Designing Compact Data Structures for Network Measurement and Control

Xiaoqi Chen

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

Adviser: Jennifer Rexford

November 2023

Abstract

This dissertation explores the implementation of network measurement and closed-loop control in the data plane of high-speed programmable switches. After discussing the algorithmic constraints imposed by the switch pipeline architecture, primarily stemming from the requirement of high-speed processing, we share our experience in tailoring algorithms for the data plane. Initially, we focus on efficient measurement algorithms, and present two works for detecting heavy hitters and executing multiple distinct-count queries; both require designing novel approximate data structures to meet the tight memory access constraints. Subsequently, we pivot towards using real-time, closed-loop control in the data plane for performance optimization, and present two works for mitigating microbursts and enforcing fair bandwidth limits; both require approximated computation and exploit the sub-millisecond reaction latency unattainable through conventional control planes. We hope by sharing our experience and techniques, which are widely applicable to various algorithms and other data-plane hardware targets, we can lay the foundation for future innovations in the field of network programming for researchers and practitioners alike.

Contents

List of Figures

Acknowledgments

I can never express enough gratitude to my advisor, Prof. Jennifer Rexford, whose unwavering mentorship, guidance, support, and inspiration have shaped my journey over the past six years. Jen nurtured my growth from a curious novice to a confident researcher, and have always encouraged me to freely explore my interests even when they're sometimes too diverse. I am also deeply appreciative of the members of my thesis committee: Prof. Mark Braverman, Prof. Dave Walker, Prof. Maria Apostolaki and Prof. Minlan Yu. All your help and advise extended far beyond the scope of this dissertation, and I am truly grateful.

I want to express my heartfelt thanks for the help and feedback I've received from all my collaborators and everyone in our research group: Shir Landau Feibish, Ori Rottenstreich, Liang Wang, John Sonchack, Oliver Michel, Mina Tahmasbi Arashloo, Rob Harrison, Robert MacDavid, Mary Hogan, Yufei Zheng, Satadal Sengupta, Sophia Yoo, Mengying Pan; as well as other colleagues in the department, particularly Zhuqi Li and Zhenyu Song.

I also want to thank my classmates from IIIS; I began my research journey while soaked in the collaborative atmosphere of our undergraduate days. All of your exemplary achievements since then have served as my beacon, always motivating me to strive for excellence.

Finally, I'd like to sincerely thank my family for their steadfast support in all my endeavors. I'm also immensely thankful to my partner, who has stood by my side through all the highs and lows of my PhD journey.

The work in Chapter 3 has been supported in part by NSF Grant CCF-1535948. The work in Chapter 4 has been supported in part by NSF Grant CNS-1704077, the NSF Alan T. Waterman Award Grant No. 1933331, a Packard Fellowship in Science and Engineering, the Simons Collaboration on Algorithms and Geometry and The Eric and Wendy Schmidt Fund for Strategic Innovation. The work in Chapter 5 has been supported in part by NSF Grant CCF-1535948, a gift from AT&T, and The Eric and Wendy Schmidt Fund for Strategic Innovation. The work in Chapter 6 has been supported in part by DARPA grant HR00112oC0107. I also want to acknowledge the generous support by the Siebel Foundation for my PhD study. This document was typeset using a modified version of the LaTeX template made by Jordan Suchow, available at github.com/suchow/Dissertate.

1 Introduction

Since its inception five decades ago, the Internet has evolved from a niche research prototype into a technological marvel connecting billions of devices across the planet. Today, networks supports many components of our modern society, encompassing digital commerce, media, entertainment, education, and healthcare. The COVID-19 pandemic and the subsequent transition towards remote-first workplace and hybrid social interactions have further emphasized the importance of connectivity. Networks have, undoubtedly, become an essential component of our daily lives.

When users open an app on their smartphone or browse a website on their computer, they often take the fast, reliable, and omnipresent networks for granted. However, under the hood, serving user requests depends on the correct operation of a myriad of networks, ranging from the carrier access network nearby to the networks in data centers across the globe. Every day, network operators face many challenges in consistently providing high quality of service, for an increasingly diverse set of user applications with many different demands. For example, live streaming and VR/AR gaming requires very low and stable latency to avoid rebuffering or viewer discomfort[122]. Data center network operators strive for ultra-high reliability, as downtime leads to thousands or even millions of dollars of loss. Meanwhile, for many Internet-of-Things and smart home devices, privacy and security are the paramount requirement. In their day-to-day operations, network operators also need to cope with misconfigurations, equipment failures, and increasingly sophisticated attacks, which they must detect and react to quickly in order to maintain high reliability and minimize any disruption.

Therefore, operators need to perform *network measurement*, to improve network performance and reliability. One common measurement task is identifying *heavy hitter flows*[*]: network packets are often grouped into *flows* based on criteria such as sharing the same source and destination addresses, or belonging to the same user or application. Real-world network traffic exhibits skewness, with a small fraction of flows often consuming a significant portion of the network's bandwidth. By measuring these heavy hitter flows and possi-

[*]Also referred to as *elephant* flows.

bly re-route them, the network can use the bandwidth of different links more efficiently[20]. Moreover, measurement is also vital for maintaining network security; operators need to continuously monitor the network traffic to detect attacks and anomalies. As an example, a network need to look out for a source host sending packets to many different ports of the same destination host, which is a sign of a Port Scanning attack and needs to be reported or blocked.

Traditionally, network operators mostly rely on coarse-grained statistics and packet sampling[42,105] for measurement. However, as network speed grows exponentially, operators are using lower and lower sampling rate — as low as 1 in 30,000 to 1,000,000 — to avoid the prohibitive overhead in storing or processing the samples. The sparse samples and aggregated statistics are often insufficient for measuring transient anomalies that only affect a small fraction of all network packets.

Software-Defined Networking (SDN) is a revolutionary approach to building and managing computer networks. Traditionally, network devices like switches and routers have fixed-function, proprietary firmware, and each of them has to be configured separately. Different standards between vendors adds to the operator's efforts and chances for configuration mistakes. SDN, on the other hand, decouples the *data plane* that takes care of forwarding individual packets quickly, from a centralized *control plane* that decides how packets should be forwarded by each device. By requiring all devices to support a uniform standard (e.g., OpenFlow[85]), a SDN can run a single centralized control plane to configure the data plane of all devices. This makes it easier to manage large-scale networks with many heterogeneous devices and complex topology.

One of the latest development in SDN is the introduction of *programmable data plane*, which empowers network operators to deploy customized packet-processing algorithms directly in high-speed switches. In contrast to the earlier SDNs supported by fixed-function data planes and standardized control-plane APIs, which are upgraded in 2-3 year cycles, programmable data planes offer network operators increased agility. Network operators can now reprogram their network devices to support new protocols and implement new features, without waiting for switch vendors to design and ship new hardware or firmware, significantly accelerating the feature deployment cycles. Bosshart et al. proposed the Protocol Independent Switch Architecture (PISA)[23], a blueprint for switching chip design that adds reconfigurability to pipeline-based packet processing. In 2017, Barefoot Networks released the Tofino[9,65] switch based on PISA. Tofino is the first commercially-available high-speed programmable switch, supporting forwarding line rate up to 6.5 Terabits per second. Since then, operators like Alibaba and Facebook have been deploying Tofino switches in the field and exploit their programmability to implement novel network functions, such as load balancing[87], service gateway[97], and so on.

Programmable data planes present an unique opportunity for fine-grained network measurement. Measurement algorithms deployed in the data plane can efficiently handle every single packet at line rate and maintain statistics in the data-plane memory, avoiding the significant accuracy loss caused by sampling. However, the memory size available in the data plane is many magnitudes smaller than the enormous volume of network traffic being processed by switches. Fortunately, there exists an extensive body of theoretical works known as *Streaming Algorithms*[91], tailor-made for processing large data with small memory. We can take inspiration from these earlier works when designing data-plane algorithms.

Nonetheless, designing efficient data-plane algorithms for programmable switches is challenging. To achieve Terabits-per-second line rate and low forwarding latency, programmable switches impose strict memory and computational constraints on any algorithm we deploy. Therefore, we cannot simply reuse an arbitrary algorithm designed for running on general-purpose CPU, and must implement some adaptations for the switch architecture. The works in this dissertation are based on switches using PISA[23]; in Chapter 2, we will discuss some background about PISA and summarize the algorithmic constraints it imposes into a few easy-to-follow rules. It is important to note that the techniques we used to tailor algorithms to these constraints are also widely applicable to other devices processing high-speed network traffic.

The most salient constraint imposed by the programmable switch, from the perspective of designing network measurement algorithms, is the limited memory access. Namely, the data-plane algorithm can only update a very small number of memory addresses when processing a packet; it cannot arbitrarily access a large number of memory addresses or access memory addresses in an arbitrary order. If we take existing measurement algorithms designed to run on general-purpose CPU, they usually need to maintain sophisticated data structures in memory, using complex memory access patterns that cannot be implemented on the switch hardware. In particular, the Streaming Algorithms theoretical model assumed arbitrarily complex memory accesses to the small local memory are allowed. Therefore, an important step of designing efficient data-plane algorithm is to redesign the data structures to confront to the limited memory access, among other computational constraints imposed by the switch pipeline.

Once measurement results are available directly in the data plane, the switch can change its forwarding configuration within microseconds. This opens up new possibilities for real-time reactions and optimizations that were previously impossible through the control plane, as it takes at least 10-100 milliseconds for the control plane to collect statistics and install new configurations. For example, microbursts are small-timescale queue anomalies that lasts shorter than a few milliseconds, causing delay and possibly packet drops; only the data plane can react quick enough to prevent microbursts from affecting user experience.

The overarching objective of this dissertation is to examine the intricate challenges and offer adaptations and solutions for effectively utilizing data plane programmability, hoping to providing practical guidance for future researchers and practitioners alike. We observe that the techniques used to adapt algorithms to the switch's hardware constraints naturally fall into two categories: those involving the redesign of measurement to provide approximated results, and those using algorithmic changes to design a fuzzy, approximated control. Therefore, we divide the rest of the dissertation into two cohesive halves, centered around the key themes of (I) *measurement* and (II) *real-time, closed-loop control*.

In the first part of the dissertation, we emphasize designing efficient algorithms and data structures for network measurement to adapt to the constraints imposed by the switch pipeline. Here we showcase two works addressing two important classes of measurement queries, heavy hitters and count distinct, noting that the techniques presented can also be applied to adapting other data structures for other measurement queries.

- In Chapter 3, we present the PRECISION heavy hitter algorithm, which uses the *partial recirculation* technique to simplify the more complex memory access pattern used by prior works. PRECISION also uses approximations to replace exact calculations for hardware-friendliness without sacrificing accuracy.

- In Chapter 4, we introduce BeauCoup, a system that enables simultaneously executing many measurement queries while using a constant number of memory access per packet. This helps meet the operator's need to concurrently monitor an ever-expanding set of traffic anomalies, while respecting the switch hardware's memory access constraints. Notably, BeauCoup is the first system of its kind to allow dynamically updating traffic queries without reloading the data-plane program and causing downtime in forwarding.

The second part of this dissertation focuses on real-time, closed-loop control in the data plane. Using closed-loop control allows us to implement many novel features in high-speed networks; here we showcase two works, one optimizes network performance via queue and burst management, the other implements resource allocation and performance isolation via network slicing. We share our experiences in applying approximation techniques at every step of the closed-loop control algorithms, hoping to provide insights for other researchers in this emerging area when designing their own closed-loop control features.

- In Chapter 5, we present ConQuest, a framework for fine-grained, flow-level queue occupancy analysis. ConQuest empowers a switch to identify the root cause of small-timescale queue anomalies, commonly known as *microbursts*. Subsequently, the switch can react in real time to mitigate microbursts and protect other flows from suffering higher latency or packet drops.

- In Chapter 6, we introduce AHAB, an algorithm for approximately enforcing scalable hierarchical bandwidth fairness using programmable switch, avoiding the latency and jitter introduced by CPU-based software switch. Given a large number of users, it is infeasible to maintain per-user state and calculate a fair bandwidth allocation; instead, AHAB uses a novel memory-efficient data structure to estimate per-user sending rate, and employs closed-loop iterative update to quickly and accurately converge to fair allocation.

2

Background of PISA switches

In this chapter, we present an overview of the programmable switches using Protocol Independent Switch Architecture (PISA) and introduce the different type of architectural constraints they impose on algorithm designers.

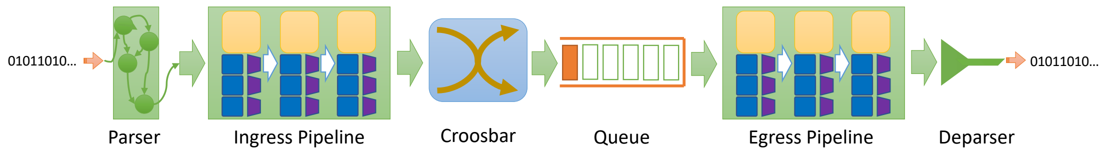


Figure 2.1: An illustration of the programmable data plane.

2.1 Switch hardware architecture

In order to handle packets using various protocols and support many different forwarding features, typical network switches include four components: the *parser*, the packet-processing *pipelines*, the *queuing* buffer, and the *deparser*. PISA switches also follow this design pattern, as illustrated in Figure 2.1, except these components are all programmable. In this Section, we present an overview on how each of these components work.

At first, in the parser, the packet (represented as bit streams from the wire) is parsed into semantic *fields* (or *headers*) and stored in the Packet Header Vector (PHV), alongside metadata such as the current timestamp and port number. The parser transitions between different states defined by the programmer, extracting some number of bytes and branching based on the extracted bytes. This allows the switch to parse packets with different protocol headers, both conventional (Ethernet, IP, TCP) and custom-defined ones. Here, we omit the technical details of the parser as it is mostly unrelated to this thesis, as measurement algorithms mostly do not use parsing/deparsing features beyond storing a few custom fields in the packet header. We refer interested readers to works that exploited the parser, such as Elmo[107], for a more in-depth discussion.

After the packet headers are parsed, the PHV is passed to the ingress pipeline. The remainder of the packet (the payload) is stored in the switch's buffer, and will only be rejoined with the header when the packet is finally departed into bits and transmitted on the wire.

To achieve Terabits-per-second line rate throughput, a switch must process a packet within 0.5-1 nanosecond. Given the highest clock rate of typical chips (1-5 gigahertz per second), the line rate translates to only a few clock cycles per packet. Since we can only finish very primitive operations within this tight time constraint, PISA uses a pipeline architecture with many distinct stages. When the switch runs at its fastest speed (at line rate with minimum sized packets), at any given time each stage is processing a different packet; all packets move to the next stage within every 0.5-1 nanoseconds. Each stage contains multiple primitive operations called *Match-Action Tables* (MAT); we can compose different MATs using multiple stages to implement complex packet-processing logic.

Each MAT will first *match* on a particular PHV field using different bit patterns. For example, in order to forward a packet to its destination, we will match on the packet's destination IP address. The program can specify the match rules in different ways: besides *exact* match, we can also use *ternary* match, which requires each bit to be "0", "1", or "*" (don't care). The latter is implemented on hardware using Ternary Content-Addressable Memory (TCAM); each ternary match rule has a priority, which helps tie-breaking when the same input bit pattern matched multiple rules. We also note that, for ease of expression, the ternary match functionality is represented by two other kinds of match logic. The longest-prefix match (*lpm*) is commonly used for routing, which represents prefix matching rules as ternary, with a more specific, longer-prefix rule having priority over a shorter-prefix rule. The range match (*range*) specifies lower and upper limits for a number, and use many prefixes of the binary representation to implement these limits.

Based on different matches, the MAT can then take different *actions* to modify the PHV. To achieve terabits-per-second throughput consistently, the pipeline needs to run at a fixed clock cycle $\geq$ 1 GHz and process one packet every several nanoseconds. This limits the complexity of pipeline stages, which can only implement some elementary actions. The most simple action is to write a value to PHV, to update a packet header field or custom-defined metadata variable. The MAT also supports a set of arithmetic operations, including addition, subtraction, and bit shifting. For example, to implement a router, we need to update `ip.ttl=ip.ttl-1` as an action. Furthermore, the MAT supports updating values stored in the *register memory* arrays available in each stage using a *register action*. The packet-processing program can first specify or calculate an index in the array, then perform a read-modify-write operation for the value in this array index. It is also possible to use values stored in PHV as operands of the read-modify-write, and write the result back into PHV. For example, to maintain statistics for the total traffic volume for each port, we can use the port number as array index; in the register action, we simply read the existing value in register memory, increment it by the size of the current packet (available as metadata in the PHV), and write the value back to register memory. In contrast to other actions, which are all stateless and always process the same packet the exact same way, *register action* allows us to implement stateful programs and is the key to network measurement.

It's important to note that between reading a value from the register memory and subsequently writing a new value, the data-plane program is constrained to executing very limited calculations that are supported by the pipeline's hardware circuit. This is because all these calculations must finish within the tight clock timing of a single pipeline stage. The extent of complexity allowable can vary based on the specific design of the switch hardware and the trade-off between expressiveness and chip area, as discussed by Domino [109].

Each pipeline stage supports running many MATs in parallel. After they simultaneously read from the PHV and performed their actions, the modified PHV is passed onto the next pipeline stage, while the current stage starts procesing the PHV of the next packet.

After going through all stages in the ingress pipeline, the packet is passed to the Traffic Manager, which first uses a crossbar to bring the bytes to the corresponding egress port (specified as metadata in PHV), then adds the packet to the queuing buffer to wait for egress processing. Alternatively, if the metadata indicated this packet should be dropped, the processing will stop here.

The egress pipeline consumes and processes packets from the queuing buffer. Note that egress pipeline has exactly the same architecture as the ingress pipeline; however, at this point, there are additional metadata (such as queuing delay) available, so some egerss-specific logic related to ports or congestion can be executed here. After finishing egress pipeline stages, the packet is deparsed into bits and transmitted.

2.2 Hardware-imposed algorithmic constraints

As we discussed in the previous section, in order to consistently achieve line-rate processing, the programmable switch hardware design only allows the data-plane program to run primitive actions in a pipeline fashion. In this section, we discuss the type of constraints relevant to algorithm designers.

2.2.1 Computational constraints

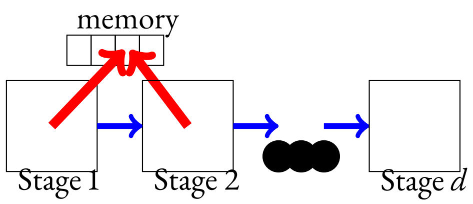
As we discussed earlier, given the tight timing requirement for implementing line-rate forwarding, the pipeline stages can only implement primitive actions. Thus, there is only a limited set of arithmetic operations available in an action, namely addition, subtraction, and shifting. We cannot calculate division or multiplication natively, nor can we perform floating-point operations, as these operations are generally much slower in a hardware circuit.

As a workaround, we can use match-action table rules to mimic low-precision floating point arithmetic, with division and multiplication results represented in lookup tables, as we will later discuss in Section 6.5. We can also use bit shifting to implement multiplication and division with integer powers of two.

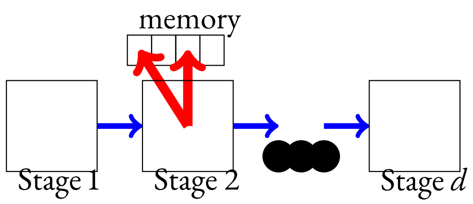
Also, to achieve low forwarding latency, the switch pipeline has a fixed, limited number of stages. While designing an algorithm, we need to avoid complex and long dependencies, so it can be completed within a small number of total pipeline stages.

2.2.2 Memory size

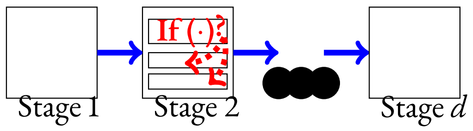
To achieve line rate of Terabits per second, the packet-processing pipeline must process one packet every 1-2 nanoseconds. Thus, the slower Dynamic RAM (DRAM) cannot meet the speed requirements, and the register memory arrays are implemented using Static RAM (SRAM), which has lower density and therefore smaller size compare to DRAM. For example, the first generation programmable switches, Intel Tofino series, offered 4-32 MB of register memory in the pipeline.



(a) Memory cannot be accessed from multiple stages



(b) Limited memory access within a stage



(c) Limited in-stage branching

Figure 2.2: Illustration of some constraints imposed by PISA switch pipeline for designing measurement algorithm.

This memory size is many orders of magnitude smaller compared to the huge volume of line-rate traffic we try to measure. Fortunately, in the theory literature, there is a class of algorithms called *streaming algorithms*, designed to have very small memory footprint while analyzing data streams. Naturally, the design principles of the streaming algorithms are a perfect fit for network measurement. However, existing streaming algorithms only considered limited memory size and assumed unlimited access to the memory. We have to adapt these algorithms to also consider the constraints on memory access patterns.

2.2.3 Memory access constraints

When designing data-plane algorithms, one of the most crucial challenges lies in ensuring the algorithm's memory access patterns follow the constraints dictated by the switch's pipeline architecture. These constraints can be distilled into three rules: 1) **partitioned memory**, 2) **limited concurrency**, and 3) **limited branching**.

Partitioned memory

The switch partitioned all its register memory between different pipeline stages. Namely, any particular register memory array can only be accessed from one pipeline stage; we cannot access the same memory from different stages. This is because at any given time, different stages are processing different packets. If a memory address is accessed by two different stages simultaneously, we cannot avoid a memory access hazard or pipeline stalling. Similarly, memory is partitioned between the ingress and egress pipelines. We illustrate this constraint in Figure 2.2a.

In the first-generation commodity programmable switch (the Intel Tofino), each pipeline stage has its own fixed amount of register memory. dRMT[38] proposed a newer architecture that allows a shared pool of memory to be more flexibly split between stages when running different data-plane programs; however, at runtime, memory is still partitioned between different stages.

Therefore, in normal cases, a packet being processed through the pipeline can only visit each memory array once. The switch also offers a feature called *packet recirculation*, which allows a packet to go through the pipeline a second time and access the same memory array twice. Recirculation also provides twice as many pipeline stages for complex computation operations. However, we note that every recirculated packet will also consume the pipeline's processing capacity, competing with incoming packets. When the fraction of recirculated packet exceeds the pipeline's reserved capacity, the switch cannot maintain line rate. We should therefore keep the fraction of recirculated packets to the minimum.

Limited concurrency.

In a pipeline stage, we can only access a small number of memory addresses within all the register memory attached to this stage. While processing a single packet, we cannot concurrently access many different indices in an array. This is because we cannot scan through all the memory within the clock cycle of a single stage. We illustrate this constraint in Figure 2.2b.

Limited branching.

Recall that while accessing register memory we can perform a read-modify-write operation. Since branching operations are expensive in circuits, the pipeline stage only supports very limited branching within a stage. Here, we cannot have complex conditions between reading and writing back to the same register memory address. We illustrate this constraint in Figure 2.2c.

Note that this constraint only applies to branching between reading and writing memory within a single stage. Between pipeline stages, we can still perform complex branching across stages using sophisticated matching rules in a MAT. The only limitation is we cannot write back to the same memory address if we use complex branching based on the value read from this particular address.

2.2.4 Summary

In summary, the programmable switch pipeline enforces computational, memory size, and memory access constraints. When designing data-plane algorithms, it is imperative that the computational operations align with the switch's capabilities. Moreover, we must ensure that the data structures utilized by these algorithms are efficient, optimizing memory size and, notably, employing straightforward memory access patterns that are compatible with the pipeline architecture.

It's worth noting that while the detail of constraints we discussed are specific to PISA switches, other high-speed networking devices will present their own computational and memory limitations following a similar pattern. This is because many of the underlying tradeoffs, particularly the limited memory size and memory access bandwidth relative to the increasing speed of network traffic, are inherent to all high-speed networking hardware. Consequently, algorithmic designs for the network data plane should prioritize the use of simple memory access patterns, as they are more likely to remain compatible with more networking hardware.

Part I

Enabling Network Measurement in the Switch Data Plane

3 PRECISION: Heavy-hitter detection via partial recirculation

In this chapter, we introduce PRECISION, an algorithm that uses *Partial Recirculation* to find heavy hitter flows on a programmable switch. As discussed in Chapter 2, the PISA switch pipeline imposed memory access constraints, making it challenging to implement prior heavy hitter algorithms. By recirculating a small fraction of packets, PRECISION simplifies the access to stateful memory to conform with the memory access constraints. We also evaluate each of the hardware-friendly adaptations made by PRECISION and analyze its effect on the measurement accuracy. Finally, we suggest two algorithms for the hierarchical heavy hitters detection problem in which the goal is identifying the subnets that send excessive traffic and are potentially malicious. To the best of our knowledge, PRECISION was the first algorithm to implement hierarchical heavy-hitter detection on PISA switches.

The work in this chapter was completed in collaboration with Ran Ben Basat, Gil Einziger, and Ori Rottenstreich. It was first presented in the IEEE International Conference on Network Protocols (ICNP) 2018[15] and later appeared in the IEEE/ACM Transactions on Networking[12].

3.1 Introduction

Identifying heavy hitter flows is an important task for network monitoring and management. For example, traffic engineering[20] may want to identify the largest (heavy hitter) flows, and forward them to use the most idle link. An intrusion detection system[50,90,93,104] may be interested in *hierarchical* heavy hitters[13,88], i.e., IP address blocks of various sizes that consume a lot of traffic.

Ideally, we can allocate some memory for every flow to store its measurement statistics, including its volume. However, this is infeasible given the large number of flows processed in high-speed switches and the limited data plane memory size. *Heavy hitter* algorithms only store flow state for the largest flows to overcome this limitation. This approach exposes a trade-off between memory space and accuracy, where additional space improves the accuracy.

There are two types of solutions for the heavy hitter detection problem — *counter-based* algorithms and *sketch-based* algorithms. Counter-based algorithms maintain a bounded-size flow cache. Only a small portion of the flows are measured, and each monitored flow has its own counter. Examples of counter-based algorithms include *Lossy Counting*[84], *Frequent*[67], *Space-Saving*[45,86], and *RAP*[11]. In sketch-based algorithms, counters are implicitly shared by many flows. Examples of sketch-based algorithms include *Multi Stage Filters*[51], *Count-Min Sketch*[48], *Count Sketch*[28], *Randomized Counter Sharing*[73], *Counter Tree*[29], and *UnivMon*[78].

Heavy hitter measurement has two closely related goals. In the *frequency estimation* problem, we wish to approximate the size of a flow whose ID is given at query time. Alternatively, in the *top-k* problem, the algorithm is required to list the *k* top flows. In general, sketch algorithms solve the frequency estimation problem but require additional efforts to address the top-*k* problem. For example, UnivMon[78] uses heaps alongside the sketches to track the top flows. FlowRadar[74] and Reversible Sketch[103] encode flow ID in the sketch, and have a small probability to fail to decode. In contrast, counter algorithms already store flow identifiers and can directly solve the top-*k* problem. While sketch algorithms are readily implementable in programmable switches, supporting top-*k* measurements is a strong motivation for deploying counter algorithms in such switches. Unfortunately, high-performance packet processing imposes severe restrictions on the programming model which makes implementing counter algorithms a daunting task.

Some applications such as attack mitigation and intrusion detection require something more sophisticated than (plain) heavy hitters[50,90,93,104]. For example, in a Distributed Denial of Service (DDoS) attack, a large number of devices collaborate to overwhelm an Internet service. In many cases, the source IP addresses of the attacking devices are different from the legitimate traffic. That is, the attack shares common prefixes which correspond to several sub-networks that do not deliver much legitimate traffic. *Hierarchical Heavy Hitters (HHH)* identify frequently appearing sub-networks. These can be used to either white-list traffic from the most frequent sub-networks of the legitimate traffic or to detect that traffic from specific sub-networks is likely due to an attack and blacklist them. Since programmable switches are powerful enough to cope with the current volume of DDoS attacks, performing HHH analysis on such switches offers exciting opportunities. Unfortunately, even (plain) heavy hitters are non-trivial to implement in programmable switches.

Contribution

We present *Partial RECirculation admisSION (PRECISION)* – a heavy hitter algorithm that is fully compatible with PISA programmable switches. We implemented PRECISION in the P4 language[22] on the Intel Tofino[65] programmable switch that achieves multiple Tbps of aggregated throughput, and deployed it in Princeton University's campus network to measure real-world heavy hitter flows. The core idea behind PRECISION is *Partial recirculation*; PRECISION recirculates a small portion of packets from unmonitored flows; we decide probabilistically or deterministically if the packet should be recirculated and pass again through the programmable switching pipeline. In the first pipeline pass, we try to match a packet to an existing flow entry; if this succeeds, we increment its counter. If unmatched, we sometimes recirculate it to claim an entry with the new packet's flow ID. Using the packet recirculation feature greatly simplifies the memory access pattern without significantly degrading throughput, while by carefully setting the recirculation portion we achieve high monitoring accuracy.

Previous suggestions include HashParallel and HashPipe[111], two counter-based heavy hitter detection algorithms proposed specifically for running on high-throughput programmable switches. They both maintain a *d*-stage flow table tailored to the pipeline architecture of programmable switches but differ in whether to recirculate an unmatched packet. HashPipe never recirculates packets and always inserts the new entry, which yields high throughput but lower accuracy. Instead, HashParallel recirculates **every** unmatched packet, which achieves much better accuracy but lowers the throughput. In contrast, PRECISION only recirculates a tiny portion of the unmatched packets with a minimal impact on performance. This approach allows PRECISION to conform to the switch pipeline's memory access constraints and also improves accuracy over HashPipe, especially for heavy-tailed workloads. We then analyze the impact of each constraint individually and find that most limitations have little effect in practice. We also show that HashPipe[111] cannot satisfy both the *limited branching* rule and the *single stage memory access* rule, requiring more sophisticated memory access not supported by the switch hardware.

Next, we suggest two methods to implement Hierarchical Heavy Hitters (HHH) detection on programmable switches. These, utilize PRECISION as a black box, and demonstrate the feasibility of HHH detection entirely in the data plane of a high performance switch. Such a capability is an important enabler for attack mitigation systems.

Finally, we evaluate PRECISION on real packet traces and show that it improves on the state-of-the-art for high-performance programmable switches (HashPipe) for the two variants of the heavy hitter problem. It is up to 1000 times more accurate than HashPipe for the frequency estimation problem and reduces the space required to correctly identify the top-128 flows by a factor of up to 32 times. When compared to general (software) heavy hitter algorithms, PRECISION often has similar accuracy compared to Space-Saving and RAP. Interestingly, approximating the desired recirculation probability appears very important, with a stage-efficient 2-approximate solution PRECISION requires at most four times as much memory as RAP. When we dedicate more hardware pipeline stages to achieve a better approximation, the performance gap between PRECISION and RAP diminished.

Outline

The chapter is structured as follows. We first introduce the reader to the heavy hitter detection problem in Section 3.2 and survey related work in Subsection 3.1.1. In Section 3.3, we discuss the implementation of PRECISION, specifically how we adapt to the limitations imposed by the PISA switch pipeline to achieve probabilistic recirculation. Here, we also discussed a deterministic variant of PRECISION. Section 3.4 shows two designs to extend PRECISION to perform the Hierarchical Heavy Hitters (HHH) measurement. In Section 3.5, we evaluate PRECISION, by first quantifying the impact of each adaptation on the accuracy, and then position it within the field by comparing it with other heavy-hitter detection algorithms. Finally, we conclude in Section 3.6.

3.1.1 Heavy Hitter Problem Definition

In this section, we present formal definitions for the heavy hitter measurement problem. Our work targets two common measurement forms, the *frequency estimation* problem and the *top-k* problem. For both, we refer to a quasi-infinite packet stream, where each packet is associated with a flow as explained below.

A flow refers to a particular subset of the packet stream that we choose to combine and analyze as a whole. For example, a flow may apply to a TCP connection or a UDP flow, in which case the five-tuple (source and destination IP, protocol, source and destination port) becomes the flow identifier. Alternatively, a flow may refer to just the source IP address, or just the destination IP and port pair. In any case, we assume that a flow identifier is available from some fields of the packet header, and that flows partition the stream such that each packet belongs to a single flow.

We denote the frequency of a network flow with ID s, or the total number of packets belonging to flow s, as f_s. For the *frequency estimation* problem, we use the OnArrival model[11], which requires an algorithm to estimate the flow frequency for each new packet it sees, and evaluates the estimation error upon each packet arrival. Formally, we reveal packets in a stream $(p_1, p_2, \ldots)$ one packet at a time, and on each packet arrival, with packet p_t belonging to some flow s. An algorithm Alg is required to provide an estimate $\widehat{f_s}$ for $f_s \triangleq |\{p_i \in s | 1 \leq i \leq t\}|$ — the number of packets belonging to flow s in $p_1, \ldots, p_t$.

The *top-k* identification problem is defined as follows: Given a stream $(p_1, p_2, \ldots)$ and a query parameter k, the algorithm outputs a set of flows containing as many of the k largest flows as possible.

3.2 Related Work

The Space-Saving algorithm

Space-Saving (SS)[86] is a heavy hitter algorithm designed for database applications and software implementations. Space-Saving maintains a fixed-size flow table, where each entry has a flow identifier and a counter. When a packet from an unmonitored flow arrives, the identifier of the minimal table entry is replaced with the new flow's identifier, and its counter is incremented. Space-Saving uses a sophisticated data structure named stream-summary which allows it to maintain the entries ordered according to counter values in constant time as long as all updates are of unit weight.

Space-Saving was designed for database workloads, which often exhibit a heavily concentrated access pattern, i.e. most of the traffic comes from a few heavy hitters. In contrast, networking traces are often heavy-tailed[11,64]. That is, a non-negligible percentage of the packets belong to tail flows or those other than heavy hitters. Unfortunately, Space-Saving works poorly on such workloads. For conciseness, we present only the Space Saving algorithm between all classical heavy hitter algorithms, as it is often considered to be the most accurate[44,45,83].

Optimization for heavy-tailed workloads

To deal with heavy-tailed workload, Filtered Space-Saving[64] attempts to filter out tail flows before inserting into flow table. It utilizes a bitmap alongside a Space-Saving instance. When a packet arrives, a hash function is used to map its flow ID into a bitmap entry. If the entry is zero, it merely sets the entry to one. Otherwise, we update the Space-Saving instance.

Maintaining additional data structures to filter tail flows may be wasteful. Therefore, *Randomized Admission Policy (RAP)*[11] suggests using randomization instead. When an unmonitored flow arrives, it is admitted only with a small probability. Thus, most tail flows are filtered while heavy hitters that appear many times are eventually admitted. Specifically, if the minimal entry has a counter value of c, RAP requires the competing flow to win a coin toss with a probability of $\frac{1}{c+1}$ to be added. The idea of RAP can be applied to the Space-Saving algorithm for software implementations. For hardware efficiency, the authors evaluate a limited associativity variant.

Unfortunately, the programming model of high-performance programmable switches is too restrictive to implement these algorithms directly. Specifically, Space-Saving evicts the minimal flow entry across all monitored flows, whereas the architecture of programmable switches does not permit finding (and replacing) the minimum element among all counters. Even for the limited associativity variant of RAP, it is still difficult to implement the randomize replacement after finding the approximate minimum value, due to same-stage memory access restriction.

High-performance switch algorithms

HashPipe[111] adapts Space-Saving to meet the design constraints of the P4[22] language and PISA[23] programmable switch architecture. The authors suggest partitioning the counters into d separate stages to fit the programmable switch pipeline. They use d hash functions that dictate which counter can accommodate each flow on each stage. They first propose a strawman solution, *HashParallel*, which makes each packet traverse all d stages while tracking the minimal value among the counters associated with its flow. If the flow is monitored, HashParallel increments its counter. If not, it recirculates the packet to replace the minimal entry among the d. The authors explain that HashParallel potentially recirculates all the packets, which halves the throughput.

Hence, they suggest HashPipe as a practical variant with no recirculation. In HashPipe, each packet's flow entry is always inserted in the first stage. They then find a rolling minimum — the evicted flow proceeds to the next stage where its counter is compared with the flow monitored there. The flow with the larger counter remains, while the smaller flow's entry is propagated further. Eventually, the smaller counter on the d^{th} stage is evicted. This allows HashPipe to avoid recirculation but introduces the problem of duplicates — some flows may occupy multiple counters, and small flows may still evict other flows.

FlowRadar[74] is another P4 measurement algorithm that follows a different design pattern. The main design difficulty to overcome is the lack of access to a fully associative hash table in programmable switches. While HashPipe and this work implement a fixed associativity table using multiple pipeline stages, FlowRadar potentially stores multiple flows within the same table entry. That is, upon hash collision the new flow identifier is XORed into the existing identifier. FlowRadar works best when the measurement is distributed, where multiple programmable switches can share their state to decode flow entries. Initially, FlowRadar recovers all flow entries that had no collision. Recovered flows are then recursively removed from the data structure, enabling for more flows to be recovered.

This approach is differentiated from our own as it attempts to perform an accurate measurement and therefore requires space which is proportional to the number of flows. In contrast, our approach provides an approximation of the flow sizes, and the required memory is independent of the number of flows. Also, FlowRadar requires multiple measurement devices each encoding a different subset of flows whereas our solution can also be implemented on a single device.

The more recently proposed CocoSketch[131] algorithm also inserts new flow ID probabilistically when the ID was not tracked by an existing counter, similar to PRECISION; CocoSketch also sets the takeover probability to be $1/(c+1)$ for an existing counter with value c. However, CocoSketch always increment the approximate minimum counter by one, even when the new flow is not inserted. The unconditional increment makes the CocoSketch algorithm easy to implement on the switch pipeline without using any recirculation. However, the extra increments also lead to worse memory efficiency and accuracy for heavy-tailed workloads.

Sampling

Instead of running algorithms in the data plane, one may also sample a fraction of packets and run sophisticated algorithms elsewhere[16]. This approach simplifies the hardware implementation but the problem migrates elsewhere. Namely, to process the samples in real time, we need additional computation and bandwidth overheads. Also, achieving high monitoring accuracy on smaller flows requires high sampling rate.

Hierarchical Heavy Hitters

MST is an HHH algorithm that utilizes an independent (plain) heavy hitter instances for each prefix length[88]. Once in a while, MST calculates the set of HHH prefixes from the heavy hitters of each prefix length. The RHHH algorithm[13] optimizes the performance of MST in software by updating a single random prefix. These algorithms are non-trivial to implement in programmable switches due to the limited programming model.

3.3 Design and Implementation of PRECISION

In this section, we present several hardware-friendly adaptations that address the architectural constraints imposed by the PISA switch hardware.

3.3.1 From fully associative to d-way associative memory access

Building on top of Space-Saving[86] and RAP[11], we first tackle the fact that a programmable switch cannot perform the fully-associative memory access to evict the minimum item. At any given pipeline stage, the algorithm can specify an index to access some location in the register array. The switch may allow accessing a small number of positions simultaneously but definitely cannot compute a global minimum across an entire register array.

We adopt the limited-associativity idea from HashParallel and HashPipe[111] to approximately evict a small element, by choosing the minimum across d randomly selected elements from d separate register arrays. With this relaxation, we can naturally spread the memory access across different hardware stages, and at each hardware stage, we only access one memory location. Specifically, we use d independent hash functions $h_1, \ldots, h_d$ to compute a different index for each stage, and at each stage, we access the $h_i(key)^{th}$ element of the i^{th} register array. Note that PRECISION performs d flow entry reads, but it does not consume exactly d hardware pipeline stages, as processing each read involves two branchings, and costs three hardware stages. We also discuss how to reduce the total number of hardware stages required in Section 3.3.6.

3.3.2 Simplified memory access

Implementation requirements of HashPipe

Although the design of HashPipe has already satisfied many restrictions imposed by PISA, its memory access pattern prevents us from implementing it in today's programmable switch hardware (that has a limited support for Paired atoms). The high-level idea of the HashPipe algorithm (see pseudocode in Algorithm 1) is to always evict the minimum out of d elements, by "carrying" a candidate eviction element through the pipeline. At each stage, we compare the counter read from register memory with that of the carried element. Then, the smaller of which is propagated further onward.

We now scrutinize the register memory access to different arrays of HashPipe, as highlighted in Algorithm 1. If we look at Line 14 and Line 23, they both access the register array *key* holding flow identifiers. The single stage memory access restriction requires that line 14 through line 23 would be placed within the same hardware pipeline stage.

However, the execution flow is branched in line 21 based on the values in another register array (*val*). Such branching violates the limited in-stage branching restriction. Referring to the model presented in Domino[109], to implement HashPipe, the simple *RAW*[*] action atoms at each stage are inadequate, and at least *Paired*[†] action atoms are required.

[*]The RAW action unit is capable of Reading an element from register memory, Add a value to it, and Write it back. See Domino[109].

[†]The Paired action unit is capable of reading two different elements from register memory, conditionally branch twice (two nested *if* s), perform addition or subtraction to the elements, and write two new values back. See Domino[109].

Algorithm 1: HashPipe[111] heavy hitter algorithm

```
1  $l_1 \leftarrow h_1(iKey)$                          ▷ Always insert in the first stage;
2  if $key_1[l_1] = iKey$ then
3      $val_1[l_1] \leftarrow val_1[l_1] + 1$;
4      end processing;
5  else if $l_1$ is an empty slot then
6      $(key_1[l_1], val_1[l_1]) \leftarrow (iKey, 1)$;
7      end processing;
8  else
9      $(cKey, cVal) \leftarrow (key_1[l_1], val_1[l_1])$;
10     $(key_1[l_1], val_1[l_1]) \leftarrow (iKey, 1)$;
11 for $i \leftarrow 2$ to $d$ do
12                                     ▷ Track a rolling minimum;
13     $l_i \leftarrow h_i(cKey)$;
14     if $key_i[l_i] = cKey$ then
15                                     ▷ Read $key_i$ ;
16         $val_i[l_i] \leftarrow val_i[l_i] + cVal$          ▷ R/W $val_i$ ;
17         end processing;
18     else if $[l_i]$ is an empty slot then
19         $(key_i[l_i], val_i[l_i]) \leftarrow (cKey, CVal)$          ▷ Write $key_i$, $val$ ;
20         end processing;
21     else if $val_i[l_i] < cVal$ then
22                       ▷ Condition on $val_i$; Violating branching constraints;
23         swap $(cKey, cVal) \Leftrightarrow (key_i[l_i], val_i[l_i])$          ▷ R/W $key_i$ ;
```

While the PISA[23] does not specifically define what features the action units need to support, Paired action atoms are more expensive to implement than RAW atoms and require 14x larger chip area than RAW atoms[109]. Therefore, today's programmable switches, the Tofino series, do not support Paired atoms. We strive to design our measurement algorithm to only require the simpler RAW atoms.

With only the simple RAW atoms, it is not possible to conditionally update a flow entry while simultaneously incrementing the corresponding counters. As long as we place flow identifier and counter in two separate register arrays, this seemingly innocuous set of operations has some inevitable in-stage branching: if we access flow identifiers first, we need to: (i) Read flow ID from flow entry array; (ii) If ID matched, increment counter; otherwise, compare the counter to the carried counter value; (iii) If the condition is satisfied, replace flow ID. This leads to a write to flow entry register memory conditioned on reading from another counter register memory. Therefore, two nested branching within the stage is inevitable.

Some may argue that we can cleverly rearrange the operations to mitigate the branching; however, even if we access the counter first, we still encounter the same restriction: (i) Read a counter from the counter register memory; (ii) Read flow ID; if ID not matched, check if the counter is smaller than the carrier counter to decide whether to replace the flow ID; (iii) Write the incremented counter value, if the ID matched. Again, the conditional write after reading another register forces two nested branching within a hardware pipeline stage (requiring Paired atom). Therefore, we cannot implement HashPipe on the first generation programmable switches available on the market.

PRECISION's solution

The implementation of PRECISION is even more challenging. We decide to replace an entry after knowing the minimum sampled counter value, but we only know this value after reaching the end of the pipeline, at which point it is too late to write to the register memory of earlier stages.

We resolve this difficulty using the recirculation feature on switches[6,118], that allows packets to traverse the pipeline again, removing all conditional branching for register access. When a packet leaves the last stage of the pipeline, instead of leaving the switch, we may choose to bring it to the beginning of the pipeline and go through all stages again. We can use metadata to distinguish between recirculated packets (which should be dropped) and regular packets that should be forwarded to their next hop.

Using recirculation allows more versatile packet processing at the cost of packet forwarding performance, as the recirculated packet will compete for resources with new incoming packets. However, we believe it's a necessary trade-off to satisfy the no-branching-within-stage constraint for high-performance programmable switches.

At the end of the pipeline, we ignore those packets already matched to flow entries and probabilistically recirculate the other packets using probability $\frac{1}{carry\_min+1}$, where $carry\_min$ is the value of the minimum sampled entry. The recirculated packet will evict and replace the minimum sampled entry. It will traverse the pipeline again to write its flow identifier into the corresponding register array when it arrives at the right pipeline stage, and also update the corresponding counter to a new value $carry\_min + 1$. In expectation, for every unmatched packet we increased the count for its flow by 1.

As a packet recirculates, it introduces a delay between the point in which we chose to admit it, and when it writes its flow ID on its second pipeline traversal. During this period other packets may increment the counter, an effect that will be overridden. Thus, the recirculation delay may have some impact on PRECISION's accuracy. The duration of such delay is architecture-specific and depends on both the queuing before entering the pipeline and the length of the pipeline. In Section 3.5.2, we evaluate its impact on PRECISION's accuracy and show that PRECISION is insensitive to such delay.

3.3.3 Efficient recirculation

We avoid packet reordering and minimize application-level performance impact by using the *clone-and-recirculate* primitive, which routes the original packet out of the switch as usual, and drops the cloned packet after it finishes the second pipeline traversal. This implies that in-flow packet order is preserved and that a packet can only be recirculated once.

Since recirculated packets compete for resources with incoming packets, we would like to minimize the number of recirculated packets. Fortunately, recirculation happens only for unmatched packets, with a probability of $\frac{1}{carry\_min+1}$, where *carry_min* is the minimal counter value the packet saw in all pipeline stages. Thus, recirculation becomes less frequent as the measurement progresses and the counters grow.

We can further bound the expected recirculation ratio at the beginning of the execution by initializing all counter registers to a non-zero minimum value. For example, if we initialize all counters to 100, we also set an upper bound 1% for the expected recirculation probability. Subsequently, because of concentration bound, the probability for having more than $(1+\varepsilon)\%$ recirculation becomes negligible. In Section 3.5.3 we show that adding an appropriate initial value has a negligible accuracy impact. We also note that in hardware switches, recirculating 1% of packets leads to at most 1% impact on throughput.

3.3.4 Approximating the recirculation probabilities

Recall that the original RAP algorithm admits packets from new flows with probability $\frac{1}{carry\_min+1}$. Intuitively, a flow needs to arrive $carry\_min + 1$ times on average to capture a counter with a value of $carry\_min + 1$.

It is straightforward to achieve this probability if a random arbitrary-range integer generator is available: we can generate an integer within $[0, \mathit{carry\_min}]$ and check if it's 0. However, we can only obtain random bits from programmable switch's hardware random source, and this effectively limits us to generate random integers within $[0, 2^x - 1]$ range. Without the capability to do division or multiplication, we cannot accurately sample with desired probability $\frac{1}{\mathit{carry\_min}+1}$.

The most simple approximation is to only use probabilities of the form 2^{-x}, which can be done by matching x random bits to zeroes. That is, we recirculate unmatched packets with probability $\frac{1}{\mathit{carry\_min}+1}$ rounded to the next smallest 2^{-x}. This is a 2-approximation of the desired recirculation probability. The recirculated packet will update the counter to 2^x. Rounding is achieved by using a ternary matching over bits of *carry_min* variable to find the highest 1 bit. The evaluation in Section 3.5.4 shows that this method has a noticeable but acceptable impact on accuracy.

We now introduce a tighter method for approximating the desired recirculation probability. Inspired by floating point arithmetic, we may decompose $\mathit{carry\_min} + 1 = 2^y \times T, T \in [8, 16)$ and use a probability of the form $\frac{1}{2^y} \times \frac{1}{\lceil T \rceil}$ to approximate $\frac{1}{\mathit{carry\_min}+1}$. We can directly implement the $\frac{1}{2^y}$, while the $\frac{1}{\lceil T \rceil}$ is approximated by randomly generating an integer between $[0, 2^N]$ and comparing it against a pre-computed constant $\lfloor \frac{2^N}{\lceil T \rceil} \rfloor$, via a lookup table. Further, to avoid non-integer number representation, we always increment the counter value by 1 upon recirculation. This achieves a $9/8$-approximation of the desired recirculation probability. In Section 3.5.4, we show that the accuracy gains are significant. Yet, this method requires an additional pipeline stage.

Algorithm 2: PRECISION heavy hitter algorithm

```
1  for $i \leftarrow 1$ to $d$ do
2      $l_i \leftarrow h_i(iKey)$ ;
3      if $key_i[l_i] = iKey$ then
4          ▷ Hardware stage $i_A$: access $key_i$ register;
5          $matched_i \leftarrow true$;
6      if $matched_i$ then
7          $val_i[l_i] \leftarrow val_i[l_i] + 1$    ▷ Hardware stage $i_B$: access $val_i$ register ;
8      else
9          $oval_i = val_i[l_i]$
10     if $(\neg matched_i) \wedge (oval_i < carry\_min)$ then
11         ▷ Hardware stage $i_C$: maintain carry minimum;
12         $carry\_min \leftarrow oval_i$;
13         $min\_stage \leftarrow i$
14 if $\bigwedge_{i=1}^{d}(\neg matched_i)$ then
15     ▷ $iKey$ not in cache; do Probabilistic Recirculation. $new\_val = 2^{\lceil \log_2(carry\_min) \rceil}$;
16     Generate random integer $R \in [0, new\_val - 1]$, by assembling $\lceil \log_2(carry\_min) \rceil$
       random bits;
17     if $R = 0$ then
18         clone and recirculate packet;
19 if packet is recirculated then
20     $i \leftarrow min\_stage$;
21     $l_i \leftarrow h_i(iKey)$;
22     $key_i[l_i] \leftarrow iKey$    ▷ Hardware stage $i_A$: access $key_i$ register ;
23     $val_i[l_i] \leftarrow new\_val$    ▷ Hardware stage $i_B$: access $val_i$ register;
24     Drop the cloned copy;
```

3.3.5 Putting all adaptations together

With all the aforementioned hardware-friendly adaptations in mind, we assemble the PRECISION algorithm, which satisfies all hardware-imposed constraints of PISA switches. Algorithm 2 is a pseudocode version of PRECISION. Line 1 reflects PRECISION's d-way associative memory access, iterating through each way. In Line 7 we increment the counter for matched packets, while unmatched packets handled between Line 15 and Line 19. We flip a coin in Line 17, and the 2-approximation of recirculation probability manifests in Line 16. Recirculated packets update register memory corresponding to their minimal entries. This is described between Line 20 to Line 24. We highlighted accesses to register memory in color, note that registers are only accessed once per stage. Each branching fits in a transition between hardware pipeline stages, removing the need to perform in-stage branching.

3.3.6 Parallelizing actions to reduce hardware stages used

Algorithm 2 presented PRECISION in its most straightforward arrangement, iterating through the *d*-way in tandem, while each uses three pipeline stages. This costs as much as $d \times 3$ hardware pipeline stages for register memory reads. Since the total number of pipeline stages is very limited, we explain how to optimize the required number of stages further, and fit a larger *d* on the same hardware. This optimization may also be applicable to other algorithms with a similar repeated register array access pattern.

Intuitively, each 'if' in the pseudocode is a branching, separating the algorithm into different hardware stages. However, it may be possible to group independent stages and reduce the total number of hardware stages needed.

In our implementation, PRECISION requires two branching for each of the *d* ways. That is, it requires three pipeline stages for each way. The stages in each way are:

- **Stage A:** Read flow ID from flow entry array.
  - **Branching:** does entry's ID match my ID?
- **Stage B:** Read/Update from the counter array.

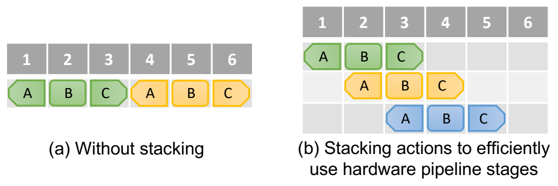

1
2
3
4
5
6
A
B
C
(a) Without stacking
(b) Stacking actions to efficiently use hardware pipeline stages

Figure 3.1: We reduce the number of pipeline stages used by stacking together independent actions between different ways. For d-way PRECISION, this reduces the number of pipeline stages required from $d \times 3$ to $d + 2$.

   - **Branching:** is counter smaller than the current minimum?

- **Stage C:** Compute and "carry" the new minimum value.

If we indeed require three hardware stages for each pair of flow entry array and counter array, a switch with X physical stages can at most implement PRECISION with $d = X/3$. This assumes that all pipeline stages serve for heavy-hitter detection. In practice, we would like to leave enough pipeline stages for other network applications.

However, our algorithm does not have a hard dependency between different groups of stages. If we denote the d ways as 1, 2, 3 and the three pipeline stages for each action as A, B, and C, we can observe that (for example) 2_A and 1_C are independent. Thus, it's not necessary to serialize everything into the pattern shown in Figure 2(a). Instead, we can "stack" operations from different groups together, as shown in Figure 2(b). Specifically, reading the flow identifier for the next flow entry array can be parallelized with incrementing a counter for the previous way's counter array and so forth. Therefore, we can parallelize different execution stages of multiple ways as there is no direct causal relation or data dependency between stage action $(i+1)_B$ and i_C, or between $(i+1)_A$ and i_B. Thus, by using the stacking pattern shown in Figure 2(b), we reduce the number of required stages to implement d-way PRECISION from $d \times 3$ to $d + 2$, amortizing to one stage per way. [‡]

For a programmable switch with a limit of X hardware stages, the actual maximum d we can implement will be smaller, because we need extra stages before and after the core algorithm for setup and teardown, such as extracting flow ID and performing random coin-tossing. Furthermore, a network switch will need to fulfill its regular duties like routing, access control, etc., and would not devote all its resources to the PRECISION algorithm. Nevertheless, we can expect any commodity programmable switch to run the $d = 2$ version of PRECISION smoothly, alongside its regular duties. When extra resources are available, we may increase d to improve accuracy as shown in Section 3.5.1.

3.3.7 A deterministic PRECISION variant

In this section, we consider a variant that replaces the probabilistic recirculation mechanism of PRECISION by a deterministic one. Intuitively, instead of admitting each packet with probability p we can admit the $1/p$'th packet. To implement this, we change the hashing scheme of PRECISION so that each flow is only hashed once (i.e., all l_i are the same, see Line 2 in Algorithm 2). We add a single counter per row that is initialized to zero and incremented for every packet that is mapped to this row. Then, if the counter equals the minimal value observed by the packet, we recirculate the packet and reset the counter.

While this approach is not suitable for adversarial traffic (where a flow can deliberately avoid being admitted), we show that on standard traffic workloads it outperforms the ran-

[‡]There is indeed a causal dependency between stage $(i + 1)_C$ and i_C when computing the carried minimum value *carry_min*, thus using only a constant number of 3 hardware stages is not possible. Also, other hardware constraints that limit the number of parallel actions in one hardware stage exists, but these are less stringent than the limit on the total number of hardware stages.

domized version (see Section 3.5.7). Such a deterministic approach has the following implementation benefits: (i) it does not need to approximate the sampling probability (e.g., we can add the 100'th packet and not approximate $1/100$ which leads to biases). (ii) it does not require random bits generation, or a lookup table and (iii) its results are easily reproducible given the packet trace. As a potential implementation drawback of this approach, the added counter requires slightly more memory than the randomized variant. However, this seems to be rather low overhead and the improved accuracy allows the deterministic variant to use fewer counters for the same performance.

3.4 Hierarchical Heavy Hitters

In this section, we suggest two implementations of Hierarchical Heavy Hitters (HHH) on programmable switches. HHH is a generalization of the heavy hitter / top-k problems in which the goal is to identify *subnets* that send an excessive amount of traffic. HHH is motivated by the need to identify the attackers in a distributed denial of service (DDoS) attacks. Intuitively, the attack has access to many devices, each of which only sends a moderate amount of traffic, eliminating detection by standard top-k solutions. If the malicious devices share a common subnet (or a small number of subnets), HHH algorithms are able to identify them by considering the aggregated traffic that originates from each network. The full definition of the HHH problem is complex and appears in various literature[46,47].

Here, we present two possible solutions for finding HHH with PRECISION with different accuracy-resources tradeoffs. The *Independent Stacking* suggestion is based on the MST algorithm[88] and requires running multiple instances of PRECISION in parallel. Each such instance is updated in parallel with one of the prefixes of the current packet

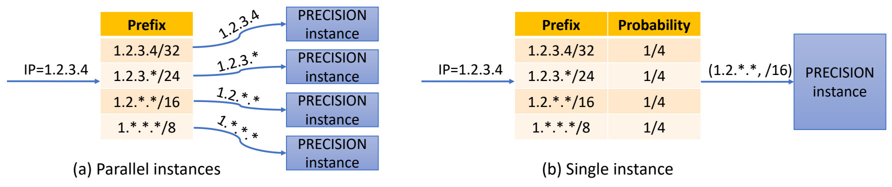


Figure 3.2: An illustration of our two implementation approaches for Hierarchical Heavy Hitters.

as illustrated in Figure 3.2(a). For example, in the common use case of source hierarchies in byte granularity, we are required to monitor (i) the total number of packets (i.e., /0 sub-network), (ii) the number of packets from each sub-network of size 8 bits, (iii) from each sub-networks of size 16 bits, (iv) from each sub-network of size 24 bits, and (v) sub-networks of size 32 bits. Clearly (i), and (ii) can be accurately counted using one and 256 counters, which requires three parallel instances of PRECISION for (iii)-(v). Such a suggestion is efficient in a pipeline architecture, as the independent PRECISION instances can be stacked together without requiring additional stages. However, Independent Stacking does not scale very well for larger hierarchies. Further, some architectures may limit the amount of stacking. Thus, our second suggestion is focused on implementing HHH using fewer hardware resources.

The basic idea of our second suggestion is to use RHHH as a model. We use a single PRECISION instance to monitor all prefix lengths, as illustrated in Figure 3.2(b). We add a pipeline stage that counts the total number of packets and then pick a prefix uniformly at random. The number for possible prefix lengths is 4 (for byte-level) or 32 (for bit-level), which are both power of 2 and easy to sample from. We update the single PRECISION instance in the same way as PRECISION does. The controller then receives the heavy hitters of the unified instance, separate them by prefix types and calculate the HHH list in the same manner as the previous work[13]. The work of RHHH showed that randomization works upon having a large number of packets. This is a reasonable assumption to make under our setup with high-throughput programmable switches. Further, the second suggestion requires only a single instance of PRECISION (with more allocated memory), therefore is applicable whenever PRECISION can work.

3.5 Prototype and Evaluation

This section presents an evaluation of PRECISION's accuracy and adaptation mechanisms. We implement PRECISION in 800 lines of P4[22] code on an Intel Tofino[65] Wedge-100 programmable switch; the prototype implementation is available on GitHub[32]. It supports d=2 stages each tracking 64k flows, saving a total of 128k heavy hitter flows (defined as source-destination IP pairs). The implementation used 15% of header metadata memory and 20% of total register memory available to save flow IDs and counters. It computed d=2 hash functions, less than 10% of totally available. Our PRECISION prototype was deployed in Princeton University's campus network to report heavy flows to network operators, and has correctly reported the flows with empirically largest volume. The prototype processes mirrored traffic from campus Internet border, and implements recirculation using ingress pipeline resubmit.

We also run PRECISION on a Python-based simulator, as simulation allows us to choose parameters freely and independently manipulate each hardware restriction. We start by studying the effect of each hardware-friendly adaptation on PRECISION's accuracy. Next, we compare PRECISION to related work, including HashPipe[111], as well as Space-Saving[86] and RAP[11] that are not directly implementable on programmable switches. We obtain the code of HashPipe from its authors, and run it on a Java-based simulator.

For evaluating frequency estimation, we measure the OnArrival *Mean Square Error (MSE)* of the algorithm, i.e.,

$$MSE(Alg) \triangleq \frac{1}{N}\sum_{t=1}^{N}(\widehat{f_s} - f_s)^2. \tag{3.1}$$

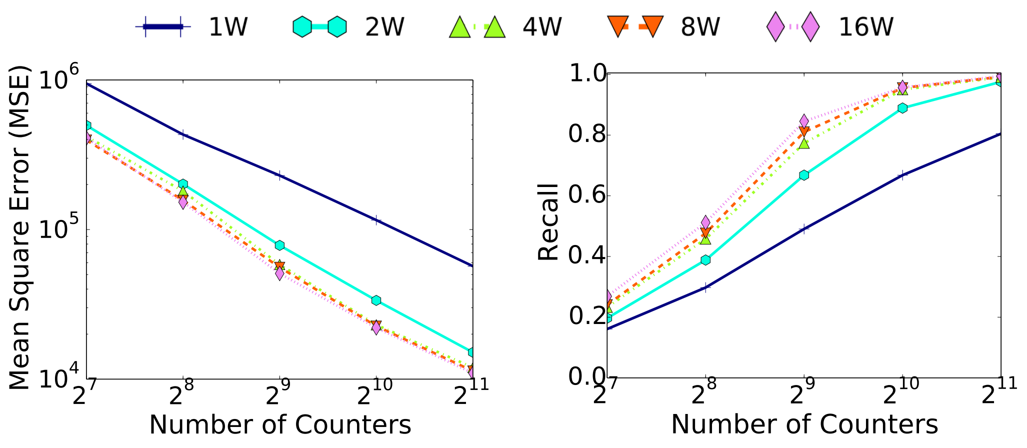
We judge the quality of the top-k based on the standard *Recall* metric that measures how many top flows it identifies. Specifically, denoting the k^{th} largest flow's frequency by F_k, when the algorithm outputs a flow set $\mathbf{C}$, quality using:

$$\text{Recall}(\mathbf{C}) \triangleq \ |e \in \mathbf{C} : f_e \geq F_k|/k. \tag{3.2}$$

Our evaluation utilizes the following datasets:

- **CAIDA**: The CAIDA Anonymized Internet Trace 2016[25] (in short, *CAIDA*). Data is collected from the Equinix-Chicago backbone link with a mix of UDP, TCP, and ICMP packets.
- **UWISC-DC**: A data center measurement trace recorded at the University of Wisconsin[19].
- **UCLA**: The University of California, Los Angeles Computer Science department packet trace[70].

We truncate each trace to its first 2 million packets, and use packets' Source-Destination IP address pair as their flow ID. In general, the CAIDA trace is heavy tailed, while the UWISC-DC trace and the UCLA traces are skewed.



(a) Frequency Estimation **(b)** Top-128

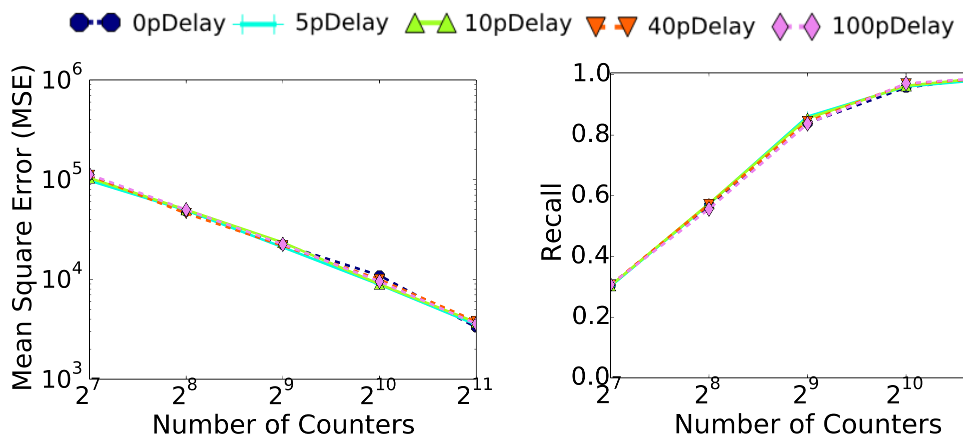
Figure 3.3: Effect of limited associativity on the frequency estimation error and top-k recall, on CAIDA trace. Using $d = 2$-way is a right balance between achieving good accuracy and saving pipeline stages usage.

We also tested our algorithm using synthetic trace with Zipf distribution and observed similar results.

All experiments were performed using a software emulated version of PRECISION, and we repeated each experiment 10 times with different random hash functions. Unless specified otherwise, the default associativity for PRECISION is 2-way.

3.5.1 Limited associativity

We start with the frequency estimation problem and measure OnArrival Mean Square Error (MSE). In this measurement, we evaluate PRECISION with a varying number of ways (d) and use the same amount of total memory for all trials. Our results in Figure 3.3a show that for this problem 1-way associativity ($d = 1$, abbreviated as 1W) is a bit too low, but 2-way is already reasonable and further increasing d has diminishing returns. Figure 3.3b evaluates how d affects the Recall in top-k problem, using 512 counters to find top-128



(a) Frequency Estimation

(b) Top-128

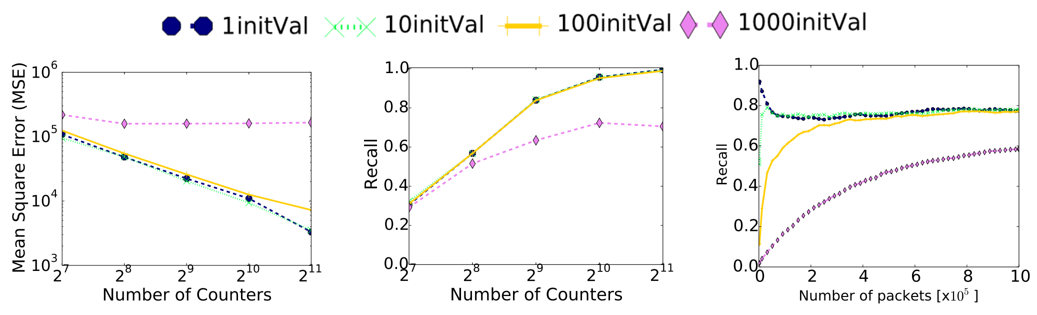
Figure 3.4: Effect of the delayed update on the frequency estimation error and top-*k* Recall, on CAIDA trace. Even a delay of 100 packets has minimal impact on the accuracy.

flows. In this metric, we see that associativity is more important than in frequency estimation. $d = 2$ requires up to 2× more counters than $d = 16$ to achieve the same recall. Changing to smaller or larger k yields similar observation.

We conclude that limited associativity incurs minimal accuracy loss in frequency estimation and is more noticeable in top-k. Our suggestion is to use $d = 2$ as it achieves the right balance between accuracy and the number of pipeline stages.

3.5.2 Entry update delay

We now evaluate the impact of update delay between the decision to recirculate and the actual flow entry update. Instead of using empirical evidence on one particular programmable switch, we simulate various possible delay values in terms of pipeline length. Figure 3.4a shows results for the MSE (Mean Square Error) in the frequency estimation problem and Figure 3.4b shows the Recall in top-k problem when trying to find the top-128 flows. As



(a) Frequency Estimation

(b) Top-128

(c) Convergence of Top-128 Recall over time

Figure 3.5: Effect of initial value on the overall frequency estimation error and top-k recall, on CAIDA trace. An initial value of 100 leads to fast convergence and does not hurt accuracy, while upper-bounding recirculation to 1%.

can be observed, the lines are almost indistinguishable. That is, update delay has a minor impact on accuracy for both metrics, even for a delay of 100 packets. We assume that practical switching pipelines would have shorter recirculation delays, as today's programmable switches have much less than 100 stages. A possible reason for this insensitivity to update delays is that replacing flow entries is already a rare and random event. Thus, the actual replacement time barely affects the accuracy even if it slightly deviates from the decision time.

3.5.3 Initial value

We now evaluate the impact of having an initial value larger than zero set to all counters. Intuitively, the initial value limits the number of recirculated packets, but also requires some time to converge. This is because having a non-zero initial value means that we need to see more unmatched packets before we claim an entry — even if that entry is empty. Figure 3.5a show results for the frequency estimation metric. As can be observed, the initial value does affect the accuracy, and the effect is small until initial value 100, but initial value 1,000 causes a large impact. A similar picture can be observed in Figure 3.5b that evaluates

Recall in the top-128 problem using 512 counters. As depicted, initial value also has a little impact up to 100, but an initial value of 1,000 results in a poor Recall.

Figure 3.5c completes the picture by showing the change of the Recall over time when trying to find top-128. As shown, the convergence time is inversely correlated with the initial value. In most cases, 1 million packets are enough for converging with an initial value of 100. We observed similar behavior for different packet traces. It appears that an initial value of 1,000 requires more packets to converge.

We conclude that a small initial value has a limited impact on the performance when the measurement is long enough. To facilitate quick convergence, we suggest an initial value of 100 (and use it in the following experiments), as it seems reasonable to upper bound recirculation to at most 1% of the packets, and the convergence time is shorter than 1 million packets, which translates to less than 10 milliseconds on fully-loaded 100 Gbps links.

3.5.4 Approximating the desired recirculation probability

We now evaluate the impact of only using random bits as random source. This limits us to approximate the ideal recirculation probability $\frac{1}{carry\_min+1}$ with a probability of the form 2^{-x} or $2^{-y} \times \frac{1}{\lfloor T \rceil}$. Figure 3.6 shows results for frequency estimation problem (a) and (b), and for the top-k problem (c) and (d). We evaluated four variants: "NoAdaptation" is the algorithm without any hardware-friendly adaptation beyond limited associativity; "2-Approximate" is the variant added with an approximate recirculation probability of 2^{-x} form; "PRECISION (2-Approximate)" is the standard PRECISION algorithm with all other hardware-friendly adaptations also added; and "9/8-Approximate PRECISION" is

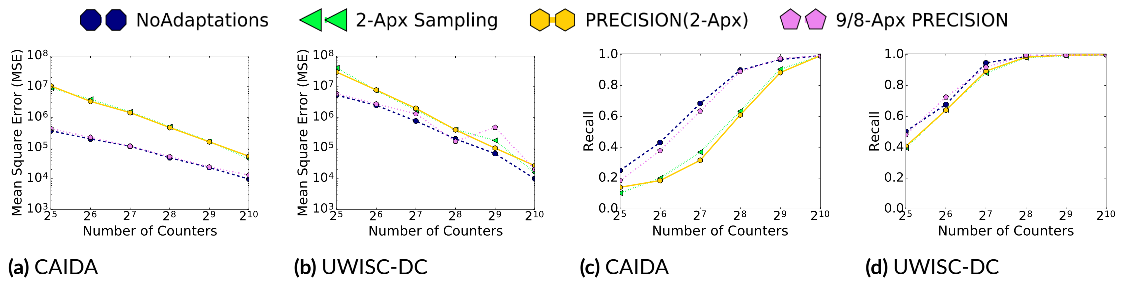


Figure 3.6: The effect of approximating the recirculation probabilities on the accuracy for frequency estimation and top-32.

the PRECISION algorithm using the $2^{-y} \times \frac{1}{\lfloor T \rceil}$ form of approximate recirculation probability.

For frequency estimation, the 2-approximation in recirculation probability increases the error noticeably (in both workloads) possibly due to counters are always bumped to the next power of 2 when replacing a flow entry, causing some overestimation. Meanwhile, using the 9/8-approximation is almost as accurate as having no restriction on the recirculation probability.

For the top-k problem, we continue with our ongoing evaluation of how many counters are needed to identify the top-32 flows. Notice that recirculation probabilities are less impactful in this metric and in both workloads we need $\approx 2\times$ as many counters as NoAdaptation to achieve the same Recall.

It is surprising at first to notice that approximating the recirculation probability has a minimal performance impact in the UWISC-DC trace for the top-k metric. The reason is the highly-concentrated nature of this trace. In such workload where heavy hitters dominate, the sizes of tail flows are too small compared with the large counters maintained for heavy hitters, thus the tail flows have little chance to evict heavy hitters regardless of how we approximate probability.

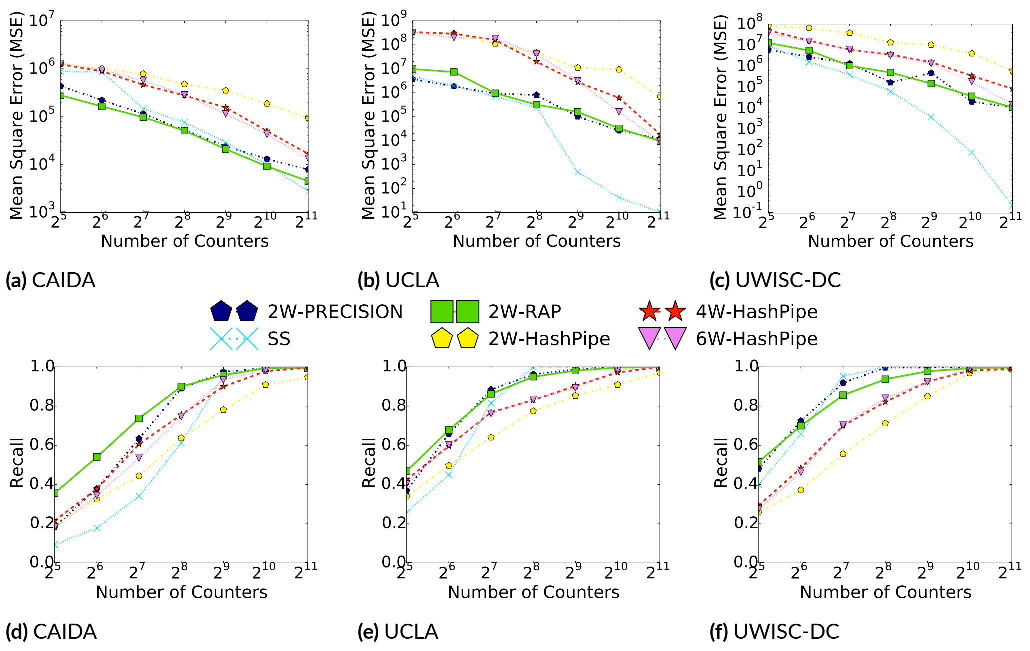


Figure 3.7: Comparative evaluation of the frequency estimation and top-32 problems.

3.5.5 Comparison with other algorithms

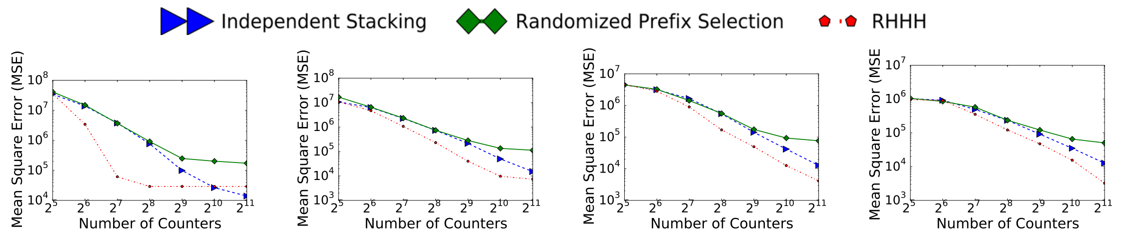
Next, we evaluate PRECISION with $d = 2$ and compare it with Space-Saving[86], and HashPipe[111] with $d = 2, 4, 6$ associativity. Similarly, we also compare with a 2-way set associative RAP[11]. Note that RAP was originally designed with a less restrictive programming model, and PRECISION adapts it to PISA switches.

Figure 3.7 shows results for the frequency estimation and top-*k* problems on the CAIDA (a), UCLA (b), and UWISC-DC (c) traces. Figures 3.7(a)-(c) shows that, for the frequency estimation problem, 2-way RAP and Space-Saving are the most accurate algorithm. They are followed by (2W-)PRECISION, which is orders of magnitude more accurate than 2W-HashPipe. PRECISION also has better performance than 4W- and 6W-HashPipe. We note that PRECISION also improves using higher associativity, as shown in Figure 3.3. Thus, we conclude the frequency estimation evaluation by saying that PRECISION is a dramatic improvement over HashPipe and is not much worse than the state-of-the-art algorithms despite its restricted programming model.

Figures 3.7(d)-(f) show the Recall performance for the top-*k* problem. In our top-32 setup, we see similar trends in all the traces, in which the best Recall is achieved by the 2-way RAP algorithm followed by PRECISION and Space-Saving. The algorithm with the lowest Recall is HashPipe, especially for d=2-way. We see that PRECISION is on par with Space-Saving and not far behind 2-way RAP. PRECISION yields similar performance in all traces and requires at most 2× more space than RAP or Space-Saving. Compared to 2W-HashPipe it requires up to 8× less space for the same Recall. PRECISION also improves over 4W- and 6W-HashPipe by up to an 4× factor.

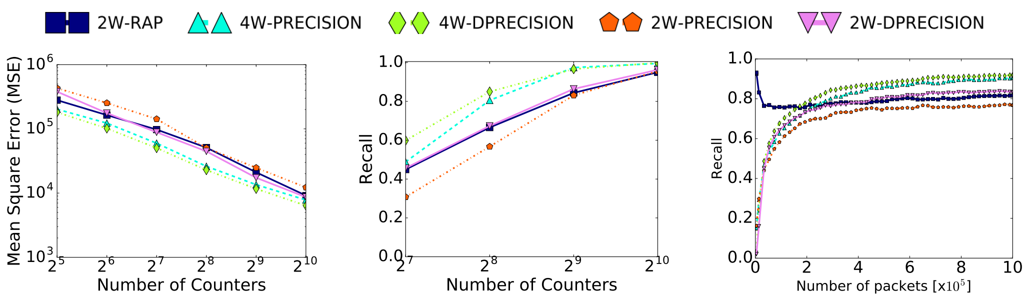
3.5.6 Hierarchical Heavy Hitters

Next we show results for our Independent Stacking, and Randomized Prefix Selection algorithms. Recall that Independent Stacking implements the MST algorithm[88] where each Space Saving instance is replaced by PRECISION. Thus, as PRECISION's accuracy is similar to that of Space Saving we use it as a baseline. That is, the accuracy of Randomized Prefix Selection can only be as good as the baseline. In Figure 3.8 we show results for these options, where each PRECISION instance is configured with the default parameters (4-way, initial value of 100, delay of 10 packets, 9/8-approximation of the sampling probability). The figure shows the obtained accuracy for different prefix lengths when varying the number of counters. As can be observed both algorithms obtain similar accuracy and



(a) net/8 **(b)** net/16 **(c)** net/24 **(d)** net/32

Figure 3.8: The error of the "Independent Stacking" and "Randomized Prefix Selection" implementations of HHH-PRECISION for a given number of counters. We also compare with the state of the art *software* solution, RHHH[13].



(a) Frequency Estimation **(b)** Top-128 **(c)** Convergence of Top-128 Recall over time

Figure 3.9: Comparison between the Deterministic PRECISION (DPRECISION) variant and the randomized one.

indeed the light-weight Randomized Prefix Selection is slightly worse. However, the differences are small and the two algorithms are comparable for each prefix size.

To illustrate the attrativeness of our solution, we add a comparison to the state of the art software HHH algorithm, Randomized HHH (RHHH)[13]. As shown, our solutions are not far behind in terms of accuracy and it is usually enough to use PRECISION with double the space for getting comparable, or better, results. The exception is for the 8-bit networks, where RHHH gets significantly better accuracy. However, one can avoid using approximation algorithms altogether for these and count each of them separately using only $2^8 = 256$ counters.

3.5.7 Deterministic PRECISION

We evaluate the deterministic variant of PRECISION and compare it to our randomized algorithm and RAP. As depicted in Figure 3.9, the deterministic version outperforms the randomized one on the CAIDA trace and, after convergence, has similar accuracy to RAP. We therefore conclude that in workloads which are not adversarial there are benefits for the deterministic approach.

3.6 Conclusions

We designed a novel heavy hitter detection algorithm, PRECISION, that confronts to the memory access constraints imposed by PISA switches and can run on the Intel Tofino programmable switch. PRECISION recirculates a small fraction of the packets for a second pipeline traversal, inducing a small (e.g., 1%) throughput overhead in order to follow the constraints. We studied the impact of each PISA architectural restriction on our algorithms' accuracy. We concluded that the most severe impact comes from the lack of access to an unrestricted random integer generator, and specifically build a better approximation technique to mitigate the impact. We also present a deterministic variant of PRECISION, and further generalized PRECISION to solve the Hierarchical Heavy Hitters problem.

We performed extensive evaluation using real and synthetic packet traces, and demonstrated PRECISION is up to 100× more accurate when estimating per-flow frequency, or saves up to 8× memory space when identifying the top-128 flows, compared with HashPipe[111], a recently suggested alternative for programmable switches.

PRECISION enables heavy-hitter measurements at Tbps-scale aggregated throughput on today's high-performance programmable switches, at competitive accuracy compared to the state-of-the-art algorithms. Furthermore, we hope that our detailed case study of adapting a measurement algorithm to the programmable switch pipeline would provide useful insights for implementing various other algorithms on such switches.

4

BeauCoup: Running multiple measurement queries simultaneously

In this chapter, we propose BeauCoup, a system based on the *coupon collector problem*, that supports multiple distinct counting queries simultaneously while making only a small constant number of memory accesses per packet. It helps network operators who need to constantly monitor network traffic for different types of anomalies and attacks, while still conforming to the constrained memory architecture of high-speed programmable switches. We implement BeauCoup on PISA commodity programmable switches, satisfying the strict memory size and access constraints while using a moderate portion of other data-plane hardware resources. Evaluations show BeauCoup achieves the same accuracy as other sketch-based or sampling-based solutions using 4x fewer memory access.

The work in this chapter was completed in collaboration with Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. It was first presented in ACM SIGCOMM 2020[36].

4.1 Introduction

Network operators constantly monitor network traffic to detect attacks, performance problems, and faulty equipment. To ensure that networks are functioning properly, network operators often need to monitor for *multiple* kinds of problems *simultaneously*, including worms, port scans, DDoS attacks, SYN floods, and heavy-hitter flows.

A variety of network-monitoring tasks can be modelled as counting the number of distinct attributes seen across a set of packets. As the simplest example, to detect a host that is spreading a worm we may look for a *super-spreader*[119], or a source IP that sends packets to many (e.g., 1000+) distinct destinations. However, there may be multiple hosts that are spreading worms, thus we need to identify *all* the source IPs sending traffic to many destinations. Furthermore, different tasks may define their keys differently: to identify victims of a DDoS attack, for example, we need to instead look for *destination* IPs that are receiving from many distinct *source* IPs. The diversity of monitoring tasks with different key definitions makes executing them simultaneously even more challenging.

Traditionally, researchers developing measurement algorithms for the switch data plane has mostly focused on the limited memory *space* in the data plane, designing compact data structures that can compute approximate answers for a single traffic-monitoring query[61,74,79,111,119,123], or multiple queries over the same key[78,123]. Extending these solutions to support multiple queries over different keys would require instantiating multiple separate data structures. Having separate data structures would consume precious memory space in the data plane, but this is not the only problem. As we discussed in Subsection 2.2.3, to maintain line rate, programmable switches only allow a small number of *memory accesses* per packet, making it infeasible to update multiple data structures for every packet.

Most existing techniques for handling *multiple* queries rely heavily on software running outside of the data plane, introducing communication overhead and latency. The simplest approach is to randomly sample packets in the data plane[10,42], and have the software compute multiple statistics on the samples. While useful for detecting high-volume flows, random sampling significantly reduces the accuracy for queries that count the number of distinct attributes. To improve accuracy, several recent works collect information about all potentially relevant flows in the data plane, and have the software compute the statistics of interest[59,74,92]. However, these solutions introduce a tension between the volume of data exported from the data plane and the number and diversity of queries that can be answered with reasonable accuracy in real time.

Instead, we need new techniques that can handle numerous heterogeneous queries directly in the data plane, despite the limited memory space and memory access. We present

BeauCoup, which supports a general query abstraction that counts the number of distinct items (i.e., with different *attributes*) seen across a set of related packets (with the same *key*), and flags the keys with distinct counts above a *threshold*. For example, when searching for worms, a packet's source IP is the key, its destination IP is the attribute, and the threshold decides how many distinct destination IPs are needed to flag a source IP as a worm sender. Our goal is to generate an alarm for those source IPs, approximately, within a reasonable error such as 20%-30% of the threshold. BeauCoup runs multiple queries simultaneously, under a strict per-packet memory access constraint. BeauCoup also allows users to define arbitrary packet-header field tuples as query keys and attributes, providing great expressiveness. The query set can be updated on the fly without the need to re-compile the data-plane program; re-compilation is required only when new header field tuples are defined.

The design of BeauCoup takes inspiration from the *coupon-collector problem*[53]. Using super-spreader detection as an example, suppose we want to know if a sender has sent packets to at least 130 different destination IP addresses. Instead of recording all destination IPs we see, we define 32 *coupons*, and map each destination IP to one of the 32 coupons uniformly at random. Now, for each packet from that sender, we extract the destination IP and *collect* its associated coupon. The coupon may be a duplicate (was already collected earlier), either because the same destination IP appears twice, or because two destination IPs map to the same coupon. We then wait until we have collected each of the 32 coupons at least once to flag the sender as a super-spreader.

The coupon-collector problem asks how many random draws (with replacement) are needed to collect all of the coupons, i.e., have every coupon drawn at least once. With 32 coupons, we need 129.9 draws in expectation. We therefore can use a 32-coupon collector to identify if a particular sender is sending to 130 (or more) distinct destination IPs. Answering a query with a different threshold (say, 1000 destination IPs) requires tuning the coupon collector's configuration, by changing the number of coupons (m), the probability (p) of drawing each coupon for a new destination IP, or the number of coupons that must be collected (n). Essentially, we are using a m-bit vector to estimate whether the number of distinct items seen has exceeded a threshold. A naive m-bit coupon collector is equivalent to either a HyperLogLog[52] register with m 1-bit hash functions, or a m-bit Bloom Filter[7] with only 1 hash function. We discuss the equivalence in more detail in Section 4.7.

The challenge in designing BeauCoup lies in applying the coupon-collection problem to *multiple queries*, each with *different keys and attributes* entirely in the data plane, under strict memory constraints. To limit memory size, BeauCoup must keep coupon state small, devote state to a key only when needed, and share memory across queries and keys. Furthermore, to limit the memory accesses when processing a packet, BeauCoup collects *at most one* coupon per packet. BeauCoup must ensure each query only draws a coupon with a small enough probability, and coordinate among different queries to avoid collecting many coupons concurrently. Thus, BeauCoup must tune the coupon-collector parameters (i.e., m, p, and n) carefully to *simultaneously* achieve accurate results for each query and ensure that the combination of queries does not violate the memory access constraint. Finally, we must implement each part of the BeauCoup algorithm using only the operations available in high-speed programmable switches.

Outline

The chapter is structured as follows. In Section 4.2, we introduce a novel algorithm for executing multiple count-distinct queries under memory size and access constraints. In Section 4.3, we discuss how a compiler optimizes the accuracy of a set of queries, subject to the memory constraints. Section 4.4 presents a prototype system design that translates high-level queries into data-plane configuration for a PISA switch. In Section 4.5 we evaluate our prototype's accuracy. We discuss some future work in Section 4.6 and compare with related work in Section 4.7. Finally, we conclude in Section 4.8.

4.2 The BeauCoup Algorithm

We now show the BeauCoup algorithm for network-monitoring queries. We first present a query model based on distinct counting, that supports a variety of network-monitoring tasks. Next, we discuss how to use coupon collectors to implement these queries. Finally, we discuss how to use coupon collectors to run multiple queries simultaneously, under a strict per-packet memory access constraint.

4.2.1 Query: Count-Distinct with Threshold

A wide variety of network-monitoring tasks can be characterized as a query q which (1) maps each packet i to a key $key_q(i)$, (2) counts the number of distinct attributes $attr_q(i)$ that appear for each key, and (3) applies a threshold T_q to the count to decide whether to report a key. That is, BeauCoup should output an alert (q, k) for query q and key k, when

Name	Key	Attribute	Threshold
Super-spreader	*srcIP*	*dstIP*	1000
DDoS victim	*dstIP*	*srcIP*	1000
Port scan	$\{srcIP, dstIP\}$	*dstPort*	100
Heavy hitter IP pair	$\{srcIP, dstIP\}$	*timestamp*	10000
Heavy hitter IP&Port pair	$\{srcIP, srcPort, dstIP, dstPort\}$	*timestamp*	10000
SYN-flood	$\{dstIP, dstPort\}$	$\{srcIP, srcPort\}$ if TCP SYN, otherwise $\emptyset$	5000

Table 4.1: Examples of count-distinct query definitions.

the packets in a time window W satisfy:

$$\left|\{attr_q(i) \mid key_q(i) = k\}\right| > T_q. \tag{4.1}$$

For the super-spreader example in the Introduction, the key is the packet's source IP, the attribute is the destination IP, and the threshold is 1000. For DDoS detection, we can instead use the destination IP as a packet's key, use the source IP as the attribute, and perhaps use a higher threshold like 10000.

In Table 4.1, we present more examples of common network-monitoring tasks under our query model. In particular, the special attribute *i.timestamp* is unique across all packets, so the user may write a query to count packets by defining $attr_q(i) = \{i.timestamp\}$, i.e., counting the number of unique timestamps seen. Filtering operations can also be expressed in this query formulation, as shown in the SYN-flood example above—by mapping irrelevant packets to a fixed value, the distinct counting query effectively ignores them.

Notation	Definition
$key_q(\cdot)$	Key definition for query q
$attr_q(\cdot)$	Attribute definition for query q
T_q	Threshold for query q
W	Time window for answering queries
Γ	Maximum memory access per packet
c	Number of accesses for collecting one coupon
S	Memory size
m_q	Total number of coupons for query q
p_q	Probability of drawing a particular coupon
n_q	Number of different coupons to collect
γ_q	Average number of coupons activated per packet

Table 4.2: Summary of notations used in this Chapter.

Many other network-monitoring tasks can be expressed in this formulation by using a combination of packet IP addresses, ports, timestamps, etc. as the query key and attribute.

Our goal is to build a system that simultaneously executes a set of queries $\mathcal{Q} = \{q_1, q_2, \dots\}$ and outputs alerts (q_j, k_j), subject to the hardware constraints of a maximum memory size S and at most Γ *memory accesses* per packet. In the rest of this section, we discuss how BeauCoup achieves $\Gamma = O(1)$, i.e., answering multiple queries in the data plane using a small *constant* number of memory accesses per packet, independent of the number of queries.

4.2.2 Updating the Coupon-Collector Table

We maintain a table with bit vectors representing the coupon collectors, as shown in Figure 4.1. Upon collecting the first coupon for the query-key pair (q, k), BeauCoup creates a

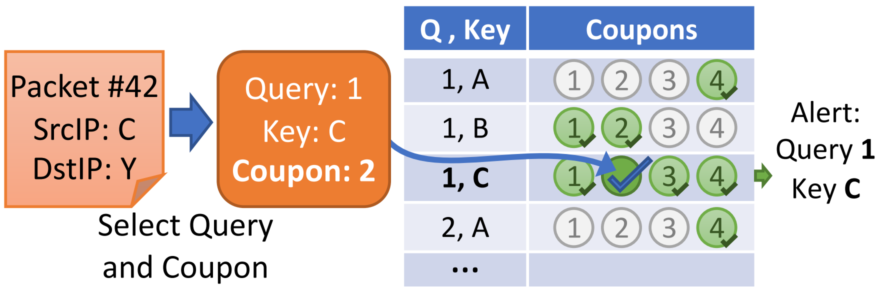


Figure 4.1: We collect coupons by updating bit vectors in an in-memory coupons table.

new table entry; when the bit vector indicates enough coupons have been collected, BeauCoup generates an alert for (q, k).

The example in Figure 4.1 uses 4-coupon collectors for all queries. When a packet arrives at the switch, BeauCoup first selects a query and a coupon. In this case, coupon #2 for query q_1 is selected, and we can extract the query key C from the packet, using the query's key definition. Now BeauCoup finds the coupon collector in the in-memory coupon table under row $(1, C)$, and collects the second coupon by marking the bit vector's second bit to 1. If there is no such row in the table, we allocate a new row and collect the single coupon. Since now all four coupons are collected at least once for row $(1, C)$, BeauCoup reports that key C satisfied query q_1. Other packets may collect coupons for other queries, or do not collect any coupon at all.

The coupon table shown in Figure 4.1 is designed to fit the hardware constraints of PISA programmable switches:

- **Compact rows:** Each row of the table stores one w-bit word as a bit vector, representing at most w coupons, where each bit represents whether a particular coupon has been collected at least once. (We also store two more words of auxiliary data per row, to record a timestamp and a checksum of the query key, which are used for detecting timeouts and hash collisions.)

- **Space efficiency:** We only maintain the bit vector for a query key when there's at least one coupon collected for that key. Therefore, although each query has many keys (e.g., 2^{32}), only a small fraction of *active keys* occupies memory. Different keys (such as keys A, B, and C for query q_1) and different queries (such as queries q_1 and q_2) effectively multiplex a shared memory space, and a new entry is created when a key collects its first coupon.

- **Limited access:** BeauCoup only needs to access the in-memory table when it needs to collect a coupon. When a packet does not produce any coupon for a query, we do not need to access memory. This effectively allows us to multiplex memory accesses across queries, by having different packets updating the table for different queries.

A coupon collector defines m coupons, a probability p for drawing each coupon in a random draw, and stops when there are at least n different coupons collected, i.e., each of these n coupons had been drawn at least once. Since BeauCoup uses a random (yet fixed) mapping from attributes to coupons, observing a new, unseen attribute is equivalent to randomly drawing a coupon. Seeing the same attribute more than once has no effect on the coupon collector, as it merely draws the same coupon again. With an appropriate combination of parameters (m, p, n), the coupon collector can be used to indicate if there are more than T_q distinct attributes seen, while automatically ignoring duplicate attributes.

4.2.3 Selecting a Query and a Coupon

We now discuss how we select one coupon for a given query, and how we coordinate between multiple queries.

Selecting one of m coupons. For every query q, with key definition key_q and attribute definition $attr_q$, BeauCoup applies a random hash function h on packet i's attribute $attr_q(i)$, where

$$h : \{attr_q(i), \forall i\} \rightarrow [0, 1), \tag{4.2}$$

and checks if the output of the hash function falls into a range. For example, suppose query q uses four coupons ($m_q = 4$) and selects each coupon with probability $p_q = 1/8$. Then, BeauCoup would map all attributes satisfying

$$h(attr_q(i)) \in [0, 1/8) \tag{4.3}$$

to coupon #1; similarly, coupons #2, #3, and #4 are associated with output ranges $[1/8, 2/8)$, $[2/8, 3/8)$, and $[3/8, 4/8)$, respectively. If the output of the hash function for packet i falls in $[0, 4/8)$, BeauCoup sets the bit for the associated coupon to 1 for that query-key pair, creating an entry in the table if needed.

If the output of the hash function falls in $[4/8, 1)$, BeauCoup does *not* need to access memory for this query on behalf of this packet, and it can use the memory access for other queries. We define γ_q as the *average* number of activated coupons allowed per packet for query q; with the random hash function h, the example query only activates

$$\gamma_q = m_q \cdot p_q = 1/2 \tag{4.4}$$

coupons per packet in expectation. A small $\gamma_q < 1$ has two main advantages. First, the coupon table does not need to maintain state for every active key. Instead, BeauCoup only allocates memory for a query-key pair upon collecting the first coupon for that key. Second, a small γ_q allows multiple queries to run concurrently under a maximum memory access constraint $\Gamma = O(1)$. In particular, when a particular query q is not collecting a coupon, BeauCoup can devote the unused memory access "budget" to collect a coupon for another query, as we discuss next.

Each query q has its own limit γ_q on how many coupons to collect per packet. For simplicity, we assume a naive fair allocation that gives each query the same share of memory accesses. Given that collecting a coupon costs c memory accesses and a total memory access budget of Γ per packet, we limit each query to collect at most

$$\gamma_q = \Gamma/|c \cdot \mathcal{Q}| \tag{4.5}$$

coupons per packet on average. Therefore, each query's coupon-collector configuration should satisfy

$$m_q \cdot p_q \leq \gamma_q. \tag{4.6}$$

However, a naive choice of hash functions could have a single packet need to collect a coupon for many different queries, even if the average rate of memory accesses is constant. To obey the strict per-packet memory access constraint Γ, BeauCoup coordinates the hash functions across the queries, first among all queries using the same attribute, and second across sets of queries using different attributes.

Grouping queries with the same attribute. Queries may have the same attribute definition (say, destination IP) but with different key definitions (say, source IP for query q_1, and source IP and source port tuple for query q_2). These queries can use the *same* hash function, applied to their common attribute, to draw their coupons. To guarantee that at most one query collects a coupon, BeauCoup divides the hash output across the queries. For example, suppose query q_1 uses $m_1 = 2$ coupons each with probability $p_1 = 1/4$, while query q_2 uses $m_2 = 2$ coupons each with probability $p_2 = 1/8$. We partition the range $[0, 1)$ of the hash output as follows: $[0, 1/4)$ for coupon #1 of q_1, $[1/4, 2/4)$ for coupon #2 of q_1, $[2/4, 2/4 + 1/8)$ for coupon #1 of q_2, and $[2/4 + 1/8, 2/4 + 2/8)$ for coupon #2 of q_2. Other output values are not associated with any coupon. We illustrate this example in Figure 4.2. We can stack additional queries using the same attribute accordingly. Note that we never run out of the $[0, 1)$ range, as long as the total memory accesses across all queries ($\sum_q m_q \cdot p_q$) is bounded by $\Gamma \leq c$, i.e., each packet collects at most one coupon.

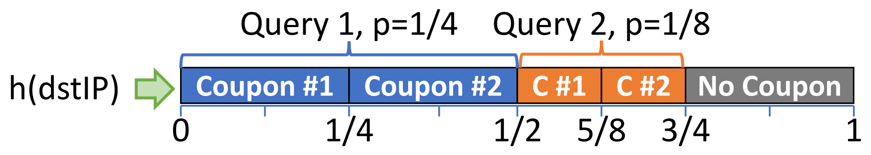


Figure 4.2: Different queries use disjoint ranges to map the random hash function's output to coupons.

Coordinating across queries with different attributes. To support queries with different attribute definitions, BeauCoupconstructs one random hash function for each unique attribute (e.g., one hash function for destination IP, one for timestamp, and so on). When a packet arrives, BeauCoup computes all of these random hash functions to determine if any hash function's output value is associated with a coupon for some query. If only one hash function draws a coupon, BeauCoup collects the coupon for the associated query and key. However, if multiple coupons are drawn, we perform tie-breaking. Currently, BeauCoup only tie-breaks if exactly two hash functions draw coupons, by tossing a coin and allowing each coupon to succeed with 50% probability. In Subsection 4.4.1 we will discuss the implementation detail of the coin toss, and prove that the probability of simultaneously drawing too many coupons for one packet is very small.

With the coordination within and across hash functions, BeauCoup can now guarantee collecting *at most one* coupon per packet, without meaningfully impacting the accuracy of individual query's coupon collectors. Each individual query still collects coupons with the right probability, as if it is the only query running in the system. Given the strict memory access constraint, such coordination is what makes it possible to run many queries simultaneously while maintaining reasonable accuracy for all of them.

4.3 The BeauCoup Query Compiler

For each query q, BeauCoup computes three coupon-collector parameters: collect n_q out of m_q coupons, each with probability p_q. Taking the threshold T_q and the average per-packet coupon limit γ_q for all queries $q \in \mathcal{Q}$ as input, the BeauCoup compiler produces the configuration of $\{m_q, p_q, n_q\}$ that maximizes accuracy. A configuration satisfies the average per-packet coupon limit as long as $m_q \cdot p_q \leq \gamma_q$, which means a query produces at most γ_q coupons per packet in expectation. However, characterizing a coupon collector's accuracy for tracking the threshold T_q is less straightforward. We want the *number of random draws* needed until the coupon collector collects enough coupons to both be *unbiased* (close to T_q in expectation) and *stable* (has small variance). In this section, we first define and analyze an accuracy metric for coupon-collector configurations, then present our method for finding the best configuration for each query.

4.3.1 Coupon Collector's Accuracy

Given a specific query threshold T_q, a coupon-collector configuration is accurate if the number of random draws it needs has an expectation close to T_q and a small variance. Let us first analyze the expectation. We note that the traditional coupon-collector problem requires $n = m = 1/p$, so we present the following analysis for our generalized coupon-collector problem ($1 \leq n \leq m, 0 \leq p \leq 1/m$):

Lemma 4.3.1. *A generalized coupon collector with m coupons in total, each coupon having probability p being drawn upon each random draw, and stops after collecting n different coupons, needs in expectation*

$$CC(m, p, n) \triangleq \sum_{j=0}^{n-1} \frac{1}{p(m-j)} \tag{4.7}$$

coupon draws.

Proof. With j coupons already collected, the probability that the next draw produces a new, unseen coupon (out of the $m - j$ remaining) is $p(m - j)$. Thus, the number of draws needed until receiving a new coupon is a geometric random variable $Geo(p(m - j))$ with expectation $\frac{1}{p(m-j)}$. We need to collect n new coupons, hence the total number of draws is

$$\sum_{j=0}^{n-1} Geo(p(m-j)) = \sum_{j=0}^{n-1} \frac{1}{p(m-j)} \tag{4.8}$$

in expectation. $\square$

However, the configuration with the closest expectation $CC(m, p, n)$ from T_q may have a large variance in the number of draws needed. Therefore, we define *Relative Error*, an accuracy metric for a distinct counting algorithm running query q with threshold T_q, that simultaneously captures the bias and variance of a coupon-collector configuration.

- **True count:** Say the algorithm first outputs an alert (q, k) after observing the input stream $i_1, i_2, \ldots, i_t$; at this time, the ground truth number of distinct attributes seen by the algorithm is $\mathcal{T} = \left|\{attr_q(i) \mid key_q(i) = k, i \in i_1, i_2, \ldots, i_t\}\right|$.
- **Absolute error:** However, the algorithm should generate an alert when there are exactly T_q distinct attributes. We define the absolute error as $|\mathcal{T} - T_q|$.
- **Relative error:** We normalize and use $\frac{|\mathcal{T}-T_q|}{T_q}$ as the relative error of output (q, k). This scaled error includes both the bias $E[\mathcal{T}] - T_q$ and the variance of $\mathcal{T}$.

By running the same algorithm many times with different random hash functions, we can have many observations of Relative Error for the same query, and we can subsequently define *Mean Relative Error* as the mean of all observations.

Next, we discuss how BeauCoup finds a coupon-collector configuration with small Mean Relative Error for every query.

4.3.2 Finding the Best Configuration

The BeauCoup compiler needs to identify one coupon-collector configuration for every query given the query's threshold T_q, and we focus on how we satisfy the strict per-packet memory access constraint. When implementing BeauCoup on PISA switches, our choice for m_q, p_q, and n_q is subject to hardware constraints. Namely, since a memory word is $w = 32$-bit we require $m_q \leq 32$, and to facilitate efficient mapping from random hash function to coupons we require p_q to be an integer power of two. Also, we must satisfy the average per-packet coupon limit γ_q: we require in expectation that we collect fewer than γ_q coupons per packet, i.e., $m_q \cdot p_q \leq \gamma_q$.

Thus, we use the following procedure to find the configuration given threshold T_q and per-packet coupon limit γ_q:

1. For all feasible coupon probabilities $p_q = 2^{-j}$, we calculate the maximum number of coupons allowed, based on both the per-packet coupon limit and the word length: $\overline{m_q} = min(w, \gamma_q/p_q)$. We stop if $\overline{m_q} < 1$.

2. For each p_q, we identify all feasible configurations $1 \leq n_q \leq m_q \leq \overline{m_q}$. We then calculate their expected number of draws $CC(m_q, p_q, n_q)$ for all feasible configurations, and accept a configuration as reasonable when it is within a 5% tolerance from T_q, i.e., $0.95T_q < CC(m_q, p_q, n_q) < 1.05T_q$. The 5% tolerance is selected because the minimum relative error for the optimal collectors is about 10%, and is relaxed when no reasonable configuration was found.

3. Given all of the reasonable configurations, we choose the optimal configuration based on their minimum relative error, according to a lookup table prepared via simulations (shown later in the Evaluation section in Figure 4.5).

4.4 BeauCoup on PISA Hardware

In this section, we describe how we implement BeauCoup on PISA programmable switches. PISA switches always process packets at line rate (at least 100Gbps per port), which requires the algorithms running on it to comply with several hardware-imposed resource constraints.

PISA switches have two kinds of memory. Ternary Content-Addressable Memory (TCAM) holds match-action rules installed by the control software, while Static Random Access Memory (SRAM) holds general-purpose register arrays that can be updated within the data plane. TCAM can simultaneously match a bit string with many match rules, and is typically used for forwarding packets by matching on the IP prefix. BeauCoup utilizes a small fraction of the available TCAM space to efficiently implement both the mapping from attributes to coupons and the tie-breaking process between queries. Meanwhile, BeauCoup collects coupons by updating SRAM entries. The SRAM memory space is limited (several megabytes), and more importantly we can only perform a small, constant number of memory accesses to SRAM per packet. In this paper, we primarily focus on the limited SRAM space and the limited number of SRAM accesses allowed.

BeauCoup's implementation has two components: the data-plane program executes the logic for collecting coupons, and the control algorithm transforms queries into coupon-collector configurations, as illustrated in Figure 4.3. Now we first introduce how we implement the data-plane program to run the coupon collectors on PISA hardware, then discuss how BeauCoup as a whole executes and updates queries.

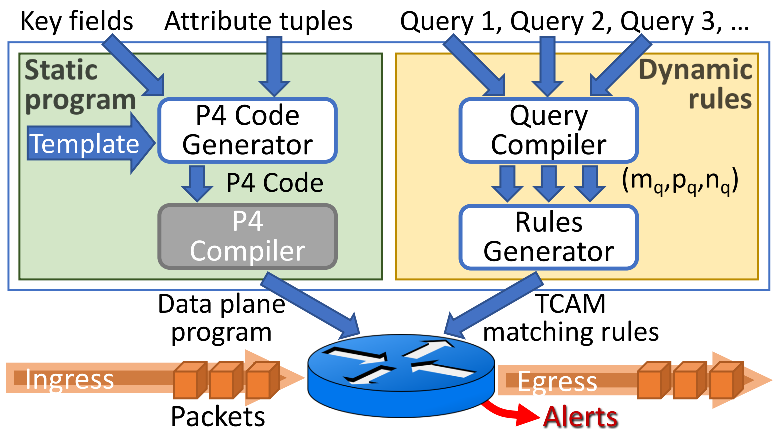


Figure 4.3: BeauCoup runs queries by installing a static data-plane program on the PISA switch, then generating and installing TCAM rules on the fly.

4.4.1 Using TCAM for Drawing Coupons

BeauCoup needs to draw coupons based on the output of random hash functions. Since each hash function maps to a large number of coupons, we utilize the TCAM to efficiently check if the hash function's output value maps to any of the ranges defined by the coupons.

Each random hash function's output is encoded into 16 bits, and each coupon's corresponding range is translated to a bit prefix match for these random bits. For example, we translate the coupons of q_1 and q_2 shown in Figure 4.2 into matching rules in Table #1 in Figure 4.4. Coupon #1 of query q_1 matches on range $[0, 1/4)$, which is transformed to a bit prefix match `00*` (the first rule in Table #1). Coupon #2 of query q_2 matches on $[2/4 + 1/8, 2/4 + 2/8)$, which is transformed to prefix `101*` (the last rule in Table #1).

After we use TCAM tables to match on every hash function's output, we use a bit vector to represent if any one of the many hash functions had matched with a coupon. As there could be zero or more coupons, we again use the TCAM to efficiently tie-break and select one coupon to collect when there may be multiple coupons available. The matching rules are trivial when there are zero or exactly one coupon matched. If there are exactly two coupons available, we flip a random coin (by using a random bit from the random number generator) to fairly tie-break and select one of the two for collection. We ignore all coupons if there are more than three. We can show this has very minor effect on BeauCoup's accuracy.

Remark We can bound the probability of having more than 3 coupons collide as follows. Assume the system uses $H \geq 3$ random hash functions, each with activation probability $x_1, x_2, \ldots, x_H$. Since the expected number of coupons per packet $\sum_{q \in Q}$ is bounded by 1, we have $\sum x_i \leq 1$. Collision happens when multiple hash functions activate; the probability for having more than 3 coupons drawn is maximized when all hash functions share the same probability, i.e., $\forall i, x_i = \frac{1}{H}$, due to the inequality of arithmetic and geometric means. In this case, the number of coupons drawn follows a binomial distribution $B(n = H, p = \frac{1}{H})$, and the probability of having more than 3 coupons can be bounded by

$$\Pr[\text{collision}] \leq \Pr\left[B(n = H, p = \frac{1}{H}) \geq 3\right]. \tag{4.9}$$

For example, if we take $H = 11$ (from the example query set we used in Section 4.5), we have $\Pr[\text{collision}] \leq 7.11\%$. This error is small compared to the 10%-20% relative error incurred by the coupon collector.

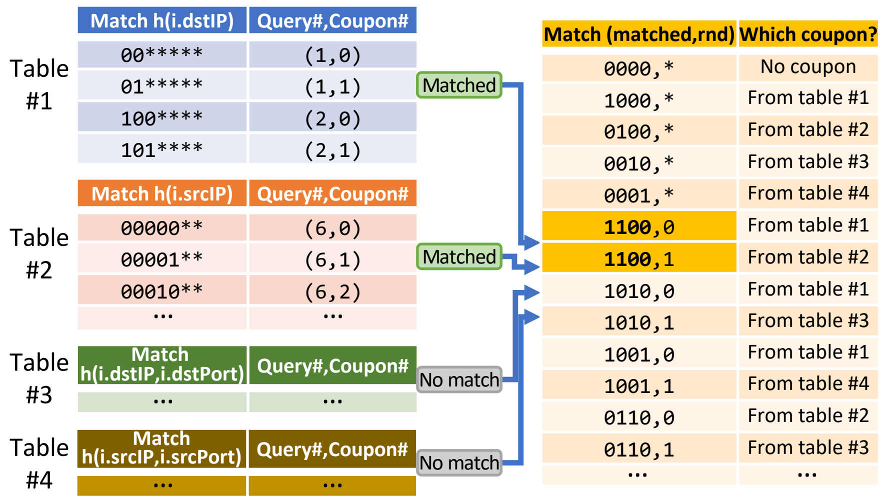


Table #1

Match h(i.dstIP)	Query#,Coupon#
`00*****`	`(1,0)`
`01*****`	`(1,1)`
`100****`	`(2,0)`
`101****`	`(2,1)`

Table #2

Match h(i.srcIP)	Query#,Coupon#
`00000**`	`(6,0)`
`00001**`	`(6,1)`
`00010**`	`(6,2)`
...	...

Table #3

Match h(i.dstIP,i.dstPort)	Query#,Coupon#
...	...

Table #4

Match h(i.srcIP,i.srcPort)	Query#,Coupon#
...	...

Match (matched,rnd)	Which coupon?
`0000,*`	No coupon
`1000,*`	From table #1
`0100,*`	From table #2
`0010,*`	From table #3
`0001,*`	From table #4
`1100,0`	From table #1
`1100,1`	From table #2
`1010,0`	From table #1
`1010,1`	From table #3
`1001,0`	From table #1
`1001,1`	From table #4
`0110,0`	From table #2
`0110,1`	From table #3
...	...

Figure 4.4: Using TCAM rules to draw coupons.

We can further derive the upper limit for this probability:

$$
\begin{aligned}
& \lim_{H\to\infty} 1 - \Pr[B(n = H, p = \frac{1}{H}) \leq 2] \\
= \quad & 1 - \lim_{n\to\infty} \sum_{k=0,1,2} C(n,k)(1/n)^k(1-1/n)^{(n-k)} \\
= \quad & 1 - \lim_{n\to\infty} \frac{5n-2}{2n-2}(1-\frac{1}{n})^n \\
= \quad & 1 - \frac{5}{2e} \approx 8.03\%
\end{aligned}
\tag{4.10}
$$

Still, this tiebreaking creates a small bias for individual coupon's activation probability; we leave the correction for this bias in the query compiler for future work.

We illustrate the coupon matching and the tie-breaking process in Figure 4.4. There are four random hash functions and four corresponding match tables (on the left) to draw coupons. After matching, Table #1 and #2 produced coupons while Table #3 and #4 did not. We use the bit vector `1100` to represent which tables produced coupons. A tie-breaking table (on the right) uses TCAM match rules to match on the bit vector `1100`, and there are two matching rules (highlighted in yellow). The table matches on the random bit to tie-break, and chooses either the coupon from Table #1 or the one from Table #2 as the final coupon for collection.

4.4.2 Recording Coupons in SRAM

After BeauCoup has selected a query q and chosen a coupon c for packet i (using TCAM matching), we need to collect c into the in-memory coupon table. We used the SRAM-based register arrays on PISA switches to record coupons and other states. Each array holds S memory words, indexed $0, 1, \ldots, S\text{-}1$, and each word has 32 bits. Given an index, we can read the existing value at this index, perform arithmetics, and write a new value; this counts as one memory access.

BeauCoup first extracts the query key $key_q(i)$ from the packet, then locates an index using the tuple $(q, key_q(i))$. We use an indexing random hash function H to map the tuple into an array index, denoted $idx = H(q, key_q(i))$.

BeauCoup defines three register arrays, each with S words. $\mathcal{TS}[\cdot]$ stores timestamps, and is used to enforce the query time window W for every coupon collector; we reclaim memory when a collector is timed out before collecting enough coupons. $\mathcal{QK}[\cdot]$ stores 32-bit checksums $checksum(key_q(i))$ and is used to detect hash collisions in the indexing hash function H, avoiding two keys adding coupons into the same collector bit vector. Finally, $\mathcal{CC}[\cdot]$ stores all the coupon collector bit vectors.

The process for collecting the coupon c for query q and key $key_q(i)$ is as follows, accessing at most three words of memory, First, we calculate the array index $idx = H(q, key_q(i))$, and encode the coupon into a variable $onehot(c)$, a 32-bit binary string "`000...010...0`" with all bits `0` except one `1` at the location corresponding to the coupon c. Subsequently, we check whether we are creating a new coupon collector or adding this coupon to an existing collector, using query time window W and current timestamp $i.timestamp$:

- **Create new collector:** If $\mathcal{TS}[idx] < i.timestamp - W$, the current collector has expired. We allocate a new coupon collector by setting $\mathcal{TS}[idx] \leftarrow i.timestamp$ as well as $\mathcal{QK}[idx] \leftarrow checksum(key_q(i))$. We initialize the collector bit vector with one coupon: $\mathcal{CC}[idx] \leftarrow onehot(c)$.

- **Update existing collector:** If $\mathcal{TS}[idx] \geq i.timestamp - W$ and $\mathcal{QK}[idx] = checksum(key_q(i))$, we accumulate into an existing coupon collector. We update its bit vector using bitwise-OR: $\mathcal{CC}[idx] \leftarrow (\mathcal{CC}[idx] \vee onehot(c))$. Now, if the number of one bits in $\mathcal{CC}[idx]$ reaches n_q, we output an alert $(q, key_q(i))$.

- **Handle collision**: If $\mathcal{TS}[idx] \geq i.timestamp - W$ yet $\mathcal{QK}[idx] \neq checksum(key_q(i))$, we encountered a hash collision; the system ignores this coupon. This indicates there are too many active coupon collectors, hence the system is running out of memory. We discuss how to address memory size constraint and hash collisions in Section 4.6.

We note that coupon collectors for different queries uses the same block of memory space, statistically multiplexing their memory demand. Therefore, we may encounter high memory load when many different queries simultaneously collect coupons for many keys. We discuss BeauCoup's memory size requirement under real-world traffic settings in Section 4.5.2.

4.4.3 Query Compiler and Code Generation

Figure 4.3 presents the high-level architecture of the BeauCoup system. Given a set of queries $\mathcal{Q}$, we first run a query compiler (using the algorithm in Section 4.3.2) to compute a configuration $\{m_q, p_q, n_q\}$ for each query q, and produce the hash functions for attributes. The *query compiler* generates an intermediate representation with the mapping from each hash function's output values to all of the coupons. Subsequently, the *rules generator* uses these mappings to generate the TCAM matching rules and the corresponding action parameters, representing the query set $\mathcal{Q}$.

Meanwhile, BeauCoup generates the P4 code for the switch using a python-based *code generator*. The generator uses an algorithm template (approximately 750 lines), written under the Jinja[102] templating language, that implements BeauCoup's data-plane algorithm. Jinja enables auto-generating repeated P4 elements, such as defining multiple hash functions and variables. Given the queries' key fields and attribute tuples as input, the code generator prepares the definition for hash functions, then expands the template into P4[22] code (approximately 1500 lines), which is subsequently compiled and installed into the PISA switch. In Listing 1 and Listing 2 we show two excerpts from the template that highlights how it helps us efficiently generate the repetitive P4 data plane program. When the TCAM matching rules are installed in the tables specified by the P4 program, the switch executes the query set $\mathcal{Q}$. We have open-sourced the complete template program, the code generator, as well as the query compiler on GitHub[30].

Although the packet parser (header field definitions), hash functions, and query key extraction rules are part of the P4 data-plane program, the TCAM matching rules can be updated on the fly. The user may frequently change the query set $\mathcal{Q}$, by first running the

```
struct ig_metadata_t {
  {% for h in hash_functions %}
    bit<16> h_{{h.id}};
    bit<1> h_{{h.id}}_matched;
    bit<8> h_{{h.id}}_query_id;
    bit<8> h_{{h.id}}_coupon_id;
    bit<8> h_{{h.id}}_query_n;
    bit<4> h_{{h.id}}_query_keydefn;
  {% endfor %}
  bit<32> coupon_onehot;
  bit<1> random_coin;
  //...
}
{% for h in hash_functions %}
  action calc_hash_{{h.id}}(){
    ig_md.h_{{h.id}}=hash_{{h.id}}.get({ {{h.fields}} });
  }
  action set_h_{{h.id}}_matched(bit<8> qid, bit<8> cid, bit<8> n, bit<4> kdf){
    ig_md.h_{{h.id}}_matched=1;
    ig_md.h_{{h.id}}_query_id=qid;
    ig_md.h_{{h.id}}_coupon_id=cid;
    ig_md.h_{{h.id}}_query_n=n;
    ig_md.h_{{h.id}}_query_keydefn=kdf;
  }
  action set_h_{{h.id}}_no_match(){
    ig_md.h_{{h.id}}_matched=0;
  }
{% endfor %}
```

Listing 1: Use Templating to generate similar code for every hash function.

query compiler and the rules generator, then installing the new matching rules, as long as all queries are using existing key fields and attribute tuples already defined in the data-plane program. This also avoids the potential network downtime caused by re-installing a new data-plane program, which would temporarily interrupt the switch's normal operation. The green shaded box on the left half of Figure 4.3 represents the heavy-weight update of the data-plane program, which is largely static, while the yellow shaded box on the right represents light-weight update of query matching rules, which can be installed swiftly with-

```
action write_onehot(bit<32> o){
  ig_md.coupon_onehot = o;
}
table tb_set_onehot {
  key = {
    ig_md.h_selected_coupon_id: exact;
  }
  size = 32;
  actions = {
    write_onehot;
  }
  default_action = write_onehot(0);
  const entries = {
    {% for i in range(32) %}
      {{i}} : write_onehot(32w{{2**i}});
    {% endfor %}
  }
}
```

Listing 2: Use Templating to generate fixed, repetitive match-action logic.

out causing downtime. Still, using a new header field in a query's key or attribute definition requires re-generating P4 code and re-compiling the data-plane program.

4.5 Evaluation

In this section, we demonstrate that BeauCoup can accurately and efficiently execute multiple queries. We first show that the query compiler produces good parameters for coupon collection. Then, we investigate BeauCoup's performance when answering queries over a real-world traffic trace, under limited memory access constraint, and show it achieves the same accuracy using 4x fewer memory accesses than alternatives. Finally, we show BeauCoup 's data-plane program only uses a modest fraction of the available hardware resources on a commodity switch.

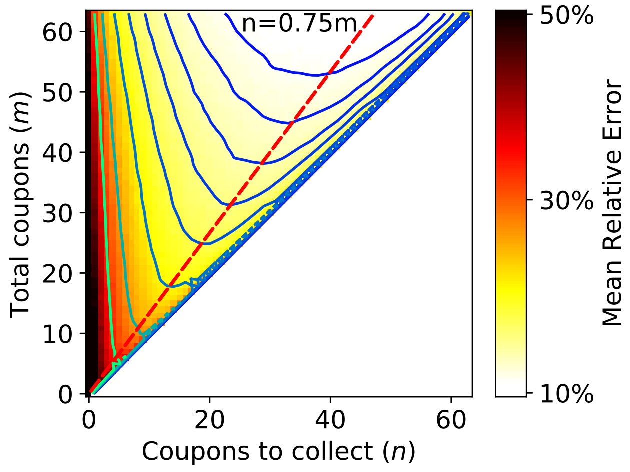


Figure 4.5: When using various coupon collector configurations, we find that collecting approximately $n = 0.75m$ out of m coupons produce the lowest error.

4.5.1 Evaluating the Query Compiler

We now investigate the coupon-collector configurations generated by the query compiler under different thresholds T_q and average per-packet coupon limit γ_q. The compiler's running time is negligible (< 1ms) given its time complexity $O(w^2|\mathcal{Q}|)$.

Recall that the query compiler outputs the configuration $\{m_q, p_q, n_q\}$ with the lowest Mean Relative Error given that its expected number of draws $CC(m_q, p_q, n_q)$ is close to the query threshold T_q. In Figure 4.5 we plot the minimum possible Mean Relative Error of various configurations, when the expected number of draws exactly matches the threshold ($T_q = CC(m_q, p_q, n_q)$). We note that adjusting p_q does not noticeably change the error, and only plotted the relationship between Mean Relative Error and (m_q, n_q) for all configurations in $2 \leq n_q \leq m_q \leq 64$.

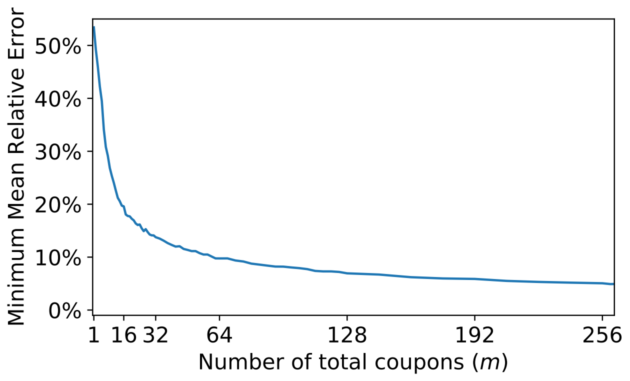


Figure 4.6: Using more coupons lead to lower Mean Relative Error. A coupon collector can achieve 13.7% minimum error when using $m = 32$ coupons.

As we can see from Figure 4.5, in general, using more coupons leads to lower error. We can further observe that for any given m_q (total coupons), the configuration with minimal Mean Relative Error corresponds to a choice of n_q around $0.75m_q$. That is, the coupon-collector configuration should stop when around three-fourths of coupons are collected, as this leads to the least variance in the number of random draws required. We also verified that the $n_q \approx 0.75m_q$ heuristic still holds with thousands of coupons, although we defer a rigorous analysis to future work. However, when memory access is extremely constrained, the compiler often selects $n_q = m_q = 1$, as the configurations using more coupons consume many more memory accesses per packet.

We now look at the relationship between the minimum Mean Relative Error and the total number of coupons (m_q), as shown in Figure 4.6. In our current prototype implementation, we restrict the query compiler to use at most $m_q = 32$ coupons, as one memory read on the PISA hardware reads a 32-bit memory word. Using $m_q = 32$ coupons achieves 13.7% minimum error, which means BeauCoup may send a super-spreader alert upon seeing 860$\sim$1140 distinct IP addresses, given the threshold 1000. We note that BeauCoup can maintain more coupons in a collector by using multiple memory words, if a higher accuracy is desired. Using $m_q = 64$ coupons achieves 9.8% minimum error, while using 128, 256, or 1024 coupons achieves 6.9%, 5.0%, or 3.1% error respectively. These errors are comparable with the HyperLogLog distinct counting algorithm using the same memory space.

4.5.2 Query Accuracy

Now we evaluate the accuracy of BeauCoup queries over real-world network traffic, by first running a single query and comparing BeauCoup with related works, then run many queries simultaneously. Our experiments mostly focus on BeauCoup's accuracy under the limited memory *access* constraint by providing abundant memory for all algorithms. We also present some results regarding limited memory space.

One Query and One Key

We first demonstrate BeauCoup's coupon collectors are an efficient way to perform distinct count queries, by comparing them against other approximate distinct counting algorithms. Here we only focus on counting distinct attributes for one particular query and one particular key, as other distinct counting algorithms are designed for only one key and cannot support multiple keys.

In this experiment, we use different algorithms to count the number of distinct source-destination IP pairs in the traffic, and stop when the estimate exceeds $T = 1000$ distinct IP pairs. All algorithms are implemented in Python. We use the CAIDA Anonymized Internet Traces Dataset 2018[26] (CAIDA trace), and repeat all runs 100 times with different random seeds.

HyperLogLog[52] is a widely-used approximate distinct counting algorithm, that counts distinct items by counting the maximum number of leading zeros seen from a random hash function. The algorithm splits its input and feeds them to multiple independent estimators, and outputs the harmonic mean across all estimators. We use a HyperLogLog instance with 64 estimators.

UnivMon[78] is the state-of-the-art multi-purpose measurement sketch that runs on PISA programmable switches, and can compute various functions over a set of attributes, including distinct counting. NitroSketch[77] performs sampling over sketch memory updates to reduce a sketching algorithm's memory access while preserving its accuracy. The authors of NitroSketch had proposed applying the NitroSketch technique to UnivMon to reduce UnivMon's average memory access per packet. We hereby refer to the new algorithm as NitroSketch-UnivMon. NitroSketch-UnivMon supports all the queries supported by UnivMon, including distinct counting. NitroSketch-UnivMon is the only sketch we are aware of that achieves fewer than one memory access per packet on average and supports distinct counting. We use 16 layers of 4x1024 CountSketch for UnivMon, and change NitroSketch's sampling parameters to let NitroSketch-UnivMon achieve different average memory access per packet.

We also include a packet sampling approach in the comparison. As analyzed by Spang & McKeown[114], it is possible to estimate the distinct number of flows (attributes) given a sampled subset of all packets, using a statistical estimator[27]. We sample each packet with a small probability p, and record each sampled packet's IP pair. Subsequently, we feed the sampled subset to the estimator.

We first note that the memory size used by BeauCoup is minimal: a coupon collector uses one word of memory, at most $w = 32$ bits. Including auxiliary data (timestamp and checksum), each key uses three words, or 96 bits. Meanwhile, one HyperLogLog instance with 64 estimators uses 320 bits of memory. As we discussed in Section 4.5.1, when using the same number of bits of memory space, coupon collectors can achieve comparable accuracy as HyperLogLog.

On the other hand, NitroSketch-UnivMon uses 256 kilobytes of memory space and is not directly comparable, as it is a multi-purpose sketch supporting more than distinct counting. It is possible to fit a handful of instances of NitroSketch-UnivMon into a switch's data-plane memory space, but it is unfeasible to run multiple queries with multiple keys, which requires thousands of instances. Packet sampling uses $O(p \cdot L)$ memory space, proportional to the sampling probability and stream length.

Since we need to simultaneously answer multiple queries under a total per-packet memory access constraint, each BeauCoup query can only make a very small number of memory accesses per packet. We now compare the accuracy of each distinct counting algorithm under the same average memory access constraint of $\gamma \leq 1$ words per packet:

- When using packet sampling, for each sampled packet, we need to access two words of memory to save its IP pair. Thus, we can satisfy the per-packet memory access constraint by setting the sampling probability to $p = \gamma/2$.

- For NitroSketch-UnivMon, we tune each layer's NitroSketch sampling probability individually to achieve $\gamma/16$ average memory access, thus making total memory

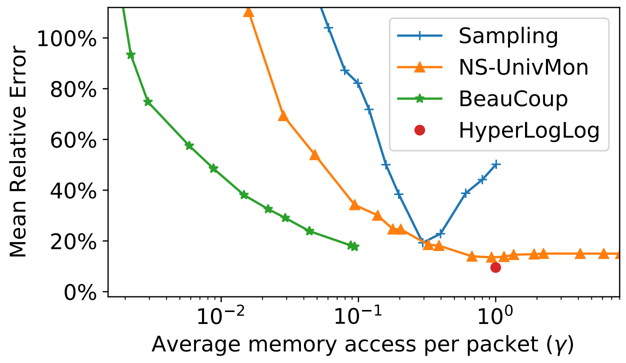


Figure 4.7: BeauCoup's coupon collector approach uses 4x fewer memory access than NitroSketch-UnivMon or sampling to achieve the same accuracy.

access across all layers to fit within γ words per packet. Since not all layers use their access budgets fully, we record the actual number of total memory accesses in experiments.

- For BeauCoup coupon collectors, recall that collecting each coupon requires accessing $c = 3$ words (for coupon vector, timestamp, and checksum). We specify an average per-packet coupon limit $\gamma_q = \gamma/c$, and use the BeauCoup query compiler to find the coupon collector configuration that satisfies the constraint. Here we also record the actual number of memory accesses.

- Finally, although HyperLogLog is very accurate, it always accesses exactly one word of memory per packet, regardless of the number of estimators. We nevertheless included its accuracy for reference.

In Figure 4.7, we show that BeauCoup's coupon collector achieves the same accuracy (Mean Relative Error, plotted on y-axis) using at least 4x fewer memory accesses (γ, plotted on *x*-axis with log scale), compared with NitroSketch-UnivMon, packet sampling, or HyperLogLog.

We note that the statistical estimator used by the packet sampling approach[114] is designed for sparse samples, looking at IP pairs sampled exactly once or twice. Thus, it works better for sparse samples and performs poorly with a very high sampling rate above 0.5, creating non-monotonicity in the figure.

To achieve less than 25% Mean Relative Error for queries, BeauCoup needs 0.04 words of memory access per packet, which means we can run about 25 queries together per word of memory access per packet, while NitroSketch-UnivMon requires 0.2 words of memory access, and can only run about five queries for the same memory access limit. At higher error ranges (e.g., to achieve less than 50% Mean Relative Error), BeauCoup only needs 0.009 words of memory access, while NitroSketch-UnivMon requires 0.09, yielding a 10x saving. The improvements are similar for other attribute definitions and thresholds.

Multiple Queries and Keys

Next, we run BeauCoup with multiple queries and observe the average relative error under varying memory access constraints. We wrote $|\mathcal{Q}| = 26$ queries that resemble monitoring demands a network administrator may have, with keys and attributes defined using combinations of source and destination IP addresses and TCP/UDP ports. The queries use various different combinations of packet header fields as their key and attribute definitions. Some queries also use the timestamp as the attribute definition—recall that we can count the number of packets by performing distinct counting over timestamps. The thresholds range from 100 to 10000, and are selected based on the likely use cases of the particular

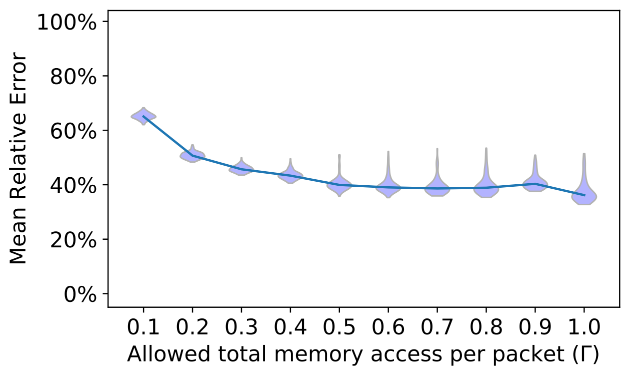


Figure 4.8: The average error of all queries gradually improve as we allow more memory access per packet, which is shared among all queries.

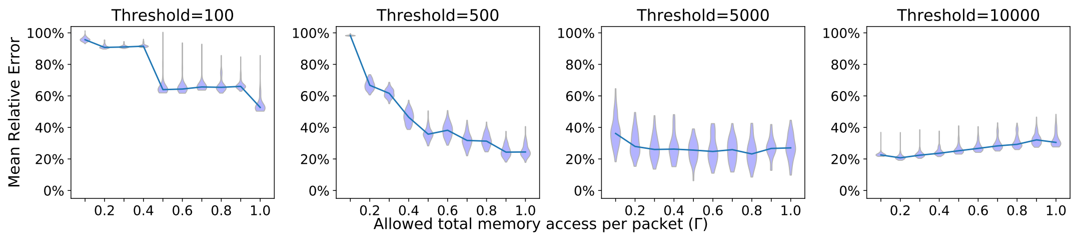


Figure 4.9: Query with the lowest threshold experiences the most significant accuracy improvement when allowing more memory access per packet.

queries. In each experiment, we set Γ, the total memory access constraint for all queries, from 0.1 to 1 access per packet. We then run the query compiler to fairly allocate memory access and generate the coupon-collector configuration for each query.

After obtaining the coupon-collector configurations, we run BeauCoup in a python-based simulator, which is behaviorally equivalent to the data-plane P4 program, but allows us to freely tune all parameters and concurrently run many simulations with different random seeds. We once again use the CAIDA trace in the following experiments.

Average accuracy across queries. Figure 4.8 shows the overall accuracy of all queries, measured by Mean Relative Error, given different total memory access limits Γ. We can ob-

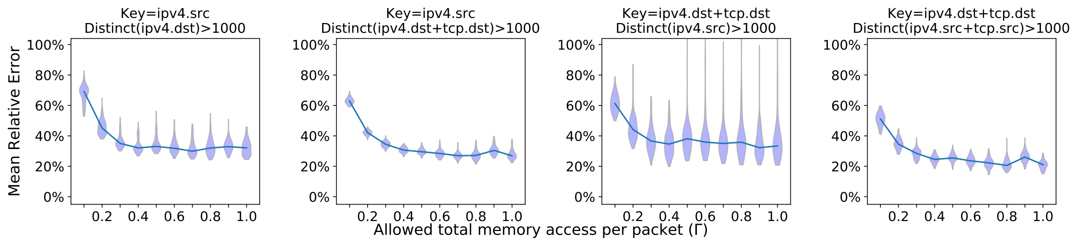


Figure 4.10: Queries with the same threshold exhibits similar accuracy improvement trend when given more allowed memory access, despite different key and attribute definitions.

serve that when the memory access limit becomes lower, the error becomes higher, and the accuracy of different queries gradually converges. This is because when we have abundant memory accesses, the queries with higher thresholds do not need to use all of their fair share of memory accesses, and can achieve better accuracy than those actually constrained by memory access; when all queries are constrained, the fair allocation policy leads to similar accuracy for all queries.

Per-query accuracy. Now we scrutinize the accuracy of each query. We first compare the effect of increasing memory access limit Γ on each query's average relative error. In Figure 4.9, we choose four different queries with various T_q from 100, 500, 5000, to 10000 and analyze their accuracy. Naturally, the query with the lowest threshold is the hardest to execute, as it requires coupons with larger probability p_q and easily exhausts its memory access budget. Increasing Γ allows the query to increase accuracy significantly. For queries with larger T_q, the improvement is not as significant.

Notably, the query with $T_q = 10000$ reaches its optimal accuracy when $\Gamma = 0.2$, and its accuracy slightly deteriorates when we allow more memory accesses. This is due to having collisions with other queries when the system draws more than one coupon and enters tie-breaking more often, which slightly skews the probability of drawing each coupon.

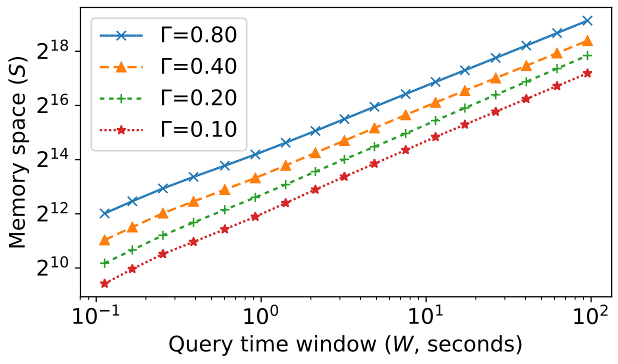


Figure 4.11: The query time window size W and the memory space S (number of coupon collector bit vectors) required by BeauCoup follows power law.

We also compare different queries with the same $T_q = 1000$ yet with different *key*$_q$ and *attr*$_q$ definitions. Here we use four queries as an example, the first one being super-spreader. As we can see from Figure 4.10, their average relative error has almost the same relationship regarding the total memory access constraint Γ. The third plot in Figure 4.10 has a slightly higher variance, and is because this particular query outputs fewer alarms in our experiment trace, hence has more outliers for the average relative error statistics.

Memory Size.

So far, we have focused on limited memory access and assumed unlimited memory size and an infinite time window. However, practical systems have a limited amount of memory (S) and can run out of space for large window size W.

We first observe that the number of unique query keys present in the traffic usually follows power law. For a stream of L packets, we can observe L^{α_q} unique keys, with α_q being specific to the traffic and different key definitions. For the CAIDA trace, α_q ranges between

Component	Match Coupons	Extract Key	Collect Coupons	Teardown	Overall
TCAM	39.6%	2.3%	0%	0%	**13.2%**
SRAM	9.1%	2.1%	26.3%	0%	**12.3%**
Instruction	25.0%	7.3%	5.4%	3.1%	**12.8%**
Hash Unit	50.0%	61.1%	29.1%	0%	**41.7%**

Table 4.3: BeauCoup's hardware resource utilization, categorized into four functional components.

0.7 to 0.85. Therefore, given the average per-packet coupon limit γ_q, we can give an upper bound $(\gamma_q L)^{\alpha_q}$ for the number of coupon collectors needed for query q, and therefore the maximum total memory needed by all queries is upper-bounded by $\sum_{q\in\mathcal{Q}}(\gamma_q L)^{\alpha_q}$.

Figure 4.11 shows the actual memory space requirement of BeauCoup with regards to different time window sizes W, when processing the same query set $\mathcal{Q}$ under the CAIDA trace, under a log-log scale. We can observe that the relationship between the memory size and window size closely follow a power law with an exponent $\alpha = 0.80$. For example, for a time window of $W = 1$ second and memory access limit of $\Gamma = 0.1$ word per packet, BeauCoup needs to store 4096 coupon collectors (48 kilobytes), while doubling the time window to $W = 2$ seconds enlarges the memory size requirement by $2^{\alpha} = 1.74$ times, to 7150 collectors (84 kilobytes). A practical system on PISA switches can easily support $65,536$ collectors, corresponding to a time window $W = 30$ seconds for the CAIDA trace. Still, BeauCoup is optimized for memory access constraint, and we defer the discussion on how to adapt BeauCoup with insufficient memory in Section 4.6.

4.5.3 Hardware Resource Utilization

To run on PISA switches and process packets at 100Gbps line rate, BeauCoup's data-plane program must satisfy other resource constraints beyond limited memory access. Beau-

Coup's auto-generated P4 data-plane program runs on an Intel Tofino[65] Wedge100 programmable switch. It consumes about 40% of the programmable switch's hash calculation units and less than 15% of other resources. We note that BeauCoup is not bottlenecked by TCAM match table size. The current version of our data-plane program supports matching each attribute's hash function output to 4096 different coupons; since every query uses at most 32 coupons, the program supports at least $\frac{4096}{32} = 128$ queries for each attribute. 4096 is the default size for the TCAM match tables set by the compiler, and can be extended as needed. Resource utilization other than TCAM is independent of the number of simultaneous queries we run.

To produce a more detailed picture of BeauCoup's resource utilization, we slice the data-plane program into four sequential functional components, and in Table 4.3 we drill down the utilization for different types of resources by each component. We can see different functional components have distinctive resource utilization profiles. For example, matching coupons extensively uses hash units to calculate random hash functions and uses TCAM to draw coupons, while not using much SRAM; in contrast, collecting coupons requires no TCAM, but uses SRAM to store the bit vectors.

Although the BeauCoup data-plane program uses more hardware resources than running one instance of HyperLogLog or UnivMon for a single key definition, we note that the data-plane program already supports various different key and attribute definitions, allowing us to install new queries on the fly without re-compiling the data-plane program. Furthermore, BeauCoup does not exhaust any one switch resource, and its unique resource usage profile co-habitates well with other typical resource-heavy switch functions or algorithms. When two algorithms use the same resource heavily but at different pipeline

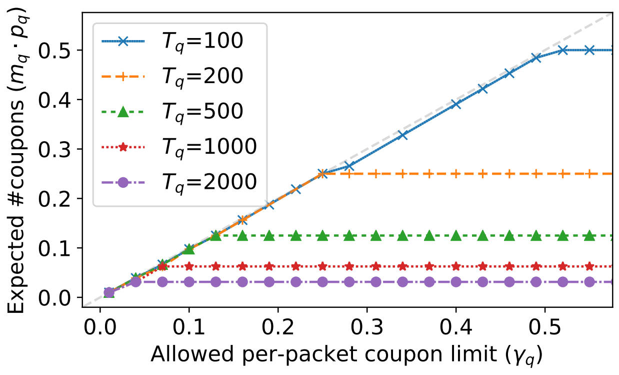


Figure 4.12: Queries with higher threshold T_q need fewer memory accesses per packet.

stages, we can tessellate them without causing resource contention. For example, performing Equal-Cost Multi-Path (ECMP) routing requires computing hash functions late in the switching pipeline, where BeauCoup does not compute many hash functions when collecting coupons; running network measurement sketches like UnivMon[78] or PRECISION (Ch.3) requires using SRAM memory early in the pipeline, whereas BeauCoup does not consume a lot of SRAM early in the pipeline when it is matching coupons.

4.6 Discussion

Fairness between queries

In this paper, we use a fair allocation policy to distribute the limited memory access among all queries.However, queries with larger thresholds require fewer memory accesses to achieve the same accuracy. Figure 4.12 evaluates the optimal configurations found by the query compiler under different per-packet coupon limit γ_q, for various query thresholds T_q. A query with a small threshold of $T_q = 100$ almost always uses all of its budget (with $m_q \cdot p_q$

very close to γ_q), while queries for larger thresholds do not need their full share. We can improve the allocation policy to redistribute these "leftover" budget to improve the accuracy of the queries with the lowest thresholds. We can repeat the process until the leftover is negligible or no query can be improved.

Multi-stage coupon table

Our current prototype uses a single hash-indexed array for storing coupons. Extending this structure to a multi-stage table would offer several benefits. First, hash collisions are inevitable even when the hash table is lightly filled; using multiple tables can provide a query-key pair more chances to insert successfully despite hash collisions. With more memory accesses, we can also allow simultaneously collecting at most 2 or 3 coupons per packet. Second, we can use multiple stages of tables to assign more coupons to each collector, for example by using two tables to implement $m = 64$ coupons per collector.

Memory space

In designing BeauCoup our main concern was supporting multiple queries with limited memory access. If memory size becomes constrained, BeauCoup has two possible ways to address the issue. First, we can voluntarily limit memory access (Γ) below the limit imposed by the hardware; a smaller Γ reduces space requirements, as demonstrated in Figure 4.11. Second, we can implement an eviction mechanism that finds the coupon collectors least likely to succeed; for example, we could look at the number of coupons not yet collected, and how much time has elapsed since the last coupon was collected by this collector.

Distributed Monitoring

Currently, BeauCoup processes traffic at a single switch. To extend BeauCoup to multiple vantage points, we could use multiple switches to run the same random hash functions and a centralized collector to collect all the coupons. Each switch only needs to send packet to the centralized collector when a new coupon is collected. We can minimize the traffic overhead by specifying a small per-packet coupon limit, and de-duplicating the coupons at the switches before sending. Similar to HyperLogLog registers, BeauCoup coupon collector vectors are trivially mergeable.

Security

Some network queries look for adversarial traffic, and an attacker is motivated to craft its attacking traffic to disrupt those queries. As BeauCoup uses random hash functions with random seeds, the attacker cannot predict which packets lead to coupon collection without knowing the seeds. However, with the seeds leaked, the attacker can precisely know which packets trigger a coupon, and thus can deliberately craft traffic to avoid being reported. We therefore should periodically replace the hash seeds and make sure they are not leaked.

Our current prototype uses the CRC-32 family of hash functions with different polynomials, natively available on the programmable switch hardware. CRC-32 is prone to linear correlation, and an attacker may recover the seed when it simultaneously controls the input packets and observes the output coupon activation (performing a *Known Plaintext Attack*). To defend against powerful attackers, a more secure BeauCoup implementation should use cryptographic hash functions. We leave this as future work.

4.7 Related Work

Approximate distinct counting

Plenty of related work discusses how to approximately count distinct elements under limited *memory space*, culminating in the widely-used HyperLogLog[52] distinct counting algorithm.[43] surveyed these prior works, which can be roughly categorized into two flavors: K-Minimum-Value and Distinct Sampling. K-Minimum-Value[8] computes a random hash function over all input elements, and uses the k smallest values observed to infer how many distinct elements exist. Distinct Sampling[58] samples new distinct elements at a small probability, and infers the count by the number of items sampled. We can sample an item out of 2^n distinct items, if we wait for n consecutive leading zeros in the output bits of a random hash function. HyperLogLog[49,52] builds upon the idea of Distinct Sampling but instead partitions the incoming stream into k sub-streams and uses k independent estimators, and outputs the harmonic mean of their estimates. Each estimator records the longest consecutive leading zeros seen from the output bits of a random hash function. We note that our implementation of a m-bit coupon collector is in fact equivalent to the HyperLogLog algorithm using $1/p$ sub-streams, with the $1/p$ estimators each output only one bit. However, we only store the output of first m estimators, truncating the other $1/p - m$ estimators to reduce memory access. Alternatively, a coupon collector can be viewed as a $1/p$-bit Bloom Filter with only one hash function, truncated to the first m bits to reduce memory access. Bloom Filters are originally designed for membership queries but can also be used for approximate distinct counting, as analyzed by Assaf et al.[7].

We also note that universal sketching (UnivMon[78]) can compute many different functions over the input frequency vector, as long as the function is monotonic and bounded by the l_2-norm. In particular, it can compute distinct counting (the l_0-norm). For input length L with A unique items (attributes), UnivMon maintains $log(A)$ different count sketches, and requires $\Gamma = O(log(A))$ memory access per packet in the worst case.

Memory model

Muthukrishnan[91] surveyed several established streaming analysis models, and used an abstraction of maintaining one high-dimensional vector. Each incoming item changes one entry in the vector. The streaming models differ in the changes they can make to items in the vector: *cash register* is addition only, *turnstile* allows addition and subtraction, and *strict turnstile* allows addition and subtraction, yet requires the entries to be always non-negative. Subsequently, queries are made against this high dimensional vector. Our paper falls under the cash register model, for each individual query and sub-streams of the input stream partitioned by the query key.

The cell probe model[71,98,124] is a limited memory access model often used to prove data structure lower bounds. Yao[124] proved that $\lceil \log(S) \rceil$ probes (memory accesses) are necessary to check whether an item exists in a memory array of size S. Larsen et al.[71] discussed other similar lower bounds on how many memory accesses are necessary to solve a certain problem. Usually, in the cell probe model the algorithm is allowed to be adaptive, meaning that it can decide which memory address to look at next based on the content of memory it has already read earlier. We adapt cell probe into stream processing to allow at most Γ memory words to be accessed per packet, while introducing a new notion of sub-constant memory access, requiring each query to access fewer than one memory word per packet *on average*. This model is abstracted from our experience working with high-speed programmable switches, yet we can also identify similar situations in other computing architectures where low latency is required or a memory cache hierarchy exists. For example, a modern CPU has a cache size of a few megabytes. The traditional streaming algorithm model strives to fit an entire data structure (sketch) within this cache size, while our model resembles limiting the number of accesses to external memory or disks, which are slower to access but considerably larger.

Pontarelli et al.[100] proposed a related model where a system has both faster on-chip memory and slower, larger off-chip memory, and can only perform a limited number of off-chip memory accesses per packet. Kim et al.[68] implemented a practical off-chip memory for PISA switches.

Reducing memory access

NitroSketch[77] is a novel technique that reduces memory access for sketching algorithms. The authors identified memory access as one of the most expensive operations when running network measurement tasks on CPUs, and proposed to sample on memory accesses to improve performance. Given a sampling probability p, all the $+1$ updates to the original sketch data structures are changed to $+1/p$ updates with probability p. A smaller p can further reduce memory accesses and accommodate faster packet processing. NitroSketch can be applied to many exising measurement sketches, including Count Sketch[28] and Count-Min Sketch[48], to improve performance without significantly impact accuracy. Compared with the naive approach of sampling packets, NitroSketch achieves better accuracy when given the same amount of memory access.

NitroSketch can be applied to UnivMon and produce a distinct counting algorithm with sub-constant memory access. UnivMon consists of multiple layers each hosting a Count Sketch. For every incoming packet, we first select which UnivMon layers to update using the original UnivMon mechanism, then each layer independently samples the counter updates into its Count Sketch using the NitroSketch mechanism, possibly using different sampling parameters according to the rate of each layer's incoming packets. The combined data structure NitroSketch-Univmon now uses sub-constant average memory access, and the accuracy loss is negligible when we reduce memory access by 50%-75% percent. However, the accuracy for distinct counting suffers greatly when we reduce memory access by 90%-99%, as we have shown in Section 4.5.2.

4.8 Conclusion

We present BeauCoup, a system for simultaneously running many distinct-counting based network monitoring queries while only using a small constant number of memory accesses per packet. This allows BeauCoup to run on on PISA programmable switches, conforming to the tight memory access constraint and using only moderate hardware resources. Evaluation showed BeauCoup uses 4x fewer memory accesses to achieve the same error rate compared with other state-of-the-art measurement sketch. BeauCoup is also the first measurement algorithm running on PISA switches that support dynamic update of measurement queries.

Part II

Real-time, Closed-loop Control in the Switch Data Plane

5 ConQuest: Measuring and mitigating microbursts in real time

In this chapter, we present ConQuest, a framework that allows fine-grained queue measurement entirely in the data plane, enabling the switch to react and mitigate microbursts in real time. Microbursts are transient traffic surges lasting shorter than a few milliseconds;

they cause high queue utilization that often leads to packet loss and delay. ConQuest uses a novel compact data structure to approximately identify the flows that contribute significantly to queue build-up, while still confronting to the memory and computational constraints of the switch data plane. The switch can then run real-time, per-flow advanced queue management that effectively mitigates microbursts and protects non-bursty traffic. Simulations show that ConQuest can identify contributing flows with 90% precision on a 1 ms timescale, using less than 65 KB of memory. Experiments with our Tofino prototype show that ConQuest help reduces flow-completion time by as much as 11%. Additionally, we show how to measure queues in *legacy* devices through link tapping and an off-path switch running ConQuest.

The work in this chapter was completed in collaboration with Shir Landau-Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. It was first presented in ACM CoNEXT 2019[35]. A preliminary short paper was presented in the ACM SIGCOMM 2018 Afternoon Workshop on Self-Driving Networks (SelfDN 2018)[34].

5.1 Introduction

In packet-switched networks, the queues that buffer packets awaiting transmission are fundamental components of the network. Much of the packet losses and delays that occur in the network are caused by backlogs in these queues. Yet, existing network devices offer surprisingly little visibility into the state of the queues, making it difficult to detect, diagnose, and fix performance problems.

Fine-grained, real-time queue monitoring is now possible with the emergence of high-speed commodity programmable switches. By running queue analysis algorithm directly in the switch data plane, we can not only detect queue-related performance anomalies and pinpoint their root cause, but also implement real-time reactions to mitigate their impact. However, in order to achieve line rate, the programmable switch hardware imposes memory and computational constraints, as we discussed in Chapter 2. Thus, we have to carefully design our queue analysis algorithm to follow those constrains, such as limited number of pipeline stages and limited arithmetic operations supported.

Most importantly, in a PISA switch, register memory are partitioned between pipelines (Subsection 2.2.3). Recall that a packet first goes through ingress pipeline processing, then waits in the queue, before going through egress pipeline. The ingress and egress pipeline cannot have access to the same register memory* – since both pipelines are processing packets at high clock rate, concurrent access to the same memory could create a memory hazard.

Due to the concurrent memory access limitation, a data structure cannot be updated both when a packet enters the queue and when it departs. Instead, we can only analyze the queue in one place, when a packet's queuing metadata becomes available. Prior research shows how to work within the PISA constraints to perform fine-grained *logging* of packet bursts[66,112,125]. However, given all of these constraints, designing a data structure that can *analyze* the queue buildup entirely in the data plane—rather than collecting logs for offline analysis—remains an unsolved problem.

*This constraint is likely true for not just PISA but due to the fundamental mismatch between the aggregated speed of the link and the speed of memory.

Contribution

ConQuest uses a novel approximate data structure called *round-robin snapshots* that allows estimating the size of individual flows [†] in the switch queuing buffer, avoiding memory updates both before entering and after departing the queue. ConQuest maintains multiple compact "snapshots" of the queue occupants over time; each packet updates one snapshot and queries multiple past snapshots. We can now analyze each flow's contribution to backlogged queues and detect the root cause of microbursts, *entirely* within the data plane, enabling immediate control actions. This is useful for a wide range of applications, from preventing congestion-related attacks to implementing active queue management (AQM) schemes.

We implement ConQuest using a P4[22] template program, which allows generating switch-specific code to accommodate the different limitations (number of pipeline stages, available register memory writes, etc.) of each hardware target. The implementation is open-sourced on GitHub[31]. We then run a prototype of ConQuest on the Intel Tofino switch[65] to demonstrate a real-world implementation of queue measurement and management in the data plane.

In trace-based simulation experiments, we quantified the benefit of queue measurement in the data plane and characterized how the number and size of snapshots affect measurement accuracy. ConQuest can achieve over 90% precision and recall using less than 65 KB of memory. Meanwhile, using the Tofino-based prototype and a testbed experiment, we

[†]Flows can be defined at various levels of granularity (e.g., five-tuple, source-destination pair, or destination address) depending on the purpose, such as detecting a single bursty TCP connection or an end-host or service receiving large bursts of traffic.

demonstrated the ConQuest-powered real-time, closed-loop control can help improve end-to-end performance.

Finally, we also share our experience implementing queue monitoring for legacy, non-programmable routers, using off-path monitoring technique. Most legacy routers only report coarse-grained queue statistics on a large timescale. We tap multiple links of a legacy device, and feeds the data into a version of ConQuest extended to match the ingress and egress observations of the same packet to perform the same queueing analysis. Fine-grained monitoring of legacy routers enables network operators to troubleshoot performance problems in their network. We use our prototype to analyze queuing in a Cisco CRS router and verify ConQuest's accuracy. We also deployed ConQuest in a campus network and successfully diagnosed queuing anomalies in the border router.

Outline

The chapter is structured as follows. In Section 5.2, we first discuss the important use cases of on-switch queue analysis. Section 5.3 discussed how we designed the ConQuest Round-Robin Snapshots data structure. In Section 5.4, we share our experience implementing ConQuest for switch hardware. In Section 5.5, we present simulation-based evaluation results that quantify ConQuest's accuracy, as well as testbed closed-loop experiments that demonstrate ConQuest can indeed mitigate microbursts and protect other traffic. In Section 5.6, we share our experience implementing queue monitoring for legacy routers using tapped traffic. We also compare ConQuest to related work in Section 5.7. Finally, we discuss some future work and conclude in Section 5.8.

5.2 Queue Measurement Use Cases

While networks typically rely on end-hosts to perform congestion control, fine-grained queue measurements at *switches* are still critical for a wide range of purposes, including:

- **Stopping congestion-related attacks.** In a Shrew attack[69], a few bursty flows (each sent every few seconds, for a short duration) cause a large transient backlog in the queue. Quickly identifying the queue buildup, and the contributing flows, enables rapid mitigation of these attacks.

- **Avoiding conflicting workloads.** Interactions across multiple connections, such as TCP Incast[4,37], can cause sudden queue buildup, leading to high tail latency for big data applications. Identifying the responsible applications enables better scheduling, load balancing, and VM placement decisions in data centers.

- **Optimizing switch configurations.** Queuing parameters, such as weights in active queue management (AQM) schemes like W-RED[94], are notoriously difficult to tune[41]. With a fine-grained understanding of queuing dynamics, network operators can better configure these parameters to the prevailing workload.

- **Deploying new AQM schemes.** While congestion control has long been an area of innovation, deploying new AQM schemes is challenging due to a lack of fine-grained queue measurement support in switches. A data-plane queue measurement primitive would provide the metrics necessary for more sophisticated AQM schemes.

- **Debugging switch implementations.** Queue management in high-speed switches is a complex mechanism, with flow control between multiple queues on different ports. Implementation mistakes by equipment vendors can lead to counter-intuitive phenomena, like high packet loss and delay during periods of low link utilization. These bugs are difficult and time-consuming to detect, let alone diagnose and fix, without better visibility into queuing dynamics.

- **Using switches with shallow buffers.** Cheaper switches with small buffers are sufficient for many networks[5,120]. Finding a way to monitor queues of legacy routers can help network operators decide whether they can adopt shallow-buffer switches without compromising performance. In addition, a data-plane queue-monitoring primitive in new commodity switches can help manage the limited buffer resources to run the network at high utilization.

5.3 ConQuest Data Structure

ConQuest needs a measurement data structure that operates in *real time* (to detect and mitigate even short-lived queue buildup as it forms), at a *fine granularity* (to pin-point individual contributing flows), and with *high accuracy* (to make good decisions). Meeting these requirements is not easy. A 100 Gbps link sends new packets every few nanoseconds, and a transient congestion event may last less than a hundred microseconds[19,130]. To analyze a queue on a small timescale, we cannot rely on packet sampling or coarse-grained statistics such as queue length, as fine-grained information about transient congestion events would be lost with the high sampling rates (as low as 1 in every 30,000 packets[101]) in today's networks. Yet, processing every packet in software would not scale to high link speeds. Instead, our data structure must operate at line rate, within the data plane.

As discussed earlier in Subsection 2.2.3, our data-plane data structure design is subject to a fundamental architecture limitation: we cannot concurrently perform updates at both ends of the queue, meaning, ConQuest cannot increment a counter as it enters the queue and later decrement it when the packet departs. This encouraged us to design a snapshot-based data structure that passively *expires* groups of packets instead of actively deleting data for old departing packets. To estimate flow-level queue occupancy in real time, ConQuest combines results across multiple snapshots, and cleans and reuses expired snapshots in the background.

5.3.1 Contributing Flows in a Queue

For a constant-rate link serving a single FIFO queue, a packet's queuing delay corresponds directly to the length of the queue when it arrives. ConQuest identifies the flows that consume a large portion of the queue and are, therefore, significant contributors to the backlog. Tracking these flows would *seem* to require per-flow counters, updates to the counters on both packet arrival and departure, and identifying the largest counters at any given time. Realizing such a data structure in the data plane is inherently difficult, due to the constraints outlined in Subsection 2.2.3 Fortunately, we do not need to estimate the contribution of all flows all of the time, just *some* flows (i.e., the most significant contributors) *some* of the time (i.e., when we see a packet of that flow and queuing is long).

Querying a flow's own contribution to the queue: For the switch to take corrective action on the flows causing the backlog, we need only identify the contribution of the *current* packet's flow to the queue. More precisely, for each packet, we ask: *while this packet was queued, what fraction of the packets (or bytes) transmitted over the link belonged to its*

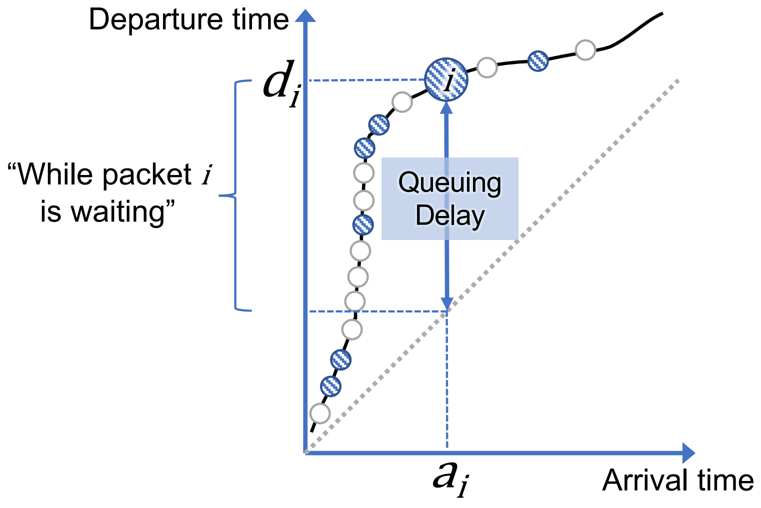


Figure 5.1: Packet departure time (d_i) vs. arrival time (a_i) in a queue. While packet i was queued, three (shaded) packets of the associated flow f_i departed.

own flow? As shown in Figure 5.1, for a packet i with flow ID f_i arriving at time a_i and departing at time d_i, all packets j with departure time $d_j \in [a_i, d_i)$ departed while packet i was waiting in the queue. Some of these packets belong to the same flow as i (i.e., $f_j = f_i$, shaded blue). As an example, in Figure 5.1, packet i was the tenth packet in the queue when it arrived, and three of those packets were from flow f_i.

Each egress pipeline witnesses packets leaving the queue as a stream of (f_i, a_i, d_i) tuples, where flow ID f_i is determined from packet headers and timestamps a_i and d_i are queuing metadata, which are available in the data plane after the packet leaves the queue. We define a queue to be congested when the queuing delay of the packets reaches a threshold of τ. When congestion occurs, ConQuest aims to identify the *contributing flows*, whose packets occupy at least an α fraction of the queue. Or, more formally:

Contributing flow Given a FIFO queue with a congestion threshold τ and a contribution threshold α, when packet i is departing with flow ID f_i and arrival/departure timestamps

a_i, d_i, if $(d_i - a_i) \geq \tau$, *and*:

$$\frac{\left|\{j \mid (a_i \leq d_j < d_i) \ \ \& \ \ (f_j = f_i)\}\right|}{\left|\{j \mid a_i \leq d_j < d_i\}\right|} \geq \alpha \tag{5.1}$$

then f_i is currently a *contributing flow*.

For ease of exposition, we assume that all packets have unit size; however, it is straightforward to extend the definitions to consider packet length.

Accuracy when it matters: Hence, to understand queue backlog, ConQuest needs to report accurate estimates (i) only for the *contributing* flows, rather than the many less significant flows, and (ii) only when the queuing delay is high. This allows ConQuest to use approximation techniques to work within the constraints imposed by PISA switches.

5.3.2 Traffic Snapshots for Bulk Deletion

To determine if a packet is part of a contributing flow, ConQuest maintains information about past packet departures. When packet i departs the queue, ConQuest queries packets from the *past* based on the time range $[a_i, d_i)$, and also inserts the current packet's flow ID and departure timestamp (f_i, d_i) into the data structure to support *future* queries.

The main challenge of performing these operations on PISA switches is to accurately *delete* information about packets whose departure timestamp has become too old to be relevant to any future packet's query. Since packets do not actively delete themselves, we group packets into fixed time-window snapshots of length T based on their departure time, allowing us to passively expire a window of past packets in bulk. Let us choose T=3 for demonstration: in Figure 5.2(a), the rightmost packet (shaded blue) with departure time 0

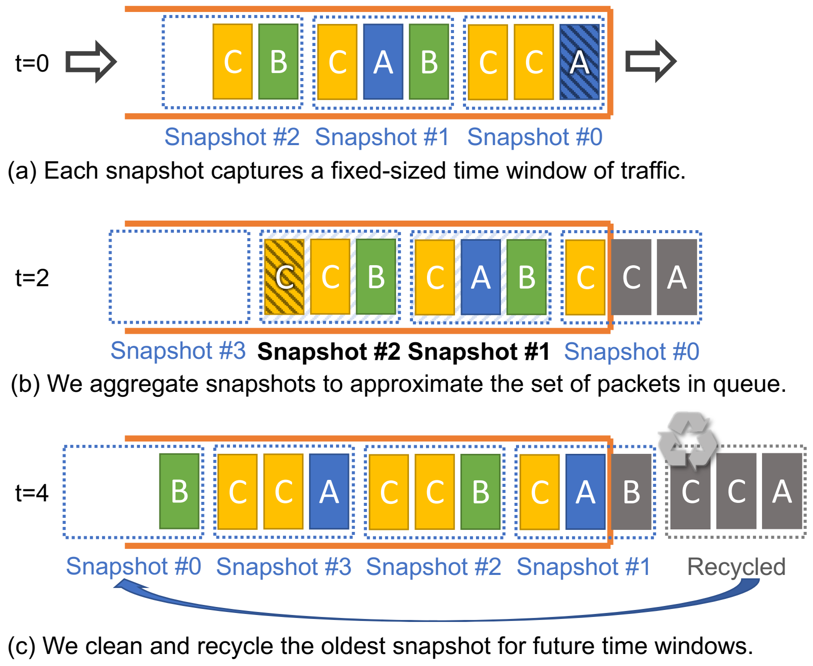

t=0
C
B
C
A
B
C
C
A
Snapshot #2
Snapshot #1
Snapshot #0
(a) Each snapshot captures a fixed-sized time window of traffic.
t=2
C
C
B
C
A
B
C
C
A
Snapshot #3
Snapshot #2
Snapshot #1
Snapshot #0
(b) We aggregate snapshots to approximate the set of packets in queue.
t=4
B
C
C
A
C
C
B
C
A
B
C
C
A
Snapshot #0
Snapshot #3
Snapshot #2
Snapshot #1
Recycled
(c) We clean and recycle the oldest snapshot for future time windows.

Figure 5.2: Time-window snapshots on a queue.

goes into snapshot $\lfloor 0/T \rfloor = 0$. The next packet from flow A (also in blue) departs later at time 4 (as shown in Figure 5.2(c)), thus falling in snapshot $\lfloor 4/T \rfloor = 1$.

For each snapshot, we count the total number of packets for each flow; for example, snapshot #0 in Figure 5.2(a) has one packet for flow A and two packets for flow C. Afterwards, we can query this snapshot to obtain the sizes of flows during this time window. Using snapshots, we can implicitly expire old packets in bulk from the system by no longer querying the oldest snapshot. We can ignore expired snapshots, or better yet, *recycle* them (illustrated in Figure 5.2(c)) as discussed in more detail in Section 5.3.4.

If the number of flows is limited and known beforehand, each snapshot could consist of simple per-flow counters. For a network with a large number of flows, per-flow counters are not feasible. While solutions such as Counter Braids[80] and FlowRadar[75] can record *precise* per-flow counts, they require offline decoding and thus cannot be queried from within the data plane. However, since we care about the *large* contributors to the queue, any *approximate* data structure that supports inserting or incrementing counts and querying flow sizes with reasonable accuracy can achieve our purpose; in our prototype, we use the Count-Min Sketch (CMS)[48] due to its ease of implementation in the data plane. The CMS can estimate flow sizes with a possible overestimation error due to hash collisions; the error bound depends on the selected size of the structure. Note that incrementing the CMS may be easily modified to estimate flow sizes based on either packet count or byte count.

Hence, ConQuest makes two kinds of approximations: (i) when we divide traffic into time windows, the query for a particular time range $[a_i, d_i)$ will be rounded into a query to an approximated range and (ii) the use of sketches can lead to overestimates in the flow counts in each window. We evaluate the effect of both types of error in Section 5.5.1.

5.3.3 Aggregating over Multiple Snapshots

Ideally, to decide if packet i belongs to a contributing flow, we would compute f_i's flow size within the departure time range $[a_i, d_i)$. ConQuest computes an approximate answer by looking at a number of recent time windows that are contained in $[a_i, d_i)$, namely from snapshot $\lceil \frac{a_i}{T} \rceil$ to snapshot $\lfloor \frac{d_i}{T} \rfloor - 1$ (we round towards the more conservative side, which also uses fewer snapshots). Thus, by aggregating the flow size of f_i in the corresponding snapshots, we know approximately how many packets from f_i departed during $[a_i, d_i)$. Since we can only aggregate an integer number of snapshots, our estimate of the queue's content will differ from the actual queue's head and tail, with "rounding error" no more than T on both sides. When the queuing delay $(d_i - a_i)$ is much larger than T, i.e., when the queue is backlogged and, therefore, we are interested in measuring, we have smaller relative error. Aggregating over multiple snapshots allows us to estimate longer queue with more snapshots and shorter queue with fewer snapshots; using only a single snapshot would result in always analyzing a fixed time window, which is less accurate given the varying queue length.

As a concrete example, in Figure 5.2(b), the leftmost packet (shaded yellow) from flow C arrived at a_i=2. This packet will ultimately depart the queue at d_i=8, assuming one packet departs the queue in each time unit. Once we know a_i and d_i in the egress pipeline, the packets of interest are those that departed in the time range $[2, 8)$, i.e., the seven packets shown *inside* the queue in Figure 5.2(b) (other than i itself); out of these packets, there are three packets from flow C (yellow packets). Snapshot #1 recorded one packet for flow C, while Snapshot #2 recorded two packets. By aggregating the two shaded snapshots, #1 and #2, we can get an approximate value 3, i.e., there are around three packets from flow C among the seven packets that departed between time $[2, 8)$.

Besides simple summation, we may also aggregate snapshots differently to compute other metrics in the data plane. This creates more applications for snapshots beyond analyzing congestion. For example, we can detect rapid changes in flow throughput in the data plane, by computing the difference between the flow sizes reported by the two most recent snapshots. This technique would help network operators locate flows which rapidly ramp-up without obeying congestion control. Furthermore, by operating only on packet arrival and departure times, ConQuest can analyze congestion under a range of queuing disciplines. In this paper, we mainly focus our discussion on a link with a single FIFO queue.

The extension to more general queuing disciplines is relatively straightforward, and we leave the technical details to Section 5.3.6.

5.3.4 Cleaning & Reusing Expired Snapshot

ConQuest only needs a constant number of snapshots to analyze a FIFO queue of bounded length served by a constant-rate link. For example, a 20 Mb queue served by a 10 Gbps link would have a maximum queuing delay $\max(d_i - a_i) = 2$ ms. If each snapshot covers a time window of length $T = 1$ ms, ConQuest needs to read from at most two past snapshots. Namely, we can choose time window T based on the total number of snapshots h, such that aggregating all snapshot time windows would approximately cover the entire queue. When a snapshot is no longer useful, we can recycle the snapshot for recording future traffic, as shown in Figure 5.2(c). Since snapshots are rotated on a very small timescale, we cannot rely on the control plane to clean expired snapshots in a timely manner; there is also no straightforward way to batch clean the register memory in data plane. Instead, we clean expired snapshots gradually, one entry at a time.

More generally, ConQuest maintains $\lceil \frac{\max(d_i - a_i)}{T} \rceil$ snapshots for *reading* (i.e., queries), one for *writing* (i.e., for inserting new packets), and one for *cleaning* (i.e., recycling), for a total of $h = \lceil \frac{\max(d_i - a_i)}{T} \rceil + 2$ snapshots. As illustrated in Figure 5.3, the roles of snapshots rotate every T seconds, synchronized with the progress of the time window. Each packet i traverses all h stages, indexing the Count-Min Sketch with its own flow ID f_i in the reading and writing stages, and indexing with a global index for clearing part of the CMS in the cleaning stage. In summary, in handling packet i from flow f_i, ConQuest performs the following operations based on its arrival and departure timestamps a_i, d_i:

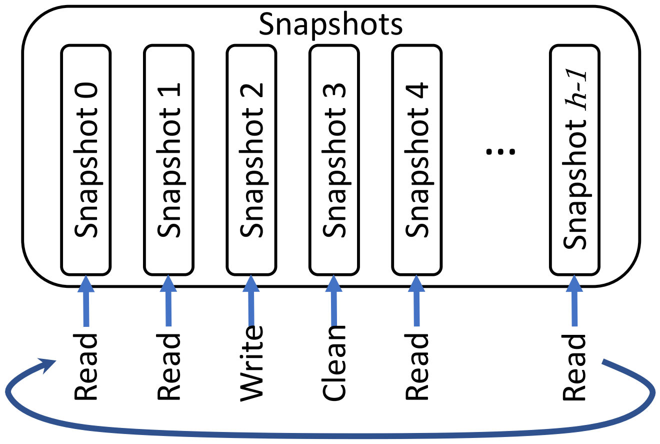


Figure 5.3: Round-Robin between h Snapshots. In any given time window, ConQuest writes into one snapshot, reads many, and cleans one for the next time window. Snapshot roles are rotated every time interval T.

- *Write:* Increment the size of flow f_i in the CMS associated with snapshot $\lfloor \frac{d_i}{T} \rfloor \bmod h$.
- *Read:* Accumulate the estimated size of flow f_i in snapshots $(\lfloor \frac{d_i}{T} \rfloor - 1) \bmod h$, $(\lfloor \frac{d_i}{T} \rfloor - 2) \bmod h, \ldots, \lceil \frac{a_i}{T} \rceil \bmod h$. The number of aggregated snapshots varies, and depends on the time the packet spent in the queue.
- *Clean:* Zero an entry in snapshot $(\lfloor \frac{d_i}{T} \rfloor + 1) \bmod h$. For a CMS with C columns, we maintain a global packet counter *cnt*, and write zero to the $(cnt \bmod C)$-th item in each row.

Using the cleaning technique described above, the CMS is cleaned after C packets—we choose C and T such that there are at least C packet departures in one snapshot time window of length T, say, at 10% link utilization. If this is not the case, i.e., the link is very underutilized such that the number of packets per time window T is smaller than C, the last packet (that departs a now-empty queue) can trigger a report to the control software to clean the snapshot; note that this software can run at a timescale relative to T, which is sig-

	Definition
a_i, d_i	Arrival and departure times of packet i
f_i	Flow identifier for packet i
w_{f_i}	Weight (size) of flow f_i across packets departed during $[a_i, d_i)$
W	Total weight (size) of all flows inserted into snapshots
h	Number of snapshots
R, C	Number of rows & columns in Count-Min Sketch
α	Threshold for identifying a contributing flow
τ	Queuing delay threshold
S	Number of pipeline stages available in switch
M	Number of concurrent register memory accesses per stage supported by switch

Table 5.1: Summary of notations used in this Chapter.

nificantly slower than the timescale of individual packets. Alternatively, if the target switch supports packet generation (such as on the Intel Tofino), when the link is underutilized the data plane can generate the additional packets needed for cleaning the snapshot and filter them before the end of the egress pipeline.

5.3.5 Error Analysis

We now analyze the worst-case estimation error for the ConQuest data structure due to hash collisions, and show that when using h Count-Min Sketches each with R columns and C rows, it achieves $\varepsilon = e/R$ additive error with failure probability $\delta = (h-2)e^{-C}$.

First of all, each snapshot Count-Min Sketch[48] provides $\varepsilon = e/R$ additive error with $\delta_{CMS} = e^{-C}$ failure rate, which means with $(1-\delta)$ probability a query with ground truth flow size w will return an estimate $\hat{w}$ satisfying $w \leq \hat{w} \leq w + \varepsilon W_{CMS}$, with W_{CMS} being the total size of all inserted flows into this CMS.

ConQuest reads from at most $h - 2$ snapshots to aggregate flow size estimates. Since each read has failure probability δ_{CMS}, we can use union bound to bound the probability of having any failure as $(h-2)\delta_{CMS}$. Therefore, the aggregate read's failure probability is $\delta = (h-2)e^{-C}$. When the aggregate read succeeded, the read error produced by each CMS is at most εW_{CMS_j}, and the total additive error for the output is bounded by $\sum_{j=1}^{h-2} \varepsilon \cdot W_{CMS_j} = \varepsilon \cdot W$, where W is the total size of all flows inserted into all $(h-2)$ snapshots. Thus, we show that ConQuest has additive error bound $\varepsilon = \frac{e}{R}$, i.e., returns estimated flow size $\hat{w}_f$ within $w_f \leq \hat{w}_f \leq w_f + \frac{e}{R} W$, with failure probability at most $(h-2)e^{-C}$.

Plugging in the parameters from our hardware prototype ($h = 4$, C=2,R=2048), we have worst-case additive error rate ε=0.0013 with maximum failure probability δ=0.27. We also analyze this error empirically in Figure 5.5.1.

5.3.6 Non-FIFO Queuing Disciplines

So far, we assumed that each link serves a single FIFO queue. In practice, links often use non-FIFO queuing, such as when an outgoing link has multiple FIFO queues (serviced by a scheduler), or even more exotic queuing disciplines. Here we describe some potential future works for utilizing ConQuest for analyzing queuing and congestion in general queuing disciplines.

Contributing flows within a traffic class. Under multiple traffic classes, a link may have one FIFO queue per class, as well as a scheduler (e.g., strict priority or weighted fair queuing). Since ConQuest considers only the packet arrival and departure times, the question "while i was waiting, what fraction of the packets transmitted over the link belonged to its own flow f_i" from Section 5.3.1 is still germane. The answer is useful to assess how much packet *i*'s flow contributes to queuing for its own traffic class, and act on the current packet accordingly. However, unlike the case of single FIFO queue, the *maximum* queuing delay can be large (and, in the worst case, unbounded), under heavy load in higher-priority traffic classes. Instead, we can specify a maximum history to maintain, and answer queries about contributing flows relative to traffic departing during that bounded period.

Contributing flows across all traffic. More generally, high queuing delays for low-priority traffic may stem almost entirely from other, higher-priority flows that receive fast service (i.e., $d_i - a_i$ is small). By querying on the narrow range $[a_i, d_i)$, ConQuest would *not* realize that packets of flow f_i are adversely affecting other (lower-priority) flows. The query for packet *i* would report that few, if any, packets of flow f_i were transmitted while packet *i* was waiting! To analyze contributing flows *across* traffic classes (or across groups of flows with FIFO scheduling), we can slightly modify the definition of a *contributing flow* to enable these significant flows with small delay to recognize the harm they do to other traffic. In particular, ConQuest can maintain an additional register to store the maximum delay (*MaxDelay*) experienced by packets that recently left the queue, and perform a query for packet *i* that considers a larger time range $[d_i - MaxDelay, d_i)$. The value of *MaxDelay* can decay gradually over time, when queuing delays are low.

5.4 P4 Hardware Switch Prototype

We implemented a prototype of ConQuest in P4 on an Intel Tofino[65] switch. We first show how to map ConQuest to different PISA targets and how we automatically generate target-specific P4 code. Then, we describe and implement some of the possible control actions the switch can take based on ConQuest measurements.

5.4.1 Mapping ConQuest to PISA Hardware

Although P4 is a target-independent programming language, different hardware targets may vary significantly from one another in characteristics like the number of pipeline stages, size of memory, number of concurrent actions in each stage, etc. Therefore, even though we designed ConQuest to fit within the PISA processing model, we still need to configure its parameters to fit within individual PISA hardware target's memory and processing capacities.

Automated generation of target-specific P4 code. To facilitate the use of our code on different targets, instead of writing a P4 program, we write a parameterized program that can be instantiated with a range of parameter values. Namely, we implement ConQuest in P4 with inline C-style macros. Once we specify parameters h, R, and C, a compiler automatically generates the expanded P4 code that fits the constraints of the specific hardware target.

The parameterized P4 program is roughly 900 lines, 60% of which are boilerplate code supporting packet parsing, hashing etc, and 40% are ConQuest's snapshot logic. The program first parses the IP and TCP/UDP headers to obtain the 5-tuple as the packet's flow ID. Then, it computes hash functions over the flow ID for reading or writing the Count-Min Sketches. The header parsing and hashing steps are programmable, and can change to use other flow ID definitions (e.g., source-destination pair, destination IP, etc.).

Mapping the logical structures to physical hardware. We now discuss how to map the *logical* structure presented in Section 5.3 to the pipeline stages in a hardware target. We assume ConQuest is only allowed to use S pipeline stages to manipulate snapshots, constrained by the capacity of the hardware target, and further limited by the other duties the switch must perform, such as packet forwarding. Additionally, we denote M as the maximum number of concurrent register accesses the target can support in each stage.

Assume a ConQuest implementation with h snapshots, and with R rows and C columns in each snapshot's CMS. Each CMS uses R register arrays, and reads/updates one entry per array for each packet. We therefore need $h \times R$ register accesses per packet in total in the worst case, which implies a necessity for at least $\lceil \frac{h \times R}{M} \rceil$ stages for memory access. Since each snapshot operates independently, at each stage we can "stack" multiple rows of different snapshots, and perform the read, write, or clean operations concurrently.

After reading the snapshots, we need another $\lceil \log_2(h-2) \rceil$ stages to sum the counts from all snapshots. We require the total number of stages used to manipulate snapshots not exceed the available stages: $\lceil \frac{h \times R}{M} \rceil + \lceil \log_2(h-2) \rceil \leq S$. Additionally, we need a small constant number of stages for pre-processing, such as computing the read/write/clean roles for snapshots and the memory addresses to use in the CMS.

ConQuest needs to fit into the hardware resource constraints of programmable switches while sharing resources with other switch functionality; under these constraints, we would choose the largest possible values of h, R, and C to achieve optimal accuracy. We further discuss the effect of each parameter on accuracy in Section 5.5.1. Furthermore, since arbitrary division is not supported on PISA hardware targets, we implement division and floor operations using bit right-shift, and implement modulus using bit slicing, which are explicitly defined in P4 specification[117]. Consequently, we choose both T and h to be integer powers of 2.

5.4.2 Actions on the Contributing Flows

In this section, we discuss how ConQuest allows the switch data plane to take action on packets based on a flow's contribution to queue backlog. As we discuss later in Section 5.5.2, we have implemented flow-based ECN marking and dropping in our ConQuest prototype to prevent contributing flows from further deteriorating congestion. We discuss these, as well as other potential solutions.

Marking/dropping based on flow weight. When the queue builds up, the data plane can mark the Early Congestion Notification (ECN) bit of the packets; if the queue grows even longer, the switch can go further and start dropping packets. In conventional Random Early Detection (RED)[54] schemes, the packets from different flows are simply dropped (or marked) with the same probability depending on average queue utilization. ConQuest enables the switch to decide actions on packet i from flow f_i, based on the current size of flow f_i in the queue when i arrived, denoted as *flow weight* w_{f_i}. As a basic example, given a threshold w_T, we mark packet from flow f only if $w_f \geq w_T$. This will throttle the heaviest flow in the queue, while leaving small flows intact. We can also probabilistically drop (or mark) the packet with probability $\Pr[\text{ECN}] \propto max(w_f - w_T, 0)$, or with other more sophisticated probability functions such as $\Pr[\text{ECN}] \propto max(w_f - w_T, 0)^2$, inspired by CHOKe[96]. In this way, ConQuest enables fast prototyping of active queue management algorithms that target contributing flows by using probabilistic dropping, based on the individual flow's size in the queue.

Our P4 prototype of ConQuest supports dynamically specifying threshold w_T at run time to achieve threshold-based ECN marking or dropping for contributing flows. We demonstrate the effectiveness of this flow-based ECN approach in Section 5.5.2. The prototype also supports piecewise constant approximation of any ECN marking (or dropping) probability function based on w_f.

Act on future packets. Upon identifying a contributing flow, the switch can feed its ID from the egress pipeline back to the ingress pipeline using packet recirculation. The ingress pipeline may then prevent this flow from exacerbating the imminent queue buildup, by re-routing, rate-limiting, or dropping its packets.

Report flow IDs. Transient congestion is sometimes not caused by individual contributing flows. In some cases, we can identify the cause of the congestion by defining flows at a coarser level of granularity. For example, TCP Incast[37] is caused by many sources sending packets to the same destination simultaneously, and can be accurately captured by defining flows by destination IP address. In other cases, ConQuest can report packets from contributing flows to a software collector for further analysis, such as aggregating the reports to detect hierarchical heavy hitters or other groupings of flows belonging to a single distributed application (e.g., coflows[39,40]). We leave these extensions as future work.

5.5 Evaluation

We evaluate ConQuest using two different setups. We use multi-factor simulation experiments to test the accuracy of ConQuest under different parameter settings. We do so by comparing ConQuest's output to the ground truth found in the simulation. Subsequently, we verify ConQuest's effectiveness in detecting and acting on the flows contributing to a backlogged queue, by evaluating our ConQuest prototype in a real-world testbed. We show that using measurements from ConQuest, the switch can throttle bursty flows to reduce median flow completion time.

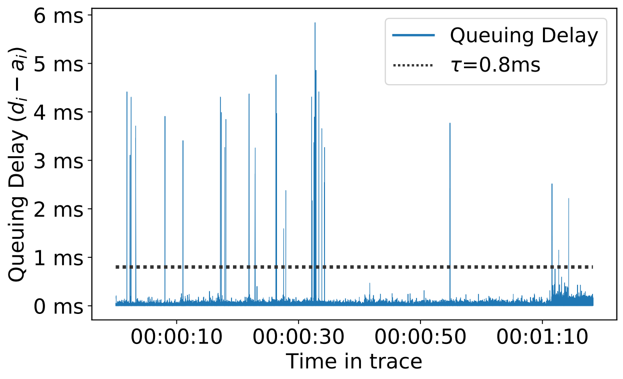


Figure 5.4: Simulated queue buildup on the UW Trace shows low average utilization with occasional bursts.

For consistency across our experiments, we match the link rates of the legacy equipment and use an egress line rate of 10 Gbps in all of our experiments, including the tapping setup in Section 5.6.3. Selecting this link rate affects the timescale of a backlogged queue. On a 10 Gbps link, using a 40 Mbit buffer space (typical in commodity switches) leads to a maximum delay of 4 milliseconds, and a snapshot time window of T=1 ms using 4 snapshots; at 100 Gbps line rate, we would have 400 microseconds maximum delay and ConQuest would run with T=100μs.

5.5.1 Multi-Factor Simulation Experiments

Simulation experiments allows us to freely tune all parameters of ConQuest that practical hardware may not permit, and gives us full detail about the queuing dynamics at any given time. Therefore, we use simulations to evaluate ConQuest's accuracy while changing its parameters.

Dataset and implementation. To simulate queuing delay, we utilize the publicly available University of Wisconsin Data Center Measurement trace *UNI1* (UW trace)[19], by feed-ing the trace through a single FIFO queue with constant 10 Gbps drain rate and unlimited queuing buffer. We use the UW data-center trace in our experiment as it is the only public trace we are aware of that exhibits significant burstiness for simulating queue buildup, while other public traces such as CAIDA are less bursty. Since the original trace is published when links are predominantly 1 Gbps or 100 Mbps, and the trace has an average throughput of only 25.3 Mbps, we replay the trace 50x faster to reach 7.5% average link utilization at 10 Gbps. As seen in Figure 5.4 the queue length exhibits a bursty pattern over time. Similar pattern arises when we calibrate the trace to 3.75% or 15% utilization (replay 25x or 100x). The maximum queue utilization during the replay is around 8 MB (6.4 ms at 10 Gbps).

We simulate the queuing delay and ConQuest snapshots using Python. When a packet i experienced queuing delay $(d_i - a_i)$ greater than τ=0.8 ms (about 1/8 of maximum queue depth observed), ConQuest reads past snapshots and reports an estimated flow size in the queue w_{f_i} for flow f_i when i entered the queue. Flow f_i is flagged as a contributing flow if w_{f_i} exceeds α fraction of the queue length. Note that for FIFO queues, "packets in queue at time a_i" is equivalent to "all packets departed during $[a_i, d_i)$". We also use simulation data to compute the ground truth contributing flows based on actual flow sizes. We first show results for α=1% as a representative threshold, and later show that ConQuest is robust for various choices of α.

We note that ConQuest is answering an imbalanced binary classification problem, as the packets belonging to contributing flows are not half of all packets queried. Therefore, we use Precision and Recall analysis to precisely describe ConQuest's accuracy. *Precision* is defined as the number of packets correctly identified by ConQuest as part of a contributing flow divided by the number of all packets reported by ConQuest. *Recall* is defined as the number of packets correctly identified by ConQuest as part of a contributing flow divided by the ground truth number of packets belonging to contributing flows. As a standard metric for evaluating a binary classifier, Precision and Recall capture how ConQuest trades false positives for false negatives and achieves balanced accuracy.

We define a flow based on the standard 5-tuple (source and destination IP address, protocol, and source and destination port). The UW trace has around $550,000$ distinct flows in total. In our queuing simulation, when the queue is congested, there are on average 63.6 distinct flows in the queue (with 130 flows at 95%-percentile and 200 flows at 99.9%-percentile), out of which there are an average of 3.7 contributing flows (for α=1%).

There are two primary design choices for ConQuest, the snapshot data structure's memory size and the snapshot time window size. Using more memory to construct larger Count-Min Sketch (CMS) data structures reduces collisions and improves accuracy. Using a smaller time window T provides better granularity when approximating the range $[a_i, d_i)$ by lowering the rounding error, at the cost of using more pipeline stages. We evaluate the effect of both on accuracy.

Effect of limited per-snapshot memory. We first evaluate the memory needed to achieve adequate accuracy. For each snapshot, we use a CMS with R=2 rows and vary the number of columns C. When C is small (hence using less memory), CMS suffers from hash collisions and over-estimates the size of flows, reporting more false positives and lowering Precision (but Recall does not change since CMS would not underestimate flow size). Figure 5.5 shows the effect of varying the total number of counters in the CMS on Precision. The Precision plateaus when there are $(R \cdot C)$=32 counters per CMS with diminishing returns

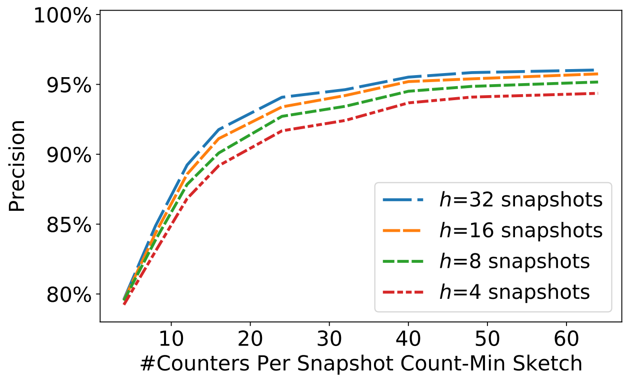


Figure 5.5: Precision vs. snapshot data structure size. Using 24-32 counters per CMS is adequate.

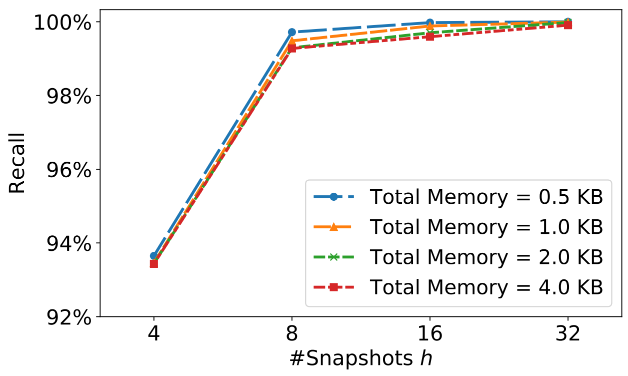


Figure 5.6: Recall vs. number of snapshots. Using 8 snapshots gives sufficiently high Recall.

for additional counters. This aligns with our observation that there are only tens of active flows in the queue during congestion and only very few heavy flows, hence even a small CMS can already distinguish heavy flows from small flows. With enough counters in the CMS, there is practically no hash collision. We note that Recall is influenced more by the number of snapshots and not by CMS size, as shown later in Figure 5.6. In real-world deployments, we should use a larger CMS with more counters if there are more active flows in the queue during congestion. We expect ConQuest to achieve nearly 100% Precision whenever the number of counters in CMS is approximately the number of active flows in queue.

Effect of snapshot time window size. Next, we evaluate the effect of snapshot window granularity on accuracy. Increasing the number of snapshots h (therefore using a shorter time window T) reduces ConQuest's rounding error when computing $\lceil \frac{a_i}{T} \rceil$ and $\lfloor \frac{d_i}{T} \rfloor$. Using fewer snapshots (and larger windows) would cause bursts that departed immediately before a_i to be erroneously included in the $[a_i, d_i)$ range, thus the rounding error would lead to lower Recall. In the worst case, ConQuest can only look at one snapshot and cannot adapt to the change in queuing delay. As shown in Figure 5.6, by aggregating h=4 snapshots we can already achieve 93% Recall, and we have diminishing returns after more than h=8 snapshots. Using more snapshots also slightly improves precision. Note that since the maximum queuing delay in the simulated queue is around 6 ms, we configure $T = (6.4/h)$ ms in all combinations, such that aggregating time window from all snapshots can approximately cover the entire queue. These results show that ConQuest can achieve high accuracy once we use enough memory, with diminishing returns for extra resources. The multiple curves in Figure 5.6 almost overlap, since providing more than enough memory yields negligible difference on Recall, or even slightly decreases Recall; this is because hash collisions lead to over estimations, creating both more false positives and true positives simultaneously.

Flow size estimation error. ConQuest produces an estimate of the size of each flow, not only the largest ones. Such estimations can help network operators analyze the flow size distribution. For example, if there is often only one large flow occupying 90% of the queue during congestion, then it may be sensible to mark or drop the heaviest flow.

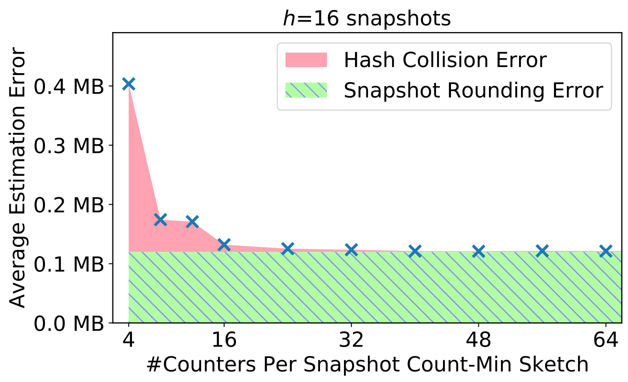


Figure 5.7: With large CMS, the effect of hash collisions becomes negligible; the flow size estimation error is mainly contributed by snapshot rounding.

As we discussed earlier, the estimated flow size reported by ConQuest is only an approximate, and contains two types of errors: a *snapshot rounding error* is caused by reading an integer number of past snapshots, when in reality the queuing delay may not be integer multiples of snapshot time window size; and a *hash collision error* happens when multiple flow IDs encounter hash collisions, causing the CMS to overestimate flow sizes. In Figure 5.7, we show ConQuest's average flow size estimation error when using a different number of counters per snapshot. We further separate the effect of hash collision from snapshot rounding by simulating a special version of ConQuest with ideal per-flow counting in each snapshot. As we can see, the error caused by hash collision diminishes quickly with more counters. With C=4 counters per snapshot the effect of hash collision is prominent; with C=64 counters the average flow size estimation error reduces to 120KB, which is mostly caused by snapshot rounding.

Attributing the estimation error. ConQuest incurs two kinds of error when estimating the size of a flow in the queue: rounding error due to querying integer number of snap-

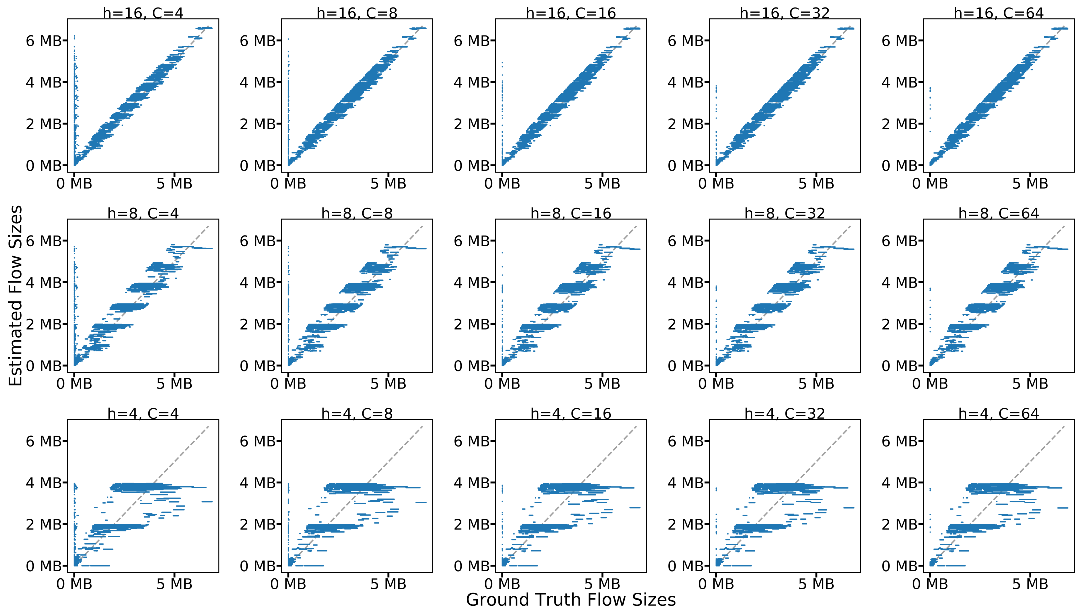


Figure 5.8: Scatter plot of estimated flow size vs. ground truth, using different number of snapshots h and different CMS width C.

shots, and overestimation by Count-Min Sketch due to hash collisions with other flow IDs. To further investigate the two kinds of errors, we draw scatter plots of ground truth flow sizes versus estimated flow sizes reported by ConQuest in our simulation experiments. In Figure 5.8, we first notice that the plots on the lower rows use fewer snapshots, hence a ladder-shaped rounding effect (due to querying integer number of recent snapshots) is prominent, while using more snapshots the estimation can have higher accuracy (closer to $y = x$ line). Meanwhile, the plots to the left use smaller CMS, causing some small flows (with ground truth flow size close to zero) to collide with larger flows, and the overestimation caused by such hash collisions is shown as dots close to the y-axis. When using larger CMS, these hash collisions start to diminish.

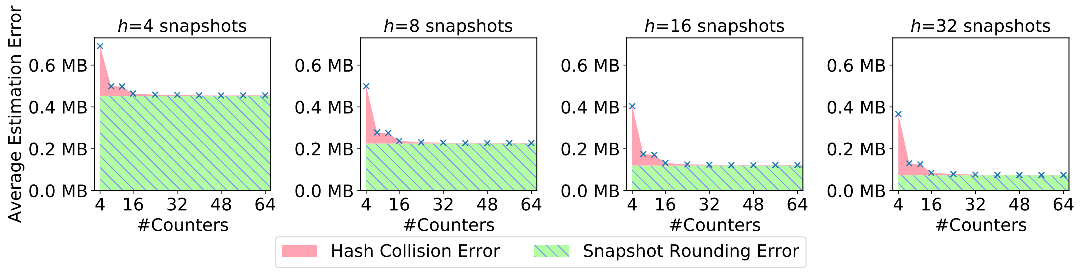


Figure 5.9: Attributing estimation errors to flow ID hash collisions and snapshot rounding errors.

We can quantify the effects of the two kinds of errors by computing the average flow size estimation error under various configurations. In Figure 5.9, we plot the average value of absolute flow-size estimation error under different configurations. We can see that the average estimation error is 120 KB for h=16 snapshots, with R=2, C=16 CMS. When using only h=4 snapshots and R=2, C=16, the average and median estimation error grows to 461 KB and 281 KB respectively. As a reference, under h=4 snapshots (T=1.6 ms), 10 Gbps line rate setup, the total traffic recorded in each time window is 2 MB.

We separate the effect of rounding versus hash collision by simulating a special version of ConQuest that does not use CMS and records exact flow sizes in snapshots, and attribute its error to rounding (plotted in shaded green). As we can see from Figure 5.9, the error caused by hash collisions diminished quickly with more counters in CMS; when ConQuest is running with adequate memory, the estimation errors are mainly caused by snapshot rounding error, and we shall note that such error will not cause significant impact for accurately identifying heavy flows.

Changing contributing flow threshold. We plot the Precision-Recall curve of ConQuest while changing the contributing flow criteria from α=1% to smaller or larger values, while using a fixed number of h=4 snapshots, each with a R=2, C=8 CMS, creating

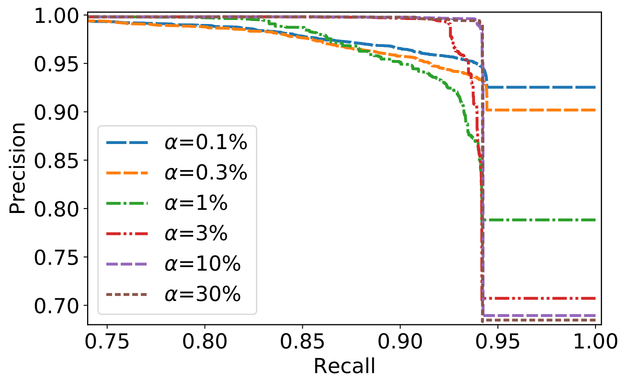


Figure 5.10: Precision-Recall curve for ConQuest under simulation, for different contributing flow thresholds.

both collision and rounding error. A smaller α value requires ConQuest to detect heavy flows earlier and report more flows, which is more challenging than reporting only one or two heavy flows when α is very large. As shown in Figure 5.10, ConQuest can consistently achieve over 90% Precision and Recall while we change α from 0.1% to 30%. In Section 5.6.3 we perform the same Precision-Recall analysis in a real world prototype and show similar results.

5.5.2 Closed-Loop Testbed Experiment

We build a testbed experiment to demonstrate ConQuest's potential to analyze and proactively manage queue buildup, by implementing a simple ConQuest-enabled Active Queue Management scheme running at line rate within the data plane. We show that our ConQuest prototype can fit into the hardware constraints of a first-generation PISA programmable switch, and furthermore, we can identify and throttle the flows contributing to congestion to reduce the workload Flow Completion Time. Although the flow-based queue management scheme we implement is primitive and far from optimal, it already demonstrated the potential of future works on building novel AQM schemes using programmable data planes.

Dataset and testbed setup. Our testbed consists of two servers and one Intel Tofino[65] Wedge-100 switch. Each server is a 20-core, 100G-memory blade server running Ubuntu 18.04, with Linux kernel version 4.15, and equipped with a Mellanox ConnectX-5 EN 100GbE NIC. We connect both servers to the programmable switch: the *sender* is connected via a 100 Gbps link, while the *receiver* is connected via a 10 Gbps *bottleneck* link. This setup is designed to cause queuing: although one TCP flow can still manage to detect the bottleneck rate correctly, the queue fills up quickly if there are many concurrent TCP flows competing for bandwidth.

We generate workload using the flow-size distribution of a data center Web rack, which are mostly small-to-medium size flows, from the Facebook Data Center Measurement study[101]. The mean and median flow size are 38.8 KB and 1.44 KB, respectively. We schedule flows using exponentially distributed inter-arrival time, choosing λ=155μs (sending one flow every 155μs on average) to achieve 20% average link utilization (2 Gbps) on the 10 Gbps bottleneck link. The sender sends one million workload flows per experiment. The sender also periodically starts one bursty flow per second, and we vary the size of bursty flows between experiments.

All flows are independently managed by the Linux kernel built-in TCP congestion control mechanism, set to New Reno, Cubic, Vegas, DCTCP, or BBR. We bind multiple IP addresses to the receiver and use the `SO_REUSEADDR` option on the sender to allow sending more than 65,536 concurrent TCP flows. We verified that the servers are not CPU contended.

We tune the baseline, flow-indiscriminative ECN setup to achieve optimal performance, by configuring the switch to mark the ECN bit for outgoing packets when the switch's queue length exceeds the threshold 4096 packets (corresponding to less than 5 ms of queuing delay). We found this to be the minimum queue size needed to allow a single TCP connection to reach line rate, based on the minimum congestion window size required under the Round-Trip Time on our testbed. We also configure the switch to drop packets when the queue length exceeds 16384 packets, although this rarely happens when the sender honors the ECN marking. Our switch is an output-buffered device and we use a single FIFO queue.

Our prototype implementation of ConQuest has h=4 snapshots, each having a CMS with R=2 rows, due to hardware pipeline constraints. We choose C=2048 columns, the largest we can effectively clean within the snapshot clean phase. in total, they use $(R \cdot C \cdot h) \cdot 4$ bytes=65 KB of register memory, less than 1% of the total available on the hardware. The prototype also utilized small fractions of several other hardware resources: it computes R=2 hash functions and performs $(R \cdot h)$=8 memory accesses, both less than 20% of total capacity. This leaves plenty of room for other switch functionality to be run in parallel with ConQuest.

We configure our prototype to rotate the snapshots every T=2 ms, such that aggregating all snapshots will approximately cover the entire queue. We configure congestion threshold to τ=2 ms. When transient congestion is identified, i.e., queuing delay exceeds τ, ConQuest will start marking ECN for flows with $w_f > w_T$=512 packets, corresponding to approximately α=25%.

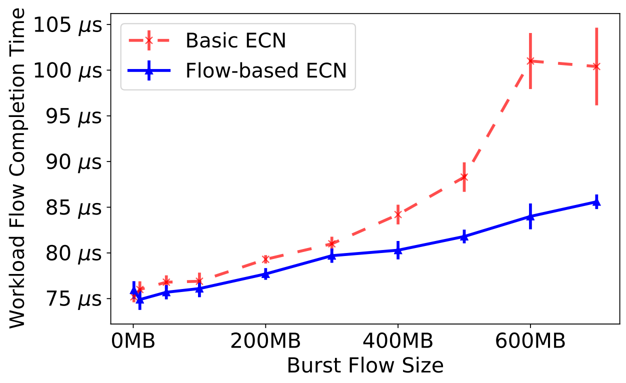


Figure 5.11: We can configure ConQuest to selectively mark ECN for only the burst flow and not for small flows, leading to better FCT for small flows workload.

Flow Completion Time comparison. Figure 5.11 shows the median Flow Completion Time (FCT) of the workload flows, i.e., the small and medium flows generated using the Facebook web rack distribution, with respect to the bytes sent on the bursty flow. Under conventional queue management, when a bursty flow fills up the queue and triggers ECN marking, some packets from small flows inevitably get marked; this is true even if probabilistic (RED-style) ECN marking is used. This leads to a growing FCT for the workload flows as the burst flow becomes larger. Instead, ConQuest only marks packets from the bursty flow, while allowing all small flows to quickly finish sending without being throttled. As a result, the bursty flow has less impact on the FCT of small flows. We have also observed slight improvements in 90%-percentile FCT; however, 99%- and 99.9%-percentile FCT deteriorates since the largest workload flows are also penalized by ConQuest.

The result shown in Figure 5.11 was performed using New Reno congestion control (ConQuest reduced median FCT by 6.9%); experiment results for other congestion control algorithms are similar (ConQuest reduced median FCT by: 7.3% for Cubic, 1.6% for Vegas,

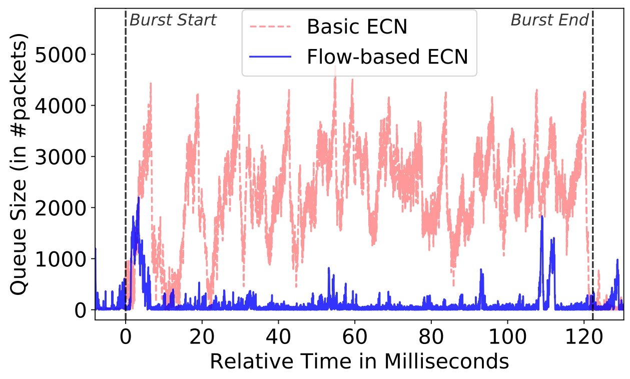


Figure 5.12: By only marking ECN on contributing flow's packets, ConQuest can effectively throttle the bursty flow and maintain a shorter queue length.

0.56% for DCTCP), except for BBR. Notably, BBR does not honor ECN marking (or dropping) as the congestion control signal and performs its own queue buffer utilization probing, and therefore ConQuest cannot affect its sending rate.

Figure 5.12 shows the queue length statistics we collected from the programmable switch, while a 50 MB burst flow interacts with the small flow workload. Under regular ECN settings the queue is quickly filled up to the ECN marking threshold, at which point all flows are subject to congestion control and queue length oscillates, until the bursty flow finishes sending. In contrast, when we enable flow-based ECN by querying ConQuest, the bursty flow is quickly throttled and the queue remains short during the entire sending period. Note that the queue length has many short spikes when ConQuest is enabled; this is because multiple short flows can all quickly finish without being marked or dropped.

Our results show that it is possible to improve network performance at the switch level with flow-level queuing analysis and queue buildup mitigation. Although the AQM scheme we implement with ConQuest in the testbed is very primitive, it already demonstrates the potential performance improvements of using programmable switches to implement sophisticated AQM algorithms. We note that in practical networks such as wide-area / carrier networks, merely adding an ECN flag cannot throttle flows immediately and effectively; we need to take other actions on the packets of contributing flows, such as dropping, rerouting, or scheduling them in a separate queue.

5.6 ConQuest for Legacy Devices

Legacy (i.e., non-programmable) routers are not designed for precise queuing analysis. They often only support polling the total queue length statistics at a coarse time interval, providing no insight into which flows occupy the queue. Existing networks are not going to replace legacy routers with programmable switches overnight. Yet advanced fine-grained queue monitoring techniques are necessary today, both for debugging existing devices and for understanding the buffer capacity needed to support their operational workload. This is especially true in carrier networks, where the upgrade cycles for network equipment are longer and network operators cannot perform measurements at end hosts. Therefore, network operators have been looking for ways to use an advanced programmable switch as a plug-in debugging tool, to measure and analyze queuing on legacy routers in their network wherever problems arise.

We propose a novel way to use ConQuest as a tool for *selectively* monitoring one legacy router, *temporarily*, in a non-intrusive manner, by tapping its existing ingress and egress links and using a programmable switch to process the tapped traffic. With one programmable switch at hand, network operators can debug any legacy device in the network, gaining on-demand visibility into its queuing dynamics and congestion in real time, without having to replace the device with a programmable one. Tapping is often readily available at the physical layer (split-fiber), or as a monitoring capability provided by the equipment vendor.

We deployed our extended prototype of ConQuest in two different settings: tapping into a border router in a campus network[‡], and tapping into a carrier-grade router in an ISP testbed.

At Princeton University, our campus network operator had identified one border router that occasionally suffers from massive packet drops under low average link utilization. We suspect transient congestion is taking place, however existing diagnostic tools only report queue buffer utilization (alongside other metrics) at minute-level granularity, without apparent anomaly. We helped our campus network operator to use a programmable switch running ConQuest to tap and analyze this border router's ingress and egress traffic, and successfully located the cause of the drops: a performance monitoring tool that failed to schedule throughput tests in series (as claimed), creating incast from multiple senders across Internet2. The queuing delay oscillates wildly from empty to full, and there are 4-5 contributing flows in the queue, all for active throughput testing. Here we see that passive monitoring powered by ConQuest was able to diagnose the performance problems caused (ironically!) by an active performance monitoring tool.

At AT&T Labs, we use a Cisco carrier-grade router to process synthetic bursty traffic, and let ConQuest analyze tapped traffic to verify its accuracy and robustness under the tapping setup. We present the details of our testbed and the results in Section 5.6.3.

[‡]The diagnostic process involves no access to personal data and has been approved by our university's Institutional Review Board.

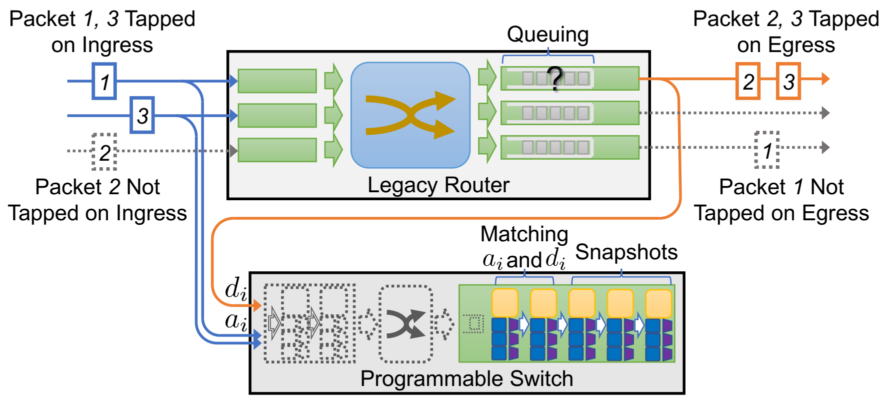

Packet 1, 3 Tapped on Ingress
Queuing
Packet 2, 3 Tapped on Egress
1
3
2
?
2
3
1
Packet 2 Not Tapped on Ingress
Legacy Router
Packet 1 Not Tapped on Egress
Matching a_i and d_i
Snapshots
d_i
a_i
Programmable Switch

Figure 5.13: Using a PISA switch to analyze queuing in a legacy router, by tapping ingress and egress links.

5.6.1 Tap Multiple Links of Legacy Router

Figure 5.13 illustrates the setup for using a PISA switch to monitor queuing in a legacy router. We tap a subset of the legacy router's ingress and egress ports and mirror their traffic to ports with a common packet-processing pipeline in the PISA switch. Ideally, we would like to tap all ingress and egress links; however, this may be impractical due to cost or tapping link availability; nevertheless we can analyze the legacy router's queuing efficiently even by tapping only a subset of the links, as we discussed in Section 5.6.2.

The PISA switch records the arrival timestamp (a_i) of a packet when it appears in a tapped ingress link, and records the departure timestamp (d_i) of a packet when it appears in a tapped egress link. To recover accurate and unbiased queuing delay ($d_i - a_i$), the tapping links for the ingress and egress ports should have equal and constant latency.

5.6.2 Match Ingress and Egress Packets

To recover the queuing delay $(d_i - a_i)$ experienced by packet i, we would like to match the appearances of packet i in both the tapped ingress link and the tapped egress link. There are several technical details to consider:

Hash digest. We hash a packet's header fields to obtain a hash digest for efficiently matching a packet's appearance on a tapped egress link with its earlier appearance on the ingress link. For IPv4 packets, we can examine the IPID field. For TCP packets, we can also observe the sequence/acknowledgement number to distinguish individual packets within the same flow. Matching IPv6/UDP packets is more challenging and we omit the implementation details.

Storage and timeout. The digest and arrival time a_i from the tapped ingress are first inserted to a hash-indexed array. Later, when a copy of the same packet appears on the egress tapping link at time d_i, we compute the same digest to fetch a_i from the array and compute the queuing delay $(d_i - a_i)$, and also clear the entry from the array.

Not seen on egress: Some packets that appear on a tapped ingress link may be dropped or routed to an untapped egress port; therefore, they never appear on the tapped egress link. For example, in Figure 5.13, packet *1* was tapped on an ingress link, but was routed to an egress port not being tapped. These packets would fill up the register array that would never be matched and are therefore useless. We solve this issue by implicitly expiring entries: we allow an entry to be evicted from the array once its arrival timestamp has aged more than the maximum possible queuing delay, and can thus be considered expired.

Not seen on ingress. A packet that arrives at the tapped egress link may not have a corresponding digest and arrival timestamp stored in memory. This may occur if the packet

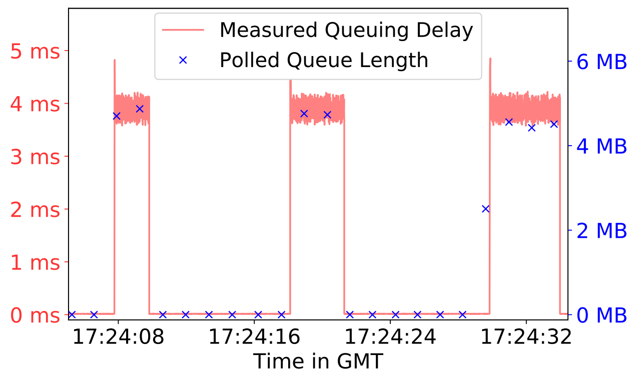


Figure 5.14: Queuing delay measured by our prototype matches the ground-truth queue-depth reported by the legacy switch.

entered the router from an untapped ingress link, or a failed insertion to the array due to hash collision. For example, in Figure 5.13 packet *2* comes in from an untapped ingress port, but appears on the tapped egress port, so d_2 is known but a_2 is unknown. We cannot query if these packets belong to a contributing flow; however, we still insert them into the current snapshot using the departure timestamp, since they contributed to the congestion at our monitored egress port.

5.6.3 Validation with a Cisco CRS Router

We built a tapping testbed to evaluate if ConQuest can accurately diagnose queuing in a legacy switch. We use a programmable Barefoot Tofino Wedge-100 switch ("programmable switch") to tap 3 ingress links and 1 egress link of a Cisco CRS 16-Slot Single-Shelf System ("legacy router"), all running at 10 Gbps. We use an IXIA traffic generator to feed traffic into the 3 ingress ports; the legacy switch is configured to route all traffic to the same egress port, into a single FIFO queue.

We extend the ConQuest P4 program to match ingress and egress packets to calculate queuing delay, and compute ground truth statistics for evaluation purpose. The combined P4 program has around $1,200$ lines of code. We also validated that our extended ConQuest prototype correctly estimates the queuing delay in a legacy switch, by comparing the queuing delay estimated by ConQuest with the ground truth queue length reported by the legacy switch. We send periodically bursty traffic into the legacy switch to create queuing. As shown in Figure 5.14, the queuing delay computed by our P4 program nicely aligns with the queue length reported by the legacy switch (divided by line rate 10 Gbps).

We configure the IXIA traffic generator to send 10 flows as background workload, ranging from 1 Mbps to 5 Gbps, and send 3 periodically bursty flows, with varying burst duration from 50 μs to 5 ms. Note that the number of flows are limited by our need to maintain ground truth per-flow counters for evaluation purpose, and ConQuest itself can work with a large number of flows, as demonstrated in Section 5.5. Since the prototype has h=4 snapshots and the maximum observed queuing delay is around 4 ms, we configure the ConQuest to use snapshot interval T=1 ms. Each snapshot uses a R=2 row, C=64 column Count-Min Sketch, which is large enough to not cause any hash collision. The congestion reporting threshold is set to τ=0.5 ms, about 1/8 of maximum queue length, similar to previous experiments.

We compared the reported packets ConQuest identified as part of a contributing flow to the per-packet ground truths we fetched from IXIA and our extended P4 prototype, and computed Precision and Recall metrics. Figure 5.15 shows the Precision-Recall curve, under different contributing flow criteria α. ConQuest consistently achieves over 90% Precision and Recall when identifying contributing flows, for α ranging from 0.1% to 30%. Al-

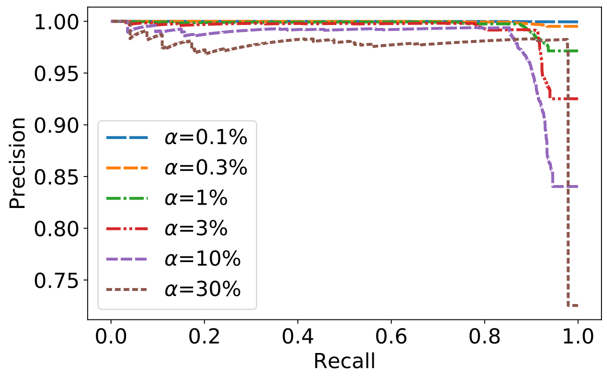


Figure 5.15: Precision-Recall curve for ConQuest's P4 prototype under tapping setup.

though we cannot support taking immediate corrective action, ConQuest still provides us unprecedented visibility and high accuracy for analyzing queuing in a tapped legacy router.

5.7 Related Work

Measuring queue buildups

Zhang et al.[130] implemented a high-precision microburst measurement framework in data-center networks, by polling multiple switches' queue depth counter at high frequency, and analyzing duration and inter-arrival time of microbursts. However, the system provides limited insight into the contents of the queue, such as which flows contributed to a microburst or the flow size distribution during the queue buildup. Several recent papers use programmable switches for fine-grain logging of traffic in the data plane. SpeedLight[125] is a general system for recording synchronized traffic statistics across multiple switches for offline analysis, including analyzing queuing dynamics. BurstRadar[66] can log packets in a ring buffer at a single switch during queue buildup for offline analysis. *Flow[112] compresses the packet logs before exporting the measurement data to reduce overhead on the remote hosts. Speedlight, BurstRadar and *Flow all provide fine-grained measurement data for *offline* analysis, but cannot identify or act on contributing flows directly in data plane. Meanwhile, HPCC[76] measured switch queue length to improve end-host based congestion control.

Data-center traffic management

In recent years there has been much work on alleviating congestion in data centers. For example, load-balancing schemes like Presto[63], DRILL[57] and CONGA[3] disperse the offered load over multiple paths, without addressing the *root cause* of queue buildup. In contrast, ConQuest enables the switches to identify and target individual flows contributing to backlogged queues. Meanwhile, data-center transport protocols such as NDP[60] and Homa[89] reduce queuing delay at switches, but they typically assume the end-host network stack (e.g., tenant VM) participates *honestly* in the protocol, or require enforcement by the underlying hypervisor or NIC. Fastpass[99] offers a centralized traffic orchestration approach for preventing queue buildup, by centrally allocating the capacity of network links to individual senders. Meanwhile, ConQuest does not impose any additional mechanisms or overheads on the end-host network stack, hypervisor, or NIC. This is especially critical for transit and enterprise networks that do not have control over the end hosts.

Fair queuing

Sharma et al.[108] proposed an approximate per-flow fair queuing mechanism using programmable switches, which reduces the bursty flow's impact on other traffic. Instead of enforcing fairness among all flows, ConQuest identifies individual flows contributing to queue backlogs, and therefore enables acting directly on those flows.

Estimating FIFO queue state

Queue Inference Engine, by Larson[72] and later improved in[21], is an algorithm to analyze the queuing state of a FIFO queue, with random arrivals following a Poisson process. QIE only uses departure timestamps as input, and can infer queuing delay based on observing consecutive departures ("busy periods"). Instead, ConQuest calculates the exact queuing delay for each packet using queuing metadata, and its goal is to analyze the heavy flows in the queue. ConQuest can be used under arbitrary packet arrival time distributions, such as microbursts.

Sliding window query

Our data structure contributes to a body of theoretical work on streaming algorithms on sliding time windows. For example, several works[1,17,24,126] propose algorithms for set membership or heavy-hitter queries on a *fixed-size* sliding window. In contrast, our work deals with a *dynamic* query window $[a_i, d_i)$, which varies across the packets in the stream. As such, the window sizes of future queries are unknown when a packet enters the data structure. ConQuest addresses this challenge by reading from a variable number of time-window snapshots. Basat et al.[14] has explored a similar dynamic window query problem and proposed advanced data structures that run on general purpose computers. To the best of our knowledge, ConQuest is the first solution to be implemented within the resource constraints of programmable switch hardware.

5.8 Conclusion

We present ConQuest, a framework for real-time, fine-grained queue analysis in the switch data plane. ConQuest reports which flows contribute to the queue buildup, and enables direct per-packet action in the data plane. We implement a ConQuest prototype on a programmable switch using only 65 KB of register memory. Testbed evaluation demonstrates ConQuest can effectively identify the contributing flows, and enable the switch to throttle them.

In addition, we propose a novel way to use ConQuest to monitor queuing in legacy network switches. We deployed this setup in both a carrier and a campus network to measure queuing and diagnose performance problems in legacy devices—as a first step in demonstrating the benefits of data-plane queue measurement to network operators.

6 AHAB: Enforce hierarchical fairness via closed-loop adjustment

In this chapter, we present AHAB, a per-user fair bandwidth enforcer running on programmable switches based on closed-loop iterative refinement. Network operators want to enforce bandwidth fairness without solely relying on congestion control running on end-user devices; however, traditional software-based schedulers struggle to achieve high throughput and low latency. Meanwhile, in edge networks, the number of users far exceeds the number of queues supported by today's switch hardware; even accurately tracking per-user sending rates may become too memory-intensive. AHAB tracks each user's approximate traffic rate and compares it against a bandwidth limit, which is iteratively updated via a real-time feedback loop to achieve max-min fairness across users. Using a novel sketch data structure, AHAB avoids storing per-user state, and therefore scales to thousands of slices and millions of users. Furthermore, AHAB supports network slicing, where each slice has a guaranteed share of the bandwidth that can be scavenged by other slices when under-utilized. Evaluation shows AHAB can achieve fair bandwidth allocation within 3.1ms, 13x faster than prior data-plane hierarchical schedulers.

The work in this chapter was completed in collaboration with Robert MacDavid and Jennifer Rexford. It was first presented in IEEE INFOCOM 2023[82] and will appear in the IEEE/ACM Transactions on Networking.

6.1 Introduction

Fair bandwidth allocation between users is an important goal for network operators, since a minority of users demanding too much bandwidth should not negatively affect other users' quality of service. Yet, leaving bandwidth allocation entirely to congestion control running on end hosts may lead to unfair allocation between different congestion control algorithms. Fair bandwidth allocation is, therefore, a necessary function of the core network. As modern networks scale to higher speed and more users, implementing per-user fair bandwidth allocation becomes increasingly more challenging.

Network slicing is a network feature that allows an operator to divide its network resources into many virtualized networks. Slicing enables operators to rapidly create new service offerings for different markets, while achieving performance isolation and quality-of-service guarantees between different slices. To support slicing, the network needs to implement both *intra-slice* fairness where different users within the same slice gets a fair share of the slice's bandwidth, as well as *inter-slice* fairness where each slice gets its share of bandwidth proportional to its specified weight. Meanwhile, the idle capacity from underutilized slices must also be fairly distributed to other over-subscribed slices.

One real-life example of a sliced network is the mobile access network. As IoT and 5G becomes prevalent, we face scalability challenges in implementing fairness. A base station may serve 100-1000 user devices, which belong to different classes of services (IoT, smartphones, mobile broadband, first responders, etc.) and have different usage patterns. Each slice (class of service) gets its guaranteed share of bandwidth; when a slice has few active users, its unused bandwidth can be distributed to users in other slices. Meanwhile, we want different users within the same slice to fairly share the limited physical-layer bandwidth: every user in the same slice should be allocated the same maximum bandwidth limit, which should be increased or decreased in real time based on both the number of active users in the slice and the total bandwidth allocated to the slice.

The slice-based fairness paradigm also exists in other scenarios. A data-center network operator may slice its network capacity into multiple classes of service (free tier, spot instances, enterprise customers, etc.) and allocate bandwidth fairly between different tenants within the same slice. Likewise, a network-layer DDoS mitigation mechanism might slice the network to serve different websites, and fairly allocate the bandwidth between all (po-

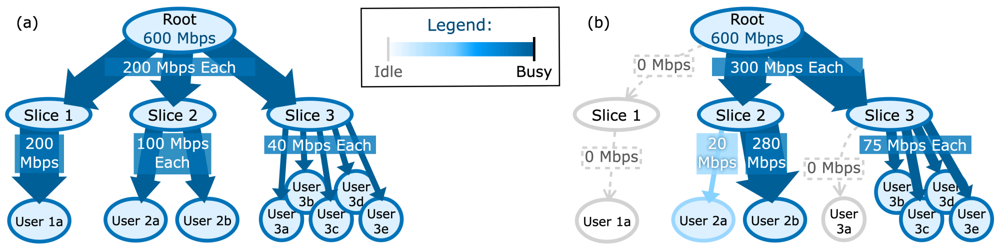


Figure 6.1: In (a), all links are fully utilized. In (b), Users 1a, 2a, and 3a do not need their fair share and the surplus is redistributed.

tentially malicious) clients visiting a particular website. We illustrate an example two-layer hierarchy in Figure 6.1, where many users (mobile devices, virtual machines, or clients) are grouped into slices, and different slices divide the total bandwidth equally. As user bandwidth demand changes constantly, the fair allocation also changes.

In all of these example use cases, the number of users within each network slice (from thousands to millions) far exceeds the number of hardware queues available on today's networking hardware, which commonly supports 8-32 queues per port. In today's mobile network, client rate-limiting and scheduling are sometimes implemented as a virtual network function running on server CPUs[62]. Such a setup supports versatile scheduling policies, yet it requires many CPU cores to serve high-speed traffic and often adds latency and jitter to the traffic. Meanwhile, maintaining ultra-low latency for latency-sensitive applications is one of the most important features in 5G and next-generation 6G networks, which already achieves sub-10ms end-to-end latency[113,121].

The emergence of high-speed programmable network devices had enabled implementing Active Queue Management (AQM) algorithms directly in the switch data plane[108,116,128,129]. Although recent works[81,106] have offloaded many mobile core network functionalities onto programmable switches, traffic scheduling is a notable exception. To the best of our knowledge, no existing work has attempted to offload scalable slice-based fair bandwidth allocation to high-speed programmable switches. Cebinae[127] enforces long-term fair bandwidth allocation but takes seconds to converge. HCSFQ[129] supports slice-based fair bandwidth allocation but requires per-user memory to monitor each user's sending rate; this not only adds control-plane overhead for adding and removing users, but also leads to scalability challenges given the limited amount of memory in the data plane.

There are two main challenges for running fair bandwidth allocation directly within the data plane of high-speed programmable switches. Firstly, the available memory is insufficient for maintaining per-user state. We therefore need to use approximate data structures, whose memory footprint scales sub-linearly with the number of users (as discussed in § 6.4). Secondly, we can only perform a limited set of arithmetic operations in the data plane. We use lookup tables to implement approximated multiplication and division, which is then used for calculating linear interpolation. This enables us to implement real-time, closed-loop iterative update for the per-user bandwidth limit (as discussed in § 6.5). Finally, without using separate queues for each user, we enforce per-user bandwidth limits via probabilistic packet dropping, achieving *approximate* fair bandwidth allocation.

Contribution

We present Approximate Hierarchical Allocation of Bandwidth (AHAB), a hierarchical per-user bandwidth limit enforcer directly implemented in the data plane of programmable switch hardware. AHAB dynamically adjusts the per-user bandwidth limit for each slice in real time, calculated using max-min fairness with the bandwidth demand of all users across all slices. The novelty of AHAB can be summarized as follows:

- **Scalability:** By using a novel approximate data structure, AHAB avoids maintaining per-user state in data-plane memory, thereby supporting millions of simultaneous users.
- **Fast Convergence:** When user traffic changes, AHAB's interpolation-based iterative bandwidth limit update converges to fair bandwidth allocation within 3.1ms, 13x faster than prior work[129].
- **Precise Enforcement:** We use probabilistic dropping to precisely enforce bandwidth limits. This allows users to steadily send at the fair rate observing the bandwidth limit, without requiring hardware queues to pace packets as needed by prior work.
- **One-stop Bandwidth Allocation:** AHAB supports an arbitrary number of hierarchy levels. Therefore, a single instance of AHAB in the core network can rate-limit traffic correctly to adhere to all downstream bandwidth bottlenecks. This is highly useful when downstream devices do not support sophisticated scheduling policies (e.g., legacy routers or thin Wi-Fi access points), or when the network operator is unable to arbitrarily adjust device configurations, possibly because the core and downstream networks are managed between different administrative entities (e.g., MVNOs and wireless carriers).

Outline

The chapter is structured as follows. Section 6.2 defines the hierarchical fair bandwidth allocation problem. Section 6.3 presents an overview of AHAB's division of labor between control and data plane. Section 6.4 discusses how AHAB overcomes the scalability challenge by avoiding per-user memory using a customized approximate data structure, while Section 6.5 describes how AHAB approximately calculates an interpolation-based bandwidth limit update given the arithmetic constraints in the data plane. Evaluation in Section 6.6 demonstrate that AHAB converges to a fair bandwidth allocation quickly within 5 ms, achieving both fairness and throughput stability. Finally, we discuss related work in Section 6.7 and conclude in Section 6.8.

6.2 Hierarchical Fair Bandwidth Allocation

AHAB needs to allocate a network slice's available bandwidth fairly between all users in different slices based on max-min fairness. In this section, we present the same problem definition as in earlier works [18,55,110,129]. For simplicity of discussion, for now we assume all users and slices in our system have equal weight, although it is trivial to add weights and allocate bandwidth proportionally.

6.2.1 Motivating Example

In Figure 6.1, we illustrate a two-level scheduling hierarchy. The root has a total downlink capacity of 600Mbps and serves three slices. There are 1, 2, and 5 users in each slice, respectively.

Figure 6.1(a) depicts the "busy" scenario where all users are downloading as fast as possible; the total bandwidth of 600Mbps is equally divided into 200Mbps per slice. The sole user in the first slice gets 200Mbps bandwidth, the users in the second slice both get 100Mbps, while the five users in the last slice each get 40Mbps.

Symbol	Definition
C_n	Bandwidth capacity allocated to slice n
D_n	Total bandwidth demand of slice n
T_n	Per-user bandwidth limit of slice n
R_m	Bandwidth demand of user $m \in child(n)$
T_{low}, T_{mid}, T_{hi}	Candidate bandwidth limit for next epoch
$f(T_*)$	Hypothetical total sending if the limit was T_*

Table 6.1: Summary of notations used in this Chapter.

When user bandwidth demand changes, the fair allocation also changes. In Figure 6.1(b) we show a scenario where user 1a and 3a leave the network (e.g., powered off) and user 2a has a low bandwidth demand (e.g., audio streaming); all other users are still downloading as fast as possible. In this case, slice 1 does not use any bandwidth, so the total bandwidth is equally divided to the other two slices (300Mbps each). User 2b can use all the remaining bandwidth in slice 2, which is 280Mbps. Users 3b-3e each can use 75Mbps of fair share.

Hierarchical fair bandwidth allocation achieve fairness in both levels: busy users in the same slice get the same bandwidth, after considering the underutilized users in the same slice; different busy slices also get the same aggregated bandwidth, after re-allocating the unused bandwidth from underutilized slices.

6.2.2 Max-min Fairness Allocation

Now we formally define the fair hierarchical bandwidth allocation based on max-min fairness and work-conserving scheduling.

Let us first denote a slice n's total bandwidth capacity as C_n, and its set of users as $child(n)$ (its "children" in the scheduling hierarchy). Each user $m \in child(n)$ has a bandwidth demand R_m, which is the maximum bandwidth it would like to consume if no bandwidth limit is enforced. We can therefore calculate slice n's total bandwidth demand as the sum of

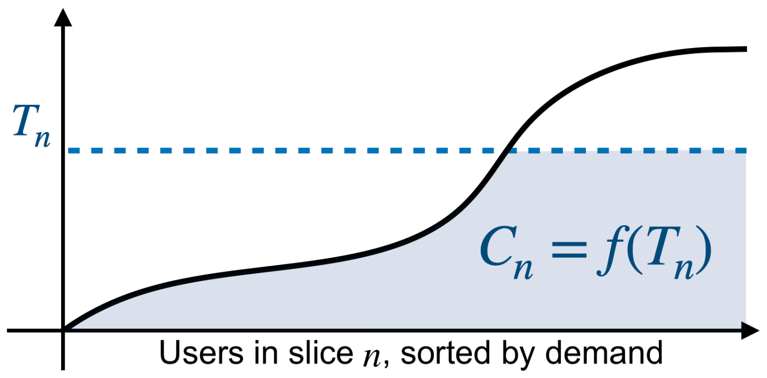


Figure 6.2: Limit T_n is enforced against those users in slice n whose demand exceeds than T_n, so that the total bandwidth consumed is exactly C_n.

all user's demand:

$$D_n = \sum_{m \in child(n)} R_m, \tag{6.1}$$

and we call slice n underutilized if $D_n \leq C_n$. In this case, a bandwidth limit is unnecessary as the slice's capacity is greater than the total demand of all users.

However, when a slice is busy, the demand exceeds capacity and we need to impose a per-user bandwidth limit, such that the *actual* total bandwidth usage of this slice equals to its capacity. Based on max-min fairness, we can define the per-user bandwidth limit as

$$T_n = \arg\max_{T} \sum_{m \in child(n)} \min(T, R_m) \leq C_n. \tag{6.2}$$

Enforcing this bandwidth limit would have no effect for users requiring less bandwidth ($R_m < T_n$) and only affects the users using more bandwidth. We can prove that such a limit T_n always exist for a busy node n, and the resulting enforcement achieves the unique max-min fairness allocation between users. For completeness, we define $T_n = \infty$ for under-utilized slices.

In Figure 6.2, we illustrate the relationship between the bandwidth limit T_n and the total bandwidth used by users in slice n. The demands $\{R_m \mid m \in child(n)\}$, when sorted from least to greatest, form a demand curve. After enforcing per-user limit T_n, each user uses $\min(T_n, R_m)$, and the total bandwidth used by all user is $\sum_{m \in child(n)} \min(R_m, T_n)$ which is represented the size of the shaded area, under the intersection of demand curve R_m and the horizontal line of limit T_n. This area increases as we increase T_n, and the ideal limit T_n means the shaded area has size exactly equal to the available bandwidth capacity C_n of this slice. Thus, our goal for finding the right bandwidth limit T_n under max-min fairness is equivalent to finding the right "horizontal cut" across the demand curve.

Note that the fair allocation is not fixed: As the user's bandwidth demand constantly changes, it is necessary to recalculate the fair allocation and update C_n and T_n. Calculating the hierarchical fair bandwidth allocation requires a two-step process: aggregating the demands upwards, then allocating the capacity downwards. First, we calculate the sum of all slice's demand, $D = \sum_n D_n$, which represents the total bandwidth demand at the root of the entire scheduling hierarchy. We then allocate the total capacity C at the root of the scheduling hierarchy for different slices. When the total demand exceeds capacity ($D > C$), we can allocate per-slice capacity C_n for each slice, using exactly the same max-min fairness principle as in Equation 6.2. Subsequently, using C_n, we can obtain the per-user rate limit T_n for each slice. When these limits are enforced for every user, we implement max-min fair bandwidth allocation across the entire scheduling hierarchy, and the total bandwidth used by all slices will equal to the root's total capacity.

The discussion here focused on a two-level scheduling hierarchy; the same definition applies to more levels.

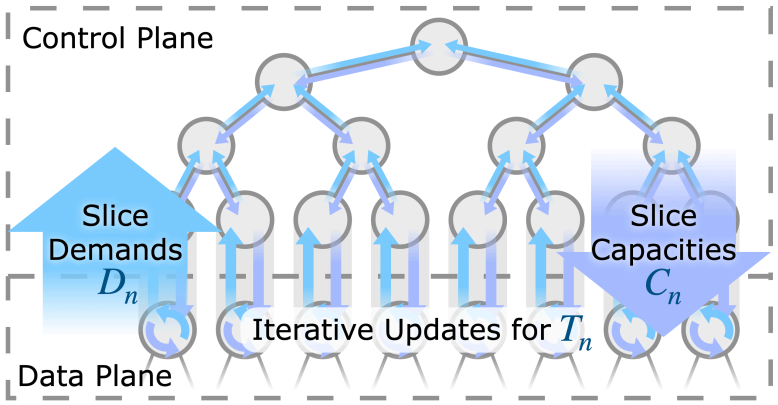


Figure 6.3: The control plane maintains inter-slice fairness by periodically reading each slice bandwidth demands D_n and writing fair capacities C_n; the data plane keeps intra-slice fairness by iteratively updating bandwidth limits T_n.

6.3 AHAB System Overview

Figure 6.3 illustrates the basic design of AHAB. At a high level, we split the bandwidth allocation process into a fast-reacting data plane component and a more sophisticated control-plane component for hierarchical updates.

6.3.1 Data Plane: Intra-slice Fairness.

To quickly react to changes in individual user's traffic, AHAB calculates iterative updates for the per-user bandwidth limit T_n fully within the data plane, using approximated linear regression. This allows the intra-slice bandwidth allocation to converge within milliseconds after a user starts or stops sending, much faster than updating using the switch control plane.

Since it is impossible to perfectly predict the constantly changing per-user traffic demand, AHAB splits the traffic into very small time epochs (on the order of milliseconds) and uses the demand distribution in the past epoch as a prediction for the next epoch. At the end of each epoch, we use this demand distribution (as illustrated in Figure 6.2) to iteratively refine the per-user bandwidth limit T_n, such that the total bandwidth used by all users in the slice will be approximately equal to the capacity C_n.

6.3.2 Control Plane: Inter-slice Fairness.

The control plane is responsible for implementing fair allocation for all the other layers of the scheduling hierarchy beyond the slice level. It periodically reads the per-slice total bandwidth demand D_n (by reading registers populated by the data plane), and calculate the aggregate demand at higher level of the hierarchy. Then, it calculates the bandwidth allocation according to max-min fairness (§6.2.2) starting from the root, until it obtains the per-slice fair allocation capacity C_n. These values are then written to registers that will be read by the data plane.

The periodic update happens every 10-20ms, mostly due to communication bottleneck (reading demands and writing capacities). Thanks to statistical multiplexing, the aggregated bandwidth demand of different slices changes on a longer timescale. Therefore, the slightly slower update of capacities has little impact on maintaining intra-slice fairness. We also note that when all slices are busy, their fair allocation is constant; the allocated capacities only changes when some slices are underutilized.

6.4 Scaling Beyond Memory Limits

In this section, we discuss how AHAB overcomes the scalability challenge imposed by hardware memory size limits. We first discuss how the bandwidth limit T_n is enforced on each user using their estimated sending rates. Subsequently, we show how AHAB avoids allocating per-user memory, using a novel approximate data structure that combines Count-Min Sketch with Low-Pass Filters to estimate per-user sending rates. Finally, we discuss how we share one approximate data structure across all slices using weight-based normalization.

6.4.1 Enforcing Bandwidth Limits

For the entire scheduling hierarchy to achieve bandwidth fairness, we must properly enforce bandwidth limit T_n on all users. Naively, we can allocate one queue per user and assign the bandwidth limit as the queue's drain rate. However, the number of users (thousands to millions) far exceeds the number of queues available in hardware switches (8-32 queues per port). Instead, as proposed by CSFQ[115] and AFD[95], we can enforce bandwidth limits using active queue management, or more specifically probabilistic dropping, as long as we know the user's sending rate. This approach does not require a traffic scheduler, and can be performed even if the switch has only a single queue.

For a user m in slice n with bandwidth limit T_n and sending rate R_m, we can enforce the bandwidth limit T_n by dropping its packets with probability

$$\Pr[drop] = 1 - \min\left(1, \frac{T_n}{R_m}\right) \tag{6.3}$$

as described in CSFQ[115] and AFD[95]. If a user uses less than the limit T_n, no packet will be dropped; otherwise, after probabilistic dropping the user's remaining packets will use bandwidth equal to T_n.

We also observe that probabilistic dropping is unfriendly for TCP flows, as TCP achieves low goodput if we perform probabilistic dropping whenever its instantaneous sending rate exceeds T_n. This is partly because TCP congestion control slows down severely upon two consecutive packet drops, by reducing its congestion window back to slow-start; meanwhile, randomized packet dropping will quite often produce consecutive packet drops.

Therefore, we specifically optimize the rate-limiting for TCP flows by using *periodic* instead of *probabilistic* dropping, and drop one packet approximately every $T_n \cdot const$ bytes, corresponding to the desired congestion window size for achieving goodput equal to T_n. We also adapt various TCP shaping techniques discussed in Nimble[116], such as adding Early Congestion Notification (ECN) flags. This way, much fewer TCP packets are dropped compared to non-TCP flows; well-behaving TCP flows enjoy relatively steady goodput while AHAB can enforce the bandwidth limit T_n effectively. We leave the open question of how to optimally enforce bandwidth limits on TCP flows with various congestion control algorithms as a future work.

6.4.2 Avoiding Per-user Memory

Knowing a user's sending rate R_m is vital for correctly enforcing the bandwidth limit. As discussed in CSFQ[115] and HCSFQ[129], asking the sender of all traffic to attach their traffic rate to each packet is an easy yet unrealistic solution, as the sender might belong to a different administrative entity and may not honestly report the rate. Therefore, AHAB needs to measure each user's sending rate directly.

Recent works[116,129] in queue scheduling within high-speed programmable switches rely on using the onboard memory to maintain per-user sending rate statistics, by allocating one traffic counter per user. However, programmable switches only have a limited amount of onboard memory in the data plane, limiting its scalability. At any given time, a core network switch may be servicing millions of users across thousands of base stations, making it infeasible to store any per-user state in memory, not to mention the hassle of keeping the memory allocation up-to-date when users constantly join or leave the network.

Instead, we build a customized memory-efficient approximate data structure to track per-user sending rate, by combining two techniques: Low-Pass Filters (LPF) and Count-Min Sketch (CMS)[48]:

- The LPF is a self-decaying counter, available as an advanced feature of the Tofino switch hardware. If we add value x at time t to a LPF with previous value v_0 and last update time t_0, its new value becomes

$$v = x + v_0 e^{-(t-t_0)/\tau}, \tag{6.4}$$

  where τ is its decay time constant. As discussed in[115], if we aggregate the packet sizes of a single user's traffic in a LPF, the LPF will report an exponentially-decayed moving sum of recent packet sizes, which is proportional to a good estimate of the user's instantaneous sending rate.

- The CMS[48] is an approximate data structure that answers frequency queries, using r rows of hash-indexed arrays each having c counters. Given an "insertion" with a size and a user ID, we find one location per row by applying r different random hash functions over its ID, and increment the counters at those location by the size; when querying the total size of a particular ID, we find the same r locations and report the minimum of the r counters.

When used to estimate size of flows in traffic, CMS is good at reporting heavy flows, as it never underestimates flow sizes. However, a vanilla CMS can only track the total number of bytes sent by a user since the CMS is initialized, not the user's instantaneous sending rate. Although it is possible to run multiple instances of CMS in a round-robin fashion to query moving-window flow rates, as discussed earlier in Chapter 5, such an arrangement adds complexity and requires 2x-4x more memory.

AHAB combines CMS with LPF by replacing individual counters in the CMS structure with LPF counters. When inserting a packet with its size and user ID, we apply the r hash functions over the user ID to locate one LPF counter per row, and add the packet size to these r LPF counters. When querying the instantaneous sending rate of the same user ID, we read the LPF counters in the same r locations, and use the minimum across their reported rate as the estimated sending rate of this user. This allows us to estimate per-user sending rate without the need to allocate per-user memory.

We note that the combined CMS-LPF estimator retains the following additive-error guarantee:

Theorem 6.4.1. *Let R_i be user i's sending rate reported by an ideal LPF counter, and $\sum_m R_m$ be the total sending rate across all users, again reported by ideal per-user LPF counters. When querying a CMS-LPF estimator of size $r \times c$, the estimated sending rate $\tilde{R}_i$ satisfies $\tilde{R}_i \geq R_i$ and*

$$\Pr\left[\tilde{R}_i \leq R_i + \varepsilon \sum_m R_m\right] \leq \delta, \tag{6.5}$$

with $\varepsilon = e/r$ and $\delta = e^{-c}$.

Proof. Note that CMS is a linear sum in the ID dimension, where each counter is the sum of a random subset of user IDs. Meanwhile, LPF is a linear sum of packet sizes in the time dimension, where the reported rate is the inner product of past packet sizes and a time-decaying constant function. The two linear operators are interchangeable. Thus, we can naturally derive the error bound using the additive error property of an unmodified r-row, c-column CMS. □

The CMS-LPF also inherits the no-underestimation guarantee from CMS. Therefore, in our context of enforcing bandwidth limit:

1. A user's traffic rate is never underestimated, ensuring that a user exceeding bandwidth limit will always be rate-limited.
2. With a small bounded probability, a user's traffic rate may be overestimated and appear higher than the limit, resulting in rate-limiting. We can limit this probability by adjusting the memory size of CMS-LPF.

6.4.3 Sharing One Rate Estimator Across Slices

Naively, AHAB would allocate one CMS-LPF estimator for each slice. However, due to the natural skewness of traffic, not all slices will have lots of "heavy" users sending at high rates. Some slices may be underutilized and have no heavy user at all, and the memory dedicated for their estimators is wasted.

Instead, we share a single CMS-LPF estimator across all slices. We can then exploit statistical multiplexing, as the heavy users and busy slices are now effectively using the unused memory sacrificed by the underutilized slices with no heavy user.

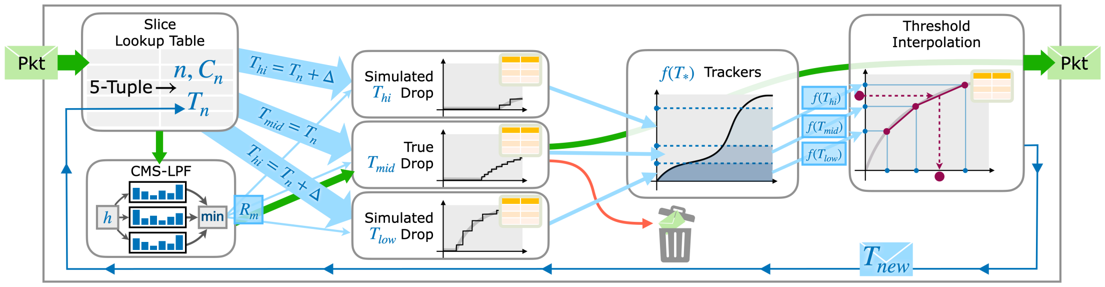


Figure 6.4: When a packet arrives, AHAB first maps it to a slice n and estimates its user's sending rate R_m using the CMS-LPF estimator (§6.4.2), then uses probabilistic dropping to enforce slice n's bandwidth limit T_n (§6.5.1). AHAB also maintains two bandwidth limit candidates T_{low}, T_{hi} and tracks the hypothetical total bandwidth usage$f(\cdot)$ (§6.5.2), which is used to derive a more accurate bandwidth limit T_{new} via approximated linear interpolation (§6.5.3).

However, we note that CMS provides an additive error guarantee, meaning that the error of each user's estimated rate is of similar magnitude regardless of the true sending rate of the user. This is not a problem for intra-slice comparison, as we only care about enforcing bandwidth limits for heavy users and can safely ignore the underutilized users. Yet, different slices may have vastly different bandwidth allocations. If two slices of capacity 100Mbps and 10Gbps naively share the same CMS-LPF structure, the 10Gbps slice will dominate; "small" users of 200Mbps in the heavy slice will overwhelm the CMS while the "heavy" users of 30Mbps in the small slice become a rounding error.

To ensure the estimation error is scaled proportionally with the bandwidth of different slices, we perform *pre-update normalization*: before packet sizes are fed into the CMS-LPF, we scale the packet size inversely proportional to the weight of their parent slice. This guarantees that we can track heavy users in each slice effectively, with the estimation error proportional to its particular slice-level sending rate, regardless of how slow or fast other slices are sending. Subsequently, when we enforce per-user bandwidth limits, the estimated sending rates we read from CMS-LPF are also compared to scaled versions of bandwidth limits.

6.5 Approximate Arithmetic in the Data Plane

To achieve line-rate packet processing and low forwarding latency, high-speed programmable switches like Intel Tofino support a limited set of arithmetic operations, and we can only perform a constant number of computational steps per packet. Thus, it is infeasible to exactly track the bandwidth demands, precisely calculate the fair allocation, or accurately compute per-user drop probabilities.

Figure 6.4 shows an overview of the AHAB data plane, where we use *approximate* arithmetic heavily to implement probabilistic dropping and interpolation-based iterative update to the bandwidth limit. We also note that the approximate arithmetic techniques presented here are widely applicable to other applications running in programmable switches, beyond fair bandwidth allocation.

6.5.1 Approximated Probabilistic Dropping

For a given user, we obtain its estimated sending rate R_m from the CMS-LPF estimator and compare it against the per-user bandwidth limit T_n of its slice. For non-TCP traffic, we need to enforce the bandwidth limit by dropping the packet with probability $1 - \frac{T_n}{R_m}$.

Since we cannot calculate exact division in the data plane, we perform approximate arithmetic using a TCAM lookup table, similar to the approximate multiplication technique used in Nimble[116].

The first step of approximate division is to truncate the numerator T_n and denominator R_m in binary form. We focus on the highest b-bits of the denominator starting from the leading 1. For example, let $b = 5$, if R_m is `0b0011010011010`, the leading b-bit number is j=`11010`. This means we can approximate $R_m \approx$ `0b11010` $\times 2^6$. We also look at the same bits in the numerator: if T_n is `0b0001111110100`, we extract the b-bit number i=`01111` and approximate it as $T_n \approx$ `0b01111` $\times 2^6$. Note that, since the denominator is always larger than the numerator ($R_m > T_n$), we will not miss any 1 in the higher bits in the nominator.

We now observe that the fraction $\frac{T_n}{R_m}$ can be approximated using these b-bit numbers, as

$$\frac{\texttt{0b0001111110100}}{\texttt{0b0011010011010}} \approx \frac{\texttt{0b01111}}{\texttt{0b11010}} \approx 0.576. \tag{6.6}$$

More formally, we use

$$\frac{I}{J} \approx \frac{i \cdot 2^N}{j \cdot 2^N} \tag{6.7}$$

to approximate division, with i and j both being b-bit numbers. We build an approximate division lookup table that maps (i, j) to an approximation of i/j. After picking the appropriate b-bit substrings i, j of the input numerator and denominator, the final step of approximate division is simply look up (i, j) in the table.

Since i and j are truncated from I and J, they both have a one-sided bias compared with the original. To reduce the error bias of lookup table entries, the entries are computed as $\frac{i+0.5}{j+0.5}$ instead of $\frac{i}{j}$, which changes the $x\%$ worst-case one-sided error to $x/2\%$ two-sided error.

Here we make two observations. First, the most-significant 1-bit in J always end up in j, so we only need to generate lookup table entries for half of the possible js starting with 1. Second, notice that the numerator is always smaller than the denominator in our use case, lookup table entries do not need to be installed for any division results greater than 1, further reducing the number of entries. For example, for $b = 5$ bits, we only need to generate entries for j =`0b10000` to j =`0b11111`, and for each j we generate $j + 1$ rules for $0 \leq i \leq j$. In total, the number of rules in the lookup table is

$$\sum_{j=2^{(b-1)}}^{(2^b)-1} (j+1) = \sum_{j=16}^{31} j + 1 = 392. \tag{6.8}$$

This table achieves an approximation error of 8.2% on average. With $b = 6$, the lookup table grows to 1552 entries and we can reduce the error to 4.6%. One such lookup table only costs approximately 9KB of data-plane memory.

The final step of implementing probabilistic dropping is to sample a number uniformly at random between $[0, 1]$ using the random number generator, and comparing it with the approximated division. We drop a packet if

$$U(0,1) > \frac{T_n}{R_m} \approx \frac{i + 0.5}{j + 0.5}. \tag{6.9}$$

Separately, for TCP traffic, we need to perform periodic rather than probabilistic dropping. We maintain a hash-indexed array of "count-down" counters and map each TCP flow to a counter using its 5-tuple. When a TCP packet arrives and the estimated rate exceeds the bandwidth limit T_n, we check its corresponding counter: if the counter is already zero, we drop this packet and reset the countdown counter to $T_n \cdot \mathit{const}$; otherwise, we spare the packet from being dropped and subtract its size from the counter. We also use another set of counters to add ECN marking periodically. This way, TCP flows can quickly converge to the desired rate limit and enjoy steady goodput.

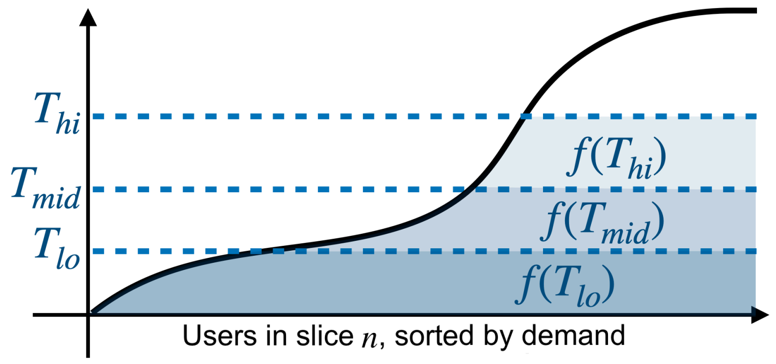


Figure 6.5: Relationship between bandwidth limit candidates T_{low}, T_{mid}, T_{hi} and total bandwidth consumed by all users $f(T_{low}), f(T_{mid}), f(T_{hi})$.

6.5.2 Tracking the Bandwidth Demand

For a given slice, it is infeasible for switches to track its entire bandwidth demand curve (shown in Figure 6.2) representing all user's sending rates, which we can neither store nor sort. However, we can track the actual bandwidth used by all users $f(T_n)$, which is a function of the currently-enforced bandwidth limit T_n and represented by the shaded area under the demand curve intersected with T_n. To get $f(T_n)$, we simply need to use a LPF to track the size of all packets that are not dropped.

Still, comparing $f(T_n)$ with C_n only tells us whether we are over- or under-utilizing the capacity C_n, i.e., whether we should increase or decrease T_n. This does not say much about what is the ideal limit or how much should we change T_n.

Adjusting T_n using a fixed step size or fixed proportion is problematic: if the steps are too large, we cannot precisely converge to the exact allocation. Yet, small step sizes mean we need to wait for many iterations before converging, when the fair rate changes dramatically.

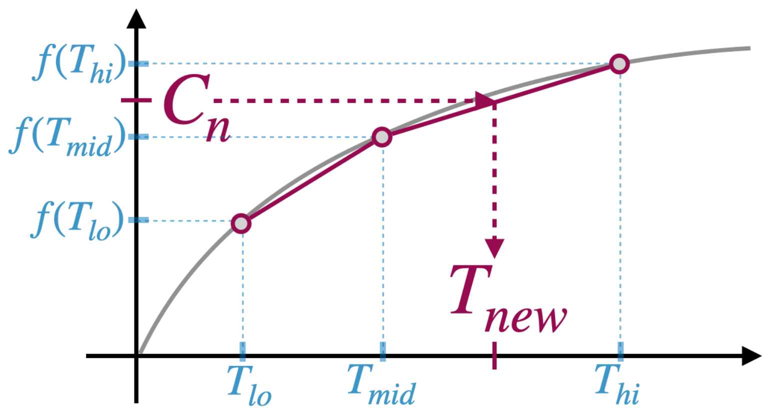


Figure 6.6: Finding new bandwidth limit using interpolation. When we plot $f(T_*)$, x-axis in this figure refers to the bandwidth limit T_* (y-axis in Fig. 6.5), while y-axis in this figure refers to the area under line T_* in Fig. 6.5.

To better analyze how to update T_n, we further specify two *candidate* bandwidth limits, a lower candidate

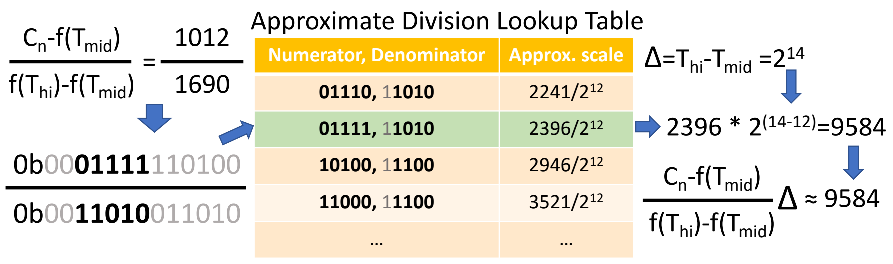
$$T_{low} = T_n - \Delta \tag{6.10}$$

and a higher candidate

$$T_{hi} = T_n + \Delta \tag{6.11}$$

where Δ is the maximum step-size we want to change T_n. In practice, we choose $T_{low} \approx 0.5 \cdot T_n$ and $T_{hi} \approx 1.5 \cdot T_n$. From now on, we also refer to $T_{mid} = 1.0 \cdot T_n$ as the middle candidate.

We now track two more hypothetical total transmitted bandwidth $f(T_{low})$ and $f(T_{hi})$, by generating two hypothetical probabilistic dropping decisions in addition to the real dropping decision. Using the same lookup table technique discussed in §6.5.1, we approximately calculate $\frac{T_{hi}}{R_m}$ and $\frac{T_{lo}}{R_m}$ and track the packets that are hypothetically not dropped under T_{low} or T_{hi} respectively. As illustrated in Figure 6.5, $f(T_{low})$, $f(T_{mid})$, and $f(T_{hi})$ are the shaded area under the demand curve intersecting with different horizontal lines.


$\frac{C_n - f(T_{mid})}{f(T_{hi}) - f(T_{mid})} = \frac{1012}{1690}$
0b0001111110100
0b0011010011010
Approximate Division Lookup Table
$\Delta = T_{hi} - T_{mid} = 2^{14}$
$2396 * 2^{(14-12)} = 9584$
$\frac{C_n - f(T_{mid})}{f(T_{hi}) - f(T_{mid})} \Delta \approx 9584$

Numerator, Denominator	Approx. scale
01110, 11010	$2241/2^{12}$
01111, 11010	$2396/2^{12}$
10100, 11100	$2946/2^{12}$
11000, 11100	$3521/2^{12}$
…	…

Figure 6.7: We use a lookup table to implement approximate linear interpolation. We first match on the highest binary bits of numerator and denominator to get a scaled division result, then multiply Δ via bit shifting for the final result.

Figure 6.6 plots the monotonically-increasing function f. The optimal bandwidth limit $\tilde{T}$ satisfies $f(\tilde{T}) = C_n$, thus we need to calculate a new limit T_{new} that is as close to $\tilde{T}$ as possible.

6.5.3 Update Bandwidth Limit via Approximate Interpolation

Instead of using a fixed step size, we can adjust the bandwidth limit much more accurately using linear interpolation, given the three candidate points on the f curve.

Let us first assume the ideal bandwidth limit lies between the lower and higher candidate points, i.e., $f(T_{low}) < C_n < f(T_{hi})$. Without loss of generality, assume we need to adjust to a higher limit, i.e., $f(T_{mid}) < C_n < f(T_{hi})$. As illustrated in Figure 6.6, we can calculate the new bandwidth limit using linear interpolation:

$$T_{new} = T_{mid} + \frac{C_n - f(T_{mid})}{f(T_{hi}) - f(T_{mid})} \times (T_{hi} - T_{mid}). \tag{6.12}$$

In Figure 6.7 we illustrate how to calculate the approximated linear interpolation using an example. To calculate

$$\frac{C_n - f(T_{mid})}{f(T_{hi}) - f(T_{mid})} = \frac{1012}{1690}, \tag{6.13}$$

we first use the same approximate division lookup table technique discussed earlier in §6.5.1, except the division results are now stored as a (mantissa, exponent) pair. We truncate the numerator and denominator to get most significant non-zero bits $i =$`01111`$/j =$`11010`, and retrieve the approximate division result

$$\frac{i+0.5}{j+0.5} = \frac{2396}{2^{12}}. \tag{6.14}$$

After the approximate division, we need to multiply the result by Δ. To make this calculation easier, we choose Δ to be a power of 2, reducing the multiplication into a bit-shift. In practice, we set $\Delta = 2^{\lfloor \log_2 (\frac{1}{2} T_{mid}) \rfloor}$, meaning $T_{low} = T_{mid} - \Delta \approx 0.5 T_{mid}$ and $T_{hi} = T_{mid} + \Delta \approx 1.5 T_{mid}$.

The divide-then-multiply calculation can be applied as a single bit shift. In the example in Figure 6.7, we have $\Delta = 2^{14}$ and need to calculate $\frac{2396}{2^{12}} \cdot 2^{14}$, which can be simplified into a left shift:

$$2396 << (14 - 12) = 9584. \tag{6.15}$$

Finally, we finish the last addition operation in the approximate linear interpolation, and obtain the new bandwidth limit

$$T_{new} = T_{mid} + 9584. \tag{6.16}$$

Similarly, when adjusting towards a lower limit, we use

$$T_{new} = T_{mid} - \frac{f(T_{mid}) - C_n}{f(T_{mid}) - f(T_{low})} \times (T_{mid} - T_{low}). \tag{6.17}$$

Notice that we use subtraction from T_{mid} instead of adding up from T_{lo} to interpolate. This is because the approximate division has a constant relative error proportional to the result. By subtracting the result from T_{mid}, we can make more accurate fine-grained adjustments near T_{mid} to better converge towards the optimal bandwidth limit. Instead, if we use

$$T_{new} = T_{low} + \frac{C_n - f(T_{low})}{f(T_{mid}) - f(T_{low})} \times (T_{mid} - T_{low}), \tag{6.18}$$

the approximated interpolation is more accurate near T_{low} and has a larger error near T_{mid}.

When C_n falls out of the range $[f(T_{low}), f(T_{hi})]$, our estimate candidates are too far off from the ideal bandwidth limit, and we clip the update by choosing $f(T_{low})$ or $f(T_{hi})$ directly. Clipping prevents overshooting caused by using linear interpolation outside of the two candidate points.

We further note that although CMS-LPF will introduce over-estimation errors across the board for all estimated rates, our closed-loop bandwidth limit update process will naturally adapt to this error. When all rates are slightly over-estimated while the bandwidth limit T_n is not yet over-estimated, users will suffer from an unnecessarily high drop probability, leading to less than C_n total traffic; AHAB will then automatically raise T_n to account for such global over-estimation.

6.5.4 Iterative Update Using Worker Packets

To achieve fast convergence towards intra-slice fairness, AHAB updates the bandwidth limit T_n fully within data plane. At the end of every epoch, AHAB calculates a new bandwidth limit T_{new} for each slice using approximate interpolation, and use it as the new bandwidth limit for the next epoch.

However, as shown in Figure 6.4, T_n is stored in a register memory lookup table near the beginning of the switch's packet-processing pipeline while the new limit T_{new} is only available in later pipeline stages; the pipeline's memory access constraint does not allow us to write T_{new} back to the same register memory directly. Therefore, at the end of every epoch, we generate one worker packet per slice by packet cloning, and use packet recirculation to let the worker packet go through the pipeline twice. The worker packet reads all the candidate bandwidth limits, performs the approximated linear interpolation calculation to derive the new bandwidth limit T_{new}, and carries it to the beginning of the packet-processing pipeline to be written into register memory.

Although the update is slightly delayed due to packet recirculation (about $0.65\mu s$), only a very small fraction of packets near the beginning of the epoch are affected, therefore the actual difference in enforcement due to the delayed update is negligible. As we show in §6.6, this closed-loop update process rapidly converges to the fair bandwidth allocation.

6.5.5 Supporting Weighted Allocation

A network operator sometimes needs to allocate bandwidth in proportion to a pre-assigned weight, for example when implementing differentiated services. AHAB supports weighted fair allocation at both the slice level and the user level.

To support weighted fair bandwidth allocation between users in the same slice, we scale each packet's length using the user's weight: if a user m has weight w_m, a packet with size S is scaled into $\frac{S}{w_m}$ before being used to calculate the user's scaled sending rate R_m in the CMS-LPF estimator. This way, we can directly compare different user sending rates R_m against the same per-user bandwidth limit T_n.

Meanwhile, the control plane is more flexible and trivially supports allocating bandwidth to different slices based on their weight. We simply divide each slice's demand and capacity by its weight before computing the max-min fairness allocation.

We also note that the weights assigned to slices / users can be easily updated at run time. To adjust the weight for a subset of users, we adjust the rules installed in the slice lookup table in the data plane; to adjust the weight of a slice, we modify it directly from the control plane.

6.6 Evaluation

Using a prototype implementation running in a hardware testbed, we show that AHAB can quickly achieve fair and stable bandwidth allocation between users. Compared to the prior state-of-the-art, HCSFQ[129], AHAB not only converges to the target fair bandwidth allocation faster (in 3.1ms), but also achieves comparable or better fairness and throughput stability. Subsequently, we use real-world traffic traces in a simulation-based experiment to show that AHAB scales well to 5.9-23.9 million users with a reasonable memory footprint, and CMS-LPF has a minimal impact on scheduling fairness.

6.6.1 Testbed Experiment Setup

We implement a AHAB prototype on an Intel Tofino[65] Wedge-100 programmable switch, using approximately 2,000 lines of P4[22]; we have released the code on GitHub[33]. We evaluate AHAB's real-world scheduling fairness using a hardware testbed with two sender and receiver servers connected via the programmable switch. Both servers have a 20-core CPU and a Mellanox ConnectX-5 2x100Gbps NIC, and run Ubuntu 20.04. The sender sends TCP flows using `iperf3` with Linux's default congestion control (cubic), and send UDP flows using either `iperf3` or a customized Go script that performs millisecond-level throughput measurement. We set the iterative update epoch time to 1ms and configure the LPF rate estimator's time constant to τ=4ms. Unless otherwise noted, we use a CMS-LPF estimator with size 3x2048.

In all experiments, we treat each flow as a unique user, using its 5-tuple (source and destination IP/port pairs) as the user ID. We note that real-world traffic may be grouped more coarsely; for example, one user in a mobile network may include all flows destined for the same device (same destination IP).

6.6.2 Fast Convergence

We now compare AHAB to the state-of-the-art of hierarchical fair queuing based on programmable switch: HCSFQ[129]. HCSFQ iteratively converges to the fair rate via Additive Increase Multiplicative Decrease, limiting its convergence speed when the number of users decreases and the fair rate increases.

To demonstrate the difference in convergence time, we program both AHAB and HCSFQ to enforce fairness between four UDP flows in a single slice, with a fixed 100Mbps capacity. All four flows have the same 100Mbps constant sending rate, but have different starting and ending time: they run between T=0-8s, T=1-7s, T=2-6s, and T=3-5s, respectively.

Figure 6.8 shows the actual bandwidth used by the four flows over time, after bandwidth limit enforcement done by AHAB (left) or HCSFQ (right). At a longer timescale (top), the two schedulers behaved similarly. However, if we zoom in to a smaller timescale and plot the millisecond-level per-flow throughput (bottom) immediately after T=7s (where flow 1

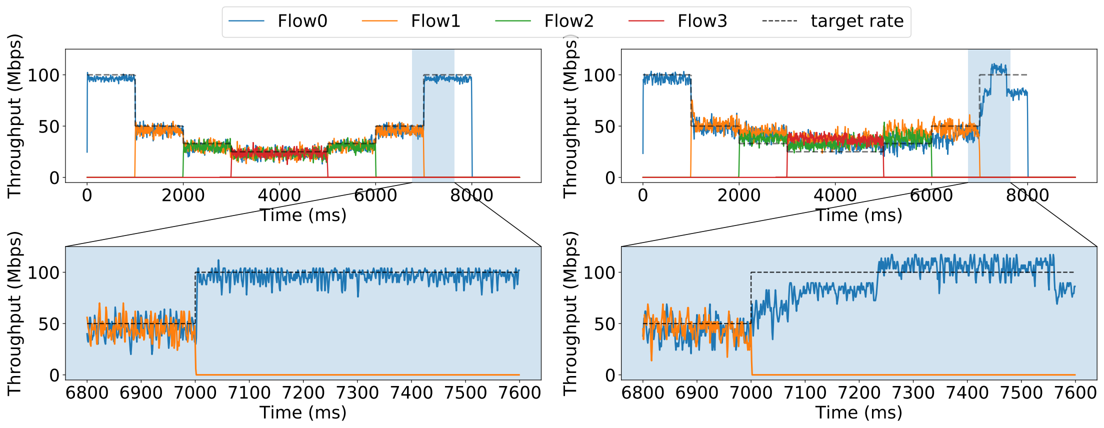


Figure 6.8: AHAB (left) converges to fair bandwidth allocation within 3.1ms on average, while HCSFQ[129] (right) needs 42.3ms.

stopped), we can see AHABconverges much faster than HCSFQ to allow flow 0 to use the full 100Mbps bandwidth. We measured the time for flow throughput to converge to within 10% of ideal fair bandwidth limit. AHAB's interpolation-based update only needs around three iterations to converge, taking only 3.1ms on average (at most 5ms). In comparison, HCSFQ needs on average 42.3ms (at most 234ms).

6.6.3 Fairness and Goodput Stability

We first demonstrate that AHAB can effectively enforce fair bandwidth allocation for TCP flows, by simultaneously running 2, 4, or 8 flows sharing a slice with fixed 1Gbps capacity.

Figure 6.9 shows the goodput over time when we run two TCP flows; although both systems achieve fairness between two flows in terms of average goodput over time, AHAB achieve better goodput stability over time. For analysis and comparison, in Figure 6.10 we plot the cumulative distribution function (CDF) of the TCP goodput of all flows, reported by `iperf3` in 1-second intervals across 60 seconds. In the ideal case, all flows exhibit the

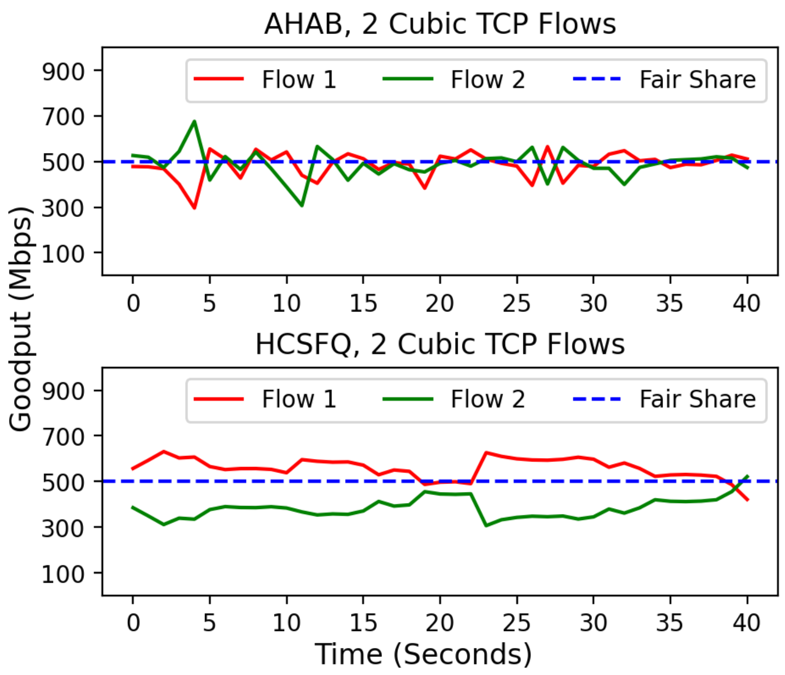


Figure 6.9: Goodput of two competing TCP flows over time.

same goodput across time, leading to a steeper CDF. The stability achieved by AHAB is comparable to that of HCSFQ: on average, the goodputs of flows enforced by AHAB are within 12.1% of ideal fair share, while those enforced by HCSFQ exhibits 15.5%.

Meanwhile, we also show approximate probabilistic dropping can effectively achieve fair bandwidth allocation for non-TCP traffic that does not react to bandwidth limiting (no congestion control), even with very different sending rates. We let multiple UDP flows share the same slice with fixed 100Mbps capacity, and configured their sending rate to be 10Mbps, 20Mbps, 30Mbps, and so on. In Figure 6.11, we show the throughput achieved by these flows, when 4, 8, and 16 flows are sent simultaneously. In the latter two cases, the slice is over-utilized and approximate probabilistic dropping kicks in. Although AHAB needs to apply vastly different dropping probabilities for the wide range of sending rates,

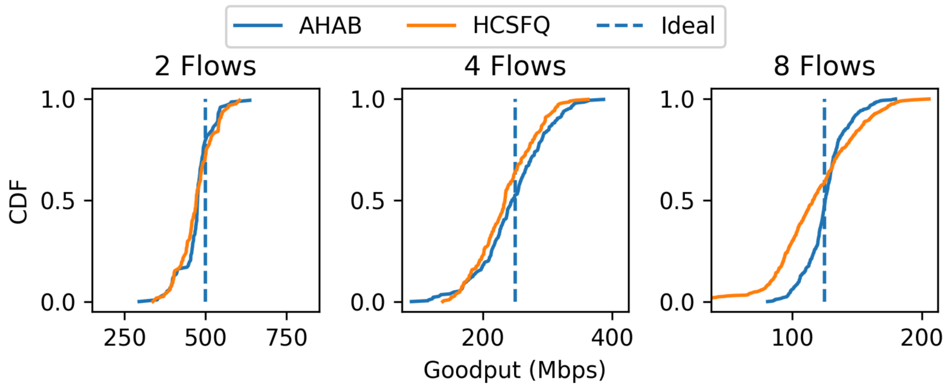


Figure 6.10: Cumulative distribution of competing TCP flows's goodput when using AHAB versus HCSFQ.

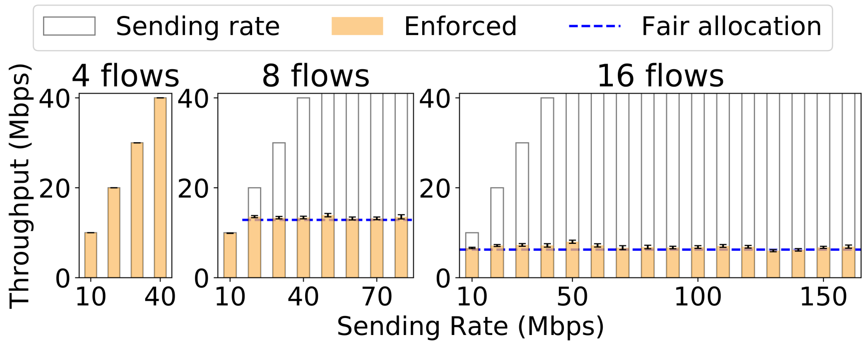


Figure 6.11: Given flows with various sending rates, AHAB's approximate probabilistic dropping achieved fair bandwidth allocation within 6% error.

the resulting allocation is quite fair. On average, the mean throughput achieved is within 4% and 6% of the fair bandwidth allocation target, for 8 and 16 flows respectively. This corresponds to the error of the approximated division using the lookup table.

Figure 6.12 demonstrates AHAB's support of weighted fairness. We start three groups of flows with weight 1x, 2x, and 4x respectively, with four flows per group, all sharing one slice with 1Gbps capacity. Flows with the same weight achieve the same throughput, proportional to their allocated weight, and their attained throughput averages within 15% and 1% of the weighted fair allocation for TCP and UDP, respectively.

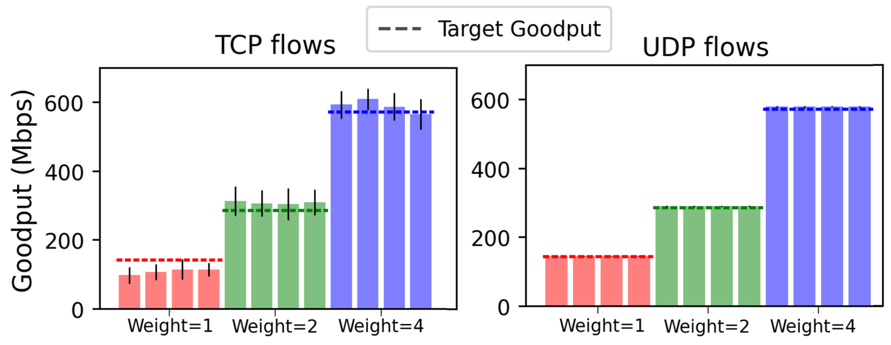


Figure 6.12: Goodput of weighted TCP and UDP flows sharing one 1Gbps slice.

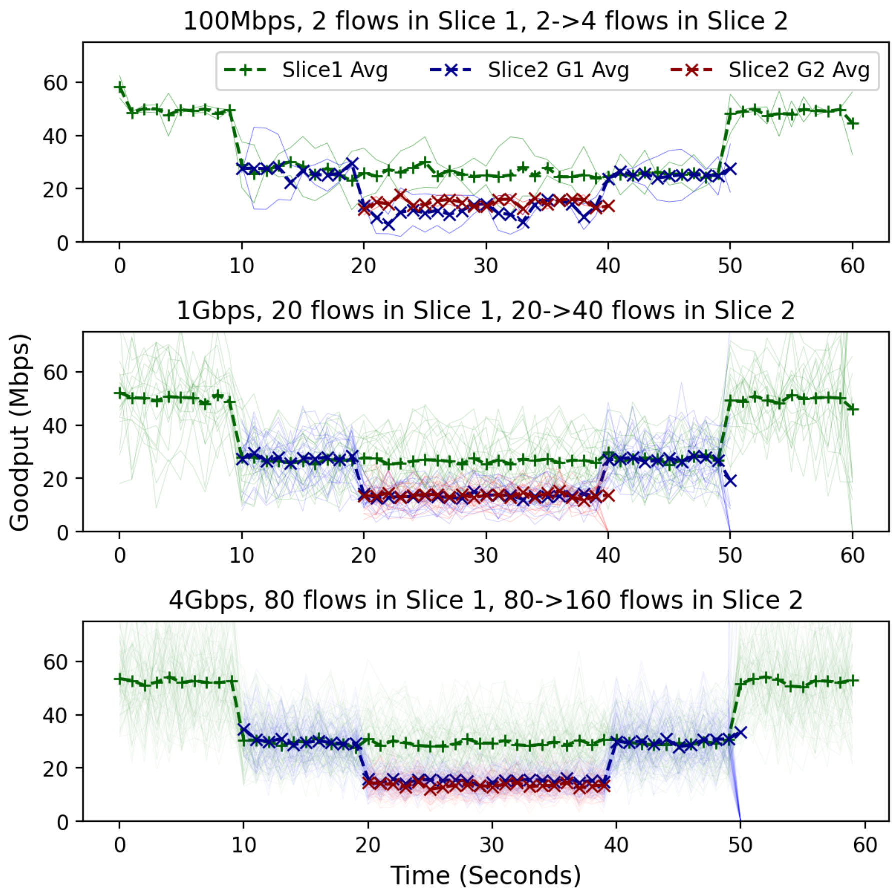


Figure 6.13: Three experiments showing two slices sharing a common bottleneck. Slice 1 uses half the bandwidth even when Slice 2 has twice as many flows.

6.6.4 Inter-slice Fairness

In Figure 6.13 we demonstrate that AHAB rapidly adjusts to the changing bandwidth demands of different slices. We set up an experiment where Slice 1 always has x users (TCP flows), and Slice 2 is initially idle with no user. At T=10s x users in Slice 2 that starts sending, lasting until T=50s. At T=20s another x users join Slice 2 and start sending until T=40s. In the ideal case, all bandwidth is fully allocated to Slice 1 between T=0-10s as well as T=50-60s, fairly shared between x users; the total bandwidth is split in half between Slice 1 and Slice 2 during T=10-50s. When more users are added to Slice 2 during T=20-40s, users in Slice 2 each get a lower share while users in Slice 1 are not affected.

Figure 6.13 shows three scenarios: the total bandwidth shared by the two slices are 100, 1000, and 4000 Mbps, respectively, and we also have x=2, 20, and 80 users proportionally. We plot and compare the average goodput attained by the users in each slice, which is also a good indicator of fairness between slices. The bandwidth allocation between slices always quickly converged to fairness (within a few RTTs). When users send UDP traffic instead, AHAB instantly achieves near-perfect fair allocation for all three cases; the result is omitted here.

6.6.5 Scalability

To evaluate AHAB's performance at scale, we run trace-based simulation experiments to understand how much memory is needed to support a large number of users.

We collected a 15-minute anonymized traffic trace from the core network of a local Internet Service Provider and played the trace through a Python-based simulator. We treat each of the 5,980,000 unique source-destination IP pairs as an user, and let all users share a single

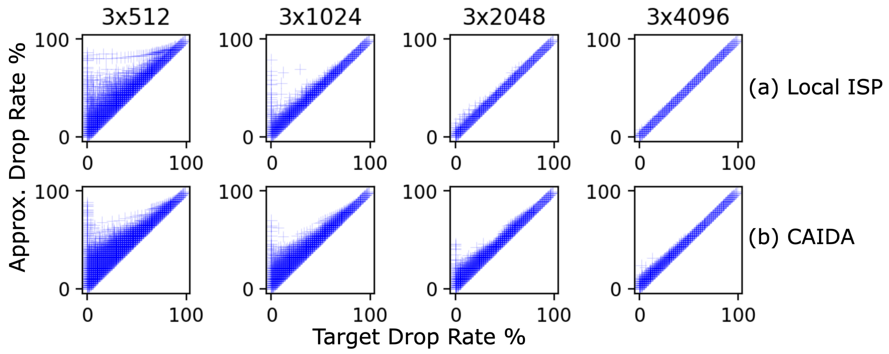


Figure 6.14: 3x4096 CMS-LPF estimator is sufficient for serving millions of users, with negligible error in drop probability.

slice with capacity C_n set to 0.84Gbps, equal to the average throughput of the trace. Due to natural fluctuations in traffic rate, the instantaneous bandwidth demand often exceeds C_n. The simulator calculates the fair per-user bandwidth limit T_n for each epoch, and then calculates the "target" probabilistic drop rate $1 - \min(\frac{T_n}{R_m})$ using the ground-truth per-user sending rate R_m.

Meanwhile, we also simulate the per-user estimated sending rate $\widetilde{R_m}$ reported by CMS-LPF estimators of different sizes, and use $\widetilde{R_m}$ to calculate the "approximated" drop probability $1 - \min(\frac{T_n}{\widetilde{R_m}})$. Shrinking the size of CMS-LPF estimator reduces the accuracy of rate estimation, which in turns leads to more error in the drop probability.

As shown by Figure 6.14(a), using a CMS-LPF estimator with size 3x512 led to a fraction of packets with drop rate higher than the target; although most errors lie in over-utilized users, some users with a target drop rate of 0% (under-utilized users) also experience significant packet drops. Meanwhile, CMS-LPF size 3x4096 is sufficient to reduce errors to negligible level. We conclude that a CMS-LPF estimator with size 3x4096 is sufficient for

CMS-LPF Dimension	Hardware Memory Utilization	Supported # of Users
2048x3 (24KB)	1.56%	2,990,000
4096x3 (48KB)	2.60%	5,980,000
16384x3 (768KB)	8.85%	23,920,000 (est.)

Table 6.2: Memory utilization and supported number of users w.r.t. different sizes of the CMS-LPF estimator.

AHAB to accurately produce per-user rate estimate for the 5,980,000 unique users in our trace.

We also run the same simulation using five minutes of CAIDA Anonymized Internet Trace 2018[26]. The trace has an average throughout of 3.5Gbps and has 7,300,000 unique flow 5-tuples. We obtain similar results, as shown in Figure 6.14(b).

Now we analyze the switch hardware resources used by AHAB, and specifically focus on the memory used by the CMS-LPF estimator. As shown in Table 6.2, a small 2048x3 CMS-LPF estimator only costs a small fraction (1.56%) of all stateful memory available on the switch hardware, yet it already supports accurately enforcing bandwidth limit for 3 million users. We can fit a much larger sketch than what we used in the prototype: allocating a 16384x3 CMS-LPF estimator costs 8.85% of the available memory. Assuming similar traffic skewness as in our ISP trace, a single programmable switch can support 23.9 million devices across all slices, which is sufficient for many application scenarios.

As for the number of slices, our prototype program supports up to 16,000 slices. The Tofino switch supports 3.2Tbps aggregated throughput, which can be shared among 2,000 downstream base stations. It is possible to expand further by adding more entries to the slice lookup table and allocating more per-slice bandwidth demand trackers, as they only occupy a small fraction of the total data-plane memory usage (with the majority being the

Resource	Instr. Words	Hash Units	TCAM
Utilization	28.6%	37.5%	5.2%

Table 6.3: Utilization of other switch hardware resources.

CMS-LPF rate estimator). The primary limiting factor on the number of slices is control-plane speed, as supporting *N* slices requires the control plane to read *N* demands and write *N* capacities per update.

We also report other resource utilization of AHAB data-plane program in Table 6.3. These resource usages are constant regardless of the number of slices and users.

6.7 Related Work

Fair Queuing using estimated rate

Core Stateless Fair Queuing[115] is a network architecture where edge nodes estimate the rate of incoming flows and attach the rate to packets, while core nodes in the network choose a fair per-flow rate and enforce it using probabilistic dropping. It requires maintaining per-flow state to estimate sending rates. Approximate Fairness through Differential Dropping[95] uses a shadow buffer that holds recent packets to approximately derive per-flow rates, and similarly performs probabilistic dropping. It is not straightforward to implement a large shadow buffer given the computational constraints present in today's high-speed programmable switches. Also, both works require one dedicated hardware queue per "slice" (group of flows).

Rank-based scheduling

In Push-In, First-Out (PIFO) queues, each packet is pushed in with a certain rank, and packets with the highest rank are transmitted first. Admit-In, First-Out[128] and SP-PIFO[2] both approximate the behavior of a PIFO queue on commodity programmable switches. AIFO uses a sample of recently admitted packets to estimate the rank distribution of packets in the queue, which is used to decide a threshold and reject low-ranked packets from being admitted. Meanwhile, SP-PIFO uses an array of strict-priority queues and dynamically adjust the mapping from ranks to queues using estimated quantile distribution of ranks. These works both assume an oracle which assigns ranks to packets.

Fair queuing in the data plane

Approximate Fair Queuing[108] implements scalable per-flow fair queuing by splitting traffic into calendar epochs. This design requires rapidly rotating the priority between multiple queues to serve different future epochs, and does not support a multi-layer scheduling hierarchy. Gearbox[56] proposed a new hardware design specifically supporting multi-level calendar queuing. Meanwhile, Hierarchical Core-Stateless Fair Queuing[129] extends CSFQ[115] and uses Addictive Increase, Multiplicative Decrease (AIMD) to iteratively find a fair per-user (per-tenant) sending rate limit using queue congestion status feedback. Although it can support multiple layers of scheduling hierarchy, its dependency on per-user memory for estimating per-user sending rates hurts scalability. The AIMD process also takes a relatively long time to adapt when the fair rate increases. Cebinae[127] uses leaky-bucket filters to estimate per-flow rate and enforce fairness by "taxing" the heavy flows, however it takes several seconds to converge to fair allocation. Nimble[116] implements precise TCP flow rate limiting by simulating queue draining in the data plane. However, it only supports fixed rates set by the control plane and requires per-flow memory. Instead, our work automatically adjusts and enforces fair per-user bandwidth limit within milliseconds timescale for millions of users.

6.8 Conclusion

We present AHAB, a data-plane hierarchical fair bandwidth limit enforcer. Using a novel approximate data structure, AHAB scales to millions of users across thousands of network slices. AHAB exploits approximate arithmetic to implement interpolation-based bandwidth limit update fully within the data plane, leading to fast convergence. Evaluation shows that AHAB converges to a fair allocation within 3.1ms, 13x faster than prior work, without sacrificing fairness or stability.

Concluding Remarks

This dissertation has explored the realms of measurement and closed-loop control in the network data plane, offering experience and insight from an algorithm design perspective to tackle the intricate challenges imposed by the switch hardware. We hope the four works presented can shed some light on a coherent, end-to-end approach for network programming, with an emphasis on adapting for the programmable switch's hardware architectural constraints.

As we conclude this journey, it's essential to acknowledge that designing and deploying algorithms for network data plane was once a meticulous effort involving a nontrivial amount of hacks and workarounds, many of which are intentionally omitted from this dissertation. Those issues are more or less specific to a particular early hardware design, and will likely disappear as the field matures. Instead, we hope the readers can focus on the more fundamental, long-term trends that shape the landscape of networking and data-plane algorithm design. For example, given the two *Moore's law*-like trends in networking hardware — network throughput doubles every 18-22 months, while memory bandwidth doubles every 24-28 months — we have every reason to keep investigating innovative algorithms that are memory access efficient. Also, the algorithmic insights unveiled here extend well beyond a particular type of networking hardware. Surprisingly, even for eBPF programs running on a conventional CPU, we find memory access bottlenecks reminiscent of those encountered within the switch pipeline, despite the underlying CPU hardware architecture providing an abundance of high-speed cache. One important future direction is, therefore, building toolchains and unified programming abstractions that support many different data-plane targets with automated adaptation and synthesis, easing the burden of algorithm designers.

Of course, we must admit uncertainty shrouds the future trajectory of network hardware evolution. Nevertheless, the approach adopted in this dissertation, which amalgamates theoretical insights with practical hardware-friendly adaptations, should prove invaluable in the years to come.

Bibliography

[1] Afek, Y., Bremler-Barr, A., Feibish, S. L., & Schiff, L. (2018). Detecting heavy flows in the SDN match and action model. *Computer Networks*, 136, 1–12.

[2] Alcoz, A. G., Dietmüller, A., & Vanbever, L. (2020). SP-PIFO: Approximating Push-In First-Out behaviors using Strict-Priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (pp. 59–76).

[3] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Lam, V. T., Matus, F., Pan, R., Yadav, N., & Varghese, G. (2014). CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM* (pp. 503–514).

[4] Alizadeh, M., Greenberg, A. G., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., & Sridharan, M. (2010). Data center TCP (DCTCP). In *ACM SIGCOMM* (pp. 63–74).

[5] Appenzeller, G., Keslassy, I., & McKeown, N. (2004). Sizing router buffers. In *ACM SIGCOMM* (pp. 281–292).

[6] Arista Networks (2017). Arista 7050X Switch Architecture ('A day in the life of a packet'). https://web.archive.org/web/20170422142409/https://www.corporatearmor.com/documents/Arista_7050X_Switch_Architecture_Datasheet.pdf. Accessed: 2017-04-22.

[7] Assaf, E., Basat, R. B., Einziger, G., & Friedman, R. (2018). Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *IEEE INFOCOM 2018* (pp. 2204–2212).: IEEE.

[8] Bar-Yossef, Z., Jayram, T. S., Kumar, R., Sivakumar, D., & Trevisan, L. (2002). Counting distinct elements in a data stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science* (pp. 1–10).: Springer.

[9] Barefoot Networks (2017). Product Brief Tofino Page. https://web.archive.org/web/20180104235002/https://www.barefootnetworks.com/products/brief-tofino/. Accessed: 2018-01-04.

[10] Basat, R. B., Chen, X., Einziger, G., Feibish, S. L., Raz, D., & Yu, M. (2020a). Routing oblivious measurement analytics. In *IFIP Networking*.

[11] Basat, R. B., Chen, X., Einziger, G., Friedman, R., & Kassner, Y. (2019). Randomized admission policy for efficient top-k, frequency, and volume estimation. *IEEE/ACM Transactions on Networking*, 27(4), 1432–1445.

[12] Basat, R. B., Chen, X., Einziger, G., & Rottenstreich, O. (2020b). Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking*, 28(3), 1172–1185.

[13] Basat, R. B., Einziger, G., Friedman, R., Luizelli, M. C., & Waisbard, E. (2017). Constant time updates in hierarchical heavy hitters. *ACM SIGCOMM*, (pp. 127–140).

[14] Basat, R. B., Friedman, R., & Shahout, R. (2018). Stream frequency over interval queries. *Proceedings of the VLDB Endowment*, 12(4), 433–445.

[15] Ben-Basat, R., Chen, X., Einziger, G., & Rottenstreich, O. (2018a). Efficient measurement on programmable switches using probabilistic recirculation. In *IEEE International Conference on Network Protocols (ICNP'18)* (pp. 313–323).

[16] Ben-Basat, R., Einziger, G., Feibish, S. L., Moraney, J., & Raz, D. (2018b). Network-wide routing-oblivious heavy hitters. In *Symposium on Architectures for Networking and Communications Systems (ANCS)* (pp. 66–73).

[17] Ben-Basat, R., Einziger, G., Keslassy, I., Orda, A., Vargaftik, S., & Waisbard, E. (2018c). Memento: Making sliding windows efficient for heavy hitters. In *ACM CoNEXT* (pp. 254–266).

[18] Bennett, J. C. R. & Zhang, H. (1996). Hierarchical packet fair queueing algorithms. In *ACM SIGCOMM* (pp. 143–156).

[19] Benson, T., Akella, A., & Maltz, D. A. (2010). Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM Internet Measurement Conference (IMC)* (pp. 267–280).

[20] Benson, T., Anand, A., Akella, A., & Zhang, M. (2011). MicroTE: Fine grained traffic engineering for data centers. In *ACM CoNEXT* (pp. 1–12).

[21] Bertsimas, D. J. & Servi, L. D. (1992). Deducing queueing from transactional data: the queue inference engine, revisited. *Operations Research*, 40(3-supplement-2), S217–S228.

[22] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3), 87–95.

[23] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., & Horowitz, M. (2013). Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM* (pp. 99–110).

[24] Braverman, V., Gelles, R., & Ostrovsky, R. (2014). How to catch l_2-heavy-hitters on sliding windows. *Theoretical Computer Science*, 554, 82–94.

[25] CAIDA (2016). CAIDA Anonymized Internet Trace, 2016-04-06.

[26] CAIDA (2018). CAIDA Anonymized Internet Traces, 2018-03-15.

[27] Chao, A. (1984). Nonparametric estimation of the number of classes in a population. *Scandinavian Journal of Statistics*, (pp. 265–270).

[28] Charikar, M., Chen, K. C., & Farach-Colton, M. (2004). Finding frequent items in data streams. *Theoretical Computer Science*, 312(1), 3–15.

[29] Chen, M., Chen, S., & Cai, Z. (2016). Counter tree: A scalable counter architecture for per-flow traffic measurement. *IEEE/ACM Transactions on Networking*, 25(2), 1249–1262.

[30] Chen, X. (2020a). Github Repository of BeauCoup. `https://github.com/Princeton-Cabernet/BeauCoup`.

[31] Chen, X. (2020b). Github Repository of ConQuest. `https://github.com/Princeton-Cabernet/p4-projects/tree/master/ConQuest-tofino`.

[32] Chen, X. (2020c). Github Repository of PRECISION. `https://github.com/Princeton-Cabernet/p4-projects/tree/master/PRECISION-tofino`.

[33] Chen, X. (2023). Github Repository of AHAB. https://github.com/Princeton-Cabernet/AHAB.

[34] Chen, X., Feibish, S. L., Koral, Y., Rexford, J., & Rottenstreich, O. (2018). Catching the microburst culprits with snappy. In *ACM SIGCOMM Workshop on Self-Driving Networks* (pp. 22–28).

[35] Chen, X., Feibish, S. L., Koral, Y., Rexford, J., Rottenstreich, O., Monetti, S. A., & Wang, T.-Y. (2019). Fine-grained queue measurement in the data plane. In *ACM CoNEXT* (pp. 15–29).

[36] Chen, X., Landau-Feibish, S., Braverman, M., & Rexford, J. (2020). Beaucoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM* (pp. 226–239).

[37] Chen, Y., Griffiths, R., Zats, D., Joseph, A. D., & Katz, R. H. (2012). Understanding TCP Incast and its implications for big data workloads. *;login*, 37(3).

[38] Chole, S., Fingerhut, A., Ma, S., Sivaraman, A., Vargaftik, S., Berger, A., Mendelson, G., Alizadeh, M., Chuang, S.-T., Keslassy, I., Orda, A., & Edsall, T. (2017). dRMT: Disaggregated programmable switching. In *ACM SIGCOMM* (pp. 1–14).

[39] Chowdhury, M. & Stoica, I. (2012). Coflow: a networking abstraction for cluster applications. In *ACM HotNets Workshop* (pp. 31–36).

[40] Chowdhury, M., Zhong, Y., & Stoica, I. (2014). Efficient coflow scheduling with varys. In *ACM SIGCOMM Computer Communication Review*, volume 44 (pp. 443–454).: ACM.

[41] Christiansen, M., Jeffay, K., Ott, D., & Smith, F. D. (2001). Tuning RED for web traffic. *IEEE/ACM Transactions on Networking*, 9(3), 249–264.

[42] Claise, B. (2004). Cisco Systems NetFlow Services Export Version 9. *RFC 3954*.

[43] Cormode, G. (2011). Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*.

[44] Cormode, G. & Hadjieleftheriou, M. (2008). Finding frequent items in data streams. *Proceedings of the VLDB Endowment*, 1(2), 1530–1541.

[45] Cormode, G. & Hadjieleftheriou, M. (2010). Methods for finding frequent items in data streams. *The VLDB Journal*, 19, 3–20.

[46] Cormode, G., Korn, F., Muthukrishnan, S., & Srivastava, D. (2003). Finding hierarchical heavy hitters in data streams. In *Proceedings of 2003 VLDB Conference* (pp. 464–475).

[47] Cormode, G., Korn, F., Muthukrishnan, S., & Srivastava, D. (2004). Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data. In *ACM SIGMOD* (pp. 155–166).

[48] Cormode, G. & Muthukrishnan, S. (2005). An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1), 58–75.

[49] Durand, M. & Flajolet, P. (2003). Loglog counting of large cardinalities. In *European Symposium on Algorithms* (pp. 605–617).: Springer.

[50] Estan, C., Savage, S., & Varghese, G. (2003). Automatically inferring patterns of resource consumption in network traffic. In *ACM SIGCOMM* (pp. 137–148).

[51] Estan, C. & Varghese, G. (2002). New directions in traffic measurement and accounting. In *ACM SIGCOMM* (pp. 323–336).

[52] Flajolet, P., Fusy, É., Gandouet, O., & Meunier, F. (2007). Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms (AOFA)*.

[53] Flajolet, P., Gardy, D., & Thimonier, L. (1992). Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3), 207–229.

[54] Floyd, S. & Jacobson, V. (1993). Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4), 397–413.

[55] Floyd, S. & Jacobson, V. (1995). Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), 365–386.

[56] Gao, P., Dalleggio, A., Xu, Y., & Chao, H. J. (2022). Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)* (pp. 551–565).

[57] Ghorbani, S., Yang, Z., Godfrey, P. B., Ganjali, Y., & Firoozshahian, A. (2017). DRILL: Micro load balancing for low-latency data center networks. In *ACM SIGCOMM* (pp. 225–238).

[58] Gibbons, P. B. (2001). Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of 2001 VLDB conference*, volume 1 (pp. 541–550).

[59] Gupta, A., Harrison, R., Canini, M., Feamster, N., Rexford, J., & Willinger, W. (2018). Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM* (pp. 357–371).

[60] Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., & Wójcik, M. (2017). Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM* (pp. 29–42).

[61] Harrison, R., Cai, Q., Gupta, A., & Rexford, J. (2018). Network-wide heavy hitter detection with commodity switches. In *ACM SIGCOMM Symposium on SDN Research (SOSR'18)* (pp. 8:1–8:7).

[62] Hasan, S., Padmanabhan, A., Davie, B., Rexford, J., Kozat, U., Gatewood, H., Sanadhya, S., Yurchenko, N., Al-Khasib, T., Batalla, O., Bremner, M., Lee, A., Makeev, E., Moeller, S., Rodriguez, A., Shelar, P., Subraveti, K., Kandi, S., Xoconostle, A., Ramakrishnan, P. K., Tian, X., & Tomar, A. (2023). Building flexible, low-cost wireless access networks with Magma. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)* (pp. 1667–1681).

[63] He, K., Rozner, E., Agarwal, K., Felter, W., Carter, J., & Akella, A. (2015). Presto: Edge-based load balancing for fast datacenter networks. In *ACM SIGCOMM* (pp. 465–478).

[64] Homem, N. & Carvalho, J. P. (2010). Finding top-k elements in data streams. *Information Sciences*, 180(24), 4958–4974.

[65] Intel (2020). Intel® Tofino™ Series Programmable Ethernet Switch ASIC. https://web.archive.org/web/20201130025235/https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. Accessed: 2020-11-30.

[66] Joshi, R., Qu, T., Chan, M. C., Leong, B., & Loo, B. T. (2018). BurstRadar: Practical real-time microburst monitoring for datacenter networks. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*.

[67] Karp, R. M., Shenker, S., & Papadimitriou, C. H. (2003). A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1), 51–55.

[68] Kim, D., Zhu, Y., Kim, C., Lee, J., & Seshan, S. (2018). Generic external memory for switch data planes. In *ACM Workshop on Hot Topics in Networks* (pp. 1–7).

[69] Kuzmanovic, A. & Knightly, E. W. (2003). Low-rate TCP-targeted denial of service attacks: The shrew vs. the mice and elephants. In *ACM SIGCOMM* (pp. 75–86).

[70] Laboratory For Advanced Systems Research, UCLA (2001). UCLA CSD Packet Traces. http://www.lasr.cs.ucla.edu/ddos/traces/.

[71] Larsen, K. G., Nelson, J., & Nguyên, H. L. (2015). Time lower bounds for non-adaptive turnstile streaming algorithms. In *ACM Symposium on Theory of Computing (STOC'15)* (pp. 803–812).

[72] Larson, R. C. (1990). The queue inference engine: Deducing queue statistics from transactional data. *Management Science*, 36(5), 586–601.

[73] Li, T., Chen, S., & Ling, Y. (2012). Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Transactions on Networking*, 20(5), 1622–1634.

[74] Li, Y., Miao, R., Kim, C., & Yu, M. (2016a). FlowRadar: A better NetFlow for data centers. In *13th USENIX Sym- posium on Networked Systems Design and Implementation (NSDI'16)* (pp. 311–324).

[75] Li, Y., Miao, R., Kim, C., & Yu, M. (2016b). FlowRadar: a better NetFlow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)* (pp. 311–324).

[76] Li, Y., Miao, R., Liu, H. H., Zhuang, Y., Feng, F., Tang, L., Cao, Z., Zhang, M., Kelly, F., Alizadeh, M., et al. (2019). HPCC: high precision congestion control. In *ACM SIGCOMM* (pp. 44–58).

[77] Liu, Z., Ben Basat, R., Einziger, G., Kassner, Y., Braverman, V., Friedman, R., & Sekar, V. (2019). Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM* (pp. 334–350).

[78] Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., & Braverman, V. (2016). One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM* (pp. 101–114).

[79] Liu, Z., Zhou, S., Rottenstreich, O., Braverman, V., & Rexford, J. (2020). Memory-efficient performance monitoring on programmable switches with lean algorithms. In *SIAM-ACM Symposium on Algorithmic Principles of Computer Systems* (pp. 31–44).

[80] Lu, Y., Montanari, A., Prabhakar, B., Dharmapurikar, S., & Kabbani, A. (2008). Counter braids: A novel counter architecture for per-flow measurement. *ACM SIGMETRICS Performance Evaluation Review*, 36(1), 121–132.

[81] MacDavid, R., Cascone, C., Lin, P., Padmanabhan, B., Thakur, A., Peterson, L., Rexford, J., & Sunay, O. (2021). A P4-based 5G user plane function. In *ACM SIGCOMM Symposium on SDN Research (SOSR'21)* (pp. 162–168).

[82] MacDavid, R., Chen, X., & Rexford, J. (2023). Scalable real-time bandwidth fairness in switches. In *IEEE INFOCOM*.

[83] Manerikar, N. & Palpanas, T. (2009). Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowledge Engineering*, 68(4), 415–430.

[84] Manku, G. S. & Motwani, R. (2002). Approximate frequency counts over data streams. In *Proceedings of 2002 VLDB Conference* (pp. 346–357).

[85] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., & Turner, J. (2008). Openflow: Enabling innovation in campus networks. *SIGCOMM Computer Communication Review*, 38(2), 69–74.

[86] Metwally, A., Agrawal, D., & Abbadi, A. E. (2005). Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the International Conference on Database Theory* (pp. 398–412).: Springer.

[87] Miao, R., Zeng, H., Kim, C., Lee, J., & Yu, M. (2017). Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM* (pp. 15–28).

[88] Mitzenmacher, M., Steinke, T., & Thaler, J. (2012). Hierarchical Heavy Hitters with the Space Saving Algorithm. In *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)* (pp. 160–174).: SIAM.

[89] Montazeri, B., Li, Y., Alizadeh, M., & Ousterhout, J. (2018). Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM* (pp. 221–235).

[90] Moshref, M., Yu, M., Govindan, R., & Vahdat, A. (2014). DREAM: Dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM* (pp. 419–430).

[91] Muthukrishnan, S. (2005). Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2).

[92] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., & Kim, C. (2017). Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM* (pp. 85–98).

[93] Nyalkalkar, K., Sinhay, S., Bailey, M., & Jahanian, F. (2011). A comparative study of two network-based anomaly detection methods. In *IEEE INFOCOM* (pp. 176–180).

[94] Olesinski, W. & Driediger, S. (2009). Fair WRED for TCP UDP traffic mix. US Patent 7,616,573.

[95] Pan, R., Breslau, L., Prabhakar, B., & Shenker, S. (2003). Approximate fairness through differential dropping. *ACM SIGCOMM Computer Communications Review*, 33(2), 23–39.

[96] Pan, R., Prabhakar, B., & Psounis, K. (2000). CHOKe-A stateless active queue management scheme for approximating fair bandwidth allocation. In *IEEE INFOCOM* (pp. 942–951).

[97] Pan, T., Yu, N., Jia, C., Pi, J., Xu, L., Qiao, Y., Li, Z., Liu, K., Lu, J., Lu, J., et al. (2021). Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM* (pp. 194–206).

[98] Patrascu, M. (2008). *Lower Bound Techniques for Data Structures*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

[99] Perry, J., Ousterhout, A., Balakrishnan, H., Shah, D., & Fugal, H. (2014). Fastpass: A centralized ”zero-queue” datacenter network. In *ACM SIGCOMM* (pp. 307–318).

[100] Pontarelli, S., Reviriego, P., & Mitzenmacher, M. (2018). EMOMA: exact match in one memory access. *IEEE Transactions on Knowledge and Data Engineering*, 30(11), 2120–2133.

[101] Roy, A., Zeng, H., Bagga, J., Porter, G., & Snoeren, A. C. (2015). Inside the social network's (datacenter) network. In *ACM SIGCOMM* (pp. 123–137).

[102] Rubio, D. (2017). Jinja templates in django. In *Beginning Django* (pp. 117–161). Springer.

[103] Schweller, R., Li, Z., Chen, Y., Gao, Y., Gupta, A., Zhang, Y., Dinda, P. A., Kao, M.-Y., & Memik, G. (2007). Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Networking*, 15(5), 1059–1072.

[104] Sekar, V., Duffield, N. G., Spatscheck, O., van der Merwe, J. E., & Zhang, H. (2006). LADS: Large-scale automated ddos detection system. In *USENIX Annual Technical Conference* (pp. 171–184).

[105] Sekar, V., Reiter, M. K., Willinger, W., Zhang, H., Kompella, R. R., & Andersen, D. G. (2008). csamp: A system for network-wide flow monitoring. In *5th USENIX Sym- posium on Networked Systems Design and Implementation (NSDI'08)* (pp. 233–246).

[106] Shah, R., Kumar, V., Vutukuru, M., & Kulkarni, P. (2020). TurboEPC: Leveraging dataplane programmability to accelerate the mobile packet core. In *ACM SIGCOMM Symposium on SDN Research (SOSR'20)* (pp. 83–95).

[107] Shahbaz, M., Suresh, L., Rexford, J., Feamster, N., Rottenstreich, O., & Hira, M. (2019). Elmo: Source routed multicast for public clouds. In *ACM SIGCOMM* (pp. 458–471).

[108] Sharma, N. K., Liu, M., Atreya, K., & Krishnamurthy, A. (2018). Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)* (pp. 1–16).

[109] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., & Licking, S. (2016a). Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM* (pp. 15–28).

[110] Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S.-T., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., & McKeown, N. (2016b). Programmable packet scheduling at line rate. In *ACM SIGCOMM* (pp. 44–57).

[111] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., & Rexford, J. (2017). Heavy-hitter detection entirely in the data plane. In *ACM SIGCOMM Symposium on SDN Research (SOSR'17)* (pp. 164–176).

[112] Sonchack, J., Michel, O., Aviv, A. J., Keller, E., & Smith, J. M. (2018). Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow. In *USENIX Annual Technical Conference* (pp. 823–835).

[113] Soós, G., Ficzere, D., Varga, P., & Szalay, Z. (2020). Practical 5G KPI measurement results on a non-standalone architecture. In *IEEE/IFIP Network Operations and Management Symposium* (pp. 1–5).

[114] Spang, B. & McKeown, N. (2019). On estimating the number of flows. In *Stanford Workshop on Buffer Sizing*.

[115] Stoica, I., Shenker, S., & Zhang, H. (1998). Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *ACM SIGCOMM* (pp. 118–130).

[116] Thapeta, V. S., Shinde, K., Malekpourshahraki, M., Grassi, D., Vamanan, B., & Stephens, B. E. (2021). Nimble: Scalable TCP-friendly programmable in-network rate-limiting. In *ACM SIGCOMM Symposium on SDN Research (SOSR'21)* (pp. 27–40).

[117] The P4 Language Consortium (2018a). P4$_{16}$ language specification. https://web.archive.org/web/20221127080230/https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html. Accessed: 2022-11-27.

[118] The P4 Language Consortium (2018b). *P4$_{16}$* Portable Switch Architecture. https://web.archive.org/web/20200711060854/https://p4.org/p4-spec/docs/PSA-v1.0.0.pdf. Accessed: 2020-07-11.

[119] Venkataraman, S., Song, D. X., Gibbons, P. B., & Blum, A. (2005). New streaming algorithms for fast detection of superspreaders. In *Network and Distributed System Security Symposium*.

[120] Wischik, D. & McKeown, N. (2005). Part I: Buffer sizes for core routers. *ACM SIGCOMM Computer Communication Review*, 35(3), 75–78.

[121] Xu, D., Zhou, A., Zhang, X., Wang, G., Liu, X., An, C., Shi, Y., Liu, L., & Ma, H. (2020). Understanding operational 5G: A first measurement study on its coverage, performance and energy consumption. In *ACM SIGCOMM* (pp. 479–494).

[122] Yang, M., Zhang, J., & Yu, L. (2019). Perceptual tolerance to motion-to-photon latency with head movement in virtual reality. In *Picture Coding Symposium (PCS)* (pp. 1–5).

[123] Yang, T., Jiang, J., Liu, P., Huang, Q., Gong, J., Zhou, Y., Miao, R., Li, X., & Uhlig, S. (2018). Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM* (pp. 561–575).

[124] Yao, A. C. (1978). Should tables be sorted? (extended abstract). In *Foundations of Computer Science* (pp. 22–27).

[125] Yaseen, N., Sonchack, J., & Liu, V. (2018). Synchronized network snapshots. In *ACM SIGCOMM* (pp. 402–416).

[126] Yoon, M. (2010). Aging bloom filter with two active buffers for dynamic sets. *IEEE Transactions on Knowledge and Data Engineering*, 22(1), 134–138.

[127] Yu, L., Sonchack, J., & Liu, V. (2022). Cebinae: Scalable in-network fairness augmentation. In *ACM SIGCOMM* (pp. 219–232).

[128] Yu, Z., Hu, C., Wu, J., Sun, X., Braverman, V., Chowdhury, M., Liu, Z., & Jin, X. (2021a). Programmable packet scheduling with a single queue. In *ACM SIGCOMM* (pp. 179–193).

[129] Yu, Z., Wu, J., Braverman, V., Stoica, I., & Jin, X. (2021b). Twenty years after: Hierarchical Core-Stateless fair queueing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)* (pp. 29–45).

[130] Zhang, Q., Liu, V., Zeng, H., & Krishnamurthy, A. (2017). High-resolution measurement of data center microbursts. In *ACM SIGCOMM Internet Measurement Conference (IMC)* (pp. 78–85).

[131] Zhang, Y., Liu, Z., Wang, R., Yang, T., Li, J., Miao, R., Liu, P., Zhang, R., & Jiang, J. (2021). CocoSketch: High-performance sketch-based measurement over arbitrary partial key query. In *ACM SIGCOMM* (pp. 207–222).