Telemetry for Next-Generation Networks

Queen Mary University of London

Submitted in fulfillment of the requirements of the Degree of Doctor of Philosophy

Jonatan Langlet

I, Jonatan Langlet, confirm that the research included within this dissertation is my own work or that where it has been carried out in collaboration with, or supported by others, that this is duly acknowledged below and my contribution indicated. Previously published material is also acknowledged below.

I attest that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge break any UK law, infringe any third party's copyright or other Intellectual Property Right, or contain any confidential material.

I accept that Queen Mary University of London has the right to use plagiarism detection software to check the electronic version of the dissertation.

I confirm that this dissertation has not been previously submitted for the award of a degree by this or any other university.

Jonatan Langlet

Date: July 7, 2024

Details of collaboration and publications:
Most of my research has been performed in closely collaborating teams. External contributions are explicitly stated as such throughout this work. Content within this dissertation has been published before in academic proceedings, authored or co-authored by me. Below is a list of these publications.

Langlet, J., Ben-Basat, R., Ramanathan, S., Oliaro, G., Mitzenmacher, M., Yu, M., & Antichi, G. (2021, November). Zero-CPU collection with direct telemetry access. In Proceedings of the 20th ACM Workshop on Hot Topics in Networks (pp. 108-115).

Langlet, J., Ben Basat, R., Oliaro, G., Mitzenmacher, M., Yu, M., & Antichi, G. (2023, September). Direct Telemetry Access. In Proceedings of the ACM SIGCOMM 2023 Conference (pp. 832-849).

Monterubbiano, A., Langlet, J., Walzer, S., Antichi, G., Reviriego, P., & Pontarelli, S. (2023). Lightweight Acquisition and Ranging of Flows in the Data Plane. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 7(3), 1-24.

Acknowledgements

First, my wholehearted thanks to my supervisor, *Dr. Gianni Antichi*, for your immense support, flexibility, and guidance, which helped me grow as a researcher. It was intense, and you were excellent.

A huge thank you to all my collaborators. I am especially grateful to *Prof. Michael Mitzenmacher* and *Dr. Ran Ben Basat* for your valuable insights and general advice.

Additionally, I am thankful to *Prof. Andreas Kassler*, who believed in me and guided me into systems research.

I would also like to express my gratitude to my defense committee: *Prof. Gareth Tyson* and *Prof. George Parisis*. You were not only immensely thorough but also provided very insightful comments and an engaging discussion.

Thank you *Zerina*, my loving and brilliant rock and support, who by now knows almost as much about this work as I do. You were always there for me, and I will always be there for you.

I am also deeply grateful to my entire family, particularly my parents, *Arnulf* and *Madeleine*. Tack för allt ni har gjort!

Finally, thank you *Cisco the dog*. I miss you so much.

Abstract

Software-defined networking enables tight integration between packet-processing hardware and centralized controllers, highlighting the importance of deep network insight for informed decision-making. Modern network telemetry aims to provide per-packet insights into networks, enabling significant optimizations and security enhancements. However, the increasing gap between network speeds and the stagnating performance of CPUs presents significant challenges to these efforts. Attempts to circumvent this slowdown by deploying monitoring functionality directly into the data plane, which is capable of line-rate processing, are hindered by the hardware's resource limitations and the data collection capacities of analysis servers.

This dissertation introduces a dual strategy to enhance centralized network insights: Firstly, it improves probabilistic network monitoring data structures, achieving fault-tolerant monitoring in heterogeneous environments with significantly higher accuracy and reduced resource demands. Secondly, it redesigns the interface between networking hardware and analysis servers to substantially lower telemetry collection and aggregation costs, thus enabling insights at unprecedented granularities. These advancements collectively mark a significant stride towards realizing the full potential of fine-grained network monitoring, offering a scalable and efficient solution to address the challenges brought by the rapid evolution of network technologies.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

AAE Average Absolute Error. 95, 100, 103, 105, 107, 108

ACL Access Control List. 68

ALU Arithmetic Logic Unit. 148, 149

ARE Average Relative Error. 95, 100, 103, 105, 107, 108

AS Autonomous System. 28, 112

ASIC Application Specific Integrated Circuit. 12, 14, 19–23, 26, 75–79, 81–83, 88, 89, 101, 141, 142, 148, 157, 162

BF Bloom Filter. 6, 26, 67, 82–88, 90, 91, 93, 94, 96–98, 100–104, 106, 107, 166, 168

BIOS Basic Input/Output System. 144

CAIDA Center for Applied Internet Data Analysis. 56, 61, 85, 86, 97, 101, 109

CMS Count-Min Sketch. 35, 67, 68, 82, 86, 88–91, 93–98, 100–109, 134

CoV Coefficient of Variation. 60, 61

CPU Central Processing Unit. 7, 12, 18, 20, 22, 28, 54, 93, 101, 108, 109, 113–118, 120, 121, 141, 142, 144–147, 151, 156, 157, 160, 161, 166, 168, 169

CRC Cyclic Redundancy Check. 47, 54, 96, 142, 143, 151

CS Count Sketch. 5, 33–35, 41, 42, 46, 53, 57, 64, 67, 68, 72, 167

DDoS Distributed Denial of Service. 16, 17

DiSketch Disaggregatable Sketch. 5, 6, 33, 34, 41–43, 46, 47, 49, 50, 54, 56–58, 60, 63, 64, 66, 71, 72, 76, 166, 167

DiSketch-Bi Load-aware DiSketch. 56, 57, 59, 60, 63

DiSketch-Uni Load-unaware DiSketch. 56–60, 63, 66

DMA Direct Memory Access. 158

DPDK Data Plane Development Kit. 117, 118, 158

DRAM Dynamic RAM. 28, 118, 161

DTA Direct Telemetry Access. 7, 47, 111, 114, 115, 120–123, 125, 126, 129, 132, 134, 140–149, 153, 155–161, 163, 165, 166, 168–170

ECMP Equal-Cost Multi-Path routing. 58

FlowLiDAR Flow Lightweight Detection and Ranging. 6, 27, 73, 75, 76, 79–84, 89, 94–97, 100, 102–108, 110, 165, 166, 168, 170

FP False Positive. 98, 101

FPGA Field-Programmable Gate Array. 29

FPR False Positive Rate. 94, 98, 103

HLL HyperLogLog. 67, 68

I/O Input/Output. 113, 117, 146

INT In-band Network Telemetry. 12, 15, 16, 111, 114–116, 118, 122, 126, 128, 130, 140, 145, 149, 150, 152, 162, 163, 169

INT-MD INT eMbed data. 15, 122

INT-MX INT eMbed instruct(X)ions. 15

INT-XD INT eXport Data. 15, 147

INT-XD/MX INT eXport Data/eMbed instruct(X)ions. 122, 128, 153

IP Internet Protocol. 56, 57, 61, 86, 122, 141

KI Key-Increment. 132, 134, 142, 159

KW Key-Write. 123, 125, 126, 128, 131, 132, 134, 136, 138, 140, 142, 145, 146, 149–152, 154, 159

MSE Mean Squared Error. 42, 61

MTU Maximum Transmission Unit. 12, 15, 16, 25

NACK Negative ACKnowledgment. 142

NIC Network Interface Card. 7, 29, 115, 120, 141, 142, 155, 158, 160, 161

NRMSE Normalized Root Mean Squared Error. 61–63

NUMA Non-Uniform Memory Access. 145

P4 Programming Protocol-independent Packet Processors. 22, 23, 54, 76, 96, 141, 142, 158, 169

PFC Priority Flow Control. 123

PINT Probabilistic INT. 122, 130

PISA Protocol Independent Switch Architecture. 6, 21, 25, 39, 76, 169

RAM Random Access Memory. 18, 22, 144, 157

RDMA Remote Direct Memory Access. 7, 27–29, 77, 111, 114, 115, 120, 121, 125, 126, 128, 129, 132, 134, 136, 140–144, 146–149, 151, 155, 158, 160–163, 166, 168, 169

RDMA-CM RDMA Communication Manager. 141, 143

RMSE Root Mean Squared Error. 61

RoCEv2 RDMA over Converged Ethernet. 121, 142

SDN Software-Defined Network. 10–12, 65

SLA Service Level Agreement. 9, 77

SmartNIC Smart Network Interface Card. 134, 158

SNMP Simple Network Management Protocol. 10

SRAM Static RAM. 4, 22, 35, 36, 96, 142, 143, 148, 162

SYN SYNchronization. 14, 160

TCP Transmission Control Protocol. 14, 47, 77, 116, 145

ToR Top of Rack. 114, 158

UDP User Datagram Protocol. 115, 121, 122, 126, 141, 147, 159, 161

Chapter 1

Introduction

This introductory chapter sets the stage for the dissertation by providing an overview of the research context and the primary focus areas of the work. The chapter outlines the critical importance of network telemetry in modern reactive networks and introduces the concept of the network control loop as a foundational mechanism. The subsequent sections will present the dissertation's specific scope, the key research objectives, a summary of major contributions, and the dissertation's overall structure.

1.1 Research Focus

This dissertation focuses on network telemetry, a crucial element for the effective operation and management of reactive networks.

Network telemetry is foundational to the behavior of reactive networks [18, 92, 105, 116, 206, 223, 229, 230]. This is characterized by the *network control loop* [175], which is depicted in Figure 1.1. The loop enables networks to dynamically adapt to changes and optimize their performance through a continuous cycle of four core actions:

(1) Measure: Network devices, such as switches, are tasked with monitoring traffic flow and their internal states [38, 116, 123]. This phase generates the telemetry data upon which network reactivity is predicated. Data

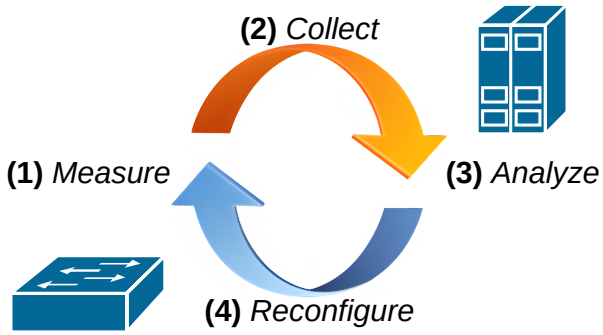


Figure 1.1: Network Control Loop

collected include metrics such as flow paths [18,116], queue depths [116], and packet losses [195,230], providing a detailed snapshot of network activity. Modern solutions can generate per-packet granular insights [103], resulting in *immense* data generation rates [123, 207].

(2) Collect: This phase involves transferring the telemetry data from network devices (e.g., switches) to queryable storage at centralized servers, establishing the foundation for a comprehensive network-wide view [36, 115, 149, 207]. Given the voluminous nature of the data produced by modern techniques, this phase can incur significant costs [115, 123, 207].

(3) Analyze: In this phase, data is aggregated and processed to construct a holistic view of the network's current state [96, 115, 171]. This step is crucial for identifying and understanding the root causes of detected or predicted issues. Analytical methods can vary greatly, from simple pre-defined rules [62] to advanced machine learning algorithms [96, 187], running either in real-time and/or on-demand [62].

(4) Reconfigure: The final step in the network control loop is reconfiguration, where insights gained from the analysis phase are used to adjust network configurations [134]. This may involve routine traffic engineering [134] to more advanced techniques such as altering packet processing pipelines within the switching hardware [216].

This iterative cycle of measurement, collection, analysis, and reconfiguration aims to continuously enhance the network's performance [134], reliability [76], and security [171]. By doing so, the network control loop ensures that the network can respond effectively to evolving conditions and demands, maintaining optimal operation at all times.

Making informed decisions requires deep insights into network operations, which in turn necessitate fine-grained telemetry [116, 231]. However, current fine-grained measurement and collection techniques are often impractical, both due to the high on-switch resource costs (step one) [92, 154, 227], and the immense load placed on centralized collection systems (step two) [115, 123, 207]. Analysis (step three) is tangential to the field of network telemetry and heavily relies on statistics and machine learning, fields that already possess a vast volume of literature and ongoing research [146, 157, 187]. Reconfiguration (step four) is triggered by telemetry-enabled applications, including appropriate traffic engineering or security decisions, and falls outside the scope of this dissertation.

My PhD research primarily focuses on the *measure* and *collect* stages of the control loop, as these two stages are foundational for constructing the data lake from which high-quality network insights are derived. The first two research chapters (Chapters 3 & 4) present novel methods to probabilistically record network measurements within network switches, providing highly resource-efficient techniques for higher-quality insights. The final research chapter (Chapter 5) presents a revolutionary approach for telemetry collection, enabling unprecedented telemetry ingestion rates and a more complete insight into the state of networks. Through this effort, I aim to enhance the adaptability of fine-grained telemetry, thereby providing tools to improve network security, performance, and reliability.

1.2 Research Objectives

My dissertation focuses on advancing the field of network telemetry by addressing key challenges in deployment, accuracy, and data collection. The main objectives are:

1. Make Sketches Network-wide Deployable Sketches are probabilistic data structures commonly used in network telemetry to estimate traffic statistics [23, 32, 45, 64, 91, 133, 137, 138, 221, 227]. However, they face practical deployment challenges, including determining optimal in-network locations and allocating sufficient resources for high-quality estimations [139, 222].
This objective aims to develop sketch disaggregation techniques that fragment these data structures and deploy them across *all* switches simultaneously. By doing this, we can increase estimation accuracy by leveraging network-wide resources and unlock additional benefits such as failure resilience. This research objective is the main research focus in Chapter 3.

2. Improve the Cost vs Accuracy Tradeoff in Sketches The estimation accuracy of sketches is directly affected by the amount of memory allocated to the data structure [221]. High-speed network switches depend on Static RAM (SRAM) to deliver statefulness, severely limiting the available memory for on-switch functions and leading to less reliable network insights due to inaccurate telemetry.
This objective aims to develop lightweight techniques to enhance the efficiency of sketch-based monitoring by decoupling flow ID storage from the data plane while maintaining a high and accurate coverage. This research objective is the main research focus in Chapter 4.

3. Alleviate the Telemetry Collection Bottleneck Telemetry collection is recognized as the primary bottleneck in fine-grained telemetry systems [191, 207], causing vendors to limit the amount of telemetry information generated and thus reducing network insight.
This objective aims to design and implement innovative techniques to bypass

current telemetry collection bottlenecks, significantly reducing operational costs and enhancing the comprehensiveness of network monitoring by addressing the challenges of voluminous data generation inherent in modern network monitoring systems. This research objective is the main research focus in Chapter 5.

1.3 Major Contributions

Building on the research objectives, this dissertation makes several key contributions to network telemetry, addressing challenges in deployment, accuracy, and data collection, to advance the state-of-the-art in network monitoring. To place the contributions in their proper context, I present the contributions within the framework of the previously introduced research objectives.

1.3.1 Network-wide Sketch Deployability

Chapter 3 presents my research on network-wide deployable sketches through novel disaggregation techniques. Key contributions include:

Introduced Spatiotemporal Disaggregation: I developed a method called *spatiotemporal disaggregation*, enabling sketches to be fragmented and deployed across multiple network switches in heterogeneous environments. This technique improves estimation accuracy by leveraging network-wide resources and offers benefits such as failure resilience.

Introduced DiSketch: I applied spatiotemporal disaggregation to a traditional Count Sketch (CS), creating DiSketch, a disaggregatable flow size estimator. DiSketch significantly reduces estimation errors by almost an order of magnitude compared to traditional aggregated sketches.

Evaluated in Data Center Networks: I demonstrated that spatiotemporal disaggregation is particularly effective in data center network environments, where resource constraints and traffic variability are common challenges.

Validated Practical Deployability: I validated the real-world deployability of DiSketch on hardware switches, ensuring that the proposed techniques are feasible for implementation in current network infrastructures.

1.3.2 Accuracy vs Cost of Sketches

Chapter 4 presents collaborative research on resource-efficient sketching, introducing novel sketch processing techniques to improve flow size estimation and flow-ID extraction efficiencies. Key contributions include:

Introduced FlowLiDAR: I introduced FlowLiDAR, a solution capable of tracking almost all network flows with modest data plane memory, independent of the flowID size.

Efficient FlowID Extraction Techniques: I introduced innovations such as lazy Bloom Filters (BFs) and differential flow detection, optimizing memory utilization and minimizing false positives. These techniques allowed FlowLiDAR to achieve higher accuracy and efficiency in flow monitoring compared to existing methods.

Evaluated FlowLiDAR in Real-world Scenarios: I evaluated FlowLiDAR using real traffic traces from ISPs, showing it can track 98.7% of flows, while other solutions only reconstructed up to 60% of flow statistics with the same memory usage.

Validated Practical Deployability: I implemented FlowLiDAR on hardware switches, validating its feasibility and practical deployability in current network infrastructures. The implementation proved FlowLiDAR is compatible with the high-throughput Protocol Independent Switch Architecture (PISA) and can be deployed in high-speed pipelined network switches.

1.3.3 High-Speed Telemetry Collection

Chapter 5 presents my research on high-speed telemetry collection, overcoming performance limitations of the data ingestion stack. Key contributions include:

Identified Underlying Collection Bottlenecks: I identified that the primary bottlenecks in telemetry data collection systems are CPU limitations and the rate of memory instructions. These constraints significantly hinder the collector's ability to process and store telemetry reports efficiently, inhibiting large-scale collection of fine-grained telemetry data.

Developed In-network Collection Techniques: I developed in-network telemetry interception techniques that bypass CPU involvement, utilizing Remote Direct Memory Access (RDMA) to write telemetry data directly from switches into collectors' memory. This approach facilitates CPU-less data ingestion and significantly reduces memory instruction rates.

Validated Practical Deployability: I implemented the proposed solution, Direct Telemetry Access (DTA), using commodity RDMA Network Interface Cards (NICs) and programmable switches, demonstrating its feasibility and practicality for deployment in real-world network environments.

Demonstrated Performance Increase: Through in-depth evaluation, I showed that DTA achieves orders-of-magnitude performance improvements, significantly surpassing state-of-the-art solutions in telemetry data collection rates.

Demonstrated Broad Support for Current Monitoring Systems: I proposed methods for integrating DTA with well-known telemetry solutions and demonstrated its integration with two prominent network telemetry systems, showing DTA's easy integration and adoptability within existing network monitoring frameworks.

1.4 Roadmap

The dissertation is structured as follows:

Chapter 2 provides a foundation in network telemetry, outlining current methods and use cases, as well as the importance and challenges of various techniques. It also presents related works within the context of this dissertation's research objectives.

Chapter 3 investigates sketch-based network monitoring and introduces *spatiotemporal disaggregation*, a highly flexible sketch deployment technique for disaggregated and heterogeneous sketching.

Chapter 4 introduces new techniques for highly accurate sketch-based monitoring and flow ID extraction under severe resource constraints.

Chapter 5 rethinks the telemetry collection stack and introduces a novel technique to overcome the current collection bottleneck, achieving immense performance improvements.

Chapter 6 concludes this dissertation by summarizing and discussing the research contributions and findings.

Chapter 2

Network Telemetry

In modern, reactive networks, the significance of network telemetry cannot be overstated. These networks can dynamically act and reconfigure based on the current state of the network, transforming it from a mere data forwarding mechanism into an intelligible and analyzable fabric. This lays the foundation for enhanced network transparency and control [194]. This evolution is pivotal for moving away from traditional, static network configurations towards dynamic, responsive infrastructures that swiftly adapt to changing conditions [93].

Network telemetry involves two primary processes: *monitoring* and *collection*. Monitoring entails the continuous on-switch observation of performance metrics and behaviors [116, 189] to identify trends [194], anomalies [69, 157], and potential issues [76]. Collection refers to the aggregation of this data from various sources across the network [115, 123, 223], enabling comprehensive analysis and informed decision-making based on a network-wide view [96, 187]. This insight serves as the backbone for critical functions in contemporary network environments, such as real-time network control [134], security monitoring [171, 187], compliance with Service Level Agreements (SLAs) [145], and general troubleshooting [76].

This chapter provides an overview of modern network telemetry, setting the stage for my dissertation.

2.1 Monitoring Techniques

In this section on monitoring techniques, I will highlight various methods used for observing and analyzing network traffic, ranging from traditional legacy solutions to more advanced techniques employed in Software-Defined Networks (SDNs), as overviewed in Table 2.1. I aim to provide a foundational understanding of the field of network monitoring, as well as insight into the drawbacks and challenges of the various techniques. While these categories are widely understood in the field, they do not have clear definitions, and some works may fall into multiple categories. However, these classifications allow for an abstract discussion without exploring the specifics of every individual work.

Category	**Description**
Active	Sends test traffic to measure network behavior (e.g., Ping and Traceroute)
Passive [38, 173, 211]	Extracts flow statistics from mirrored traffic
Mirroring [60, 174, 231]	Copies user packets for separate analysis
In-band [27, 63, 116, 167]	Embed measurements into user traffic
Sketching [32, 45, 91, 138, 221] [23, 64, 133, 137, 227]	Uses probabilistic structures for data summarization
Flow-based [92, 155, 230]	Aggregates measurements around network flows
Query-based [77, 79, 158, 159]	Extracts data based on pre-defined queries
Event-based [155, 230]	Reports data around detected events (e.g., packet losses)

Table 2.1: Overview of Network Monitoring Solutions

Legacy solutions are foundational solutions from before the advent of SDN and provide basic insights into the state of a network. For instance, traditional network monitoring tools like Simple Network Management Protocol (SNMP) and NetFlow were commonly used to gather and analyze network data. These methods have evolved, reflecting a balance between resource utilization, accuracy, and cost, with new solutions expanding the capabilities to track more detailed and fine-grained network performance metrics Legacy monitoring techniques are often categorized as either *active* or *passive*.

SDN centralized network control to controller servers, necessitating deeper network insight and telemetry to enable more complex decision-making. Following this development, programmable data planes [24] (see Section 2.4.1)

enabled more telemetry use cases and accelerated the development of advanced monitoring schemes. Modern SDN measurement systems can be categorized as *in-band*, *sketch-based*, *query-based*, *flow-based*, *event-based*, and *mirroring*.

2.1.1 Monitoring Categories

In this section, I will provide a brief explanation of the various monitoring categories presented in Table 2.1, delivering an overview of the state of network monitoring. Traditionally, legacy monitoring has been classified into active and passive categories. These classifications form the basis of our discussion, followed by an elaboration on more modern techniques. It is important to note that mirroring is a technique that spans both legacy and modern telemetry and is intriguing enough to warrant its discussion.

Active

Active monitoring, also known as probing, involves injecting synthetic traffic into the network to assess network behavior. The most common techniques are *ping* [76] and *traceroute* [1]. While these solutions are light on network resources, they have several drawbacks. Primarily, the forwarding path of these probes may not align exactly with that of user traffic, and may not accurately reflect the experience of actual user traffic. Additionally, the techniques are quite limited in design, failing to extract detailed information such as flow path tracing or stateful metrics.

Passive

Legacy passive monitoring systems analyze network traffic based on mirroring and proxy reporting, exemplified by NetFlow [38], sFlow [211], and IPFIX [173]. These measurement systems are constrained by switch performance and typically employ sampling techniques to reduce computational load, which in turn reduces measurement accuracy [15].

Mirroring

Mirroring is a technique often used for monitoring in both legacy and SDN networks by cloning user traffic for analysis. The primary distinction between legacy and SDN mirroring lies in the latter's ability to intelligently select which packets to analyze, enhancing efficiency and relevance [231] Mirroring is typically implemented in one of two ways: either by copying packets from the packet-forwarding Application Specific Integrated Circuit (ASIC) to the operating system running on the switch's CPU, or by forwarding a duplicate of the packet to a dedicated packet-analyzing server. Switches' packet-forwarding ASICs can process billions of packets per second, far exceeding the packet processing capabilities of CPUs, thus necessitating selective sampling and filtering techniques in both scenarios to manage the load [174, 231].

Despite its limitations in providing truly per-packet insights due to the need for sampling, mirroring offers significant advantages. Access to raw user traffic allows for complex and retrospective analysis based on future requirements, such as examining encapsulation headers or packet payloads at arbitrary packet depths. However, since these solutions are unable to monitor all user traffic continuously, they risk missing transient phenomena affecting only specific packets (see Section 2.3 for a detailed discussion on sampling).

Moreover, the efficiency of software-based raw packet analysis is inherently lower compared to in-hardware analysis and aggregation techniques [210].

In-band

At the other extreme, in-band monitoring systems embed measurements into the user packets themselves during transit, providing unprecedented per-packet granular insight into the network state. The state-of-the-art in in-band monitoring is In-band Network Telemetry (INT) [116], which is further elaborated in Section 2.1.2. Numerous competing alternatives have arisen [18, 27, 63, 167]. However, these are accompanied by several challenges and limitations, most notably the immense load they place on the underlying collection systems [194, 205] and a reduction of the Maximum Transmission Unit (MTU) due to appending data into user packets [18, 194].

Sketches

Sketches are probabilistic data structures that employ hashing techniques to provide estimated statistics about data streams [78]. Known for their excellent compatibility with hardware and typically O(1) insertion logic, sketches are widely utilized in network monitoring [23, 32, 45, 64, 91, 133, 137, 138, 221, 227]. They can estimate a broad range of streaming data statistics without modifying user packets[1]. However, the accuracy of these estimations is directly influenced by the allocated memory for sketching, with the accuracy/cost tradeoff being an active area of research [91, 154, 183]. Further elaboration on sketches and a discussion of specific use cases are provided in Section 2.1.3.

Flow-based Monitoring

Flow-based monitoring aggregates measurements around the flow 5-tuple directly on switches, significantly reducing data collection load by exporting only summarized per-flow statistics [92, 155, 230]. This approach offers substantial efficiency gains, potentially lowering collection costs by orders of magnitude [230]. However, it necessitates additional on-switch resources for storing and processing these statistics [92, 230], leading to increased use of switching resources and delays in reporting times. Such delays can impact the system's responsiveness, especially in scenarios requiring immediate data analysis [134]. Moreover, by prioritizing aggregation, there is a risk of overlooking transient phenomena or anomalies that affect only a minimal number of packets, as the granularity of insight is diminished [231]. Thus, flow-based monitoring represents a compromise, balancing between the comprehensive visibility offered by per-packet monitoring [116] and the scalability of traditional flow-level sampling [211], albeit at the cost of increased on-switch complexity.

[1]Sketches themselves do not modify user packets. However, there are sketch collection techniques, most notably LightGuardian [227], that use in-band techniques to transfer sketches to central collection using user packets.

Query-based Monitoring

Query-based monitoring offers a tailored approach to network telemetry, specifically designed to mitigate the overload on collection systems caused by fine-grained monitoring [77, 158, 159]. By collecting only data pertinent to *predefined queries*, such as the number of established TCP connections per host for detecting TCP SYN flooding attacks [77], it ensures efficiency and relevance in data collection. This method aligns closely with the needs of modern network environments, enabling precise monitoring for specific scenarios like security breaches or performance issues.

Despite its targeted effectiveness, query-based monitoring faces challenges with scalability and retrospective analysis. The complexity of in-network aggregators required to support an expanding array of queries grows in scale, leading to increased in-ASIC resource consumption [77, 159]. Moreover, its inability to examine characteristics not specified by existing queries limits its comprehensive insight into network behavior [86, 118, pg.54]. However, recognizing its value in reducing telemetry collection load and its capability in identifying critical pre-defined network events, I have ensured support for query-based monitoring in my collection research in Chapter 5.

Event-based Monitoring

Event-based monitoring can sometimes be viewed as a sub-category of query-based monitoring, in that it only reports information that is pre-defined as being of interest [69, 230]. In this approach, traffic statistics are maintained on the switch, which is used to detect and report anomalous network behaviors, such as packet losses or latency spikes. This reduced collection load has similar trade-offs to the aforementioned query-based systems, including a narrow and predefined view of the network, limited retrospective analysis capabilities, and increased on-switch complexities and resource usage.

2.1.2 In-band Network Telemetry (INT)

INT [74] is an in-band monitoring scheme that was jointly proposed by Barefoot, Arista, Dell, Intel, and VMware in 2015. Since its introduction, INT has emerged as the state of the art for fine-grained network telemetry, offering unparalleled visibility into the network state. Several commodity fixed-function switches now offer support for this monitoring technology [35,164,200]. INT operates in three distinct modes, each tailored to meet various monitoring requirements and mitigate the trade-offs between the depth of metadata and the impact on the network.

INT eMbed data (INT-MD): This mode, often referred to as the "classic" mode, involves embedding both the INT instructions and measurements directly into user packets. At each hop along the path, the instruction header is read, and local measurements are inserted into the packet. At the last hop, known as the INT *sink*, all instructions and measurements are stripped from the packets before they are sent for central collection, while the (now unmodified) user packet continues to its destination. While this mode offers the most comprehensive telemetry data, it also imposes the greatest modification on the packets, potentially reducing the available MTU.

INT eMbed instruct(X)ions (INT-MX): This mode embeds INT instructions within packets but does not embed the per-hop measurements. Instead, as the packet traverses the network, switches read the packet-carried instructions and export their local measurements directly to centralized collection systems. The INT sink then removes the instruction header before forwarding the packet to its destination. This method limits packet modification to the instruction header only, preventing any additional increase in packet size regardless of the number of nodes transited (aside from the fixed-size instruction header).

INT eXport Data (INT-XD): Known as the *postcard* mode, this method enables INT nodes to export metadata directly from their data plane to the monitoring system, based on pre-configured INT instructions. This mode requires no packet modification, ensuring minimal impact on the data flow.

Despite its various iterations, INT faces challenges, particularly in managing the high volume of telemetry data generated by per-packet measurements, which may overwhelm collection systems [123,124]. This necessitates selective monitoring or the integration of event detection systems to efficiently handle the data [207,209]. Moreover, embedding telemetry data in packets can reduce the available MTU [18], effectively limiting the system's goodput and leading to degraded network performance.

2.1.3 Sketching-based Monitoring

Sketches, as probabilistic data structures, have found a niche in network telemetry for their space efficiency and capability to approximate streaming data monitoring. Their ability to deliver highly useful traffic measurements has significantly contributed to their popularity. By embedding sketches directly within the network switches themselves, full-coverage, non-sampled measurements of network traffic are realized. This approach ensures every user packet is accounted for in time-window measurements, with only the aggregated data periodically relayed to centralized collection systems.

The research community has recognized the value of sketches and has proposed a variety of sketching techniques to estimate a broad set of streaming data statistics. A significant portion of my PhD research has focused on sketch-based monitoring, with a primary emphasis on a fundamental metric: *frequency estimation*.

Frequency Estimation

Frequency estimation is a fundamental application of sketches in network telemetry, achievable through various sketching techniques [32, 45, 139, 154, 196, 221, 227]. A common use case involves counting the number of packets or bytes in a network flow. As a foundational metric, frequency estimation is instrumental across numerous operational contexts, including investigating congestion issues [221], making informed offloading decisions [136], Distributed Denial of Service (DDoS) detection [196, 221], and even as a step in packet loss detection [227].

Frequency estimation provides a list of frequencies for all flows, termed a *frequency vector*. The extracted frequency vector can be used to calculate arbitrary frequency norms, which are useful in diverse statistical analyses. One such use case is in *entropy estimation*, which aids various network measurement applications including DDoS detection [156], load balancing [143], and traffic classification [220].

2.2 Telemetry Collection

Telemetry Collection plays a pivotal role in network management by centralizing the vast array of measurements generated across the network [12, 34, 77, 93, 105, 115, 163]. This process begins when network devices generate measurements, compiling them into detailed telemetry reports. These reports are then transmitted to centralized telemetry collectors, where the incoming data undergoes aggregation and is meticulously organized into various data structures. This structured organization is critical for ensuring the data remains queryable with high efficiency.

Centralization of telemetry data serves as the backbone for comprehensive network oversight, enabling a unified network-wide perspective. Such a centralized viewpoint is instrumental in guiding control strategies [5, 85, 134], making informed decisions, and facilitating thorough troubleshooting [76, 105, 193]. Analysis typically supports both real-time operations and retrospective analyses, allowing network operators to respond to immediate issues and review historical data for long-term planning, troubleshooting, and optimization.

2.2.1 Report Transmission towards Collectors

Before discussing collection approaches, it is essential to understand how telemetry data is transmitted to central collection servers.

Telemetry data transmission to central collection systems can adopt various methodologies, each with its own performance characteristics and impacts on the telemetry framework:

In-band Transmission In this approach, telemetry data is embedded within user packets [116, 227]. End-hosts may undertake initial aggregation and pre-processing tasks before dispatching this data to central collection systems [92, 227]. This method distributes the processing burden across multiple servers, aiding scalability. Despite its efficiency, it necessitates substantial infrastructure modifications, making it viable primarily within data center networks where end-hosts reside under a unified domain [92]. Further, the requisite pre-processing can introduce delays in central system reactivity.

Pushing Approach Also known as *streaming telemetry*, this method involves sending telemetry reports directly to the central collection upon their generation in the data plane [74], facilitating rapid responses based on streaming telemetry. While this approach ensures quick reactivity, the continuous export of numerous small reports demands highly capable collection systems to manage the influx efficiently [123]. This approach, coupled with sampling or selection techniques, is commonly used in industry [34, 163].

Polling Approach Contrary to automatic data push methods, polling involves retaining telemetry data on the switches, allowing central collectors to request this information as needed. This method is resource-light on the collection side but can result in significant delays and reduced system reactivity [114]. Additionally, there's a risk of missing critical real-time events due to the on-demand nature of data retrieval [114]. Moreover, on-switch storage imposes a load on internal switch buses to move measurements into aggregation in the slower CPU-adjacent RAM [230]. The industry is moving away from polling due to the poor insight these techniques deliver [34, 163].

On-switch Aggregation In this approach, telemetry data is temporarily stored and possibly pre-processed on the switches themselves before being sent to central collection [159, 230]. Techniques such as deduplication and event detection can be applied locally, reducing the data volume

sent to the collection and thereby enhancing efficiency [230]. While this method can significantly decrease the collection load, it may introduce delays in central analysis and demand additional in-network resources, including increased memory and processing load on switches [230].

2.2.2 Collection Stacks

Once reports have reached a collector, they will enter the *collection stack*. Network telemetry collection consists of three main steps: *receive*, *store*, and *present* [34], which together form a complete collection stack.

Receive forms the interface between the generated telemetry data and the collection software. This step is responsible for ingesting incoming reports, transforming, and filtering this data to make it more manageable further down the collection stack.

Store takes the pre-processed telemetry data and stores it in queryable data structures. Time-series databases are typically well-suited for streaming telemetry collection and are commonly used [36, 66].

Present analyzes and processes the aggregated measurements, delivering actionable outputs including visualizations and alert generation.

A notable telemetry collection solution is the *TIG stack*, which is supported by large industrial vendors including Cisco [34] and Huawei [95]. In this setup, Telegraf [99] is used to receive telemetry data, which is then stored in InfluxDB [98], and visualized or alerted upon using Grafana [121]. The TIG stack is highly versatile and supports a wide range of incoming data types, while performing basic data transformation and filtering to optimize aggregation. Although TIG is common, competing solutions include the ELK stack [57], Tetration [40], and NPM [188].

Unfortunately, all of these solutions are software-based and struggle to scale alongside the increasing capacity of purpose-built ASICs [182], which enable high-speed packet forwarding [39, 162]. As previously mentioned, modern network telemetry can generate an immense amount of data by

leveraging these ASICs, and operators need to weigh the level of insight against the cost of deploying powerful collection clusters [115, 123, 207].

2.3 Telemetry Sampling

An underlying assumption of this dissertation is that offline processing of telemetry data is too costly to be feasible; otherwise, we could simply route every user packet through a CPU and perform any arbitrary analysis in real-time. This approach is unfortunately not economically viable in large-scale deployments due to the gap between network and CPU speeds. Legacy solutions address this issue by sampling or selecting what to monitor based on pre-defined rules. For example, a monitoring system might mirror every x^{th} packet to a CPU, which then extracts relevant information from the sampled traffic, such as flow identifiers or packet latencies. The scalability issue of collection might be similarly addressed, where telemetry reports contain only sampled information to reduce the generated telemetry load to manageable levels.

Telemetry data sampling can indeed be effective in certain cases, such as when only approximated aggregate states are required. Examples include determining the average data corruption rates in specific network sectors, assessing the packet loss rates of particular switches, or generally inferring statistical distributions about the aggregate network traffic.

However, data sampling fails to reliably answer more tailored queries that are not solely focused on the aggregate network state. For example, queries concerning individual network flows cannot be reliably addressed based on randomly sampled measurements. These queries are crucial for enabling both intricate automated control and more detailed manual insight.

Consider the straightforward scenario where two traffic endpoints experience poor communication performance. Troubleshooting these flows would first require identifying the network path of this communication, likely followed by more tailored queries such as queue mapping inside specific switches or issues related to pipeline traversal and rule matching at the time the

issue occurred. These queries rely on retrospection, which assumes that the requested data has already been collected. However, sampling, by its very nature, cannot guarantee the availability of this information at query time.

Recent research has tailored the sampling of on-switch measuring programs to predefined queries [77, 159]. However, query-based sampling necessitates runtime reprogrammability, an area still under active research and currently unfeasible in high-throughput PISA switches due to their compile-time allocation of resources and computational logic [61, 218]. Additionally, non-reprogrammable solutions impose a significant resource burden on the switch, limiting the complexity of the queries [77]. Moreover, these query-based measuring solutions fail to capture the necessary measurements post-fact if a relevant query was not predefined; thus, sporadic issues could be challenging to troubleshoot if one does not already know which traffic profiles should be monitored. In short, queries about historical events are not easily answered through these systems.

I would therefore argue that while sampling solutions play an important role, they have not yet reached a state where they alone can realize the potential of highly reactive networks envisioned by the networking community. To achieve this, we need to either improve the efficiency of in-band monitoring or enhance the performance of the collection and analysis stacks.

2.4 Switching Hardware

To understand the rationale behind advocating for in-network measuring, preprocessing, and aggregation, it is essential to grasp the fundamentals of switching hardware.

Modern high-speed network switches can process billions of packets per second [198, 202]. This exceptional throughput is achieved through the use of ASICs, which perform basic operations often built around lookup tables to maximize packet throughput [24, 147, 160, 202]. These ASICs are highly specialized for packet processing, sacrificing general computability for speed and efficiency.

Per-packet statefulness and buffering typically rely on low-capacity Static RAM (SRAM) due to its low latency, which significantly limits the available memory in the forwarding plane [102, 164, 198, 202][2].

This limitation in computational complexity and memory is partially mitigated by integrating traditional RAM and a general CPU on the switching board. These components handle control, basic telemetry, and various offline functionalities that do not need to be triggered at line rates [39]. However, the generic CPU is too slow to manage the rate of packet processing and primarily serves as a support system, while the much faster ASIC focuses on per-packet logic [39].

2.4.1 Programmable Switches

Recent advancements in network programmability have been primarily driven by the development of programmable switching ASICs that allow for custom packet forwarding logic [25, 26, 162].

This evolution can be traced back to OpenFlow [147], which marked a substantial shift by enabling the control plane to dynamically update the forwarding rules of network switches. Despite this innovation, OpenFlow's reliance on fixed-function forwarding logic embedded into the hardware by manufacturers placed inherent limitations on network operators, confining them to predefined protocols. Over time, the number of supported protocols has grown, increasing hardware complexity while still not providing support for custom protocols [24].

The constraints of OpenFlow drove the development of fully programmable data planes, a revolution led by the introduction of Programming Protocol-independent Packet Processors (P4) [24]. P4 empowers network operators with unprecedented control, allowing for the creation of custom packet processing logic and tailor-made protocols to suit specific network needs. This newfound flexibility has paved the way for a plethora of advanced network functions, from custom routing decisions [14,33,148] and sophisticated firewall

[2]The amount of memory available for forwarding logic, including in-ASIC telemetry functionality, is often as low as O(10MB).

implementations [53, 224] to in-network acceleration [186, 204] and extremely fine-grained network monitoring techniques [18, 116].

Programmability Limitations

Despite these advances, programmable switches face intrinsic limitations in the complexity and number of permissible per-packet operations to maintain high packet throughput [44]. These constraints underscore the ongoing challenge within network design: balancing the demand for advanced, customizable network functions against the operational imperatives of maintaining speed and efficiency in packet processing.

To give an overview, consider the case of P4-programmable switching ASICs [162] where programmability is restricted by several factors:

No computational loops Switches operate as feed-forward pipelines and do not support traditional computational loops.

Basic mathematical operations Mathematical operations are limited to basic arithmetic (+, -), without support for more complex operators such as arbitrary multiplication or division[3].

No floating-point operations There is no support for floating-point operations, forcing developers to design algorithms around pure integers and bitstrings.

Limited length of logical dependency chains The number of dependent steps, where each step relies on the output of a previous step, is limited[4]. This severely limits the complexity of on-switch functions [17].

Limited memory access There are significant restrictions on how memory can be accessed and processed, stemming from the feed-forward nature of the switching architecture[5].

[3]Multiplication and division by powers of two can sometimes be achieved through bit-shifting.

[4]The exact length of a dependency chain is confidential.

[5]The exact memory access limitations are confidential.

Navigating these limitations to design effective algorithms remains a persistent challenge for networking researchers and developers [17, 48, 88, 109, 117].

All algorithms and techniques presented in this dissertation are developed with these hardware restrictions in mind and are validated through hardware prototypes.

2.5 Related Work

Building upon the background provided, this section places my dissertation's contributions within the broader context of existing literature. By examining related works within my research objectives, we can better understand the advancements made, identify the remaining gaps, and see how my research addresses these gaps.

2.5.1 Network-wide Deployable Sketches

In Chapter 3, I design a novel solution for sketch disaggregation in heterogeneous environments. In this approach, fragments operate autonomously with reasonable knowledge assumptions for participating switches. This section explores works related to sketch disaggregation.

Traditionally, sketches have been deployed monolithically [32, 45, 139]. Recent interest, however, has shifted towards network-wide deployments to enhance measurement flexibility [29, 47, 75, 129, 197, 222, 227]. For example, LightGuardian [227], a sketch-based network-wide telemetry system published in 2021, demonstrates these benefits. In their approach, each switch hosts two *SuMax* sketches: one actively populated and one being collected. This supports new sketch measurements, including latency jitter and packet loss detection, using a probabilistic in-band collection method to reduce centralized collection costs. Nonetheless, this system does not address heterogeneous environments, incurs substantial resource costs, and employs monolithic (i.e., non-disaggregated) sketches on each switch.

Sketch disaggregation has recently gained traction. DISCO [29] pioneered this approach in 2020, advocating for sketch disaggregation to ease deployments in resource-scarce environments. It introduced a method for per-row disaggregation of sketches to enhance flow size estimation and heavy hitter detection accuracy. However, it did not explore applications in heterogeneous environments, hardware viability, or complex sketches beyond flow size estimation.

Further research by Cornacchia et al. [47] highlighted the detrimental effects of traffic patterns on per-row disaggregated sketches, notably increased hash collisions and accuracy degradation due to load imbalances. They proposed that sketch fragments sample a subset of traffic to process, but their algorithm assumes full in-band knowledge of packet paths and fragment dimensions, thus introducing considerable overheads and limiting deployment flexibility.

In 2023, Gu et al. [75] proposed per-column disaggregation as an alternative and discussed how to handle load imbalances. They proposed using per-flow lookup tables to achieve proportional counter allocation between per-path hops, which unfortunately imposes severe memory overheads. Further, as discussed in Chapter 3, the per-column disaggregation approach is inefficient in high-performance switching architectures like Protocol Independent Switch Architecture (PISA) leading to high computational footprints.

Li et al.'s 2024 preprint [129] addresses the traffic imbalance issue by proposing a deployment and incrementation strategy that ensures load balancing across sketch rows. Their method selectively deploys rows across the network, supporting a variable number of rows per fragment. The ingress switch determines the number of rows each hop should process per packet, inserting this information as a new header for ingressing packets and allocating traffic based on hop capacities. However, this places high burdens on ingress switches, requiring extensive knowledge of network paths and fragment dimensions, and reduces the MTU of the network through extended packet headers, risking frame fragmentation and reduced goodput.

Chapter 3 introduces my solution, spatiotemporal disaggregation, which is the first technique allowing fragments to function autonomously while

mitigating the impact of heterogeneity on estimation accuracies.

2.5.2 Memory-efficient Sketching

In Chapter 4, I focus on memory-efficient sketching by decoupling flow identifiers from the switching ASIC. My solution eliminates the need for on-switch storage of identifiers and enables more efficient sketching by excluding short flows from the counting mechanism while preserving their queryability. This section examines works related to this solution

Achieving comprehensive flow coverage is a common goal in network telemetry. Per-flow measurements must be aggregated within the packet-forwarding data plane hardware to support high packet rates, employing techniques such as sketches. However, retrieving and utilizing these measurements also necessitates knowledge of the flow identifiers (e.g., 5-tuples) under which the measurements are aggregated.

Current solutions such as FlowRadar [133], TurboFlow [190], FlowMap [213], and Hashflow [228] employ in-switch storage of flow identifiers, which incurs significant memory costs. This section examines these approaches and their limitations

FlowRadar [133] utilizes a Bloom Filter (BF) to detect new flows and stores flow identifiers and counters in an Invertible Bloom Lookup Table (IBLT) [172]. The IBLT contents are periodically sent to the controller for inversion to extract flows. However, FlowRadar requires approximately 20% extra memory to ensure IBLT invertibility and maintain a low false positive rate in the BF. Additionally, its memory requirements depend on the flowID size, and its accuracy significantly declines if the number of flows exceeds the IBLT's invertibility threshold.

FlowMap [213] improves upon FlowRadar by replacing the IBLT with an independent hash table for flow identifiers and a separate two-level hash table for counters. The extraction process, implemented as a linear programming problem, reduces memory overhead but is computationally expensive. To mitigate this, counters are divided into groups, enabling faster extraction by solving smaller linear programming problems. However, as with the others,

FlowMap requires in-switch memory for flow identifier storage and lacks demonstrated hardware compatibility for high-speed architectures.

TurboFlow [190] and HashFlow [228] similarly use hash tables to store flow identifiers and associated counters. TurboFlow handles hash table collisions by evicting flows to the controller, while HashFlow uses two tables, placing colliding elements from the first table into the second. Collisions in the second table result in the eviction of the flow with fewer packets.

There are a few works that decouple flowID storage from the data plane. NZE sketch [91] and PR-sketch [183] are two such works, relocating flowID storage to the control plane. NZE splits traffic into elephants, stored directly in a hash table, and mice, with flowIDs stored in the control plane and counters handled by a standard sketch. This method requires careful memory allocation between the hash table and sketch, and its sensitivity to flowID size limits the number of storable keys. PR-sketch, although it sends all flowIDs to the control plane, still contains on-switch algorithm inefficiencies that impact accuracy, along with a lengthy offline analysis.

Chapter 4 introduces Flow Lightweight Detection and Ranging (FlowLiDAR), a novel approach that continuously sends flowIDs to the control plane, significantly reducing data plane memory requirements. This approach also includes a powerful flow identification extraction algorithm that minimizes the number of on-switch counters needed for frequency estimation.

2.5.3 Telemetry Collection Performance

In Chapter 5, I identify the current collection bottleneck and redesign the collection stack to overcome this bottleneck using in-network indexing and data structure population via Remote Direct Memory Access (RDMA). This section explores works related to high-speed telemetry collection and in-network RDMA generation.

Collection is recognized as the primary bottleneck in fine-grained network-wide telemetry. Prior works have focused on enhancing the performance of collectors' stacks [115, 207] and reducing the load through offloaded pre-processing [131] and in-network filtering [103, 120, 209, 230].

For instance, INTCollector [207] augments the on-server collection stack by introducing a kernel-space filtering mechanism to lighten the load on the indexing and storage units, resulting in minor performance enhancements. However, this design still faces inherent performance limitations due to kernel-space's reliance on the CPU. Vestin et al. advanced this by shifting the filtration process to the network card [209], further improving the collection performance. Unfortunately, these methods depend on identifying statistical outliers and events to prevent redundant data entries into the collector. Consequently, they do not improve collection performance when all reported data should be stored, as in cases where filtering is already implemented at the telemetry reporters for event-based or query-based telemetry.

Confluo [115] offers an entirely new telemetry collection stack for high-speed networks, introducing the Atomic MultiLog data structure for highly concurrent operations and versatile monitoring and query capabilities. Although Confluo significantly increases collection performance, it struggles with fundamental performance limitations and scaling to collection rates beyond 10Gbps due to its software nature.

Another strategy involves engaging end-hosts in network-wide telemetry [92, 193], i.e., hosts on both ends of monitored network flows, distributing pre-processing costs across the network. However, this requires significant investments, infrastructure changes, and assumes that end-hosts are willing and able to assist in collection, which is not always guaranteed, such as in Autonomous Systems (ASs).

Given the seemingly fundamental bottleneck of centralized collection, most research has concentrated on reducing telemetry volume through event-based telemetry, query-based telemetry, sketches, and sampling, as covered earlier in Section 2.1.

Switch-to-Server RDMA

Recent works have demonstrated that switches can generate RDMA instructions to access server Dynamic RAM (DRAM) for expanded memory in their stateful network functions [117, 178]. These works are interesting for

specific scenarios but are unsuitable for the queryable aggregation required for telemetry collection, which necessitates new in-network indexing algorithms not supported by commodity RDMA.

Programmable Network Interface Cards (NICs) have been shown to expand upon RDMA with new and customized memory verbs [9], potentially leading to new RDMA verbs capable of populating queryable structures suitable for telemetry. Field-Programmable Gate Array (FPGA) network cards, in particular, show promise for high-speed custom RDMA verbs [144, 184] However, these solutions have not been adopted by the industry, and prototypes fail to deliver competitive performance. Nonetheless, the expansion of RDMA capabilities on NICs remains a promising area for future research in telemetry collection stacks.

Chapter 3

Sketch Disaggregation

This chapter explores the deployment of sketching data structures and develops a functional solution for disaggregated, network-wide monitoring. This investigation is a crucial step towards achieving the first research objective, "Make Sketches Network-wide Deployable", by introducing *spatiotemporal disaggregation*. This technique enables the fragmentation and deployment of sketches across multiple network switches, leveraging network-wide resources to enhance estimation accuracy, provide failure resilience, and reduce sketch extraction delays.

The chapter addresses the practical deployment challenges of sketch-based monitoring in data center networks. By developing a method capable of disaggregation in heterogeneous environments, the aim is to achieve high levels of accuracy and efficiency, even in volatile real-world conditions. This method ensures reliable measurements for effective reactive network operation.

Attributions This chapter exclusively contains my contributions.

3.1 Introduction

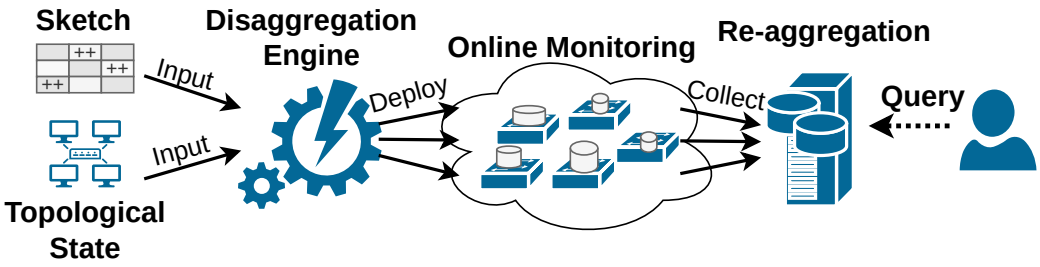


Figure 3.1: Overview of Sketch Disaggregation.

A fundamental challenge for on-switch network monitoring solutions is that switches (programmable or otherwise) generally have little available memory and limited computation abilities [25, 26, 162]. On top of that, there are many different use cases for in-network computing, including security [140, 217, 219], machine learning aggregation [125, 130, 177], storage for database systems [128, 136, 204], and various network functionalities [84, 134, 148, 168]. Unfortunately, these many use cases exacerbate the limitations of switches, as the already minimal resources available need to be shared across multiple functions, degrading the accuracy of deploying streaming analytics placed alongside them.

This chapter focuses on the telemetry problem of tracking flow statistics compactly. Many previous works use sketch data structures for this purpose, focusing on minimizing the sketch size while optimizing the size-accuracy tradeoff [32, 45, 139, 227]. However, as I show in Section 3.2, it remains hard to fit sketches with the tight constraints of switches, especially when multiple functionalities (e.g., security and machine learning) are enabled at the same time. A key observation is that in networks, the same packet usually traverses multiple nodes (e.g., switches) and we can leverage *residual resources* across these nodes to improve the accuracy, despite having different packets going through different switches and different switches having different resources available. While I focus on network telemetry in this dissertation, this approach presented in this chapter is applicable to similar settings that

naturally appear in distributed databases, such as where the search for a key traverses multiple nodes in a distributed index with different searches taking different routes (e.g., a B-tree), and we can therefore harness resources along the network path [3, 212].

This chapter describes a distributed sketching approach that disaggregates the sketches into *fragments* over multiple nodes (Figure 3.1). Given a query time interval, each node fragment can provide an estimate, and my methods can produce a single accurate estimate from the ensemble of estimates from the nodes along a flow's path. While this sounds simple, note that there are numerous challenges that we need to handle in terms of inequalities in sketch accuracy due to heterogeneity: (1) different nodes have different resource availabilities (2) different nodes see different amounts of traffic (3) different flows have different path lengths (so flows have different numbers of fragments to aggregate). Additionally, we must manage the fact that the switches have very limited computation capabilities, where even operations such as multiplication and division may not be supported, and due to the fast line rates, minimal computation can be performed for each packet.

I designed new techniques for disaggregating sketches over multiple nodes (Section 3.4). The main innovation of this approach is to make query windows divisible, where sketch fragments on individual nodes consist of smaller subepochs, which can be re-combined to answer queries over time intervals. The division into smaller subepochs allows us to tailor epochs to different node resources and traffic while allowing for them to be combined effectively across time and space.

I apply this technique on Count Sketch to build Disaggregatable Sketch (DiSketch): a disaggregated flow size estimator for heterogeneous environments, and then describe how to implement it on hardware switches (Section 3.5). I compare DiSketch against traditional aggregated sketch deployments, as well as versus SuMax - a recent sketch natively built for a network-wide deployment (Section 3.6).

The main contributions in this chapter are:

- I develop *spatiotemporal disaggregation*, which allows sketches to be fragmented and deployed across multiple network switches in heterogeneous environments. This technique leverages network-wide resources to improve estimation accuracy and provides additional benefits, such as failure resilience.
- I apply spatiotemporal disaggregation to a traditional Count Sketch (CS), creating DiSketch, a disaggregated flow size estimator. DiSketch significantly reduces estimation errors by nearly an order of magnitude compared to traditional aggregated sketches.
- I demonstrate the effectiveness of spatiotemporal disaggregation in data center network environments through an evaluation and comparison against traditional sketches.
- I validate the real-world deployability of DiSketch on hardware switches, confirming that the proposed techniques are feasible for implementation in current network infrastructures.

Application	Examples	Memory
Basic Packet Processing	switch.p4 [152]	30%
Security	Ripple [219], Jaqen [140], Bedrock [217]	+10-50%
Machine Learning	SwitchML [177], ATP [125], THC [130]	+10-40%
Storage/Database	DistCache [136], NETACCEL [128], Cheetah [204]	+20-30%
Networking	SilkRoad [148], HPCC [134], SwRL [84], Sailfish [168]	+5-40%

Table 3.1: The on-switch memory cost of network functions.

3.2 Motivation

Recent advancements in streaming analytics have led to the development of compact sketch-based solutions, as described in several works [89, 90, 91, 139, 154, 214, 227]. These studies have demonstrated the feasibility of implementing these structures within the constraints of modern switches, achieving minimal estimation errors [90, 91, 154, 227]. However, sketches are memory-intense data structures, and their accuracy directly depends on the amount of memory dedicated to sketching. As I show later, deploying sketches on switches alongside other functionality competing for limited memory results in significant accuracy degradation.

My survey of recent literature on in-network functions, which encompasses applications ranging from security to machine learning acceleration and networking, reveals a significant demand for switch resources. For instance, essential packet processing capabilities alone, such as L2/L3 forwarding, consume approximately 30% of a switch's memory, as shown in Table 3.1. The inclusion of additional functionalities further reduces the available memory for sketches.

To quantify the impact of this resource competition, I analyzed the Static RAM (SRAM) requirements of both established (i.e., Count Sketch [32], Count-Min Sketch [45]) and recently proposed sketches (i.e., UnivMon [139], SuMax [227]) for heavy hitter detection over a 30-second window (similar to previous works [139, 214, 222]), using real-world traffic traces from an Internet backbone [31]. I conducted experiments across a wide range of memory

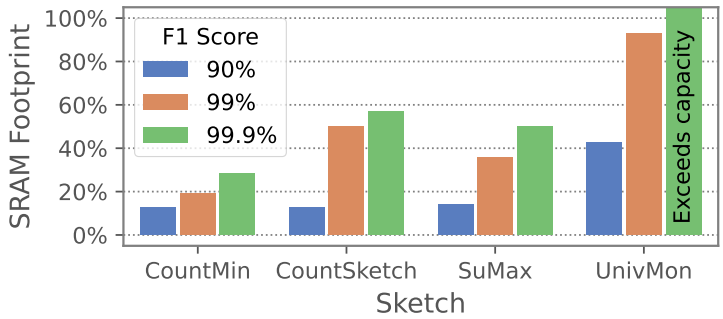


Figure 3.2: On-switch memory cost to achieve an accuracy target while monitoring 30s of real-world backbone traffic.

allocations for sketching to determine the required amount of memory to achieve pre-set target accuracies of 90%, 99%, and 99.9% in heavy hitter detection [1]. My findings, depicted in Figure 3.2, show that to achieve a 99% F1 score, the memory demand of sketches ranges from 20% to nearly 90% of a switch's SRAM, underscoring the challenge of maintaining high monitoring accuracy while co-locating sketches with other functions. A high monitoring accuracy is essential as the base of responsive diagnosis [132, 179], and a 99% accuracy already indicates a significant amount of incorrect flow classification. An improved classification accuracy requires even more memory and can sometimes exceed the switch's memory capacity, even in isolation.

Moreover, even if a sketch is not co-located with any other in-network function, it's important to note that the traffic volume observed by each switch can vary significantly, even among switches with the same logical role (e.g., edge switches in a data center) [19, 51, 176]. For instance, Meta reports that edge switches near caching racks handle approximately three times more traffic than those near web servers, as well as five times more simultaneous heavy hitter flows [176]. Consequently, deploying the measurement on a network topology cut (e.g., all edge switches [81]) results in varying degrees of accuracy, even when all switches have the same memory allocated for measurement.

[1] The exact dimensions of the sketches cannot be disclosed due to confidentiality regarding the hardware capacities of the switch.

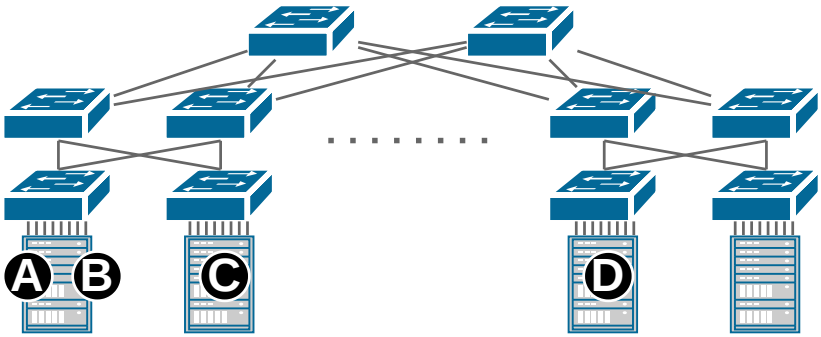


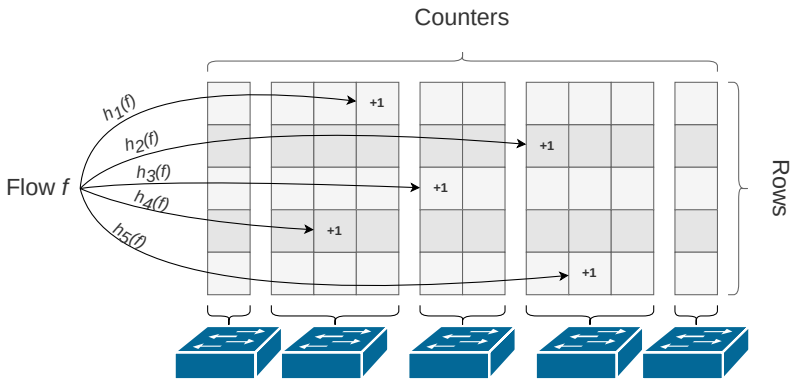
Figure 3.3: A Fat-Tree Topology. Packets traverse either 1 (A-B), 3 (A.C), or 5 (A-D) switches.

Acknowledging these challenges, it becomes evident that deploying complete sketches on a single switch is inefficient. Given these constraints, disaggregating sketches across multiple switches emerges as a promising solution. However, these solutions must be resilient against traffic and memory heterogeneities.

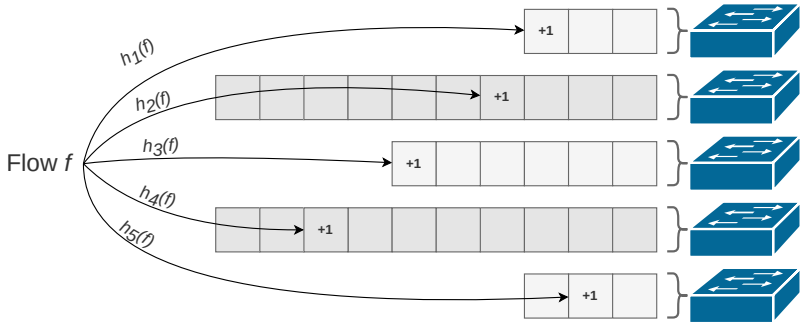
3.3 Sketch Disaggregation

Here I consider disaggregating *sketches*, which for this purpose can be viewed as a matrix structure in which each cell is an identical copy of a simpler data structure, usually a counter. I further assume that when an element (e.g., a packet) is inserted into the data structure, its key (e.g., flow ID) is mapped via uniform hashes into one or more cells in each row, and the cells are updated as appropriate. Examples of such sketches include ones for frequency estimation [32,46,58,91,227], set membership [23,59], sparse recovery [68,100], frequency moments estimation [7, 139], entropy estimation [41, 82, 139], ℓ_p samplers [42, 43], and many others.

Understanding the challenges of sketch disaggregation across multiple nodes begins with a depiction of a data center network's architecture. A classic fat-tree topology, commonly referenced in literature and employed in real-world deployments, is illustrated in Figure 3.3. This topology highlights



(a) Per-column disaggregation. Fragments host full-depth sketches.



(b) Per-row disaggregation. Fragments host one row each.

Figure 3.4: Visualization of sketch disaggregation directions. Note that fragments can have different amounts of cells.

the existence of numerous paths between any two end nodes, with the number of hops varying based on the nodes' locations. For example, flows between (A) and (B) traverse just a single switch, while any path between (A) and (D) contains five switches. Sketch disaggregation is then the process of dividing a sketch across network paths, where each network node hosts a fragment of the sketch. After central collection and re-aggregation, per-path fragments can be queried together to answer queries similarly to traditional aggregated sketches. With this in mind, there are two straightforward approaches to disaggregating sketches: *per-column* and *per-row* disaggregation.

In per-column disaggregation (Figure 3.4a), each network hop hosts all

sketch rows of the sketch matrix, but only a piece of each column. Keys are still mapped to one cell in each row, which should be selected uniformly at random by a hash function. Since each column is disaggregated across multiple hops, it is necessary to know the number of upcoming switches on the path and their respective configurations to build a "logical" column in which an index can be computed uniformly. Knowing the path, as well as the configuration of all switches, is a strong assumption for the data plane. Previous work on sketch disaggregation attempts to solve this through lookup tables in each fragment, containing entries for every network flow [75]. Requiring such a lookup table is incredibly costly, and defeats the purpose of sketching. If we already allocate per-flow information, why not simply allocate per-flow counters directly instead of using a probabilistic structure?

In per-row disaggregation (Figure 3.4b), each node holds specific sketch rows in each fragment. Typically each fragment will hold one row, with the row size determined by available memory. Note that, unlike standard sketches, the differences in row sizes due to memory lead to an "irregular" shaped matrix, and my work focuses on coping with the resulting variance from estimates. As each fragment is equivalent to independent sketch rows, they can function in isolation. and there is no need for a lookup table as with per-column aggregation. Hardware overheads are further lowered in Protocol Independent Switch Architecture (PISA) (which is the base of programmable Tofino switches). This architecture dedicates resources to specific functions at compile-time, and there is no need to host incrementing logic for more than a single row.

To illustrate the disaggregation overheads, I implemented a standard, a per-row (DISCO [29]), and a per-column (Distributed Sketch [75], lookup table excluded[2]) disaggregated count sketch on a Tofino switch, each using the same amount of memory for sketching. Given that the hardware pipeline, including per-function and per-packet computational units, is statically configured at compile-time, we can infer the exact computational footprint imposed by the various structures. Figure 3.5 presents these resource overheads, comparing

[2]Exclusion of the lookup table cost from Distributed Sketch emphasizes the inherently higher base footprint of per-column disaggregation, irrespective of indexing technique.

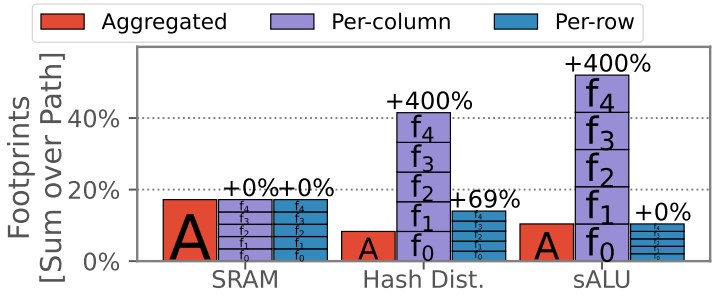


Figure 3.5: The disaggregation direction has a significant impact on computational resources. Shown here are full-path footprints in a Tofino programmable switch, broken down per switch/fragment.

the total resources required along a path with those of a traditional aggregated sketch. For example, we see that per-column disaggregation requires 5x as many stateful ALU instructions as a per-row disaggregated sketch, which tends to be one of the most limiting resources when developing packet processing pipelines in switches.

Regardless of the choice of disaggregation direction, there are several key challenges:

Challenge 1: Nodes across the network can have varying resources, a result of deploying distinct in-network functions at different switches. Some functions, such as security mechanisms, might be more suitably deployed at switches near endpoints [177, 204, 217], while others fit better within the network core [168].This diversity leads to a heterogeneous use of memory, ruling out per-column disaggregation due to the high computational overhead and substantial memory requirements for stateful per-flow counter allocation. Per-row disaggregation in highly heterogeneous deployments can experience accuracy degradation, where tiny fragments introduce significant errors to the composite sketch. Alternatively, these fragments become essentially negligible when assigned an importance proportional to their relatively high error.

Challenge 2: The volume of traffic can differ widely across nodes due to the design of data center networks and their traffic patterns. For example, the fat-tree topology facilitates massive multi-path routing [4], yet imbalances

persist despite load-balancing efforts [6, 71, 112, 113, 208]. These imbalances occur at the granularity of epochs (i.e., on the order of seconds), and therefore present a significant issue to disaggregated sketching. Studies have shown that much traffic remains local, with only a fraction traversing the entire datacenter [176]. Consequently, switches near end hosts experience higher traffic volumes than those at the core, affecting the accuracy of sketch fragments under heavy loads.

Challenge 3: The path lengths of different flows vary, influenced by the data center's network topology and traffic distribution. While some traffic remains local, affecting only a few nodes, other flows span across the network. This variance means that some traffic benefits from more extensive observation by multiple fragments, whereas others do not. For flows traversing only a single fragment, the accuracy degradation is akin to using a one-row sketch, which can yield significant inaccuracy. To demonstrate this effect, I deploy DISCO, a per-row disaggregated Count Sketch, in a fat tree topology using the same experimental parameters as further down in Section 3.6.1. We show a breakdown of the per-path-length's impact on heavy hitter detection in Figure 3.6.

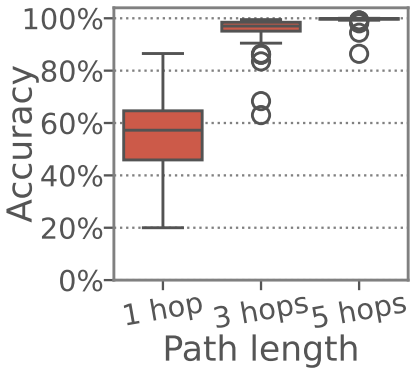


Figure 3.6: Path-lengths' impact on per-row disaggregation.

While per-row disaggregation generally offers better resource efficiency than per-column, designing robust techniques for its deployment in heterogeneous environments presents a key challenge. In Section 3.4, I introduce a general technique that can be used to deploy per-row disaggregated versions of sketches. Following this, I provide a concrete example of for sketch disaggregation by presenting DiSketch in Section 3.5, a disaggregated flow size estimator built for heterogeneous deployments.

3.4 Disaggregation Techniques

As previously discussed, the accuracy of sketch fragments varies across the system due to the heterogeneous loads and sizes of the fragments. This leads to a suboptimal resource utilization and can, in the worst cases, even lead to a reduction in accuracy compared to traditional aggregated sketches.

I will focus on Count Sketch disaggregation in this section by presenting *DiSketch*, a disaggregatable CS for heterogeneous environments. Furthermore, I will explain two techniques: *key-based sampling* and *subepoching*. These techniques collectively form the spatiotemporal indexing of DiSketch.

3.4.1 Per-Key Sampling

Sampling can be used to reduce the relative estimation variance between fragments. For example, a fragment that has less memory or under a higher load than other fragments can use sampling to process only a subset of incoming flows. This way, accuracies across fragments *for flows that are sampled* yield similarly high accuracies, allowing for easier aggregation across fragments. All fragments yield a decent minimum accuracy, regardless of theirs size of loads that they are under.

For example, the error of a Count Sketch row with w counters is bounded by the second frequency moment [32]. Given the unavailability of the exact frequency vector, we can approximate the second moment by the sum of the squared counters $\widehat{F_2} = \sum_{i=1}^{w} C_i^2$ [8], where C_i represents the value of the i-th counter. Following recently proven variance bounds on Count Sketches [126], we approximate the per-row Mean Squared Error (MSE) as a function of the key sampling rate p:

$$\text{MSE} \approx \widehat{F_2} \cdot p/w \implies p \propto w/\widehat{F_2}$$

The per-fragment sampling rate is statically defined at the start of each sketching epoch, using historical data. The fragments accuracies are balanced by adjusting the per-fragment sampling rate p as a function of w and $\widehat{F_2}$

relative to the network-wide averages $\overline{\mathbf{w}}$ and $\overline{\mathbf{F}_2}$:

$$p(w, \widehat{F_2}) = \min \left\{ 1 \ , \ \frac{w}{\overline{\mathbf{w}}} \times \frac{\overline{\mathbf{F}_2}}{\widehat{F_2}} \times s \right\} \tag{3.1}$$

, where s is the sampling factor denoting the sampling rate p of an "average" fragment. I discuss the assumption of in-band access to these values in Section 3.5.1, and empirically demonstrate its effectiveness further down in Section 3.6.2.

Direct application of sampling on fragments faces challenges due to the inability to normalize sampling rates across network paths, given the lack of real-time on-switch access to fragment widths and loads for any arbitrary network path. This discrepancy leads to significant risk in heterogeneous networks: heavily loaded paths or paths with smaller fragments may fail to sample some flows, rendering them not queryable or highly vulnerable to hash collisions in the case of a single on-path sampler.

3.4.2 Per-Fragment Subepoching

Sketch fragments are periodically exported for central collection, where they are aggregated and stored in chronological order. The period between a fragment's reset and its exportation is referred to as the sketching *epoch*. As the duration of an epoch increases, the number of processed items rises, leading to a higher estimation error. Previous work assumes that all rows in a sketch monitor the same time window, meaning all rows are exported and reset simultaneously. To my knowledge, no prior work investigates per-row epochs. However, introducing variability in epoch durations among fragments provides an opportunity to improve the accuracy of disaggregated sketches in heterogeneous environments.

All network-wide DiSketch fragments agree on the length of a full sketching epoch: T_e. However, fragments will simultaneously keep their own de facto epoch duration that is a power-of-two divisible of T_e. The set of available durations is therefore in the set $\{T_e, \frac{T_e}{2}, \frac{T_e}{4}, \ldots\}$. I refer to this fragment-specific epoch duration as the *subepoch duration* T_n, where $T_n = \frac{T_e}{n}$. Therefore, each

fragment exports and resets their counters n times per epoch, as n distinct *subepoch records*. These records are sent for central collection and aggregation, where they are awaiting queries.

The number of subepochs in fragment f_i is based on the time T_i at which the fragment error is expected to reach a pre-determined target error[3]. Therefore, the number of subepochs in a fragment will depend on a combination of the amount of allocated sketching memory as well as the expected load and flow size distribution traversing the fragment. The subepoch duration T_i is chosen to be the closest available T_n so that $T_i \approx T_e$, aligning the fragment with other fragments reaching similar error levels at similar times. I discuss the choice of a per-fragment n further down in Section 3.5.1.

Packets traversing several network nodes may be processed by fragments operating under different subepoch durations, for instance, fragments of varying sizes along a network packet's path for how subepoch durations are chosen). Thanks to the power-of-two divisibility of these durations and system-wide time synchronization, subepochs from different fragments can be seamlessly merged into perfectly overlapping sketching windows of length T_e[4]. Consequently, T_e is established as the temporal granularity for queries, with the query window being any arbitrary multiple of T_e.

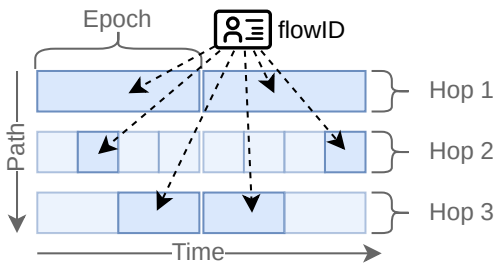


Figure 3.7: Spatiotemporal indexing, the base of DiSketch.

However, this approach has a problem. While all fragments may achieve similar estimation accuracies by the end of their respective subepochs, smaller

[3]Choosing a target error is up to the network operator. See Section 3.5.1 for a discussion.

[4]Without time synchronization, subepochs from different fragments risk being offset, and might not combine into perfectly overlapping full-length epochs.

fragments require concatenation of numerous subepochs to cover the entire query window. For a set of n independent estimations, each with variance e_i, the cumulative estimation variance is $\sum_{i=1}^{n} e_i$. As subepochs are aggregated to provide an estimate for a complete epoch, their estimation errors accumulate. Therefore, fragments of different sizes will *not* maintain the same estimation variance even with subepoching, compromising the reliability of estimations from smaller fragments.

To address this disparity, I integrate subepoching with key-based sampling by introducing *spatiotemporal indexing* (Figure 3.7), which is elaborated on in the following section.

3.5 DiSketch - A Spatiotemporal Count Sketch

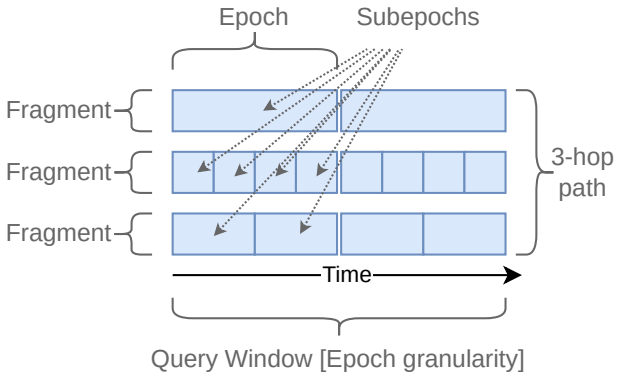


Figure 3.8: Terminology Overview. DiSketch epochs are divided into predefined subepochs. Fragments are autonomous and consist of single rows operating within fixed subepochs. Queries are executed against composite sketches, comprising subepoch records from all on-path fragments.

To demonstrate my proposed disaggregation techniques, I design and evaluate a disaggregatable frequency estimation sketch, monitoring flow sizes in a network. I aim to design a sketch that can effectively utilize heterogeneous

available memory at switches while surpassing the accuracy of both monolithic and naively disaggregated sketches.

At an overview, DiSketch is based on per-key sampling combined with per-fragment subepoching. This is accomplished by hashing a key (e.g., a flowID) into exactly one subepoch during each epoch, leading to flows being sporadically tracked by fragments. For example, a large fragment f_i where $T_i = T_e$ has $n = 1$, and tracks all flows all of the time, while a smaller fragment with $n = 4$ only tracks a quarter of flows at any given time.

Together, per-key sampling and subepoching form *spatiotemporal indexing*, which disaggregates sketches across space and time according to the capacities of each fragment. This way, DiSketch achieves high accuracy even in highly heterogeneous environments without imposing unreasonable assumptions or overheads such as knowledge of all fragments further down on a packet's path while ensuring that all flows remain queryable. Each fragment operates autonomously, without requiring direct communication or knowledge about any other individual fragment.

3.5.1 DiSketch Fragments

The fragments of DiSketch each consist of a single CS row, and are deployed to switches network-wide to cover every network path.

Equalizing fragment errors

Heterogeneity implies width or load heterogeneity between fragments, and DiSketch handles this through a combination of per-key sampling and per-fragment subepoching. Each fragment divides its local sketching epoch into subepochs, in which the CS only processes a subset of encountered keys. As previously mentioned in Section 3.4.2, the aim is to achieve a similar estimation error for each fragment in the network.

The drawback of this approach is that fragments only monitor a specific key intermittently, in specific subepochs, resulting in potential blind spots at specific time intervals. A fragment's estimations are essentially extrapolated from the rates seen while a key is tracked. This is not an issue for keys with

relatively uniform arrival times but can lead to estimation errors in keys with highly bursty arrival patterns. My evaluation, using real-world backbone traces, demonstrates a high accuracy on traffic patterns seen by ISPs.

Exporting Counters

DiSketch fragments' records are exported at the end of each subepoch, encapsulated in a data structure comprising the switch ID, counters, start and end times of the subepoch, and the hash seeds. DiSketch does not enforce a specific technology for record exportation, and one could, for example, download the memory buffers to the on-switch OS and send them for collection through TCP, or employ an in-band collection technique similarly to LightGuardian [227]. My solutions presented in this chapter result in a relatively low load on collection, and a high-speed collection solution such as my Direct Telemetry Access (DTA) technique (further down in Chapter 5), is not strictly necessary.

Following the export, counters are reset to prepare for the subsequent sketching subepoch. For the final subepoch of each epoch, the hash functions are also updated to prevent persistent hash collisions and mitigate continuous blind spots for certain keys on network paths with limited memory availability. The hash function replacement is performed locally by each fragment, without necessitating direct communication or collaboration with other fragments. For example, new hash functions could be chosen by generating a random Cyclic Redundancy Check (CRC) polynomial, or picked from a pre-generated list.

Fragments with a high number of subepochs increase the frequency with which counter information must be sent to centralized collection. However, the overall volume of data sent per time unit is approximately the same regardless of subepoch size, as smaller subepochs have fewer counters, and the overall volume of data collected centrally is similar to that of traditional monolithic sketches. Switches are expected to export at most a few megabytes per data epoch.

I discuss how subepoch records are aggregated and queried further down

in Section 3.5.2.

Number of Per-Fragment Subepochs

Fragments only monitor any given flow in one subepoch per epoch, and the number of subepochs in a fragment is essentially the inverse of the sampling probability p. Thus, the number of subepochs n is calculated based on the sampling probability p:

$$n = 2^{\lfloor \log_2 1/p \rceil} \tag{3.2}$$

Combining Equation 3.2 with Equation 3.1 from Section 3.4.1 gives us a function to calculate the number of per-fragment subepochs:

$$n = 2^{\left\lfloor -\log_2 \left(\min\left\{ 1 \ , \ \frac{w}{\overline{\mathbf{w}}} \times \frac{\overline{\mathbf{F_2}}}{\widehat{F_2}} \times s \right\} \right) \right\rceil} \tag{3.3}$$

Base Number of Subepochs

Determining the base number of subepochs for a fragment is essential. As shown in Equation 3.1, the sampling rate s is pivotal here, with a base formula $n = \frac{1}{s}$ for perfectly average fragments (i.e., $\widehat{F_2} = \overline{\mathbf{F}_2}$ and $w = \overline{\mathbf{w}}$). In other words, the number of subepochs is inversely linearly proportional to s. Unfortunately, finding an optimal s is not straightforward and is a balance:

- Increasing s reduces the number of subepochs, thus reducing the risk of per-flow blindspots where all on-path fragments sample away a flow simultaneously. When s is high, more fragments will have $n = 1$, i.e., a single subepoch per epoch. This can lead to a high accuracy variation for $n = 1$ fragments, leading to suboptimal composite accuracy in heterogeneous environments.

- Decreasing s results in more subepochs. This reduces the occurrence rate of $n = 1$ fragments, but generally increases the rate of per-fragment blind spots, thus increasing the risk of per-flow blind spots. The effect is negligible for flows with perfectly uniform transmission, but significant for highly bursty traffic patterns.

I acknowledge the complexity of this setting, and I have not investigated this tradeoff in-depth. My evaluation uses an arbitrary value of $s = 0.5$, leading to an average of $n = 2$ subepochs per fragment. This setting is most certainly not optimal. Spatiotemporal disaggregation is likely able to perform even better than my evaluation shows, under more optimized settings. I leave this investigation as future work.

Short Paths

Different flows traverse different network paths, which can be as short as a single hop (e.g., for intra-rack traffic). DiSketch detects single-hop network paths[5], where a key traverses just a single fragment, and handles them as a special case to provide better estimates. In these cases, collision resilience is re-established by mapping keys to three different counters. Specifically, one of three independent index-selecting hash functions is randomly chosen per packet. The counter output is normalized (that is, multiplied by three) at query-time to account for the distribution of updates between counters, and the median from these counters is used as the output from the fragment. This way, no additional memory logic is imposed on the fragments. Instead, just a small computational overhead is added for random number generation to select the hash function.

Required In-Band Knowledge

DiSketch fragments compute an appropriate number of subepochs according to Equation 3.3. Therefore, we need to motivate the in-band availability of the upcoming second frequency moment $\widehat{F_2}$, the network-wide average second frequency moment $\overline{\mathbf{F}_2}$, and the network-wide average fragment width $\overline{\mathbf{w}}$.

An exact second frequency moment F_2 for the upcoming epoch is practically unknowable. First, it is based on the ground truth frequencies of the flows, which is unavailable - hence sketching. Second, it requires perfect

[5]Single-hop paths can be detected by a switch when both the ingress and egress ports connect to either a server or a network egress, assuming that DiSketch fragments are deployed on every network switch.

prediction of the upcoming traffic distribution across the network. As shown in Section 3.4.1, a past F_2 can be reasonably approximated as $\widehat{F_2}$ by summing the squared counters of the previous sketching epoch. $\widehat{F_2}$ will act as a prediction of the F_2 of the upcoming epoch, motivated by traffic patterns being relatively predictable over a few seconds [21].

Similarly, the average network-wide traffic load is reasonably predictable, with movements on the timescale of hours [176]. Even on these timescales, the changes in network-wide utilization are relatively mild. Therefore one can either approximate this with a single static average or send updates to fragments on an hourly basis.

The final knowledge assumption is the network-wide average fragment width. This information is trivially accessible in static sketch deployments, where fragments are computed and distributed centrally. In deployments with dynamic reconfiguration, where fragments can grow and shrink during runtime, then this information is more troublesome. One could either approximate the network-wide average through a fixed value or send periodic updates to fragments when changes to the network-wide average occur.

3.5.2 Out-of-Band Querying

In this section, I explain how the exported DiSketch subepoch records are aggregated and queried at the collector to form comprehensive network flow size estimations.

A centralized collector retrieves subepoch records from fragments at the end of each fragment subepoch. These records are chronologically stored and aggregated per fragment, to be retrieved and combined with other fragments' records to answer queries for arbitrary network paths and time windows. An overview of the query algorithm of DiSketch is presented in Algorithm 1, and a step-by-step explanation is as follows:

Step 1 - Retrieving the network path

To answer a query for flow *flowID*, we first need to retrieve the network path S that those packets have traversed. DiSketch does not supply network paths

Algorithm 1 DiSketch Query Processing

Input: *flowID*, t_{start}, t_{end}
Output: Composite estimation for *flowID* over the query window
Step 1: Identify switches S involved in path(*flowID*).
Step 2: Extract relevant subepoch records SE:
    (a) For each switch S_i, retrieve the set of stored subepochs SE_i where $t_{start} \leq SE_{i,x}.start \leq t_{end}$.
    (b) Exclude subepochs in SE not sampling flowID.
Step 3: Normalize subepoch records SE to $\hat{SE}$:
    (a) Determine minimum subepoch length T_{min} across SE.
    (b) For each SE_i, compute normalization factor $r_i = \frac{SE_i.T}{T_{min}}$.
    (c) Divide each SE_i into r_i chronologically contiguous subepochs $\hat{SE}_{i,j}$, where each $\hat{SE}_{i,j}.T = T_{start}$, $\hat{SE}_{i,j}$.output $= \frac{SE_i.output}{r_i}$, and $\hat{SE}_{i,j}.start = SE_i.start + j \times T_{min}$.
Step 4: Compute query output from $\hat{SE}$:
    (a) Group $\hat{SE}$ records by time-window into W where $|W| = \frac{t_{end}-t_{start}}{T_{min}}$. For each W_i, set $W_i = \{\hat{SE}_x.output | \hat{SE}_x.start = t_{start} + iT_{min}\}$.
    (b) Calculate the composite estimation as $Output = \sum_{i=0}^{|W|-1} median(W_i)$.

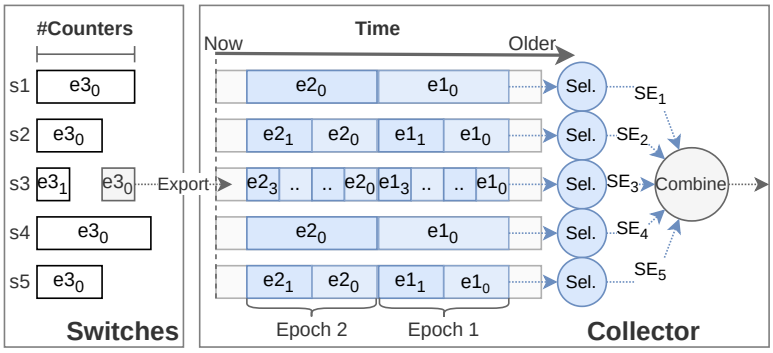


Figure 3.9: DiSketch epochs are dynamic composites built from subepochs of autonomous sketching fragments. Subepoch records are collected from all fragments and centrally aggregated. Relevant records are selected and processed on a per-query basis, building dynamic queryable composites.

directly and requires path tracing telemetry to be present in the network, or that network paths can be centrally calculated. Rapid path changes occurring within an epoch can be troublesome, and I discuss this in the chapter discussion (see Section 3.7.4).

Step 2 - Retrieving the subepoch records

We need to retrieve the subepoch records that have monitored the flow during the query window. The subepoch storage is visualized in Figure 3.9. Records are stored chronologically, aggregated under each fragment, and retrieving the relevant subepochs is therefore straightforward. The sampling hash is re-computed per epoch for each fragment and all *flowID*-sampling subepochs are retrieved, yielding a list of subepochs (SE) that monitored the *flowID* during the query window. This is visualized as (1) in Figure 3.10. We now have a list of all relevant records, and the next two steps describe how these are combined to yield a single estimation of the key frequency during the queried time interval.

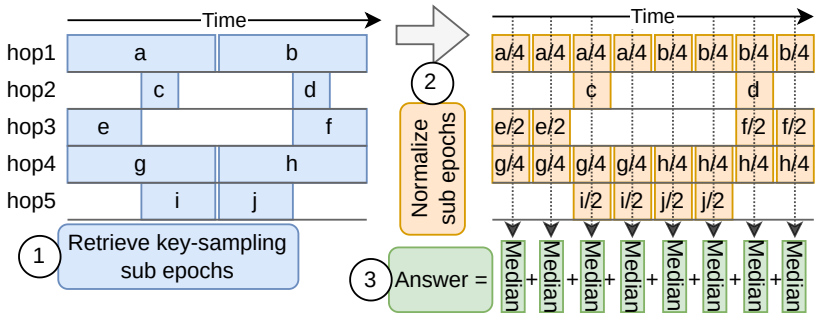


Figure 3.10: Querying a key in DiSketch. On-path records are retrieved, and the subepochs that sampled the key during the query window are selected. Records are normalized and queried as a composite.

Step 3 - Normalizing the subepochs

These subepoch records in SE are queried as single-row Count Sketches, using the record-specific indexing- and signage hash seeds, yielding a list of frequency-estimation outputs from the subepochs. The subepoch records in SE are normalized into $\hat{SE}$ so that they are all equal-length, matching the length of the shortest subepoch duration T_{min} present in SE. This is done by computing a normalization factor r_i for each subepoch record SE_i, so that $r_i = \frac{SE_i.T}{T_{min}}$. The record SE_i is split into r_i contiguous subrecords $\{\hat{SE}_{i,0}, ..., \hat{SE}_{i,r_i-1}\}$ each of length T_{min}. Each new subrecord is assigned the estimation-output $\hat{SE}_{i,j}.output = \frac{SE_i.output}{r_i}$. The normalized subepoch vector $\hat{SE}$ comprises normalized subepochs records of lengths T_{min} where $\sum_{i=1}^{|SE|} SE_i.output = \sum_{i=1}^{|\hat{SE}|} \hat{SE}_i$.output, thereby retaining total output integrity. Subepoch normalization is visualized as (2) in Figure 3.10.

Step 4 - Composing a composite sketch output

The normalized subepoch records are clustered together into groups based on the records' start times. Intra-group medians are calculated, yielding a list of per-time-window flow size estimates. Finally, these per-group estimates are summed together to yield a flow size estimation based on the composite sketch. This final step is visualized as (3) in Figure 3.10.

3.5.3 Hardware Implementation of DiSketch

Demonstrating the hardware feasibility of my disaggregation technique, I have implemented DiSketch in Tofino, a P4-programmable switch. My technique is lightweight, imposing minor on-switch overheads on top of naive per-row disaggregation. Subepoch durations are decided at compile-time, matching the width of the deployed fragment. Integrated time synchronization triggers epoching by generating a control packet to the switch-local CPU. Subepoching uses almost the exact same mechanisms as traditional epoching, imposing minor overheads. Subepoch sampling is done by adding a single hash computation using the built-in CRC engine with a custom polynomial, resulting in a stateless and re-computable mapping between keys and subepochs. The re-introduction of redundancies for single-hop paths is triggered if both the ingress and egress ports are connected to hosts and randomly select one of three CRC polynomials to use for the index computation in the counter array.

As described in Section 3.4.1, I approximate the second frequency moment F_2 as $\widehat{F_2} = \sum_{i=1}^{w} C_i^2$ [8]. Expanding DiSketch with predictive load awareness requires updating $\widehat{F_2}$ after every counter update, necessitating one additional stateful ALU call per fragment. Say c is the value of the counter that will be modified, we can then update $\widehat{F_2}$ accordingly:

$$\widehat{F_2} = \begin{cases} \widehat{F_2} - c^2 + (c+1)^2, & \text{if incrementing} \\ \widehat{F_2} - c^2 + (c-1)^2, & \text{if decrementing} \end{cases} \tag{3.4}$$

We can then use $\widehat{F_2}$ on-switch for computing n at the end of each epoch[6].

3.6 Evaluation

Topologies. The network topologies and memory distributions from Figure 3.11 are used throughout the evaluation, demonstrating disaggregation in a variety of scenarios. This encompasses four scenarios, which integrate

[6]The accuracy of $\widehat{F_2}$ compared to F_2 is impacted by subepoching, since the counters on which it is based periodically reset. I leave this investigation as future work.

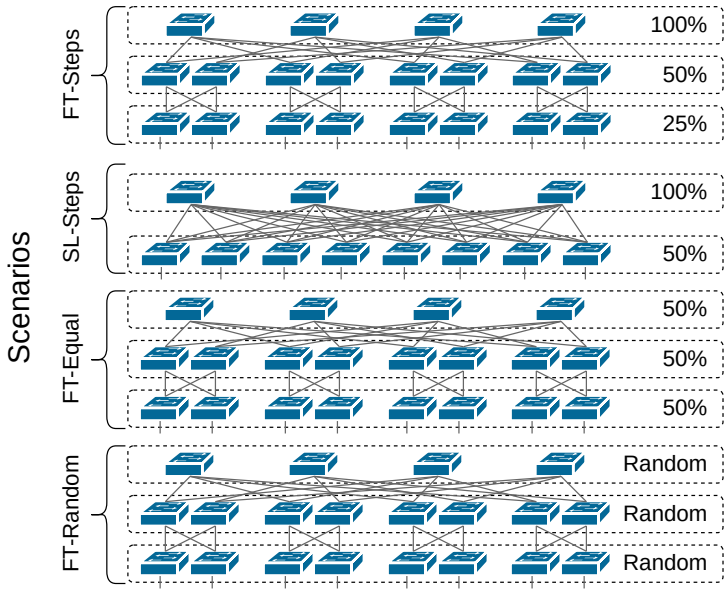


Figure 3.11: Network topologies and memory distributions used during evaluation.

two common network topologies – FatTree (FT) and SpineLeaf (SL) – with three distinct memory distributions as illustrated. The Random memory distribution method generates pseudo-random memory pools for switches. This approach ensures the generation of memory pools with a pre-defined average size and inter-switch memory size heterogeneity, designed to correspond with the targeted Gini inequality index specified for the tests. The generated memory pools are randomly distributed to switches, not considering their topological position. A fully random distribution is likely a poor approximation of real-world deployments, but grants an insight into the robustness of my solution. As a default, FT_Random has gini = 0.4, resulting in significant width heterogeneity. An example per-switch width distribution at gini = 0.4 is: [13%, 14%, 16%, 31%, 32%, 32%, 44%, 49%, 66%, 72%, 88%, 114%, 117%, 163%, 172%, 175%, 178%, 203%, 205%, 216%], i.e., fragment-sizes ranging from *13%* to *216%* of the mean. For comparison, FT_Steps has a Gini of 0.28, Sl_Steps 0.17, and FT_Equal 0.

Due to the financial costs of building a large-scale testbed with sufficient

programmable switches, all probabilistic properties of my technique are evaluated through comprehensive simulation. This simulator's functionality has been verified to be identical to the hardware prototype.

Traces. In this evaluation, I use the real-world CAIDA-NYC Equinix packet trace [31], recorded at a backbone link in 2019. If nothing else is stated, then I have replayed 30 seconds of traffic (∼16M packets, covering ∼1.2M flows). I only have access to traffic recorded at a single network link, and we uniformly map IP addresses to hosts in the network to allow for network-wide communication.

I chose to use the CAIDA trace because it provides an extensive and detailed packet trace necessary for my analysis. Specifically, I needed a very long packet trace to ensure that the DiSketch structure would be adequately saturated and tested under substantial traffic volumes. Unfortunately, I did not have access to data center traces that were large enough to meet these requirements.

It is important to note that the flow size distribution in the CAIDA trace may not directly translate to other network environments, such as data centers. Traffic patterns significantly impact the efficiency of spatiotemporal disaggregation and thus affect DiSketch. Intra-flow burstiness, for instance, might go undetected by per-fragment blind spots. Data center traffic characteristics can vary widely depending on the specific scenario, and a single trace is unable to universally represent all possible traffic patterns. While the CAIDA trace serves as a useful proxy, it does not definitively represent data center traffic, which would likely exhibit different flow size distributions and patterns.

To mitigate DiSketch's susceptibility to blind spots, the base number of subepochs can be reduced. However, this comes at the cost of reduced effectiveness in mitigating heterogeneity-driven inaccuracies (see the discussion in Section 3.5.1).

DiSketch Models. Here, I evaluate two different versions of DiSketch: *DiSketch-Uni* and *DiSketch-Bi*. Their designs are the same except for the information that n (i.e., the number of fragment subepochs) is based on. In DiSketch-Uni, the choice of n for a fragment (yielding 2^n subepochs) is based

the ratio of the fragment size to the network-wide average fragment size, and does not take into account inter-fragment load heterogeneity. In contrast, DiSketch-Bi bases n both on the width ratios, as well as on the ratio of the fragments' loads to the network-wide average load for the upcoming epoch. In practice, the load for an upcoming epoch is likely unknown and has to be predicted based on knowledge about the network as well as historical data. The load pattern tends to be relatively predictable on the order of seconds [21]. The DiSketch-Bi evaluated in this section has full knowledge about the incoming traffic load [7].

3.6.1 DiSketch Estimation Accuracy

In this section, I aim to show the overall benefit of disaggregation and DiSketch on sketching accuracy. Here, I evaluate DiSketch-Uni, the "simple" DiSketch model that is unaware of the per-switch load and only mitigates size heterogeneities. DiSketch-Uni is deployed with an exportation rate of 16 (see Section 3.6.4 for the impact of this choice). For simplicity, I focus purely on utilizing memory gaps in this experiment[8]. Load heterogeneity is not explicitly modeled, and IP addresses in the packet trace are uniformly distributed across the hosts. Count Sketch is deployed monolithically, allocating all available memory at the core switches towards sketch counters, while DISCO [29] and DiSketch use all available network-wide memory for sketching. The experimental results are presented in Figure 3.12.

We see clearly how the monolithic Count Sketch fails to deliver acceptable accuracies under resource gap deployments since it is restricted to the memory size of single nodes. DISCO and DiSketch both use identical memory and counter logic. However, per-node epoching as well as time-based sampling significantly reduces the estimation errors of DiSketch.

[7] I refer back to the previous discussion in Section 3.5.1 about the feasibility of load prediction.

[8] DiSketch is already demonstrated to impose a small footprint on non-memory resources. Computational resource gaps are less interesting to investigate since having too few computational resources means that a fragment can not deploy at all.

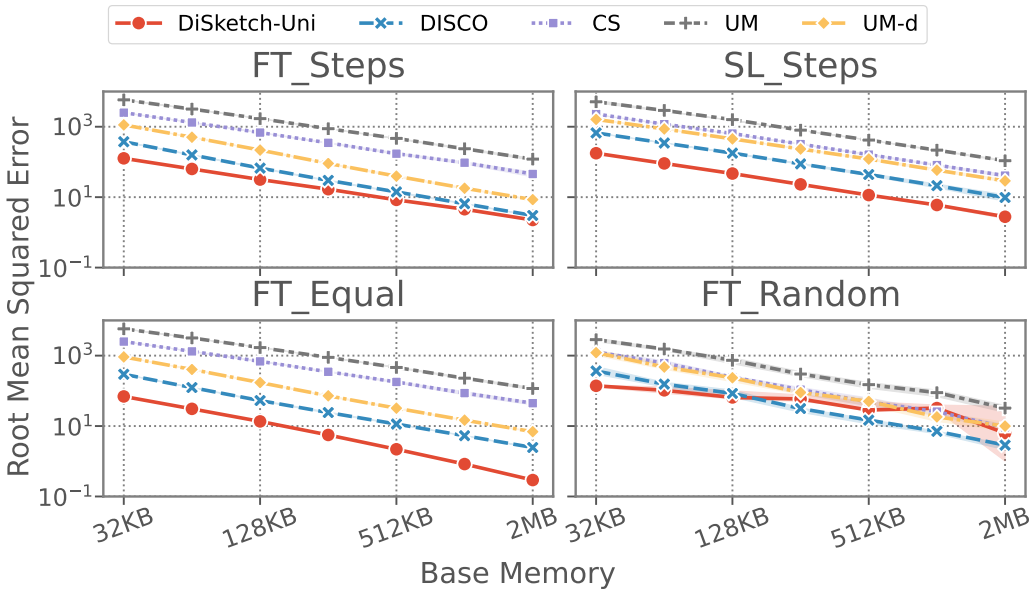


Figure 3.12: The flow size estimation error of load-unaware DiSketch-Uni in comparison with traditional deployments, at various memory sizes.

Note the unstable accuracy of DiSketch in the FT_Random scenario. Here, some flows are unlucky and only encounter tiny fragments that are performing time-wise sampling. All fragments in a path can accidentally sample away a key during an overlapping window, resulting in momentary monitoring blind spots. This showcases the importance of deploying sketch disaggregation techniques that fit the environment.

Recall that this experiment investigates Load-unaware DiSketch (DiSketch-Uni). This design is unaware of the load within an epoch and is susceptible to accuracy degradation when some fragments are unexpectedly overloaded with traffic. We can see this effect in the low-error high-memory deployment scenarios, especially in $>$ 1MB FT_Random. Investigations of the raw experimental data show that poor accuracy presents when large fragments are simultaneously highly loaded due to suboptimal loadbalancing[9]. DiSketch-Uni

[9]The experiment employs Equal-Cost Multi-Path routing (ECMP) to load balance traffic in the topology.

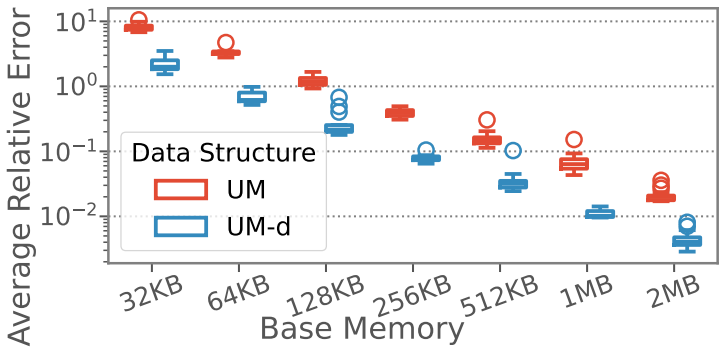


Figure 3.13: The error in entropy estimation of UnivMon, with and without disaggregation.

calculates subepoching based purely on fragment sizes, which risks assigning higher-than-optimal importance for over-loaded fragments. I am confident that Load-aware DiSketch (DiSketch-Bi) would outperform DiSketch-Uni, and I plan to include these experiments in a full publication following my doctoral graduation. For now, please refer to the following section 3.6.2 for a demonstration of the benefit of load-aware subepoching.

Takeaway: Sketch disaggregation reduces monitoring errors, and DiSketch-Uni further improves the monitoring accuracy by a near order of magnitude. Subepoch sampling can result in momentary blind spots in low-memory paths.

Entropy Estimation

In this section, I inspect the benefit of disaggregation on a non-flow size estimation sketch, namely UnivMon. Using the same experimental setup as before, we deploy UnivMon for entropy estimation queries in the FT_Random scenario. This sketch is both deployed aggregated on core switches, as well as disaggregated across the network. Here, I show the impact of disaggregation on entropy estimation, using the FT_Random with the same parameters as the previous experiment. I present the results in Figure 3.13, where we see a significant error reduction of UnivMon when disaggregated.

Takeaway: My disaggregation techniques are not limited to flow size

estimation, and can reduce entropy estimation errors in UnivMon by approximately 80%.

3.6.2 Performance in Heterogeneous Environments

Here, I assess how well DiSketch handles sketching in heterogeneous environments (i.e., where the traffic load and memory sizes vary between switches). I evaluate the flow size estimation accuracy of the two DiSketch models (i.e., DiSketch-Uni and DiSketch-Bi) as well as DISCO [29] which performed well during the previous full-scale experiments in (Section 3.6.1).

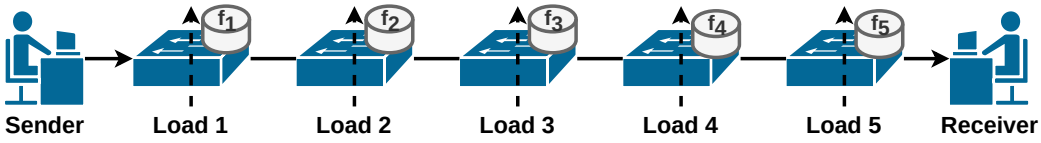


Figure 3.14: Experimental Setup in Heterogeneity Tests.

Topology. As opposed to the prior experiment, these tests simulate a single 5-hop network path (shown in Figure 3.14). The load heterogeneity of networks depends on a variety of factors including the network topology, load balancing, topological position of nodes, and the traffic patterns of connected hosts/clusters. By focusing on a single path, I gain precise control of the per-switch traffic volumes and can set the heterogeneity levels freely. Traffic on other semi-overlapping paths is emulated as per-switch background traffic.

Generating Heterogeneity. I simulate a range of heterogeneity levels for fragment sizes and traffic loads of the hops, defined as the Coefficient of Variation (CoV), i.e., the relative standard deviation. I evaluate multiple CoVs ranging from perfectly homogeneous (CoV = 0) to significantly heterogeneous (CoV = 1.8). The total amount of background traffic ($259K$ packets) and the number of on-path counters (5120) is fixed across tests[10]. I generate

[10]The small scale in these simulations allowed me to evaluate numerous heterogeneity settings within a reasonable time, and yields the same general pattern as full-scale heterogeneity simulations. Full-scale DiSketch experiments were presented in the previous Section 3.14

two random lists of 5 integers, one for the per-hop load, and another for the per-fragment width. Both of these lists are generated so that the sum equals the pre-set total amount of packets/counters, and that their coefficients of variation approximately match the test. For instance, a background traffic distribution with CoV ≈ 1.5 can be [204189(78.7%), 18364(7.1%), 29(0.01%), 2265(0.9%), 34675(13.4%)]. The generated loads and widths are independent and are randomly distributed to the simulated nodes.

Network Traffic. There is a total of six different packet streams: five background traffic streams passing through one switch each, and one evaluation stream that traverses the entire path. I use the aforementioned CAIDA packet trace as the base of all network traffic in this simulation, and map IP addresses into the different traffic streams. Packets in each stream are then replayed chronologically in the order they appear in the packet trace. There is a single sender/receiver pair in the simulation, at opposite ends of the simulated path. $2.6K$ packets are transmitted in this stream and across the full path, comprising ~ 300 flows. The network traffic is replayed perfectly mixed so that each time window contains traffic from the streams in proportion to their sizes.

Evaluation Metric. I evaluate the flow size estimation accuracy of the disaggregated sketch, retrieved by querying the flows from the evaluation stream at the end of the simulation. For this, I use the Normalized Root Mean Squared Error (NRMSE) [16]. NRMSE provides a dimensionless measure of error by normalizing the Root Mean Squared Error (RMSE) based on the total number of packets in the stream. The RMSE is calculated as the square root of the Mean Squared Error (MSE), which quantifies the average squared difference between the estimated and actual number of packets in a flow. The NRMSE is then obtained by dividing the RMSE by the total number of packets in the stream, ensuring that the error metric is independent of the scale of the data. Formally, NRMSE is defined as follows:

$$\text{NRMSE} = \frac{\text{RMSE}}{\text{numpkts\_total}} = \frac{\sqrt{\text{MSE}}}{\text{numpkts\_total}}$$

A straightforward metric such as average error is not used, as it does not account for the magnitude of individual errors and can mask large

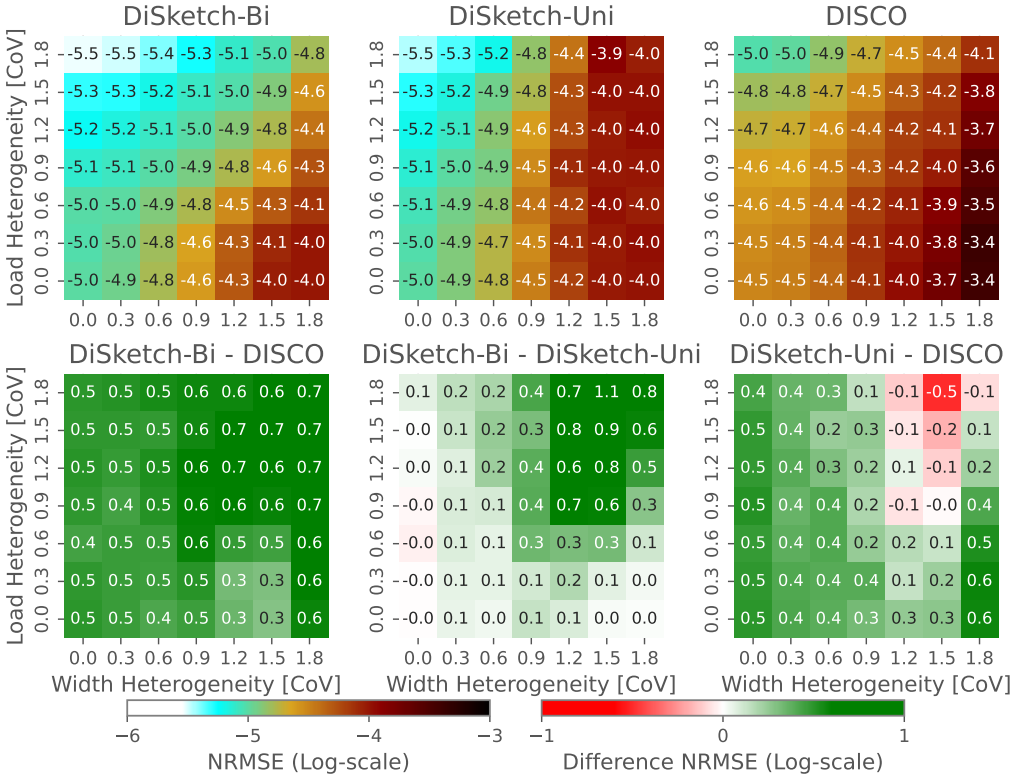


Figure 3.15: Heterogeneity's impact on flow size estimation, showing the log-NRMSE at different levels of heterogeneity.

discrepancies between estimated and actual values. In contrast, NRMSE, by considering the squared differences, provides a more robust and sensitive measure of the model's performance [16]. In practice, a change in NRMSE can indicate a significant improvement or degradation in model accuracy, with smaller values indicating better performance.

Results. The average NRMSE at each heterogeneity pair is presented in Figure 3.15. To further simplify the comparison, I show the difference in NRMSE at the bottom. This experiment reveals a clear link between network heterogeneity and the performance of sketches. Fragment width heterogeneity has a detrimental effect on performance, whereas traffic load heterogeneity enhances it. The resilience of sketches to hash collisions, achieved through

counter redundancy, explains this relationship.

High heterogeneity in fragment width, as seen in the random width distributions[11], leads to a general reduction in the size of most fragments. Although one fragment may grow large and experience a lowered collision rate, this does not offset the diminished performance caused by the reduced effectiveness of all redundant measurements, regardless of their individual contributions to the composite output. Conversely, significant traffic load heterogeneity condenses the majority of background traffic onto a limited number of hops, thereby reducing the load on the remaining nodes and enhancing the performance of most fragments.

The performance benefit of load heterogeneity essentially disappears for DiSketch-Uni, where subepoching relies solely on fragment sizes without considering traffic loads. Large DiSketch fragments significantly influence the final output, which correspondingly leads to poor performance when traffic overload makes these large fragments poor estimators. This issue is exemplified through the DiSketch-Uni model, which underperforms compared to DISCO (shown in the right graph) in environments characterized by both high width and load heterogeneity. Note, however, that the load-unaware DiSketch-Uni unsurprisingly performs well in homogeneous load environments. DiSketch-Bi, the load-aware DiSketch model, retains the positive effects of load heterogeneity. Consequently, I advocate for the implementation of load-aware subepoching in the deployment of disaggregated sketches within environments exhibiting significant load heterogeneity.

Finally, in comparison with DISCO, DiSketch-Bi consistently enhances the accuracy of flow size estimation without requiring additional memory resources. The extent of improvement varies with the evaluation parameters, achieving a reduction of NRMSE between 0.2 and 0.8. This range of improvement signifies a near-order-of-magnitude enhancement in performance.

[11]The shape of the distribution can affect the results. For instance, a "balanced" heterogeneous distribution, in which each shrinking fragment is counterbalanced by another growing fragment, generally outperforms a "one takes all" distribution where every fragment except one shrinks uniformly. The results presented here reflect the average from numerous simulations with entirely random width and load distributions.

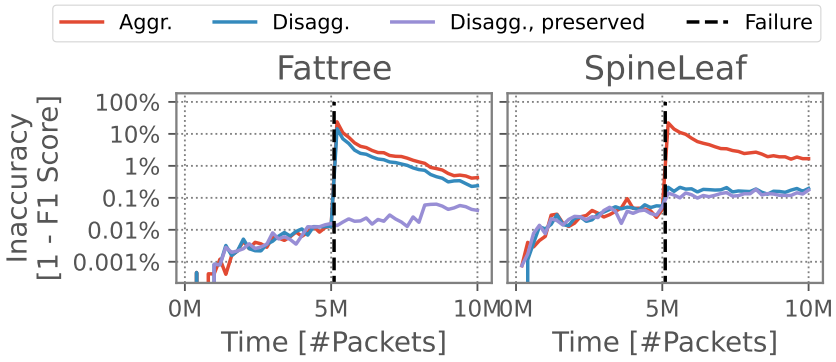


Figure 3.16: Disaggregated sketches are failure resilient, partially retaining the state after failure events. Shown here is the inaccuracy of real-time heavy hitter detection.

Takeaway: The environmental heterogeneity has a significant impact on the sketch performance, and disaggregated sketches benefit from load heterogeneity. DiSketch consistently improves sketching performance in heterogeneous environments, *reducing errors by almost an order of magnitude*.

3.6.3 Failure Resilience

Network failures are critical events that require thorough investigation and troubleshooting to identify their root causes. Loss of telemetry data during these incidents is especially problematic, as operators depend on information from the affected time and location for their insights. Disaggregated sketches have their counters spread out across multiple switches, and therefore retain much of their state during node failures. In this section, I evaluate how effective disaggregation is at maintaining accuracy measurements during switch failure events. I replay 10M network packets through two topologies, fat tree, and spine leaf, so that all flows traverse the network core. Here, I perform real-time heavy-hitter detection through a standard Count Sketch, either deployed aggregated on core switches or disaggregated across the network. I chose to keep row widths identical in all switches, regardless of deployment, so the baseline accuracy is the same for both aggregated and

disaggregated sketches. A failure event is triggered at the halfway point in the experiments, disconnecting one of the core switches and forcing the re-routing of affected flows. I present the results in Figure 3.16.

Note the impact that the topology has on the failure resilience of the disaggregated sketch. Failure resilience of disaggregated sketches is thanks to an overlap in the set of on-path sketch rows for a flow before and after a failure event. Paths in spine leaf consist of three nodes: two edges and one core. When a core switch fails, the post-failure sketch will still retain 2/3 rows, which is still enough so that a majority of rows retain the original statefulness; we see how disaggregation leads to highly accurate measurements immediately following a failure. The fat tree consists of up to five hops: two edges, two aggregation, and one core. The number of original rows remaining on-path post-failure is either 2/5, 3/5, or 4/5; therefore, a worst-case post-failure event can lead to most hops being replaced in a flow path. We see this effect in the fat tree results, where the disaggregated sketch also experiences a significant drop in accuracy after the failure (although somewhat smaller than for aggregated sketches). If the re-routing algorithm is modified to preserve as much of the original path as possible post-failure, then this effect disappears and disaggregation is highly failure resilient. Given that disaggregated sketches are assumed to be deployed in Software-Defined Network (SDN) environments, operators would have full control of post-failure routing and fast re-route methods.

Takeaway: Disaggregated sketches are highly failure resilient, depending on the topology and re-routing algorithm, reducing the post-failure drop in accuracy from basically complete to barely noticeable.

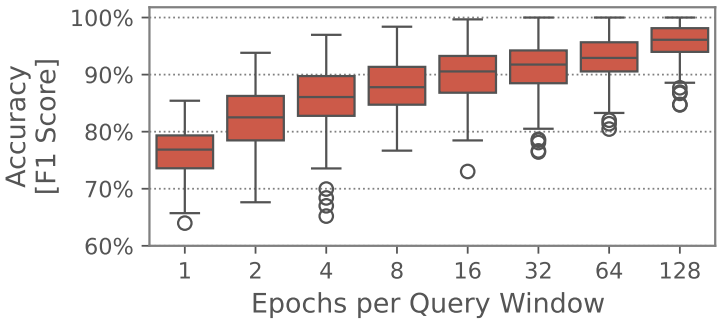


Figure 3.17: Higher-rate exportation increases DiSketch performance.

3.6.4 Rate of Epoching

Subepochs are re-computed at the start of each epoch, altering the subepoch sampling and counter-indexing of all DiSketch fragments. This both counteracts persistent hash collisions, as well as avoids persistent time-wise blind spots for unlucky flows. Here, I demonstrate the impact that a fine-grained temporal fragmentation has on the sketching accuracy of DiSketch, by altering the rate of epoching. I keep the query window fixed across tests but vary the length (and therefore the amount) of epochs within it. I simulate DiSketch-Uni in the FT_Random scenario, and present the results in Figure 3.17.

We see that frequent epoch re-computation results in significant accuracy improvements, where the median F1 score can be increased from 77% to $> 99\%$ entirely by increasing the rate of epoching, with no change in on-switch memory consumption. There are two tradeoffs to increasing the exportation rate: the load on collection increases, and offline queries require more computations to answer. Offline queries require two hash computations per-hop and per-epoch within the query window: once to retrieve the subepoch, and once to compute the intra-subepoch index. Thus, the query complexity grows linearly with the number of epochs. The temporal granularity of DiSketch should therefore be adjusted according to the needs of the network, ensuring that neither the collection stacks nor analysis engines are overwhelmed.

However, it should be noted that disaggregation increases the speed of sketch extraction by allowing parallel extraction from multiple nodes,

effectively using the memory-extraction bandwidth of all switches instead of just the bandwidth of a few sketch-hosting switches.

Takeaway: Querying over multiple epochs results in significantly increased estimation accuracies, at the expense of a linear increase in collection load and query complexity.

3.7 Discussion & Future Work

This section provides a brief discussion of spatiotemporal disaggregation and proposes future research into the techniques presented in this chapter.

3.7.1 Outlier Sensitivity

Several aspects of the design rely on global properties such as the amount of traffic or memory allocated at switches (Section 3.4). While this approach is suitable in many cases, there could be others in which a single or few switches skew the parameters and degrade the overall accuracy. For example, consider a scenario where intra-rack traffic is prominent [176]. In such cases, it might be appropriate to change the average value to a metric more robust to outliers, such as the median or trimmed mean. Alternatively, if the skewness is the result of inter-zone heterogeneity, and most traffic stays intra-zone, then it could be argued that per-zone normalization is preferred over network-wide normalization.

3.7.2 Other Data Structures

This dissertation chapter presented the idea of spatiotemporal disaggregation and demonstrated its usefulness through the example of a Count Sketch. However, I expect these ideas to be useful beyond that and envision that they can be applied to other similar sketches (e.g., Count-Min Sketch, Bloom Filter, and HyperLogLog), as well as possibly other non-sketching data structures. The exact technical requirements for spatiotemporal disaggregatability, as

well as structure modifications required for individual sketches, are left as future work.

Access Control Lists

There are some network functions, such as Access Control Lists (ACLs), that could benefit from disaggregation. ACLs can be memory-heavy structures, and disaggregation would spread out this footprint over multiple devices. Further, destination filters could be placed closer to the destination servers, while source filters could be placed near those network clusters, or at network ingress points. Finally, disaggregation would change the ACL placement strategy, and potentially reduce the risk of invalid placement leading to a fraction of traffic circumventing filtering due to invalid placement and/or routing. However, a new set of challenges would arise, such as designing algorithms for optimal rule placement that allows for efficient use of resources, while also allowing for early reactivity and triggering of blocking rules.

HyperLogLog

HyperLogLog (HLL) is a sketch-like data structure used for cardinality estimation, e.g., to estimate the total number of distinct network flows in a stream of packets. Similarly to flow size estimating sketches (e.g., CS, Count-Min Sketch (CMS), and UnivMon), the amount of allocated memory increases the accuracy of the estimation. If the HLL registers (or counters) are split into multiple virtual rows, similar to flow-size estimators, one can likely apply many of the disaggregation techniques mentioned in this chapter. I envision that we would see similar benefits when spatiotemporal disaggregation is applied to HLL, but leave the design of this data structure as future work.

3.7.3 Finding an Optimal Subepoch Granularity

Spatiotemporal disaggregation is built around subepoching, where the monitoring epoch is dynamically divided into briefer subepochs based on the capacities of the individual fragments. However, one critical aspect has not

been investigated: the average number of subepochs that a fragment will use. I am referring to specifying s in Equation 3.1. As discussed in Section 3.5.1, choosing the static value of s is non-trivial, and is a tradeoff that depends on multiple factors including the flow traffic patterns and expected heterogeneity levels. This warrants a full theoretical analysis, as well as experimentation to demonstrate the impact of this choice in various network environments.

3.7.4 Path Stability

Some load balancing schemes, such as flowlet switching [208], frequently alter the path of flows at incredibly short timescales. These techniques result in irregular and unstable flow paths, impacting the practicality of sketch disaggregation. For instance, if flow paths change during a sketching epoch, then portions of the flow increments within that epoch could end up in different sets of fragments. Without awareness of these path changes, the estimation accuracies of the analysis engine could suffer. If the analysis engine is unaware of such paths, then the estimation accuracies would suffer. However, it is possible to design disaggregation techniques that accommodate path changes. Assuming fine-grained path tracing is already implemented, the analysis engine could incorporate all fragments traversed during the epoch into the composite sketch output. Weighting could be employed based on the duration during which a flow has traversed each fragment. The development of precise techniques to perform sketch-based estimations under these conditions remains an area for future work.

Alternatively, one could configure the load balancing techniques to only perform re-routing at epoch transitions. This adjustment would ensure that each sketching epoch contains the same set of fragments, except in cases of re-routing triggered by failures. This method could stabilize the path data within each epoch, improving the consistency and accuracy of sketch-based monitoring.

3.7.5 Query-Time Weighting

DiSketch is decent at equalizing the errors between heterogeneous fragments, to some degree. However, there are three sources of inequality that it can not mitigate:

Max-length subepochs. Fragments with a high amount of memory and/or a low amount of traffic load can be dynamically configured to run a single subepoch per epoch ($n = 1$), which is the extreme setting for the most highly capable fragments. Unfortunately, it is not possible to configure less than one subepoch per epoch (i.e., we can not sample $> 100\%$ of flows at any given time). Therefore, spatiotemporal disaggregation is unable to equalize the accuracy of the most highly performing fragments, and the $n = 1$ group can contain a wide range of fragment capabilities.

Subepoch bucketing. A fragment can only be configured with a power-of-two number of subepochs. This limits the efficiency of sketch-time error equalization since a subepoch group can contain fragments that just barely qualified, to other fragments that are nearly twice as capable and *almost* got promoted to a subepoch halving. This leads to error inequalities that subepoching is unable to solve at sketching-time.

Load prediction. Spatiotemporal disaggregation, as it is implemented in DiSketch, attempts to configure upcoming epochs by extrapolating from historical information to predict the load of the upcoming epoch. However, this is highly unlikely to be entirely accurate in practice, and the load is almost guaranteed to deviate from the prediction. Inaccurate predictions can very well lead to suboptimal epoch configuration, where some fragments in a subepoch group might be much more or less capable than they were expecting.

Potential Solution. These accuracy inequalities are not currently handled, and the output of each fragment is treated as just as important within each subepoch window. All of these expected errors are thankfully computable at query time and can be used to retroactively adjust the query-time importance of fragments. The subepoch configuration itself can not be modified retroactively, but one could apply weighting to the output of each fragment

during composite generation and querying. A naive solution is to replace the per-subepoch-window median calculation (see Figure 3.10) with a weighted median. Query-time re-equalization techniques warrant a deeper investigation and are left as future work.

3.7.6 F_2 Heuristic for Spatiotemporal Disaggregation

Spatiotemporal disaggregation, as implemented in DiSketch, extrapolates from the traffic load of historical epochs to configure the upcoming epochs. For this, I approximate the second frequency moment $\widehat{F_2}$ according to Equation 3.4. However, this heuristic is developed under the assumption that a sketch has monitored an entire epoch and continuously monitored every traffic flow during this time. This is not the case in spatiotemporal disaggregation. Instead, counters are periodically reset within each epoch, and the set of monitored flows is different within each subepoch. Therefore, the accuracy of this heuristic is impacted. The F_2 accuracy needs to be investigated, and the heuristic itself would potentially benefit from a modification to take spatiotemporal disaggregation into account. I leave this as future work.

3.8 Chapter Summary

In this chapter, I addressed the first research objective of my dissertation: "Make Sketches Network-wide Deployable". To achieve this, I developed and presented *spatiotemporal disaggregation*, a novel technique for deploying sketch-based data structures across multiple network switches. This technique leverages network-wide resources to enhance estimation accuracy, provide failure resilience, and reduce sketch extraction delays. I exemplified spatiotemporal disaggregation by applying it to Count Sketch (CS), leading to the development of DiSketch.

By fragmenting and distributing sketches over multiple nodes, DiSketch efficiently handles the inherent challenges posed by heterogeneous environments, such as varying resource availability and traffic loads. The combination of spatial and temporal indexing through subepoching ensures that each fragment operates within its capacity while maintaining high accuracy and reliability in flow size estimation. My solution ensures the autonomous operation of fragments, avoiding unreasonable assumptions and overheads for inter-switch communication.

My evaluation of DiSketch demonstrated significant improvements in estimation accuracy, reducing errors by up to an order of magnitude compared to traditional monolithic and naively disaggregated sketches. Additionally, disaggregation showed high resilience to network failures, maintaining accurate monitoring even during critical events.

Chapter 4

Lightweight Sketching and Flow Extraction

This chapter presents my collaborative research on sketch processing techniques for extremely low-error flow size estimation and flow-ID extraction under severe memory constraints, such as in network switches. This work addresses the second research objective, "Improve the Cost vs Accuracy Tradeoff in Sketches", by optimizing per-sketch memory utilization.

To achieve this, I introduce Flow Lightweight Detection and Ranging (FlowLiDAR), a novel solution capable of tracking nearly all network flows while requiring only a modest amount of data plane memory. These methods enhance the usefulness of probabilistic monitoring techniques by providing more reliable measurements for the control loop.

While this project overlaps with the previous chapter on sketch disaggregation, the focus here is on maximizing efficiency within individual sketches rather than deploying them network-wide. Both techniques can be applied simultaneously with minor intercompatibility changes, offering a comprehensive approach to sketch-based network monitoring. Although this work implicitly targets data center networks, it is likely applicable in broader contexts.

Attributions The research presented in this chapter is the result of collaborative efforts. As such, determining the originator of individual parts can be challenging. Nevertheless, the following list outlines some of the individual contributions that I made to the work in this chapter, excluding general contributions such as discussion participation and narrative/writing contributions:

- I provided unique expertise on the hardware and its limitations, contributing significantly to all on-switch algorithmic- and system designs to ensure real-world compatibility.
- I developed the hardware prototype and conducted hardware evaluations.
- I designed and created all evaluation figures. This work includes requesting supplementary simulation experiments to be performed when results were inconclusive or incomplete.

4.1 Introduction

The previous chapter provided a robust solution for network-wide sketching, resulting in a deployment strategy for highly accurate and flexible sketch-based monitoring. Although the resource costs are effectively spread across multiple nodes, the overall in-network resource footprint remains high. Further, the chapter did not investigate efficient methods of extracting the actual queryable keys (e.g., flow IDs). As will become apparent, key extraction is an open issue where improvements have a great potential to reduce the overall in-network costs.

To reduce this cost, the research community is proposing various solutions [91, 133, 222]. Some rely on probabilistic data structures with bounded errors to store counters and only track the flowIDs for heavy flows (e.g., ElasticSketch [222]) so to enable the reconstruction of *(flowID, counter)* tuples. The problem is that they can suffer unacceptable inaccuracies when required to track *short flows* [91]. Others encode flowIDs and associated counters directly in the Application Specific Integrated Circuit (ASIC) (e.g., FlowRadar [133]) or adopt signal-processing techniques to limit the amount of resources to be used (e.g., NZE [91]). Although they can potentially track all flows in the network, they experience a loss in accuracy when fine-grained flow-level telemetry (i.e., flow IDs more specific than the standard 5-tuple) is needed. An alternative approach in this scenario is to send the flowIDs to the control plane and keep only the counters in the data plane as done in the PR-sketch [183]. This makes the data plane memory independent of the flowID size. However, the data plane memory needed for the flow-detection filter and counters is still large, limiting the number of flows that can be feasibly monitored at high accuracies.

In this chapter, I present Flow Lightweight Detection and Ranging (FlowLiDAR), a new solution that can track *nearly all flows in the network*. As in PR-sketch, FlowLiDAR decouples flowIDs from their associated counters. FlowLiDAR introduces key innovations over the PR-sketch design that allow it to significantly reduce the amount of data plane memory needed to achieve close to 100% accuracy in per-flow estimations. For example, FlowLiDAR

uses an exact equation solving in the data plane to extract the size of the flows from the counters at the end of a measurement epoch. I also propose a new mechanism, named *lazy updates* that eliminates the need to use counters in the on-switch ASIC for short flows of up to a few packets. This not only reduces the data plane memory required but also reduces the complexity of the equation solving and improves its accuracy. I implemented FlowLiDAR in P4 and evaluated it on the same real-world packet trace as Disaggregatable Sketch (DiSketch). As shown further down in the chapter, FlowLiDAR improves the accuracy of flow counting when compared against state-of-the-art solutions such as NZE, the PR-Sketch, and Elastic Sketch by orders of magnitude. Moreover, while FlowLiDAR successfully tracks 98.7% of existing flows, other techniques can only reconstruct at most 60% of them under there same resource restrictions.

The main contributions in this chapter are:

- I introduce FlowLiDAR, which decouples flowIDs from their associated counters, significantly reducing memory usage in the data plane.
- I develop the *lazy updates* mechanism, a Bloom filter variant that further reduces memory requirements and complexity by not using counters for short flows.
- I demonstrate the superior accuracy and efficiency of FlowLiDAR through an extensive comparative evaluation.
- I validate the real-world deployability of FlowLiDAR through a hardware prototype, including algorithmic modifications that ensure compatibility with the high-speed Protocol Independent Switch Architecture (PISA) architecture.

4.2 Motivation

Tracking as many network flows as possible is of paramount importance [91, 133, 190, 213]. This is because it allows us to capture events that otherwise

would be easily missed. These events include transient loops, blackholes, and switch faults (operators report that table corruptions lead to packet losses and incorrect forwarding for some traffic [231]). Those may affect just a few packets during a very short period, introducing transient losses into the network. The problem is that even just a few losses can cause significant tail-latency increases and throughput drops for both Transmission Control Protocol (TCP) and Remote Direct Memory Access (RDMA) traffic, potentially leading to violations of Service Level Agreements (SLAs) as well as revenue loss [50]. Unfortunately, a high flow coverage rate with high precision is becoming increasingly difficult, mainly because of three main trends:

Trend #1: Networking devices now support many use-cases. The rise of programmable data planes has allowed the research community to develop a widespread number of applications including congestion control [80, 134], load balancing [5, 113], caching [108], and machine learning acceleration [177]. ***Consequence***: as more functions are added to the data plane, higher pressure is put on switches' memory and computational resources, reducing their (already limited) capabilities for flow-level telemetry.

Trend #2: Link speeds outpace memory capacity. Over the last decade, switching ASICs have increased their aggregate capacity from less than a terabit per second (Tb/s) to more than 50 Tb/s[1]. This capacity increase has enabled state-of-the-art switches to support more physical ports and higher bandwidth, reducing the number of switches required to handle the same workload as before. As a direct consequence, there is now a need to

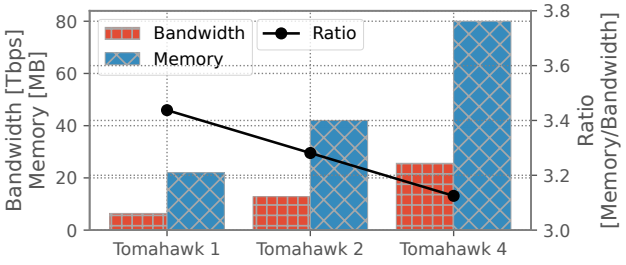


Figure 4.1: Memory/bandwidth ratio of different Broadcom Tomahawk switch generations

[1]A detailed discussion is available at https://elegantnetwork.github.io/posts/A-Summary-of-Network-ASICs/

support a much higher number of simultaneous flows. An analysis of real traffic traces has estimated the presence of *120,000 active flows per gigabit per second (Gbps) of traffic*, potentially leading to over 3,000 million active flows in 25.6 Tbps switches [178]. Unfortunately, the increase in switch speed has consistently outpaced the growth of internal ASIC memory. In Figure 4.1, I compare the aggregate bandwidth against the internal memory available in the latest generations of state-of-the-art Broadcom switches[2]. This comparison shows that the memory is not keeping up with the bandwidth.
Consequence: there is a need for more memory-efficient flow recording technology.

Trend #3: A need for longer flowIDs. The rise of virtualization, involving technologies such as VXLAN, which encapsulate network traffic to enable the creation of virtualized network overlays, and the rapid convergence of cloud-native service access APIs based on the HTTP communication protocol [11,13] are posing new challenges in flow-level telemetry. Virtualization and HTTP are distinct reasons for this increase. On one hand, virtualization through encapsulation increases the length and complexity of flow identifiers due to encapsulation. On the other hand, the increasing generality of higher-layer protocols like HTTP has led operators to explore their feasibility beyond the Web, for media streaming (e.g., WebRTC[3]), remote procedure calls (e.g., gRPC[4]), data center networking [13], or transport of DNS. When everything is encoded into HTTP, basic L2–L4-layer insight into traffic is no longer adequate to understand network behavior [11]. The traditional five-tuple becomes ineffective when the destination port is uniformly 80 (HTTP) or 443 (HTTPS) regardless of whether the traffic is a REST API call or a long-lived media stream.
Consequence: There is a need for storing more fine-grained flow identifiers using additional packet header fields, which increases the memory requirements in the ASIC.

[2]Data are taken from https://people.ucsc.edu/~warner/buffer.html
[3]`https://webrtc.org/`
[4]`https://grpc.io/`

4.2.1 Limits of Current Solutions

To address these challenges, state-of-the-art solutions commonly use probabilistic data structures [133, 139, 222] to reduce the switch memory requirements at the cost of query accuracy. Despite their theoretical error bounds, existing solutions still suffer to provide comprehensive guarantees for *all flows* in a network. This is because existing algorithms are typically designed to provide guarantees for specific flows (e.g., heavy hitters [58, 180] or super-spreaders [226]) and/or aggregated flow statistics (e.g., cardinality [64] or traffic distribution [119]). As a consequence, when applying these solutions to the entire traffic, the derived bounds are too coarse-grained to work. This leads to a considerable gap: only a small portion of flows benefit from the theoretical bounds, while the remaining flows still exhibit poor accuracy.

A recent paper has highlighted this issue and proposed a solution [91], named Near Zero Error (NZE) which, as its name states, reduces the errors but does not eliminate them. In more detail, the accuracy for small flows is still not guaranteed, especially if many bits are needed for the flow identifiers. Therefore, the problem remains when considering trend #3: adopting increasingly longer flow identifiers. This is addressed by the PR-sketch, which sends all flowIDs to the control plane and keeps only the flow detector and counters in the ASIC [183]. However, the PR-sketch still requires a significant amount of ASIC memory per flow (see Section 4.5.3 for details), making it less efficient than existing solutions except for very large flowIDs. To better understand this, FlowLiDAR is compared against numerous state-of-the-art algorithms (i.e., Elastic Sketch [222], NZE [91],

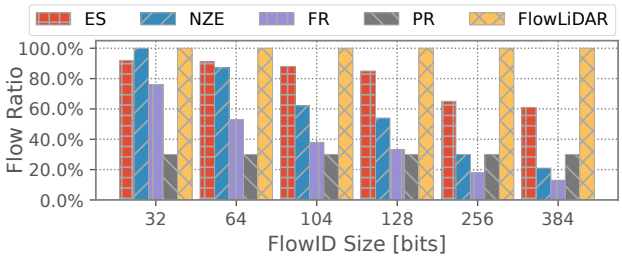


Figure 4.2: Fraction of flows that can be monitored with a fixed amount of memory for ElasticSketch (ES), NZE, FlowRadar (FR), PR-Sketch (PR), and FlowLiDAR

FlowRadar [133], and the PR-sketch [183]). In Figure 4.2, I present the fraction of flows that can be accurately tracked using a fixed memory size of 10 MB to perform size estimation of 1.2 million active flows [178].

In that experiment, NZE successfully tracks *all flows* only when flowIDs are 32 bits (e.g., flowIDs are a single IPv4 address). Longer flow identifiers must be adopted for more exact insights, which inhibits the ability of NZE to track all flows. Indeed, when using a standard 5-tuple, the flow coverage drops to just 60%. On the other hand, ElasticSketch can track only 90% of flows with high accuracy ($< 1\%$ relative error) with 32 bits, but its performance degrades more gracefully[5]. Similarly, FlowRadar is unable to track all flows for any flowID size, and its coverage is lower than both that of NZE and Elastic Sketch. When more packet header fields need to be considered, as in the case of tunneled connections requiring VXLAN + 5-tuple or when upper layer protocol headers are needed, the flow coverage drops further. For instance, when requiring 256-bits flowIDs, the flow coverage of Elastic Sketch, NZE, and FlowRadar drops to approximately 60%, 35%, and 20% respectively, which is unacceptably low. Finally, the PR-sketch flow coverage does not depend on the flowID size as expected but it is below 40%, which is worse than existing schemes for small flowID sizes. FlowLiDAR, however, is able to consistently track over 99% of flows regardless of the flowID size.

[5]Elastic Sketch uses two main data structures: a hash table for the heavy part, which is flowID size dependent, and a large count sketch to count the small flows. Here, the memory is split between the two structures, varying the flowID size to bound the average relative error below 50%.

4.3 FlowLiDAR Overview

This section gives an overview of FlowLiDAR, followed by a detailed description of each component underlying the solution.

4.3.1 Overall Approach

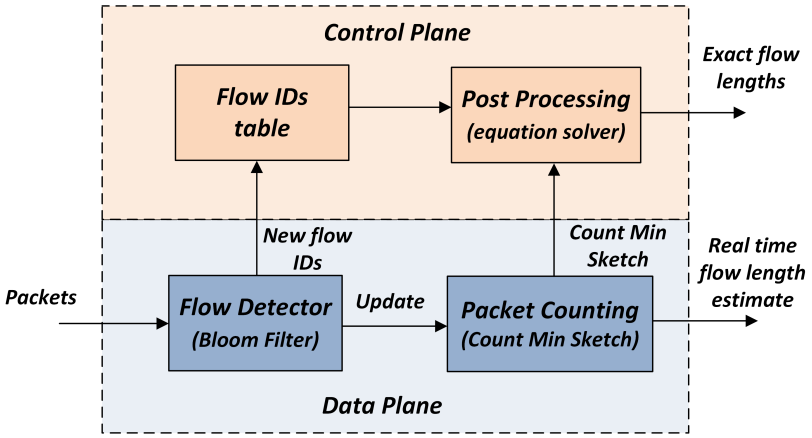


Figure 4.3: Block diagram of the proposed FlowLiDAR. The data plane detects new flows, sends the IDs to the control plane, and counts the packets. The control plane stores the flow IDs and periodically computes an exact flow frequency vector

The concept of FlowLiDAR is illustrated in Figure 4.3 and makes use of both switch data and control planes. The idea, similarly to [183] is to place all the functions that have to be done per packet in the data plane while those that are much less frequent are placed in the control plane. In more detail, flow detection and counting of packets are done in the data plane while the processing of new flows and a fine-grained computation of the flow frequencies is done by the control plane. This approach needs to send information between the control and data planes, and the interface bandwidth may be a potential issue. Further down, I discuss why this should not be the case in switching ASICs that have high-speed links between both planes and

propose variants of FlowLiDAR that can reduce the amount of information sent on that interface.

This split is based on the observation that new flows, in some environments such as campus networks, are only a small fraction of the packets [110]. This has been corroborated by analyzing the three CAIDA ISP packet traces, where flows average approximately 15 packets per minute. The results show that in a one-second window, the average number of packets per flow is single-digit which increases in larger windows. Therefore, storing flowIDs in the switch is immensely costly, and I suggest detecting new flows in the ASIC, and sending new IDs to the control plane. This eliminates the need to store the flowIDs in the limited data plane memory, dramatically reducing the memory footprint. With this idea in mind, what we need to have in the ASIC is a mechanism to detect new flows, for example through a Bloom Filter (BF) [133]. This detector has to be in the data plane to be able to handle the immense speeds of the hardware. In FlowLiDAR, the detector is a Bloom Filter (BF) that is optimized to reduce its memory footprint [87].

The same reasoning applies to the counters used to count packets, which also have to process every packet and thus also have to be placed in the ASIC. In more detail, FlowLiDAR uses a Count-Min Sketch (CMS) for packet counting [169]. This enables us to provide in-data-plane flow size estimations in real-time. Additionally, snapshots of the BF and CMS are sent periodically to the control plane, and the data structures are (mostly) reset at the onset of a sketching epoch.

The control plane stores the FlowIDs, and when it receives a snapshot of the CMS and BF it computes the exact flow frequency vector using a more complex algorithm that models the CMS as a system of equations. Again this is possible as it is done much less frequently than the per-packet operations done in the data plane. Even if extracting the exact values is the main aim of FlowLiDAR, I want to highlight that a further memory reduction is possible at the cost of accepting an approximate resolution of the CMS system of equations, and this solution is therefore able to deliver highly memory-efficient on-switch sketch. Details about this option are provided in Section. 4.3.4 and evaluated in 4.5.4. In the following subsections, I describe each of the

FlowLiDAR components in more detail, as well as the relationship between them.

4.3.2 Flow Detector

The flow detector identifies new flows and sends their IDs to the control plane. Here, I describe three methods to implement the Flow Detector. The first method is based on a standard BF and provides a baseline for the other two methods.

I developed a second BF variant, which I call a lazy BF. This variant reduces the false positive probability of the standard method at the expense of an increase in the bandwidth required by the control plane, as well as introducing dependencies between some in-ASIC computations. For the rest of the paper, when nothing else is stated, this is the flow detection algorithm used in this chapter.

We could use a third method if the control plane bandwidth becomes a bottleneck. This method only sends FlowIDs that were not present in the previous epoch toward the control plane. For this, one can use a pair of BFs, where one stores the FlowIDs of the previous epoch and the other stores those of the current epoch. In all cases, flow detection is done as soon as the first packet of the flow returns a negative on the BF and it is immediately reported to the control plane, a process that takes only a few microseconds in the worst case.

Standard BF The detection of FlowIDs is done per measurement epoch so that, after sending it to the control plane, the detector is reset to start a new epoch. To this end, the flow detector has to check all the packets and thus has to perform simple operations. As in [133], [183], FlowLiDAR uses a BF to detect new flows. The overall approach is to update the BF with every new packet (so that subsequent packets of the flow return a positive) and send the FlowID to the control plane if this is the first time that the flow was seen by the BF. Using a BF eliminates the need to store raw FlowIDs, thus reducing the memory requirements. However, it has the drawback of

risking false positives, resulting in a few flows not being detected and sent to the control plane. For a given number of target flows, the probability of false positives can be made small by appropriately selecting the BF array size m and the number of arrays and hash functions k. The initial flow detection algorithm is shown in Algorithm 2.

Algorithm 2 Initial algorithm for flow detection

```
1: Reset the BF
2: Start the epoch timer
3: for Each packet with FlowID x do
4:     Query element x in the filter
5:     if Negative then
6:         Add x to the filter
7:         Send FlowID to the control plane
8:     end if
9:     Send x to packet counting block
10:    Timer expired? If yes restart the process
11: end for
```

The BF maps each FlowID to k independent arrays of m bits. The use of independent arrays per hash function makes it possible to access each position in parallel and facilitates the implementation in a programmable data plane as will be seen in section 4.4. Additionally, it enables a more advanced flow detection scheme, which I implemented in FlowLiDAR, and is described next.

The fraction of flows that are not detected during an epoch can be estimated by computing the false positive probability of the filter when each new flow arrives and then adding all those probabilities together. The false positive probability of a filter that has k arrays of m bits and on which i elements have been inserted can be approximated by:

$$P(i) \approx (1 - e^{-\frac{i}{m}})^k \tag{4.1}$$

Then if on an epoch there are n flows, the fraction of false positives can be estimated by adding the probabilities of the second flow $P(1)$, the third flow $P(2)$, and so on until the n^{th} flow obtaining:

$$FPP \approx \frac{\sum_{i=2}^{n} P(i-1)}{n} \tag{4.2}$$

Lazy updates BF The advanced BF scheme, which I denote as lazy updates BF, is based on the observation that depending on the epoch duration, most flows only have a single or just a few packets in that period. To maximize sketching accuracy, a short epoch is preferred, and the main limiting factor is the speed at which the structure can be exported and analyzed. For instance, Table 4.1 presents the percentage of flows containing only one, two, or three packets within a one-second epoch, based on the CAIDA traces.

1-packet	2-packets	3-packets	more than 3 packets
39%	18%	10%	33%

Table 4.1: Percentage of flows having one, two, or three packets

When that is the case, it may be beneficial not to set all the bits for the new flow in the BF to one but just one at a time. This would reduce the number of positive bits in the filter, thus reducing the false positive probability. For example, if 40% of flows only have a single packet, this would be reduced by close to 40% the insertions on the second to k^{th} arrays. Using independent arrays is better in this configuration to reduce the false positive rate similar to what happens in d-left hashing [141]. Deriving the false positive probability for flows in this advanced scheme is more complex but an approximation can be easily obtained. Let us denote by $l(j)$ the fraction of flows that have j or more packets in the epoch. Consider a filter with k arrays on which i elements have been inserted. Then in the j^{th} array, there will be approximately $i \cdot l(j)$ elements inserted. With that assumption, the false positive probability of the filter can be computed as the product of the fraction of bits set to one in each array which is given by $(1 - e^{-\frac{i \cdot l(j)}{m}})$ obtaining:

$$P_a(i) \approx \prod_{j=1}^{k} (1 - e^{-\frac{i \cdot l(j)}{m}}) \tag{4.3}$$

where $l(j)$ accounts for the fact that a fraction of the flows has j or less packets and thus are not inserted on tables $j+1, \ldots, k$ unless they suffer false positives on the previous tables. This approximation would tend to underestimate the false positive probability as there will be false positives. For example, a flow with a single packet may find the bit set on the first BF table and would thus be inserted on the second and so forth. Finally, the fraction of flows that are false positive can be estimated by using $P_a(i)$ instead of $P(i)$ in equation (4.2).

The drawback of using this advanced scheme is that multi-packet flowIDs may be sent to the control plane several times. Let us consider the bandwidth needed to send the FlowIDs to the control plane. Each FlowID has 13 bytes in IPV4[6]. On average the number of packets per flow is much larger than one, for example even when considering one-second windows, the CAIDA traces have on average multi-packet flow. Considering an average packet size of 500 bytes, the overhead would be below 13/(500*5) so roughly 0.5% which would be acceptable in most cases. Indeed as discussed before, bandwidth grows faster than on-chip memory and thus using a small fraction of the bandwidth in exchange for a large reduction in the memory needed (as FlowIDs no longer need to be stored on-chip) is attractive.

Finally, we can also take advantage of this BF optimization to not send the first packets to the packet counting block. This has the additional benefit of reducing the load on the CMS as we will see shortly. The advanced flow detection is presented in Algorithm 3. Note that although the algorithm describes a serial implementation, the for loop over the k arrays has no temporal dependencies and thus can be unfolded and executed in parallel with each value of i corresponding to one of the filter arrays.

[6]The FlowID is composed of the source and destination IP addresses, the protocol, and the source and destination ports.

Algorithm 3 Advanced algorithm for flow detection using lazy updates for the BF

```
1: Reset the BF
2: Start the epoch timer
3: for Each packet with FlowID x do
4:     for i = 1 to k do
5:         if h_i(x) == 0 then
6:             Set h_i(x) = 1 and i = k
7:             Send FlowID to the control plane
8:             break
9:         else
10:            if i == k then
11:                Send x to packet counting block
12:            end if
13:        end if
14:    end for
15:    Timer expired? If yes restart the process
16: end for
```

Differential Flow Detector with a pair of BFs As discussed before, the control plane bandwidth is not expected to be a bottleneck in switching ASICs. However, there may be other architectures in which it may be an issue. For those cases, one option is only sending the FlowIDs that are not present in the previous epochs to the control plane. This way, only the new flows are sent to the controller, while the old ones are retrieved from the snapshot taken in the previous epoch. In particular, in this configuration, the Flow Detector uses two BFs. In the first BF, called $oldBF$ all the FlowIDs of the previous epoch have been inserted. This BF is checked when a packet arrives to avoid sending a FlowID that is already in the snapshot stored in the control plane. The second BF, called $currentBF$, works as the standard BF and stores all the FlowIDs that have packets in the current epoch. A FlowID is sent to the controller only if it is not present in both BFs. At the end of a measurement period, the set of active FlowIDs can be retrieved by merging the FlowIDs sent in the current epoch and the FlowIDs of the previous snapshot that are still active. These can be extracted from the previous snapshot by checking which ones are positive in the $currentBF$ indicating that with high probability they are still active. At the end of the measurement epoch, the $oldBF$ stores the content of the $currentBF$, and the $currentBF$ is reset. The differential flow detection is presented in Algorithm 4. I want you to note that the use of a BF pair has been explored in previous literature, but mainly to avoid filter overload [22, 142].

4.3.3 Packet Counting

The other data plane component maps each flow to several arrays of counters and increments one counter per array as in a CMS. This enables a fast estimation of the flow size by just taking the minimum of those counters.

Additionally, the CMS contents are sent periodically to the control plane for further analysis. In that analysis, it is beneficial to have as many counters with a value of zero as possible. To achieve this, I exploit the BF used to detect new flows as a counter for the first packets (those sent to the control plane) so that only flows with more than one (or a few if we use the lazy

Algorithm 4 Algorithm for flow detection using a pair of BFs

```
1: copy currentBF into oldBF
2: reset currentBF
3: Start the epoch timer
4: for Each packet with FlowID x do
5:     Query element x in the currentBF
6:     if Negative then
7:         Add x to the currentBF
8:         Query element x in the oldBF
9:         if Negative then
10:            Send FlowID to the control plane
11:        end if
12:    end if
13:    Send x to the packet counting block
14:    Timer expired? If yes restart the process
15: end for
```

version) packets in the epoch use the counters. This has a large benefit as a significant fraction of the flows no longer need a counter.

In contrast to a standard CMS, FlowLiDAR splits the CMS into a set of smaller CMS, each indexed by a master hash function. This choice will introduce a somewhat increased error but permits splitting the flow analysis (presented in the next section) to a set of disjoint sparse linear systems, thus providing a significant analysis speed-up compared to packet counting using a traditional non-split CMS. In essence, this is a tradeoff between accuracy and offline processing costs.

4.3.4 Postprocessing

I did not participate directly in the development of the analysis solution presented in this subsection. However, I am including it here to grant the reader a complete view of the work, as well as the full context of my contributions.

The control plane continuously collects FlowIDs from the ASIC, and at

the end of the epoch receives the populated BF CMS. Using these, the control plane can compute the exact values of the flow sizes. This is denoted as postprocessing of the information and is done in three stages, some of which are preprocessing for the final stage in the postprocessing: *BF preprocessing*, *CMS preprocessing*, and *CMS equations solving*.

BF Preprocessing

An interesting observation is that when lazy updates are used, the BF can be used to identify a fraction of the flows. In more detail, the FlowIDs collected on the control plane can be tested on the snapshot of the BF received from the data plane, and those that return a negative have for sure not been added to the CMS. This means that their number of packets corresponds to the number of times that the flowID has been received in the control plane. Therefore, the exact packet frequency is obtained for those flows. Additionally, we can remove them from further consideration, thereby simplifying the problem. The BF preprocessing is described in Algorithm 5.

Algorithm 5 BF preprocessing with lazy updates

```
1: Compute the set of distinct flows D encountered by the data plane during the
   epoch.
2: Create an empty set C for flows to be processed in the CMS.
3: for each FlowID x in D do
4:     Query element x in the BF
5:     if Negative then
6:         Set the number of packets x as the number of times it
           was received in the control plane.
7:     else
8:         Add x to C
9:     end if
10: end for
11: Use set C in the CMS preprocessing step.
```

CMS Preprocessing

Similarly, for any flow that maps to a counter in the CMS with a value of zero, we can get the exact number of packets by counting the number of times that the flowID has been received in the control plane. In the case of a "classic" BF, this is at most one packet, while the lazy BF could be interpreted as multiple non-counted packets. The flow is known to have more packets than the CMS has counted since the flowID has been sent to the control plane after a return-negative in the BF where CMS is not updated. Again, these flows can be removed from further consideration. The CMS preprocessing is described in Algorithm 6.

Algorithm 6 CMS preprocessing

```
1: Get set C of FlowIDs from the previous preprocessing step.
2: for each FlowID x in C do
3:     Query element x in the CMS
4:     if estimate packet count == 0 then
5:         Set the number of packets x as the number of times it
           was received in the control plane.
6:         Remove x from C
7:     end if
8: end for
9: Use set C in the CMS equations solving step.
```

Exact CMS equation solving. Let now $\mathcal{F} = \{f_1, \ldots, f_n\}$ be the set of FlowIDs for which the packet count could not yet be identified. Assume for now that there are no false positives in the BF and hence $\mathcal{F}$ is fully known in the control plane (the effect of false positives is commented below). Moreover, we have access to the vector $\mathbf{b} = [b_1, \ldots, b_m]^T$ of counters stored in the CMS. The packet-count for flow f_j is the number of times that f_j has been received in the control plane plus the number x_j of times that f_j has been added to the CMS. We write $\mathbf{x} := [x_1, \ldots, x_n]^T$ as a vector. The challenge is to compute $\mathbf{x}$ given $\mathbf{b}$, $\mathcal{F}$ and the k hash functions $h_1, \ldots, h_k$ used in the CMS.

This problem can be captured by a system of linear equations

$$A\mathbf{x} = \mathbf{b} \tag{4.4}$$

where $a_{ij} \in \{0, 1\}$ indicates whether $i \in \{h_1(f_j), \ldots, h_k(f_j)\}$, i.e. whether flow f_j has contributed to the counter b_i. Here it is assumed that $h_1(f_j), \ldots, h_k(f_j)$ are pairwise distinct such that a flow cannot contribute multiple times to the same counter.

The most immediate question is whether $\mathbf{x}$ is uniquely determined by equation (4.4) or whether there exists $\mathbf{x}' \neq \mathbf{x}$ with $\mathbf{b} = A\mathbf{x} = A\mathbf{x}'$. This is equivalent to the existence of $\mathbf{x}'' \neq 0$ with $A\mathbf{x}'' = 0$, which exists if and only if the columns of A are linearly dependent.

Assuming that the k hash functions behave like fully random functions, the columns of $A \in \{0, 1\}^{m \times n}$ are *stochastically* independent and contain exactly k ones per column in uniformly random positions. Such matrices have been studied in the literature on cuckoo hashing and random boolean formulas [52, 55, 170]. Sharp threshold behavior concerning the load factor $c = \frac{m}{n}$ has been demonstrated. More precisely, there exists a constant $c_k^* \in (0, 1)$ such that the following holds. For any $c < c_k^* - \varepsilon$ the matrix A has linearly independent columns with probability $1 - n^{-\Omega(1)}$, even when $\mathbb{F}_2 = \{0, 1\}$ is used as the underlying field [52, 55, 170] (this implies independence for underlying fields $\mathbb{Q}$ and $\mathbb{R}$). For any $c > c_k^* + \varepsilon$ there exists with probability $1 - n^{-\Omega(1)}$ a set C of columns of A and a set R of rows of A with $|C| > |R|$ such that all 1-entries within C are within R [52, 67]. This precludes the independence of the column set C over any field. The threshold values are reproduced in Table 4.2.

k	3	4	5
c_k^*	0.918	0.977	0.992

Table 4.2: Thresholds for CMS equation solving.

The behavior seen in a simulation with 100 trials with $k = 3, 4$ and CMS size of $m = 2K, 4K, 8K, 16K$ as shown in Figure 4.4 matches these asymptotic predictions quite well.

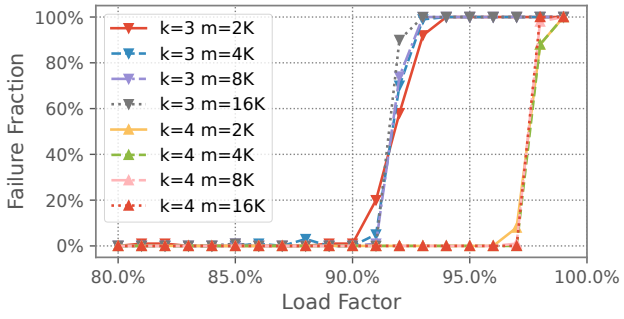


Figure 4.4: Percentage of full-rank matrices with different load factors and k values

Note that the complexity of solving equation (4.4) is super-linear in m. To improve running times for large m, one could attempt to adopt variants of structured Gaussian elimination as was done in [70] for solving equation (4.4) over finite fields. Alternatively one could use a smaller load factor exploiting that below the so-called *peeling threshold* equation (4.4) can be solved in linear time over any group with probability $1 - n^{-\Omega(1)}$ [83, 153].

The algorithm relies on a splitting hash function and solves a set of several small systems, instead of a single bigger system solving the various systems in parallel, taking advantage of Central Processing Unit (CPU) multi-core architectures. In the evaluation section, I will show how this split reduces the computation time.

On the issue of false positives. In the case where at least one false positive has occurred in the BF, we only know a subset $\mathcal{F}_1 \subset \mathcal{F}$ of the remaining FlowIDs, and we do not know $\mathcal{F}_2 = \mathcal{F} \setminus \mathcal{F}_1$. The underlying equation is then $A_1x_1 + A_2x_2 = b$ where A and x are sliced into two parts relating to $\mathcal{F}_1$ and $\mathcal{F}_2$. With no knowledge of A_2 there is no hope of recovering the counts x_2 for $\mathcal{F}_2$. There are, however, several methods for approximately recovering the counts x_1 for $\mathcal{F}_1$ under reasonable assumptions. In an insightful paper by Ting [203], $\epsilon := A_2x_2$ is modeled as a random error vector and our task is to find x_1 that maximizes the likelihood of $\epsilon = b - A_1x_1$. If we assume that the $|\mathcal{F}_2|$ entries of ϵ are independently sampled from a log-concave distribution D, then we obtain a convex optimization problem. For instance,

if D is assumed to be a normal distribution, then we recover the linear least squares method where $||A_1 x_1 - b||^2$ is to be minimized. This method already yields decent results in practice [127] despite the unfounded assumption on D (e.g. a normal distribution does not guarantee $\epsilon \geq 0$). For even better accuracy, Ting [203] proposes methods for estimating D based on those entries of b to which no key in x_1 has contributed.

In my implementation, I compute an approximate value of x_1, called $\hat{x_1}$. We have $\hat{x_1} = A_1^{-1} b$. The approximation error is $e = \hat{x_1} - x_1 = A_1^{-1} \epsilon$. Since FlowLiDAR targets a small False Positive Rate (FPR), there is a small value of $||\epsilon||$, since only a few elements of the vector ϵ are different from zero, thus it is expected that also $||e||$ will be small. This is confirmed by the experiments, as will be shown later in the chapter's evaluation.

Furthermore, it is always possible to compare the solution $\mathbf{x}$ provided by the equation solver to the minimum among the k rows of the CMS corresponding to the x_i variable, choosing the minimum among these two values. This guarantees that the error due to the BF will be similar to the approximation of the traditional CMS in the worst case.

Finally, splitting the system into a set of smaller independent systems further alleviates this problem. In fact, for most of the subsystems, the occurrence of false positives has a negligible impact on the overall error, while for the few subsystems in which this error is significant, the result is bound to those achieved by the traditional CMS approximation.

Approximate CMS Equations Solving

If the rank r of the matrix is less than the number of variables n the system is underdetermined and has multiple solutions. In particular, the system has several free variables that are $l = n - r$. In the following, an algorithm is described that selects the l free variables that minimize the absolute error. This algorithm can be used when the exact CMS equation solving fails. The algorithm, presented as Algorithm 7 selects the system equations with the smallest constant terms b_i and imposes as value of the corresponding variables b_i/n_i, where n_i is the number of variables appearing in the i-th

system equation. Fixing the value of these unknowns corresponds to setting some additional rows to the **A** matrix. The procedure is repeated until the sum of the n_i fixed variables reaches the value of l. The algorithm reduces the underdetermined problem to a linear system with exactly one solution, which can be solved using the standard method used to resolve the sparse linear system of equation (4.4).

Algorithm 7 Algorithm for the selection of the free variables

1: Sort the vector **b** in ascending order
2: **for** Each b_i **do**
3: Get the variables $x_a, ..x_b$ corresponding to row i of the matrix **A** that differs from 0
4: set the values of $x_a, ..x_b$ to b_i/n_i
5: set $l = l - n_i$
6: **if** $l == 0$ **then**
7: break the loop
8: **end if**
9: **end for**
10: Solve the $\mathbf{Ax} = \mathbf{b}$ system with the additional equations given by row 4.

This algorithm will select one of the possible solutions for solving the linear system, but it can not be sure that this is the actual distribution of the number of packets per flow. However, I will show in the FlowLiDAR evaluation that the average absolute error

$$AAE = \frac{1}{n} \sum_i |x_i - \hat{x_i}|$$

and the average relative error

$$ARE = \frac{1}{n} \sum_i \frac{|x_i - \hat{x_i}|}{x_i}$$

obtained using this algorithm are better than the Average Absolute Error (AAE) and Average Relative Error (ARE) obtained both using the least square method proposed by PR-sketch and the standard CMS algorithm (that for the variables x_i takes the minimum among the buckets addressed by f_i).

Version	Pipeline Stages	SRAM	sALU	TCAM	Hash Bit
FlowLiDAR	2	5.1%	11.3%	0.3%	2.7%
Lazy FlowLiDAR	3	5.4%	11.3%	0.3%	3.1%

Table 4.3: Resource footprint imposed by FlowLiDAR in Tofino 2. These numbers are based on a 4x128K BF, and 64 5x1K 16-bit CM sketches

4.4 Implementation

I implemented FlowLiDAR in 500 lines of $P4_{16}$ code[7] for Tofino 2, using the Barefoot SDE 9.7 [2]. The results are obtained using a 4x128K BF, with 64 count-min sketches each containing 5 rows of 1K 16-bit counters. The resource costs, as shown in Table 4.3, are relatively modest and leave plenty of room for any co-located functionality at the switch.

Note that my prototype uses a relatively high number of stateful ALUs. This is because an efficient implementation without recirculation necessitates a dedicated stateful ALU for each of the 9 hash functions used for the combined BF and CMS dimensions. Here, the indexing in the BF and CMS and the selection of which sketch to apply are all based on the switch-native Cyclic Redundancy Check (CRC) engine, using custom polynomials that are statically configured at compile-time for a high level of independence between the hash functions.

Finally, it is worth noting that the lazy version of FlowLiDAR requires more stages (+3 compared to +2 stages). This is due to the introduction of a strict dependency between the BF-bits, which forces the compiler to stretch the BF across several hardware stages.

This implementation demonstrates that the design is compatible with the high-throughput PISA architecture, proving it can effectively operate within high-speed pipelined network switches.

[7]The P4 source code and the simulator are available at this link: `https://github.com/FlowLidar/FlowLidar`.

4.5 Evaluation

A software simulator has been developed for evaluating FlowLiDAR. The simulator reproduces the behavior of the hardware prototype while providing more flexibility in terms of the number of used hashes and sizing of data structures, thus providing better insights into the behavior of FlowLiDAR. The code has then been used to monitor the flows in three CAIDA traces using FlowLiDAR with different configuration parameters. In particular, I consider three different traces taken from the 2016 dataset [30]: (C1) 21/01/2016 Minute 13:00, (C2) 18/02/2016 Minute 13:30, and (C3) 17/03/2016 Minute 14:00. The characteristics of the three CAIDA traces are reported in Table 4.4. Before proceeding to discuss the results, it is important to note that the relative performance of the different sketches will be similar when the link speeds increase from the 10G of the CAIDA traces to faster links such as those currently used in modern data centers.

Although ISP traffic, such as CAIDA, is not a perfect representation of data center environments, I need extensive packet traces that are only available from ISP traces to saturate the data structures. Please refer back to Section 3.6 in the previous chapter for a more in-depth explanation of this choice and its drawbacks.

short name	duration	# of pkts	# of flows	average bit rate	average packet size
C1	60 sec	31M	905K	2.1 Gbps	509B
C2	58 sec	31M	781K	3.1 Gbps	722B
C3	60 sec	34M	579K	4.1 Gbps	898B

Table 4.4: Characteristics of the CAIDA traces used in the evaluation

In the first experiment, I use one-second sketching epochs, a BF with four arrays of 128K bits, and a CMS with 64 arrays of 1K counters of 16 bits. The basic scheme and the lazy update are evaluated and the results are shown in Figures 4.5-4.12.

The plots report the percentage of flows with exact results per epoch,

the percentage of flows not detected due to False Positives (FPs), and the bandwidth to the control plane. It can be noticed that FlowLiDAR with the standard BF can exactly (with no error) estimate more than 80% of flows. Small-scale experiments show that for 95% of flows, the error is less than or equal to 1. Consequently, the average absolute and relative error of FlowLiDAR are well below those of the traditional CMS evaluation (see Figures 4.7-4.8). The fraction of flows with an error greater than one is due to the pollution caused by the untracked flows (Figure 4.6), which are the flows not detected due to a false positive in the BFs. The plots also show that lazy updates greatly reduce the rate of false positives (Fig, 4.6). This improves both the fraction of flows with zero error, which approaches 100%, and the average absolute and relative errors (Fig, 4.7-4.8).

The drawback of the use of lazy updates is the additional required bandwidth, which grows from 60K flowIDs per epoch of the standard BF to 130K flowIDs per epoch (Fig, 4.12).

4.5.1 Benefit of Lazy Update BF

Since the lazy updates have an additional margin, this section presents the same experiment but reduces the CMS to half, using 32 arrays of 1K counters of 16 bits. The lazy update with 32x1K and 64x1K CMSs are compared and the results are shown in Figures 4.9-4.10. In this case, the experiment show that the use of lazy update enables decreasing the size of the CMS without affecting the quality of the results. Also with the 32x1K CMS, it is possible to achieve nearly 100% of the exact results.

In the second experiment, I explore the parameters of the lazy update. The same experiments are performed while changing the value of the BF and CMS parameters. In particular, three configurations (4x128K,6x64K,8x32K) are selected for the lazy update BF and two configurations (32x1024, 64x512) for the CMS. The parameters are chosen to get insight on the compromise between the bandwidth required by the lazy BF and the memory saving on the lazy BF due to the fewer number of bits set to 1 in the lazy BF for the flows with less than k packets. Table 4.5 reports the FPR and the required

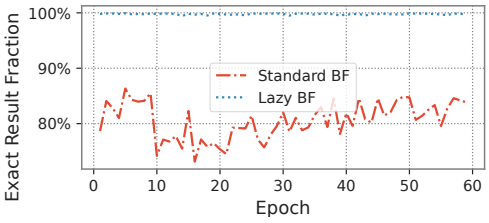


Figure 4.5: Percentage of flows with exact result

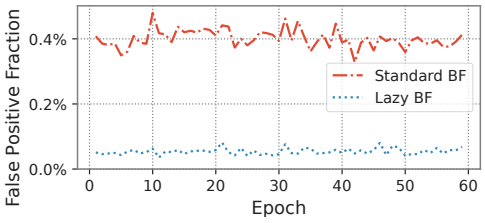


Figure 4.6: Flows not detected due to FPs in the BF

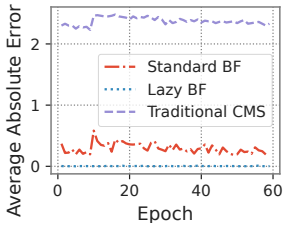


Figure 4.7: Average Absolute Error

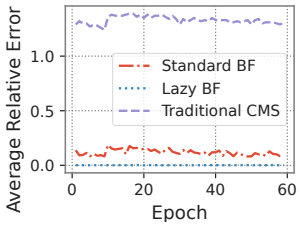


Figure 4.8: Average Relative Error

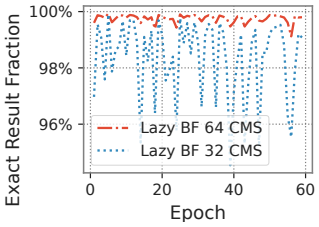


Figure 4.9: Percentage of flows with exact result for 32 and 64 CMS and lazy BF

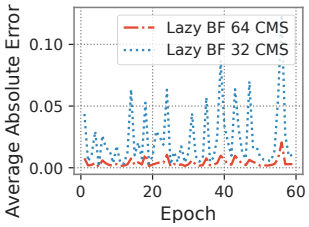


Figure 4.10: Average Absolute Error for 32 and 64 CMS

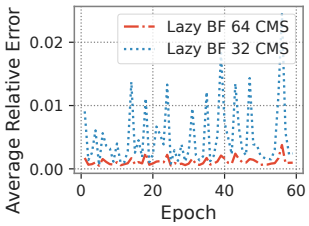


Figure 4.11: Average Relative Error for 32 and 64 CMS

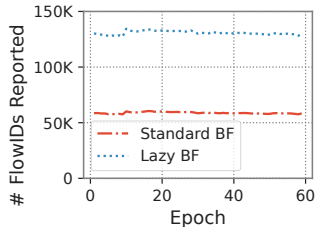


Figure 4.12: Bandwidth to the control plane

bandwidth (BW) for the three lazy BF configurations, the AAE and the ARE both for the 32x1024 and 64x512 CMS configurations. The results show that the lazy update can save 25% of memory (from the 512Kbits of the 4x128K to 384Kbits of the 6x64K) with a negligible penalty for the false positive rate, a 20%-25% of BW overhead and a better value of the AAE and ARE both for the 32 and the 64 CMS configurations. With the 8x32K BF FlowLiDAR saves 50% with a BW overhead between 25%-35% and a slightly worse AAE. As mentioned earlier, this is a tradeoff between accuracy and bandwidth load that has to be decided by network engineers and operators during system deployment.

metric	k	C1 trace	C2 trace	C3 trace
FP	4	0.0563%	0.0473%	0.0052%
	6	0.0300%	0.0250%	0.0013%
	8	0.1739%	0.1438%	0.0029%
BW (# of flows)	4	130K	129K	72K
	6	154K	151K	88K
	8	163K	160K	97K
AAE (32x1K CMS)	4	0.0194	0.0137	0.00043
	6	0.0031	0.0021	0.00006
	8	0.0202	0.0128	0.00013
AAE (64x512 CMS)	4	0.0176	0.0127	0.00039
	6	0.0030	0.0020	0.00006
	8	0.0186	0.0121	0.00014
ARE (32x1K CMS)	4	0.0052	0.0037	0.00009
	6	0.0008	0.0005	0.00002
	8	0.0060	0.0039	0.00003
ARE (64x512 CMS)	4	0.0047	0.0035	0.00008
	6	0.0007	0.0005	0.00002
	8	0.0057	0.0037	0.00003

Table 4.5: Analysis of lazy update benefit

4.5.2 Bandwidth and Epoch Resolution

One of the possible issues of the FlowLiDAR approach is the need to send to the controller a significant amount of data. In particular, for each epoch it is

necessary to send to the controller: (i) all the active flows detected by the BF and, (ii) the snapshot of the CMS stored in the data plane. Furthermore, if advanced strategies based on the Lazy updates BF or the BFs pair for differential flow detection are used, also (iii) the snapshot of the BFs must be sent to the controller. Even if the mechanisms to forward these data to the controller can be different, we can assume that they use the same communication channel such for example a PCIe connection between the ASIC and the on-switch CPU. It is thus important to understand how much bandwidth is required and how the use of different epoch lengths affects this bandwidth. A smaller epoch period will provide a better resolution of the network snapshot and thus should be preferable. On the other side, the BW will be directly proportional to the number of epochs in a second, thus at first look, a too-short epoch period could saturate the available bandwidth. However, it is also worth noticing that in a shorter period, there will be fewer active flows, thus less pressure on the communication channel. Moreover, a smaller number of flows also permits us to reduce the size of both the BF and the CMS, thus allowing us to significantly reduce the overall amount of data to send to the controller. To summarize, smaller structures can be extracted quickly, while larger structures need a longer time to be extracted, and therefore a longer epoch duration.

A set of experiments was performed using the three aforementioned CAIDA traces to get an insight into the relationship between bandwidth and epoch duration. In particular, 10 epoch periods are selected, distributed with an exponential scale between 1 ms and 1 second (*i.e* with values 1 ms, 2 ms, 4 ms, 8 ms, .. 1024 ms). Also, the size of the BF was scaled in the same way, starting from a BF size of 1Kb and doubling the size at each step. This choice allows a fixed contribution to the bandwidth that is of 1Mb/sec $\approx$ 128KB/sec, which is fairly small. With the above configurations, the FP rate is around 2% for the differential BF, less than 1% for the standard BF, and less than 0.05% for the lazy updates. Figure 4.13 shows the results in terms of FP rate for the three options taken into account.

The second contribution is due to the size of the CMS. If we aim to achieve an exact solution, the CMS load should be less than 0.97 when k=4, thus the

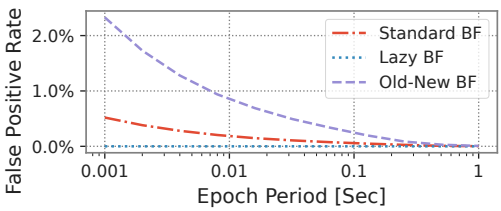


Figure 4.13: False positive rate with different BF as a function of the epoch period

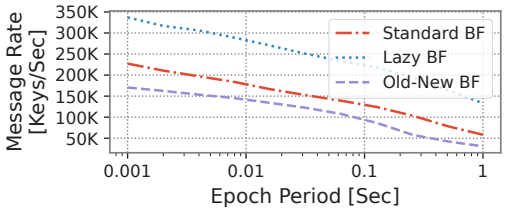


Figure 4.14: Number of flows sent to the controller per second as a function of the epoch period

number of counters in the CMS should be slightly greater than the number of active flows. Considering a 2B counter, in the best case in which we almost fill the CMS up to 97%, the required bandwidth is directly proportional to the number of flows sent to the controller. The third contribution is simply the number of active flows that are sent to the controller.

In Figure 4.14 I report the number of flows sent to the controller. It is possible to see that, as expected the number of flows sent per second decreases for higher epoch periods, but also if we want to run FlowLiDAR at a high resolution of 1 ms, the amount of data to send to the controller is still manageable. From the above data, we can estimate the overall required bandwidth as follows. If we suppose that the FlowID is 16B, such as the standard 5-tuple of 104 bits plus some additional information, and considering the 2B for each CMS counter, we can estimate the overall bandwidth for the controller with the standard BF as $C_{BW} = 128KB + (16+2) \cdot n_f$. For the lazy updates BF the actual number of flowIDs sent to the controller is $\hat{n}_f > n_f$, while the number of counters in the CMS is reduced since the lazy updates avoid the insertion in the CMS of the flows with less than 4 packets. The data reported in Figure 4.14 shows a $\hat{n}_f \approx 1.5 \cdot n_f$. Furthermore, we can estimate a 50% reduction in the CMS size and thus the BW for the lazy updates can be computed as $C_{BW} = 128KB + (24+1) \cdot n_f$. For the case of the differential flow detection, we can reduce the amount of flows sent to the controller by around 25%, corresponding to a BW of $C_{BW} = 128KB + (12+2) \cdot n_f$.

The above-presented evaluation shows that the FlowLiDARs system re-

quires a bandwidth of 6.4MB/sec (3.6 MB/sec) in the worst case of 1 ms resolution with lazy updates (differential BFs) and 1.5MB/sec (0.94 MB/sec) in the case of 1-second resolution. Supposing that the connection between the control plane and the data plane is provided by a PCIe interface, also the worst case of 6.4MB/sec is fully sustainable using only a fraction of the available PCIe bandwidth. Even if we consider a 20X speedup to mimic the behavior of a 100 Gbps link, as done as an example in Sonata [77], the worst case requires on the order of 1 Gbps of PCIe bandwidth. Applying the same speedup, in Figure 4.13 and Figure 4.14 the x-axis should be scaled by 20x to estimate the FPR and bandwidth for a fully used 100 Gbps link.

4.5.3 Comparison with Other Solutions

In this section, I compare FlowLiDAR with FlowRadar [133], the NZE sketch [91], the PR sketch [183] and the ElasticSketch [222]. If FlowRadar has sufficient memory, it can provide an exact result for almost all the monitored flows. Instead, with insufficient memory, the IBLT decoding process fails, no FlowIDs can be recovered, and no flow estimation can be done. Therefore, for the comparison between FlowRadar and FlowLiDARs, I estimated the minimum amount of memory needed to achieve 99% of exact results. Instead, for NZE, PR-sketch, and ElasticSketch, the amount of memory is fixed. Several metrics are evaluated, namely the required bandwidth, AAE, ARE, and the percentage of flows with no error. For FlowLiDARs, a lazy BF of (4x128 Kbits) and a CMS of (32x1Kx16bits) were used, with an overall memory of 128 KB. For NZE the code from the NZE repository is used, allocating for the BF in the NZE sketch the same size as the lazy BF in FlowLiDARs, and the amount of memory used by the FlowLiDARs CMS corresponds to the sum of the CMS and hash tables used in the NZE sketch. In detail, 32KB was allocated to the NZE CMS and 32KB to the hash table. For the PR-sketch the configuration is similar to FlowLiDARs: 64KB for the BF and 32Kx16bits for the CMS.

Note that the size of the lazy BF of FlowLidar and of the BF of NZE and PR-sketch is related to the number of undetected flows, *i.e.* flows that are not

monitored by the system. To monitor more than 99% of flows, for a trace with around 60K flows, at least 64KB of memory is needed for the standard BF used in PR-sketch (see equation (4.1)). This corresponds to a ratio between BF and CMS different from the one used in the original PR-sketch paper. Using the default ratio of 12.5% leads to a ratio of undetected flows of around 20%.

For ElasticSketch, 25% of memory is allocated to the heavy part (32KB) and 75% of memory to the light part (96KB), following the configuration proposed in their paper [222]. Note that since the original paper of ElasticSketch does not mention the case in which all the monitored FlowIDs are sent to the controller at the end of the measurement epoch, the BW usage is not included in this dissertation either. Following the configuration of [133], FlowRadar requires around 1.4 MB to correctly decode traces C1 and C2, and around 800KB for trace C3. Thus FlowLiDARs provides a memory saving between 6x and 10x compared to FlowRadar.

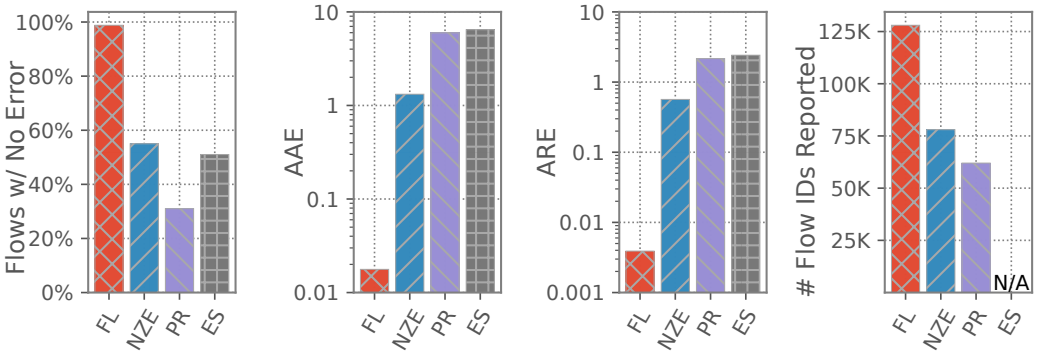


Figure 4.15: Comparison between FlowLiDAR (FL), NZE, PR-Sketch (PR), and ElasticSketch (ES)

In Figure 4.15, I show the comparison with NZE, PR-sketch, and ElasticSketch for one trace with an epoch time of 1 second. However, the results are similar for other traces and configurations and consistently show that FlowLiDARs provides better results at the expense of a slight increase in the number of flowIDs sent to the control plane. In particular, FlowLiDARs has much better results both in terms of ARE and AAE, as well as in terms of flows with no error[8]. This is mainly due to two reasons: first of all, in FlowLiDARs, small flows are directly counted by the control plane and do not use the CMS counters; second, FlowLiDARs avoids a hash table in the data plane, which permits doubling the size of the CMS. Both of these improve the possibility of exact results using the resolution method described in section 4.3.4.

[8]Note that the performance characteristics of PR-sketch, as well as FlowLiDARs, depends on the overall configuration of the data structure (e.g., the ratio of memory used for filtering vs sketching). Here, the ratio is chosen to be the same in both systems to provide a fair comparison.

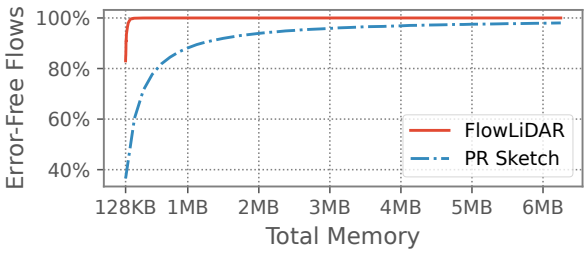


Figure 4.16: Comparing FlowLiDAR and PR-sketch in terms of error-free flows at different allocated memory sizes

Finally, since the PR-sketch is conceptually close to FlowLiDAR, an additional comparison was conducted on the memory efficiency, which I present in Figure 4.16. The fraction of flows tracked without any estimation errors was recorded (data refers to the C1 trace) when a varying amount of memory was allocated for filtering and sketching. The PR-sketch achieves higher error-free rates as more memory is allocated but at a much lower rate than FlowLiDARs. For example, to achieve a target error-free rate of 90%, PR-sketch would require *8x as much memory as FlowLiDARs*. This clearly shows the benefits of the innovations introduced in this chapter. The better performance of FlowLiDARs is mainly due to the lazy BF, which reduces the number of monitored flows, thus increasing the memory size range in which the exact resolution can be used. However, even in the case of an underdetermined system, the use of an ad-hoc approximate resolution provides better results than the least square method used in PR-sketch, as discussed below.

4.5.4 FlowLiDAR Approximate Resolution

As mentioned in section 4.3.4, when the number of flows is higher than the number of CMS rows, the system is underdetermined, and there are multiple possible solutions. A set of experiments was performed that reduced the size of the CMS to understand the quality of the approximate solution provided by Algorithm 7. In particular, the lazy update BF is used with $k = 4$ and the overall memory of the CMS was reduced from 512 Kbits (the 32x1K configuration discussed in section 4.5.1), which can provide the exact resolution, down to 32Kbits of a 32x64 CMS configuration. The collected data proves that even with a small amount of memory, it is possible to have

a good estimation of the flow size. The results are compared with the ones achieved using the least square method proposed in PR-sketch[9] and with the estimated values obtained using the standard CMS estimation based on the minimum value. Note that the least square method simply picks one of the possible solutions (all of them have the L^2-norm equal to zero), and therefore does not give a guarantee on the actual error between the proposed solution and the actual values. Instead, FlowLiDARs selects the free variables of the system to fix among the smallest ones, thus minimizing the error between the free variables and the actual values. Table 4.6 presents the percentage of exact estimations for different configurations, along with the AAE and ARE of the FlowLiDARs approximate resolution. These results are compared with those obtained using the least squares method and the standard CMS estimation method.

	size (bits)	#exact	#exact lstsq	AAE	AAE lstsq	AAE std	ARE	ARE lstsq	ARE std
32x1K (exact)	512K	99.0%	99.0%	0.019	0.019	3.468	0.0052	0.0052	1.97
32x512	256K	65.0%	60.8%	5.11	5.75	12.9	0.58	1.17	7.57
32x256	128K	63.3%	58.8%	9.11	18.0	42.6	1.32	3.68	24.31
32x128	64K	63.2%	58.8%	21.93	50.4	121	3.94	10.3	68.54
32x64	32K	63.1%	58.8%	45.41	129.9	315	8.57	26.5	177

Table 4.6: Performance of FlowLiDAR approximate CMS resolution

The table shows that the approximate resolution is still able to provide exact estimations in 60% of cases. These values are mostly due to the use of the lazy update BF that counts the number of packets in the flow based on the number of occurrences of the FlowID sent to the control plane. The main benefit of the approximate resolution appears on the obtained AAE and ARE values, which are significantly smaller than the ones achievable using the standard evaluation of the CMS. In particular, when the memory available for the CMS is small, the approximate resolution provides a much better estimation than the standard one. For example, comparing the 32x64 configuration that requires 32Kbits, it has around a 3x better AAE and ARE

[9]This can be seen as an improved PR-sketch since it first exploits the benefit of the lazy update mechanism and after uses the least square method.

compared to the least square method used in PR-sketch, a 7x better AAE and a 20x better ARE compared to the one achievable using the traditional CMS.

4.5.5 Equation Solving Time

Another aspect of the FlowLiDAR implementation to take into account is the processing time needed for CMS equation solving. It is known that the processing time grows more than quadratically [49] and thus it is important to reduce the size of the CMSs used by FlowLiDAR. On the other hand, the use of smaller CMSs has a slightly negative impact on the probability of exactly solving the CMS equation. In more detail, since a single hash function is used to select the CMS for a given flow, the load factor of each CMS will vary. Therefore some CMSs will be overloaded, and if they go over the resolution threshold the exact CMS resolution will fail. Note that this is not a dramatic event, since in case of failure, FlowLiDAR uses as a fallback the estimation of size based on the minimum. To better investigate this aspect, a set of experiments was performed on an 8 core, 16 threads Intel i7-10700K CPU clocked at 3.80 GHz to evaluate the processing time of a CMS exact resolution using a single core, varying the number of rows of the system equation. The results are presented in Table 4.7.

CMS size	256	512	1K	2K	4K
Processing time (ms)	0.6	2.8	15	100	680

Table 4.7: Processing Time for Exact CMS resolution

From Table 4.7 we can identify a CMS size that allows a line rate decoding using the parameters selected in the previous section. In particular, for the shortest epoch period of 1 ms, 2 CMS of 256 elements are sufficient to store the active flows in one epoch (that are less than 300 in the three CAIDA traces used in the simulations), and can also be decoded in a time interval less than the epoch period using 2 CPU threads. For longer periods the scenario is less challenging since we can exploit multiple cores to perform the exact resolution of different CMS equations in parallel. Furthermore, a greater epoch period permits to increase in the size of the single CMS. For instance, it is feasible to deploy 64 1K CMSs with a 1-second resolution, which a single thread can process in approximately 960 milliseconds.

4.6 Chapter Summary

In this chapter, I addressed the second research objective of my dissertation: "Improve the Cost vs Accuracy Tradeoff in Sketches". To achieve this, I developed and presented FlowLiDAR, a novel solution that optimizes per-sketch memory utilization to provide highly accurate flow size estimates even under severe memory constraints.

FlowLiDAR introduces several key innovations. Firstly, it eliminates the need to store FlowIDs in the data plane by sending them to the control plane, significantly reducing memory requirements. Secondly, it enhances accuracy by solving the sketch as a linear programming problem, made efficient through a multi-sketch per-switch deployment. Thirdly, the lazy update mechanism in the Bloom filter reduces false positives as well as reduces the number of flows counted by the sketch itself.

The FlowLiDAR approach efficiently combines these techniques to drastically reduce data plane memory usage while achieving excellent accuracy. By decoupling FlowIDs from their associated counters and implementing lazy updates, FlowLiDAR addresses the inherent challenges posed by the high-speed, memory-constrained environments typical of network switches.

My evaluation of FlowLiDAR demonstrated substantial improvements in estimation accuracy compared to state-of-the-art alternatives. For instance, FlowLiDAR requires at least 6x less memory than FlowRadar. When compared with NZE, PR-sketch, and ElasticSketch configured with the same amount of memory, FlowLiDAR reduced the errors by orders of magnitude. Moreover, FlowLiDAR successfully tracks 98.7% of existing flows, whereas other techniques only reconstruct at most 60% of flows with similar resources, greatly increasing the flow coverage.

Chapter 5

High-Speed Collection

The two previous research chapters focused on collection-light sketch-based monitoring. However, sketches have limitations in versatility, primarily supporting simple key-based estimations. As discussed in the background chapter, comprehensive network insights necessitate multiple in-network monitoring solutions, many of which generate vast amounts of data, necessitating sampling to manage the collection bottleneck.

To address the third research objective, "Alleviate the Telemetry Collection Bottleneck", this chapter develops advanced techniques and algorithms for efficient and scalable data aggregation. Here, I introduce Direct Telemetry Access (DTA), a high-performance telemetry collection system designed to improve the cost vs. insight tradeoff in fine-grained network telemetry.

DTA can aggregate and transfer hundreds of millions of telemetry reports per second from switches to a centralized collector, improving collection rates by an order of magnitude compared to current solutions. Additionally, to further enhance scalability, DTA is horizontally scalable, supporting multiple parallel collectors. This system is built on Remote Direct Memory Access (RDMA) and introduces novel reporting primitives to ensure seamless integration with existing telemetry mechanisms such as In-band Network Telemetry (INT) and Marple.

Although DTA primarily focuses on large-scale data collection in modern data centers, it also applies to other large-scale environments, such as

Autonomous Systems (ASs).

Attributions The contributions in this chapter were led and delivered by me, except for the stochastic analysis, which was performed by Prof. Michael Mitzenmacher and Dr. Ran Ben Basat. All co-authors in the already published papers participated in general discussions during weekly meetings.

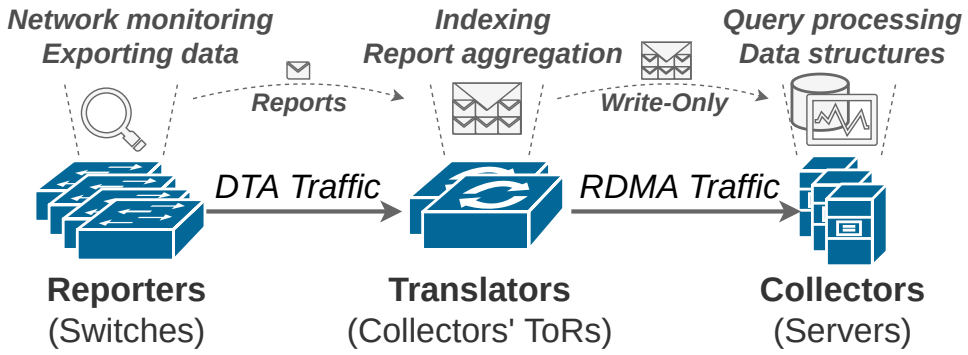


Figure 5.1: An overview of the telemetry data flow in DTA.

5.1 Introduction

Although sketches fill an important niche in network telemetry, they are limited to estimations of specific metrics and need to be supplemented with other monitoring systems for a holistic insight [78, 194]. Similarly to sketches, most monitoring solutions aggregate per-switch data into centralized collectors [12, 34, 77, 93, 105, 115, 163], commonly located in an ordinary rack within the datacenter fabric [94, 151], to grant a network-wide view.

Unfortunately, as telemetry gets more fine-grained, the amount of data to send to a collector increases, and it is progressively harder to scale data collection systems [115, 207, 230]. Indeed, a switch can generate up to millions of telemetry reports per second [159, 230] and a data center network can comprise thousands of them [76]. Also, the amount of data keeps growing with larger networks and higher line rates [185].

Existing research boosts scalability in data collection by improving the collector's network stacks [115, 207], by aggregating and filtering data at switches [103, 120, 159, 209, 230], or by reducing the exported information through switch cooperation [131]. However, as I show, a collector can easily become either Central Processing Unit (CPU)- or memory-bounded (§5.2). This is due to the amount of data processing (i.e., Input/Output (I/O), parsing, and data insertion) required for every incoming report.

I propose Direct Telemetry Access (DTA) — a telemetry collection system (Figure 5.1) optimized for aggregating and moving hundreds of millions of reports per second from switches into queryable data structures in collectors' memory. In designing DTA, I considered four key goals: (1) relieving a collector's CPU from processing incoming reports while also (2) greatly lowering the number of memory access into it. Those aspects dramatically reduce overheads at the collectors. Furthermore, I wanted (3) to be compatible with state-of-the-art telemetry reporting solutions (e.g., In-band Network Telemetry (INT) [103], Marple [159]) while (4) imposing minimal hardware resource overheads at switches.

To meet the first goal, we could simply have switches generate Remote Direct Memory Access (RDMA) (Remote Direct Memory Access) [101] calls to a collector's memory. RDMA is available on many commodity network cards [104, 199, 215] and can perform hundreds of millions of memory writes per second [199], significantly faster than the most performant CPU-based telemetry collector [115]. Previous work [117] has shown that one can generate RDMA instructions between a switch and a server for network functions. However, it is challenging to adopt RDMA between multiple switches and a collector for telemetry systems as RDMA performance degrades substantially when multiple clients write to the same server [111]. Furthermore, managing RDMA connections at switches is costly in terms of hardware resources and this would conflict with my fourth goal.

Instead, I developed a solution where the telemetry data exported by switches is encapsulated into a custom and lightweight protocol. This encapsulated data is intercepted by the last hop switch in front of the collector, commonly the Top of Rack (ToR) switch, which I refer to as a DTA *translator*. The translator converts the encapsulated data into standard RDMA calls for the corresponding memory (§5.3).

To achieve the first goal, the CPU avoids processing reports by design, as data is inserted directly into a collector's memory via RDMA.

For the second goal, the translator aggregates and batches reports before invoking RDMA calls, inserting the data into a collector's memory using RDMA-compatible write-only data structures that enable indexing of aggre-

gates without reading from memory, thus reducing memory pressure on the collector's memory based on the needs of the telemetry data.

For the third goal, I designed several switch-level RDMA-extension primitives (*Key-Write*, *Postcarding*, *Append*, and *Key-Increment*), available to reporting switches. The translator converts these into standard RDMA calls, ensuring compatibility with many telemetry systems (§5.4).

Finally, for the fourth goal, telemetry-reporting switches use my User Datagram Protocol (UDP)-based protocol to send reports, freeing them from managing RDMA, a responsibility solely handled by the translator.

I implemented DTA using commodity RDMA Network Interface Cards (NICs) and programmable switches (§5.5) and my evaluation (§5.6) shows that it can process and aggregate over 400M INT reports per second, without any CPU involvement, which is 16x faster than the state-of-the-art CPU-based collector for high-speed networks [115]. Further, when the received data can be recorded sequentially, as in the case of temporally ordered event reports, it can ingest up to a billion reports per second, 41x more than state-of-the-art.

The main contributions in this chapter are:

- I show that collectors can easily become either CPU- or memory-bounded, greatly limiting their ability to process reports and store them in queryable data structures.

- I propose *Direct Telemetry Access*, a novel telemetry collection system generic enough to support major telemetry reporting solutions proposed by the research community (e.g., Marple) or industry (e.g., INT).

- I validate the hardware feasibility of DTA by prototyping it using commodity RDMA NICs and programmable switches, all of which are released as open source.

- I provide an in-depth evaluation demonstrating DTA's significant collection capacities compared to current solutions, while simultaneously reducing memory and CPU overheads.

System	Per-switch Report Rate
INT Postcards (Per-hop latency, 0.5% sampling)	19 Mpps
Marple [159] (Flowlet sizes)	7.2 Mpps
Marple [159] (TCP out-of-sequence)	6.7 Mpps
NetSeer [230] (Loss events)	950 Kpps

Table 5.1: Per-reporter data generation rates by various monitoring systems, as presented in their papers and verified through my experiments. Numbers are based on 6.4Tbps switches.

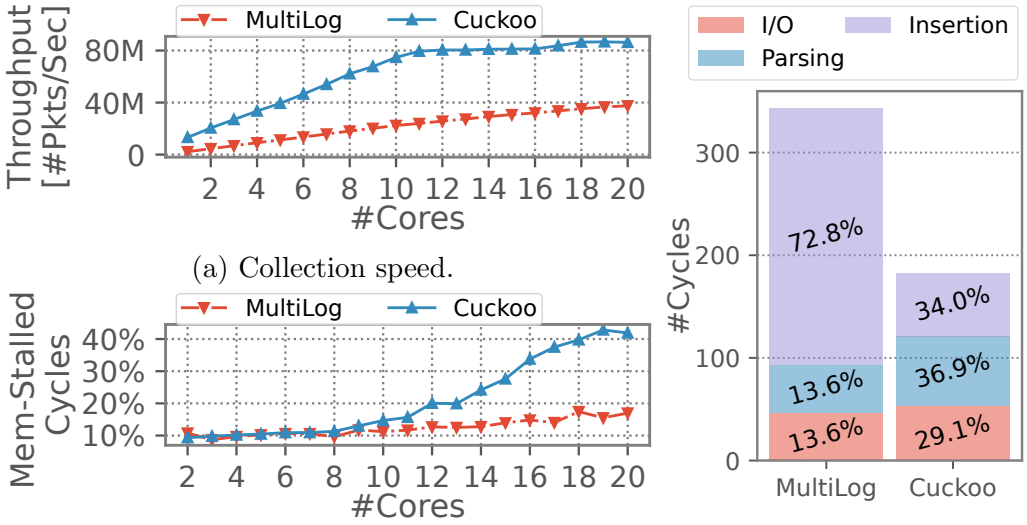
5.2 Motivation

Telemetry systems are commonly composed of two main components: (1) switches reporting data and (2) collectors, specialized software installed in dedicated servers located in ordinary racks within the data center fabric, that store the reported data [94, 151]. As telemetry systems move to fine-grained real-time analysis with support for network-wide queries, report collection becomes the new key bottleneck [115].

I investigated several state-of-the-art telemetry systems and summarize the reporting rate generated by a single switch in Table 5.1, based on the numbers available in the corresponding papers.[1] For example, postcarded INT [103] could generate up to $19M$ reports per second when enabled on a commodity 6.4Tbps switch and in the presence of a standard load of ≈40% [225]. Other solutions export less data, either because they pre-process and filter data at switches [77, 230], or because they focus on more specific tasks, thus limiting the data to report [159].

The main takeaway is that state-of-the-art solutions can easily generate *millions of reports per second per switch*. However, to be able to gather a network-wide view at datacenter scale, we may need to collect data from thousands of switches [76] and this requires high-performance collection stacks [115, 207]. For each report from a switch, collectors spend CPU cycles

[1]INT does not advertise a telemetry reporting rate. Thus, as an example, I chose an arbitrary sampling rate of 0.5% to keep overheads reasonably low.



(a) Collection speed.

(b) Memory stalling.

(c) Breakdown.

Figure 5.2: The performance of CPU-based collectors. MultiLog is CPU-bound, while Cuckoo is memory-bound as with 20 cores, 42% of the cycles are spent waiting for a memory operation to finish.

to receive the data (i.e., I/O), parse it (extract content from the report), and insert it in a queryable data structure for later use (i.e., indexing) [36, 115, 149, 207].

Here, an important trade-off must be considered: while more complex indexing mechanisms can efficiently answer various types of queries, they typically require more CPU cycles for data insertion. Consider, for example, a simple collector that uses only a hash table to record incoming reports. This solution works well for storing and retrieving counters (e.g., Netflow flow records [38]). However, such a solution might be impractical for certain types of queries, such as temporal queries that examine a time interval (such as analyzing losses [230], congestion [74], suspicious flows [120] or latency spikes [225] that happens at a certain period in time).

To better understand this trade-off, I have deployed a state-of-the-art Data Plane Development Kit (DPDK)-based telemetry collector allowing storage

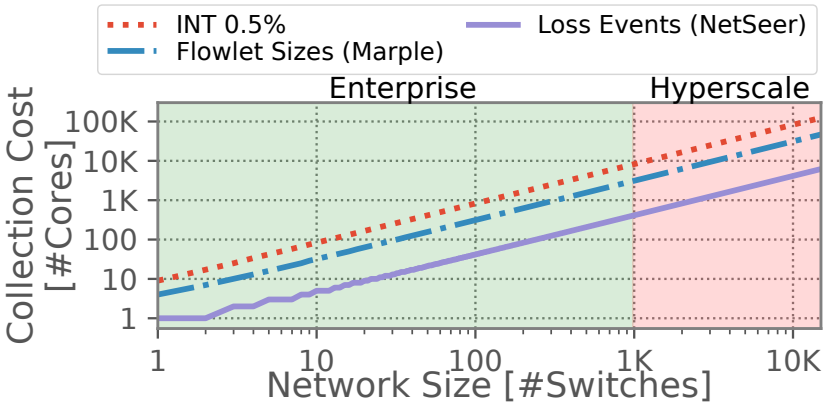


Figure 5.3: Number of cores needed for single-metric collection with MultiLog at various network sizes.

and diverse queries through an Atomic MultiLog [115] (from now on I simply refer to it as "MultiLog").[2] I used a high-speed server equipped with 2x Intel Xeon Silver 4114 CPUs with 10 cores each clocked to 2.20GHz, and 2x32GB Dynamic RAM (DRAM) clocked to 2.67GHz. I compared the performance of this system to a DPDK-based lightweight solution which employs only a simple cuckoo hash table to store the received information (I refer to it as "Cuckoo"). I analyzed their behavior when receiving and storing INT reports and found that the MultiLog collector is *CPU bounded*: indeed, its ability to ingest reports grows linearly with its number of cores (Figure 5.2a). Moreover, the majority of its CPU cycles, around 72.8%, are spent in inserting the data into its internal database (Figure 5.2c). The main takeaway is that a complex indexing scheme can significantly impact the collector's performance.

To put this in perspective, in Figure 5.3, I show the number of cores that would be needed for a growing size of a data center network when employing the MultiLog collector in the presence of switches reporting different information. Here, we see that for networks comprising around a thousand switches [76], we would need to dedicate nearly $10K$ cores just for collection.

[2]Atomic MultiLog is the basic storage abstraction in Confluo [115] and is similar in interface to database tables.

For example, in a $K = 28$ fat tree, this would correspond to over 11% of servers (assuming 16 cores each), and the problem worsens for smaller networks.

In contrast, the lightweight Cuckoo scheme can ingest more reports per second (Figure 5.2a) using the same number of cores. However, a new bottleneck arises: in my tests, we see that with more than 11 cores it becomes *memory bounded*. For example, with 20 cores, 42% of cycles are spent waiting for a memory operation to finish (Figure 5.2b). This is because the high number of reports received puts tremendous stress on the memory subsystem, which must be read and written to parse the reports, calculate the hashes, and resolve collisions.

5.2.1 Design Goals

Based on these observations, I argue that an effective telemetry collection method should:

1. Minimize the number of cores required for data collection.
2. Lower the number of memory accesses per report.
3. Be compatible with state-of-the-art telemetry reporting systems.
4. Use minimal hardware resources to get reports to a collector.

5.3 Direct Telemetry Access Overview

DTA leverages *translators*, which are the last-hop switches adjacent to the collectors. Translators receive telemetry data from *reporters* (i.e., switches exporting telemetry data), encapsulated in my lightweight custom protocol. They then aggregate and batch the reports and use standard RDMA calls to write them directly into queryable data structures in the collectors' memory (Figure 5.1). In the following, I discuss how, with this architecture, DTA meets the goals set above.

Meeting goal #1. A strawman solution to meet the first goal could have switches write their reports directly in collectors' memory with RDMA calls [124]. This would zero any CPU requirements at collectors by design. Although this idea appears attractive, and generating RDMA instructions directly from switches is possible [117], it becomes problematic when applied to telemetry collection. Namely, It is inefficient to support multiple RDMA senders writing in the same servers [111]. This is paramount for network telemetry, where numerous switches report their data to a collector. Additionally, RDMA NICs can only handle a limited number of active connections (also known as *queue pairs*) at high speed. Increasing the number of queue pairs degrades RDMA performance by up to 5x [54]. This limits the total number of switches that can generate telemetry RDMA packets to a collector before performance starts degrading. Alternatively, several switches can share the same queue pair, but RDMA assumes that every packet received at the collector has a strictly sequential ID, which is impractical for a distributed network of switches. DTA overcomes these challenges by having the translator, which is the last-hop switch before the collector, act as the RDMA writer. Further, by aggregating the reports we can optimize the number of CPU cycles needed for querying as related information is stored contiguously.

Meeting goal #2. I propose two techniques to lower the number of accesses into collectors' memory. First, I aggregate reports at the translator, thereby writing each aggregate using a single write rather than one per report. Second,

while telemetry data has to be stored in the collectors' memory in such a way that it is easy to query [115], even simple data structures like hash tables often require an excessive number of memory accesses, e.g., for conflict resolution. Instead, I design RDMA-compatible write-only data structures that enable the indexing of aggregates without reading from memory. Additionally, by circumventing the CPU for data ingestion, this solution also removes I/O and parsing overheads of report collection.

Meeting goal #3. I propose several powerful primitives available at the translator that can be used by state-of-the-art telemetry reporting systems (Table 5.2). The primitives abstract away many common challenges (e.g., deciding where to write data to or how to leverage the small switches' memory) and allow telemetry system designers to seamlessly benefit from my optimizations (e.g., CPU and memory accesses minimization).

Meeting goal #4. In DTA, to minimize in-network hardware resources utilization, reporting switches simply use a UDP-based lightweight protocol to send reports to the translator. That way, we alleviate the burden of RDMA generation and aggregation in all switches but the translators. Indeed, the standard RDMA communication protocol, RDMA over Converged Ethernet (RoCEv2), requires maintaining expensive per-connection metadata and generating appropriate headers and associated checksums.

5.4 DTA Primitives

DTA allows easy integration with state-of-the-art telemetry monitoring systems [18, 74, 77, 159] through my four collection primitives that together support a wide range of telemetry solutions: *Key-Write*, *Postcarding*, *Append*, and *Key-Increment*. These primitives provide for placing data in the right place at the collector's memory during reporting time, to alleviate as much as possible the cost of query execution.

I show in Table 5.2 that the primitives are sufficiently generic to support many state-of-the-art telemetry systems. Additionally, in Section 5.4.5, I

Primitive	Example monitoring	Description
Key-Write *(key,data)*	INT-MD [74, 116] (Path Tracing)	INT sinks reporting *5x4B* switch IDs using *flow 5-tuple* keys
	Marple [159] (Host counters)	Reporting *4B* counters using *source IP* keys, through non-merging aggregation
	PacketScope [201] (Flow troubleshooting)	Report *fixed-size* per-flow per-switch traversal information using *(switchID,5-tuple)* as key
	PINT [18] (Per-flow queries)	*1B* reports with *5-tuple* keys, using redundancies for data compression through $n = f(pktID)$
	Sonata [77] (Per-query results)	Reporting *fixed-size* network query results using *queryID* keys
Postcarding *(key,hop,data)*	INT-XD/MX [74, 116] (Path Measurements)	Switches report *4B* INT postcards using *(flow 5-tuple, hop)* keys
	Trajectory Sampling [56] (Path Frequencies)	Collection of unique packet labels from all hops for sampled packets
Append *(listID,data)*	dShark [65] (Parser-Grouper transfer)	Parsers append packet summaries to lists hosted by Grouper-servers
	INT [74, 116] (Congestion events)	INT sinks append *4B* reports to a list of network congestion events
	Marple [159] (Lossy connections)	Report *13B* flows to a list with packet loss rate greater than threshold
	NetSeer [230] (Loss events)	Appending *18B* loss event reports into network-wide list of packet losses
	PacketScope [201] (Pipeline-loss insight)	On packet drop: send *14B* pipeline-traversal information to central list of pipeline-loss events
	Sonata [77] (Raw data transfer)	Appending *query-specific* packet tuples from switches to lists at streaming processors
Key-Increment *(key,counter)*	Marple [159] (Host counters)	Reporting *4B* counters using *source IP* keys, through addition-based aggregation
	TurboFlow [190] (Per-flow counters)	Sending *4B* counters from evicted microflow-records for aggregation using *flow key* as keys

Table 5.2: Existing telemetry monitoring systems, mapped into the primitives proposed by the current iteration of DTA.

discuss DTA extensions with new primitives, for scenarios where my proposed algorithms are insufficient.

I show the structure of a DTA report in Figure 5.4. The telemetry payload exported by a switch, which depends on the specific monitoring system being used, is encapsulated into a UDP packet that carries custom headers.

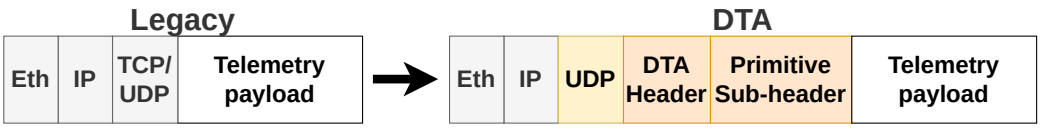


Figure 5.4: DTA supports legacy telemetry systems through encapsulation with new headers.

The *DTA header* (specifying the DTA primitive) and *primitive sub-header* (containing the primitive parameters) are used by the translator to decide what and where to write in the collectors' data structures. This flexibility is essential as the various monitoring systems require writing telemetry in different ways for efficient analysis at the collector. In the following sections, I discuss my proposed primitives. This description assumes that no DTA messages are lost, which could be either through Priority Flow Control (PFC) or a custom flow control solution as discussed later in §5.7. The primitives themselves would still work even in case of severe in-transit loss of reports, although with degraded probabilistic guarantees which is not accounted for in the following theoretical analysis. In most cases, report loss is not detrimental to the functioning of DTA, except for the obvious loss of telemetry insight. However, Postcarding might experience collection latencies following a report-loss, which is discussed in that section.

5.4.1 Key-Write

Key-Write (KW) (visualized in Figure 5.5) is designed for key-value pair collection. Storing per-flow data is one scenario where this primitive is useful (additional examples are in Table 5.2). For an overview, I present the Key-Write translation and querying in Algorithm 8 and Algorithm 9.

Key-value indexing is challenging when the keys come from arbitrary domains (e.g., flow 5-tuples) and we want to map them to a small address space using simple write operations. Due to the lack of read-before-write operations at high performances, I was unable to integrate collision-free techniques such as Cuckoo hashing at line rates [166]. Both to my, as well as

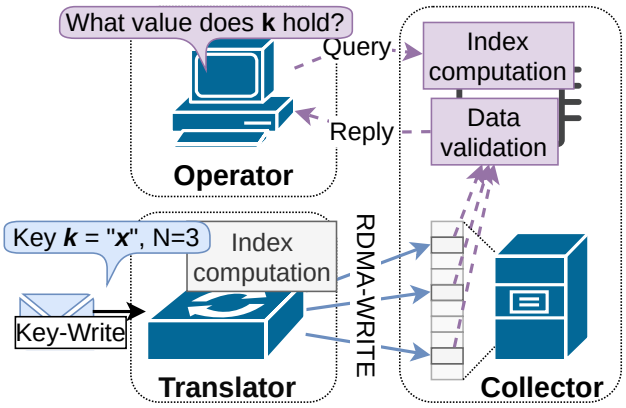


Figure 5.5: Key-Write Overview.

Algorithm 8 DTA-to-RDMA translation in Key-Write

Input: Redundancy N, Key K, Telemetry data D
$Bufstart \leftarrow$ Address to start of RDMA memory buffer
$Buflen \leftarrow$ Number of allocated KeyVal slots
$Slotlen \leftarrow$ Size of one KeyVal slot
function CraftWrite(n, K, D)
    $Slot \leftarrow h_0(n, K)$ mod $Buflen$
    $Dest \leftarrow Bufstart + Slot \times Slotlen$
    $Csum \leftarrow h_1(K)$
    Write $(Csum, D)$ to address $Dest$ through RDMA
end function
for $n = 0$ **to** N **do**
    CraftWrite(n, K, D)
end for

my collaborators' knowledge, no blind-write and collision-free data structures exist that would solve this problem. Due to this, I chose to keep the proposed solutions probabilistic, and the probabilistic properties are discussed further down, including an in-depth evaluation.

Therefore, Key-Write (KW) provides a probabilistic key-value storage of telemetry data and is designed for resource-efficient data plane deployments. This is achieved by constructing a central key-value store as a shared hash table for all telemetry-generating network switches. Indexing per-key data in this hash table is performed statelessly without collaboration through global hash functions. However, data written to a single memory location is highly susceptible to overwrites due to hash collisions with another key's write. The algorithm, therefore, inserts telemetry data as N identical entries at N memory locations to achieve partial collision tolerance through built-in data redundancy. In addition, a checksum of the telemetry key is stored alongside each data entry, which allows queries to be verified by validating the checksum. Although checksum computation imposes a computational and storage overhead, this algorithm significantly outperforms current solutions as is shown in the evaluation.

The network and hardware resource overheads of KW are further reduced by moving the indexing and redundancy generation into the DTA translator. This design choice effectively reduces the telemetry traffic by a factor of the level of redundancy and further reduces the telemetry report costs in the individual switches by replacing costly RDMA generation with the much more lightweight DTA protocol (§5.6.3). Isolating KW logic within collector-managing translators eliminates the associated resource cost for all other switches.

The following stochastic analysis contains content written by one of my collaborators for our co-authored paper. I have included the full analysis here to provide a comprehensive understanding of the algorithms. As discussed later in Section 5.4.6, rigorous bounds can be derived on the probability that KW succeeds. Two potential errors can occur: (i) failure to locate a stored value for a given key (i.e., no matching checksum found at the key's indices); (ii) returning an incorrect value for a given key (i.e., false-positive checksum

matches at the key's indices).

Denoting the number of slots by M, the number of pairs written after the queried key by αM, and the checksum length by b bits, the probability of (i) is bounded by:

$$(1 - e^{-\alpha \cdot N})^N \cdot (1 - 2^{-b})^N \tag{5.1}$$
$$+ (1 - e^{-\alpha \cdot N})^N \cdot (1 - (1 - 2^{-b})^N - N \cdot 2^{-b} \cdot (1 - 2^{-b})^{N-1}) \tag{5.2}$$
$$+ \Big(\sum_{j=1}^{N-1} \binom{N}{j} \cdot (1 - e^{-\alpha \cdot N})^j \cdot e^{-\alpha \cdot N(N-j)} \cdot (1 - (1 - 2^{-b})^j) \Big). \tag{5.3}$$

Here, (5.1) bounds the probability that all N locations are overwritten with other checksums; (5.2) bounds the probability that all locations are overwritten and at least two items with our key's checksum write different values; and (5.3) bounds the probability that not all slots are overwritten, but at least one is overwritten with the query key's checksum. The probability of giving the wrong output (ii) is also bounded by

$$(1 - e^{-\alpha \cdot N})^N \cdot N \cdot 2^{-b}. \tag{5.4}$$

For example, if $N = 2, b = 32, \alpha = 0.1$, the chance of not providing the output is less than 3.3%, while the probability of wrong output is bounded by $1.6 \cdot 10^{-11}$. This aligns with the best effort standard of network telemetry (e.g., INT is often collected using UDP, and packet loss results in missing reports) while having a negligible chance of wrong output. Note that this error is significantly lower than with $N = 1$ (which results in not providing output with probability 9.5%) and higher than for $N = 4$ (probability 1.2%). However, increasing N also has implications to throughput (more RDMA writes) and is not always justified; I elaborate on this tradeoff in §5.6.5 and show that $N = 2$ is often a good compromise.

DTA also lets switches specify the importance of per-key telemetry data by including the level of redundancy (N), or the number of copies to store, as a field in the KW header. Higher redundancy means a longer lifetime before overwrites, as I discuss in §5.6.5. As the level of redundancy used at report time may not be known while querying, the collector can assume by

Algorithm 9 Querying the Key-Write storage

1: **Input:** Redundancy N, Key K, Consensus threshold T
2: **Output:** D_{winner}
3: $Buflen \leftarrow$ Number of allocated KeyVal slots
4: $Storage \leftarrow$ Array size $Buflen$ with $\langle Csum, D \rangle$ elements
5: **function** GetSlot(n, K)
6: $\quad Slot \leftarrow h_0(n, K) \bmod Buflen$
7: $\quad$ **return** $Storage[Slot]$
8: **end function**
9: $Csum \leftarrow h_1(K)$
10: **for** $n = 0$ **to** N **do**
11: $\quad (Csum_{slot}, D) \leftarrow$ GetSlot(n, K)
12: $\quad$ **if** $Csum == Csum_{slot}$ **then**
13: $\quad\quad$ Add D to list of candidates
14: $\quad$ **end if**
15: **end for**
16: $D_{winner} \leftarrow$ candidate D if D appears at least T times

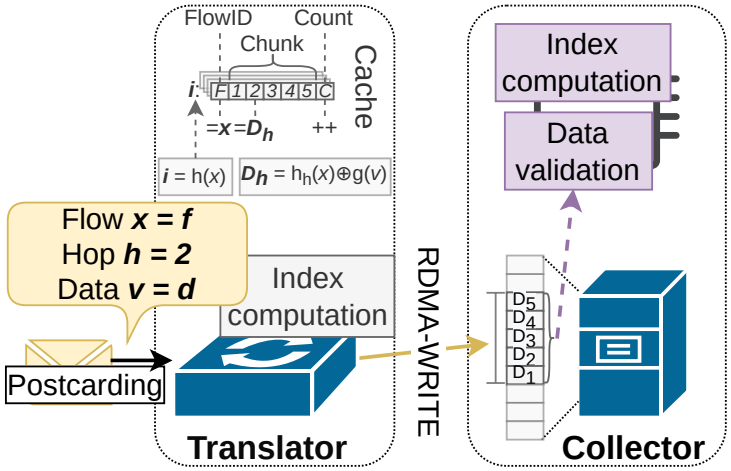


Figure 5.6: Postcarding Overview.

default a maximum (e.g., $N = 4$) redundancy level. If the data was reported using fewer slots, unused slots would appear as empty or overwritten entries (collision).

5.4.2 Postcarding

Here I describe one of the more intricate DTA primitives: Postcarding (visualized in Figure 5.6).

One of the most popular INT working modes is postcarding (INT eXport Data/eMbed instruct(X)ions (INT-XD/MX) [74]), where each switch generates *postcards* when processing selected packets and sends them to the collector (e.g., for tracing a flow's path.) A report is a collection of one postcard from each hop. Intuitively, while the KW primitive could be used to write all postcards for a given packet, this is likely inefficient for several reasons. First, each packet can trigger multiple reports translating into multiple RDMA writes even if $N = 1$ (e.g., one per switch ID for path tracing). In turn, for answering queries with KW (e.g., outputting the switch ID list), the collector needs to make multiple random-access reads, which is unnecessarily slow. Further, adding the KW's checksum to each hop's information is wasteful and degrades the storage size vs. queryability tradeoff.

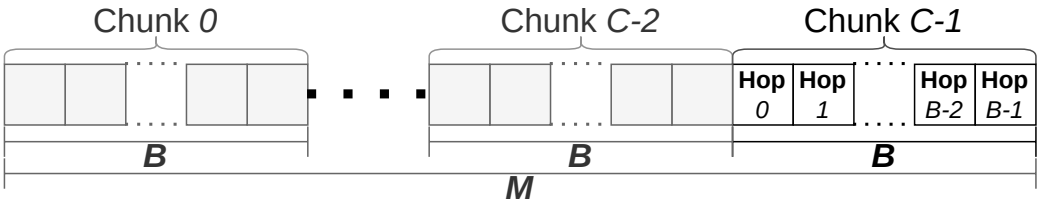


Figure 5.7: The Postcarding memory structure at the collector.

For ease of presentation, I first explain how to reduce the number of writes and later elaborate on how to decrease the width of each slot.

My observation is that if we know a bound B on the number of hops a packet traverses (e.g., five for fat tree topology), then we can improve the above by writing all of a packet's postcards into a consecutive memory block. To that end, DTA breaks the M memory locations into chunks of size B, yielding $C = M/B$ chunks. The i'th postcard for a packet/flow ID x is written into $B{\cdot}h(x) + i$, where h maps identifiers into chunks (i.e., $h(x) \in \{0, \ldots, C-1\}$). This way, the report for all up to B is consecutive in the memory, as shown in Figure 5.7. Note that this primitive assumes a maximum path length that is knowable in a network topology.

DTA Postcarding uses a mapping from IDs to postcards at the translator to reduce the number of RDMA writes. That is, the translator shall buffer postcards $0, 1, \ldots, B-1$ before writing the report to the collector's memory using a single RDMA write, once B flow postcards are counted in the translator. Further, answering queries will thus require a single memory random access. As not all packets follow a B hop path, egress switches can provide a packet's path length inside postcards, and translators can use this value to trigger writes before the postcard counter reaches B. This path length can either be defined in each postcard, or at the final node egressing the network. Additionally, reports may be flushed due to collisions within the switch's buffer.

Finally, we can reduce the number of bits needed for each location compared with writing the value and checksum to each slot. By leveraging the B postcards, I amplified the success probability; the report is output only

if *all* checksums are valid, thereby minimizing the chance of wrong outputs. To achieve this, we use $b > \log_2 |V|$ bits per location to get a collision chance of approximately $(|V| \cdot 2^{-b})$ for each location and $(|V| \cdot 2^{-b})^B$ overall. Here, V represents the set of all possible values (e.g., all switch IDs). As noted by Probabilistic INT (PINT) [18], $|V|$ is often smaller than 2^{32} (although the INT standard requires that each value is reported using exactly four bytes [74]), allowing the use of small b values.

Let g be a hash function that maps values $v \in V$ into b-bit bitstrings, where b is the desired slot width. We use a "blank" value $\sqcup$ to denote that values for a given hop were not collected (potentially because the path length was shorter than B); this way, each flow always writes all hops' values, minimizing the chance of false output due to hash collisions. Then, when receiving a postcard value $v_{x,i} \in V$ from the i'th hop of flow/packet ID x, we write $\text{checksum}(x,i) \oplus g(v_{x,i})$ into location $B \cdot h(x) + i$ (here $\text{checksum}(x,i)$ also returns a b-bit result and $\oplus$ is the bitwise-xor operator). When answering queries about x, we check if there exists ℓ such that for all $i \in \{0, \ldots, \ell - 1\}$ there exists a value $v_{x,i} \in V$ for which $\text{checksum}(x,i) \oplus g(v_{x,i})$ is stored in slot $B \cdot h(x) + i$ and for all $i \in \{\ell, \ldots, B-1\}$ $\text{checksum}(x,i) \oplus g(\sqcup)$ is stored. If so, we output that the postcard reports were $v_{x,0}, v_{x,1}, \ldots, v_{x,\ell-1}$. In this case, we say that the chunk contains valid information. Note that checking the existence of such $v_{x,i}$ can be done in constant time using a pre-populated lookup table that stores all key-value pairs $\{(g(v), v) \mid v \in V \cup \{\sqcup\}\}$.

My approach generalizes with redundancy $N > 1$: we use N hash functions $h_1, \ldots, h_N$ such that $\text{checksum}(x,i) \oplus g(v_{x,i})$ is written into locations $\{B \cdot h_j(x) + i \mid j \in \{1, \ldots, N\}\}$. For answering queries, we output $v_{x,0}, v_{x,1}, \ldots, v_{x,\ell-1}$ if it appears in a valid subset of the N chunks, and all other chunks contain invalid information.

Section 5.4.7 analyzes the primitive and proves that the probability of not providing an output is bounded by:

$$(1 - e^{-\alpha \cdot N})^N \cdot \left(1 - \left((|V| + 1) \cdot 2^{-b}\right)^B\right)^N \tag{5.5}$$
$$+ (1 - e^{-\alpha \cdot N})^N \cdot \left(1 - \left(1 - \left((|V| + 1) \cdot 2^{-b}\right)^B\right)^N\right.$$

$$- N \cdot \left((|V|+1) \cdot 2^{-b}\right)^B \cdot \left(1 - \left((|V|+1) \cdot 2^{-b}\right)^B\right)^{N-1} \right) \tag{5.6}$$

$$+ \sum_{j=1}^{N-1} \binom{N}{j} \cdot (1 - e^{-\alpha \cdot N})^j \cdot e^{-\alpha \cdot N(N-j)} \cdot \left(1 - \left(1 - \left((|V|+1) \cdot 2^{-b}\right)^B\right)^j\right). \tag{5.7}$$

It also shows that the chance of wrong output is bounded by:

$$(1 - e^{-\alpha \cdot N})^N \cdot N \cdot \left((|V|+1) \cdot 2^{-b}\right)^B. \tag{5.8}$$

Consider a numeric example to contrast these results with using KW for each report of a given packet. Specifically, suppose that we are in a large data center ($|V| = 2^{18}$ switches) and want to run path tracing by collecting all (up to $B = 5$) switch IDs using $N = 2$ redundancy. Further, let us set $b =$ 32-bit per report and compare it with 64 bits (32 for the key's checksum and 32 bits for the switch ID) used in KW, and that $C \cdot \alpha$ packets' reports were collected after the queried one, for $\alpha = 0.1$. The probability of not outputting a collected report (5.5-5.7) is then at most 3.3% and the chance of providing the wrong output (5.8) is lower than 10^{-22}. In contrast, using KW for postcarding gives a false output probability of $\approx 8 \cdot 10^{-11}$ (in at least one hop) using twice the bit-width per entry!

5.4.3 Append

Some telemetry scenarios are not easily managed with key-value stores. A classic example is when a switch exports a stream of events, where a report would include an event identifier and an associated timestamp (e.g., packet losses [230], congestion events [74], suspicious flows [120], latency spikes [225]). A key-value store is inappropriate for this scenario; rather, a list or queue is a better abstraction.

I therefore provide a primitive that allows reporters to append information into global lists, with a pre-defined telemetry category in each list. I call this primitive *Append* (visualized in Figure 5.8). For an overview, see Algorithm 10 and Algorithm 11.

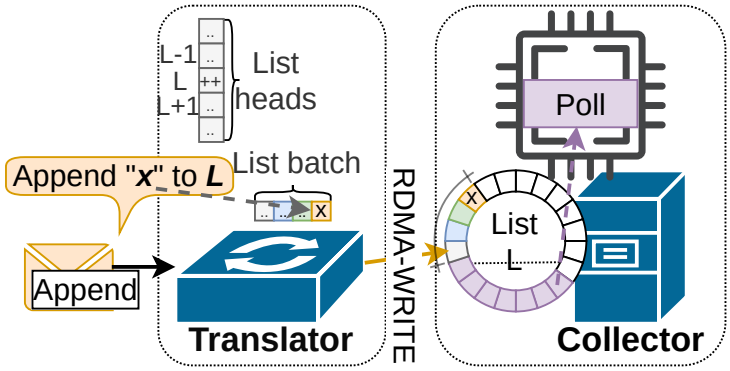


Figure 5.8: Append Overview.

Telemetry reporters simply have to craft a single DTA packet declaring what data they want to append into which list, and forward it to the appropriate collector. The translator then intercepts the packet and generates an RDMA call to insert the data in the correct slot in the pre-allocated list. The translator utilizes a pointer to keep track of the current write location for each list, allowing it to insert incoming data per list. Append inserts reports sequentially and contiguously into memory. This leads to an efficient use of memory and strong query performance. Translation also allows us to *significantly* improve on the collection speeds by batching multiple reports together in a single RDMA operation.

5.4.4 Key-Increment

Key-Increment (KI) (Figure 5.9) is similar to the KW primitive, but allows for addition-based data aggregation. That is, the Key-Increment (KI) primitive does not instruct the collector to set a key to a specific value, but it instead *increments* the value held by the key. For example, switches might only store a few counters in a local cache, and evict old counters from the cache periodically when new counters take their place [159, 190]. Sketches, as explained in previous chapters, are also great candidates for these types of collections by delivering purely counter-based measurements. The KI primitive can then achieve collection of these counters at RDMA rates. As

Algorithm 10 DTA-to-RDMA translation in Append

```
1: Input: List ID L, Data D
2: ListBuffers ← Vector with |Lists| buffer pointers
3: BufferLengths ← Vector with |Lists| buffer lengths
4: Heads ← Vector with |Lists| head-offsets
5: BatchSize ← The global batch size
6: BatchPointer ← Vector with |Lists| integers
7: Batches ← 2D-vector sized [|Lists|][BatchSize − 1]
8: function WriteBatch(L,D)
9:     Batch ← (Batches[L], D)
10:    Address ← ListBuffers[L] + Heads[L]
11:    Write Batch to address Address through RDMA
12:    Heads[L] += BatchSize
13:    if Heads[L] == BatchSize then
14:        Heads[L] ← 0
15:    end if
16: end function
17: if BatchPointer[L] == BatchSize then
18:    WriteBatch(L,D)
19:    BatchPointer[L] ← 0
20: else
21:    Batches[L][BatchPointer[L]] ← D
22:    BatchPointer[L] += 1
23: end if
```

Algorithm 11 Querying the Append storage

```
1: Input: List ID L
2: Output: data
3: ListBuffers ← Vector with |Lists| buffer pointers
4: BufferLengths ← Vector with |Lists| buffer lengths
5: Heads ← Vector with |Lists| head-offsets
6: data ← ListBuffers[L] + Heads[L]
7: Heads[L] ← (Heads[L] + 1) mod BufferLengths[L]
```

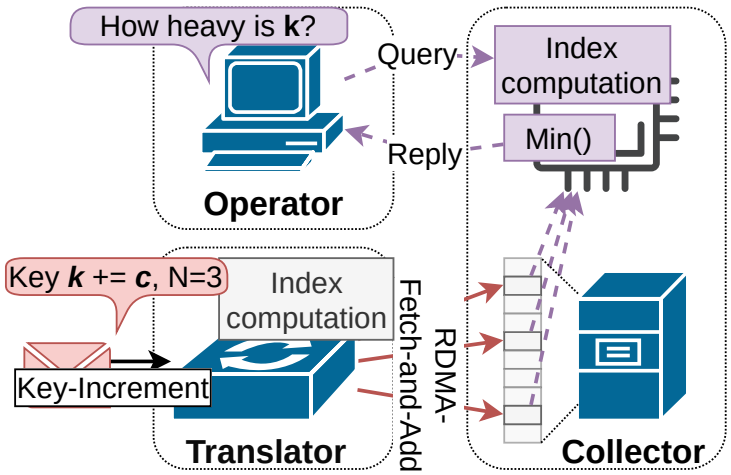


Figure 5.9: Key-Increment Overview.

with KW, the translator reduces network overheads compared with a more naive design. For an overview, see Algorithm 12 and Algorithm 13.

The KI memory acts as a Count-Min Sketch [46] and the translator increments N values using the RDMA Fetch-and-Add primitive. On a query, KI returns the minimum value from these N locations. Hash collisions may lead to value overestimation, with error guarantees matching those of Count-Min Sketch (CMS) [46]. The counters' memory may be reset periodically, similar to sketching epochs, depending on the needs of the application.

5.4.5 Custom Primitives

DTA is easily extensible to other primitives by introducing new translation paths at translators, although they would remain constrained by the limitations imposed by the switching hardware [148]. Some of these limitations could be overcome by implementing the translator logic into Smart Network Interface Cards (SmartNICs) (see §5.7). For example, one could extend DTA to support the collection of sketch-based measurements by crafting a more tailored version of KI. This could allow for either in-network discovery of network-wide heavy hitters, or aggregation of counters at the translator to

Algorithm 12 DTA-to-RDMA translation in Key-Increment

Input: Redundancy N, Key K, Counter C
$\mathit{Bufstart} \leftarrow$ Address to start of RDMA memory buffer
$\mathit{Buflen} \leftarrow$ Number of allocated KeyVal slots
function CraftWrite(n, K, C)
    $\mathit{Slot} \leftarrow h_0(n, K) \bmod \mathit{Buflen}$
    $\mathit{Dest} \leftarrow \mathit{Bufstart} + \mathit{Slot} \times 4$
    Increment Dest by C through RDMA Fetch&Add
end function
for $n = 0$ **to** N **do**
    CraftWrite(n, K, C)
end for

Algorithm 13 Querying the Key-Increment storage

Input: Redundancy N, Key K
Output: C_{winner}
$\mathit{Buflen} \leftarrow$ Number of allocated KeyVal slots
$\mathit{Storage} \leftarrow$ Array size Buflen with $\langle C \rangle$ elements
function GetSlot(n, K)
    $\mathit{Slot} \leftarrow h_0(n, K) \bmod \mathit{Buflen}$
    return $\mathit{Storage}[\mathit{Slot}]$
end function
$\mathit{Counters} \leftarrow$ empty list
for $n = 0$ **to** N **do**
    $\mathit{Counters}[n] \leftarrow$ GetSlot(n, K)
end for
$C_{winner} \leftarrow \min(\mathit{Counters})$

decrease the collection load at compute servers. Additionally, the translator does not have to be a semi-passive data aggregator as presented here, and primitives could be designed to be more active. For example, one could use techniques similar to the ones presented by Gao et al. [69] to derive the network state directly at the translator based on the intercepted telemetry reports, thereby offloading even parts of analysis from the telemetry collectors.

5.4.6 Stochastics of the Key-Write primitive

Similarly to before, this stochastic analysis contains equations and content provided in part by one of my collaborators for our co-authored paper.

Because Key-Write (KW) treats the RDMA memory as a large key-value hash table where only checksums of keys are stored and values may be overwritten over time, we must consider the possibility that when we make a query, we are unable to return an answer, or we may return an incorrect answer. Here, I call the case where we have no answer to return an *empty return*, and the case where we return an incorrect answer a *return error*. The probability of an empty return or a return error depends on the parameters of the system, and on the method we choose to determine the return value. Below are some of the possible tradeoffs, as well as a mathematical analysis of the KW primitive.

Let us first consider a simple example. When a write occurs for a key-value pair, in the hash table N copies of the b-bit key checksum and the value are stored at random locations. Let us assume that checksums are uniformly distributed for any given key throughout this analysis. When a read occurs, let us suppose that we return a value if there is only a single value amongst the N memory locations matching that checksum (the value could occur up to N times, of course).

An empty return can occur, for example, if none of the N locations have the right checksum when we perform a query. That is, all N copies of the key have been overwritten, and none of the N locations currently hold a value for another key with the same checksum. To analyze this case, let us consider the following scenario. Suppose that we have M memory cells total and that

there are $K = \alpha M$ updates of *distinct* keys between when our query key q was last written, and when we are making a query for its values. The Poisson approximation can then be used for the binomial (as is standard in these types of analyses and accurate for even reasonably large M, N, K; see, for example, [28, 150]). Using such approximations, the probability that any one of the N locations is overwritten is given by $(1 - e^{-KN/M})$, and that all of them are overwritten is $(1 - e^{-KN/M})^N$. The probability that all of them are overwritten and the key checksum is not found is approximated by

$$(1 - e^{-KN/M})^N \cdot (1 - 2^{-b})^N = (1 - e^{-\alpha N})^N \cdot (1 - 2^{-b})^N.$$

An empty return can also be encountered if the N cells contain two or more distinct values with the same correct checksum, as there is no way to distinguish the correct and incorrect values.
This probability is lower bounded by

$$\sum_{j=1}^{N-1} \binom{N}{j} (1 - e^{-\alpha N})^j e^{-\alpha N(N-j)} (1 - (1 - 2^{-b})^j) \quad ,$$

and upper bounded by

$$\Big(\sum_{j=1}^{N-1} \binom{N}{j} (1 - e^{-\alpha N})^j e^{-\alpha N(N-j)} (1 - (1 - 2^{-b})^j) \Big)$$
$$+ (1 - e^{-\alpha N})^N (1 - (1 - 2^{-b})^N - N \cdot 2^{-b} (1 - 2^{-b})^{N-1}).$$

The first summation is the probability that at least one of the original N locations is not overwritten, but at least one overwritten location gets the same checksum (we pessimistically assume that it obtains a different value). The second expression adds a term for when all original values are overwritten and two or more obtain the same checksum. Note that we need to give bounds as values in overwritten locations may or may not be the same.

There could be a return error if all N copies of the original key are overwritten and one or more of those cells are overwritten with the same

checksum and the same (incorrect) value. This probability is lower bounded by

$$(1 - e^{-\alpha N})^N N 2^{-b} (1 - 2^{-b})^{N-1},$$

which is the probability that all of the original locations are overwritten and a single overwriting key obtains the checksum, and upper bounded by

$$(1 - e^{-\alpha N})^N (1 - (1 - 2^{-b})^N),$$

the probability that the original locations are overwritten and at least one overwriting key obtains the checksum.

There are many ways to modify the configuration or return method to lower the empty returns and/or return errors, at the cost of more computation and/or more memory. The most natural is to simply use a larger checksum, and I suggest a 32-bit checksum to be appropriate for many situations. However, note that at "Internet scale" rare events will occur, even matching of 32-bit checksums, and so this should be considered when utilizing KW information. One can also use a "plurality vote" if more than one value appears for the queried checksum; additionally one can require that a checksum/value pair occur at least twice among the N values before being returned. (Note that, for example, requiring the consensus of two values can be decided per query without changing anything else; one can decide for specific queries whether to trade off empty returns and return errors this way.) Additional ideas from coding theory [72, 135], including using different checksums for each location or XORing each value with a pseudorandom value, could also be applied. As a default, I suggest a 32-bit checksum and a "plurality vote".

5.4.7 Stochastics of the Postcarding Primitive

Here, we calculate:

(a) The probability that a flow's values fail to be reported because the flow has been overwritten.

(b) The probability that a flow is reported with incorrect values.

We assume that the number of *reports* (up to B postcards that belong to the same flow/packet) since the queried ID is $\alpha \cdot C$.

For (a), we consider several reasons (similar to (5.1)-(5.3)) for failing to report the values and analyze them separately.

- All of the queried flow's chunks are overwritten by other flows and none of them produce valid information. We have the probability that a slot is overwritten bounded by $(1 - e^{-\alpha \cdot N})$. Also, the probability of a given overwritten slot to *not* produce valid information is: $1 - \left((|V| + 1) \cdot 2^{-b}\right)^B$. Therefore, the overall probability of this event is at most

$$(1 - e^{-\alpha \cdot N})^N \cdot \left(1 - \left((|V| + 1) \cdot 2^{-b}\right)^B\right)^N . \tag{5.9}$$

- All the flow's chunks are overwritten and at least two produce valid information arrays that differ. This probability is bounded by:

$$(1 - e^{-\alpha \cdot N})^N \cdot \Bigg(1 - \left(1 - \left((|V| + 1) \cdot 2^{-b}\right)^B\right)^N - N \cdot \left((|V| + 1) \cdot 2^{-b}\right)^B \cdot \left(1 - \left((|V| + 1) \cdot 2^{-b}\right)^B\right)^{N-1}\Bigg). \tag{5.10}$$

- At least one chunk (but not all) is overwritten and produces valid information. This error probability is at most

$$\sum_{j=1}^{N-1} \binom{N}{j} \cdot (1 - e^{-\alpha \cdot N})^j \cdot e^{-\alpha \cdot N(N-j)} \cdot \left(1 - \left(1 - \left((|V| + 1) \cdot 2^{-b}\right)^B\right)^j\right). \tag{5.11}$$

Next, we analyze the probability of replying incorrectly (b). This happens when all the queried key's chunks are overwritten and all valid chunks hold the same information. Then, the probability of such an error is at most:

$$(1 - e^{-\alpha \cdot N})^N \cdot N \cdot \left((|V| + 1) \cdot 2^{-b}\right)^B . \tag{5.12}$$

Consider a numeric example to contrast these results with using KW for each report of a given packet. Specifically, suppose that we are in a large data center ($|V| = 2^{18}$ switches) and want to run path tracing by collecting all (up to $B = 5$) switch IDs using $N = 2$ redundancy. Further, let us set $b = 32$-bit per report (as is a common standard) and compare it with 64 bits (32 for the key's checksum and 32 bits for the switch ID) used in KW and that $C \cdot \alpha$ packets' reports were collected after the queried one, for $\alpha = 0.1$. We have that the probability of not outputting a collected report (5.9-5.11) is at most 3.3% and the chance of providing the wrong output (5.12) is lower than 10^{-22}. In contrast, using KW for postcarding gives a false output probability of $\approx 8 \cdot 10^{-11}$ (in at least one hop) using twice the width per entry! This improvement is due to a couple of reasons. First, we leverage the difference between the number of switches (e.g., $|V| = 2^{18}$) and the width of the value field (hardcoded at 32-bits per the INT standard [74]). Second, we leverage the fact that each packet carries multiple (e.g., $B = 5$) reports to amplify the success probability and mitigate the chance of wrong output. Further, for reports for which we can buffer all postcards at the translator (which depends on the allocated memory and the number of simultaneous postcard reports generated), this approach reduces the number of RDMA writes by a factor of B.

5.5 DTA Implementation

My codebase includes approximately $5K$ lines of code divided between the logic for the DTA reporter (§5.5.1), the translator (§5.5.2), and collector RDMA service (§5.5.3). The hardware resource footprints are presented later in Sections §5.6.3 and §5.6.4. I have released DTA in open-source [122], and I

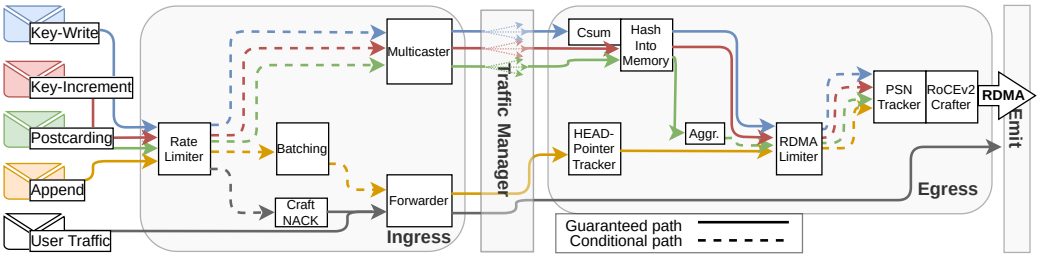


Figure 5.10: A translator pipeline with support for Key-Write, Key-Increment, Postcarding, and Append. Five paths exist for pipeline traversal, used to process different types of network traffic in parallel while efficiently sharing pipeline logic.

am in contact with multiple labs working on follow-up research based on this codebase.

5.5.1 Reporter

The reporter takes ≈ 700 lines of P4_16 for the Tofino Application Specific Integrated Circuit (ASIC). Controller functionality is written in ≈ 100 Python lines, and is responsible for populating forwarding tables and inserting collector Internet Protocol (IP) addresses for the DTA primitives.

DTA reports are generated entirely in the data plane and the logic is in charge of encapsulating the telemetry report into a UDP packet followed by the two DTA-specific headers where the primitive and its configuration parameters are included.

5.5.2 Translator

The translator has a control program written in 800 lines of Python that runs on the switch CPU. This control program sets up the RDMA connection to the collector by crafting RDMA Communication Manager (RDMA-CM) packets, which are then injected into the ASIC and forwarded to the RDMA NIC at the collector. Connection-initializing replies from the collector are

forwarded from the translator ASIC down to the control program for parsing and processing to establish the RDMA queue pairs.

The translator ASIC pipeline (visualized in Figure 5.10) is written in $2K$ lines of P4_16 for the Tofino ASIC. This pipeline includes support for internal line-rate processing of the DTA primitives, RDMA generation, basic user-traffic forwarding, as well as RDMA queue-pair resynchronization and rate limiting to ensure stable RDMA connections in case of congestion events at collectors' NICs. Rate limiting can be configured to generate a Negative ACKnowledgment (NACK) sent back to a reporter in case of a dropped report during these congestion events.

The RDMA logic is shared by all primitives. This includes lookup tables filled with RDMA metadata, Static RAM (SRAM) storage for the queue pair packet sequence numbers, and the task of crafting RoCEv2 headers. The DTA packets themselves are used as the base for RDMA generation. This is done by completely substituting the DTA headers with the specific RoCEv2 headers required by the DTA operation.

The redundancy in Key-Write (KW), Key-Increment (KI), and Postcarding is generated by the packet replication engine through multicasting (*Multicaster* in Figure 5.10). The switch CPU crafts specific multicast rules to force the ASIC to emit several packets at the correct egress port as triggered by a single DTA ingress.

Key-Write and **Key-Increment** both follow the same fundamental logic, with the main difference being the RDMA operation that they trigger. KW triggers RDMA Write operations, while a KI triggers RDMA Fetch-and-Add. Both cause N packet injections into the egress pipeline, using the multicast technique. The Tofino-native Cyclic Redundancy Check (CRC) engine is used to calculate the N memory locations as well as a concatenated $4B$ checksum in the case of KW translation. Carefully selected CRC polynomials are used to create several independent hash functions using the same underlying CRC engine[3]

Postcarding uses an SRAM-based hash table with 32K slots storing fixed-

[3]There is further support for longer checksums, at the cost of reduced space efficiency, if query errors are not acceptable even in very rare cases.

size 32-bit payloads. The Tofino-native CRC engine is used for indexing and value encoding. The hop-specific checksums are implemented through custom CRC polynomials instead. Emissions are triggered either by a collision or when a row counter reaches the path length. Note that for efficient implementation, the RDMA payload sizes must be powers of 2. This is due to the use of bitshift-based multiplication during address calculation. Consequently, for 5-hop paths, chunk sizes are increased from 20 bytes (5×4 bytes) to 32 bytes. This adjustment sacrifices storage efficiency to achieve a reduced switch footprint.

Append has its logic split between ingress and egress, where ingress is responsible for building batches, and egress tracks per-list memory pointers. Batching of size B is achieved by storing $B - 1$ incoming list entries into SRAM using per-list registers. Every Bth packet in a list will read all stored items, and bring these to the egress pipeline where they are sent as a single RDMA Write packet. Lists are implemented as ring buffers, and the translator keeps a per-list head pointer to track where in server memory the next batch should be written. My open-sourced prototype supports tracking up to $131K$ simultaneous lists.

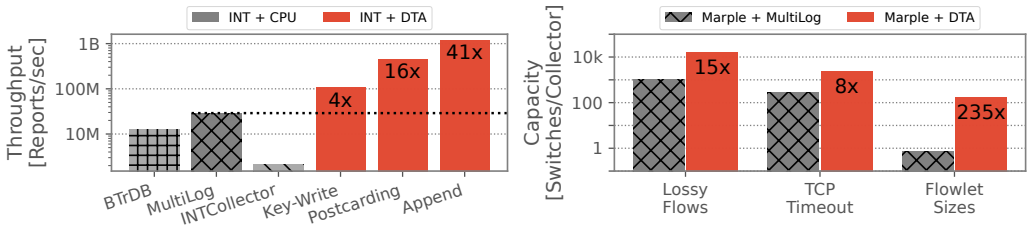
5.5.3 Collector

The collector is written in $1.3K$ lines of C++ using standard Infiniband RDMA libraries and has support for per-primitive memory structures and querying the reported telemetry data. The collector can host several primitives in parallel using unique RDMA-CM ports, and advertise primitive-specific metadata to the translator using RDMA-Send packets.

5.6 Evaluation

In this section, I show that:

- DTA supports very high collection rates (§5.6.1).
- DTA imposes a negligible memory pressure at collectors (§5.6.2).



(a) Generic $4B$ INT collection. (b) Diverse Marple queries.

Figure 5.11: A performance comparison of DTA against state-of-the-art CPU-based collectors. These use 16 cores for data ingestion, while DTA essentially bypasses the CPU entirely for data ingestion by using RDMA. (b) MultiLog vs DTA when using Marple as a monitoring system running on switches.

- DTA is lightweight (§5.6.3, §5.6.4).
- DTA's primitives are fast (§5.6.5, §5.6.6, §5.6.7).

I used two x86 servers connected through a BF2556X-1T [161] Tofino 1 [102] switch with 100G links. Both servers mount 2x Intel Xeon Silver 4114 CPUs, 2x32GB DDR4 Random Access Memory (RAM) @ 2.6GHz, and run Ubuntu 20.04 (kernel 5.4). One server acts as a DTA report generator using TRex [37]. The other, equipped with an RDMA-enabled Mellanox Bluefield-2 DPU [165], acts as the collector. Here, server Basic Input/Output System (BIOS) has been optimized for high-throughput RDMA [106], and all RDMA-registered memory is allocated on 1GB huge pages. My experiments consistently showed a 0% packet loss rate at the translator-collector link. The reporter-translator path is more complex to emulate, and these evaluations assume a 0% loss rate for incoming reports. This is done to isolate the performance of DTA since reporter-translator forwarding of reports is outside my research scope.

5.6.1 DTA in Action

Here I first investigate if DTA scales better than CPU-based collectors in the presence of telemetry volumes generated by large-scale networks. To do so, I compare the performance of DTA and state-of-the-art CPU-collectors when

coupled with two different monitoring systems: INT [74,116] and Marple [159]. Here, I use a DTA configuration with $N = 1$ and batching of size 16, while CPU-collectors use 16 dedicated CPU cores in the same Non-Uniform Memory Access (NUMA)-node.

The collectors in Figure 5.11a collect generic $4B$ INT reports that are available for offline queries using the flow 5-tuple as the key. I test INTCollector [207], to the best of my knowledge the only open source INT collector that uses InfluxDB for storage. I also study BTrDB [10], and the state-of-the-art solution for high-speed networks, Confluo, based on MultiLog technology. Key-Write (KW) inserts each report into its key-value store, and Postcarding assumes 5-hop aggregation with no intermediate reports. Append instead inserts the reports into one of the available data lists[4]. As Figure 5.11a shows, DTA improves on key-based INT collection by at least 4x, or up to 16x when aggregating the postcards into 5-hop tuples, with even higher performance gains if pre-categorized and chronological storage through Append suffices.

I also integrated Marple with DTA and MultiLog and configured them to support the same queries against the collected data (i.e., Lossy Flows, Transmission Control Protocol (TCP) Timeout, and Flowlet Sizes). Here, *Lossy Flows* reports high loss rates together with their corresponding flow 5-tuples, and DTA uses the Append primitive to store the data chronologically in several lists, allowing operators to retrieve the most recently reported network flows with packet loss rates in one of several ranges. *TCP Timeouts* reports the number of TCP timeouts per flow in recent time, and DTA uses the KW primitive to allow operators to query the number of timeouts experienced by any arbitrary flow. *Flowlet Sizes* reports flow 5-tuples together with the number of packets in their most recent flowlets, and DTA appends the flow identifiers to one of the available lists to allow the construction of per-flow histograms of flowlet sizes.

In contrast to the earlier chapters, this section uses real data center traffic [20], since these experiments are short-lived, and the length of available data center traces suffices. I found that DTA increases the number of Marple

[4]The reporters were configured to arbitrarily select a list ID in Append per report, emulating some underlying data categorization.

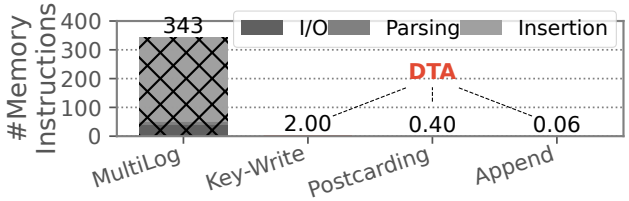


Figure 5.12: Average number of memory instructions per report for ingestion of INT postcards.

reporters (i.e., network switches) that a collector can support before the rate of data generation overwhelms the collector, and I show this result in Figure 5.11b. Their queries cost as well as their performances are analyzed in later sections (§5.6.5, §5.6.7).

Takeaway: DTA improves on data collection speeds compared with CPU-based collectors by *one to two orders of magnitude* when integrated with state-of-the-art telemetry systems while supporting the same types of queries.

5.6.2 Reduced Memory Pressure

In Figure 5.12, I present the average number of memory instructions required per report for the DTA primitives when configured with a redundancy level of 2, path length of 5 hops, and batch size of 16 elements. These parameters were chosen according to the previous discussions in Section 5.4. Intuitively, changes in redundancy level and batch sizes linearly impact memory instruction rate, while the path length has a sub-linear impact due to early emission.

DTA imposes a low pressure on memory. This is achieved mostly because no accesses are needed for I/O and report parsing, regardless of the indexing scheme used. Some DTA primitives use less than a single memory instruction per report on average, owing to their aggregation and batching techniques, which can intelligently insert several reports simultaneously with a single RDMA operation. For example, KW, the primitive that imposes the heaviest load on memory, needs just 0.58% as many accesses as MultiLog.

Takeaway: DTA *significantly* reduces the number of memory accesses re-

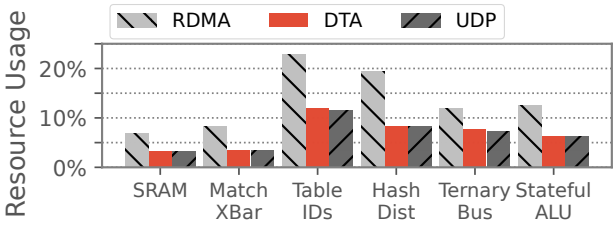


Figure 5.13: Hardware resource costs of a DTA Reporter compared to an RDMA-generating reporter, and a baseline UDP-based reporter. Note how DTA imposes an almost identical resource footprint to UDP.

quired for report ingestion.

5.6.3 Cost of Generating DTA Reports

I compared the hardware costs associated with generating DTA reports against either directly emitting RDMA calls from switches, or creating UDP-based messages as generally done by CPU-based collectors. For this, I used a switch implementing a simple INT eXport Data (INT-XD) system and, in Figure 5.13, I show the cost associated with the change of its report-generation mechanism.

Here, we can see that DTA is nearly as lightweight as UDP, while RDMA generation is much more expensive. Since both memory and computational resources are statically assigned to fixed computations and dependencies, the actual resource cost is known at compile-time and is unchanging during run time. Additionally, my design does not need *any* recirculation, which means that the translator can ingress reports at line-rate thanks to the properties of the target hardware architecture.

Takeaway: DTA halves the resource footprint of reporters compared with RDMA-generating alternatives, and has a *similar resource footprint to simple UDP generation*. The downside is the higher cost for translator logic (Table 5.3).

	SRAM	Match Crossbar	Table IDs	Ternary Bus	Stateful ALU
Base footprint	13.2%	10.6%	49.0%	30.7%	25.0%
Batching	+3.2%	+7.2%	+7.8%	+7.8%	+31.3%

Table 5.3: Resource footprint of a translator in Tofino while supporting Key-Write, Postcarding, and Append. Append is batching 16x4B reports.

5.6.4 Cost of Translation

RDMA generation, including DTA report interception and primitive processing, is entirely located within the translator. These are relatively heavy computations, and the cost of translation has to be investigated. Table 5.3 shows the resource usage of the translator, alongside the additional costs of including Append batching.The footprint of the DTA translator is mainly due to its concurrent built-in support for several different primitives. Application-dependent operators might reduce their hardware costs by enabling fewer primitives.

Batching of Append data has a relatively high cost in terms of memory logic (Stateful Arithmetic Logic Unit (ALU)), due to my non-recirculating RDMA-generating pipeline requiring access to all $B-1$ entries during a single pipeline traversal. It is worth noting that batching also has the potential for a tenfold increase in collection throughput, and I conclude that it is a worthwhile tradeoff. A compromise is to reduce the batch sizes, as they linearly correlate with the number of additional stateful ALU calls.

Deploying multiple simultaneous Append-lists does not require additional logic in the ASIC, it just necessitates more statefulness for keeping per-list information (e.g., head-pointers and per-list batched data). Note that the actual SRAM footprint of the translator is small, and my experiments show that the translator can support hundreds of thousands of simultaneous lists for complex setups, which is much more than the 255 lists included at the time of evaluation. The number of lists has not shown any impact on the

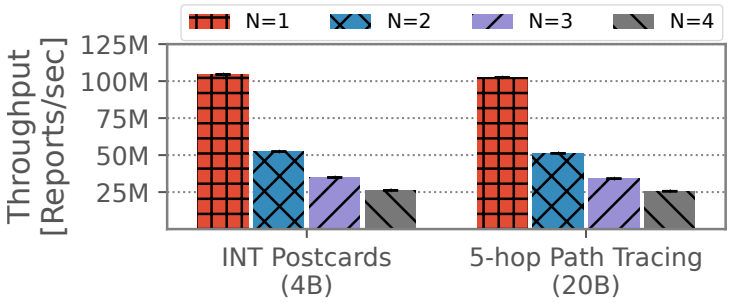


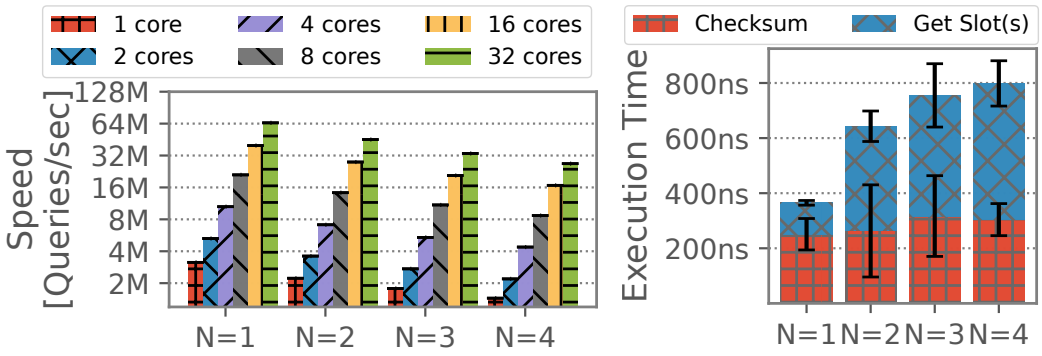
Figure 5.14: Per-flow path tracing collection rates, using the DTA Key-Write primitive, either as INT-XD/MX postcards (4B) or full 5-hops path as in INT-MD (20B).

runtime performance.

Takeaway: A translator pipeline which simultaneously supports the KW, Postcarding, and Append primitives fits in first-generation programmable switches, while *leaving a majority of resources freed up for other functionality.* Batching can impose a high toll on the Stateful ALUs.

5.6.5 Key-Write Performance

I have benchmarked the collection performance of the DTA KW primitive using INT as a use case. I instantiated a $4GiB$ key-value store at the collector and had the translator receive either $4B$ or $20B$ encapsulated INT messages from the reporter (my traffic generator). The former case emulates the scenario of having INT working in postcard mode with event detection (so some hops may not generate a postcard), while the latter reproduces an INT path tracing configuration on a 5-hops topology where the last hop reports data to a collector. I repeated the test using different levels of redundancy (N) and reported the results obtained in Figure 5.14. Notice the expected linear relationship between the throughput and level of redundancy since each incoming report will generate N RDMA packets towards the collector. However, one might still prefer the performance tradeoff against the increased data robustness in the collector storage, which allows for successful queries



(a) Query rate (b) Per-query breakdown

Figure 5.15: Key-Write primitive querying performance.

against much older telemetry reports. Furthermore, the collection rate is unaffected by the increase in the telemetry data size until the $100Gbps$ line rate is reached. I saw that this was the case for telemetry payloads of $16B$ or larger in my tests.

Takeaway: KW can collect $100M$ *INT reports per second* and its performance depends on the redundancy level.

Key-Write Query Speed

Querying for data stored in the key-value store using the KW primitive requires the calculation of several hashes. Here I evaluate the *worst case* performance scenario when the collector has to retrieve every redundancy slot before being able to answer a query. Specifically, I queried $100M$ random telemetry keys, with a key-value data structure of size $4GiB$ containing $4B$ INT postcards data alongside $4B$ concatenated checksums for query validation. Figure 5.15a shows the speed at which the collector can answer incoming telemetry queries using various redundancy levels (N).

KW query processing can be easily parallelized, and I found the query performance to scale near-linearly when I allocated more cores for processing. For example, 4 cores could query 7.1 million flow paths per second with $N = 2$, while 8 cores manage 14.2 million queries per second.

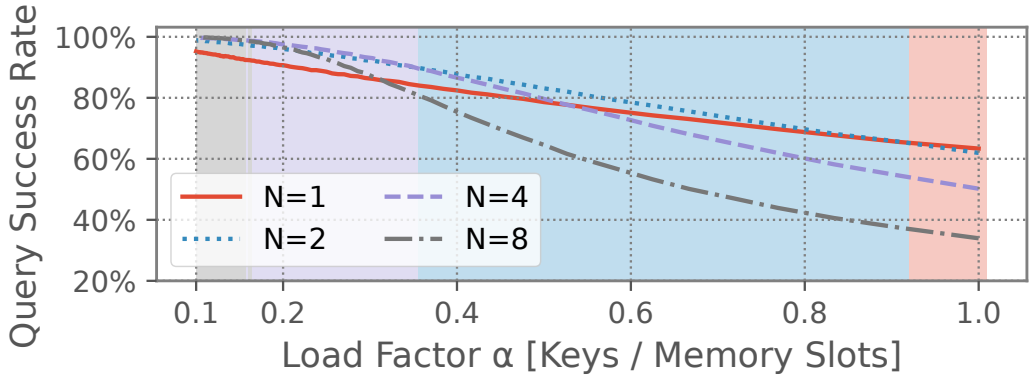


Figure 5.16: Average query success rates delivered by the Key-Write primitive, depending on the key-value store load factor and the number of addresses per key (N). The background color indicates optimal N in each interval.

Figure 5.15b shows the time breakdown serving queries. Most of the execution time is spent calculating CRC hashes, for either verifying the concatenated checksum (*Checksum*), or calculating memory addresses of the N redundancy entries (*Get Slot*). The query performance is therefore highly impacted by the speed of the CRC implementation[5], and more optimized implementations should see a performance increase.

Takeaway: Because of RDMA, my Key-Value store *can insert entries faster than the CPU can query*. The performance of the CRC implementation plays a key role.

Key-Write Redundancy Effectiveness

The probabilistic nature of KW cannot guarantee final queryability on a given reported key due to hash collisions with newer data entries. I show in Figure 5.16 how the query success rate[6] depends on the load factor (i.e., the total number of telemetry keys over available memory addresses), and the redundancy level (N). There is a clear data resiliency improvement by

[5]I used the generic Boost libraries' CRC: `https://www.boost.org/`.

[6]The query success rate is defined as the probability at which a previously reported key can be queried from the key-value store.

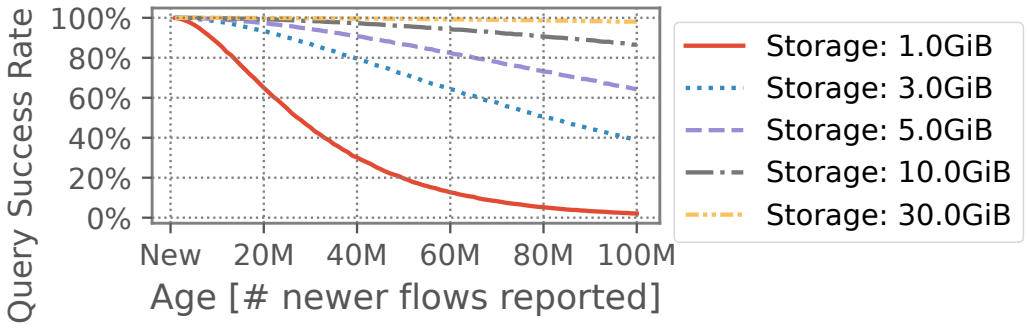


Figure 5.17: DTA Key-Write ages out eventually. This figure shows INT 5-hop path tracing queryability of 100 million flows at various storage sizes.

having keys write to $N > 1$ memory addresses when the storage load factor is in reasonable intervals. When the load factor increases, adopting more addresses per key does not help because it is harder to reach consensus at query time. The background color in Figure 5.16 indicates which N delivered the highest key-queryability in each interval.

Higher levels of redundancy improve data longevity, but at the cost of reduced collection and query performance as demonstrated previously in Figures 5.14 and 5.15. Determining an optimal redundancy level therefore has to be a balance between enhanced data queryability and a reduction in primitive performance. $N = 2$ is a generally good compromise, showing great queryability improvements over $N = 1$ at a range of densities.

Takeaway: Increasing the redundancy of all keys does not always improve the query success rate in KW. An optimal redundancy should be set on a case-by-case basis.

Key-Write Data Longevity

Data reported by the KW primitive will age out of memory over time due to hash collisions with subsequent reports, which overwrites the memory slots. Figure 5.17 shows the queryability of randomly reported INT 5-hop path tracing data (i.e., $20B$) at various storage sizes and report ages, with redundancy level $N = 2$ and $4B$ checksums. For example, a key-value storage

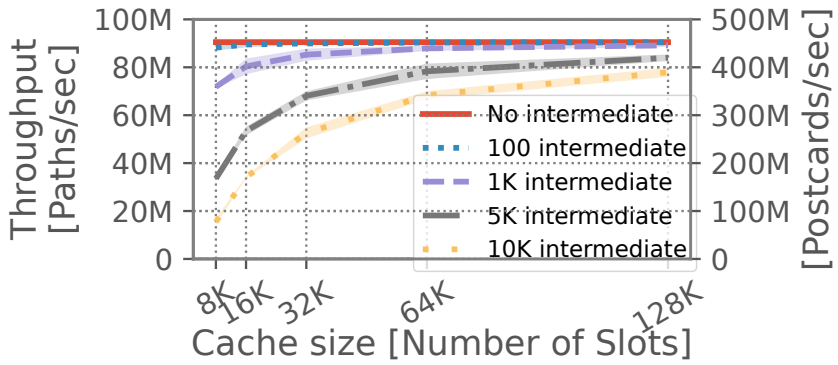


Figure 5.18: INT-XD/MX postcard collection, using the DTA Postcarding primitive at various buffer/cache sizes. A report is defined as a successfully aggregated 5-hop path (containing 5 postcards, one per hop).

as small as 3GiB is enough to deliver 99.3% successful queries against flows with as many as 10 million subsequently reported paths, which however falls to 44.5% when 100 million subsequent flows are stored in the structure. However, increasing storage to 30GiB would allow an impressive 99.99% query success rate for paths with 10 million subsequent reports, or 98.2% success even for flows as old as 100 million subsequent reports.

Retrospection requires historical persistent storage of telemetry data. Due to this, telemetry operators should perform a periodic transfer of data from the DTA transient storage into persistent storage if retrospection is essential.

Takeaway: It is possible to record data from around 10M flows in the key-value store while maintaining a 99.99% queryability with just 30GiB of storage.

5.6.6 Postcarding Performance

The Postcarding primitive has been benchmarked for aggregating and collecting INT-XD/MX postcards across 5-hop network paths. The number of other flows appearing at the translator while aggregating per-flow postcards increases the risk of premature buffer emission. Figure 5.18 shows us the effect that the number of intermediate flows and the size of the buffer has

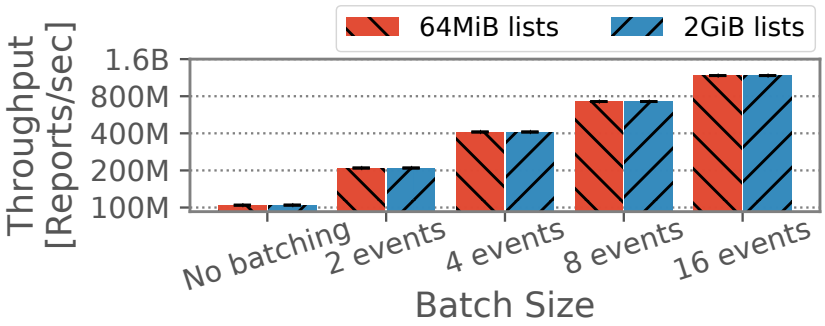


Figure 5.19: Telemery event-report collection, using DTA Append and different batch sizes. Performance increases linearly with batch sizes until we achieve line rate with batches of $4x4B$. The collection speed is not impacted by the list sizes.

on the aggregation performance, with a maximum achieved collection rate of $90.5 MPaths/s$ ($452.5 MPostcards/s$).[7]

Comparing the performance to KW in Figure 5.14, where we would need 5 different reports to collect a full path, we see a significant performance gain by the Postcarding primitive.

For a straightforward understanding of the performance of the primitive, no path length variations are included in this experiment as this would lead to increased experimental complexities such as path length distributions. Intuitively, the performance of Postcarding is sublinearly impacted by the path length, and longer path lengths generally lead to a higher collection rate (thanks to the increased efficiency of in-translator batching).

Takeaway: The performance of Postcarding depends on the rate of buffer collisions in the translator during the aggregation phase and can *improve upon the best-case Key-Write performance by up to* $4.3x$ for 5-hop collection.

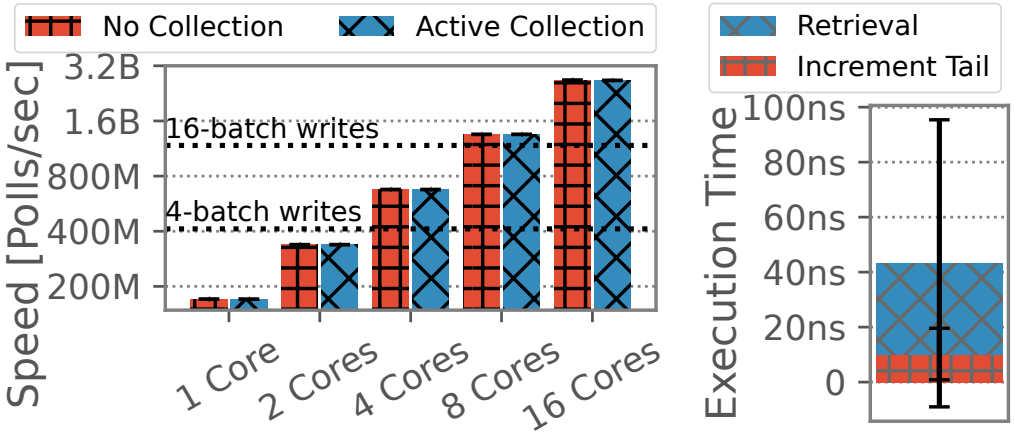
[7]Early emissions (i.e., path-reports with missing postcards) are counted as failures in this test despite being potentially useful (e.g., knowing 4 out of 5 hops in a path), and are not included in the collection throughput.

5.6.7 Append Performance

I have benchmarked the performance of the Append primitive for collecting telemetry event reports, both at different batch sizes and the size of the allocated data list, while reporting data into a single list. The results are shown in Figure 5.19.

As expected, there is no performance impact from different report sizes until we reach the line rate of $100G$ for large batch sizes after, which the performance increases sublinearly. The results in Figure 5.19 show this effect for $4B$ queue-depth reports, where we reach line rate at batches of 4. The DTA base performance is bounded by the RDMA message rate of the NIC, which is the current collection bottleneck of my system, and the high performance of the Append primitive is due to including several reports in each memory operation. Performing equivalent tests with up to $131K$ parallel lists showed a negligible performance impact.

Takeaway: The Append primitive is able to collect *over* 1 *billion telemetry event reports per second*.



(a) Polling rate.

(b) Poll breakdown.

Figure 5.20: Append primitive querying performance. Append-lists are queried either while collecting no reports or at 50% capacity (while collecting *600M* reports per second). Collection has a negligible impact on the data retrieval rate, and the processing rate scales near-linearly with the number of cores. The dotted lines show the maximum collection rates at different batch sizes.

Append Data Access Rate

Figure 5.20a shows the raw list polling rates, which is the speed at which appended data can be read into the CPU for processing. I assume that collection runs simultaneously to the CPU retrieving data from the lists in real-time, by having the translator process 600 million Append operations per second in batches of size 16, which approximates collection at half capacity. Simultaneously collecting and processing telemetry data show no noticeable impact on either collection or processing, showing that DTA is not memory-bounded even at this speed[8].

Extracting telemetry data from the lists is a very lightweight process, as shown in Figure 5.20b, requiring a pointer increment, possibly rolling back to

[8]DTA is neither memory- nor CPU bounded in these tests, regardless of the collection rate, but is instead limited by the message rate of the network card

the start of the buffer, and then reading the memory location. I allocated multiple lists equal to the number of CPU cores used during the test to prevent race conditions at the tail pointer[9].

My tests showed that just 8 cores proved capable of extracting every telemetry report even when large batches reported at maximum capacity. This leaves us with much processing power for complex real-time telemetry processing. We see that the collector can even retrieve list entries faster than the RAM clock speed, likely thanks to cache prefetching.

Takeaway: The CPU retrieves appended reports *faster than they can be collected* (Figure 5.19), with margin left for further processing.

5.7 Discussion

This section provides a brief discussion of DTA and proposes future research into the techniques presented in this chapter.

5.7.1 Generality and Scope

DTA is not intended to be a competitor of existing data plane assisted monitoring systems [18, 74, 77, 116, 159, 190, 201, 201, 230]. These either focus on extracting new metrics or reducing the costs of telemetry monitoring through intelligent pre-processing and filtering within the switching ASIC. Nevertheless, these systems generate a significant amount of telemetry information, especially with large-scale networks, multiple queries, and/or fine telemetry granularities (Table 5.1).

DTA can be coupled with existing telemetry systems and serve as an interface between the on-switch monitoring functions and the telemetry analysis back-end in the control plane. To achieve broad compatibility with a variety of monitoring solutions, I have designed several generic and highly flexible primitives to simplify the integration of DTA into both existing and

[9]DTA does not necessarily limit the number of polling cores per list, and several cores can poll a single list by, for example, assigning them a set of non-overlapping indexes in the list and using per-core tail pointers.

future telemetry environments. As a consequence, with DTA, I replace only the report ingestion mechanism of the telemetry collector (e.g., DPDK along with data structure population), not the rest of the collector (e.g., data analysis and decision-making). For example, it is possible to couple the streaming analysis engine of Sonata [77] with DTA: in this scenario, DTA is in charge of transferring data from switches to collector's memory, while the original Sonata's engine performs analysis on the received data. For a more extensive list of examples, refer to Table 5.2 that recap how DTA can be integrated into various telemetry systems to enhance their performances.

5.7.2 In-NIC Translation

There are two main approaches that I have considered on where to deploy the translator: a SmartNIC located at the collector and the last-hop switch (which I explored in this chapter). A SmartNIC would allow me to completely remove RDMA traffic: the NIC data plane would process incoming DTA packets and instead translate them directly into local Direct Memory Access (DMA) calls. Exploring DTA translation in SmartNICs is left for future work. Nevertheless, I believe that my P4 implementation can be a starting point for P4-capable NICs [192].

Further, one benefit of placing translation at the last-hop Top of Rack (ToR) switch is that it allows DTA collectors to function with relatively cheap and high-performant commodity RDMA-capable NICs, without requiring NIC extensions with DTA translation capacities. Building translation into the ToR is more efficient in a rack with multiple collectors, all managed by a single translator since the actual translation logic only has to reside in a single device. Further, one could extend in-ToR translation with more complex functionality, such as dynamically load balancing between collectors, or fragmenting the storage buffers across multiple physical machines.

5.7.3 Multiple Collectors

It is beneficial to enable collection at multiple servers for scalability or resiliency. DTA can scale horizontally by deploying additional collectors and relies on reporter-based load balancing of telemetry data among collectors. However, we need to ensure that the load balancing is stateless and can be centrally recalculated, to ensure scalability and efficiency in finding the storage locations for queries. The destination IP addresses of DTA reports are decided on a per-primitive basis. The KW, Postcarding, and KI primitives are designed as distributed key-value stores. Reporters use a hash of the telemetry key to determine the destination collector for storing the information. This design allows horizontal scaling of collection capacity by deploying more servers and updating the collector-mapping lookup tables hosted in each reporter. Append traffic selects a collector based on the chosen list ID, as decided by pre-loaded lookup tables. This ensures that all per-category telemetry data is efficiently aggregated in a single location, and telemetry scaling can be achieved by deploying additional lists to host data for the Append primitive.

Additionally, network operators could decide to collect query-specific measurements at specific collectors, storing relevant measurements near each other to enhance the query performance. However, this is a choice that is not DTA-specific, and already exists regardless of which telemetry collection solution is being used.

5.7.4 Flow Control in DTA

Best-effort transport protocols, e.g., UDP, are used by many well-known telemetry systems (e.g., [38, 103]). Similarly, DTA does not assure reliable delivery. However, it can be used in conjunction with flow control mechanisms that allow for lossless delivery of data [73, 97].

5.7.5 Query-Enhancing Extensions

In some cases, queries may be known ahead of time, in which case a translator can aid in their processing. For example, while switches can measure the queuing latency of a flow, we are often interested in knowing the end-to-end delay [181], which can be expressed as follows:

```
SELECT flowID,path WHERE SUM(latency) > T
```

Knowing the query ahead of time, DTA translators can wait for postcards from all switches through which the SYNchronization (SYN) packet of the flow was routed, sum their latency, and report the flow and path only when the combined latency is over a threshold T. A method such as this could be used in conjunction with the Append primitive to populate a list of high-latency (flowID, path) tuples alongside a complete list of all network/sector-wide tuples.

5.7.6 Push Notifications

An advantage CPU-based collectors have over DTA is that the CPU can trigger analysis tasks as soon as it receives reports. In the case of DTA, for key-value store operations, the CPU must first find out if new data has been written into the memory; however, I assume for Append operations that the CPU is continuously monitoring the lists to allow for equivalent reactivity to CPU-based solutions. Additionally, DTA packets can include an *immediate flag*, allowing a translator to notify a collectors' CPUs of new data arrivals through RDMA immediate interrupts (e.g., indicating issues in a flow). Determining which reports should carry this flag is outside the scope of DTA.

5.7.7 The Next Bottleneck

DTA significantly reduces the cost of telemetry ingestion mainly by bypassing any CPU processing. In my experiments, the new bottleneck is the message rate of the RDMA NICs at the collectors. To address this message rate

limitation, DTA already supports multi-NIC collectors. Future RDMA-capable NICs will likely have higher message rates, leading to an even higher DTA performance.

A possible future bottleneck is the memory speed where we store the telemetry data structures. However, current-generation DRAM can achieve billions of memory transfers per second and is likely to increase further in the future. Therefore, telemetry ingestion might no longer be seen as the main bottleneck in telemetry systems going forward, if the CPU is bypassed. Instead, given the increasing sophistication and complexity of data analysis tools, the de facto bottleneck might instead be the rate at which reports can still be meaningfully analyzed in real time.

Alternatively, in a data lake deployment, where data is passively collected without real-time analysis, the rate at which data is transferred from transient DTA storage to persistent, long-term storage in queryable data structures may become a bottleneck. This issue is particularly pronounced if the transfer process involves complex processing and re-indexing of the data.

5.7.8 Security Considerations

The introduction of DTA introduces new complexities to networks, potentially creating additional attack vectors and security vulnerabilities. For instance, malicious actors could forge false telemetry data to disrupt network control operations or to conceal ongoing attacks. Since DTA reports are encapsulated in UDP, they may be vulnerable to UDP Session Hijacking attacks making such attacks plausible [107].

To mitigate these risks, several strategies can be considered. One approach is to have reporters cryptographically sign telemetry reports before transmission, ensuring their authenticity and integrity. Another strategy is to enforce specific network paths for telemetry reports, ensuring that these reports originate only from the network switches that are stated as senders, and are not maliciously injected into the network through a network ingress link.

This issue is not necessarily DTA-specific, and impacts other systems such

as INT. I leave research into mitigation strategies as future work.

5.7.9 Batching Trade-offs

Batching greatly increases telemetry collection rates. However, this requires in-ASIC statefulness. Even though the SRAM footprint of batching is relatively small (3% in my §5.6.4 example), I saw a significant impact on the available memory logic. Each memory operation is limited to a 32-bit bus, requiring multiple memory operations to process batch entries larger than $4B$. Complex deployments with large telemetry payloads might therefore have to reduce the batch size to free up switch memory logic (e.g., a batch with $8B$ entries might halve the batch size compared with $4B$ entries to keep a similar footprint). One possible alternative is to reduce the hardware resources by allowing RDMA-crafting packets to traverse the pipeline multiple times while retrieving the batch. For example, batches may use half as much memory logic if allowed to recirculate once, by re-using memory logic between pipeline traversals. This also allows us to increase the batch sizes further to reduce the load on the collector's NIC, at the cost of increased egress-pipe traffic during RDMA-creation for Append operations.

5.8 Chapter Summary

In this chapter, I addressed the third research objective of my dissertation: "Alleviate the Telemetry Collection Bottleneck". To achieve this, I developed and presented DTA, a high-performance telemetry collection system designed to enhance the cost vs. insight tradeoff in fine-grained network telemetry.

DTA introduces several key innovations. First, it leverages RDMA to enable direct memory access, significantly increasing collection rates by an order of magnitude compared to current solutions. A pivotal innovation within DTA is the translator, which allows the extension of RDMA capabilities at line-rate to support queryable data structures suitable for telemetry storage. This translator intercepts telemetry data and efficiently converts it into RDMA calls, creating an abstraction that allows high-speed data structure population without necessitating custom NIC changes.

Through this abstraction, DTA introduces novel reporting primitives that ensure seamless integration with existing telemetry mechanisms like INT and Marple. These primitives enable efficient and flexible data handling tailored to various telemetry requirements.

My evaluation of DTA demonstrated substantial improvements in collection rates and efficiency. Specifically, DTA can process and aggregate over 400 million INT reports per second, a 16x improvement over state-of-the-art CPU-based collectors. Additionally, when collecting data sequentially, DTA can handle up to a billion reports per second, representing a 41x increase in performance.

Chapter 6

Conclusion

This chapter concludes the research by highlighting several advancements in network telemetry that significantly enhance the accuracy and granularity of telemetry reports. Through the development of novel techniques such as *spatiotemporal disaggregation*, *FlowLiDAR*, and *Direct Telemetry Access (DTA)*, we have addressed critical challenges related to deployment, accuracy, and data collection in modern network environments, while ensuring compatibility with high-speed switches.

This concluding chapter summarizes the main contributions of this work and their relation to my research objectives.

6.1 Objectives Revisited

This dissertation aimed to address three primary research objectives: making sketches network-wide deployable, improving the cost vs. accuracy tradeoff in sketches, and alleviating the telemetry collection bottleneck.

1. Make Sketches Network-wide Deployable: Chapter 3 achieved this objective through the development of *spatiotemporal disaggregation*, which allows sketches to be fragmented and deployed across multiple network switches in heterogeneous environments. This approach enhances estimation

accuracy and provides failure resilience, as demonstrated by the creation and evaluation of DiSketch, a fully disaggregatable frequency estimator.

2. Improve the Cost vs Accuracy Tradeoff in Sketches: Chapter 4 enhanced the memory efficiency of sketching by introducing FlowLiDAR, a solution that decouples flow identification storage from the limited on-switch memory using innovative techniques such as lazy Bloom Filters (BFs) and differential flow detection. These innovations significantly reduce the memory required for accurate frequency estimation, thereby improving the memory efficiency of sketch-based monitoring.

3. Alleviate the Telemetry Collection Bottleneck: Chapter 5 addressed the collection bottleneck by introducing DTA. This system leverages in-network pre-processing and indexing of reports to enable CPU-bypassing collection via Remote Direct Memory Access (RDMA), significantly increasing telemetry collection rates. This system was validated through implementation on commodity hardware and demonstrated significant performance improvements.

6.2 Summary of Contributions

The research chapters collectively demonstrated the successful realization of the initial research objectives. In this section, I provide a detailed overview of my primary contributions towards achieving each research objective.

6.2.1 Network-wide Deployable Sketches

Chapter 3 made significant strides towards achieving the first research objective of making sketches network-wide deployable to grant placement-agnosticism, failure tolerance, and accuracy enhancements. The primary contributions in this regard are:

Spatiotemporal Disaggregation: I developed a novel method called *spatiotemporal disaggregation*, which enables robust fragmentation and deployment of sketches across multiple network switches. This technique leverages network-wide resources to enhance estimation accuracy, provide failure resilience, and reduce sketch extraction delays. Spatiotemporal disaggregation is developed for dynamic deployment in highly heterogeneous environments, where both the available per-node memory as well as traffic load and flow size distributions might vary.

DiSketch: By applying spatiotemporal disaggregation to the traditional Count Sketch (CS), I created Disaggregatable Sketch (DiSketch), a disaggregated frequency estimator that can answer common network telemetry queries such as retrieving the number of packets or data volumes in any network flow. DiSketch significantly reduces estimation errors by almost an order of magnitude compared to traditional aggregated sketches, demonstrating the applied efficacy of spatiotemporal disaggregation.

These contributions advance the state-of-the-art in sketch-based telemetry by providing a scalable and efficient method for deploying estimators across an entire network, overcoming challenges related to resource constraints and traffic variability.

6.2.2 Optimizing Cost vs Accuracy in Sketches

Similarly, Chapter 4 presents several innovations to further improve on the cost vs accuracy tradeoff in sketches by focusing not on network-wide deployment, but on the processing within individual sketches. The primary contributions in this regard are:

Lazy Bloom filter updates: By rethinking the computational dependencies within standard BFs, I present a structure called a *lazy Bloom filter*. This data structure is functionally similar to standard BFs but offers a few key benefits. These include a reduced false positive rate and the capability to filter out keys (e.g., traffic flows) with up to a set number of occurrences.

FlowLiDAR: The chapter also presents *Flow Lightweight Detection and Ranging (FlowLiDAR)*, a solution capable of tracking almost all network flows with modest data plane memory, independent of the flowID size. It leverages the lazy BF, along with key innovations such as *differential flow detection*, decoupling flowID storage from the data plane, and using linear programming. FlowLiDAR drastically reduces the memory required for highly accurate frequency estimation compared with current alternatives.

6.2.3 Alleviating the Telemetry Collection Bottleneck

Finally, Chapter 5 argues for a fundamental redesign of traditional telemetry collection stacks to overcome critical bottlenecks, immensely increasing telemetry collection rates. The primary contributions in this regard are:

Direct Telemetry Access (DTA): I developed DTA, a high-performance telemetry collection system that leverages RDMA to increase data collection rates significantly. DTA achieves this by enabling direct memory access, bypassing the CPU for processing incoming telemetry reports, greatly enhancing the efficiency and scalability of telemetry data collection.

Translation Mechanism: A pivotal innovation within DTA is the *translator*. This component intercepts telemetry data from switches and converts them into RDMA calls, which isolates data structure management into a single device which reduces the deployment costs of switches and allows seamless RDMA extensions.

Novel RDMA Verbs: The translator allows support for high-speed data structure population in a way that distributed RDMA generation cannot. Leveraging this benefit, I introduce new CPU-bypassing memory verbs called *DTA primitives*: *Key-Write*, *Postcarding*, *Append*, and *Key-Increment*.

Broad Compatibility and Performance Enhancements: I demonstrate how these primitives support a wide range of existing monitoring systems and telemetry scenarios, including state-of-the-art systems like In-band Network Telemetry (INT) and Marple. Further, DTA is shown to significantly increase the collection capacities of these systems.

6.3 Implications of the Research

While designed within P4-programmable Protocol Independent Switch Architecture (PISA) constraints, the algorithms are not inherently dependent on this architecture or even on programmable networks. The computational limitations of this architecture, outlined in Section 2.4.1, are stringent but necessary to ensure consistent line rate throughput at immense speed. My prototypes show that these algorithms and solutions are compatible with *at least some* high-speed networking hardware and work at terabit per second speeds. With this effort, I aim to advance next-generation network telemetry, providing technology for detailed insights at reasonable costs.

Despite mentioning sample-free telemetry throughout this dissertation, achieving this is not my sole objective. Instead, by developing technology *capable of* sample-free telemetry, I simultaneously open up for much less restrictive sampling or selection methods than those commonly used. For

example, a spatiotemporally disaggregated FlowLiDAR would offer network-wide visibility into flows at a lower cost. Simultaneously, DTA can *significantly* increase the insight in existing telemetry systems at *much* lower infrastructure costs than before.

I envision accessible and sophisticated network monitoring at a level of granularity previously reserved for the most critical networks of organizations with immense budgets. Such insight greatly enhances real-time detection and mitigation of issues such as operational disruptions, link flooding attacks, and penetration attempts through precise fingerprinting.

Multiple labs have shown interest in leveraging my technology for real-time machine learning analysis of traffic patterns. While I cannot guarantee the computational feasibility of these real-time systems due to potential high inference costs, the improved level of insight inarguably enhances the accuracy of machine learning inferences if they prove computationally viable.

6.4 Limitations

This dissertation advances sketch-based monitoring but does not address the on-switch costs of non-sketch technologies. Sketches, while useful, do not capture all networking events, requiring supplementary technologies. For instance, identifying packet loss events and their causes can consume up to 30% of switches' computational capacities [230], indicating that diverse network telemetry remains costly. Approximately half of these costs go toward reducing telemetry reports [230]. With advancements from this dissertation, cheaper report-reduction techniques might suffice.

Additionally, DTA enhances server-side collection but does not consider network load from data transmission or costs of subsequent analyses. This burden is typical of fine-grained monitoring systems, including spatiotemporal disaggregation which increases telemetry volumes by several factors. Integrating analytical and pre-processing capabilities into network infrastructure could leverage hardware speed and reduce data transit distances, aligning with trends in query-based monitoring [77, 159, 223]. Despite these challenges,

sketches remain an efficient component for many counter-based queries. Additionally, query-based monitoring can still generate high telemetry loads [159], making this dissertation's contributions remain relevant.

6.5 Final Remarks

This dissertation significantly advances network telemetry, particularly in sketch-based monitoring and telemetry collection. The contributions enhance measurement accuracies and data extraction rates, providing deeper insights into network states and traffic patterns while maintaining computational efficiency and cost-effectiveness.

Consequently, these innovations pave the way for enhanced performance optimization and improved real-time network security monitoring, ultimately leading to more robust and reliable next-generation networks.

Bibliography

[1] Kanak Agarwal, Eric Rozner, Colin Dixon, and John Carter. Sdn traceroute: Tracing sdn forwarding without changing network behavior. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 145–150, 2014.

[2] Anurag Agrawal and Changhoon Kim. Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–32. IEEE Computer Society, 2020.

[3] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed b-tree. *Proc. VLDB Endow.*, 1(1), 2008.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

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

[6] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proc. of ACM SIGCOMM*, 2014.

[7] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29, 1996.

[8] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[9] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 38–44, 2020.

[10] Michael P Andersen and David E Culler. Btrdb: Optimizing storage system design for timeseries processing. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 39–52, 2016.

[11] Gianni Antichi and Gábor Rétvári. Full-Stack SDN: The Next Big Challenge? In *Symposium on SDN Research (SOSR)*. ACM, 2020.

[12] Arista. Telemetry and analytics. `https://www.arista.com/en/solutions/telemetry-analytics`, 2022. Accessed: 2022-02-02.

[13] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. Leveraging Service Meshes as a New Network Layer. In *Workshop on Hot Topics in Networks (HotNets)*. ACM, 2021.

[14] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A {High-Speed}{Load-Balancer} design with guaranteed {Per-Connection-Consistency}. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 667–683, 2020.

[15] Ran Ben Basat, Gil Einziger, Junzhi Gong, Jalil Moraney, and Danny Raz. q-max: A unified scheme for improving network measurement

throughput. In *Proceedings of the Internet Measurement Conference*, pages 322–336, 2019.

[16] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. Salsa: self-adjusting lean streaming analytics. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 864–875. IEEE, 2021.

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

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

[19] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of SIGCOMM IMC*, 2010.

[20] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Conference on Internet Measurement (IMC)*. ACM, 2010.

[21] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the seventh conference on emerging networking experiments and technologies*, pages 1–12, 2011.

[22] Giuseppe Bianchi, Elisa Boschi, Simone Teofili, and Brian Trammell. Measurement data reduction through variation rate metering. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[23] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[24] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[25] Broadcom. Broadcom Tomahawk 5. `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78900-series`.

[26] BROADCOM. Trident Programmable Switch. `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series`, 2017.

[27] F Brockners, S Bhandari, S Dara, C Pignataro, H Gredler, J Leddy, S Youell, D Mozes, T Mizrahi, P Lapukhov, et al. Requirements for in-situ oam. In *Working Draft, Internet-Draft draft-brockners-inband-oam-requirements-03*. 2017.

[28] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

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

[30] Caida. The CAIDA UCSD Anonymized Internet Traces, 2016. `http://www.caida.org/data/passive/passive_2016_dataset.xml`.

[31] CAIDA. Passive monitor: equinix-nyc, 2019.

[32] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Proc. of ICALP*, 2002.

[33] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. Purr: a primitive for reconfigurable fast reroute: hope for the best and program for the worst. In *Proceedings of the 15th international conference on emerging networking experiments and technologies*, pages 1–14, 2019.

[34] Cisco. Explore model-driven telemetry. `https://blogs.cisco.com/developer/model-driven-telemetry-sandbox`, 2019. Accessed: 2021-06-24.

[35] Cisco. Cisco nexus 34180yc and 3464c programmable switches data sheet, 2020.

[36] Cisco. How to scale IOS-XR Telemetry with InfluxDB . `https://community.cisco.com/t5/service-providers-knowledge-base/how-to-scale-ios-xr-telemetry-with-influxdb/ta-p/4442024`, 2021.

[37] Cisco. Trex. `https://trex-tgn.cisco.com/`, 2022. Accessed: 2022-01-25.

[38] Cisco. Cisco ios netflow. `https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html`, 2023. Accessed: 2023-02-08.

[39] Inc. Cisco Systems. Cisco nexus 9300-ex platform switches architecture, 2017.

[40] Inc. Cisco Systems. Cisco tetration, 2024.

[41] Peter Clifford and Ioana Cosma. A simple sketching algorithm for entropy estimation over streaming data. In *Artificial Intelligence and Statistics*, pages 196–206. PMLR, 2013.

[42] Edith Cohen, Rasmus Pagh, and David Woodruff. Wor and p's: Sketches for l_p-sampling without replacement. *Advances in Neural Information Processing Systems*, 33:21092–21104, 2020.

[43] Michael B Cohen and Richard Peng. Lp row sampling by lewis weights. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 183–192, 2015.

[44] The P4 Language Consortium. P416 language specification, 2017.

[45] Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *J. Algorithms*, 2005.

[46] 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.

[47] Alessandro Cornacchia, German Sviridov, Paolo Giaccone, and Andrea Bianco. A traffic-aware perspective on network disaggregated sketches. In *2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet)*, pages 1–4. IEEE, 2021.

[48] Penglai Cui, Heng Pan, Zhenyu Li, Jiaoren Wu, Shengzhuo Zhang, Xingwu Yang, Hongtao Guan, and Gaogang Xie. Netfc: Enabling accurate floating-point arithmetic on programmable switches. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

[49] Timothy A Davis, Sivasankaran Rajamanickam, and Wissam M Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016.

[50] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's

highly available key-value store. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, 2007.

[51] Christina Delimitrou, Sriram Sankar, Aman Kansal, and Christos Kozyrakis. Echo: Recreating network traffic maps for datacenters with tens of thousands of servers. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 14–24. IEEE, 2012.

[52] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *Proc. 37th ICALP (1)*, pages 213–225, 2010.

[53] Damu Ding, Marco Savi, Federico Pederzolli, Mauro Campanella, and Domenico Siracusa. In-network volumetric ddos victim identification using programmable commodity switches. *IEEE Transactions on Network and Service Management*, 18(2):1191–1202, 2021.

[54] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[55] Olivier Dubois and Jacques Mandler. The 3-XORSAT threshold. In *Proc. 43rd FOCS*, pages 769–778, 2002.

[56] Nick G Duffield and Matthias Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM transactions on networking*, 9(3):280–292, 2001.

[57] Elasticsearch. Elk stack: Elasticsearch, kibana, beats, and logstash, 2024.

[58] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on*

Applications, technologies, architectures, and protocols for computer communications, pages 323–336, 2002.

[59] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[60] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 339–350, 2010.

[61] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 635–649, 2022.

[62] Marcial P Fernandez. Comparing openflow controller paradigms scalability: Reactive and proactive. In *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pages 1009–1016. IEEE, 2013.

[63] Giuseppe Fioccola, Alessandro Capello, Mauro Cociglio, Luca Castaldelli, Mach Chen, Lianshu Zheng, Greg Mirsky, and Tal Mizrahi. Alternate-marking method for passive and hybrid performance monitoring. Technical report, 2018.

[64] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[65] Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. 2019.

[66] The Linux Foundation. Prometheus - monitoring system and time-series database, 2024.

[67] Nikolaos Fountoulakis and Konstantinos Panagiotou. Sharp load thresholds for cuckoo hashing. *Random Struct. Algorithms*, 41(3):306–333, 2012.

[68] Sumit Ganguly. Counting distinct items over update streams. *Theoretical Computer Science*, 378(3):211–222, 2007.

[69] Sam Gao, Mark Handley, and Stefano Vissicchio. Stats 101 in p4: Towards in-switch anomaly detection. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 84–90, 2021.

[70] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In *Proc. 15th SEA*, pages 339–352, 2016.

[71] Soudeh Ghorbani, Zibin Yang, P Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 225–238, 2017.

[72] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 792–799. IEEE, 2011.

[73] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure flow control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779–805, Renton, WA, 2022. USENIX Association.

[74] The P4.org Applications Working Group. Telemetry report format specification. `https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report_latest.pdf`, 2020. Accessed: 2021-06-23.

[75] Liyuan Gu, Ye Tian, Wei Chen, Zhongxiang Wei, Cenman Wang, and Xinming Zhang. Per-flow network measurement with distributed sketch. *IEEE/ACM Transactions on Networking*, 2023.

[76] 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 *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[77] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proc. of ACM SIGCOMM*, 2018.

[78] Hui Han, Zheng Yan, Xuyang Jing, and Witold Pedrycz. *Information Fusion*, 82:58–85, 2022.

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

[80] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.

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

[82] Nicholas JA Harvey, Jelani Nelson, and Krzysztof Onak. Sketching and streaming entropy via approximation theory. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 489–498. IEEE, 2008.

[83] George Havas, Bohdan S. Majewski, Nicholas C. Wormald, and Zbigniew J. Czech. Graphs, hypergraphs and hashing. In *Proc. 19th WG*, pages 153–165, 1993.

[84] Yongchao He, Wenfei Wu, Xuemin Wen, Haifeng Li, and Yongqiang Yang. Scalable on-switch rate limiters for the cloud. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[85] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *NSDI*, volume 10, pages 249–264, 2010.

[86] Hinne Hettema. *Agile Security Operations: Engineering for agility in cyber defense, detection, and response*. Packt Publishing Ltd, 2022.

[87] J. Hill, M. Aloserij, and P. Grosso. Tracking network flows with p4. In *IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*, 2018.

[88] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 193–207, 2022.

[89] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proc. of ACM SIGCOMM*, 2017.

[90] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximatemeasurement with automated statistical inference. In *Proc. of ACM SIGCOMM*, 2018.

[91] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. Toward nearly-zero-error sketching via compressive sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 1027–1044, 2021.

[92] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 404–421, 2020.

[93] Huawei. Overview of telemetry. `https://support.huawei.com/enterprise/en/doc/EDOC1000173015/165fa2c8/overview-of-telemetry`, 2020. Accessed: 2021-06-24.

[94] Huawei. Telemetry. `https://support.huawei.com/enterprise/en/doc/EDOC1100196389`, 2021.

[95] Ltd. Huawei Technologies Co. Tig configuration and usage, 2024.

[96] Jonghwan Hyun, Nguyen Van Tu, and James Won-Ki Hong. Towards knowledge-defined networking using in-band network telemetry. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE, 2018.

[97] IEEE 802.11Qbb. Priority Based Flow Control. 2011.

[98] InfluxData Inc. Influxdb overview, 2024.

[99] InfluxData Inc. Telegraf — influxdata, 2024.

[100] Piotr Indyk and Milan Ruzic. Near-optimal sparse recovery in the l1 norm. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 199–207. IEEE, 2008.

[101] Infiniband Trade Association. Infinibandtm architecture specification, 2015. Volume 1 Release 1.3.

[102] Intel. Intel® tofino™ series programmable ethernet switch asic. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html`, 2016. Accessed: 2022-01-25.

[103] Intel. In-band network telemetry detects network performance issues. `https://builders.intel.com/docs/networkbuilders/in-band-network-telemetry-detects-network-performance-issues.pdf`, 2020. Accessed: 2021-06-04.

[104] Intel. Intel® ethernet network adapter e810-cqda1/cqda2. `https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/network-adapters/ethernet-800-series-network-adapters/e810-cqda1-cqda2-100gbe-brief.html`, 2020. Accessed: 2021-06-11.

[105] Intel. Intel deep insight network analytics software. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/network-analytics/deep-insight.html`, 2021. Accessed: 2021-06-10.

[106] Intel. Performance Tuning for Mellanox Adapters. `https://support.mellanox.com/s/article/performance-tuning-for-mellanox-adapters`, 2022.

[107] Vineeta Jain, Divya Rishi Sahu, and Deepak Singh Tomar. Session hijacking: threat analysis and countermeasures. In *Int. Conf. on Futur-*

istic Trends in Computational Analysis and Knowledge Management, 2015.

[108] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2017.

[109] Matthews Jose, Kahina Lazri, Jérôme François, and Olivier Festor. Inrec: In-network real number computation. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 358–366. IEEE, 2021.

[110] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Boryło. Flow length and size distributions in campus internet traffic. *Computer Communications*, 167:15–30, 2021.

[111] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance {RDMA} systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 437–450, 2016.

[112] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-aware load balancing at the virtual edge. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 323–335, 2017.

[113] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[114] Elie F Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE access*, 9:87094–87155, 2021.

[115] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 421–436, 2019.

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

[117] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 90–106, 2020.

[118] Nikola Kostic. What is a data lake & how to build one?, 2023.

[119] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, and Jia Wang. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. In *Special Interest Group for the Computer Performance Evaluation (SIGMETRICS)*. ACM, 2004.

[120] Jan Kučera, Diana Andreea Popescu, Han Wang, Andrew Moore, Jan Kořenek, and Gianni Antichi. Enabling event-triggered data plane monitoring. In *Proceedings of the Symposium on SDN Research*, page 14–26. Association for Computing Machinery, 2020.

[121] Grafana Labs. Grafana: The open observability platform, 2024.

[122] Jonatan Langlet. Direct Telemetry Access source code. `https://github.com/jonlanglet/DTA`, 2023.

[123] Jonatan Langlet, Ran Ben Basat, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. Direct telemetry access. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 832–849, 2023.

[124] Jonatan Langlet, Ran Ben-Basat, Sivaramakrishnan Ramanathan, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. Zero-cpu collection with direct telemetry access. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 108–115, 2021.

[125] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[126] Kasper Green Larsen, Rasmus Pagh, and Jakub Tětek. Countsketches, feature hashing and the median of three. In *International Conference on Machine Learning*, pages 6011–6020. PMLR, 2021.

[127] Gene Moo Lee, Huiya Liu, Young Yoon, and Yin Zhang. Improving sketch reconstruction accuracy using linear least squares method. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 24–24, 2005.

[128] Alberto Lerner, Rana Hussein, Philippe Cudré-Mauroux, and U eXascale Infolab. The case for network accelerated query processing. In *CIDR*, 2019.

[129] Fuliang Li, Kejun Guo, Jiaxing Shen, and Xingwei Wang. Effective network-wide traffic measurement: A lightweight distributed sketch deployment. 2024.

[130] Minghao Li, Ran Ben Basat, Shay Vargaftik, ChonLam Lao, Kevin Xu, Xinran Tang, Michael Mitzenmacher, and Minlan Yu. THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression. In *USENIX Symposium on Networked Systems Design and Implementation*, 2024.

[131] Yiran Li, Kevin Gao, Xin Jin, and Wei Xu. Concerto: cooperative network-wide telemetry with controllable error rate. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 114–121, 2020.

[132] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. {DETER}: Deterministic {TCP} replay for performance diagnosis. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 437–452, 2019.

[133] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 311–324, 2016.

[134] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. 2019.

[135] Richard J Lipton. A new approach to information theory. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 699–708. Springer, 1994.

[136] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. {DistCache}: Provable load balancing for {Large-Scale} storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

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

[138] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[139] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proc. of ACM SIGCOMM*, 2016.

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

[141] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. Optimizing Bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys and Tutorials*, 21(2):1912–1949, 2019.

[142] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[143] Uri Mahlab, Peter E Omiyi, Harel Hundert, Yotam Wolbrum, Or Elimelech, Itamar Aharon, Katya Shishchenko Ziv Erlich, and Segev Zarakovsky. Entropy-based load-balancing for software-defined elastic optical networks. In *2017 19th International Conference on Transparent Optical Networks (ICTON)*, pages 1–4. IEEE, 2017.

[144] Wassim Mansour, Nicolas Janvier, and Pablo Fajardo. Fpga implementation of rdma-based data acquisition system over 100-gb ethernet. *IEEE Transactions on Nuclear Science*, 66(7):1138–1143, 2019.

[145] Jonatas Marques, Kirill Levchenko, and Luciano Gaspary. Intsight: Diagnosing slo violations with in-band network telemetry. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 421–434, 2020.

[146] Kayol S Mayer, Jonathan A Soares, Rossano P Pinto, Christian E Rothenberg, Dalton S Arantes, and Darli AA Mello. Machine-learning-based soft-failure localization with partial software-defined networking telemetry. *Journal of Optical Communications and Networking*, 13(10):E122–E131, 2021.

[147] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

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

[149] Microsoft. Cloud Service Fundamentals: Telemetry - Reporting. `https://azure.microsoft.com/fr-fr/blog/cloud-service-fundamentals-telemetry-reporting//`, 2013.

[150] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[151] Tal Mizrahi, Vitaly Vovnoboy, Moti Nisim, Gidi Navon, and Amos Soffer. Network telemetry solutions for data center and enterprise networks. *Marvell, White Paper*, 2018.

[152] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. Fast in-network gray failure detection for isps. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 677–692, 2022.

[153] Michael Molloy. Cores in random hypergraphs and boolean formulas. *Random Struct. Algorithms*, 27(1):124–135, 2005.

[154] Andrea Monterubbiano, Jonatan Langlet, Stefan Walzer, Gianni Antichi, Pedro Reviriego, and Salvatore Pontarelli. Lightweight acquisition and ranging of flows in the data plane. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(3):1–24, 2023.

[155] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 129–143, 2016.

[156] Seyed Mohammad Mousavi. *Early detection of DDoS attacks in software defined networks controller*. PhD thesis, Carleton University, 2014.

[157] Sukhyun Nam, Jiyoon Lim, Jae-Hyoung Yoo, and James Won-Ki Hong. Network anomaly detection based on in-band network telemetry with rnn. In *2020 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pages 1–4. IEEE, 2020.

[158] Srinivas Narayana, Jennifer Rexford, and David Walker. Compiling path queries in software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 181–186, 2014.

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

[160] Grotto Networking. Switch architectures, 2021.

[161] APS Networks. Advanced programmable switch. `https://www.aps-networks.com/wp-content/uploads/2021/07/210712_APS_BF2556X-1T_V04.pdf`, 2019. Accessed: 2022-01-25.

[162] Barefoot Networks. Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/.

[163] Juniper Networks. Overview of the junos telemetry interface. https://www.juniper.net/documentation/us/en/software/junos/interfaces-telemetry/topics/concept/junos-telemetry-interface-oveview.html, 2021. Accessed: 2021-06-24.

[164] Juniper Networks. Qfx5130 switch datasheet, 2024.

[165] NVIDIA. Nvidia bluefield-2 dpu. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2021. Accessed: 2022-01-25.

[166] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[167] Tian Pan, Enge Song, Zizheng Bian, Xingchen Lin, Xiaoyu Peng, Jiao Zhang, Tao Huang, Bin Liu, and Yunjie Liu. Int-path: Towards optimal path planning for in-band network-wide telemetry. In *IEEE INFOCOM 2019-IEEE Conference On Computer Communications*, pages 487–495. IEEE, 2019.

[168] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 194–206, 2021.

[169] F. Pereira, N. Neves, and F. M. V. Ramos. Secure network monitoring using programmable data planes. In *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2017.

[170] Boris Pittel and Gregory B. Sorkin. The satisfiability threshold for k-XORSAT. *Combinatorics, Probability & Computing*, 25(2):236–268, 2016.

[171] Stanislav Ponomarev and Travis Atkison. Industrial control system network intrusion detection by telemetry analysis. *IEEE Transactions on Dependable and Secure Computing*, 13(2):252–260, 2015.

[172] Salvatore Pontarelli, Pedro Reviriego, and Michael Mitzenmacher. Improving the performance of invertible bloom lookup tables. *Inf. Process. Lett.*, 114(4):185–191, apr 2014.

[173] Jürgen Quittek, Tanja Zseby, Benoit Claise, and Sebastian Zander. Requirements for ip flow information export (ipfix). Technical report, 2004.

[174] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. *ACM SIGCOMM Computer Communication Review*, 44(4):407–418, 2014.

[175] Jennifer Rexford. Closing the network control loop, 2020.

[176] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[177] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[178] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. A high-speed stateful packet processing approach for tbps programmable switches. In *Proceedings*

of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), 2023.

[179] Brandon Schlinker, Italo Cunha, Yi-Ching Chiu, Srikanth Sundaresan, and Ethan Katz-Bassett. Internet performance from facebook's edge. In *Proceedings of the Internet Measurement Conference*, pages 179–194, 2019.

[180] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A. Dinda, Ming-Yang Kao, and Gokhan Memik. Reversible Sketches: Enabling Monitoring and Analysis over High-Speed Data Streams. In *Transactions on Networking, Volume: 15, Issue: 5*. IEEE Press, 2007.

[181] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. Continuous in-network round-trip time monitoring. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 473–485, New York, NY, USA, 2022. Association for Computing Machinery.

[182] John Shalf. The future of computing beyond moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, 2020.

[183] Siyuan Sheng, Qun Huang, Sa Wang, and Yungang Bao. Pr-sketch: monitoring per-key aggregation of streaming data with nearly full accuracy. *Proceedings of the VLDB Endowment*, 14(10):1783–1796, 2021.

[184] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[185] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's

datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, page 183–197. Association for Computing Machinery, 2015.

[186] Suneet Kumar Singh, Christian Esteve Rothenberg, Jonatan Langlet, Andreas Kassler, Péter Vörös, Sándor Laki, and Gergely Pongrácz. Hybrid p4 programmable pipelines for 5g gnodeb and user plane functions. *IEEE Transactions on Mobile Computing*, 2022.

[187] Arunan Sivanathan, Hassan Habibi Gharakheili, and Vijay Sivaraman. Managing iot cyber-security using programmable telemetry and machine learning. *IEEE Transactions on Network and Service Management*, 17(1):60–74, 2020.

[188] LLC. SolarWinds Worldwide. Network performance monitor, 2024.

[189] Robin Sommer and Anja Feldmann. Netflow: Information loss or win? In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 173–174, 2002.

[190] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.

[191] Enge Song, Tian Pan, Chenhao Jia, Wendi Cao, Jiao Zhang, Tao Huang, and Yunjie Liu. Int-filter: Mitigating data collection overhead for high-resolution in-band network telemetry. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[192] Pensando Systems. Pensando dsc-100 distributed services card. `https://pensando.io/wp-content/uploads/2020/03/DSC-100-ProductBrief-v06.pdf`, 2021. Accessed: 2022-01-23.

[193] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX}*

Symposium on Networked Systems Design and Implementation ({NSDI} 18), pages 453–456, 2018.

[194] Lizhuang Tan, Wei Su, Wei Zhang, Jianhui Lv, Zhenyi Zhang, Jingying Miao, Xiaoxi Liu, and Na Li. In-band network telemetry: A survey. *Computer Networks*, 186:107763, 2021.

[195] Lizhuang Tan, Wei Su, Wei Zhang, Huiling Shi, Jingying Miao, and Pilar Manzanares-Lopez. A packet loss monitoring system for in-band network telemetry: Detection, localization, diagnosis and recovery. *IEEE Transactions on Network and Service Management*, 18(4):4151–4168, 2021.

[196] Lu Tang, Qun Huang, and Patrick PC Lee. A fast and compact invertible sketch for network-wide heavy flow detection. *IEEE/ACM Transactions on Networking*, 28(5):2350–2363, 2020.

[197] Lu Tang, Yao Xiao, Qun Huang, and Patrick PC Lee. A high-performance invertible sketch for network-wide superspreader detection. *IEEE/ACM Transactions on Networking*, 31(2):724–737, 2022.

[198] Dell Technologies. Dell emc powerswitch s4048t-on switch, 2020.

[199] Mellanox Technologies. Connectx®-6 vpi card. `https://www.mellanox.com/files/doc-2020/pb-connectx-6-vpi-card.pdf`, 2020. Accessed: 2021-05-12.

[200] Mellanox Technologies. Nvidia spectrum-4. `https://nvdam.widen.net/s/pjlcwnrdbn/ethernet-switches-spectrum-4-asic-datasheet-us`, 2023. Accessed: 2024-04-04.

[201] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. Packetscope: Monitoring the packet lifecycle inside a switch. In *Proceedings of the Symposium on SDN Research*, pages 76–82, 2020.

[202] Cisco Tim Stevenson. Nexus 9000 architecture, 2020.

[203] Daniel Ting. Count-min: Optimal estimation and tight error bounds using empirical error distributions. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2319–2328, 2018.

[204] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2407–2422, 2020.

[205] Pang-Wei Tsai, Chun-Wei Tsai, Chia-Wei Hsu, and Chu-Sing Yang. Network monitoring in software-defined networking: A review. *IEEE Systems Journal*, 12(4):3958–3969, 2018.

[206] Nguyen Van Tu, Jonghwan Hyun, and James Won-Ki Hong. Towards onos-based sdn monitoring using in-band network telemetry. In *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 76–81. IEEE, 2017.

[207] Nguyen Van Tu, Jonghwan Hyun, Ga Yeon Kim, Jae-Hyoung Yoo, and James Won-Ki Hong. Intcollector: A high-performance collector for in-band network telemetry. In *2018 14th International Conference on Network and Service Management (CNSM)*, pages 10–18. IEEE, 2018.

[208] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 407–420, 2017.

[209] Jonathan Vestin, Andreas Kassler, Deval Bhamare, Karl-Johan Grinnemo, Jan-Olof Andersson, and Gergely Pongracz. Programmable event detection for in-band network telemetry. In *2019 IEEE 8th international conference on cloud networking (CloudNet)*, pages 1–6. IEEE, 2019.

[210] Liang-Min Wang, Tim Miskell, John Morgan, and Edwin Verplanke. Design of a real-time traffic mirroring system. In *IM*, pages 793–796, 2021.

[211] Mea Wang, Baochun Li, and Zongpeng Li. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 628–635. IEEE, 2004.

[212] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, New York, NY, USA, 2022. Association for Computing Machinery.

[213] Xiong Wang, Hanyu Liu, Jun Zhang, Jing Ren, Sheng Wang, and Shizhong Xu. Flowmap: A fine-grained flow measurement approach for data-center networks. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2019.

[214] Theophilus Wellem, Yu-Kuen Lai, Chao-Yuan Huang, and Wen-Yaw Chung. A flexible sketch-based network traffic monitoring infrastructure. *IEEE Access*, 7:92476–92498, 2019.

[215] Xilinx. Xilinx embedded rdma enabled nic. `https://www.xilinx.com/support/documentation/ip_documentation/ernic/v3_0/pg332-ernic.pdf`, 2021. Accessed: 2021-06-11.

[216] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 651–665, 2022.

[217] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. Bedrock: Programmable network support for secure {RDMA} systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2585–2600, 2022.

[218] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Hongyi Liu, Matty Kadosh, Alan Lo, Aditya Akella, Thomas Anderson, Arvind Krishnamurthy, TS Eugene Ng, et al. A vision for runtime programmable networks. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 91–98, 2021.

[219] Jiarong Xing, Wenqing Wu, and Ang Chen. Ripple: A programmable, decentralized {Link-Flooding} defense against adaptive adversaries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3865–3881, 2021.

[220] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. Profiling internet backbone traffic: behavior models and applications. *ACM SIGCOMM Computer Communication Review*, 35(4):169–180, 2005.

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

[222] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proc. of ACM SIGCOMM*, 2018.

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

[224] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[225] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings*

of the 2017 Internet Measurement Conference, page 78–85. Association for Computing Machinery, 2017.

[226] Qi Zhao, Abhishek Kumar, and Jun Xu. Joint Data Streaming and Sampling Techniques for Detection of Super Sources and Destinations. In *Conference on Internet Measurement (IMC)*. USENIX Association, 2005.

[227] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *NSDI*, pages 991–1010, 2021.

[228] Zongyi Zhao, Xingang Shi, Zhiliang Wang, Qing Li, Han Zhang, and Xia Yin. Efficient and accurate flow record collection with hashflow. *IEEE Transactions on Parallel and Distributed Systems*, 33(5):1069–1083, 2021.

[229] Yu Zhou, Jun Bi, Tong Yang, Kai Gao, Jiamin Cao, Dai Zhang, Yangyang Wang, and Cheng Zhang. Hypersight: Towards scalable, high-coverage, and dynamic network monitoring queries. *IEEE Journal on Selected Areas in Communications*, 38(6):1147–1160, 2020.

[230] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.

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