] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *Proc. of USENIX NSDI*, 2013. 17, 19, 30, 99

[104] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of ACM SIGCOMM*, pages 126–138, 2020. 2

[105] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *Proceedings of NDSS*, 2020. 49

[106] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 207–222, 2021. 3, 14, 16, 46, 47, 49, 50, 51

[107] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *18th USENIX Symposium on Networked Systems Design and Implementation*, 2021. 99

[108] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. Flymon: enabling on-the-fly task reconfiguration for network measurement. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 486–502, 2022. 50, 51

[109] Peng Zheng, Theophilus Benson, and Chengchen Hu. 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*, pages 98–111, 2018. 47, 50

[110] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: intent-driven network traffic monitoring. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 295–308, 2020. 5, 14, 15, 49

[111] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015. 14