Compact Algorithms for Measuring Network Performance

Yufei Zheng

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: Professor Jennifer Rexford

September 2024

© Copyright by Yufei Zheng, 2024.

All rights reserved.

Abstract

For network administrators, performance monitoring provides valuable insights into network quality and security. The emergence of programmable network devices makes it possible to measure performance metrics in the data plane, where packets fly by. Running measurement tasks directly in the data plane improves efficiency, preserves user privacy, and enables the possibility of real-time actions based on the analysis.

However, programmable data planes have limited memory size and memory accesses. Meanwhile, measuring performance metrics fundamentally requires a large amount of memory resources, as each packet must be processed relative to its predecessor in the same flow.

This dissertation tackles the specific challenges that arise from running performance measurements with limited memory. The first part of this dissertation introduces new data-plane algorithms for monitoring two canonical performance metrics related to Transmission Control Protocol (TCP): delay and TCP packet reordering.

- In measuring delay distribution, existing algorithms often exhibit bias against larger delays. We present *fridges*, a novel data structure that allows for the correction of the *survivorship bias* due to hash collisions. The main idea is to keep track of the probability p that a delay sample is collected, and count it as $\frac{1}{p}$ samples. Using multiple fridges together, we can further improve the accuracy by computing a weighted average of single-fridge estimators.

- We present efficient algorithms for identifying *IP prefixes* with heavy packet reordering. First, we sample as many flows as possible, regardless of their sizes, but only for a short period at a time. Next, we separately monitor the large flows over long periods, in addition to the flow sampling. Both algorithms measure at the flow level, and aggregate statistics at the prefix level.

Existing counter-based algorithms for identifying heavy-hitter flows could also be modified to measure performance metrics. However, many of such algorithms, despite of showing good empirical performance, lack theoretical guarantees. In the second part, we present the first formal analysis for the performance of one such algorithm, the Random Admission Policy (RAP).

- We show that, for a highly skewed packet stream with k heavy flows, RAP with memory size k stores $O(k)$ heavy flows with constant probability.

Acknowledgements

It has been a privilege to work with my advisor, Jennifer Rexford. Jen put up with my stubbornness, provided all the flexibility I needed to explore, all the while offering insightful advice, whenever I needed it, on topics spanning research, careers, and life in general. Without Jen, I would not be able to finish this journey.

Several other professors have played key roles throughout these past years. I would like to thank Mark Braverman, for working with me during a tumultuous period of time, and pivotally, pointing me to Jen. It was through my brief conversations with Mark that I gradually learned to be open-minded in research, and developed my current taste for problem selection. Huacheng Yu also helped me tremendously. I must have been a huge pain, for constantly showing up in meetings having no clue what to do. Huacheng has taught me numerous techniques that, towards the end, finally make me less clueless. To Jen, Mark, and Huacheng, I need to apologize for abandoning some of the projects, and consuming your valuable time that could have led to better publications. Outside of Princeton, I want to express my gratitude to Gill Barequet from Technion. Gill always believed in me, and offered me opportunities that jump-started my research in computer science.

I have been extremely fortunate to be surrounded by the wonderful people in our office. I am particularly thankful to Xiaoqi Chen and John Sonchack, for fixing my code on multiple occasions, and helping me with installations I could never figure out for myself; to Fengchen Gong and Anchengcheng Zhou, whose presence in the office provided so much comfort to me during stressful times; to Sophia Yoo, for our deeply personal conversations, and your constant kindness; to Sata Sengupta, for our trips to Copa America, US Open, and your thoughtfulness in our discussions on many things that could have been a nightmare to talk about; and finally, to Mary Hogan, my golf instructor, my fellow F1 spectator, for all the talking, texting and laughing, it has been so gratifying to realize our friendship goes way beyond all the fun we had.

Above all, I am eternally grateful for having the best parents I could ever ask for. I come to realize, just how rare it is to be so purely happy to spend time with one's parents, and how bold my parents must have been to let me make all the decisions for myself growing up, and keep calm when I messed up. Thank you for instilling all the love, happiness, and freedom in my life.

To my parents.

Contents

Abstract . . . iii

Acknowledgements . . . v

List of Tables . . . xii

List of Figures . . . xiii

Bibliographic Notes . . . 1

1 Introduction **2**

1.1 Network Performance Monitoring . . . 2

1.1.1 Active Probing vs. Passive Monitoring . . . 3

1.1.2 Performance Metrics . . . 3

1.1.2.1 Round-trip delay . . . 4

1.1.2.2 TCP packet reordering . . . 4

1.2 Performance Monitoring in the Data Plane . . . 5

1.3 Challenges in Designing Data-plane Algorithms for Performance Measurement . . . 6

1.4 Contributions . . . 10

2 Unbiased Delay Measurement in the Data Plane **12**

2.1 Passive Delay Monitoring Problem . . . 12

2.1.1 Simple delay monitoring. . . . 12

2.1.2 Bias against large delays. . . . 14

2.1.3 The delay distribution. 15
2.2 Unbiased Delay Estimation 16
2.2.1 Correcting for survivorship bias. 16
2.2.2 Single fridge algorithm. 16
2.2.2.1 Probability of survival. 17
2.2.2.2 Approximated CDF. 19
2.2.3 Tuning the entry probability (p). 20
2.3 Expanding Beyond a Single Fridge 21
2.3.1 Using many pipeline stages per fridge. 22
2.3.2 Using multiple fridges. 24
2.4 Evaluation 27
2.4.1 Comparison with simple algorithm. 29
2.4.2 Choosing the best entry probability. 30
2.4.3 Beyond a single fridge. 32
2.5 Hardware Implementation 33
2.5.1 Implementing the fridge table 34
2.5.2 Correcting the bias 35
2.5.3 Prototype evaluation 36
2.6 Related Work 38
2.7 Conclusion 39

3 Detecting TCP Packet Reordering in the Data Plane 40
3.1 Problem Formulation: Identify Heavy Out-of-Order IP Prefixes 41
3.1.1 Flow-level reordering statistics 41
3.1.1.1 Definitions at the flow level 41
3.1.1.2 A strawman solution for identifying out-of-order heavy flows 43

3.1.1.3 Memory lower bound for identifying out-of-order heavy flows 43
3.1.2 Prefix-level reordering statistics 45
3.1.2.1 Problem statement 45
3.1.2.2 Bypassing memory lower bound 46
3.2 Traffic Characterization 47
3.2.1 Heavy-tailed size and out-of-orderness 47
3.2.2 Correlation among flows in a prefix 49
3.2.3 Packet inter-arrival times within a flow 51
3.3 Data-Plane Data Structures for Out-of-Order Monitoring 52
3.3.1 Sample flows over short periods 52
3.3.1.1 Flow sampling with array 53
3.3.1.2 Performance guarantee 55
3.3.1.3 Decrease the number of false positives 58
3.3.2 Separate large flows 59
3.3.3 Track heavy flows over long periods 60
3.4 Evaluation 61
3.4.1 Performance comparisons 62
3.4.1.1 Metrics 62
3.4.1.2 Performance evaluation 63
3.4.1.3 Performance discrepancies of the flow-sampling algorithm under different workloads 65
3.4.2 Hardware feasibility 67
3.4.3 Parameter robustness 68
3.5 Related work 70
3.6 Conclusion 71

4 An Analysis of Random Admission Policy **72**

4.1 Background 72

4.2 Deriving the Performance Bound 74

4.2.1 Preliminaries 75

4.2.2 Case (I) 77

4.2.3 Case (II) 79

4.2.4 Case (III) 80

4.2.5 Aggregating all cases 90

4.3 Conclusion 91

5 Conclusion **94**

5.1 Summary of Contributions 94

5.2 Future Directions 96

5.3 Final Remarks 97

Bibliography **99**

List of Tables

3.1 Data-plane resource usage in Tofino1. 68

List of Figures

2.1 Compared with the simple approach, the fridge tracks how many insertions each entry survives, and assigns higher weights to less likely RTTs. 13

2.2 (M=2^{10}, p=$2^{-10.4}$)-fridge produces a visibly more accurate delay CDF, compared with the simple algorithm using the same memory size M=2^{10} and expiration threshold T=2^9ms. The fridge and the simple algorithm achieves maximum relative error of 25% and 36% respectively for percentile delay queries. 30

2.3 For a fridge with $M = 2^{12}$ memory size processing synthetic trace with ground truth delay between $(0, 2^{12})$ms (top), error is minimized around entering probability $p = 2^{-9}$; for ground truth delay in $(0, 2^6)$ms (bottom), the best p increases to around 2^{-4}, which leads to a shorter average lifetime in the fridge. 31

2.4 Compared with the single fridge single stage variant, using two fridges provides more benefits as memory size M decreases. 32

2.5 The single- and two-fridge algorithms exhibit much lower estimation error for 50, 95, 99th-percentile delays, compared with the 1- and 4-stage simple algorithms, under various total memory sizes. 33

3.1 Different source prefixes send packets over different paths. Packets on a path are colored differently to show that traffic from a single prefix has a mix of packets from different flows. While flows from a single prefix may split over parallel subpaths, they do share many portions of their network resources. 41
3.2 Heavy-tailed distributions in a 5-minute campus trace. 49
3.3 A split violin plot showing prefix sizes, distributions of flow sizes in each prefix, and what fraction of reordering in a prefix comes for which flow size. A split violin of rank r refers to the r-th largest prefix in the trace. 49
3.4 Pearson coefficient on varying timescales shows that a positive correlation exists between the reordering of a flow and that of its prefix. 50
3.5 A modification of PRECISION for tracking out-of-order packets. . . 60
3.6 The flow-sampling algorithm achieves great accuracy in small memory ranges, and the hybrid scheme further improves the accuracy when more memory is available. 64
3.7 Through sending more reports to the control plane, we can decrease the false-positive rate of the flow-sampling algorithm while further improving its accuracy. 64
3.8 The accuracy of the flow-sampling algorithm is workload dependent. . 66
3.9 The effect of changing parameters on the accuracy of the flow-sampling algorithm and PRECISION. 69

Bibliographic Notes

The material presented in chapter 2 is joint work with Xiaoqi Chen, Mark Braverman, and Jennifer Rexford, and has been previously published and presented at SIAM APOCS 2022 [56]. The material presented in chapter 3 is joint work with Huacheng Yu and Jennifer Rexford, and has appeared in an arXiv paper [57]. The material presented in chapter 4 is joint work with Huacheng Yu.

Chapter 1

Introduction

In modern societies, many of us take fast and reliable networks for granted. Thanks to various services built on the Internet, such as cloud services, video chatting and streaming, the world is connected like never before. Any dip in the performance of the Internet, be it due to congestion or attacks, could be disruptive to both companies and individuals.

1.1 Network Performance Monitoring

Network administrators are responsible for maintaining organizations' network quality and security. Network performance monitoring provides valuable insights into both these aspects. Performance monitoring helps in identifying bottlenecks and latency issues that can impact network speed [46]. By analyzing performance metrics, network administrators can optimize the network to ensure it operates at peak efficiency. When performance issues arise, performance analysis helps pinpoint the network path that experiences congestion [57]. Moreover, unusual performance metrics can be indicative of security threats such as route hijacking and traffic interception [43]. Performance monitoring allows network administrators to route traffic away from malicious routes, and reconfigure legitimate routing information.

1.1.1 Active Probing vs. Passive Monitoring

Traditionally, performance monitoring often involves active probing, which is the process of injecting probe packets into the network, and gathering information about their responses. There are several aspects that make active probing undesirable. The probes add extra load on the network, and their experience is not necessarily representative of what realistic traffic experience. For example, probe packets may not be assigned the same priority as regular data packets, due to quality of service settings, leading to different treatment in terms of bandwidth allocation and latency. And, if the probes are sent and/or received by end hosts at the network periphery, the measurements might capture access-network or end-host issues, rather than the core network problems.

Therefore, passive monitoring, achieved by analyzing existing traffic in a network, is often preferable. Without needing to inject probe packets, passive monitoring does not affect the performance of a network. And by observing what packets are going through, passive monitoring provides a more realistic view of network utilization and congestion.

1.1.2 Performance Metrics

Naturally for performance, we want to understand what happens across packets from the same *flow*. A flow typically refers to a sequence of packets that belong to a communication session. Consequently, packets from the same flow share a common set of attributes, such as source and destination IP addresses, source and destination ports, and protocol type. In contrast to metrics that are based on counting traffic volume, to measure network performance, each packet must be processed in conjunction with its predecessor in the same flow.

For passive performance monitoring, it is natural to focus on Transmission Control Protocol (TCP). TCP packets contain crucial information such as source and

destination ports, sequence numbers, acknowledgement numbers and flags, which not only allows us to identify flows and compute various metrics, but also provides context for the network behavior we observe. Moreover, TCP is widely used by many of today's Internet applications, and accounts for the vast majority of network traffic. TCP performance monitoring paints a representative picture of what most packets experience.

In this thesis, we consider two canonical performance metrics related to TCP: round-trip delay, and TCP packet reordering.

1.1.2.1 Round-trip delay

The round-trip delay refers to the time difference between sending a request, and receiving its corresponding response. For example, the two-way delay for a TCP handshake is the time between a client sending TCP SYN packet to a server and receiving the corresponding TCP SYN/ACK packet, while the delay for a DNS lookup is the time between a DNS query packet and its corresponding response packet. We can also measure the time between the beginning of a TCP flow (SYN packet) and its ending (FIN or RST packet), which characterizes the duration of a flow that corresponds to the delay an application experienced when downloading a file.

Fine-grained information about network delays is especially useful in practice. It is common for an Internet Service Provider (ISP) and its clients to specify a target delay distribution in their Service-Level Agreements (SLAs) [52, 53]. For example, an ISP might need to determine if more than 5% of TCP handshake delay exceeds 50ms.

1.1.2.2 TCP packet reordering

A reordered packet is a packet whose sequence number is out-of-order, with respect to its predecessor in the same flow. Transmission Control Protocol (TCP) perfor-

mance problems are often associated with packet reordering. Packet loss, commonly caused by congested links, triggers TCP senders to retransmit packets, leading these retransmitted packets to appear out of order. Also, the network itself can cause packet reordering, due to malfunctioning equipment or traffic splitting over multiple links [40]. TCP overreacts to inadvertent reordering by retransmitting packets that were not actually lost and erroneously reducing the sending rate [8, 40]. In addition, reordering of acknowledgment packets muddles TCP's self-clocking property and induces bursts of traffic [7]. Perhaps more strikingly, reordering can be a form of denial-of-service (DoS) attack. In this scenario, an adversary persistently reorders existing packets, or injects malicious reordering into the network, to make the goodput low or even close to zero, despite delivering all of the packets [1, 25].

1.2 Performance Monitoring in the Data Plane

For passive performance monitoring, traditional methods often involve two separate steps: creating copies of the traffic out of the data plane, and analyzing the traffic offline. With the emergence of programmable network switches, we now have the option to do analysis directly in the data plane, as packets fly by. Simple packet-reordering statistics can be collected directly as part of high-speed packet processing, given software platforms like eBPF [47] and DPDK [49], smart network interface cards [41, 55], and ASIC-based switches [10, 26, 42]. With flexible parsing, we can extract the header fields we need to analyze the packets in a flow. Using arrays or dictionaries, we can keep state across successive packets of the same flow. In addition, simple arithmetic operations allow us to detect reordering, compute the delay, and tally count.

Running measurement tasks directly in the data plane has several benefits. By processing packets on the fly, fine-grained analysis no longer comes with significant

data collection overhead, and sensitive user data never needs to be transported outside of the data plane. Moreover, it enables real-time actions on each packet based on the analysis.

However, processing packets efficiently for high link speeds imposes significant constraints on memory:

- **Memory size:** Modern data planes have a limited amount of memory, especially compared to the number of concurrent flows on high-speed links.
- **Memory accesses:** Since memory bandwidth has not kept pace with link bandwidth, modern data planes can only access memory a few times per packet.

Plus, network devices perform other tasks—packet forwarding, access control, and so on—that demand a share of the already limited memory resources. Furthermore, since the data plane has limited bandwidth for communicating with the control-plane software, we cannot offload monitoring tasks to the control plane. As such, we need to design compact data structures that work within these constraints.

1.3 Challenges in Designing Data-plane Algorithms for Performance Measurement

The constraints of the programmable data plane often makes it impossible for any algorithm to generate exact answers. Fortunately for many measurement tasks, approximate answers suffice, and people often turn to the vast literature on streaming algorithms for inspiration. However, existing works have mainly focused on volume-based metrics. For instance, to identify the k heaviest flows in a packet stream, there are both sketch-based ([19, 13]) and counter-based ([37, 20, 6]) algorithms. A sketch-based algorithm keeps an array of approximate counters, and performs updates based on hash values, while a counter-based algorithm keeps track of both the identifiers

and the approximate counts, but only for a subset of the traffic. Counter-based algorithms are of particular interest, since they also allow the possibility of measuring performance metrics. By storing identifiers in the data structure, these algorithms can be extended to match packets from the same flow, thus giving, and compute performance metrics for heavy flows on the way [57].

Among counter-based algorithms, SpaceSaving [37] has drawn the most attention in the networks community, due to its simplicity and asymptotic optimality in reporting frequent elements. Random Admission Policy (RAP) [6] is a notable variant of SpaceSaving. Through incorporating the idea of probabilistic insertions, RAP exhibits higher accuracy over SpaceSaving in both identifying heavy flows and estimating the sizes of these flows, especially in small memory regimes. However, due to the process of finding the minimum counter, both SS and RAP require per packet memory accesses linear in the memory size, as a result, neither can be directly implemented on programmable switches. Hardware friendly variants of SpaceSaving and RAP include HashPipe [44] and PRECISION [4]. To reduce the number of memory accesses, HashPipe uses a pipeline of d hash-indexed arrays, with a small constant d, and only require one memory access in each array. PRECISION combines ideas from RAP and HashPipe and achieved the best empirical accuracy for identifying heavy flows in the data plane.

To use these algorithms as part of the performance monitoring, we also want to know, given a memory size, how many heavy flows are we monitoring. As opposed to sketch-based algorithms, which the theory community has extensively studied, all the above counter-based algorithms beyond SpaceSaving, while showing promising empirical performance, lack rigorous theoretical guarantees. Proving theoretical results for these algorithms turned out to be highly nontrivial. In order to understand the performance of PRECISION, the data-plane state-of-the-art, we need techniques to analyze two of its algorithmic components: (1) random admission, as in the RAP al-

gorithm, and (2) approximating the global minimum using the minimum of d counter values. In Chapter 4, we present our theoretical results on the performance of RAP, which serves as a valuable first step towards providing bounds for PRECISION.

The lack of theoretical guarantees is clearly not the only problem, as focusing on heavy flows using counter-based algorithms does not always suffice for the purpose of performance monitoring. In monitoring TCP packet reordering, whilst heavy flows might make up most traffic, they do not necessarily give us a full picture of flows from all sources to a destination. By focusing solely on heavy flows, we might overlook congestion that is happening on parts of the network. In the case of round-trip delay, where we need to match pairs of requests and responses, there is not even the notion of heaviness in this context. These examples highlights the need to design data-plane algorithms specialized in measuring performance metrics. Next we delve into the specific challenges we encounter in monitoring delay and packet reordering in the data plane.

Mitigating bias in estimating delay distribution. To measure delay in the data plane, Chen et al. [15] proposed a simple scheme which saves the request ID and a timestamp into an array, applying a hash function over the ID to obtain an array index. When the corresponding response arrives, we use the hash function again to look at the same index, and if the matching request exists, calculate this request's delay and update the statistics of the delay distribution.

However, not all requests eventually receive a response, and those orphaned requests without a response create a dilemma. A new request being inserted may suffer from a hash collision with an existing request. If we discard the new request and keep the existing one, the array will soon fill up with stale, orphaned requests. But simply overwriting upon every hash collision is even worse: to produce a delay sample, the request must stay in the array for long enough without being overwritten.

For large-delay pairs, the request needs to survive a large number of new insertions into the data structure without a hash collision. Consequently, more small-delay samples are produced, while large-delay pairs are undersampled and therefore *biased* against. This phenomenon is especially problematic in applications like verifying SLAs or measuring tail latency, since larger delays are exactly the anomalies we hope to catch.

Previous efforts to mitigate the bias against larger delays include finding a middle ground between favoring the existing entries and overwriting aggressively. Chen et al. [15] used a large expiration threshold that corresponds to the 99th-percentile delay in the network, and only evicts upon hash collision when a record gets too old. Yet it is hard to accurately choose such a threshold, particularly when the prior distribution is unknown. Moreover, setting such a conservative threshold means orphaned requests stay too long in the memory, reducing the number of valid matches.

In Chapter 2, we opt for an algorithmic approach in mitigating the bias. The main idea is that, if a delay sample is collected with probability p, to get an unbiased count of this sample, we can count it as $\frac{1}{p}$ samples in a tally. By keeping track of the number of insertions each request survives, we are able to systematically account for the bias each delay sample encounters.

Circumventing memory lower bound in detecting TCP packet reordering. TCP packet reordering are intrinsically small probability events. Identifying flows with heavy reordering fundamentally requires a large amount of memory, which the programmable data plane cannot afford. To make it feasible, Liu et al. [35] considered the problem of detecting flows with a large number of reordered packets, with the assumption that reordered packets always arrive within some fixed period of time. Not only does this assumption appear unnatural, the algorithm also requires a priority queue to maintain the set of the most recent received packets, which violates the

memory access constraint. The authors have to further use a two-way cuckoo table to approximate the priority queue. However, unless the size of the table is large, the effect of hash collisions cannot be ignored.

In monitoring packet reordering, we notice that identifying every affected flow is not necessarily what is important for network administrators. Packet reordering is typically a property of a network path, due to congested or flaky links. As such, it is useful to report reordering at a coarser level, such as to identify the IP prefixes associated with performance problems. Since routing is determined at the IP prefix level, a network administrator could choose to route the traffic for an IP prefix through providers whose paths are not experiencing significant reordering. However, this does *not* obviate the need to maintain state for at least some flows, as TCP packet reordering is still a flow-level phenomenon. Fortunately, we can identify prefixes with heavy packet reordering without needing to track *all* of the flows, because packets traversing the same path at the same time are often correlated in their out-of-orderness. In Chapter 3, we show how this correlation helps in saving memory.

1.4 Contributions

This dissertation focuses on the design and analysis of compact algorithms for measuring network performance in the data plane. In the first part of this dissertation, we consider the problems of measuring delay and TCP packet reordering in the data plane, and leverage probabilistic techniques to work with hardware constraints.

- In Chapter 2, We present *fridges*, a novel data structure that corrects for the *survivorship bias* due to hash collisions, producing *unbiased* estimates of the delay distribution. The key idea is to consider a sample that was lucky enough to survive many insertions into the data structure as a representative for other similar samples that did not survive. We also show how to combine results

from multiple fridges, each optimized for a different range of delays, for further accuracy gains. Simulation experiments show our design outperforms prior work using naive hash-indexed arrays, achieving 2x-4x memory saving. We implement a prototype P4 program running on the Intel Tofino programmable switch, using only moderate hardware resources.

- In Chapter 3, we present efficient algorithms for identifying *IP prefixes* with heavy packet reordering under memory restrictions. First, we sample as many flows as possible, regardless of their sizes, but only for a short period at a time. Next, we separately monitor the large flows over long periods, in addition to the flow sampling. In both algorithms, we measure at the flow level, and aggregate statistics and allocate memory at the prefix level. Our simulation experiments, using packet traces from campus and backbone networks, and our P4 prototype show that our algorithms correctly identify 80% of the prefixes with heavy packet reordering using moderate memory resources.

In the second part of this dissertation, we revisit RAP, a counter-based algorithm for identifying heavy flows, through a purely theoretical lens, hoping to not only provide performance guarantees for this particular algorithm, but also shed light on the theoretical performance of other data-plane algorithms that built on RAP (d-way RAP [6], PRECISION [4]).

- In Chapter 4, we consider the performance of RAP on highly skewed streams. Specifically, we show that, for a highly skewed packet stream with k heavy flows, RAP with memory size k stores $O(k)$ heavy flows with constant probability.

Chapter 2

Unbiased Delay Measurement in the Data Plane

In this chapter, we propose a data-plane algorithm that produces a provably unbiased delay distribution, specifically designed for programmable switches using the Protocol Independent Switch Architecture (PISA) [9]. Our algorithms tackle the bias by keeping track of the probability of getting each sample, and applying a correction factor inversely proportional to this probability when computing the distribution.

2.1 Passive Delay Monitoring Problem

In this section, we first introduce the delay monitoring problem, and a "simple" data-plane solution. Then, we explain why the simple algorithm is biased against large delays, and define our goal of producing an unbiased estimate of the delay distribution.

2.1.1 Simple delay monitoring.

In delay monitoring, a stream of packets represents a stream of *requests* and *responses*, where we hope to match a response with its corresponding request, to generate use-

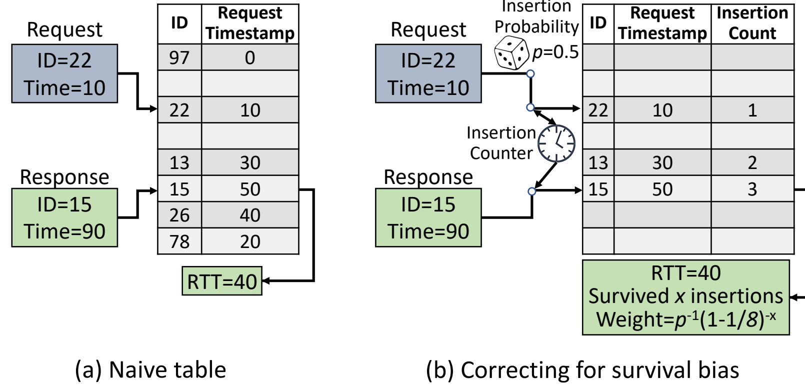


Figure 2.1: Compared with the simple approach, the fridge tracks how many insertions each entry survives, and assigns higher weights to less likely RTTs.

ful delay statistics. Examples of request/response pairs include NTP request and response, DNS request and response over UDP, TCP handshake pairs (between SYN and SYN-ACK packets), and TCP flow duration (between TCP SYN and FIN/RST packets). In real-world applications, some requests never receive a response, for instance due to server failures or cyber attacks. We assume each request and its potential response carry the same identifier (ID) unique for the pair that allows matching, if the response exists. For example, we can extract IP addresses, port numbers, and sequence numbers from a TCP SYN packet, and match them with that of a TCP SYN-ACK packet. After matching a response to a request, the resulting delay sample contributes to some larger analysis of the delay distributions.

Today's programmable switches using the PISA pipeline architecture have tens of megabytes of register memory that we can read or write in the data plane, while processing individual network packets. To meet the strict speed requirements of line-rate packet processing (Terabits per second), the switch imposes several constraints on what packet-processing logic we can implement. The pipeline has a fixed number of stages, therefore each packet is processed within a constant number of clock cycles. To avoid memory hazard due to concurrent memory access, all register memory arrays are allocated to a specific pipeline stage, and we can only access one index of each register memory array when processing each packet.

Given the memory access constraints, we cannot implement traditional hash tables with sophisticated collision resolution logic. The *hash-indexed array* is a good candidate data structure to implement a simple algorithm for matching packet pairs and continuously generating delay samples, as illustrated in Figure 2.1(a).

1. **Request:** For each request, we compute $hash(ID)$ to find an array index and write in the request ID and current timestamp. To avoid filling the memory with orphaned requests, we favor the new request upon hash collisions, by overwriting any existing request in the same array index.

2. **Response:** For each response, we again compute $hash(ID)$ as the index. If there is a request with the same ID, we calculate the difference t of their timestamps, report a delay sample t, and remove the request; otherwise, there is no match and no sample is recorded.

2.1.2 Bias against large delays.

The limited memory in the data plane poses a significant challenge. A higher delay means the request needs to stay in memory longer while waiting for its response to arrive, which translates to using more memory at any given point. Since we choose to always overwrite existing requests upon hash collisions, when memory is limited a request is more vulnerable to eviction the longer it remains in memory. This leads to a bias against larger delays.

At first glance, we could solve this problem by favoring *existing* requests on collisions. However, a response may never arrive, causing orphaned requests to consume the memory. An alternative is to set an expiration threshold and evict a record if the request stays in the data structure longer than the threshold. This method seemingly achieves a balance between overwriting aggressively and favoring existing requests conservatively, but in practice it is hard to find the right threshold [15]. A

large threshold leads to an array full of stale requests, while a small threshold causes bias against larger delays, as such requests are quickly evicted before their responses arrive.

2.1.3 The delay distribution.

In practice, we often hope to combine individual delay reports to generate some statistics of interest. In this work, we specify the output of our algorithms to be an approximated distribution of delays of all request/response pairs in the stream, and in particular we can look at the Cumulative Distribution Function (CDF) of the delay. We note that the estimation error of many real-world delay metrics commonly seen in Service Level Agreements (SLAs) can be translated to the difference between estimated and ground truth delay CDF curves. For example, the error in measuring 95th-percentile delay is the horizontal distance between the CDF curves at $y = 95\%$, while the error in measuring the fraction of RTTs above 40ms is the vertical distance between the CDF curves at $x = 40ms$. We therefore formalize the problem as follows:

Definition 1 (Delay CDF)**.** Given a stream of requests and responses with identifier ID_i, timestamp t_i, and type $c_i \in \{\texttt{req}, \texttt{resp}\}$ differentiating requests and responses, we pair a request i with its response i' when $ID_i = ID_{i'} \wedge (c_i, c_{i'}) = (\texttt{req}, \texttt{resp})$. Each request has a unique ID, and has at most one matching response. The *delay* of a request/response pair (i, i') is defined to be $t_{i'} - t_i$. Let $f(t)$ denote the number request/response pairs in the stream with delay t, and $F(t) = \frac{\sum_{\tau<t} f(\tau)}{\sum_t f(t)}$ the ground truth CDF. On seeing the entire stream, we hope to output a close approximation $\hat{F}(t)$ of $F(t)$.

In this work, we do not intend to distinguish between the delay CDF of all packets versus some subset of packets, e.g., those of one particular flow or application. The algorithms presented in § 3 are general enough to be applied in either case.

2.2 Unbiased Delay Estimation

Contrary to previous work [15], we do not attempt to mitigate bias by fine-tuning the frequency of overwriting records. Instead, we overwrite aggressively to take in all new requests, and correct for bias as samples are collected. We start this section with the idea of bias correction (§ 2.2.1). Next we describe the single *fridge* algorithm and its mechanism of applying correction factors (§ 2.2.2).

2.2.1 Correcting for survivorship bias.

The phenomenon that larger delays are undersampled traces back to the step where any report, whether the delay is large or small, is considered as one sample. Instead, we should cherish each sample with a high delay—it should account for not only itself, but other requests with the same delay that are evicted before their responses arrived.

Thus, we define a correction factor to counter this survivorship bias. If a sample has a probability q to survive without being evicted, upon seeing its report we set the correction factor to $\frac{1}{q}$, and count it as $\frac{1}{q}$ samples. This way, evictions from the data structure no longer lead to biases, since the collected large-delay reports can compensate for the missing ones.

To track a sample's survival probability, we can analyze the number of insertions between the time this sample's request is inserted and its response arrives. Each insertion has a small probability to evict the request due to hash collision, thus the sample's survival probability diminishes as there are more insertions before the response.

2.2.2 Single fridge algorithm.

Now we discuss the design of a single *fridge* data structure, inspired by humans checking how long a grocery item has stayed in a refrigerator when taking it out.

We maintain a global insertion counter and stamp its count alongside every inserted request. Subsequently, when the response arrives we find the number of insertions that happened between the request-response pair and calculate a correction factor,as illustrated in Figure 2.1(b).

We formally compute the correction factor using the inverse of the probability of a sample being collected. We define a data structure, *fridge*-(M, p), to be a hash-indexed array of size M equipped with an entering probability p, which dictates that each new request is inserted into the array with probability p (and discarded with probability $1 - p$).

2.2.2.1 Probability of survival.

A request's probability of survival decreases as it stays longer in the fridge, measured by the number of other requests being inserted between this request and its response. We track this number by maintaining a global insertion counter. For each new request, an existing request in the fridge has a $\frac{p}{M}$ chance to suffer from a hash collision and be evicted; thus requests stay for roughly $\frac{M}{p}$ insertions, which we call the *average lifetime* of a fridge.

We define x as the number of insertions that happened between a request and its matching response, which can be calculated by subtracting the recorded value of the global insertion counter (at request time) from its current value (at response time). Consider a sample with delay t that survives x insertions between its request and response: it must survive three independent events: (1) its request enters the fridge (with probability p), (2) the request survives the next x insertions into the array of size M, and (3) the sample has delay t in the underlying ground truth distribution. Denote $f(t)$ as the true number of samples with delay t, and n the true number of

samples in the stream, we have

$$\mathbb{P}[\text{getting a sample with delay } t] = p \cdot \left(1 - \frac{p}{M}\right)^x \cdot \frac{f(t)}{n}. \tag{2.1}$$

Note that the first two terms indicates a sample's survival probability $q = p \cdot \left(1 - \frac{p}{M}\right)^x$. We therefore set a correction factor of $\frac{1}{q} = p^{-1} \cdot \left(1 - \frac{p}{M}\right)^{-x}$ for each report we observe. After seeing the entire stream, we get a collection of reports, where each contains its delay t_i and its correction factor, in the form of a 2-tuple: $\left(t_i, p^{-1}\left(1 - \frac{p}{M}\right)^{-x_i}\right)$, with x_i being the number of insertions the sample survives. From the reports we can obtain an estimator of $f(t)$ for all t, denoted as $\hat{f}(t)$, by summing up all correction factors that correspond to t.

We show in Lemma 2.1 that $\hat{f}(t)$ is an unbiased estimator of $f(t)$ with bounded variance.

Lemma 2.1. *Let Y_i be in indicator of sample i, $Y_i = 1$ if sample i has delay t, and $Y_i = 0$ otherwise, then*

$$\hat{f}(t) := \sum_{i \in [n]} p^{-1} \left(1 - \frac{p}{M}\right)^{-x_i} Y_i$$

is an unbiased estimator of $f(t)$, and

$$\mathrm{Var}[\hat{f}(t)] = \frac{f(t)}{n} \sum_{i \in [n]} \left(p_1^{-1}\left(1 - \frac{p_1}{M}\right)^{-x_i} - \frac{f(t)}{n}\right).$$

Proof. Fix any t, and from Eq. 2.1,

$$\mathbb{E}[Y_i] = \mathbb{P}[\text{get a sample with delay } t] = p\left(1 - \frac{p}{M}\right)^{x_i} \frac{f(t)}{n},$$

$$\mathrm{Var}[Y_i] = p\left(1 - \frac{p}{M}\right)^{x_i} \frac{f(t)}{n}\left(1 - p\left(1 - \frac{p}{M}\right)^{x_i} \frac{f(t)}{n}\right).$$

By definition, $\hat{f}(t) = \sum_{i\in[n]} p^{-1}\left(1-\frac{p}{M}\right)^{-x_i} Y_i$, then

$$\mathbb{E}[\hat{f}(t)] = \sum_{i\in[n]} p^{-1}\left(1-\frac{p}{M}\right)^{-x_i}\mathbb{E}[Y_i] = f(t).$$

Since indicator Y_i's are independent,

$$\begin{aligned}\mathrm{Var}[\hat{f}(t)] &= \sum_{i\in[n]} p^{-2}\left(1-\frac{p}{M}\right)^{-2x_i}\mathrm{Var}[Y_i] \\ &= \frac{f(t)}{n}\sum_{i\in[n]}\left(p^{-1}\left(1-\frac{p}{M}\right)^{-x_i} - \frac{f(t)}{n}\right).\end{aligned} \tag{2.2}$$

□

2.2.2.2 Approximated CDF.

We estimate the CDF $\hat{F}$ by aggregating and normalizing the individual point-wise estimate $\hat{f}$ using discrete integration,

$$\hat{F}(t) := \sum_{i,t_i\leq t}\frac{\hat{f}(t_i)}{\sum_i \hat{f}(t_i)}, \forall t\in\{t_i\}_i. \tag{2.3}$$

To compare $\hat{F}$ with the ground truth CDF F, it is convenient to think of $\hat{F}$ and F as continuous functions. Throughout this work, we assume linear interpolation for both $\hat{F}$ and F.

Though $\hat{f}(t_i)$ is unbiased, and so is $\sum_i \hat{f}(t_i)$ by linearity of expectation, it is important to note that the unbiasedness is not preserved under division. Therefore, $\hat{F}$ is not necessarily unbiased. Nonetheless, the single fridge algorithm improves on the simple algorithm considerably in terms of reducing bias (§ 2.4.1). Later in § 2.4.2, we show how the choice of p affects accuracy of the estimated CDF and how to identify a good p based on the fridge size and the maximum delay.

2.2.3 Tuning the entry probability (p).

Given a fixed M, the entry probability p needs to be strategically chosen such that the (M, p)-fridge operates optimally. Recall that on average each request survives M/p insertions (the "average lifetime" of the fridge) before being evicted.

- If delays are short such that most request-response pairs we are measuring are less than M/p insertions apart, almost all responses will arrive before any collision happens to the request. For most samples, the insertion counter x will be 0, therefore the correction factor is always p^{-1} with no effect of the survivorship bias. In short, memory is "underutilized".

- If the average lifetime M/p is too short (there are often more than M/p insertions between request-response pairs), most requests cannot survive long enough until their responses arrive — a request will suffer from hash collisions with overwhelming probability. The very lucky requests that did survive will have very large correction factors, leading to an enormous estimation variance. Although our estimation is still unbiased in this case, it has little practical value. Under this scenario, memory is "oversubscribed".

- Ideally, the fridge operates in the regime with its average lifetime M/p aligning with the number of insertions between most request-response pairs. In this case, x (the number of insertions survived by a sample) is close to M/p.

We note that in the ideal regime, given a large M and the assumption $x \approx M/p$, typical delay samples should have a correction factor in the order of $p^{-1}\left(1 - \frac{p}{M}\right)^{-M/p} \approx e/p$. Thus, medium-delay samples weights about e times more than samples with very short delay.

Inferring the number of insertions (x). We further observe that, with a constant traffic rate, the number of insertions survived by a sample (x) is proportional to the delay observed by this sample. Therefore, it is possible to not record an

insertion counter alongside each request, and use the delay to approximately recover x and calculate the correction factor. However, in many network applications, delay is correlated with short-term spikes in traffic rate (which cause transient congestion), which are precisely the anomalous events we want to scrutinize. Thus, we still opt to record the exact insertion counters.

Regarding the number of table entries (M). We note that although we expect a fridge to have thousands of entries in practice, in the extreme case we require $M \geq 2$, as we estimate each survived request's survivorship bias using the hash collisions happened in the other $M-1$ entries. A single-entry fridge with $M = 1$ is a degenerative case, as we cannot observe any "survivorship bias" (all samples have $x = 1$). In the special case of $M = 2$, the surviver's correction factor doubles every time the other entry is replaced due to a new insertion. This estimation clearly has a large variance, and a larger M is preferred — the estimate of survivorship bias becomes more accurate as we observe the fate of more requests stored in other entries.

As the fridge size M is limited by the memory available under the given hardware environment, given a fixed M it is important to provision the fridge with the appropriate entry probability p, based on the traffic rate (number of requests per second) and the delay we expect to observe. In § 2.4.2 we evaluate the effect of choosing p on a fridge's accuracy, and demonstrate the fridge has some tolerance regarding this choice.

2.3 Expanding Beyond a Single Fridge

In this section, we discuss how to extend the single fridge design to possibly achieve better measurement accuracy. We first discuss why a simple design using multiple hardware pipeline stages to build a single fridge will, surprisingly, hurt high-delay

samples (§ 2.3.1). Then, we show how to build multiple fridges and combine their output correctly (§ 2.3.2).

2.3.1 Using many pipeline stages per fridge.

On a PISA programmable switch [9], we are limited to accessing only one index per register array when processing a packet. Algorithms often span multiple pipeline stages and allocate multiple register arrays to improve performance. For example, the delay measurement algorithm in [15] achieves the best accuracy when the same total memory size is split into 4-6 arrays in separate pipeline stages, each indexed with a different hash function.

An insertion checks one location per stage, and only fails when it suffers hash collisions on all these locations. Compared with using only a single stage, the multi-stage design achieves better memory utilization and lowers the probability of failed insertions, thanks to the "power of two choices" phenomenon.

Naturally, when running the fridge algorithm, we could also consider a multi-stage design. Similar to [15], we could use multiple memory arrays indexed by different hash functions. However, to deal with stale requests, upon a hash collision we must favor inserting the new request and evict the existing request. We could implement a scheme similar to HashPipe [**?**], where the request evicted in the first stage is inserted again in the second stage, and likewise for later stages. The propagation stops when an empty array slot is encountered, and a request is finally abandoned when it is evicted from the very last pipeline stage.

To produce an unbiased delay estimator, we need to find the right correction factor under this design. We note that a request in the first few stages is not at risk of eviction, thus its survival has probability one; only a request appearing in the last stage needs a correction factor for its survival probability, based on x_i, the number of insertions it survived in the last stage.

Unfortunately, this design works poorly for samples with larger delay. Assume we build a D-stage fridge with total memory M and entry probability p, we can calculate the number of insertions a request can survive in the fridge as a probability distribution. Note that each stage has M/D entries, and for simplicity we assume the memory is full of requests with no empty slots. For one insertion, a request currently in stage s has probability $p \cdot D/M$ suffering from a collision and move to stage $s+1$. Thus, the number of insertions a request survives in stage s follows the geometric distribution $l_s = Geo(p \cdot D/M)$. We can thus write the lifetime distribution of the D-stage fridge, i.e., the total number of insertions survived before a request is evicted from the last stage, as $L_D = \sum_{s=1}^{D} l_s = \sum_{s=1}^{D} Geo(p \cdot D/M)$, with expectation $D \cdot \frac{M}{p \cdot D} = M/p$. Meanwhile, for an ordinary fridge, we can simply plug in $D = 1$: its lifetime distribution is simply $Geo(p/M)$ with expectation M/p.

Although items in the multi-stage fridge have the same expected lifetime $\frac{M}{p}$ as those in a single-stage fridge, its lifetime distribution is the sum of D i.i.d. geometric variables, which is more concentrated and has a lighter tail than a single geometric variable. This is to say, for a large delay $T > M/p$ and $D > 1$, we have

$$\mathbb{P}[Geo(p/M) \geq T] > \mathbb{P}[\sum_{s=1}^{D} Geo(p \cdot D/M) \geq T]. \tag{2.4}$$

Therefore, requests with delay higher than the fridge's average lifetime have a much smaller survival probability.

This phenomenon is most obvious when we consider the extreme case: with $D = M$ stages each having array size 1, the fridge essentially becomes a FIFO queue, and the lifetime distribution becomes very narrowly concentrated. With every request spending almost the same time in fridge, a request whose delay is higher than the average lifetime has almost no chance to survive. Instead, we want the exact opposite: the lifetime distribution should be heavy-tailed, so requests have some probability of

staying in the fridge for much longer than $\frac{M}{p}$ insertions, so our fridge can collect some samples for large delay. Thus, analytically the multi-staged design performs poorly; we have also verified this phenomenon empirically.

Thus, we should never use a multi-stage fridge. When we need to utilize more memory than the capacity of a single pipeline stage, we should simply merge the memory across multiple stages into one large logical hash-indexed array and build a single-stage fridge. We also note that proposals like dRMT [16] would enable stateful memory allocation across stages, so a simple one-stage algorithm can use the entire stateful memory directly.

2.3.2 Using multiple fridges.

Requests in one fridge have an average lifetime $\frac{M}{p}$, which can be adjusted to fit the typical delay of the input traffic stream for higher accuracy. However, internet traffic exhibits a wide range of delays, due to different geographic distances, server behavior, and congestion conditions. Thus, a single fridge targeting a particular $\frac{M}{p}$ may be inadequate.

To cover a wide range of delays, we split the memory into N fridges with size $M_1 + \cdots + M_N = M$. Requests and responses are directed to one of the fridges via a hash function, while each fridge has its own entry probability $p_1 + \cdots + p_N < 1$ and targets a different average lifetime $M_1/p_1, \ldots, M_N/p_N$. When a response matches in fridge k, we calculate the correction factor $p_k^{-1}(1 - p_k/M_k)^{-x_i}$ based on fridge k's entering probability p_k and the probability for surviving x_i insertions in this fridge.

Since different fridges have different average lifetime, they have different variance when estimating various ranges in the delay distribution. We need to combine their output strategically to produce the final estimated delay distribution with minimum estimation variance. In § 2.4.3, we demonstrate that using multiple fridges can indeed produce more accurate estimates than a single fridge when memory size is limited.

We now describe the process of combining multiple fridge's output using the Inversed Variance Weighting method [17] in more detail.

Variance of each fridge. For a sample coming from fridge k (with size M_k and entry probability p_k) that survives x insertions, we set its correction factor as $p_k^{-1}\left(1-\frac{p_k}{M_k}\right)^{-x}$ following Lemma 2.1. Summing up all correction factors for a t coming out of fridge k gives an unbiased estimator $\hat{f}_k(t)$ for $f(t)$, the true number of samples with delay t, where the unbiasedness follows again from Lemma 2.1. Let k_i be the index of the fridge sample i comes from, then the variance of $\hat{f}_k(t)$ follows directly from Eq. 2.2,

$$\mathrm{Var}[\hat{f}_k(t)] = \frac{f(t)}{n} \sum_{i\in[n],k_i=k} \left(p_k^{-1}\left(1-\frac{p_k}{M_k}\right)^{-x_i} - \frac{f(t)}{n}\right). \tag{2.5}$$

The weighted average of estimators. Similar to classical sketching algorithms such as CountMin [18] and CountSketch [14], in the multi-fridge algorithm, we keep a set of N basic unbiased estimators $\{\hat{f}_1(t), \hat{f}_2(t), \ldots, \hat{f}_N(t)\}$ for $f(t)$, each coming from a fridge. Since the variance of each individual estimator could be large, we combine them to get a better estimator. However, unlike [18, 14], our basic estimators have different variance, so instead of simply taking their min or the median, we leverage this fact to compute a weighted average of the estimators.

It is well-known in statistics [17] that given N unbiased estimators with bounded variance, we can set weights optimally so that the weighted average of these estimators has the minimum possible variance.

Theorem 2.2. *Given N unbiased estimators $\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_N$ with bounded variance $\mathrm{Var}[\hat{f}_1], \mathrm{Var}[\hat{f}_2], \ldots, \mathrm{Var}[\hat{f}_N]$ respectively, the set of N weights $\{w_1, w_2, \ldots, w_N\}$ with $w_k = \frac{\frac{1}{\mathrm{Var}[\hat{f}_k]}}{\sum_k \frac{1}{\mathrm{Var}[\hat{f}_k]}}$, $k \in [N]$ minimizes the variance of the combined unbiased estimator $\sum_k w_k \hat{f}_k$.*

To combine estimators using Theorem 2.2, we need to associate each sample with a weight, so as to calculate weights $\{w_1, w_2, \ldots, w_N\}$ for a fixed delay t. Therefore, the report sample i in the multi-fridge algorithm becomes a 4-tuple that keeps the weight of the sample, to be determined next, as well as the fridge index k_i, on top of the delay t_i and the correction factor as in the single fridge case.

However, Theorem 2.2 cannot be directly cast into our multi-fridge setup, since we do not know variance $\mathrm{Var}[\hat{f}_1]$, $\mathrm{Var}[\hat{f}_2]$, …, $\mathrm{Var}[\hat{f}_N]$ exactly. We work around this issue by approximating weights w_k for each fridge k. From Eq. 2.5, we can safely focus on estimating

$$\frac{f(t)}{n} \sum_{i \in [n], k_i = k} p_k^{-1} \left(1 - \frac{p_k}{M_k}\right)^{-x_i} \tag{2.6}$$

in the variance, since $\frac{f(t)}{n} << 1 << p_k^{-1}\left(1 - \frac{p_k}{M_k}\right)^{-x_i}$. Yet, it would be false to assume the $\frac{f(t)}{n}$ factor outside of the summation cancels out in w_k, so we can obtain the rest of (2.6) precisely from summing over correction factors from all reports. This would have produced an underestimation, since n is the true number of samples, and the fridges can only report fewer than n samples due to hash collisions.

We therefore use the unbiased estimator of (2.6),

$$\sum_{i \in [n]} p_k^{-2} \left(1 - \frac{p_k}{M_k}\right)^{-2x_i} Z_i, \tag{2.7}$$

where indicator $Z_i = 1$ if sample i comes from fridge k and has delay t, and $Z_i = 0$ otherwise. An argument similar to that in Lemma 2.1 suffices to verify the unbiasedness of (2.7).

Putting all elements together, the report of a sample with delay t that survives x insertions in fridge k, the 4-tuple (fridge index, delay, correction factor, weight

factor), has the following form

$$\left(k, t, p_k^{-1}\left(1-\frac{p_k}{M_k}\right)^{-x}, p_k^{-2}\left(1-\frac{p_k}{M_k}\right)^{-2x}\right).$$

We obtain estimator $\hat{f}_k(t)$ by summing over all correction factors of samples with RTT t from fridge k,

$$\hat{f}_k(t) := \sum_{i\in[n]} p_k^{-1}\left(1-\frac{p_k}{M_k}\right)^{-x_i} Z_i.$$

Denote the unbiased estimator of (2.6) as $\hat{V}_k(t)$, by (2.7),

$$\hat{V}_k(t) := \sum_{i\in[n]} p_k^{-2}\left(1-\frac{p_k}{M_k}\right)^{-2x_i} Z_i.$$

Finally, we obtain our multi-fridge estimator $\hat{f}(t)$ through a weighted average of estimators from all fridges $\{\hat{f}_1(t), \hat{f}_2(t), \ldots, \hat{f}_N(t)\}$,

$$\hat{f}(t) := \sum_k \hat{w}_k(t)\hat{f}_k(t), \text{ where } \hat{w}_k(t) = \frac{\frac{1}{\hat{V}_k(t)}}{\sum_k \frac{1}{\hat{V}_k(t)}}.$$

This concludes the process of combining the output of multiple fridges. Note that despite of the approximation, we always have $\sum_{k\in[N]} \hat{w}_k(t) = 1$, and $\hat{f}(t)$ is hence unbiased for being a convex combination of unbiased estimators.

2.4 Evaluation

In this section, we use real-world and synthetic traffic traces to show that the fridge algorithm can effectively reduce bias in delay measurement, compared with prior works. To experiment with different parameter settings, we run all tests using a Python-based simulator. We discuss and evaluate a prototype running on hardware programmable switches in § 2.5.

Distance metric. We evaluate the accuracy of single- and multi-fridge algorithms by computing the distance between the ground truth CDF $F(t)$ and the estimated CDF $\hat{F}(t)$ computed by our algorithms. As discussed in § 2.1.2, the CDF is closely related to criteria specified in SLAs. For example, "95th-percentile delay" is where the delay CDF curve crosses $y = 95\%$. Since real-world delays vary widely, absolute error is not an effective metric; we instead look at the relative error of percentile delay queries: $\left|\log_2\left(\frac{\text{Estimated}}{\text{Ground Truth}}\right)\right|$, which corresponds to the horizontal distance between the estimated and ground truth CDF curve under logarithmic x-axis. We are interested in the relative error of typical percentile queries (at 50%, 95%, and 99%), as well as the maximum error for any percentile between $[5\%, 95\%]$, i.e., the maximum horizontal gap between the CDF curves between $y \in [5\%, 95\%]$.

Dataset. We use both real and synthetic network traffic traces in our experiments.

- Real-world traffic (§ 2.4.1, 2.4.3): We use a bi-directional anonymized traffic trace that contains 10 million packets across 11.4 seconds, collected from a 10Gbps border link of a local ISP network. We extract round-trip delay samples by treating outgoing TCP data packets as requests, and looking for their matching incoming TCP acknowledgment packets as responses. The trace includes 61% requests and 39% responses. Approximately 13% of all requests have a matching response, as the TCP delayed-ACK mechanism only sends one response for every two (or more) requests, and malicious traffic such as port-scanning attacks generates many orphan requests. The average round-trip delay across all samples is 57.8 milliseconds.

- Synthetic trace (§ 2.4.2): We also generated synthetic traces to explore our data structure's performance characteristics under other traffic distributions. We first generate request packets arriving at a constant rate of 1 million packets per second, and randomly select a 40% subset to generate responses. Subse-

quently, given a maximum delay of Tms, we randomly sample a delay from a log-uniform distribution between $(0, T)$ms for each response. Finally, we combine the requests and responses and sort them by their timestamps. The trace contains 0.5 million delay samples, with approximately 1.75 million packets in total. Although the synthetic trace is not fully realistic, it allows us to test our data structure by changing the delay distribution.

Unless otherwise noted, we repeat each experiment ten times with different hash seeds and combine estimated CDFs, to reduce variance from individual runs and highlight the bias. We note that our algorithm's output exhibits a similar variance comparable to [15]; the output distribution is almost the same across different runs, unless memory is extremely limited.

2.4.1 Comparison with simple algorithm.

We first show that our fridges achieve higher accuracy than the simple algorithm described in § 2.1.1.

In Figure 2.2, we visualize the advantage of the fridge algorithm by plotting the estimated delay CDF curves alongside the ground truth (shaded). We run the single-fridge algorithm using entry probability $p = 2^{-10.4}$ and memory size $M = 2^{10}$, and the simple algorithm in [15] using the same memory size on the real-world traffic trace. The simple algorithm uses an expiry threshold $T = 2^9$ms, close to the 99%-percentile delay in the ground truth, as suggested by the authors of [15].

As we can see from Figure 2.2, our unbiased fridge algorithm closely reproduces the ground truth CDF curve, especially near the tail of the distribution. The simple algorithm produces a CDF curve biased against high delays, underestimating percentile delay queries.

The fridge algorithm estimates 50th, 95th, and 99th percentile delay much more accurately than the simple algorithm. We also plotted the maximum horizontal gap

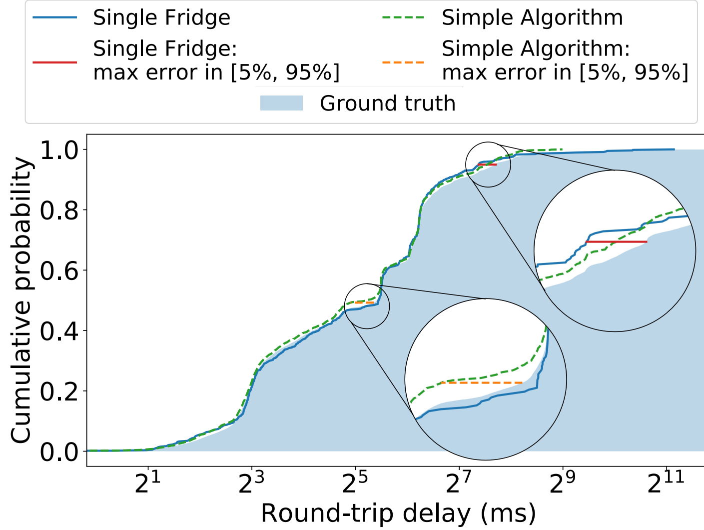


Figure 2.2: (M=2^{10}, p=$2^{-10.4}$)-fridge produces a visibly more accurate delay CDF, compared with the simple algorithm using the same memory size M=2^{10} and expiration threshold T=2^9ms. The fridge and the simple algorithm achieves maximum relative error of 25% and 36% respectively for percentile delay queries.

between estimated and ground truth CDF curves in $[5\%, 95\%]$, which corresponds to the maximum relative error when answering percentile delay queries for any percentile between 5% and 95%. The fridge algorithm has a maximum relative error of 25%, while the simple algorithm has a maximum relative error of 36%.

2.4.2 Choosing the best entry probability.

Requests in a (M, p)-fridge have an average lifetime of M/p insertions. Although the fridge algorithm's output is guaranteed to always be unbiased, we can improve its accuracy by choosing p carefully to reduce the estimator's variance. In general, we want more samples to reduce the variance of the fridge's estimation. When memory is limited, a higher p shrinks the average lifetime, so fewer large-delay samples can survive; however, a very small p means not many requests enter the fridge in the first place, thus it cannot produce many samples either.

As high-delay samples are the hardest to measure, intuitively, the best p that maximizes accuracy should make the fridge's average lifetime ($\frac{M}{p}$ insertions) roughly

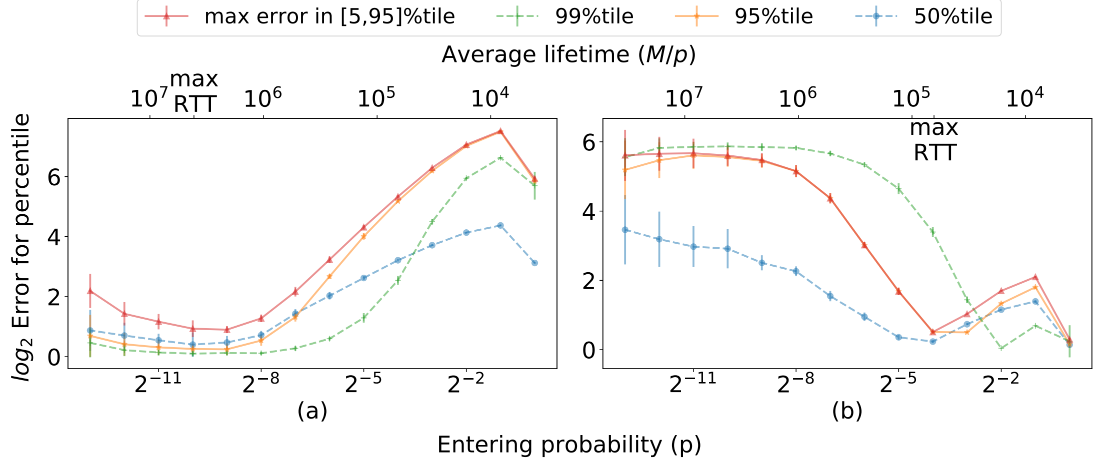


Figure 2.3: For a fridge with $M = 2^{12}$ memory size processing synthetic trace with ground truth delay between $(0, 2^{12})$ms (top), error is minimized around entering probability $p = 2^{-9}$; for ground truth delay in $(0, 2^6)$ms (bottom), the best p increases to around 2^{-4}, which leads to a shorter average lifetime in the fridge.

equal to the number of insertions between request-response pairs that experience the maximum delays we are interested in measuring. In Figure 2.3, we show the error of the fridge algorithm under different entering probabilities, under a small memory size $M = 2^{12}$. We use two different synthetic traces, one with delay range $(0, 2^{12})$ms and another with delay range $(0, 2^6)$ms.

In Figure 2.3(a), the largest delay 2^{12}ms corresponds to 4×10^6 insertions between request and response. The best choices of p indeed appear near average lifetime $M/p = 4 \times 10^6$. For Figure 2.3(b), the largest delay 2^6ms corresponds to 6.4×10^3 insertions, and we observe an increase in the best choice of p (thus decreased average lifetime). Also, the fridge's accuracy is not very sensitive to the exact choice of p, as we observe similar accuracy when choosing any p within 0.5x-2x of the optimal.

We conclude that network administrators deploying the algorithm should provision p according to the expected maximum delay to be measured in the network, by aligning the fridge's average lifetime to the number of insertions under this delay. It is likely not necessary to tune p continuously to adapt to the slight temporal changes in traffic patterns, as the fridge is robust against a 0.5x-2x change in optimal p, and

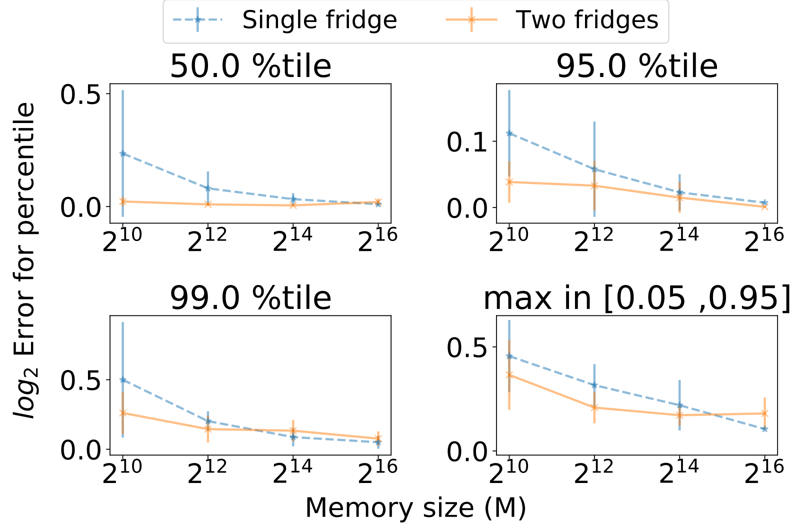


Figure 2.4: Compared with the single fridge single stage variant, using two fridges provides more benefits as memory size M decreases.

we only observe a 1.2x-1.3x diurnal change in per-minute average traffic rate in our network. However, p should be re-calibrated whenever the traffic rate or the average delay changes more than 2x.

2.4.3 Beyond a single fridge.

We now show that using multiple fridges improves accuracy by experimenting with a two-fridge algorithm.

The two fridges each use half of the total memory size ($M_1 = M_2 = \frac{M}{2}$), and we find the best entry probabilities (p_1, p_2) using grid search; the outputs of two fridges are then combined using Inversed Variance Weighting (as discussed in § 2.3.2) to produce the final estimated CDF. In Figure 2.4, we show the two-fridge algorithm is more accurate than a single fridge when processing the real-world traffic trace, especially in the more challenging regime with smaller memory size.

Finally, we compare our single- and two-fridge algorithms with both the single-stage and the four-stage simple algorithms. We similarly tuned the simple algorithm to use the best expiration threshold for each memory size. Figure 2.5 shows that

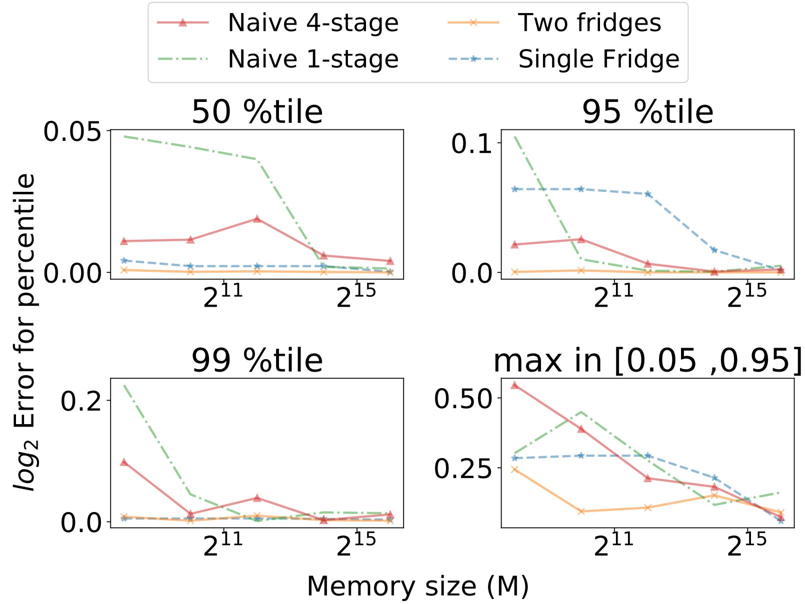


Figure 2.5: The single- and two-fridge algorithms exhibit much lower estimation error for 50, 95, 99th-percentile delays, compared with the 1- and 4-stage simple algorithms, under various total memory sizes.

the two-fridge algorithm performs consistently better than the simple algorithms, achieving the same accuracy using 2x-4x smaller memory.

It is not yet clear how to systematically find the best entry probability for more than two fridges without using exhaustive search, or how much further we can reduce error by using three or more fridges. We leave these as future work.

2.5 Hardware Implementation

We build a prototype of the fridge data structure that runs on the Intel Tofino high-speed programmable switch, that can measure traffic delay distribution at line-rate of 100 Gbps per port across 16 ports (1.6 Tbps aggregated throughput). The prototype program is written in the P4 [50] language and has approximately 800 lines of code. Its source code is available on GitHub[1] .

[1] https://github.com/Princeton-Cabernet/p4-projects/tree/master/Fridge-tofino

Our prototype measures TCP handshake delay distribution by parsing handshake packets, hashing the IP address pair, port number pair, and TCP sequence number together into a request/response ID; it is straightforward to adapt the program to measure other delays by similarly defining and calculating request/response IDs using other information in the packets.

In this section, we briefly discuss some technical details about implementing the algorithm on hardware programmable switches, and present evaluations of our prototype.

2.5.1 Implementing the fridge table

We implement the fridge using three hash-indexed arrays: $IDS[\cdot]$, $TS[\cdot]$, and $CTR[\cdot]$, storing the request packet's ID, its timestamp, and the insertion counter, respectively.

When a request arrives, we first combine the relevant packet header fields (and compress them through a digesting hash function) to generate a 32-bit request ID, as well as generating a random array index $idx = hash(ID) \in [M]$, which specifies a location in the fridge. Subsequently, we invoke the pseudorandom number generator to generate a 32-bit random number $r \in [0, 2^{32})$, and compare it against a threshold: the request is only inserted if $r < p \cdot 2^{32}$. This way, we implement the entry probability p, as the request is ignored with probability $1 - p$. We also maintain an additional register as the fridge's insertion counter, which is incremented by one for each inserted request. Subsequently, we write this request's ID, the current timestamp, and the insertion counter into $IDS[idx]$, $TS[idx]$, and $CTR[idx]$, respectively.

When a response arrives, we calculate the same ID and index $idx = hash(ID)$, and check if a matching ID is currently stored in $IDS[idx]$. If so, we generate a delay sample by calculating the delay (the difference between current timestamp and $TS[idx]$) and the number of survived insertions (x, the difference between current insertion counter and $CTR[idx]$), and also erase the current values. Otherwise, if the

stored ID mismatched, the request didn't survive and we simply do not produce a sample.

We note that the process of generating IDs using a 32-bit hash digest might lead to mismatching request and response sharing the same ID. However, the probability for an ID mismatch is much lower than that of a hash collision on the shorter *idx* (8-16 bits), therefore a request is much more likely to be evicted by an unrelated response than suffering from an ID mismatch and produce an incorrect delay sample.

2.5.2 Correcting the bias

Given the delay t and survived insertion count x in a reported sample, we can calculate the single-fridge bias correction factor in the switch data plane.

As the programmable switch only supports basic arithmetic operations, we cannot exactly calculate $p^{-1}\left(1-\frac{p}{M}\right)^{-x}$; instead, we notice p and M are known constants, and exploit P4's match-action semantics to match x with a list of prefixes, effectively building a lookup table with pre-computed x ranges and the corresponding correction factor.

The prefix matching logic available on the programmable switch was originally used for routing network packets over the internet by IP address prefixes. Given that the bias correction factor $p^{-1}\left(1-\frac{p}{M}\right)^{-x}$ is a monotonic function over x, it is straightforward to implement a lookup table that matches on the bit prefixes of the binary representation of x and outputs the correction factor. To save memory, we do not implement all possible correction factors exactly, and instead only map x approximately to several integer correction factors starting from $1/p$ with 1.1x increment. The programmable switch hardware supports matching using different bit prefix lengths, which is very handy given that the correction factor has x in its exponent.

For example, with $p = 0.5$ and $M = 2^{16}$, the first prefix-matching rule matches $x \in [0, 7 \times 2^{11})$ and outputs correction factor 2, and the next rule matches $x \in [7 \times 2^{11}, 9 \times 2^{12})$ and outputs correction factor 3. We use a python script to automatically generate these rules based on p and M.

Subsequently, we tally the delay samples and build a histogram in a register array, by adding up the correction factors of delay samples that fall into certain delay ranges. In our prototype, we maintain a histogram with 32 bins using $log(t)$ as the bin index, however it is straightforward to discretize the distribution differently or use more bins.

This implementation allows the data-plane program to track the distribution of delay in real time, enabling diverse applications such as real-time SLA monitoring and dynamic rerouting. We can either maintain an overall delay distribution, or split the traffic into different subsets and maintain a separate delay distribution for each subset.

Still, we note that various approximations incur additional error in the measurement. One may collect all the produced samples and perform the bias correction outside of the data plane, using a program running on a server (with no arithmetic constraints), to exactly calculate the correction factors and the delay distribution. This also allows running the more complex correction operations required by the multi-fridge algorithm.

2.5.3 Prototype evaluation

We evaluate our prototype fridge implementation by running it on an Intel Tofino Wedge-32X programmable switch, processing the same real-world traffic trace used in § 3.4. We use the MoonGen [22] traffic generator to replay the trace to the switch at real-world speed, by reading the pcap file from a ramdisk. The traffic generator

runs on a server with two 10-core Intel Xeon 4114 CPUs and a Mellanox ConnectX-5 100Gbps NIC, using Ubuntu 20.04 and DPDK 19.05.

We check that the fridge is producing samples correctly, by running it under various memory sizes M and collecting all the samples reported using a server running packet capturing. Analyzing the raw samples produces a delay distribution CDF closely matching the ground truth, unless M is set to be very small. The results closely match what we observe under simulation. We also analyze the effect of approximating the correction factor using a lookup table of x in the data plane, and find it only negligibly affects the resulting CDF: we observe the maximum relative error increases by between 0.2% and 0.9%, which is at least one order of magnitude smaller than the relative error between the fridge's estimated distribution and the ground truth.

A fridge with $M = 2^{16}$ entries costs about 6.4% of the total register memory available on the programmable switch. We only need $M = 2^{12}$ entries to process the real-world traffic trace used in § 3.4 and produce an accurate delay CDF, and under this configuration we only consume 1.7% of the total register memory. Besides the register memory allocated for the fridge, the prototype program also uses 23.6% of hash units (for array indexing), 7.3% of Ternary Content Addressable Memory (TCAM, for prefix lookup tables) and less than 5%-10% of any other hardware resource.

Given that we only use moderate hardware resources, we believe our prototype program's performance is sufficient to process traffic at the switch's maximum line rate; unfortunately, our packet generator server can only replay trace at speeds up to 8Gbps (due to single-core CPU bottleneck) and generate synthetic traffic at approximately 80Gbps. We have validated the prototype behaved correctly under both cases. At 80Gbps, the prototype data-plane program is processing more than 160 million requests and responses per second, which is 80 times faster than a simulator

written in C++ (processing 2 million requests/responses per second on a single CPU core).

2.6 Related Work

Measuring delay. PingMesh [24] and NetBouncer [48] measure round-trip delay by running active measurement on end hosts, and calculating the time difference between outgoing probes and incoming replies. Active measurement can generate comprehensive reports periodically, however the probe might not experience the same delay as actual application traffic [3]. Meanwhile, Ruru [21] and [2] measure TCP handshake delay by passively observing the three-way handshake packets. This is helpful for producing a flow-level distribution of delays. [51] measures delay for all TCP packets, by adding a timestamp as a TCP option header. This method can produce accurate samples, as long as intermediate firewalls do not drop the option header and the client correctly echoes back the timestamp. Instead, [30] passively measured TCP packets by observing sequence numbers, and produced delay estimates for many but not all packets. However, these methods all require exporting a large number of packets from the data plane for off-path analysis, which incurs significant networking and computational overhead when measuring high-speed networks.

Delay measurement in the data plane. [23] and [15] both measure delay directly in the switch data plane. Dapper tracks TCP flows individually, and produces one delay sample per round-trip for each TCP flow; this requires pre-allocating memory for every TCP flow being tracked. Meanwhile, [15] works directly with packets from all flows. This allows better memory utilization (almost the entire memory is used at all times) and does not require per-flow state, however it leads to the bias issue against long delays, which we addressed in this work.

Quantile sketch. KLL [32] and DDSketch [36] are quantile sketches that approximately measure samples in a distribution and produce an estimate of certain quantiles using only small memory. QPipe [27] implements a quantile sketch that runs fully within the programmable switch data plane. Our work does not measure quantiles directly as we only re-weight the produced samples to ensure the resulting distribution is unbiased, and we rely on subsequent post-processing to aggregate the samples and produce statistics such as quantiles. It is possible to feed the samples and their weights output by a fridge into a quantile sketch, such that we can approximately answer queries about percentile delay without the need to save the entire distribution.

2.7 Conclusion

In this paper, we show how to compute unbiased estimates of delay in the data plane, using the *fridge* data structure that tracks the number of evictions while the request remains in the fridge. By correcting for the probability of eviction due to hash collisions, we can produce accurate delay distributions that closely match the ground truth. Evaluation shows that our algorithm is indeed much more accurate at estimating delay percentiles, compared with prior works using the same amount memory. The two-fridge algorithm achieved the same accuracy while saving 2x-4x memory. We also build and validate a prototype implementation of the fridge running on high-speed programmable switches, that measures the unbiased delay distribution accurately and efficiently within the data plane.

Chapter 3

Detecting TCP Packet Reordering in the Data Plane

In this chapter, we present data structures that detect and report packet-reordering statistics to the control plane.

- We first sample as many flows as possible, regardless of their sizes, but only for a short period at a time. Capitalizing on the correlation, we can capture the extent of reordering in prefixes by observing only snippets of their flows. This flow-sampling approach performs especially well when given a small amount of memory.

- When more memory is available, we can further improve the accuracy by monitoring heavy flows over longer periods of time in a separate data structure, and only sampling the rest of the flows.

The interplay between measuring at the flow level and acting at the prefix level lies at the heart of this problem. To decide which set of flows to monitor, we need to incorporate prefix identity in managing the data structures, which gives rise to the idea of allocating memory at the prefix level.

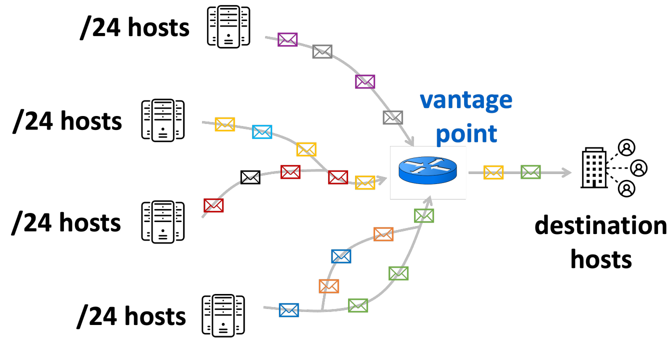


Figure 3.1: Different source prefixes send packets over different paths. Packets on a path are colored differently to show that traffic from a single prefix has a mix of packets from different flows. While flows from a single prefix may split over parallel subpaths, they do share many portions of their network resources.

3.1 Problem Formulation: Identify Heavy Out-of-Order IP Prefixes

Consider a switch close to the receiving hosts, where we observe a stream of incoming packets (Figure 3.1). Our goal is to identify the senders whose paths to the receivers are experiencing performance problems, through counting out-of-order packets. In § 3.1.1, we first introduce notations and definitions at the flow level, and show that identifying flows with heavy reordering is hard, even with randomness and approximation. Later, in § 3.1.2, we extend the definitions to the prefix level, then discuss possible directions to identify heavy out-of-order prefixes.

3.1.1 Flow-level reordering statistics

3.1.1.1 Definitions at the flow level

Consider a stream S of TCP packets from different remote senders to the local receivers. In practice, TCP packets may contain payloads, and sequence numbers advance by the length of payload in bytes. But, to keep the discussions simple, we assume sequence numbers advance by 1 at a time, and we ignore sequence number

rollovers. We note that these assumptions can be easily adjusted to reflect the more realistic scenarios. Then, a packet can be abstracted as a 3-tuple (f, s, t), with $f \in \mathcal{F}$ being its flow ID, $s \in [I]$ the sequence number and t the timestamp. In this case, a flow ID is a 4-tuple of source and destination IP addresses, and the source and destination TCP port numbers.

Let $S_f = \{(f, s_i, t_i)\}_{i=1}^{N_f} \subseteq S$ be the set of packets corresponding to some flow f, sorted by time t_i in ascending order. We say the packets of flow f are perfectly *in-order* if $s_{i+1} = s_i + 1$ for all i in $[N_f - 1]$. By common alternative definitions [40], the ith packet in flow f is *out-of-order* if it has:

Def. 1 a lower sequence number than its predecessor in f, $s_i < s_{i-1}$.

Def. 2 a sequence number larger than that expected from its predecessor in f, $s_i > s_{i-1} + 1$.

Def. 3 a smaller sequence number than the maximum sequence number seen in f so far, $s_i < \max_{j \in [i-1]} s_j$.

When $s_i < s_{i-1}$ in flow f, we sometimes say an *out-of-order event* occurs at packet i with respect to Def. 1. Out-of-order events with respect to other definitions are similarly defined. Under each definition, denote the number of out-of-order packets in flow f as O_f, a flow f is said to be *out-of-order heavy* if $O_f > \varepsilon N_f$ for some small $\varepsilon > 0$.

In practice, none of these three definitions is a clear winner. Rather, different applications may call for different metrics. From an algorithmic point of view, Def. 1 and Def. 2 are essentially identical, in that detecting the out-of-order events only requires comparing adjacent pairs of packets. An out-of-order event with respect to Def. 3, however, is far more difficult to uncover, as looking at pairs of packets is no longer enough—the algorithm always has to record the maximum sequence number (over a potentially large number of packets) in order to report such events. In this

paper, we focus on Def. 1 and show that easy modifications to the algorithms can be effective for Def. 2.

3.1.1.2 A strawman solution for identifying out-of-order heavy flows

A naive algorithm that identifies out-or-order heavy flows would memorize, for every flow, the flow ID f, the sequence number s of the latest arriving packet from f when using Def. 1, and the number of out-of-order packets o. When a new packet of f arrives, we go to its flow record, and compare its sequence number s' with s. If $s' < s$, the new packet is out-of-order and we increment o by 1.

For Def. 2, we simply save the expected sequence number $s+1$ of the next packet when maintaining the flow record, and compare it to that of the new packet, according to Def. 2. We see that different definitions only slightly altered the sequence numbers saved in memory, and we always decide whether an out-of-order event has happened based on the comparison.

3.1.1.3 Memory lower bound for identifying out-of-order heavy flows

To show that identifying out-of-order heavy flows is fundamentally expensive, we want to construct a worst-case packet stream, for which detecting heavy reordering requires a lot of memory. For simplicity, we consider the case where heavy reordering occurs in only one of the $|\mathcal{F}|$ flows, and let this flow be f. If f is also heavy in size, it suffices to use a heavy-hitter data structure to identify f. Problems arise when f is not that heavy on any timescale, and yet is not small enough to be completely irrelevant. A low-rate, long-lived flow fits such a profile. Unless given a lot of memory, a heavy-hitter data structure is incapable of identifying f. Moreover, since the packet inter-arrival times for a low-rate flow are large, to see more than one packet from f, the record of f would need to remain in memory for a longer duration, relative to other short-lived or high-rate flows.

Next we formalize this intuition, and show that given some flow f, it is infeasible for a streaming algorithm to always distinguish whether O_f is large or not, with memory sublinear in the total number of flows $|\mathcal{F}|$, even with randomness and approximation.

Claim 3.1. *Divide a stream with at most $|\mathcal{F}|$ flows into k time-blocks $B_1, B_2, \ldots, B_k$. It is guaranteed that one of the following two cases holds:*

1. *For any pair of blocks B_i and B_j with $i \neq j$, there does not exist a flow that appears in both B_i and B_j.*
2. *There exists a unique flow f that appears in $\Theta(k)$ blocks.*

Then distinguishing between the the two cases is hard for low-memory algorithms. Specifically, a streaming algorithm needs $\Omega(\min(|\mathcal{F}|, \frac{|\mathcal{F}|}{k} \log \frac{1}{\delta}))$ bits of space to identify f with probability at least $1 - \delta$, if f exists.

Claim 3.1 follows from reducing the communication problem MostlyDisjoint stated in [31], by treating elements of the sets as flow IDs in a packet stream.

Claim 3.1 implies the hardness of identifying out-of-order heavy flows, as the unique flow f may have many packets, but not be heavy enough for a heavy-hitter algorithm to detect it efficiently. Deciding whether such a flow exists is already difficult, identifying it among other flows is at least as difficult. Consequently, checking whether it has many out-of-order packets is difficult as well.

The same reduction also implies that detecting duplicated packets requires $\Omega(|\mathcal{F}|)$ space. In fact, Claim 3.1 corroborates the common perception that measuring performance metrics such as round-trip delays, reordering, and retransmission in the data plane is generally challenging, as it is hard to match tuples of packets that span a long period of time, with limited memory.

3.1.2 Prefix-level reordering statistics

3.1.2.1 Problem statement

Identifying out-of-order heavy flows is hard; fortunately, we do not always need to report individual flows. Since reordering is typically a property of a network path, and routing decisions are made at the prefix level, it is natural to focus on heavily reordered prefixes. Throughout this paper, we consider 24-bit source IP prefixes, as they achieve a reasonable level of granularity. The same methods apply if prefixes of a different length are more suitable in other applications.

By common definitions of the flow ID, the prefix g of a packet (f, s, t) is encoded in f. To simplify notations, we think of a prefix g as the set of flows with that prefix, and when context is clear, S also refers to the set of all prefixes in the stream. Let $O_g = \sum_{f \in g} O_f$ be the number of out-of-order packets in prefix g. A prefix g is *out-of-order heavy* if $O_g > \varepsilon N_g$ for some small $\varepsilon > 0$, where N_g is the number of packets in prefix g.

For localizing attacks and performance problems, it is not always sensible to catch prefixes with the highest fraction of out-of-order packets. When a prefix is small, even a single out-of-order packet would lead to a large fraction, but it might just be caused by a transient loss. In addition, with the control plane being more computationally powerful yet less efficient in packet processing, there is an apparent trade-off between processing speed and the amount of communication from the data plane to the control plane. As a result, we also want to limit the communication overhead incurred.

Therefore, for some $\varepsilon, \alpha, \beta$, our goals can be described as:

1. Report prefixes g with $N_g \geq \beta$ and $O_g > \epsilon N_g$.
2. Avoid reports of prefixes with at most α packets.
3. Keep the communication overhead from the data plane to the control plane small.

3.1.2.2 Bypassing memory lower bound

As a consequence of Claim 3.1, it is evidently infeasible to study all flows from a prefix and aggregate all of that information to determine whether to report the prefix. So why would reporting at the prefix level circumvent the lower bound? In practice, TCP packet reordering can be categorized into:

1. *TCP-induced reordering:* This is when TCP reacts to the problems in the network. For instance, congestion might cause TCP to lose packets and trigger retransmissions, which leads to apparent packet reordering. In this case, TCP reordering is the *symptom* of a congestion problem.

2. *Network-induced reordering:* Flaky network equipment might actually reorder packets. The TCP end-point then receives a misleading signal from the way the packets arrive, wrongly assuming that some packets were lost, and over-reacting to perceived congestion. In this case, packet reordering is a *cause* of a performance problem.

Distinguish between these two types of reordering is difficult. Fortunately, we want to detect both kinds of reordering, since both are indicative of TCP experiencing trouble. Moreover, both indicate some problem along the end-to-end path. Therefore, we expect flows traversing the same path at the same time to be positively correlated in their out-of-orderness, under both types of reordering. This effectively means that we only need to study a subset of flows from a prefix to estimate the extent of reordering this prefix suffers. How large a subset needs to be depends on the strength of the correlation. Interestingly, weak correlation is already sufficient to offer significant benefits.

We state the correlation assumption that all of our algorithms are based on as follows, and postpone its verification to §3.2.2: Let f be a flow chosen uniformly at

random from all flow in prefix g. If $N_g > \alpha$, and g has at least two flows, $\frac{O_g - O_f}{N_g - N_f}$ and $\frac{O_f}{N_f}$ are positively correlated.

3.2 Traffic Characterization

This section presents several traffic traits that drive our algorithm design. For all of our measurement and evaluation, we make use of the following real-world packet traces:

- **Campus:** Two anonymized packet traces, collected ethically from a border router on a university campus network on June 5, 2019, and May 9, 2022, respectively.
- **Backbone:** CAIDA Anonymized Internet Traces from 2018 [11] and 2019 [12].

Note that only packets with payloads are relevant for our application, as TCP sequence numbers must advance for our algorithms to detect reordering events. We therefore preprocess the trace to only contain flows from servers to clients using source and destination port numbers, with the rationale that these senders are more likely to generate continuous streams of traffic.

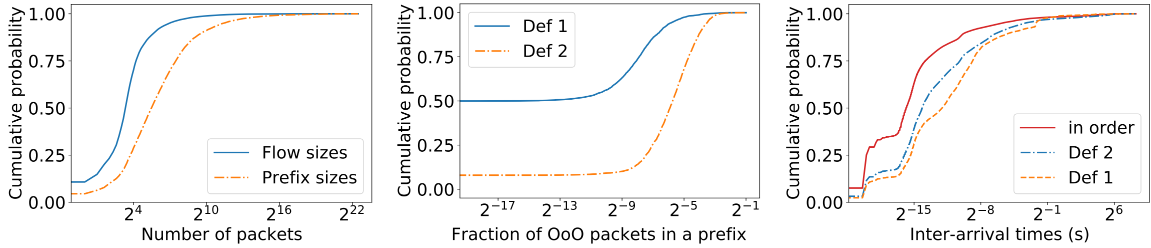
3.2.1 Heavy-tailed size and out-of-orderness

Consistent with numerous prior measurement studies, in our 5-minute campus trace (Figure 3.2a), most flows are small, and only a few flows are large. However, a small fraction of flows and prefixes tend to account for a large fraction of the traffic. For example, in this trace, more than 90% of the packets belong to the 5% largest flows or prefixes. Out-of-orderness in prefixes is similarly heavy-tailed; only a small fraction of prefixes have a significant fraction of packets out-of-order (Figure 3.2b), e.g., only less than 12% of prefixes with at least 2^7 packets have more than 1% of packets out

of order by Def. 1. Out-of-order events defined by Def. 2 are more prevalent. But, even so, packet reordering remains a low-probability event, with less than 10% of the prefixes of size at least 2^7 experiencing more than 7% out-of-order packets by Def. 2.

If most packet reordering occurred in heavy flows and prefixes, detecting heavy reordering would be easy, by solely focusing on large flows and prefixes using heavy-hitter data structures. However, what happens in reality is quite the opposite. To see that, we use an unconventional split violin plot (Figure 3.3) to show three sets of information: the prefix size (color of the violin), the flow size distribution in a prefix (the left half of the violin), and the fraction of reordered packets for that prefix that lie within flows of certain size (the right half of the violin). Each split violin corresponds to a heavily reordered prefix with at least $\beta = 2^7$ packets, using Def. 2 with $\varepsilon = 0.02$. By comparing the left halves of all violins, we see a wide variation of flow sizes in prefixes with heavy reordering, and the sizes of such prefixes can be orders-of-magnitude different. We see that many prefixes do not have any large flows. Moreover, the largest flows in each heavily reordered prefix do not necessarily contain most of the out-of-order packets in that prefix. The 131-largest prefix gives one such example. Though the 50 largest flows in this prefix have size 2^7 or larger, almost 95% of the total out-of-order packets in this prefix comes from flows with size smaller than 2^7. Such a prefix would be very difficult for a heavy-hitter data structure to catch without investing significant memory. Thus, by zooming in on large flows and prefixes, we would inevitably miss out on many prefixes of interest without any large flow.

Fortunately, to report a prefix with a significant amount of reordering, we need not measure every flow in that prefix, as flows in the same prefix have some correlation in their out-of-orderness. As it turns out, the fraction of out-of-order packets in a prefix is positively correlated with that of a flow within the prefix, which we verify next.



(a) A small fraction of flows and prefixes account for a large fraction of the traffic.

(b) Out-of-order heavy prefixes are rare. Here prefixes have at least $\beta = 2^7$ packets.

(c) Out-of-order events defined by Def. 1 exhibit the highest inter-arrival times.

Figure 3.2: Heavy-tailed distributions in a 5-minute campus trace.

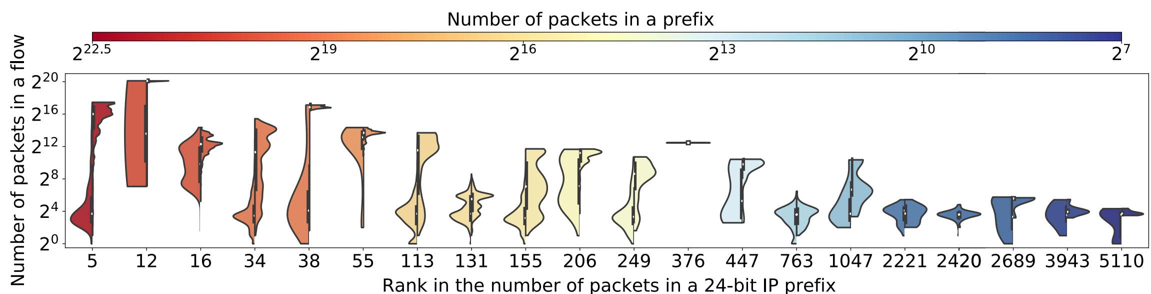


Figure 3.3: A split violin plot showing prefix sizes, distributions of flow sizes in each prefix, and what fraction of reordering in a prefix comes for which flow size. A split violin of rank r refers to the r-th largest prefix in the trace.

3.2.2 Correlation among flows in a prefix

Let f be a flow drawn uniformly at random from a set of flows. Let X be the random variable representing the fraction of out-of-order packets in flow f, $X = \frac{O_f}{N_f}$. Denote g as the prefix of flow f, let Y be the random variable denoting the fraction of out-of-order packets among all flows in prefix g excluding f, that is, $Y = \frac{O_g - O_f}{N_g - N_f}$, where N_g is the number of packets in prefix g. To ensure that $N_g > N_f$, the prefixes we sample from must have at least two flows. We use the *Pearson correlation coefficient (PCC)* to show that X and Y are positively correlated, which implies that the out-of-orderness of a flow f is statistically representative of other flows in the prefix of f. Essentially a normalized version of $\text{Cov}(X, Y)$, PCC always lies in the interval $[-1, 1]$, and a positive PCC indicates a positive linear correlation. Lacking a better

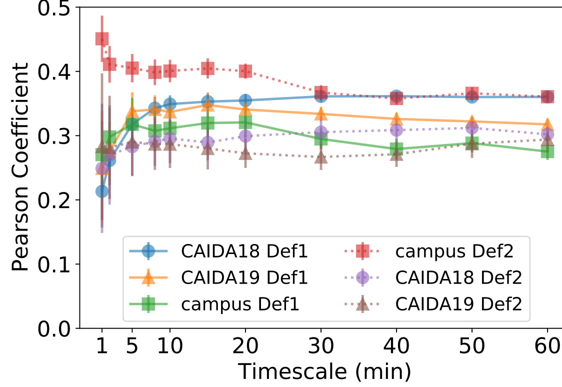


Figure 3.4: Pearson coefficient on varying timescales shows that a positive correlation exists between the reordering of a flow and that of its prefix.

reason to believe the correlation between X and Y is of higher order, we shall see that PCC suffices for our analysis.

Given a traffic trace, let S be the set of flows whose prefixes have at least two flows. We compute the PCC as follows:

1. Draw n flows from S, independently and uniformly at random.

2. For each of the n flows f_i, let $x_i = \frac{O_{f_i}}{N_{f_i}}$, $y_i = \frac{\sum_{f' \in g, f' \neq f_i} O_{f'}}{\sum_{f' \in g, f' \neq f_i} N_{f'}}$,

3. The PCC $r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$, where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$.

We perform $m = 100$ tests on each traffic trace using both definitions of reordering, on timescales ranging from 1 minute to 60 minutes (Figure 3.4). Each point shows the average of $m = 100$ tests, where we draw $n = 0.5\% \cdot |S|$ flows in each test. The result indicates that, a positive correlation, albeit weak, exists between X and Y for all tested traces on all timescales, and the correlation tends to stabilize after a small time period such as five minutes. Note that the choice of n is not essential for the purpose of demonstrating correlation. However, the fact that the correlation exists in a random subset of flows and prefixes lays the foundation for the discussion of algorithm designs (§ 3.3).

To understand what a weak correlation means for flow sampling, consider a hypothetical example where the PCC is 1. As the out-of-orderness of a flow is statistically identical to that of its prefix, observing any flow suffices for understanding the reordering of its prefix. Then, we can simply keep one entry for each prefix, and never switch between different flows in a prefix. On the other hand, if the PCC is 0, as far as reordering is concerned, observing a single flow provides little information of its prefix. In that case, we would have no choice but to observe almost all flows. Though we are not yet able to quantify the relationship between the PCC and how many flows to sample, it is evident that having a weak correlation means that our algorithm lies somewhere between the two cases. As we shall see in § 3.3, the proposed algorithm capitalizes on this weak correlation to attain reasonable accuracy using small memory.

3.2.3 Packet inter-arrival times within a flow

We also study the inter-arrival time of packets within a flow to understand how efficient the flow sampling algorithm can be. Due to TCP windowing dynamics, where the sender transmits a window of data and then waits for acknowledgments, in-order packets tend to have small inter-arrival times. Depending on the definition, reordering can be a result of gaps in transmission of non-consecutive packets (Def. 2), or worse yet the retransmissions of lost packets (Def. 1), which often lead to larger inter-arrival times.

Indeed, Figure 3.2c shows that the inter-arrival times of out-of-order packets using Def. 2 tend to be smaller than that of the out-of-order packets using Def. 1, with the inter-arrival times of in-order packets being the smallest. This implies that, to detect the reordering events in Def. 1, the algorithm has to store records for a longer waiting period, which potentially exhausts more memory resources.

3.3 Data-Plane Data Structures for Out-of-Order Monitoring

At a high level, a data-plane algorithm generates reports of flows with potentially heavy packet reordering on the fly, and a simple control-plane program parses through the reports to extract their prefixes. Each report includes the prefix, the number of packets monitored, and the number of out-of-order packets of a suspicious flow. At the end of the time interval, we can also scan the data-plane data structure to generate reports for highly-reordered flows remaining in memory. On seeing reports, a control-plane program simply aggregates counts from reports of the same prefix, and outputs a prefix when its count exceeds a threshold.

In the data plane, we keep state at the flow level, and consider prefix information in allocating memory. Assuming a positive correlation between the out-of-orderness of a prefix and that of the flows from that prefix, we need not monitor all flows in their entirety to gain enough information about a prefix. This leads to the simple yet effective flow-sampling algorithm in § 3.3.1, where we sample as many flows as possible, but only over a short period at a time. Though it is not enough to only measure reordering in heavy flows (§ 3.2.1), in § 3.3.2, we show that there are still benefits from combining a heavy-hitter data structure with the flow-sampling array.

3.3.1 Sample flows over short periods

To sieve through a large number of flows with limited memory, the turnover rate has to be high. This means that, the algorithm has to be somewhat oblivious to the various statistics of a flow, such as flow sizes and inter-arrival times, when choosing to admit or evict a flow. To set the stage for later discussions, throughout this paper, we refer to the unit of memory allocated to keep one flow record as a *bucket*. Now,

rather than one bucket per flow, the main idea is to use one bucket to quickly check over multiple flows in turn.

3.3.1.1 Flow sampling with array

Under the strict memory access constraints, we again opt for a hash-indexed array as a natural choice of data structure, where each row in the array corresponds to a bucket, and all buckets behave independently. To check many prefixes for reordering, we do not want some prefix with a huge number of flows to dominate the data structure. To this end, we assign flows from the same prefix to the same bucket, by hashing prefixes instead of flow IDs, a technique we use in all our algorithms.

Therefore, we fix a bucket $\mathfrak{b}$, and consider the substream of packets hashed to $\mathfrak{b}$. When a packet (f, s, t) arrives at $\mathfrak{b}$, there are three cases:

1. If $\mathfrak{b}$ is empty, we always admit the packet, that is, we save its flow record f, sequence number s, timestamp t in $\mathfrak{b}$, together with the number of packets n and the number of out-of-oder packets o, both initilized to 0.

2. If flow f's record is already in $\mathfrak{b}$, we update the record as in the strawman solution (§ 3.1.1.2), and update the timestamp in memory to t.

3. If $\mathfrak{b}$ is occupied by another flow's record (f', s', t', n', o'), we only admit f if f' has been monitored in memory for a sufficient period specified by parameters T and C, or the prefix of f' could be potentially heavily reordered with respect to another parameter R. That is, f overwrites f' with record $(f, s, t, n = 0, s = 0)$ only if one of the following holds:

   (a) f' is stale: $t - t' > T$.

   (b) f' has been hogging $\mathfrak{b}$ for too long: $n' > C$.

   (c) f' might belong to a prefix with heavy reordering: $o' > R$.

In Case 3c, the algorithm sends a 3-tuple report (g', n', o') to the control plane, where g' is the prefix of flow f'. On seeing reports from the data plane, a simple control-plane program keeps a tally for each reported prefix g. Let $\{(g, n_i, o_i)\}_{i=1}^{r}$ be the set of all reports corresponding to a prefix g. The control-plane program outputs g if $\sum_{i=1}^{r} n_i \geq \alpha$, for the same α in § 3.1.2.1. In the following sections, we refer to the data-plane component together with the simple control-plane program as the flow-sampling algorithm.

Lazy expiration of flow records in memory Due to memory access constraints, many data-plane algorithms *lazily* expire records in memory on collisions with other flows, as opposed to actively searching for stale records in the data structure. We again adopt the same technique in the algorithm above, though here it is more nuanced. We could imagine a variant of the algorithm where a flow is monitored for up to $C+1$ packets at a time. That is, when the $(C+1)$st packet arrives, we check whether to report this flow, and evict its record. Compared to this variant, lazy expiration helps in preventing a heavy flow being admitted into the data structure consecutively, so that the heavy flow can be evicted before a integer multiple of $(C+1)$ packets, should another flow appear in the meantime.

Robustness of flow sampling For the flow-sampling method to be effective, the data structure needs to sample as many flows as possible. Therefore, it is not desirable to keep a large flow in memory when we have already seen many of its packets, and learned enough information about its packet reordering. This means the packet count threshold C should not be too large. Neither do we want to keep a flow, regardless of its size, that has long been finished. We can eliminate such cases by setting a small inter-arrival timeout T.

Now the question is, how small can these parameters be. Real-world traffic can be bursty, meaning that sometimes there are packets from the same flow arriving

back-to-back. In this case, even if we overwrite the existing flow record on every hash collision ($T = 0$ and $C = 1$), the algorithm still generates meaningful samples. When the memory is not too small compared to the number of prefixes, and hash collisions are rare, the algorithm might even have good performance. However, setting small $T > 0$ and $C > 1$ makes the algorithm more robust against worst-case streams. Consider a stream of packets where no adjacent pairs of packets come from the same flow. On seeing such a stream, a flow-sampling algorithm that overwrites existing records on every hash collision with another flow will no doubt collect negligible samples. In contrast, small $T > 0$ and $C > 1$ allow a small period of time for a flow in memory to be monitored, and hence gives a better chance of capturing packet reordering.

3.3.1.2 Performance guarantee

In this section, we analyze the number of times a flow with a certain size is sampled. Consider a prefix g when the hash function is fixed. Let $\mathfrak{b}$ be the bucket prefix g is hashed to, and we know all the flows as well as the prefixes that are hashed to $\mathfrak{b}$. With a slight abuse of notation, we write $g \in \mathfrak{b}$ when the bucket with index $h(g)$ is $\mathfrak{b}$. We also write $f \in \mathfrak{b}$ when f's prefix is hashed to $\mathfrak{b}$. To capture the essence of the flow-sampling algorithm without excessive details, we make the following assumptions:

1. Each packet in S is sampled i.i.d. from distribution $(p_f)_{f \in \mathcal{F}}$, that is, each packet belongs to some flow $f \in \mathcal{F}$ independently with probability p_f. Consequently, each packet belongs to some prefix g independently with probability $p_g = \sum_{f \in g} p_f$.

2. Let $p_{f|\mathfrak{b}} = \frac{p_f}{\sum_{f' \in \mathfrak{b}} p_{f'}}$, $p_{g|\mathfrak{b}}$ can be similarly defined. Only a flow f with $p_{f|\mathfrak{b}}$ greater than some $p_{\min}$ will get checked, where we think of $p_{\min}$ as a fixed threshold depending on the inter-arrival time threshold T and distribution $(p_f)_{f \in \mathcal{F}}$.

3. A flow is checked exactly $C+1$ packets at a time.

Note that Assumption (2) is a way to approximate the effect of T, where we assume a low-frequency flow would soon be overwritten by some other flow on hash collision. In contrast to Assumption (3), the flow sampling algorithm does *not* immediately evict a flow record with $C+1$ packets, if there is no hash collision. In this way, though f is monitored beyond its original $C+1$ packets, once a hash collision occurs, the collided flow would seize f's bucket. By imposing Assumption (3), the heavier flows would likely benefit by getting more checks, while the smaller flows would likely suffer. Empirically, the eviction scheme of the flow-sampling algorithm (§ 3.3.1.1) achieves better performance in comparison to Assumption (3).

Lemma 3.2. *Given the total length of stream $|S|$, distributions $(p_f)_{f\in\mathcal{F}}$, with the assumptions above, for a fixed hash function h and any $\varepsilon,\delta\in(0,1)$, a prefix g in bucket $\mathfrak{b}$ is checked at least $(1-\delta)t_1 p_{g|\mathfrak{b}}$ times with probability at least $1-e^{-p_{min}t_1 CF_{\mathfrak{b}}\cdot\frac{\varepsilon^2}{24}}-e^{-\frac{\varepsilon^2|S|\sum_{g\in\mathfrak{b}}p_g}{3}}-e^{-\frac{\delta^2 t_1 p_{g|\mathfrak{b}}}{2}}$, where $t_1=\left\lfloor\frac{|S|\sum_{g\in\mathfrak{b}}p_g}{(1+\frac{\varepsilon}{2})CF_{\mathfrak{b}}}\right\rfloor$ and $p_{g|\mathfrak{b}}=\frac{\sum_{f\in g:p_{f|\mathfrak{b}}\geq p_{min}}p_f}{\sum_{f'\in\mathfrak{b}}p_{f'}}$.*

Proof. Let $S_{\mathfrak{b}}$ the substream of S that is hashed to $\mathfrak{b}$. Given $|S|$, the length $|S_{\mathfrak{b}}|$ of substream $S_{\mathfrak{b}}$ is a random variable, $\mathbb{E}|S_{\mathfrak{b}}|=|S|\sum_{g\in\mathfrak{b}}p_g$, then by Chernoff bound,

$$\mathbb{P}[|S_{\mathfrak{b}}|<(1-\varepsilon)\,\mathbb{E}|S_{\mathfrak{b}}|]<e^{-\frac{\varepsilon^2\,\mathbb{E}|S_{\mathfrak{b}}|}{3}}=e^{-\frac{\varepsilon^2|S|\sum_{g\in\mathfrak{b}}p_g}{3}}. \tag{3.1}$$

Let t be a random variable denoting the number of checks in $\mathfrak{b}$. Let random variable $X_{i,j}$ be the number of packets hashed to $\mathfrak{b}$ after seeing the jth packet till receiving the $(j+1)$st packet from the currently monitored flow, where $i\in[t]$ and $j\in[C]$. $X_{i,j}$s are independent geometric random variables, and $X_{i,j}\sim Geo(p_{f_i|b})$, where f_i is the flow under scrutiny during the ith check, by Assumption 2, $p_{f_i|b}\geq p_{\min}$.

Next we look at $X = \sum_{i=1}^{t} \sum_{j=1}^{C} X_{i,j}$, the length of the substream in $\mathfrak{b}$ after t checks,

$$\mathbb{E}\, X = \sum_{i=1}^{t} \sum_{j=1}^{C} \mathbb{E}\, X_{i,j} = \sum_{i=1}^{t} \sum_{j=1}^{C} \sum_{\substack{f \in \mathfrak{b}: \\ p_{f|b} \geq p_{\min}}} p_{f|b} \cdot \frac{C}{p_{f|b}} = tCF_{\mathfrak{b}}, \tag{3.2}$$

where $F_{\mathfrak{b}} = \left| \{ f \in b \mid p_{f|b} \geq p_{\min} \} \right|$. By the Chernoff-type tail bound for independent geometric random variables (Theorem 2.1 in [28]), for any $\varepsilon \in (0, 1)$,

$$\mathbb{P}[X > (1 + \frac{\varepsilon}{2})\, \mathbb{E}\, X] < e^{-p_{\min}\, \mathbb{E}\, X (\frac{\varepsilon}{2} - \ln(1 + \frac{\varepsilon}{2}))} \leq e^{-p_{\min} t C F_{\mathfrak{b}} \cdot \frac{\varepsilon^2}{24}}. \tag{3.3}$$

Let t_1 be the largest t such that $(1 + \frac{\varepsilon}{2})\, \mathbb{E}\, X < \mathbb{E}\, |S_{\mathfrak{b}}|$, we have $t_1 = \left\lfloor \frac{|S| \sum_{g \in \mathfrak{b}} p_g}{(1 + \frac{\varepsilon}{2}) C F_{\mathfrak{b}}} \right\rfloor$. Consider two events:

(i) The number of checks t on seeing $S_{\mathfrak{b}}$ is less than t_1.

Applying 3.3 on t_1, we have that with probability at most $e^{-p_{\min} t_1 C F_{\mathfrak{b}} \cdot \frac{\varepsilon^2}{24}}$, after seeing $(1 - \varepsilon)\, \mathbb{E}\, |S_b|$ packets, the number of checks is at most t_1. Together with 3.1, by union bound,

$$\mathbb{P}[t < t_1] < e^{-p_{\min} t_1 C F_{\mathfrak{b}} \cdot \frac{\varepsilon^2}{24}} + e^{-\frac{\varepsilon^2 |S| \sum_{g \in \mathfrak{b}} p_g}{3}}. \tag{3.4}$$

(ii) Prefix g is checked less than $(1 - \delta) t_1 p_{g|\mathfrak{b}}$ times. By Chernoff bound, this event holds with probability at most $e^{-\frac{\delta^2 t_1 p_{g|\mathfrak{b}}}{2}}$.

The Lemma follows from applying the union bound over these two events. □

Counterintuitively, the proof of Lemma 3.2 suggests hash collisions are in fact harmless in the flow-sampling algorithm, for a flow that is not too small (which corresponds to $p_{f|\mathfrak{b}}$ greater than some $p_{\min}$ in Assumption 2). To see that, suppose we add another heavy flow to bucket $\mathfrak{b}$, $\mathbb{E}\, |S_{\mathfrak{b}}|$ would increase by some factor x, which means $\mathbb{E}\, X$ would increase by the same factor. Since $F_{\mathfrak{b}}$ would only increase by 1, if

$F_{\mathfrak{b}}$ is large enough, by (3.2), t would also increase by roughly a factor of x, while $p_{f|\mathfrak{b}}$ decreases by roughly a factor of x. Then $t \cdot p_{f|\mathfrak{b}}$ is about the same with or without the added heavy flow. Therefore, colliding with heavy flows does not decrease the number of checks of a flow that is not too small, as long as the total number of flows in a bucket is large enough, which is usually the case in practice.

3.3.1.3 Decrease the number of false positives

Since the parameters of the flow-sampling algorithm are chosen so that many flows are sampled, and some might get sampled multiple times, it is possible for the algorithm to capture many out-of-order events, but not every one of them indicates that the prefix is out-of-order heavy. After all, there is only a weak correlation between the out-of-orderness of flows and that of their prefixes, not to mention that even if the correlation is stronger, we are inferring the extent of reordering on a scale much larger than the snippets of flows that we observe. In such cases, the algorithm could output many false positives.

To reduce the number of false positives, we could imagine feeding the control plane more information, so that the algorithm can make a more informed decision about whether the fraction of out-of-order packets exceeds ε, for each reported prefix. To this end, we modify the flow-sampling algorithm to always report before eviction, even if the number of out-of-order packets is below threshold R. Again denote $\{(g, n_i, o_i)\}_{i=1}^{r}$ as the set of all reports corresponding to a prefix g, the control plane outputs g if $\sum_{i=1}^{r} n_i \geq \alpha$, and $\frac{\sum_{i=1}^{r} o_i}{\sum_{i=1}^{r} n_i} > c \cdot \varepsilon$, for some tunable parameter $0 < c \leq 1$. The parameter c compensates for the fact that we only monitor a subset of the traffic, so the exact fraction of out-of-order packets we observe might not directly align with ε.

3.3.2 Separate large flows

Though hash collisions generally do not affect the flow-sampling algorithm's ability to check flows that are not too small, there is still the possibility that a small flow just so happens to arrive and finish during the short period when another flow is being monitored in that bucket. Such a small flow would never get a second chance to enter the data structure. If we could instead continuously monitor some large flows in a separate data structure, then for a small flow f that is hashed to a bucket $\mathfrak{b}$ that no longer contains large flows, $p_{f|\mathfrak{b}}$ would increase, which would increase the number of checks it gets. For some prefixes whose out-of-order packets concentrate only in one small flow, separating large flows greatly improves the chance of catching them.

Therefore, we propose a hybrid scheme, where the packets first go through a heavy-hitter (HH) data structure, and the array only admits flows that are not being monitored in the HH data structure. We again assign flows with the same prefix to the same set of buckets, and the array part of the data structure behaves exactly as depicted in § 3.3.1.1. For the HH part, we report flows whose fraction of out-of-order packets is above ε. We describe the specifics on the HH data structure in § 3.3.3.

Note that a subtly different design choice would be to have the array admit the set of flows whose prefixes are not being monitored in the HH data structure. This would have made more sense, if all the heavily reordered prefixes have most of their out-of-order packets concentrated among the heaviest flows in that prefix. But as we have seen in Figure 3.3, this is not always the case. Compared to our proposed hybrid algorithm, this variant would be less accurate. However, it certainly reduces the number of false positives and the number of reports generated by the data-plane algorithm, since in this case, a much smaller set of flows would be monitored by the array. In this work, we choose to prioritize accuracy over other aspects, so we prefer the hybrid algorithm in last paragraph to this variant.

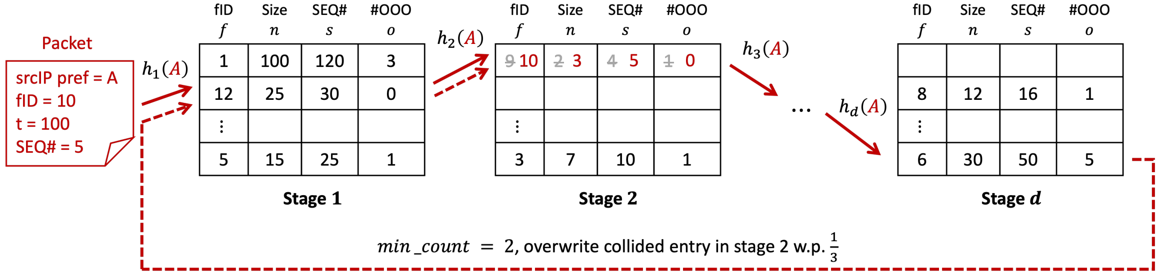


Figure 3.5: A modification of PRECISION for tracking out-of-order packets.

In any practical setting, the correct memory allocation between the HH data structure and the array in the hybrid scheme depends on the workload properties: the relationship of flows to prefixes, the heaviness of flows and prefixes, and where the reordering actually occurs. Next we understand how these algorithms behave under real-world workloads.

3.3.3 Track heavy flows over long periods

To capture out-of-orderness in heavy flows, we want a data structure that is capable of simultaneously tracking heaviness and reordering. The SpaceSaving [37] data structure fits naturally for the task, as we can maintain extra state for each flow record, while the data structure gradually identifies the flows with heavy volume by keeping estimates of their traffic counts. However, when overwriting a flow record to admit a new flow, SpaceSaving needs to go over all entries to locate the flow with the smallest traffic count, which makes it infeasible for the data plane due to the constraint on the number of memory accesses per packet.

Thus, we opt for PRECISION [4], the data-plane adaptation of SpaceSaving, which checks only a small number of d entries when overwriting a flow record. We emphasize that the specifics about how PRECISION works are not, in fact, important in this context. It is enough to bear in mind that with a suitable data-plane friendly heavy-hitter algorithm, tracking reordering is exactly the same as in the strawman

solution (§ 3.1.1.2), but applied only to heavy flows. Figure 3.5 shows the modified PRECISION for tracking out-of-order packets using d stages.

We again assign flows from the same prefix to the same set of buckets, by hashing prefixes instead of flow IDs. In a PRECISION data structure with d stages, at the end of the stream, at most d heaviest flows from each prefix g would remain in memory. Doing so effectively frees up buckets that used to be taken by a few prefixes with many heavy flows, and allows more prefixes to have their heaviest flows measured.

3.4 Evaluation

We start this section by evaluating our flow-sampling algorithm and hybrid scheme (§ 3.4.1) using a Python simulator on real-world traces introduced in § 3.2. As much as we wish that each trace is representative, we cannot simply assume that every network administrator running our algorithms in their networks would get the exact same performance. Therefore, we delve into the intricacies of multiple distributions underlying the real-world traffic workload, to explain how they affect the performance of our algorithms. In § 3.4.2, we verify that our P4 prototype of the flow-sampling algorithm for the Tofino1 switch only consumes a small amount of hardware resources, as promised. Finally, we recognize that the optimal parameters for our algorithms are often workload dependent. Thus, we do not attempt to always find the optimum; instead, we show in § 3.4.1 that *reasonably* chosen parameters already give good performance. In § 3.4.3, we see that the parameters we used previously for evaluations are indeed representative, and the algorithms are robust against small perturbations.

3.4.1 Performance comparisons

3.4.1.1 Metrics

We begin by introducing the three metrics we use throughout this section to evaluate our algorithms. Let $\hat{G}$ denote the set of prefixes output by an algorithm $\mathcal{A}$.

- **Accuracy**: Let $G_{\geq\beta} = \{g^* \in S \mid N_{g^*} \geq \beta, O_{g^*} > \varepsilon \sum_{g \in S} O_g\}$ be the ground truth set of heavily reordered prefixes with at least β packets. Define the *accuracy* A of algorithm $\mathcal{A}$ to be the fraction of ground-truth prefixes output by $\mathcal{A}$, that is,

$$A(\mathcal{A}) = \frac{\left|\hat{G} \cap G_{\geq\beta}\right|}{|G_{\geq\beta}|}.$$

- **False-positive rate**: Let $G_{>\alpha} = \{g^* \in S \mid N_{g^*} > \alpha, O_{g^*} > \varepsilon \sum_{g \in S} O_g\}$, then the *false-positive rate* of $\mathcal{A}$ is defined as

$$FP(\mathcal{A}) = \frac{\left|\hat{G} \setminus G_{\geq\alpha}\right|}{|G_{\geq\alpha}|}.$$

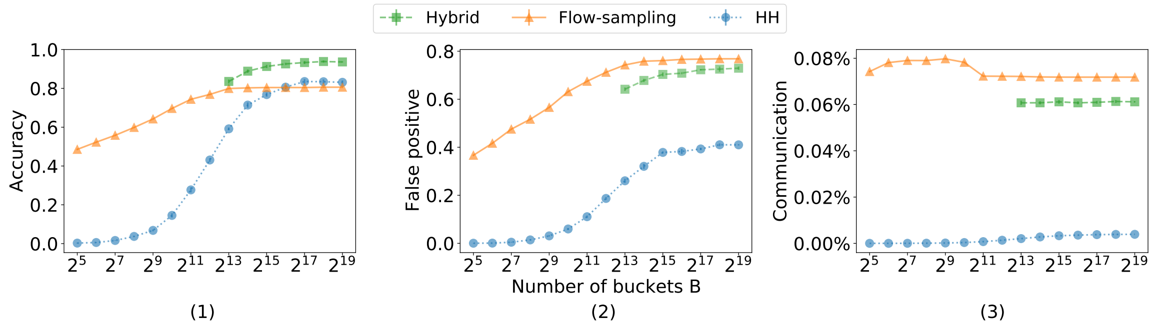
- **Communication overhead**: The *communication overhead* from the data plane to the control plane is defined as the number of reports sent by $\mathcal{A}$, divided by the length of stream S, where the number of reports also accounts for the flow records in the data structure that exceed the reporting thresholds.

Unless otherwise specified, each experiment is repeated five times with different seeds to the hash functions, and with parameters $T = 2^{-15}, C = 2^4$, $R_{\text{array}} = 1$, $R_{\text{HH}} = 0.01$, and $d_{\text{HH}} = 2$ (see § **??** for details on the parameters of the HH data structure). We are interested in identifying prefixes with at least $\beta = 2^7$ packets, with more than $\varepsilon = 0.01$ fraction of their packets reordered. Additionally, we do not wish to output prefixes with at most $\alpha = 2^4$ packets, irrespective of their out-of-orderness.

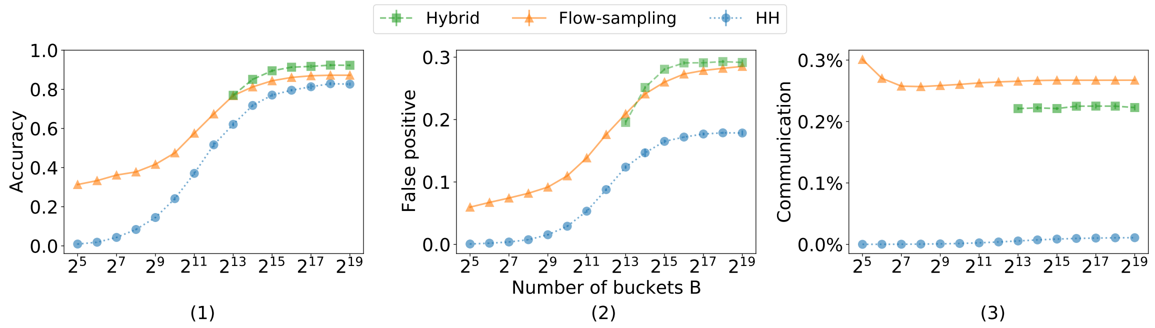
3.4.1.2 Performance evaluation

To the best of our knowledge, we are the first to consider the problem of detecting heavily reordered prefixes, and existing related works are not directly comparable. We therefore compare our proposed algorithms to a heavy-hitter (HH) data structure that tracks reordering (§ 3.3.3). Figure 3.6a shows the performance of the flow-sampling algorithm, the hybrid scheme, and the HH data structure using Def. 1, on a 5-minute campus trace consisting of $82,359,405$ server-to-client packets, which come from $545,973$ flows and $16,988$ 24-bit source IP prefixes. In fact, the specific length of the trace, and whether we choose to study reordering events of Def. 1 or Def. 2, do not affect the overall trend of these curves. To show that the performance curves in Figure 3.6a is representative, Figure 3.6b presents the performance of proposed algorithms on a 10-minute CAIDA 2019 [12] trace using Def. 2. The trace contains $61,791,947$ server-to-client packets that come from $2,717,709$ flows and $54,148$ 24-bit source IP prefixes.

Flow-sampling algorithm achieves great accuracy with small memory. If heavy reordering were concentrated in large flows, the HH data structure would perform very well with a small amount of memory. As seen in § 3.2.1, real-world traffic does not always behave in that way, rendering the HH data structure ineffective when the memory is small compared to the number of prefixes (2^{14}). This is where the performance of the flow-sampling algorithm significantly dominates that of the HH data structure. Note that this particular trace contains more than 2^{19} flows and more than 2^{14} prefixes. However, using only 2^5 buckets, the original version of the flow-sampling algorithm is already capable of reporting half of the out-of-order prefixes. To put it into perspective, reordering happens at the flow level, and assigning even one bucket per prefix to detect reordering already requires a nontrivial solution, while



(a) Performance on a 5-minute campus trace for Def. 1.



(b) Performance on a 10-minute CAIDA 2019 trace for Def. 2.

Figure 3.6: The flow-sampling algorithm achieves great accuracy in small memory ranges, and the hybrid scheme further improves the accuracy when more memory is available.

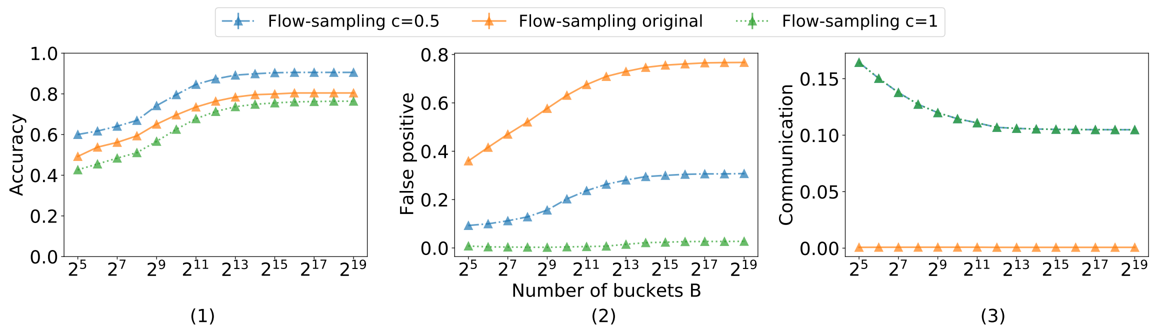


Figure 3.7: Through sending more reports to the control plane, we can decrease the false-positive rate of the flow-sampling algorithm while further improving its accuracy.

the flow-sampling algorithm achieves good accuracy using orders-of-magnitude less memory.

If we are willing to generate reports for more than 10% of the traffic, with an increased communication overhead comes a reduced false-positive rate (Figure 3.7). Moreover, with a more carefully chosen parameter c that controls how many prefixes

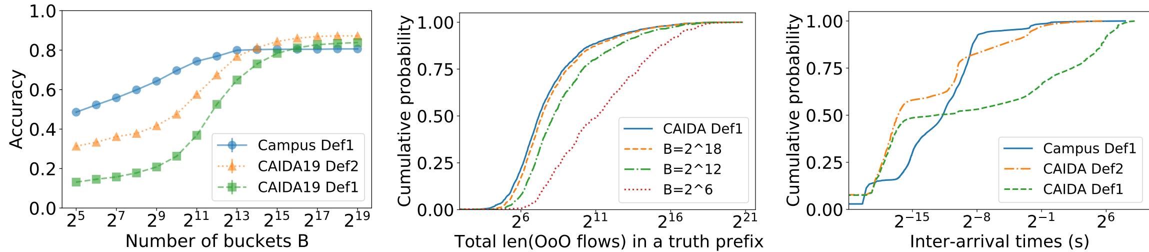
to report (§ 3.3.1.3), the extra information sent to the control plane helps in further improving the accuracy.

The hybrid scheme improves the accuracy when given more memory. To fairly compare the hybrid scheme with the flow-sampling algorithm, we need to determine the optimal memory allocation between the HH data structure and the array. Lacking a better way to optimize the memory allocation, we turn to experiments with our packet trace. Given a total of B buckets, we assign $\lfloor x \cdot B \rfloor$ buckets to the HH data structure, $B - \lfloor xB \rfloor$ buckets to the array, and conduct a grid search on $x \in I = \{0.1, \ldots, 0.9\}$ to find the value of x that maximizes the performance of the hybrid scheme. We evaluate the hybrid scheme using the optimal x we found for each B.

Admittedly, grid I may not be fine-grained enough to reveal the true optimal allocation; nonetheless, it conveys the main idea. When available memory is small, the accuracy gap between the HH data structure and the flow-sampling algorithm is huge, sparing part of the memory for filtering large flows does not improve over the flow-sampling algorithm. As memory increases, the accuracy gap between the flow-sampling algorithm and the HH data structure decreases, and the hybrid scheme starts to show accuracy gains.

3.4.1.3 Performance discrepancies of the flow-sampling algorithm under different workloads

In our numerous experiments on different traces, the accuracy of the flow-sampling algorithm always dominates that of the HH data structure, when given much less memory than the number of prefixes in the trace. However, we cannot always expect to catch 50% of the heavily reordered prefixes using just $B = 2^5$ buckets. For instance, Figure 3.8 shows the accuracy of the flow-sampling algorithm when running on a 5-



(a) The accuracy of the flow-sampling algorithm may differ under different workloads.

(b) A heavily reordered prefix is easier to capture if the total length of its flows with reordered packets is longer.

(c) The algorithm is more accurate with small memory when reordered packets arrive shortly after their predecessors.

Figure 3.8: The accuracy of the flow-sampling algorithm is workload dependent.

minute campus trace using Def. 1, and a 10-minute CAIDA 2019 [12] trace using Def. 1 as well as Def. 2. The results are evidently workload-dependent, but what exactly are the traffic characteristics that dictate such performance discrepancies? The answer to this question epitomizes the intricacies involved in understanding the multiple distributions present in real-world traffic.

To identify the subset of traffic that directly affects accuracy, we go back to how the flow-sampling algorithm reports a prefix. If we look at a heavily reordered prefix, its flows enter the data structure from time to time. But for the algorithm to report it, the array has to see some flows from this prefix that actually have out-of-order packets. The perfectly in-order flows would never contribute to the reporting of its prefix. Now, suppose the reordered packets appear uniformly at random during the time its flow is being monitored, then what matters is the total length (number of packets) of the flows that have out-of-order packets in this prefix. The higher the length, the easier it is for the flow-sampling algorithm to catch it. This is in fact an indirect implication of Lemma 3.2. It can also be seen in Figure 3.8b, which shows the CDF of the total length of the flows that have out-of-order packets among all heavily reordered prefixes reported by the flow-sampling algorithm using different memory sizes. The ground-truth prefixes reported by the smallest memory are the easiest to

catch, and the total length of out-of-order flows in such prefixes tends to be larger. As we increase the memory size, the algorithm reports more ground-truth prefixes with shorter total lengths of out-of-order flows.

However, this is not the whole story. For the traces in Figure 3.8, the CDFs of the total length of reordered flows in ground-truth prefixes turn out to be similar in shape. So what else in the traffic distribution is causing the difference in accuracy? The caveat is that reordered packets may not appear uniformly at random, and their inter-arrival times play a major role as well. For each dataset in Figure 3.8, we plot the inter-arrival times of their out-of-order packets in the ground truth. We see that the campus trace, for which the flow-sampling algorithm is the most accurate in the small-memory regime, has 85% of its out-of-order packets in the ground truth arriving within $2^{-8.6}$ seconds of its predecessor in the same flow. In contrast, in the CAIDA trace, more than 15% of the out-of-order packets corresponding to Def. 1 do not arrive until 32 seconds after its predecessor's arrival. When the memory is small, to sieve through many flows and prefixes, we simply cannot afford wasting much time on one flow, since we may then end up missing many out-of-order events with large inter-arrival times.

3.4.2 Hardware feasibility

We implement a P4 prototype of the flow-sampling algorithm on a Tofino1 switch using 128 lines of code in Lucid [45]. The Lucid-compiled P4 program takes up to 33.33% of the pipeline stages in Tofino1 when using no more than $B = 2^{15}$ buckets. With $B = 2^{16}$ buckets, using not even half of the resources in the first 41.67% pipeline stages, we are able to report 80.44% of the heavily reordered prefixes in the 5-minute campus trace using Def. 1, and 81.19% and 86.08% in the 10-CAIDA 2019 trace using Def. 1 and Def. 2 respectively. Out of the pipeline stages the algorithm makes use of, the resource usage of the prototype with different number of buckets is summarized

Resources	$B = 2^8$	$B = 2^{16}$
Stages	33.33%	41.47%
TCAM	38.54%	26.67%
SRAM	9.38%	33.75%
Hash units	45.83%	36.67%
Instructions	19.53%	15.62%

Table 3.1: Data-plane resource usage in Tofino1.

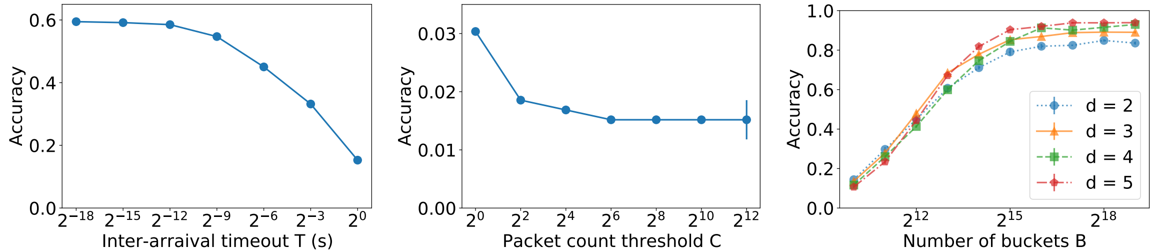
in Table 3.1. In contrast, merely storing the per-flow states for a 10-minute CAIDA 2019 [12] trace could take more register memory than a Tofino1 switch could offer.

3.4.3 Parameter robustness

We started the evaluation using reasonably chosen parameters. Now we verify that all parameters in our algorithms are either easily set, or robust to changes.

To reveal how thresholds T and C individually affect the accuracy of the flow-sampling algorithm, ideally we want to fix one of them to infinity, and vary the other. In this way, only one of them governs the frequency of evictions. Applying this logic, when studying the effect of T (Figure 3.9a), we fix C to a number larger than the length of the entire trace. We see that as long as T is small, the algorithm samples enough flows, and has high accuracy.

Evaluating the effects on a varying C turns out to be less straight-forward. If we make T too large, the algorithm generally suffers from extremely poor performance, which makes it impossible to observe any difference that changing C might bring. If T is too small, the frequency of eviction would be primarily driven by T, and C would not have any impact. And it is not as simple as setting T larger than all inter-arrival times, since eviction only occurs on hash collisions, inter-arrival time alone only paints part of the picture. All evidence above points to the fact that T is the more important parameter. Once we have a good choice of T, the accuracy boost from optimizing C is secondary. Armed with this knowledge, we fix a $T = 2^5$, an ad



(a) The accuracy of the flow-sampling algorithm with varying T, and fixed $B = 2^8$, $R = 1$ and $C = 10^8$.

(b) The accuracy of the flow-sampling algorithm with varying C, with fixed $B = 2^8$, $R = 1$ and $T = 2^5$.

(c) The accuracy of the flow-sampling algorithm with varying d, with fixed $R = 0.01$.

Figure 3.9: The effect of changing parameters on the accuracy of the flow-sampling algorithm and PRECISION.

hoc choice that is by no means perfect. Yet it is enough to observe (Figure 3.9b) that having a small C is slightly more beneficial.

However, C cannot be too small, as inserting a new flow record into the array requires recirculation in the hardware implementation. Programmable switches generally support recirculating up to $3\% - 10\%$ of packets without penalty. Here we set C to be 16, which allows us to achieve line rate.

Given that each non-small flow is continuously monitored for roughly $C = 16$ packets at a time, we report its prefix to the control plane when we encounter any out-of-order packet, that is, $R = 1$.

It is observed in [4] that a small constant $d > 1$ only incurs minimal accuracy loss in finding heavy flows. Increasing d leads to diminishing gains in performance, and adds the number of pipeline stages when implemented on the hardware. Therefore, $d = 2$ is preferable for striking a balance between accuracy and hardware resources.

Building on [4], we evaluate PRECISION for $d = 2, 3, 4, 5$, for reporting out-of-order heavy prefixes. The results in Figure 3.9c show that when the total memory is small, using fewer stages provides a slight benefit. The opposite holds when there is ample memory. However, as the performance gap using different d is insignificant, we also suggest using $d = 2$ for hardware implementations.

3.5 Related work

Characterization of out-of-orderness on the Internet. Packet reordering is first studied in the seminal work by Paxson [40]. It has since been well understood that packet reordering can be caused by parallel links, routing changes, and the presence of adversaries [7]. In typical network conditions, only a small fraction of packets are out-of-order [40, 54]. However, when the network reorders packets, TCP endpoints may wrongly infer that the network is congested, harming end-to-end performance by retransmitting packets and reducing the sending rate [7, 33, 34]. Metrics for characterizing reordering are intensively studied in [38] and [29], though many of the proposed metrics are more suitable for offline analysis. In addition to the network causing packet reordering, the stream of packets in the same TCP connection can appear out of order because congestion along the path leads to packet losses and subsequent retransmissions. Our techniques for identifying IP prefixes with heavy reordering of TCP packets are useful for pinpointing network paths suffering from both kinds of reordering—whether caused by the network devices themselves or induced by the TCP senders in response to network congestion.

Data-plane efficient data structures for volume-based metrics. For heavy-hitter queries, HashPipe [44] adapts SpaceSaving [37] to work with the data-plane constraints, using a multi-stage hash-indexes array. PRECISION [4] further incorporates the idea of Randomized Admission Policy [?] to better deal with the massive number of small flows generally found in network traffic. We extend PRECISION to keep reordering statistics for large flows. However, such an extension cannot be used to detect flows with a large number of out-of-order packets with a reasonable amount of memory.

Data-plane efficient data structures for performance metrics. Liu et al. [35] proposes memory-efficient algorithms for identifying flows with high latency, or lost, reordered, and retransmitted packets. Several solutions for measuring round-

trip delay in the data plane [15, 56, 43] have a similar flavor to identifying out-of-order heavy prefixes, as in both cases keeping at least some state is necessary, with the difference that for reordering we generally need to match more than a pair of packets.

Detecting heavy reordering in the data plane. Several existing systems can detect TCP packet reordering in the data plane. Marple is a general-purpose network telemetry platform with a database-like query language [39]. While Marple can analyze out-of-order packets, the compiler generates a data-plane implementation that requires per-flow state. Unfortunately, such methods consume more memory than the programmable switch can offer in practice. The algorithm proposed by Liu et al. [35] for detecting flows with a large number of out-of-order packets remains the work most related to ours. We note that our lower bound on memory consumption in § 3.1.1.3 is stronger than a similar lower bound (Lemma 10) in [35], as we also allow randomness and approximation. Liu et al. [35] considers out-of-order events specified by Def. 3, and works around the lower bound by assuming out-of-order packets always arrive within some fixed period of time. In contrast, we circumvent the lower bound using the more natural observation that out-of-orderness is correlated among flows within a prefix, and identify heavily reordered prefixes instead of flows.

3.6 Conclusion

In this paper, we introduce two algorithms for identifying out-of-order prefixes in the data plane. In particular, our flow-sampling algorithm achieves good accuracy empirically, even with memory that is orders-of-magnitude smaller than the number of prefixes, let alone the number of flows. When given memory comparable to the number of prefixes, our hybrid scheme using both a heavy-hitter data structure and flow sampling slightly improves the accuracy.

Chapter 4

An Analysis of Random Admission Policy

In this chapter, we consider the performance of the Random Admission Policy (RAP) [6].

4.1 Background

Let S be a stream of elements, where each element x belongs to some universe U. If x appears f times in S, f is the *frequency* of x, and we also say that x has f *occurrences*. We start by laying out the algorithm we consider in this chapter.

The Algorithm. Let $RAP(m)$ denote the Random Admission Policy algorithm (RAP) using m buckets, where each bucket stores both an element and a counter. $RAP(m)$ works as follows:

1. Initialize m empty buckets with counter value $C_i \leftarrow 0$ for all $i \in [m]$;

2. At each step, the algorithm sees an element x in the stream:

    (a) If x is in some bucket i, update counter $C_i \leftarrow C_i + 1$;

(b) Otherwise, find the minimum counter. Let $i_{\min} = \arg\min_{i\in[m]} C_i$. With probability $\frac{1}{C_{i_{\min}}}$, store x in bucket $i_{\min}$ and update counter $C_{i_{\min}} \leftarrow C_{i_{\min}} + 1$.

Related work. In the existing literature, identifying the k most frequent elements is the problem closest to what we study in this chapter. Solving this problem deterministically is believed to be hard in the worst case [13]. A relaxed version, commonly denoted as $\textsc{ApproxTop}(S, k, \varepsilon)$, refers to finding k elements in S, such that each of the k elements has frequency $f_i > (1-\varepsilon)f_k$, where f_k is the kth highest frequency. For any stream, SpaceSaving [37] solves $\textsc{ApproxTop}(S, k, \varepsilon)$ using $\min(|U|, \frac{|S|}{\varepsilon f_k})$ buckets. Among variants of SpaceSaving in the networks community [44, 6, 4], performance guarantees exist only for a simplified version of RAP, and only for i.i.d. Zipfian streams with skew α over universe U [5]. Specifically, when $\alpha < 1$ and $|S| \to \infty$, SpaceSaving requires $O(|U|^{1-\alpha})$ buckets to identify the k most frequent elements with probability 1, while simplified RAP achieves a better bound of $O(|U|^{\frac{1-\alpha}{1+\alpha}})$. For $\alpha = 1$ and $|S| \to \infty$, SpaceSaving requires $O(\log |U|)$, while simiplified RAP only needs $O(\sqrt{\log |U|})$. However, these guarantees do not transfer to the original RAP, since the simplified RAP differs from the original by assuming the entering probability in Step 2b is an absolute constant, and the proof heavily relies on this assumption to go through. Moreover, as the authors in [5] correctly pointed out, the i.i.d. assumption on the stream is also restrictive, as there are many real-world streams that exhibit time locality.

Our result. In this chapter, we consider arbitrarily ordered streams. To capture the performance of $RAP(m)$ under the highly skewed network traffic, we make the following simplifying assumptions to start with. Given some small constant $\varepsilon < 1$, we assume the stream consists of k heavy elements, each of frequency f, and $\varepsilon k f$ number of distinct light elements. Under these assumptions, we show that, if the

stream is sufficiently skewed, the RAP algorithm with k buckets are already capable of identifying a constant fraction of the heavy elements with constant probability.

Theorem 4.1. *Given $\varepsilon < 10^{-7}$, RAP(k) stores at least $0.65k$ heavy elements at the end of the stream, with probability at least 0.7.*

We present the proof of Theorem 4.1 in § 4.2, and discuss how to relax the assumptions and several extensions of Theorem 4.1 in § 4.3.

4.2 Deriving the Performance Bound

First observe that, if a counter value is large, its bucket is more likely to be storing a heavy element. Then to show the data structure stores a constant fraction of heavy elements at the end of the stream, it suffices to show that there are a constant fraction of large counters at the end. We leave this argument to § 4.2.5.

A major challenge in analyzing RAP is the handling of the combination of updating counters deterministically (Step 2a) and overtaking counters with varying probabilities (Step 2b). However, if the total counter value is large, there must be many steps where some counter is incremented deterministically, which indicates that a large fraction of the counters are large. It is trickier when the total counter value is not as large. A key observation is that, for a bucket that is saving some element x, the pace at which its counter value increases is related to how bunched up the occurrences of x are. Suppose the bucket is currently storing the occurrence t of x, if later there are many occurrences of x that are not very far from the tth occurrence, the counter value is very likely to increase quickly. Such an occurrence t is particularly *good* to us when it is stored in the smallest counter, since this implies, that even the smallest counter value would be large at the end.

In § 4.2.1, we formalize the notion of good occurrences and introduce in what sense is a bucket considered good. Following the above intuition, we consider three cases:

- Case (I) at least $2ac\varepsilon fk$ steps where the bucket with the smallest counter value is good (§ 4.2.2);
- Case (II) the total counter value at the end of the stream is at least $(1 - r + a\varepsilon)kf$ (§ 4.2.3); and
- Case (III) at most $2ac\varepsilon kf$ steps where the bucket with the smallest counter value is good, and the total counter value at the end of the stream is at most $(1 - r + a\varepsilon)kf$ (§ 4.2.4).

Note that Case (I) and Case (II) are not disjoint, but this is not a problem. The conditions of these two cases are carefully chosen, such that together they lead to a constant fraction of large counters.

4.2.1 Preliminaries

Let x be a heavy element of frequency f. Denote x_j as the index of the j-th occurrence of x in the stream. We start by defining good occurrences.

Definition 2 (good occurrence)**.** The t-th occurrence of x is *good* with respect to some constant factor c, if for all $t < i \leq f$, $x_i - x_t \leq c(i - t)k$. Conversely, the t-th occurrence of x is *bad* with respect to some constant factor c, if there exists $t < i \leq f$, such that $x_i - x_t > c(i - t)f$.

For each bad occurrence of some heavy element x, there must exist an interval of the stream, in which the occurrences of x are sparse. Take all such intervals, each from a bad occurrence of x, the union of the intervals, due to the sparsity, cannot

contain too many occurrences of x. This means that, each heavy element can only have a constant fraction of bad occurrences.

Claim 4.2. *For any element x of frequency f, the number of bad occurrences of x is at most $\frac{(1+\varepsilon)f}{c}$.*

Proof. For any bad occurrence t of x, let j be the minimum index greater than t, such that $x_j - x_t > c(i-t)k$. Define an index set $I_t = \{t, t+1, \ldots, j-1\}$. Note that given t, I_t is uniquely determined.

Fact 4.3. *Suppose the t-th and the t'-th occurrences of x are both bad, and $|I_t| \leq |I_{t'}|$, then either $I_t \cap I_{t'} = \emptyset$, or $I_t \subseteq I_{t'}$.*

Proof. W.l.o.g. assume $t < t'$, and suppose for contradiction that $I_t \cap I_{t'} \subsetneq I_{t'}$. Let $j = \max\{I_t\}+1$. By definition of I_t, $x_{t'} - x_t \leq ck(t'-t)$. Similarly, $x_j - x_{t'} \leq ck(j-t')$. Then

$$x_j - x_t = x_j - x_{t'} + x_{t'} - x_t \leq ck(j - t' + t' - t) = ck(j-t),$$

contradicting the fact that j is minimum. □

By Fact 4.3, $\bigcup_{\text{bad } t\in[f] \text{ of } x} I_t$ is a union of disjoint sets. Since for all bad t of x, the average gap between two adjacent occurrences in I_t is greater than ck, the average gap between two adjacent occurrences in $\bigcup_{\text{bad } t} I_t$ is also larger than ck. Additionally, $\max\{\bigcup_{\text{bad } t} I_t\} - \min\{\bigcup_{\text{bad } t} I_t\} \leq (1+\varepsilon)kf$. Then $|\bigcup_{\text{bad } t} I_t| \leq \frac{(1+\varepsilon)kf}{ck} = \frac{(1+\varepsilon)f}{c}$. By definition of I_t, all bad occurrences t of x are in $\bigcup_{\text{bad } t} I_t$, therefore, the number of bad occurrences is upper bounded by $\frac{(1+\varepsilon)f}{c}$. □

Then a constant fraction of the stream must consist of good occurrences. To connect good occurrences to the increments of counter values, it is convenient to introduce one more notion, the good buckets.

Definition 3 (good bucket)**.** A bucket $\mathbf{B}$ storing an element x is *good* at step t, if there exists step $t' < t$, such that the following holds:

(1) $\mathbf{C}$ is empty or storing $x' \neq x$ at t', and x is stored in $\mathbf{B}$ at all steps from $t'+1$ to t.

(2) The occurrence of x at step $t'+1$ is good.

Otherwise, we say $\mathbf{B}$ is *bad* at step t.

Note that by Definition 3, whether there is a bad occurrence of x appearing from $t'+2$ to t has no impact on the goodness of bucket $\mathbf{B}$. $\mathbf{B}$ is good as long as the occurrence of x at $t'+1$ is good, and a good $\mathbf{B}$ can only become bad once it is taken over by a bad occurrence, when its counter value become the smallest.

4.2.2 Case (I)

First we consider Case (I), where there are at least $2ac\varepsilon fk$ steps where the bucket with the smallest counter value is good. For brevity, we say some bucket is the smallest when its counter value is the smallest. In steps where the smallest bucket is good, in the amortized sense, the smallest counter value increases by at least 1 for every ck steps. Then, over the entire stream, given that there are at least $\Omega(\varepsilon kf)$ steps where the smallest bucket is good, the smallest counter value will be $\Omega(\varepsilon f)$ at the end of the stream. Since light elements are distinct, as we shall see in § 4.2.5, a counter value of $\Omega(\varepsilon f)$ is large enough for the bucket to be storing a heavy element with constant probability.

Fix a bucket $\mathbf{B}_i$, for each step t where $\mathbf{B}_i$ is bad at step $t-1$ and good at step t, define an interval $I_{tj}^{(i)} = [t, j]$, such that $\mathbf{B}_i$ is good at all steps from t to $j-1$, and bad at step j.

Claim 4.4. *For each interval $I_{tj}^{(i)}$, the counter value of $\mathbf{B}_i$ increases by at least $\lceil \frac{j-t}{ck} \rceil$ from steps t to j.*

Proof. Since $\mathbf{B}_i$ is good from steps t to $j-1$, by Definition 2, the average number of steps between adjacent increments of $\mathbf{B}_i$ is at most ck. Then the increment from t to

$j-1$ is at least $\left\lfloor \frac{j-t-1}{ck} \right\rfloor + 1 \geq \left\lceil \frac{j-t-1}{ck} \right\rceil$, where the increment 1 comes from the first good occurrence at step t. At step j, the counter value increases by 1, which gives that the total increment from t to $j-1$ is at least $\left\lceil \frac{j-t-1}{ck} \right\rceil + 1 \geq \left\lceil \frac{j-t}{ck} \right\rceil$. □

For the rest of the discussion, we omit j in $I_{tj}^{(i)}$ and only write $I_t^{(i)}$, as j is uniquely determined by t. When a set of intervals is disjoint, increments from each interval can be summed up to get the total increments of the smallest counter value over the union of these intervals. Noticing that $\{I_t^{(i)}\}_{t,i}$ is not necessarily disjoint, we make use of the folklore Claim 4.5 to only sum up increments from a disjoint subset of $\{I_t^{(i)}\}_{t,i}$.

Claim 4.5. *Let I be a set of intervals. There exists a set I' of disjoint intervals, $I' \subseteq I$, such that $|I'| \geq \frac{1}{2}|I|$.*

Proof Sketch. Take two sets I' and I'' of disjoint intervals greedily from I. Since $|I' \cup I''| = |I|$, at least one of I' and I'' has length at least $\frac{1}{2}|I|$. □

Lemma 4.6. *If there are at least $2ac\varepsilon fk$ steps where the smallest counter is good, the smallest counter value is at least $\lceil a\varepsilon f \rceil$ at the end of the stream.*

Proof. Consider the union of all intervals $\bigcup_{i,t} I_t^{(i)}$. By definition, $\bigcup_{i,t} I_t^{(i)}$ contains all the steps where the smallest bucket is good. Given that there are at least $2ac\varepsilon fk$ steps where the smallest counter is good, $\left|\bigcup_{i,t} I_t^{(i)}\right| \geq 2ac\varepsilon fk$. By Claim 4.5, there exists a set I' of disjoint intervals, $I' \subseteq \bigcup_{i,t} I_t^{(i)}$, such that $|I'| \geq \frac{1}{2}\left|\bigcup_{i,t} I_t^{(i)}\right| \geq ac\varepsilon fk$. Since the first and the last step in I' corresponds to some buckets being the smallest, the total increment on the smallest counter value over the entire stream is lower bounded by that over I', which is $\left\lceil \frac{ac\varepsilon fk}{ck} \right\rceil = \lceil a\varepsilon f \rceil$ following Claim 4.4.

Moreover, the disjointness of I' indicates that, if we take two adjacent intervals $I_{t_1}^{(i_1)}$ and $I_{t_2}^{(i_2)}$, where the smallest counter value increases from v_1 to v_1' on $I_{t_1}^{(i_1)}$, v_2 to v_2' on $I_{t_2}^{(i_2)}$, we must have $v_1 < v_1' < v_2 < v_2'$. Therefore, given that the smallest counter value is always 0 to begin with, the smallest counter value at the end of the

stream is lower bounded by the total increment on the smallest counter over the entire stream. ☐

4.2.3 Case (II)

Next we consider the simplest case, where the total counter value is at least $(1-r+a\varepsilon)kf$ at the end of the stream. Since the largest and the smallest counter value at the end of the stream cannot be more than f apart, if the total counter value at the end of the stream is large, there cannot be a large fraction of small counters.

Denote $v_i(t)$ as the counter value of bucket $\mathbf{B}_i$ after seeing the t-th element in the stream. Let $v(t) = \sum_{i\in[k]} v_i(t)$ be the total counter value after seeing the t-th element in the stream. Recall that the stream has length $(1+\varepsilon)kf$, then $v((1+\varepsilon)kf)$ denotes the total counter value at the end of the stream.

Claim 4.7. *The difference between the largest and the smallest counter value is at most f at the end of the stream.*

Proof. Fix any largest bucket $\mathbf{B}_l$ at the end of the stream, that is, its counter value $v_l((1+\varepsilon)kf) = \max_{i\in[k]} v_i((1+\varepsilon)kf)$. Suppose $\mathbf{B}_l$ is storing x at the end of the stream, let t be the step such that $\mathbf{B}_l$ stores some $x' \neq x$ at time t, and $\mathbf{C}_l$ stores x from time $t+1$ to $(1+\varepsilon)kf$. Let $\mathbf{B}_s$ be any smallest bucket at the end of the stream. As counter values are non-decreasing, $\mathbf{B}_s$'s counter value v_s satisfies $v_s((1+\varepsilon)kf) \geq v_s(t)$. Moreover, a bucket only switches to saving a new element when it is the smallest, which means $v_s(t) \geq v_l(t) \geq v_l((1+\varepsilon)kf) - f$. Altogether, we have $v_s((1+\varepsilon)kf) + f \geq v_l((1+\varepsilon)kf)$. ☐

Lemma 4.8. *If the total counter value $v((1+\varepsilon)kf) \geq (1-r+a\varepsilon)kf$, then the number of large buckets $|\{i \in [k] \mid v_i((1+\varepsilon)kf) \geq a\varepsilon f\}| \geq (1-r)k$.*

Proof. Suppose $|\{i \in [k] \mid v_i((1+\varepsilon)kf) \le a\varepsilon f\}| > rk$, we get a contradiction from

$$\begin{aligned} v((1+\varepsilon)kf) &< rk \cdot a\varepsilon f + (1-r)k \cdot (a\varepsilon f + f) && \text{(Claim 4.7)} \\ &= a\varepsilon kf + (1-r)kf \\ &= (1-r+a\varepsilon)kf. \end{aligned}$$

□

4.2.4 Case (III)

When there are at most $2ac\varepsilon kf$ steps where the bucket with the smallest counter value is good, and the total counter value at the end of the stream is at most $(1-r+a\varepsilon)kf$, we will not directly argue about the counter values as in previous cases. Instead, we will show that, the conditions of Case (III) lead to contradictions with constant probability. Then, it must fall into Case (I) and Case (II) with constant probability.

We begin with a subtly wrong but intuitive argument. On the one hand, good occurrences make up a large constant fraction of the stream (Claim 4.2). On the other, by conditions of Case (III), we do not have too many steps where we deterministically increment some counter value. Together they imply that, there are many steps where we increase the number of good buckets. If we can show the expected increase on the number of good buckets is at least $2k$ on some portion of the stream, then considering that there are only k buckets in total, we must run into contradiction with some probability. Recall that we can only change a bad bucket into a good one when its counter value is the smallest. Then to bound the expected increase on the number of good buckets, we need the range of the smallest counter value.

Divide the stream into stages according to the smallest counter values, where in stage j, the smallest counter value lies in $[2^j, 2^{j+1})$. Further denote L_j as the length of stage j, G_j the number of steps in stage j where the smallest counter is good, I_j the

number of steps where the next occurrence updates a counter in memory, and B_j the number of steps where the next occurrence is bad. Then in stage j, we increase the number of good buckets with probability at least $\frac{1}{2^{j+1}}$ in each of the $L_j - G_j - B_j - I_j$ steps, and decrease the number of good buckets with probability at most $\frac{1}{2^j}$ in each of the G_j steps. Therefore, the expected increase on the number of good buckets in stage j is lower bounded by

$$(L_j - G_j - B_j - I_j) \cdot \frac{1}{2^{j+}} - G_j \cdot \frac{1}{2^j}. \tag{4.1}$$

As long as there exists a stage j, such that (4.1) is greater than $2k$, we can use concentration to show that this leads to a contradiction.

While this argument provides great intuition on what we are about to do, the probabilities are subtly incorrect. Conditioned on being in Case (III), the randomness in Step 2b of the algorithm might be affected. Nonetheless, parts of this argument can be rescued. Next in Lemma 4.9, we show a version of $(L_j - G_j - B_j - I_j) \cdot \frac{1}{2^{j+}} - G_j \cdot \frac{1}{2^j} > 2k$ parameterized by a constant x, where we treat factors like $\frac{1}{2^{j+1}}$ as numbers we magically chose, rather than the probabilities in the flawed argument. Finally the introduction of x helps us bound the probability of getting contradiction using the union bound, thus circumventing the issue of conditioning.

Lemma 4.9. *Fix a stream, for each fixed sequence of memory states satisfying Case (III), given constant $0 < x < 1$, if $\varepsilon < (r - \frac{1}{c})(2ac(\frac{1}{x^2} + 1) + a(\frac{4}{x} + 1) + \frac{1}{c} - 1)^{-1}$, there exists a stage $j = j(x)$, such that*

$$(L_j - G_j - B_j - I_j) - \frac{G_j}{x^2} > \frac{2^{j+1} \cdot k}{x} \tag{4.2}$$

Proof. Suppose for contradiction that for all stage j, $(L_j - G_j - B_j - I_j) - \frac{G_j}{x^2} > \frac{2^{j+1} \cdot k}{x}$. Taking the sum over all stages, we have

$$\sum_j L_j - (1 + \frac{1}{x^2}) \sum_j G_j - \sum_j B_j - \sum_j I_j \leq \frac{k}{x} \sum_j 2^{j+1}. \tag{4.3}$$

Since $\sum_j L_j$ is bounded by the total stream length $(1+\varepsilon)kf$, the upper bounds on $\sum_j G_j$ and $\sum_j I_j$ are given by the condition of Case II, and $\sum_j B_j \leq \frac{(1+\varepsilon)kf}{c}$, the number of bad occurrences in the stream,

$$\begin{aligned} &\sum_j L_j - (1 + \frac{1}{x^2}) \sum_j G_j - \sum_j B_j - \sum_j I_j \\ \geq &(1+\varepsilon)kf - 2ac(1 + \frac{1}{x^2})\varepsilon kf - \frac{(1+\varepsilon)kf}{c} - (1 - r + a\varepsilon)kf \\ = &kf \left(r - \frac{1}{c} - \left(2ac \left(\frac{1}{x^2} + 1 \right) + \frac{1}{c} + a \right) \varepsilon \right). \end{aligned}$$

On the other hand, we first notice that $j < \log(a\varepsilon f)$, since the smallest counter value at the end of the stream is smaller than $a\varepsilon f$ (as we are done otherwise). Then

$$\frac{k}{x} \sum_j 2^{j+1} < \frac{k}{x} \sum_{j=0}^{\log(a\varepsilon f)} 2^{j+1} < \frac{4}{x} a\varepsilon kf.$$

For $\varepsilon < (r - \frac{1}{c})(2ac(\frac{1}{x^2}+1) + a(\frac{4}{x}+1) + \frac{1}{c} - 1)^{-1}$, we have $r - \frac{1}{c} - \left(2ac\left(\frac{1}{x^2}+1\right) + \frac{1}{c} + a\right)\varepsilon > \frac{4}{x}a\varepsilon$, which contradicts (4.3). □

Consider all sequences of actions by the algorithm that lead to memory states corresponding to Case (III). For any sequence $\mathcal{S}$ of actions in Case (III), there exists a stage j satisfying (4.2) by Lemma 4.9. This gives us a way to categorize such sequences of actions. We assign $\mathcal{S}$ to group j, if stage j in $\mathcal{S}$ satisfies (4.2). For an $\mathcal{S}$ that has multiple stages where (4.2) is satisfied, we only assign $\mathcal{S}$ to one group. Then it follows that all groups together form a partition of the set of all sequences of actions in Case (III).

Let $\mathcal{S}$ be a sequence of actions from a fixed group j. $\mathcal{S}$ induces a sequence of memory states, which further determines L_j, G_j, B_j and I_j. Let $N_1 = L_j - G_j - B_j - I_j$, and $N_2 = G_j$, we have $N_1 > N_2$ as a consequence of Lemma 4.9. Further denote n_1 and n_2 as the number of steps where the number of good counters are increased and decreased in stage j, respectively. We look at the probability that $\mathcal{S}$ occurs:

$$\begin{aligned}\mathbb{P}[\mathcal{S} \text{ occurs}] &= \mathbb{P}[\text{all stages } i \neq j \text{ of } \mathcal{S} \text{ occur}] \cdot \mathbb{P}[\text{stage } j \text{ of } \mathcal{S} \text{ occurs}] \\ &\leq \mathbb{P}[\text{all stages } i \neq j \text{ of } \mathcal{S} \text{ occur}] \\ &\quad \cdot \left(\frac{1}{2^j}\right)^{n_1} \left(1 - \frac{1}{2^{j+1}}\right)^{N_1 - n_1} \left(\frac{1}{2^j}\right)^{n_2} \left(1 - \frac{1}{2^{j+1}}\right)^{N_2 - n_2}. \end{aligned} \tag{4.4}$$

$$\begin{aligned}\sum_{\mathcal{S}: j, N_1, N_2, n_1, n_2} \mathbb{P}[\mathcal{S} \text{ occurs}] &\leq \left(\frac{1}{2^j}\right)^{n_1} \left(1 - \frac{1}{2^{j+1}}\right)^{N_1 - n_1} \left(\frac{1}{2^j}\right)^{n_2} \left(1 - \frac{1}{2^{j+1}}\right)^{N_2 - n_2} \\ &\quad \cdot \sum_{\mathcal{S}: j, N_1, N_2, n_1, n_2} \mathbb{P}[\text{all stages } i \neq j \text{ of } \mathcal{S} \text{ occur}] \\ &\leq \left(\frac{1}{2^j}\right)^{n_1} \left(1 - \frac{1}{2^{j+1}}\right)^{N_1 - n_1} \left(\frac{1}{2^j}\right)^{n_2} \left(1 - \frac{1}{2^{j+1}}\right)^{N_2 - n_2}. \end{aligned} \tag{4.5}$$

Finally we upper bound the probability that there is no contradiction from sequences $\mathcal{S}$ with matching j, N_1, N_2, n_1, n_2, that is, the probability of $n_1 - n_2 \leq k$:

$$
\begin{aligned}
\mathbb{P}[n_1 - n_2 \leq k] &\leq \sum_{n_1 - n_2 \leq k} \binom{N_1}{n_1}\binom{N_2}{n_2} \sum_{\mathcal{S}: j, N_1, N_2, n_1, n_2} \mathbb{P}[\mathcal{S} \text{ occurs}] \\
&\leq \sum_{n_1 - n_2 \leq k} \binom{N_1}{n_1}\left(\frac{1}{2^j}\right)^{n_1}\left(1 - \frac{1}{2^{j+1}}\right)^{N_1 - n_1} \\
&\qquad \cdot \binom{N_2}{n_2}\left(\frac{1}{2^j}\right)^{n_2}\left(1 - \frac{1}{2^{j+1}}\right)^{N_2 - n_2} \\
&= \sum_{n_1 = k}^{N_2} \binom{N_1}{n_1}\left(\frac{1}{2^j}\right)^{n_1}\left(1 - \frac{1}{2^{j+1}}\right)^{N_1 - n_1} \\
&\qquad \cdot \sum_{n_2 = n_1 - k}^{n_1} \binom{N_2}{n_2}\left(\frac{1}{2^{j+1}}\right)^{n_2}\left(1 - \frac{1}{2^{j+1}}\right)^{N_2 - n_2} \cdot 2^{n_2} \\
&\leq \sum_{n_1 = k}^{N_2} \binom{N_1}{n_1}\left(\frac{1}{2^j}\right)^{n_1}\left(1 - \frac{1}{2^{j+1}}\right)^{N_1 - n_1} \cdot 2^{n_1} \\
&\qquad \cdot \sum_{n_2 = n_1 - k}^{n_1} \binom{N_2}{n_2}\left(\frac{1}{2^{j+1}}\right)^{n_2}\left(1 - \frac{1}{2^{j+1}}\right)^{N_2 - n_2} \\
&= \sum_{n_1 = k}^{N_2} \binom{N_1}{n_1}\left(\frac{1}{2^{j-1}}\right)^{n_1}\left(1 - \frac{1}{2^{j+1}}\right)^{N_1 - n_1} \\
&\qquad \cdot \sum_{n_2 = n_1 - k}^{n_1} \binom{N_2}{n_2}\left(\frac{1}{2^{j+1}}\right)^{n_2}\left(1 - \frac{1}{2^{j+1}}\right)^{N_2 - n_2}. \quad (4.6)
\end{aligned}
$$

In the remainder of this section, we spend all the efforts on bounding (4.6). Let $f(n_1) = \binom{N_1}{n_1}\left(\frac{1}{2^{j-1}}\right)^{n_1}\left(1 - \frac{1}{2^{j+1}}\right)^{N_1 - n_1}$, $g(n_2) = \binom{N_2}{n_2}\left(\frac{1}{2^{j+1}}\right)^{n_2}\left(1 - \frac{1}{2^{j+1}}\right)^{N_2 - n_2}$. Note that $g(n_2)$ is precisely the PMF of a binomial distribution with parameters N_2 and $2^{-(j+1)}$, $g(n_2)$ is hence unimodal. By setting $\frac{g(n_2+1)}{g(n_2)} \leq 1$, we find that $g(n_2)$ monotonically increases when $n_2 < \frac{N_2}{2^{j+1}}$. $f(n_1)$ follows a similar trend. $\frac{g(n_1+1)}{g(n_1)} \leq 1$ implies that $n_1 \geq \frac{N_1+1}{2^{j-1}+\frac{3}{4}} - 1$. Let $n_1^* = \frac{N_1}{2^{j-1}}$, then for some small constant fraction x, we can ensure that $f(n_1)$ monotonically increases when $n_1 \leq x n_1^*$.

To bound (4.6), we partition the range of $k \leq n_1 \leq N_2$ into two smaller ranges separated by $x n_1^*$, and first consider the sum over $k \leq n_1 \leq x n_1^*$. Since $\sum_{n_2 = n_1 - k}^{n_1} g(n_2) \leq$

1, and $f(n_1)$ monotonically increases in this range,

$$\sum_{n_1=k}^{xn_1^*} f(n_1) \sum_{n_2=n_1-k}^{n_1} g(n_2) \leq (xn_1^* - k + 1) \cdot f(xn_1^*) \leq xn_1^* \cdot f(xn_1^*). \tag{4.7}$$

Next we upper bound $f(xn_1^*)$:

$$\begin{aligned}
f(xn_1^*) &= \binom{N_1}{xn_1^*} \left(\frac{1}{2^{j-1}}\right)^{xn_1^*} \left(1 - \frac{1}{2^{j+1}}\right)^{N_1 - xn_1^*} \\
&\leq \left(\frac{eN_1}{xn_1^*} \cdot \frac{1}{2^{j-1}} \cdot \frac{1}{1-2^{-(j+1)}}\right)^{xn_1^*} \cdot \left(1 - \frac{1}{2^{j+1}}\right)^{N_1} && (\binom{n}{k} \leq \left(\frac{en}{k}\right)^k) \\
&= \left(\frac{1}{x} \cdot \frac{e}{1-2^{-(j+1)}}\right)^{xn_1^*} \cdot \left(1 - \frac{1}{2^{j+1}}\right)^{N_1} && (\text{definition of } n_1^*) \\
&\leq \left(\frac{2e}{x}\right)^{xn_1^*} \left(1 - \frac{1}{2^{j+1}}\right)^{N_1} && (\text{for } j \geq 0, \frac{e}{1-2^{-(j+1)}} \leq 2e) \\
&= \left(\frac{2e}{x} \cdot \left(1 - \frac{1}{2^{j+1}}\right)^{\frac{N_1}{xn_1^*}}\right)^{xn_1^*} \\
&\leq \left(\frac{2e}{x} \cdot e^{-\frac{N_1}{xn_1^* \cdot 2^{j+1}}}\right)^{xn_1^*} && (1 - x \leq e^{-x} \text{ for } x > 0) \\
&= \left(\frac{2e}{x} \cdot e^{-\frac{1}{4x}}\right)^{xn_1^*}. && (\text{definition of } n_1^*)
\end{aligned} \tag{4.8}$$

Together with (4.7), we get $\sum_{n_1=k}^{xn_1^*} f(n_1) \sum_{n_2=n_1-k}^{n_1} g(n_2) \leq xn_1^* \cdot \left(\frac{2e}{x} \cdot e^{-\frac{1}{4x}}\right)^{xn_1^*}$. We choose $x = \frac{1}{30}$, and check that $xn_1^* \cdot \left(\frac{2e}{x} \cdot e^{-\frac{1}{4x}}\right)^{xn_1^*} < \frac{1}{6}$. Therefore, for $x = \frac{1}{30}$ we have

$$\sum_{n_1=k}^{xn_1^*} f(n_1) \sum_{n_2=n_1-k}^{n_1} g(n_2) < \frac{1}{6}. \tag{4.9}$$

To bound $\sum_{n_1=xn_1^*+1}^{N_2} f(n_1) \sum_{n_2=n_1-k}^{n_1} g(n_2)$, we make use of the fact that the maximizers of $f(n_1)$ and $g(n_2)$ are far apart. Let $n_2^* = \frac{N_2}{2^{j+1}}$. First we deduce several useful consequences of Lemma 4.9.

Corollary 4.10. *Fix a stream, for each fixed sequence of memory states satisfying Case II, given constant* $0 < x < 1$, *if* $\varepsilon < (r - \frac{1}{c})(\frac{1}{x^2} + \frac{4a}{x} + \frac{1}{c} + a)^{-1}$, *there exists a stage* j, *for which the following holds simultaneously:*

(1) $xn_1^* - k > n_2^*$;

(2) $\frac{1}{4}(xn_1^* - k) > \frac{n_2^*}{x}$; *and*

(3) $n_1^* > \frac{4}{x}k$.

Proof. By Lemma 4.9, given x and ε, there exists a stage j, such that $N_1 - \frac{N_2}{x^2} > \frac{2^{j+1} \cdot k}{x}$. That is, $\frac{xn_1^*}{4} - k > \frac{n_2^*}{x}$. Corollary 4.10((1)) follows from $xn_1^* - k > \frac{xn_1^*}{4} - k > \frac{n_2^*}{x} > n_2^*$. Corollary 4.10((2)) follows from $\frac{1}{4}(xn_1^* - k) > \frac{xn_1^*}{4} - k > \frac{n_2^*}{x}$. Corollary 4.10((3)) follows from $\frac{xn_1^*}{4} - k > \frac{n_2^*}{x} > 0$. □

By Corollary 4.10((1)), $g(n_2)$ monotonically decreases as n_2 goes from $n_1 - k$ to n_1, and $n_1 > xn_1^*$, then

$$\sum_{n_1 = xn_1^* + 1}^{N_2} f(n_1) \sum_{n_2 = n_1 - k}^{n_1} g(n_2) \leq \sum_{n_1 = xn_1^* + 1}^{N_2} kf(n_1)g(n_1 - k). \tag{4.10}$$

Let $h(n) = kf(n)g(n-k)$, and consider

$$r(n) = \frac{h(n+1)}{h(n)} = \frac{4}{(2^{j+1} - 1)^2} \cdot \frac{N_1 - n}{n+1} \cdot \frac{N_2 - n + k}{n - k + 1}. \tag{4.11}$$

For the choice of $x = \frac{1}{30}$, we have $r(xn_1^*) < 1$:

$$\begin{aligned}
r(xn_1^*) &= \frac{4}{(2^{j+1}-1)^2} \cdot \frac{N_1 - xn_1^*}{xn_1^* + 1} \cdot \frac{N_2 - xn_1^* + k}{xn_1^* - k + 1} \\
&< \frac{4}{(2^{j+1}-1)^2} \cdot \frac{N_1}{xn_1^*} \cdot \frac{N_2}{xn_1^* - k} \\
&< \frac{16}{2^{2(j+1)}} \cdot \frac{N_1}{xn_1^*} \cdot \frac{N_2}{xn_1^* - k} && \left(\frac{1}{2^{j+1}-1} \leq \frac{2}{2^{j+1}}\right) \\
&= 4 \cdot \frac{4N_1}{2^{j+1} \cdot xn_1^*} \cdot \frac{N_2}{2^{j+1} \cdot (xn_1^* - k)} \\
&= \frac{4n_2^*}{xn_1^* - k} < 1. && \text{(Corollary 4.10(2))}
\end{aligned}$$

By (4.11), $r(n)$ monotonically decreases as n increases, combined with $r(xn_1^*) < 1$, this means that $r(n) < 1$ for all $n > xn_1^*$, and consequently, $h(n) < h(xn_1^*)$ when $n > xn_1^*$. This allows us to further upper bound (4.10).

Analogous to (4.7), as a first attempt, we may try upper bounding (4.10) by $N_2 \cdot h(xn_1^*)$. However, since N_2 is relatively large, such a loose upper bound would not lead to a constant probability. Instead, we partition the range of $xn_1^* \leq n_1 \leq N_2$ into finer intervals of lengths $2^i \cdot xn_1^*$, $i = 0, 1, \ldots, i_{max}$, where i_{max} is the minimum i such that $\sum_{i=0}^{i_{max}} 2^i \cdot xn_1^* \geq N_2 - xn_1^*$. (Here i_{max} is for notational conveniences, the specific value of i_{max} is not used later on.) Then we make use of the monotonicity of $h(n)$ on each such interval:

$$\sum_{n_1 = xn_1^*+1}^{N_2} kf(n_1)g(n_1 - k) \leq \sum_{i=0}^{i_{max}} 2^i xn_1^* \cdot h(2^i xn_1^*) \tag{4.12}$$

To bound $f(2^i x n_1^*)$ and $g(2^i x n_1^* - k)$, we replicate the steps in (4.8), and again use Lemma 4.9:

$$\begin{aligned}
f(2^i x n_1^*) &= \binom{N_1}{2^i x n_1^*}\left(\frac{1}{2^{j-1}}\right)^{2^i x n_1^*}\left(1-\frac{1}{2^{j+1}}\right)^{N_1 - 2^i x n_1^*} \\
&\leq \left(\frac{eN_1}{2^i x n_1^*}\cdot\frac{1}{2^{j-1}}\cdot\frac{1}{1-2^{-(j+1)}}\right)^{2^i x n_1^*}\cdot\left(1-\frac{1}{2^{j+1}}\right)^{N_1} && \left(\binom{n}{k}\leq\left(\frac{en}{k}\right)^k\right) \\
&= \left(\frac{1}{2^i x}\cdot\frac{e}{1-2^{-(j+1)}}\right)^{2^i x n_1^*}\cdot\left(1-\frac{1}{2^{j+1}}\right)^{N_1} && (\text{definition of } n_1^*) \\
&\leq \left(\frac{2e}{2^i x}\right)^{2^i x n_1^*}\left(1-\frac{1}{2^{j+1}}\right)^{N_1} && \left(\text{for } j\geq 0, \frac{e}{1-2^{-(j+1)}}\leq 2e\right) \\
&= \left(\frac{2e}{2^i x}\cdot\left(1-\frac{1}{2^{j+1}}\right)^{\frac{N_1}{2^i x n_1^*}}\right)^{2^i x n_1^*} \\
&\leq \left(\frac{2e}{2^i x}\cdot e^{-\frac{N_1}{2^i x n_1^*\cdot 2^{j+1}}}\right)^{2^i x n_1^*} && (1-x\leq e^{-x} \text{ for } x>0) \\
&= \left(\frac{2e}{2^i x}\cdot e^{-\frac{1}{4x\cdot 2^i}}\right)^{2^i x n_1^*}. && (\text{definition of } n_1^*)
\end{aligned} \tag{4.13}$$

$$
\begin{aligned}
& g(2^i x n_1^* - k) \\
=& \binom{N_2}{2^i x n_1^* - k} \left(\frac{1}{2^{j+1}}\right)^{2^i x n_1^* - k} \left(1 - \frac{1}{2^{j+1}}\right)^{N_2 - 2^i x n_1^* + k} \\
\leq & \left(\frac{eN_2}{2^i x n_1^* - k} \cdot \frac{1}{2^{j+1}} \cdot \frac{1}{1 - 2^{-(j+1)}}\right)^{2^i x n_1^* - k} \cdot \left(1 - \frac{1}{2^{j+1}}\right)^{N_2} \\
=& \left(\frac{N_2}{2^{j+1}(2^i x n_1^* - k)} \cdot \frac{e}{1 - 2^{-(j+1)}}\right)^{2^i x n_1^* - k} \cdot \left(1 - \frac{1}{2^{j+1}}\right)^{N_2} \\
=& \left(\frac{\frac{N_2}{2^{j+1}}}{2^i \cdot 4x(\frac{N_1}{2^{j+1}} - \frac{k}{2^i \cdot 4x})} \cdot \frac{e}{1 - 2^{-(j+1)}}\right)^{2^i x n_1^* - k} \cdot \left(1 - \frac{1}{2^{j+1}}\right)^{N_2} \\
\leq & \left(\frac{1}{2^i \cdot 4x} \cdot \frac{\frac{N_2}{2^{j+1}}}{\frac{N_1}{2^{j+1}} - \frac{k}{x}} \cdot \frac{e}{1 - 2^{-(j+1)}}\right)^{2^i x n_1^* - k} \cdot \left(1 - \frac{1}{2^{j+1}}\right)^{N_2} \\
\leq & \left(\frac{x}{2^i \cdot 4} \cdot \frac{e}{1 - 2^{-(j+1)}}\right)^{2^i x n_1^* - k} \cdot \left(1 - \frac{1}{2^{j+1}}\right)^{N_2} && \left(\frac{\frac{N_2}{2^{j+1}}}{\frac{N_1}{2^{j+1}} - \frac{k}{x}} \leq x^2 \text{ by Lemma 4.9}\right) \\
\leq & \left(\frac{ex}{2^{i+1}}\right)^{2^i x n_1^* - k} \cdot \left(1 - \frac{1}{2^{j+1}}\right)^{N_2} && \text{for } j \geq 0, \frac{e}{1 - 2^{-(j+1)}} \leq 2e \\
=& \left(\frac{ex}{2^{i+1}} \cdot \left(1 - \frac{1}{2^{j+1}}\right)^{\frac{N_2}{2^i x n_1^* - k}}\right)^{2^i x n_1^* - k} && \left(1 - \frac{1}{2^{j+1}}\right)^{\frac{N_2}{2^i x n_1^* - k}} \leq 1 \\
\leq & \left(\frac{ex}{2^{i+1}}\right)^{2^i x n_1^* - k}. && (4.14)
\end{aligned}
$$

Plugging in (4.13) and (4.14) gives

$$
\begin{aligned}
2^i x n_1^* \cdot h(2^i x n_1^*) &= 2^i x n_1^* \cdot k \cdot f(2^i x n_1^*) \cdot g(2^i x n_1^* - k) \\
&\leq 2^i x n_1^* \cdot k \cdot \left(\frac{2e}{2^i x} \cdot e^{-\frac{1}{4x \cdot 2^i}}\right)^{2^i x n_1^*} \cdot \left(\frac{ex}{2^{i+1}}\right)^{2^i x n_1^* - k} \\
&= 2^i x n_1^* \cdot k \cdot \left(\frac{ex}{2^{i+1}}\right)^{-k} \cdot \left(\frac{2e}{2^i x} \cdot e^{-\frac{1}{4x \cdot 2^i}} \cdot \frac{ex}{2^{i+1}}\right)^{2^i x n_1^*} \\
&= 2^i x n_1^* \cdot k \cdot \left(\frac{ex}{2^{i+1}}\right)^{-k} \cdot \left(\frac{1}{4^i} \cdot e^{2 - \frac{1}{4x \cdot 2^i}}\right)^{2^i x n_1^*}. \qquad (4.15)
\end{aligned}
$$

Notice that the terms in (4.15) related to n_1^*, denoted as $l_i(n_1^*) = n_1^* \left(\frac{1}{4^i} \cdot e^{2 - \frac{1}{4x \cdot 2^i}}\right)^{2^i x n_1^*}$, is maximized at $n_i = \frac{4}{2^{i+2} x (i \ln 4 - 2)}$, and monotonically decreases when $n_1^* > n_i$. Fur-

thermore, n_i is maximized when i is minimized. This means that, as long as the lower bound of n_1^* is greater than $n_0 = \frac{4}{1-8x}$, we can upper bound (4.15) by replacing n_1^* with its lower bound. One can easily check that Corollary 4.10((3)) gives such a lower bound on n_1^*. Then we have

$$2^i x n_1^* \cdot h(2^i x n_1^*) \leq 2^i \cdot 4k^2 \cdot \left(\frac{ex}{2^{i+1}}\right)^{-k} \cdot \left(\frac{1}{4^i} \cdot e^{2-\frac{1}{4x \cdot 2^i}}\right)^{2^i \cdot 4k}. \tag{4.16}$$

Denote the RHS of (4.16) as a_i. To upper bound (4.12) by a constant, we notice that the sequence of $\{a_i\}_i$ is dominated element-wise by a convergent geometric series. To see that,

$$q_i \triangleq \frac{a_{i+1}}{a_i} = 2^{1-k} \cdot \left(\frac{e}{2^{i+2}}\right)^{2^{i+3}k}.$$

q_i decreases very fast as i increases, we have $q_i \leq q_0 = 2^{1-k} \cdot \left(\frac{e}{4}\right)^{8k} < 0.05$ for all $i \geq 0$. Therefore, $\{a_i\}_i$ is dominated by a geometric series with coefficient $a_0 = 4k^2 \cdot \left(\frac{ex}{2}\right)^{-k} \cdot e^{8k-\frac{k}{x}} < 0.006$ and common ratio q_0, then

$$\sum_{n_1 = x n_1^* + 1}^{N_2} f(n_1) \sum_{n_2 = n_1 - k}^{n_1} g(n_2) \leq \frac{a_0}{1 - q_0} < 0.0064. \tag{4.17}$$

(4.9) together with (4.17) gives the upper bound on the probability in (4.6), that is, $\mathbb{P}[n_1 - n_2 \leq k] < \frac{4}{23}$.

4.2.5 Aggregating all cases

To conclude, notice that both Case I (§ 4.2.2) and Case II (§ 4.2.3) are deterministically good, in the sense that either all counters are relatively large at the end, or at least a significant fraction of the counters are relatively large. And the conditions of Case III (§ 4.2.4) induce contradiction with probability at least $\frac{19}{23}$, which means we

are in the good Cases I and II with probability at least $\frac{19}{23}$. Then, we argue that a large counter is unlikely to be storing a light element at the end.

For a light element l, the probability that l is stored in a counter with value at least $a\varepsilon f$ is upper bounded by $\frac{1}{a\varepsilon f}$, due to the distinctness of the light elements. By linearity of expectation, there can only be at most $\varepsilon kf \cdot \frac{1}{a\varepsilon f} = \frac{k}{a}$ counters in expectation with value at least $a\varepsilon f$ that stores light elements at the end. Markov's inequality then gives that, with probability at least $1 - \frac{1}{b}$ for some constant $b > 1$, there can be at most $\frac{bk}{a}$ light elements stored in the counters with value at least $a\varepsilon f$. Finally by union bound, with probability at least $\frac{19}{23} - \frac{1}{b}$, there must be at least $1 - r - \frac{b}{a}$ fraction of the counters storing heavy elements at the end.

To get the specific constants in Theorem 4.1, first recall that x was set to $\frac{1}{30}$ in bounding (4.9). Plugging $x = \frac{1}{30}$ into the upper bound on ε in Lemma 4.9, we get that given $\varepsilon < (r - \frac{1}{c})(1802ac + 121a + \frac{1}{c} - 1)^{-1}$, with probability at least $\frac{19}{23} - \frac{1}{b}$, $RAP(k)$ stores at least $(1 - r - \frac{b}{a})k$ number of heavy elements at the end of the stream. Ideally we want to set constants a, b, c and r such that the upper bound on ε, the probability, and the fraction of heavy elements are all large. However, for each of a, b and r, increasing the constant would inevitably increase one term, but decrease another. It is only possible to optimize the upper bound on ε over c, given a and r, by setting $c = \frac{\sqrt{12988816a^2+7208ar(121a+r-1)}}{3604ar} + 1$. Theorem 4.1 then follows from setting $a = 85$, $r = \frac{1}{4}$, and $b = c = 8$.

4.3 Conclusion

In this chapter, we proved the first performance guarantee for RAP on highly skewed streams (Theorem 4.1). To present an argument that is as clean as possible, we strictly assumed all the k heavy elements have the same frequency, and all light elements are distinct. In fact, this assumption, together with the specific memory

size k we used, is not fundamental to our approach. In this section, we discuss several potential extensions.

Relaxing the assumption on the stream. When heavy elements have frequencies that lie in $[uf, f]$ for some constant $0 < u < 1$, we would lose a constant factor in Lemma 4.8, which causes a constant factor loss on the fraction of buckets storing heavy elements. If the light elements are no longer distinct, it would be messier to account for the probability that a light element l is stored in a bucket with counter value $O(\varepsilon f)$ (§ 4.2.5), as l would not necessarily need to enter the bucket in the last step. However, as long as the frequency of l is upper bounded by a small constant, we can still bound that probability, which consequently impacts all terms in Theorem 4.1 by constant factors. Regardless, it is clear that a result similar to Theorem 4.1 holds for a more general stream.

Generalizing to different memory sizes. The easiest way to adapt the argument to accommodate a different memory size m would be to set $m = uk$ for some constant u. This immediately changes Lemma 4.8 and Lemma 4.9 by constant factors, and for (4.6), we will bound $\mathbb{P}[n_1 - n_2 \leq uk]$.

Improving constants. In § 4.2.5, we a the version of Theorem 4.1 before nailing down all the constants. Admittedly, given its current form, there is no hope to come to a performance bound that closely matches what was observed in practice. However, there are still places where we could potentially improve the constants. For the probability of contradiction (§ 4.2.4), notice that the probability upper bound in (4.9) is quite large compare to that in (4.17). To get (4.9), we simply used the loose bound of $\sum_{n_2=n_1-k}^{n_1} g(n_2) \leq 1$. A more careful analysis could further lower the probability of having no contradiction in Case (III) (§ 4.2.4). In addition, also in

Case (III), we could optimize the way the stream is partitioned into stages. Then, in adjusting (4.2) and (4.6), we may improve all terms in Theorem 4.1.

Chapter 5

Conclusion

Performance monitoring is crucial in running today's network. With the emergence of programmable network switches, we now have the option to run performance measurements directly in the data plane. However, performance metrics are fundamentally expensive to measure. To make matters worse, programmable data planes are highly constrained in both the memory size and the number of memory accesses per packet. To address these challenges, this dissertation focuses on the design and analysis of compact algorithms for measuring network performance in the data plane. Next we summarize our contributions, future directions, and conclude with final remarks.

5.1 Summary of Contributions

The technical chapters in this thesis are divided into two parts. In the first part (§ 2 and 3), we propose novel algorithms for monitoring delay and TCP packet reordering in the data plane. Both works leverage probabilistic techniques to attain good accuracy with limited memory resources. In the remaining part (§ 4), we derive the first formal performance guarantees for RAP, introducing techniques to analyze counter-based algorithms in practice that often couples deterministically updating counters with probabilistically inserting new elements.

§ 2 introduces two data-plane algorithms, built on the fridge data structure, for generating unbiased delay distribution. Instead of counting each delay sample collected as one sample, the main idea is to think of each sample as a representative, which accounts for not only itself, but also other samples with the same delay that are not collected due to hash collisions. By keeping track of the number of insertions into the data structure, for each delay sample collected, the single-fridge algorithm computes the probability p that this sample is collected, and tallies a correction factor of $\frac{1}{p}$ as its count. Stacking multiple fridges together, our multi-fridge algorithm further improves the accuracy by computing a weighted average of the unbiased estimators from each fridge. To validate our algorithms, we build a prototype fridge implementation on high-speed programmable switches, which measure the unbiased delay distribution accurately, within the data-plane constraints.

§ 3 delves into the problem of monitoring TCP packet reordering in the data plane. First we show that identifying out-of-order heavy flows fundamentally requires a memory size linear in the number of flows. This lower bound result, together with the fact that routing is decided on the prefix level, steer us into designing two algorithms for identifying out-of-order heavy prefixes instead. In a crucial measurement study, we notice that a positive correlation exists between the reordering of a randomly chosen flow, and that of its prefix. Capitalizing on this correlation, the flow-sampling algorithm samples as many flows as possible, and infer the extent of reordering in prefixes using the flow snippets it captures. Further accuracy gains can be found by deploying a hybrid approach. Using an adapted counter-based heavy-hitter algorithm, we separately monitor the reordering of heavy flows over long periods of time, and only apply the flow-sampling algorithm on the rest of the flows. To show that the algorithms are lightweight enough for hardware resources, we provide a P4 prototype of the flow-sampling algorithm implemented using Lucid [45].

§ 4 is dedicated to the analysis of RAP. We show that, when the stream is highly skewed, RAP stores a constant number of heavy elements with constant probability. A major challenge in the analysis is to deal with the random process of inserting to a changing minimum counter with varying probabilities, simultaneously with the deterministic process of updating counter values. In an effort to decouple these processes, we consider the performance of RAP under three cases: (1) there are many steps where the smallest counter value increments regularly; (2) the total counter value at the end of the stream is large; and (3) Neither of (1) and (2) holds. We show that cases (1) and (2) are deterministically good, meaning that at least a large fraction of counter values is would be large at the end of the stream. Meanwhile, the conditions of case (3) induce contradiction with constant probability. Finally, we complete the argument showing large counters are likely to store heavy elements.

5.2 Future Directions

Measuring Unbiased Statistics for Network Queries The fridge design opened up many possibilities for measuring unbiased statistics for the "join-over-time" queries, where two or more packets across the traffic stream need to be joined together. It remains open whether our fridge design, or more generally, the idea of correcting for survivorship bias, is applicable to measuring statistics beyond the distribution of delays.

Leveraging Correlation in Performance Monitoring Notice that measuring reordering is fundamentally memory-intensive, yet we leverage the correlation of out-of-orderness among flows in the same prefix so that compact data structures can be effective. In fact, there is nothing special about out-of-orderness. Other properties of a network path could very well lead to similar correlation. For other performance

metrics that suffer from memory lower bounds, it would be intriguing to see whether such correlation helps in squeezing good performance out of limited memory.

Deriving Performance Guarantees for Counter-based Algorithms in Practice In Chapter 4, we have considered the performance of RAP in terms of how many heavy elements are stored in the data structure at the end of a stream. Empirically, RAP also shows a significant advantage over SpaceSaving [37] in approximating the frequency of heavy elements [6]. In our proof, we have shown that a constant fraction of the counter values are at least $\Omega(\varepsilon f)$ with constant probability. In fact, these counter values should be closer to f. It would be interesting to develop new techniques for deriving better bounds on the counter values.

More broadly, notice that RAP is a special case of the PRECISION algorithm [6], when the number of stages d is equal to the memory size m. Our currently analysis cannot be applied to PRECISION with a smaller d, as our argument on the smallest counter value in RAP does not apply for a minimum of d counter values in PRECISION. It remains to open to come up with a proof that gracefully generalizes to all number of stages d.

5.3 Final Remarks

In recent years, we observe a rigid dichotomy between the algorithms that the theory community considers, and the algorithms stemming from applied communities who work with hardware implementations. In practice, specialized hardware often comes with unique constraints beyond the memory size that the theory community mostly concerns about. Meanwhile, applied researcher come up with algorithms that empirically work well within those constraints, without understanding their theoretical guarantees. The discrepancy in the interests of the two communities are of course well-warranted. However, we do argue that, at least from the perspective of the net-

works community, having theoretical understanding of what we are running makes it easier to allocate the scarce hardware resources, and interpret the results we see. This dissertation marks our attempts to design practical algorithms in monitoring network performance, while providing forms of theoretical insight. It is our hope that works with a similar flavor could eventually gain traction, as more people turn to specialized hardware for performance gains.

Bibliography

[1] Imad Aad, Jean-Pierre Hubaux, and Edward W Knightly. Impact of denial of service attacks on ad hoc networks. *IEEE/ACM Transactions on Networking*, 16(4):791–802, 2008.

[2] Fatih Abut. A distributed measurement architecture for inferring TCP round-trip times through passive measurements. *Turkish Journal of Electrical Engineering & Computer Sciences*, 27(3):2106–2120, 2019.

[3] K Auerbach. Limitations of ICMP echo for network measurement. https://iwl.com/idocs/limitations-of-icmp-echo-for-network-measurement, 2004.

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

[5] Ran Ben Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Randomized admission policy for efficient top-k and frequency estimation. *arXiv preprint arXiv:1612.02962*, 2016.

[6] Ran Ben Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[7] Jon CR Bennett, Craig Partridge, and Nicholas Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on Networking*, 7(6):789–798, 1999.

[8] Ethan Blanton and Mark Allman. On making TCP more robust to packet reordering. *ACM SIGCOMM Computer Communication Review*, 32(1):20–30, 2002.

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Conference*, pages 99–110, 2013.

[10] Broadcom. Silicon innovations in programmable switch hardware, April 2020.

[11] CAIDA. The CAIDA UCSD anonymized Internet traces 2018 - December 20th. https://www.caida.org/data/passive/passive_dataset.xml, 2018.

[12] CAIDA. The CAIDA UCSD anonymized Internet traces 2019 - January 17th. https://www.caida.org/data/passive/passive_dataset.xml, 2019.

[13] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[14] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[15] Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, and Jennifer Rexford. Measuring TCP round-trip time in the data plane. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*, pages 35–41, 2020.

[16] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. dRMT: Disaggregated programmable switching. In *ACM SIGCOMM Conference*, pages 1–14, 2017.

[17] William G Cochran. The combination of estimates from different experiments. *Biometrics*, 10(1):101–129, 1954.

[18] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The Count-Min Sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[19] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[20] Graham Cormode and Shan Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *ACM Transactions on Database Systems (TODS)*, 30(1):249–278, 2005.

[21] Richard Cziva, Christopher Lorier, and Dimitrios P Pezaros. Ruru: High-speed, flow-level latency measurement and visualization of live internet traffic. In *ACM SIGCOMM Posters and Demos*, pages 46–47, 2017.

[22] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A scriptable high-speed packet generator. In *Proceedings of the Internet Measurement Conference*, pages 275–287, 2015.

[23] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of TCP. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 61–74, 2017.

[24] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*, volume 45, pages 139–152, 2015.

[25] Amir Herzberg and Haya Shulman. Stealth DoS attacks on secure channels. In *Network and Distributed System Symposium*, 2010.

[26] Intel. Tofino: P4-programmable Ethernet switch ASIC that delivers better performance at lower power. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html`.

[27] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. QPipe: Quantiles sketch fully in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, pages 285–291, 2019.

[28] Svante Janson. Tail bounds for sums of geometric and exponential variables. *Statistics & Probability Letters*, 135:1–6, 2018.

[29] Anura Jayasumana, N Piratla, T Banka, A Bare, and R Whitner. Improved packet reordering metrics, June 2008. RFC 5236.

[30] Hao Jiang and Constantinos Dovrolis. Passive estimation of TCP round-trip times. *ACM SIGCOMM Computer Communication Review*, 32(3):75–88, 2002.

[31] Akshay Kamath, Eric Price, and David P. Woodruff. A simple proof of a new set disjointness with applications to data streams. In *Computational Complexity Conference*, July 2021.

[32] Zohar Karnin, Kevin Lang, and Edo Liberty. Optimal quantile approximation in streams. In *IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 71–78, 2016.

[33] Michael Laor and Lior Gendel. The effect of packet reordering in a backbone link on application throughput. *IEEE Network*, 16(5):28–36, 2002.

[34] Ka-Cheong Leung, Victor OK Li, and Daiqin Yang. An overview of packet reordering in transmission control protocol (TCP): Problems, solutions, and challenges. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):522–535, 2007.

[35] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Symposium on Algorithmic Principles of Computer Systems*, pages 31–44. SIAM, 2020.

[36] Charles Masson, Jee E Rim, and Homin K Lee. DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment*, 12(12):2195–2205, 2019.

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

[38] Al Morton, Len Ciavattone, Gomathi Ramachandran, Stanislav Shalunov, and Jerry Perser. Packet reordering metrics, November 2006. RFC 4737.

[39] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, pages 85–98, 2017.

[40] Vern Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1997.

[41] Pensando. AMD Pensando Infrastructure Accelerators. `https://www.amd.com/en/accelerators/pensando`.

[42] Pensando. Smart Switches. `https://www.amd.com/system/files/documents/pensando-smartswitches.pdf`, 2022.

[43] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. Continuous in-network round-trip time monitoring. In *ACM SIGCOMM*, pages 473–485, 2022.

[44] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SIGCOMM Symposium on SDN Research*, pages 164–176, 2017.

[45] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *ACM SIGCOMM Conference*, pages 731–747, 2021.

[46] Peng Sun, Minlan Yu, Michael J Freedman, and Jennifer Rexford. Identifying performance bottlenecks in cdns through tcp-level monitoring. In *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack*, pages 49–54, 2011.

[47] Suricata. Suricata - eBPF and XDP. `https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html`, 2018.

[48] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In *USENIX Networked Systems Design and Implementation*, pages 599–614, 2019.

[49] The Linux Foundation. `https://www.dpdk.org/`.

[50] The P4 Language Consortium. $P4_{16}$ language specification. `https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html`, November 2018.

[51] Bryan Veal, Kang Li, and David Lowenthal. New methods for passive estimation of TCP round-trip times. In *International Workshop on Passive and Active Network Measurement*, pages 121–134. Springer, 2005.

[52] Verizon. IP latency statistics. `https://enterprise.verizon.com/terms/latency/`, 2021. Accessed: 2021-10-29.

[53] Verizon. Service level agreements. `http://www.verizonenterprise.com/solutions/public_sector/state_local/contracts/calnet3/sla/`, 2021. Accessed: 2021-10-29.

[54] Yi Wang, Guohan Lu, and Xing Li. A study of Internet packet reordering. In *International Conference on Information Networking*, pages 350–359. Springer, 2004.

[55] Xilinx. Adaptive infrastructure acceleration. `https://www.xilinx.com/applications/data-center/network-acceleration.html`.

[56] Yufei Zheng, Xiaoqi Chen, Mark Braverman, and Jennifer Rexford. Unbiased delay measurement in the data plane. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 15–30. SIAM, 2022.

[57] Yufei Zheng, Huacheng Yu, and Jennifer Rexford. Detecting tcp packet reordering in the data plane. *arXiv preprint arXiv:2301.00058*, 2022.