DynATOS+: A Network Telemetry System for Dynamic Traffic and Query Workloads

Chris Misa, Ramakrishnan Durairajan, and Reza Rejaie, *University of Oregon;* Walter Willinger, *NIKSUN, Inc.*

***Abstract*—Network telemetry systems provide critical visibility into the state of network traffic. By leveraging modern programmable switch hardware, significant progress has been made to scale these systems to production network traffic workloads. Less attention has been paid towards efficiently utilizing these hardware targets' limited resources in the face of dynamics such as the composition of the traffic workload as well as the number and types of queries running at any given point in time. However, both of these dynamics have implications on resource requirements and query accuracy.**

Building on our prior work DynATOS, which argues that this dynamics problem motivates reframing telemetry systems as *resource schedulers*, we present in this paper the design, implementation, and evaluation of DynATOS+. DynATOS+ relies on the same efficient time-division approximation and scheduling algorithm that DynATOS uses and that allows for user-defined query accuracy and latency specifications that are intended to result in tradeoffs with respect to query execution to reduce hardware resource usage. However, unlike DynATOS, DynATOS+ significantly reduces the burden on end users to express their queries by allowing them to use simple-to-state accuracy goals. For example, the method for specifying per-query accuracy goals in DynATOS+ no longer requires end users to either know the average range of query results in advance or to submit multiple trial queries to tune their accuracy goal specifications. We perform extensive simulation-based evaluations that (i) show that this new functionality of DynATOS+ works in practice, (ii) illustrate in detail the tradeoffs that result with respect to query execution and hardware resource usage for a wide range of systems parameters, and (iii) allow for an assessment of system performance under changing query workloads on top of changes in the composition of traffic workloads that has eluded previous work in this area.

***Index Terms*—Network Traffic Monitoring, Programmable Switch Hardware, Resource Scheduling.**

I. Introduction

NETWORK telemetry systems provide users (e.g., network administrators, researchers) with critical insights into the state of the network by collecting information about individual packets and processing this information into high-level features in near real-time. Typically, these features are the results of user-defined queries, where a query is expressed as a sequence of high-level operations such as filter and reduce [1]–[3]. Generated query results drive management decisions such as deploying defensive measures in the face of an attack or updating routing to avoid congestion. A key functionality of telemetry systems is to determine how best to leverage available resources (*e.g.*, network hardware resources, such as switch ASICs or NICs; software-programmable resources, such as general-purpose CPUs) to execute a given set of queries. Due to massive traffic volumes and often stringent timing requirements, state-of-the-art telemetry systems typically make use of programmable network hardware (*e.g.*, programmable switch ASICs [4], [5]) and also apply approximation techniques (*e.g.*, sketches [6]–[8]).

In executing user-defined queries, telemetry systems must cope with two independent and challenging sources of dynamics. First, the resources required to execute any given query depend on the underlying composition of the traffic workload (*e.g.*, the number of flows that satisfy the given query, where flows are identified by a flow key such as the commonly-used IP 5-tuple). For example, a DDoS-detection query that counts the number of sources contacting each destination might require a counter for each active destination on the network, but the number of active destinations may vary over time [7]. The accuracy guarantees of state-of-the-art approximation techniques like sketches [6] likewise depend on the traffic composition and are expressed, for example in the case of count-min sketch, in terms of the L1 norm. Consequently, if the traffic changes, accuracy can no longer be guaranteed. Second, the number and type of concurrent queries submitted by a user can vary over the system's deployment. For example, an operator might need to submit follow-up queries to pinpoint the root cause of increased congestion. Both of these sources of dynamics affect data plane resource usage implying that telemetry systems must dynamically adjust resource allocations.

Several recent efforts [3], [7] have made progress towards coping with both of these sources of dynamics individually and in isolation, but do not address challenges arising from their simultaneous presence in network telemetry systems. For example, ElasticSketch [7] presents a method for dynamically coping with changes in only certain aspects of traffic (*e.g.*, packet rate, available bandwidth, flow size distribution). However, this effort relies on a fixed flow key which forces users to reload the switch pipeline to run queries that require a different flow key. On the other hand, Newton [3] and FlyMon [9] describe techniques to update query operations during runtime which enables users to dynamically add and remove queries as their monitoring needs change. However, these efforts do not consider the problem of adjusting resource allocations between concurrent queries as traffic composition changes. To the best of our knowledge, no other recent line of work addresses these simultaneous sources of dynamics in an efficient switch hardware-based system.

In this work, we argue that, in order to simultaneously

Manuscript received November 27, 2022; revised October 26, 2023; accepted January 26, 2024.

address these sources of dynamics, *telemetry systems should be reframed as active resource schedulers for query operations*. In particular, telemetry systems must manage finite switch hardware processing resources while adapting to varying numbers and types of queries as well as varying traffic composition. To support this argument, our prior work [10] made the following key contributions.

Time-division approximation method. Viewing telemetry systems as online schedulers enables a new approximation technique based on time-division approximation. At a high-level, this technique observes that query operations do not need to run all the time. Instead, operations can execute during strategically placed sub-windows of the overall time window (*e.g.*, an operation could execute for 3 of 8 equal-duration sub-windows of a 5 s overall time window). This technique is grounded in cluster sampling theory which allows us to estimate error and future resource requirements.

Adaptive scheduling algorithm. We provide a closed loop adaptive scheduling algorithm which leverages time-division approximation to execute operations from many user-defined queries on a single switch ASIC. Our scheduling algorithm leverages multi-objective optimization to balance between multiple high-level goals such as prioritizing accuracy, latency, or reduced volume of reported data across queries.

Evaluation in a functional hardware prototype. To evaluate our proposed techniques, we implement *DynATOS*, a telemetry operation scheduling system which leverages programmable switch hardware to answer dynamically submitted queries. Our implementation of *DynATOS* assumes a single runtime programmable switch hardware capable of executing a restricted number of primitive operations as supported by a telemetry module found in a widely available off-the-shelf switch ASIC. We evaluate *DynATOS* on our hardware prototype and through simulation showing that *(i)* time-division approximation is more robust than sketches to changes in traffic composition while offering a similar accuracy, overhead tradeoff space, *(ii)* our adaptive scheduler is able to meet query accuracy and latency goals in the presence of traffic and query dynamics, and *(iii)* the overheads in our scheduling loop are minimal and dominated by the time required to report and process intermediate results from the switch.

In this paper, we extend our prior work along several directions to develop *DynATOS+*.[1] For one, we improve our original approach to give network administrators better control over expressing query accuracy requirements and show how *DynATOS+* offers this new functionality "under the hood". This improvement simplifies network administrators' use of our system in practice by ensuring that formulating and submitting queries no longer requires any guesswork (e.g., advance knowledge of the average range of query results) or time-consuming tuning efforts (e.g., running multiple trial queries). Moreover, we complement our prior work with a new evaluation study that *(i)* includes a comprehensive sensitivity analysis of *DynATOS+'s* performance with respect to key systems parameters, and *(ii)* demonstrates *DynATOS+'s* unique ability to simultaneously deal with traffic and query dynamics.

[1] *DynATOS+* stands for Dynamic Approximate Telemetry Operation Scheduler. The "+" represents the delta of our current system compared to *DynATOS* [10] as described in § III-B.

II. Background & Motivation

A. Dynamic Telemetry Use Cases

Example II.1. Consider a scenario where a telemetry system is executing the DDoS and port scanning detection tasks described in Sonata [2][2]. The first stage of these tasks finds a set of distinct flow keys in each time window or epoch (*e.g.*, IPv4 source, destination pairs every 5 s for DDoS). Note that both tasks can be sufficiently satisfied with approximate results: it does not matter exactly how many DDoS sources are detected so long as a sufficiently many are detected to exceed the query's threshold. Given this, suppose both queries are associated with accuracy goals which quantify each query's tolerance for error (*e.g.*, underestimation of the number of DDoS sources).

Suppose traffic follows a stable pattern for several epochs with only small changes in the number of distinct elements considered by both tasks. During this time, the telemetry system is able to allocate adequate resources for these two queries to achieve good accuracy. Now, suppose at some later epoch traffic composition changes so that a much larger number of sources are seen (either due to a natural event like a flash crowd or due to an actual DDoS attack). This larger number of sources increases the number of pairs that both queries must keep track of. Either more resources will need to be allocated or accuracy will suffer.

While this example only considers a pair of queries, in realistic settings administrators likely need to monitor for a wide variety of attacks simultaneously (*e.g.*, the 11 queries described in Sonata [2]). Moreover, features like number of sources or destinations commonly overlap in these types of attack detection queries so that an anomalous change in one feature may upset the resource requirements of a large number of simultaneous queries.

Example II.2. Consider a scenario where a network administrator wants to understand the root cause of TCP latency on their network. In this scenario, the administrator would like to first run queries to detect when latency increases and for which hosts or subnets [11]. Again note that absolute accuracy is not as important as quickly identifying when overall latency has increased beyond some threshold, hence this query can be submitted with lower accuracy targets. Once detected, the administrator must submit a large number of queries to test possible causes of high latency such as re-transmissions or deep queues [1] with filter operations so that these queries only apply to the subnets experiencing latency. Also, follow-up queries that require precise packet-level semantics (*e.g.*, exact counts of TCP packets) can be submitted with higher accuracy goals. Note that this root-cause analysis phase may require several rounds of querying with tens of simultaneous queries in each round before the root cause of the latency can be determined.

[2] The DDoS task finds destinations receiving from large numbers of distinct sources and the port scanning task finds sources sending to a large number of distinct destination ports.

While the above examples focus on two particular tasks, the underlying concepts—of dealing with large shifts in query resource requirements caused by changes in traffic and of executing multiple queries over time in a dependent manner while respecting diverse accuracy goals—are commonly encountered in network operations.

B. Telemetry System Requirements

To deal with changes in traffic as well as changing sets of queries (as discussed in the examples above), telemetry systems must dynamically update allocation of limited hardware resources to query operations at runtime. In particular, the telemetry system itself must monitor traffic composition (*e.g.*, by inspecting the results of queries as it returns them to network administrators), and react to observed changes based on their impact on the resource requirements of currently executing queries. The ability to deal with these dynamics along with commonly assumed capabilities considered in prior work (*e.g.*, [2]) lead to the following requirements telemetry systems must meet.

R1: Query diversity. Marple [1] and Sonata [2] outline how a small set of parameterized stream processing operators can enable a wide range of telemetry queries. Telemetry systems must support these kinds of generic query description interfaces, allowing filtering over packet header values, grouping by arbitrary header fields, chaining operations, and joining the results of multiple operation chains.

R2: Approximate execution. Executing telemetry queries over the massive volumes of data flowing through networks poses heavy resource requirements. Furthermore, many telemetry queries are equally effective when computed approximately [12]. Therefore, telemetry systems should expose approximation techniques that allow trading off reduced result accuracy for lower resource requirements.

R3: Traffic dynamics. Composition of traffic (*e.g.*, the number of distinct keys observed each epoch) changes over time, and changes may be slow, regular, and easy to predict (*e.g.*, daily cycles) or fast and hard to predict (*e.g.*, flash crowds). As discussed in Example II.1, these changes in traffic composition lead to changes in the resource requirements for different groups of queries. Telemetry systems should robustly handle these changes without compromising query accuracy or latency [7].

R4: Query dynamics. The queries a network administrator needs to run change over time. Some of these changes may be infrequent (*e.g.*, adding new queries to monitor a newly deployed service), while some of these changes may be rapid and time-sensitive (*e.g.*, adding new queries to debug a performance anomaly or to pinpoint and block a network attack). Telemetry systems should be able to handle these dynamic query arrivals and removals, realizing updates within a few milliseconds and without incurring network downtime [3].

R5: Switch hardware acceleration. Due to massive traffic volumes, stringent timing requirements, and the limited speed of a single CPU core, executing telemetry queries on CPU-based systems is prohibitively expensive [2], [13]. As a result, telemetry systems must leverage resource-constrained hardware targets [4], [5], [14] for high-speed per-packet processing.

Approach	R1	R2	R3	R4	R5
Static switch-based	✓				✓
Runtime-programmable	✓	✓		✓	✓
Dynamic allocation		✓	✓	✓	✓
Sketch-based	✓	✓			✓
Software-based	✓	✓	✓	✓	
DynATOS	partial		partial		✓
DynATOS+	✓	✓	✓	✓	✓

TABLE I: Summary of how different approaches relate to the requirements of § II-B.

After one or more stages of filtering and aggregation in switch hardware, at the end of each epoch a relatively small batch of intermediate results can be forwarded to CPU-based systems for any remaining query operations [2].

C. State-of-the-art and their Limitations

State-of-the-art approaches each satisfy a subset of the requirements set forth above, but face limitations which hinder their ability to satisfy all requirements simultaneously.

Static switch-based approaches. Marple [1] and Sonata [2] compile traffic queries into static hardware description languages like P4 [15], demonstrating the efficiency of switch hardware in computing query results. However, these approaches fail to satisfy R4 since changing queries incurs seconds of network downtime (see [3]).

Runtime-programmable approaches. Recently, BeauCoup [16], Newton [3], FlyMon [9], and other "runtime-programmable" efforts [17], [18] demonstrate techniques to allow network administrators to add and remove queries at runtime without incurring downtime. These efforts lay a technical foundation to address R4, but do not address the challenge of R3. Our previous work *DynATOS* [10] introduced methods to simultaneously address R1-5, but *(i)* lacked a sufficiently general control over query accuracy which left network administrators with significant tunning challenges to simultaneously address R1 and R2 and *(ii)* stopped short of demonstrating our method's ability to simultaneously address R3 and R4.

Dynamic allocation approaches. DREAM [12] and SCREAM [19] develop dynamic allocation systems for telemetry operations addressing both R3 and R4. However, these approaches do not satisfy R1 because they require query-specific accuracy estimators.

Sketch-based approaches. Many telemetry efforts address R2 by leveraging sketches [6], [20]–[23] to gather approximate query results under the stringent operation and memory limitations faced in the data plane. However, the accuracy of sketches is tightly coupled to both the resources allocated (*e.g.*, number of hash functions or number of counters) and the underlying composition of traffic (*e.g.*, number of flows) making sketches insufficient for R3 and R4. ElasticSketch [7], which is explicitly concerned with changes in packet rate, available bandwidth for telemetry reports, and flow size distribution, may appear to be an exception to this. However, ElasticSketch's approach—which, upon detecting that the sketch associated with "heavy keys" is full, requires doubling the size of that sketch—does not directly address the specific notion of traffic dynamics implied by R3 and is non-trivial to implement in switch hardware (hence failing R5). Moreover,

ElasticSketch fails to address R4 or R1 since the sketch's implementation requires fixing a single flow key. Similarly, UnivMon [23] satisfies R4 by computing multiple metrics from a single sketch, but also fails to satisfy R1 because the sketch requires fixing a single flow key. Given R5, running independent parallel instances of sketches for all possible flow keys (as suggested in [13]) is infeasible (*e.g.*, consider supporting arbitrary-length source prefix queries using 32 independent sketches).

Software-based approaches. Several prior efforts leverage the capabilities of general-purpose CPUs to process traffic queries. For example, Trumpet [24] installs triggers on end hosts, OmniMon [25] and switch pointer [26] share tables between end hosts and switches in network, and SketchVisor [27] and NitroSketch [28] tune sketch-based approximation techniques for virtual switches. While these approaches work well in settings like data centers where all infrastructure is under a single administrative domain, in many settings (*e.g.*, campus or enterprise networks) it is too expensive (in terms of infrastructure cost and/or latency) to pass all packets through software and impractical to instrument end hosts.

Scheduling distributed stream processing operations. Several efforts [29]–[33] address the challenge of efficiently scheduling stream processing operations to maximize resource utilization. However, these efforts do not consider the particular types of accuracy and latency constraints encountered in scheduling telemetry operations on switch hardware.

Limitations of current hardware-based approaches. To illustrate the limitations of current static approaches [1]–[3] in dealing with R3 and R4, we implement the two queries mentioned in Example II.1 and run them over a traffic excerpt from the MAWILab data set [34] which features pronounced traffic dynamics. This excerpt starts with relatively stable traffic, then suddenly, due to an actual DDoS attack or other causes (which we do not claim to identify), around the 20^{th} 5 s time window (or *epoch*) contains a large number of sources sending regular pulses of traffic. As suggested in [2], [3], we use bloom filters tuned for the initial normal traffic to approximate the lists of distinct pairs required by the first stage of both queries.

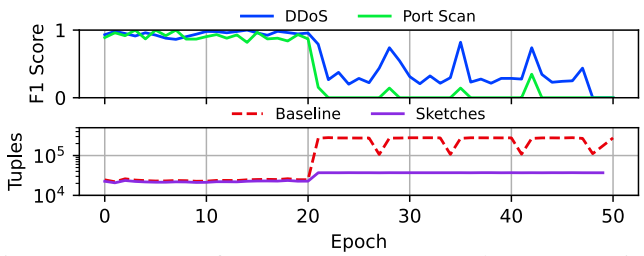


Fig. 1: Accuracy of concurrent DDoS and port scanning queries under extreme traffic dynamics.

Figure 1 shows the F1 score[3] of these approximate query implementations along with the number of tuples returned to the collector in each epoch. Before the change in number of sources, the approximation methods for both queries return highly accurate results while sending relatively few tuples. However, when the number of sources increases, the approximation accuracy of both queries suffers since the actual number of ground truth tuples (the "Baseline" trace) far exceeds the number each query was tuned for. Taking the static approach in this example shows that *when certain events of interest occur, the accuracy of multiple queries can be significantly impacted due to fixed assumptions about traffic composition.* Of course, the telemetry system initially could have tuned these queries for the anticipated number of sources, but this would lead to significant wastage of resources under normal traffic conditions. Moreover, it is hard to know what to tune for without prior knowledge of such an anomaly in particular and changes in traffic composition in general.

III. *DynATOS+* System Design

A. Overview

To tackle the above-mentioned limitations, we build *DynATOS+*. At its core, *DynATOS+* is composed of three main components as shown in Figure 2. Network administrators submit queries along with accuracy goals as mentioned in § II-A. They can express these accuracy goals either in absolute terms (as target standard error) or in relative terms (as target coefficient of variation) and submit them to the scheduler via a high-level REST API. The scheduler then translates queries into their primitive operations and constructs schedules for how these operations should be run on switch hardware given a stateful awareness of current traffic compositions based on observed results of previously executed queries. These schedules are then handed to a (state-less) runtime component which communicates with switch hardware to execute the primitive operations and collect intermediate results. Once ready, the runtime component gathers all results and passes them back to the scheduler and network administrators.

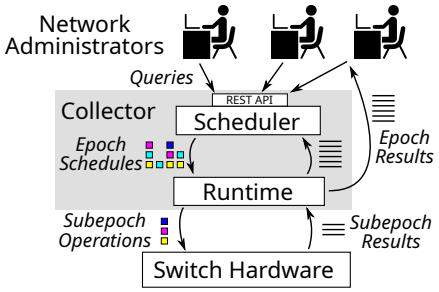


Fig. 2: Architecture of *DynATOS+*.

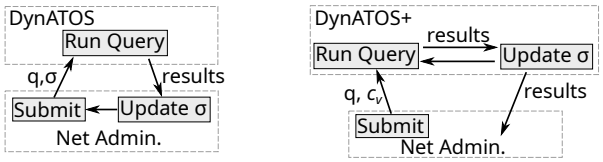
B. Difference between DynATOS+ and DynATOS

The key difference between *DynATOS+* and *DynATOS* [10] is the addition of a new type of per-query accuracy goal that network administrators can specify and that we measure using the coefficient of variation (CV).[4] Compared to *DynATOS*'s use of the standard error to express per-query accuracy goals, from a network administrator's perspective, using the CV is a more reasonable and practical choice because the CV measure is relative to the magnitude of the underlying values computed, that is, it is "unit-less" or "dimension-less". By basing accuracy goals on CV and automatically converting to standard error based on current traffic conditions, *DynATOS+* frees

[3]Computed by comparing with ground truth, the F1 score is a measure of query accuracy defined as the harmonic mean of precision and recall.

[4]Coefficient of variation is a standard statistic defined as the ratio of standard deviation to mean.

network administrators from a potentially painful accuracy goal tuning process (illustrated in Figure 3a). For example, the network administrator in Example II.1 can simply specify an accuracy goal by setting a CV target of $\pm 10\%$ for both DDoS and Port Scan queries and then let *DynATOS+* automatically find the corresponding standard error.



(a) Original method using only standard-error accuracy goal σ [10].

(b) Updated method using coefficient of variation accuracy goal c_v.

Fig. 3: Comparison of method to tune target accuracy goals between *DynATOS* and *DynATOS+*.

To further illustrate the advantage of CV-based accuracy goals in *DynATOS+*, consider a scenario where new queries must be added without prior knowledge of expected query results. Suppose the network administrator determines that the increase in observed sources is not caused by a DDoS attack, but by a flash crowd of requests to a particular web service. The administrator then seeks to submit queries to monitor new TCP connection rates across a set of destination servers with a relative accuracy goal (expressed relative to the mean number of TCP connections) to help tune their load balancing to absorb the flash crowd. However, since traffic to these particular servers has never been queried in this way, the administrator does not know what value to choose for the target standard-error σ.

As shown in Figure 3a, when using *DynATOS*, the network administrator has to manually submit the TCP connection rate query with exploratory settings of target standard error goal σ (*e.g.*, $\sigma \in \{\pm 100, \pm 200, \pm 400\}$) and adjust σ over the course of a couple of queries before a reasonable value is found. In *DynATOS+*, on the other hand, the administrator can simply specify a single accuracy goal based on the scale-free coefficient of variation (*e.g.*, $c_v = \pm 10\%$) and *DynATOS+* automatically adjusts σ to satisfy the requested c_v as shown in Figure 3b. We describe more details of *DynATOS+'s* c_v-based accuracy goals in § IV.

C. Scheduling horizon.

Since queries can arrive at any time, we must decide when and for how far into the future resources should be scheduled. We first examine several possible approaches to this problem, then describe our approach in the next paragraph. One option is to compute the schedule each time a new query arrives and adjust all existing queries to the new schedule. While this option minimizes the time a query has to wait before it can start executing, it complicates the realization of accuracy and latency goals since the duration of the scheduling horizon (*i.e.*, until the next query arrives) is unknown when forming the schedule. Alternatively, we could compute the new schedule each time all queries in the prior schedule terminate. While this option ensures schedules can be executed exactly as planned, newly submitted queries may experience a longer delay.

We choose, instead, to make scheduling decisions at fixed windows of time which we call *epochs* (*e.g.*, every 5 s). This allows a balance between the two schemes mentioned above: queries must wait at most the duration of one epoch before executing and during an epoch queries are ensured to execute according to the schedule. In particular, we divide the scheduling epoch into N *subepochs* and our scheduler assigns subsets of the submitted queries to each subepoch as shown in Figure 4. Subepochs provide flexibility to schedule different queries at different times while also providing concrete resource allocation units. Queries submitted during an epoch are checked for feasibility and only considered in the following epoch. For example, in the figure, Q4 is added sometime during epoch 2, but cannot be scheduled until epoch 3. During the epoch, the scheduler collects intermediate results for each subepoch in which a query is executed and aggregates these subepoch results based on the query's aggregation operation. Once an epoch completes, results of complete queries are returned, while new and incomplete queries (*e.g.*, queries that have not yet met their accuracy goal, or queries that have a longer latency goal) are considered for the next epoch. For example, in Figure 4 Q3 completes execution in the second subepoch of epoch 2 and its results are returned during the scheduler invocation before epoch 3. We further assume that each query executes over traffic in a single epoch and telemetry tasks requiring longer measurement durations than our scheduling epoch can simply re-submit queries.

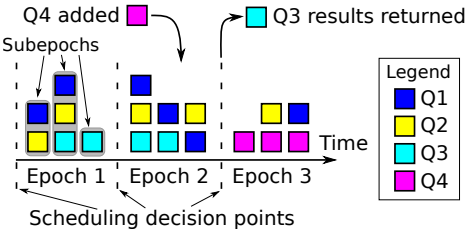


Fig. 4: Example of scheduling 4 queries with $N = 3$ subepochs per epoch.

D. Design Challenges

In order to successfully leverage the ideas introduced in § III-A to satisfy the goals set forth in § II-B, we must solve several concrete design challenges discussed below.

D1: Approximating generic query results. Efforts like Marple and Sonata develop expressive query description languages which map into data plane computation models. However, approximation of query operations is often necessary due to limited data plane resources and massive traffic volumes. It is unclear how state-of-the-art approximation methods can be leveraged to work with queries expressed in languages like Marple or Sonata. As illustrated in § II-C, the currently proposed baseline approach of simply replacing stateful reductions in Sonata queries with sketch-based primitives requires prior knowledge of worse-case traffic conditions and does not perform well under dynamic traffic scenarios.

On the other hand, directly setting a fixed number of subepochs in which to run query operations as discussed in § III-A does not consistently translate into query result

accuracy. To illustrate, Figure 5 shows the accuracy (as F1 score) of four queries on a single trace with a fixed allocation of 6 out of 8 subepochs (details of the queries, metrics, and settings used in this experiment are given in § VI). The accuracy of each query is clearly different with median over the trace ranging from ∼0.48 for TNC to ∼0.93 for DDoS demonstrating the challenge in reasoning about query accuracy from the number of subepochs in which the query is executed.

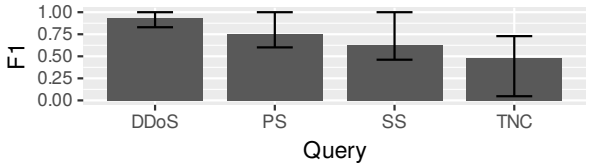


Fig. 5: Median F1 score of various queries on a single trace (MAWILab [34] 2022-08-30) under fixed allocation of 6 out of 8 subepochs in all epochs. The different accuracy achieved by each query illustrates the challenge of translating the number of subepochs executed to query accuracy.

D2: Estimating accuracy of approximations. Approximate query results must be accompanied with a sound estimate of their accuracy. This is critical for network administrators to understand the system's confidence in detecting a particular event or reporting a particular metric and equally critical in dynamic telemetry systems to inform the balance of resources between approximate queries. Prior efforts have made progress towards this goal [8], [12], [19], but none anticipate accuracy estimation for current state-of-the-art generic query descriptions in dynamic telemetry systems.

We extend the illustration of Figure 5 by considering a single port scan (PS) query run in a fixed number of subepochs. Each line in Figure 6 shows distribution of accuracy achieved by PS (run with the corresponding number of subepochs) over a sample of 28 traces from MAWILab [34]. The wide spread of each distribution on the x-axis indicates that query accuracy varies significantly in response to different traffic compositions in the sampled traces, regardless of number of subepochs. Even when all 8 subepochs are allocated, the query still achieves less than perfect accuracy on some traces due to the brief system down times during reconfiguration between subepochs. As a result, even if the relationship between number of subepochs and accuracy could be captured statically for individual queries, query accuracy would still vary greatly depending on the underlying traffic composition.

D3: Allocating finite hardware resources among variable sets of queries under traffic dynamics. Very few prior efforts address the need of a telemetry system to evaluate multiple concurrent queries on finite hardware resources. In order to handle traffic dynamics, such a system must dynamically update resource allocations based on the estimated accuracy of each query. Moreover, since it is possible that the given resources will be insufficient to meet the accuracy of all queries, such a system must enable network administrators to express query priorities and allocate resources with respect to these priorities.

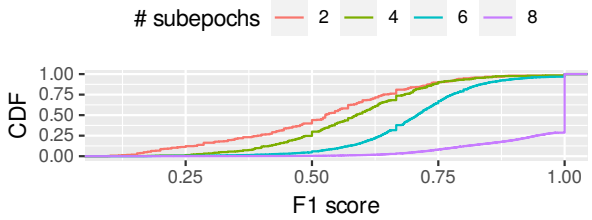


Fig. 6: Distribution of F1 score (w.r.t. ground truth) of the port scan query for different (fixed) numbers of subepochs over sample of 28 traces from MAWILab. For each fixed number of subepochs, the query achieves a wide range of F1 scores over different traces implying the system must dynamically estimate accuracy.

E. Our Solutions

We develop a novel approximation method based on cluster sampling theory and runtime programmable capabilities to address D1 and D2. Cluster sampling is known to be a good fit for scenarios where the overheads (*e.g.*, cost) of sampling large groups of the population (*e.g.*, subepochs) are significantly lower than the overheads of sampling individual population members (*e.g.*, packets) [35]. Runtime programmability exposes exactly such a scenario: large groups of packets can be sampled each subepoch and only (small) aggregate results need to be reported to the collector. In contrast, per-packet sampling, where each packet needs to be considered a candidate for sampling, incurs non-trivial per-packet overheads (in switch hardware) and a copy of each sampled packet needs to be sent to the collector resulting in non-trivial communication overheads.

We leverage cluster sampling to address D1 by applying it to the first aggregation operator in multistage queries. For example, in the DDoS query we only approximate computation of the distinct source-destination pairs list and execute all subsequent operations exactly. The intuition behind this is that each aggregation operator in a telemetry query reduces the volume of data passed to the next operator. Therefore, reducing the resource requirements and volume of data emitted from the first aggregation reduces the load on all subsequent operators.

Cluster sampling naturally addresses D2. As the underlying traffic composition changes, each sampled subepoch presents a snapshot of the changed traffic composition. § IV describes the details of how we use the formal error bounds of cluster sampling to actively adapt resource allocations as traffic composition changes across multiple epochs. Compare this with sketch-based traffic monitoring. Because a sketch's accuracy degrades as the number of keys increases (see Figure 1), it is difficult to tell from the sketch counters alone if the traffic composition has changed. Moreover, without knowing the ground truth traffic composition, it is difficult to determine the accuracy of a sketch-based result in order to drive allocation decisions in future epochs.

To address D3, we integrate our approximation technique in a scheduler that determines how a number of concurrent queries should be executed on a single switch hardware,

balancing resources between queries to satisfy accuracy and latency goals set by network administrators. As described in § V, our scheduler uses a novel multi-objective optimization formulation of the problem of when to run which queries given query priorities and resource constraints. This formulation allows the scheduler to balance between the goals of multiple concurrent queries, sometimes allocating less than the exact number of subepochs when queries have lower priority and resources are scarce (*e.g.*, due to a large number of concurrent queries).

Finally, we develop a runtime system leveraging these ideas to efficiently execute schedules on switch hardware, gather intermediate results, apply factors to correct for sampling, and return results to network administrators in a high-level format. Based on these results, administrators can then decide to execute new queries in the subsequent epoch, or to re-execute the same queries.

To illustrate how our key ideas apply to telemetry system design, we return to the examples introduced in § II-A. First, cluster sampling method can be applied to the first aggregation stage of a wide range of queries including the DDoS, port scanning, and TCP latency queries. Second, when the number of sources increase in Example II.1, each sampled subepoch in our system will return a proportionally increased number of tuples thereby increasing the volume of reports exported while maintaining high accuracy. Compare this to sketches where the increased number of sources would increase the number of hash collisions reducing accuracy of results. Note that state-of-the-art approaches [36], [37] dynamically allocate new sketches when the number of keys changes (to preserve sketch accuracy). However, this approach is infeasible in current switch hardware. Finally, when the network administrator begins digging into the root causes of TCP latency in Example II.2, our scheduling approach handles the burst of queries by adapting the optimization problem to account for the new query's accuracy and latency goals while assigning query operations to limited switch hardware.

F. Limitations and Assumptions

Monitoring problems addressed by *DynATOS*+. As shown in Figure 7, *DynATOS*+ can monitor traffic queries whose first steps (in the Sonata [2] paradigm) are filter, key-by, reduce followed by arbitrary post-processing. In the current work we only apply approximation to the first three operators (*i.e.*, to the first aggregation) and compute post-processing exactly. Moreover, *DynATOS*+*'s* approximation method implies the traffic features computed by these queries satisfy the following assumptions.

- Feature values do not fluctuate excessively over measurement durations of one or two seconds.
- The monitoring task can be implemented using features gathered at a single point in the network.
- Features are constructed from packet header fields and/or other switch-parsable regions of the packet.
- Features can be computed using atomic filter, map, and reduce operations.

Under these assumptions monitoring tasks like detecting microbursts [38], identifying global icebergs [39], and detecting patterns in TCP payloads [40] cannot be efficiently executed using *DynATOS*+. However, tasks like the DDoS, port-scanning, and latency detection examples of § II-A, along with a wide range of tasks considered in prior efforts with similar assumptions (*e.g.*, [1], [2], [12]) can be effectively executed using *DynATOS*+.

`Filter` *pred* → `KeyBy` *key* → `Reduce` *op* → Post-proc.

Fig. 7: Visualization of type of queries supported by *DynATOS*+ as a pipeline of atomic operations.

Switch hardware model. In the following, we assume a restricted runtime programmable switch hardware model. In this model, switch hardware is able to execute the first `Filter`, `KeyBy`, and `Reduce` operators shown in Figure 7 for a number of independent queries maintaining dynamic allocation of per-key state between queries. Similar to Newton [3], our switch hardware allows arbitrary parameterization of these operators *at runtime*. For example, switch hardware could execute the filter and reduce commands required by the Sonata TCP new connections queries for a period of time, then quickly (*e.g.*, within a few milliseconds) be re-programmed to execute the filter and reduce commands required by the Sonata DDoS query. We note that our scheduling methods are independent of this particular switch hardware model and could readily be applied to more fully programmable ASICs [5], [15].

Network-wide scheduling. Ultimately, administrators need to query traffic across different logical or physical domains of their network. This implies that telemetry systems should collect information from a distributed set of switches (or other monitoring points) and provide a global view of network traffic. In this work, we consider only a single monitoring point (*e.g.*, a critical border switch) and leave the challenges of distributed scheduling of telemetry operations to future work. Nonetheless, a single switch deployment on a enterprise or data center border switch can still be highly effective in executing the types of queries considered.

IV. Time-Division Approximation in *DynATOS*+

Accuracy tradeoff. Given fixed scheduling epochs, *DynATOS*+ trades off accuracy for reduced resource requirements by sampling a subset of the subepochs in which to execute a particular query. Suppose the query executes in a total of E epochs and that each epoch is divided into N equal-duration subepochs. Let $t_{i,j}$ be the query's result in the i-th subepoch of the j-th epoch, S_j be the set of which subepochs are actually sampled in the j-th epoch, $n_j = |S_j|$ be the number of subepochs sampled in the j-th epoch, and $s^2_{t_j}$ be the sample variance of the $t_{i,j}$'s in the j-th epoch. Using results from cluster sampling theory [35], the estimator

$$\hat{t}_E = \frac{1}{E}\sum_{j=1}^{E}\frac{N}{n_j}\sum_{i\in S_j} t_{i,j} \tag{1}$$

can be shown to be unbiased for the mean ($t_E = \frac{1}{E}\sum_{j=1}^{E}\sum_{i=1}^{N} t_{i,j}$) and has standard error

$$SE(\hat{t}_E) = \frac{N}{E}\sqrt{\sum_{j=1}^{E}\left(1-\frac{n_j}{N}\right)\frac{s_{t_j}^2}{n_j}}. \quad (2)$$

We use Equation 1 to estimate query results after E epochs and Equation 2 to determine when accuracy goals have been fulfilled. Assuming the query has already executed in $E-1$ epochs without achieving the target standard error σ, we can rearrange Equation 2 as

$$n^{acc} = \frac{s_{t_E}^2 N^2}{E^2\sigma^2 - \left(\sum_{j=1}^{E-1}\text{Var}\left(\hat{t}_j\right)\right) + N s_{t_E}^2} \quad (3)$$

to estimate n^{acc}, the number of subepochs in which a query should execute in the E-th epoch. Details of this rearrangement are given in Appendix A-A of our Supplementary Materials. Note that if $\sigma = 0$, then $n^{acc} = N$ and the query will be executed in all of the subepochs in its first epoch. As σ increases, n^{acc} decreases freeing more of the subepochs for other queries.

Latency tradeoff. In addition to the accuracy tradeoff discussed above, we can tradeoff latency for reduced resource requirements by executing a query's operations across several epochs. The key observation enabling this tradeoff is that by spreading the sampled subepochs over several epochs, the query can reduce its per-epoch requirements while still attaining its accuracy goal. Network administrators leverage this tradeoff by specifying larger latency goals on queries that do not require fast returns.

Suppose a particular query has a latency goal of $\tilde{E}$ epochs. Again, assuming the query has already executed in $E-1$ epochs, we need to estimate the number of subepochs in which the query should be allocated n^{lat} in the E-th epoch with $1 \leq E \leq \tilde{E}$. First, we break the sum in Equation 2 into *past* ($1 \leq j < E$) and *future* ($E < j \leq \tilde{E}$) components. We then have,

$$n^{lat} = \frac{s_{t_E}^2 N^2}{\tilde{E}^2\sigma^2 - N^2\left(past + future\right) + N s_{t_E}^2}. \quad (4)$$

While the $past$ component can be calculated directly using observations from prior epochs, the $future$ component must be estimated based on the number of subepochs the query expects to receive in future epochs. Administrators can tune this expected number of subepochs based on current and expected query workloads. (See Appendix A-B of our Supplementary Materials for more details.)

Approximation based on relative error goals. A key challenge with the formulation described in Equations 3 and 4 is that $s_{t_j}^2$ (and hence $SE(\hat{t}_E)$) varies based on several aspects of traffic (*e.g.*, burstiness) and system parameters (*e.g.*, epoch duration). As a result, it is often challenging to determine an appropriate value of target standard error σ before a query is run. We address this challenge for the common scenario where network administrators submit the same (or similar) query (queries) in contiguous epochs by accepting *relative* accuracy goals which we express using the coefficient of variation (see III-B), call the *target coefficient of variation*, and denote by c_v. For queries submitted with c_v (instead of σ) as their accuracy goal, the *DynATOS+* scheduler (see Figure 2) maintains an internal estimated target standard error $\hat{\sigma}$. When making scheduling decisions before each epoch, *DynATOS+* uses the current value of $\hat{\sigma}$ in place of σ in the same methods described previously in this section and in § V. After each epoch, *DynATOS+* computes a new target standard error $\hat{\sigma}' = c_v \cdot \hat{t}_E$ and updates $\hat{\sigma}$ to follow the new target using EWMA (*i.e.*, $\hat{\sigma}_{new} = \alpha\cdot\hat{\sigma}' + (1-\alpha)\cdot\hat{\sigma}_{old}$). Note that, although we could have directly adapted Equations 1-4 using the definition of c_v, we found that doing so translated the inter-epoch burstiness of the point estimate $\hat{t}_E$ into n^{acc} and n^{lat}, making them too unstable to drive consistent scheduling decisions. The EWMA-based approach smooths over local burstiness while still removing dependence of the target accuracy goal on the relative magnitude of query results.

Correcting distinct operators. While the previous sections discuss foundations for making sound approximations of packet/byte counts, many useful queries also involve identifying and counting distinct elements. To correct estimates for a common class of such distinct queries (including the DDoS query considered in § II-A), we leverage the Chao estimator without replacement [41], [42]. The intuition behind the Chao estimator is that the number of rare elements in the sample (*e.g.*, the number of flow keys observed exactly once or twice across all subepochs of a query) give an approximation of the number of rare elements in the underlying traffic and hence can be used to remove the bias induced by flow keys missed due to sampling. Similar to the cluster sampling estimators described earlier in this section, the Chao estimator can be used to obtain point and standard error estimates based only on the observed samples. Due to limited space, we describe details of how the Chao estimator is applied in Appendix B of our Supplementary Materials.

Comparison with sketch-based methods. As discussed in § II-C, state-of-the-art telemetry systems primarily rely on sketch-based approximation. However, we argue that the cluster sampling method described in this section is preferred for situations where traffic composition may change in unexpected and unpredictable ways. Consider again the scenario described in Example II.1 where the number of distinct sources observed increases suddenly. Suppose another telemetry system was using count-min sketch [20] (which computes similar point estimates as the approach described in this section) in the same scenario. The accuracy of query results produced by count-min sketch is given by $\hat{t}_i < t_i + \varepsilon||t||_1$ where $\hat{t}_i$ is the estimated query result for the i-th distinct element (source-destination pair in the example), t_i is the ground-truth query result, $||t||_1 = \sum_i |t_i|$ is the ground-truth L1-norm of all observed elements, and ε is a constant based on the number of sketch counters allocated. When the number of sources observed increases, $||t||_1$ also increases proportionally loosening the upper bound on $\hat{t}_i$. However, $||t||_1$ is a ground-truth value (depending on t_i, *not* the estimate $\hat{t}_i$) which must be estimated offline and cannot easily be extracted from the sketch counters. As a result, the network administrator would receive no indication from the telemetry system that the error of results may be critically compromised. On the other hand, in

our approach each sampled subepoch will reflect the increased number of sources and estimated query results (Equation 1) and result accuracy (Equation 2) will continue to accurately reflect observed traffic.

V. Scheduling in *DynATOS+*

A. Optimization Formulation

We cast the task of generating query schedules as an optimization problem and adapt well-known techniques to generate schedules through this casting. We apply our optimization formulation every epoch to determine which queries should execute in each of the N subepochs as shown in Algorithm 1. First, in line 2 we use the DISENTANGLE method of Yuan et al. [43] to break the submitted queries Q into disjoint traffic slices K (based on their filter predicates) and save the mapping between queries and slices in $s_{i,k}$. Line 3 then computes the minimum number of stateful update operations required by the reduce operators of all queries in each particular slice. These steps ensure that, even when the filter predicates of multiple submitted queries overlap, we can use combined update operations in U and disjoint traffic slices in K to compute all queries on a single switch hardware stage. Next, lines 4 through 6 compute estimates of the memory and subepoch requirements of each query. Finally line 7 creates and solves the optimization problem described below. If a feasible solution cannot be found, line 9 falls back to a heuristic scheduling method described in our Supplementary Materials.

Algorithm 1 Method for determining subepoch schedule

```
1:  procedure GET-SCHEDULE(Q, u, SE)
2:      K, s ← DISENTANGLE(Q)
3:      U ← COMBINE-UPDATES(u, K, s)
4:      m ← ESTIMATE-MEMORY
5:      n^acc ← EQUATION 3(σ)
6:      n^lat ← EQUATION 4(σ, E)
7:      d ← SOLVE-OPTIMIZATION
8:      if d is infeasible then
9:          d ← GET-HEURISTIC-SCHEDULE
10:     end if
11:     return d
12: end procedure
```

Inputs. Table II shows the particular inputs and outputs of this optimization problem. Of the input variables, t_k, u_i, $s_{i,k}$, T, A, and M are known exactly based on submitted query requirements and available switch resources, while m_i, n_i^{acc}, and n_i^{lat} must be estimated based on observation of past epochs. Our current implementation uses EWMA to estimate m_i and $s^2_{t_E}$ (as required by n_i^{acc} and n_i^{lat}) independently for all update operation types. We leave exploration of more sophisticated estimation approaches to future work. Scheduling decisions are encoded in the $d_{i,j}$ indicator variables which determine which queries should execute in each subepoch. We do not consider the division of switch memory between queries since memory is dynamically allocated during the aggregation operation (see § III-F).

Variable	Description
Q	index set of queries ready for execution
SE	index set of subepochs
K	index set of all disjoint traffic slices
U_k	index set of all update operations in slice k
t_k	number of TCAM entries required by slice k
u_i	index of update operation required by query i
$s_{i,k}$	indicator that query i requires slice k
m_i	memory required in each subepoch by query i
n_i^{acc}	number of subepochs required for accuracy goal for query i (§ IV)
n_i^{lat}	number of subepochs required for latency goal for query i (§ IV)
T	total available TCAM entries
A	total number of available switch ALUs
M	total available SRAM counters
$d_{i,j}$	indicator that query i executes in subepoch j

TABLE II: Variables used in optimization formulation of scheduling problem. The sole outputs $d_{i,j}$ determine the schedule for the next epoch.

C1: $\forall j \in SE,\ \sum_{k \in K} t_k \mathbf{I}\left[\bigvee_{i \in Q} d_{i,j} s_{i,k} = 1\right] \leq T$

C2: $\forall j \in SE,\ k \in K,\ \sum_{u \in U_k} \mathbf{I}\left[\bigvee_{i \in Q} d_{i,j} s_{i,k} \mathbf{I}[u_i = u] = 1\right] \leq A$

C3: $\forall j \in SE,\ \sum_{i \in Q} d_{i,j} m_i \leq M$

C4: $\forall i \in Q,\ \sum_{j \in SE} d_{i,j} \geq 2$

TABLE III: Scheduling problem constraints to respect (C1) TCAM capacity requirement, (C2) switch ALU capacity, (C3) SRAM capacity, and (C4) query minimal progress requirement. $\mathbf{I}[]$ is the indicator function.

O1: minimize $\sum_{i \in Q} \left| \left(\sum_{j \in SE} d_{i,j} \right) - n_i^{acc} \right|$

O2: minimize $\sum_{i \in Q} \left| \left(\sum_{j \in SE} d_{i,j} \right) - n_i^{lat} \right|$

O3: minimize $\sum_{i \in Q, j \in SE} d_{i,j} m_i$

TABLE IV: Objective functions considered in the multi-objective formulation.

Constraints. We impose the constraints shown in Table III to satisfy two high-level requirements: *(i)* respecting switch resource limits (C1, C2, C3) and *(ii)* forcing minimal progress in each query and ensuring variance estimates are well-defined (C4). Note that C2 captures the fact that if two queries rely on the same update operation, they can be merged to use a single ALU. In the case that the estimated quantity m_i turns out to be violated by traffic conditions in the subsequent epoch, we simply drop new aggregation groups once the available switch memory is totally consumed.

Objectives. In computing the schedule of each epoch, we consider the objective functions listed in Table IV. O1 seeks to satisfy accuracy goals by minimizing the distance to the value of n^{acc} computed in Equation 3, O2 seeks to satisfy latency goals by minimizing the distance to the value of n^{lat} computed in Equation 4, and O3 seeks to limit the maximum volume of data that needs to be returned from the switch in a single subepoch. We expose the Pareto front of these objective functions using linear scalarization which allows administrators to express the importance of each objective by

submitting weights and is computationally efficient.

Problem shape and size. Note that the only variables that are solved during evaluation of the optimization problem are the $d_{i,j}$ which determine which queries execute in each subepoch. Also, the number of constraints is linear in the maximum of the number of subepochs and the number of queries. For example, if 100 queries are to be executed in 8 subepochs, the resulting optimization problem has 800 binary variables and 124 constraints. Overall, the on-line optimization problems in *DynATOS+* are much smaller and simpler compared to the off-line optimization problems considered in, *e.g.*, Sonata [2] (since Sonata considers the product of all possible query partitionings and different prefix-level refinement plans). We observe that off-the-shelf optimization solvers (*e.g.*, [44]) are able to solve *DynATOS+'s* problems in a number of milliseconds (compared to the 20 minute time limit set on the optimization solver in Sonata) making their use in our on-line system feasible.

Additionally, since the number of queries ready for execution in each epoch is given by the particular query arrival process, the only system parameter that impacts problem size is the number of subepochs per epoch. Intuitively, configuring *DynATOS+* to run with more subepochs per epoch exposes more opportunities in the optimization problem to multiplex larger numbers of queries on the given processing resources. However, the number of subepochs is directly linked with both epoch duration and subepoch duration (in particular epoch duration is subepoch duration times number of subepochs). Assuming network administrators fixed epoch duration based on their particular monitoring requirements (since epoch duration fixes minimum latency across all queries), adding more subepochs also leads to shorter subepochs reducing the fraction of time spent actually monitoring traffic compared to the fixed amount of time required to reconfigure switch hardware betweeen each subepoch. In light of these facts, the number of subepochs (or, equivalently, subepoch duration) must be configured carefully to avoid either too constrained optimization problems (too few subepochs) or too much reconfiguration overhead (too many subepochs). We provide an empirical illustration of this tuning requirement in § VI-E which demonstrates that between 4 and 8 subepochs per epoch achieves a sort of sweet spot between these two extremes regardless of epoch duration.

Challenges of Online Optimization. *DynATOS+* inherits several techniques developed in *DynATOS* to deal with slow and infeasible optimization problems. The details of these techniques are described in [10] and Appendix C of our Supplementary Materials.

VI. Evaluation

A. Experimental Setup

Setting. In our prior work, we evaluated *DynATOS* on a BCM 56470 series [45] System Verification Kit (SVK) switch running BroadScan [46] which implements the telemetry operations described in § III-F. We extend our previous evaluation in this work using packet-level simulation.

Default parameters. We use five-second scheduling epochs to allow sufficient measurement duration without incurring excessive delay of results which must wait for epoch boundaries. We divide epochs into $N = 8$ subepochs so that the schedule has sufficient options for arranging queries without making subepochs too short to generate useful samples. We set objective weights to balance between priorities and suppose queries will get all future subepochs when evaluating Equation 4. We set the target CV to $c_v = 5\%$ by default. We set $\alpha = 1/2$ in the EWMA estimation described in § V-A. Unless indicated, points show median and error bars show 5^{th} and 95^{th} percentiles over all epochs of the trace.

Query workloads. As shown in Table V, we use *DynATOS+* to implement four of the telemetry queries originally introduced by Sonata [2] and used in several recent efforts. We report the accuracy of approximate implementations of these queries as F1 score (the harmonic mean of precision and recall) by comparing against ground truth computed offline. In addition to static queries, we generate dynamic query workloads based on random processes to evaluate *DynATOS+*. To simulate workloads with different levels of bursty query arrivals, we use a fractional Poisson process [47], [48] to generate query arrival times. Fractional Poisson processes generalize the classic Poisson process by adding a parameter μ which controls the spread of the inter-arrival distribution. When $\mu = 1$ the fractional Poisson process converges to the classic Poisson process. As μ goes to zero, the inter-arrival distribution spreads out inducing long-term dependencies or burstiness in the query arrival rate. To illustrate, Figure 8 shows synthetically generated fractional Poisson processes with the same mean rate, but different value sof μ. We normalize query arrival times so that all workloads have a mean query arrival rate of 1 query per second over a 900 s workload duration. Our workloads are publicly released at [49] to support validation of our results and to facilitate benchmarking of similar systems in the future.

Query Name	Description	# in [50]
DDoS	Find dests. that recv. from large number of sources.	5
Port Scan (PS)	Find sources that send to large number of ports.	4
Super Spreader (SS)	Find sources that send to large number of dests.	3
TCP New Cons. (TNC)	Find dests. that recv. large number of TCP SYN packets.	1

TABLE V: Queries used in this evaluation.

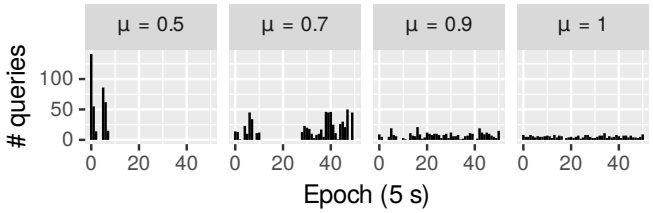
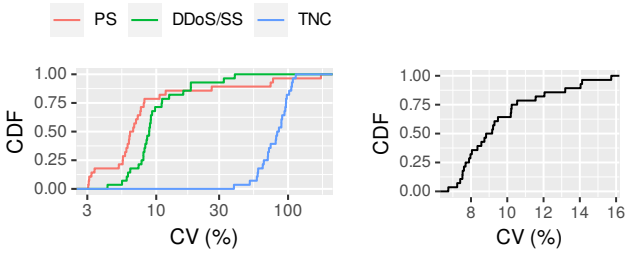


Fig. 8: Number of queries submitted each 5-second epoch for example fractional Poisson query arrival processes with different "burstiness" parameter μ. (Only first 50 epochs are shown for clarity.)

Traces. To understand how *DynATOS+* performs on a wider variety of representative traffic, we took a simple random sample of 28 days from MAWILab's [34] 2015 dataset. Each day consists of a 15 min trace starting at 2 pm. To illustrate changes in traffic composition, we compute the CV of number of keys per-epoch and number of packets per-epoch for each trace. Figure 9 shows the distribution of CVs over all traces in our sample. The distribution of key-based CVs (Figure 9a) and count-based CVs (Figure 9b) are wider for 2015 compared to other recent years indicating these traces provide higher diversity in traffic composition for evaluating *DynATOS+*.



(a) Number of distinct keys per epoch.

(b) Number of packets per epoch.

Fig. 9: Distribution of CVs over our sample of 28 traces from MAWILab [34].

Implementation. We implement the *DynATOS+* scheduler in ∼14k lines of C and C++. Following ProgME [43], we use BDDs to represent query filter conditions in our implementation of the DISENTANGLE algorithm (§ V-A). We use the open source CBC implementation [44] to solve the optimization problems described in § V-A. Our implementation also defers some result processing operations to the time spent waiting for results from switch hardware to improve efficiency. The simulation used in this section is built on the same software components implemented for our hardware prototype system. At a high-level the only modification we make is to substitute a separate software module which implements the same interfaces as the switch hardware controller. This software module also simulates the impact of time spent reconfiguring hardware by dropping r seconds worth of traffic each time it is reprogrammed. Based on our previous evaluation of hardware latency overheads [10], we set r to be 10 ms.

B. Impact of Traffic Dynamics

To evaluate the impact of traffic dynamics, we run each query from Table V over each of the 28 traces sampled from MAWILab. To demonstrate the tradeoff between accuracy (F1 score) and load on collector (tuples, bytes) each query is run over each trace for four settings of target CV $c_v \in \{0.1, 0.5, 1.0, 1.5\}$.

Figure 10 shows load on collector as a percentage of total number of bytes required to compute ground truth (y-axis) against F1 score (x-axis) for each setting of c_v summarized over all 28 traces. With the exception of the TCP new connections query, all queries achieve similar accuracy ranges for each c_v value. For example, at $c_v = 1.5$ DDoS and super spreader achieve median F1 score of ∼0.9 while port scan achieves median F1 score of ∼0.8 for ∼20% median reduction in bytes sent to the collector. As shown in Figure 9a, the underlying CV of the number of keys observed in each epoch is much higher for TCP new connections (median of ∼84%), likely a product of how this query only looks at SYN packets and each SYN packet is typically associated with a new flow. As a result, the relationship between c_v and F1 score is *quantitatively* different, though *qualitatively* follows a similar pattern as for the other queries. For example, at $c_v = 0.5$ TCP new connections achieves median F1 score of ∼0.9 and an ∼11% reduction in bytes sent to the collector. Overall, as a rule-of-thumb, we note that for distinct-count queries (*e.g.*, DDoS, Port Scan, and Super Spreader), c_v of up to 1.5 results in reasonable accuracy whereas for count-based queries like TCP new connections, c_v should be kept lower (*e.g.*, up to 0.5).

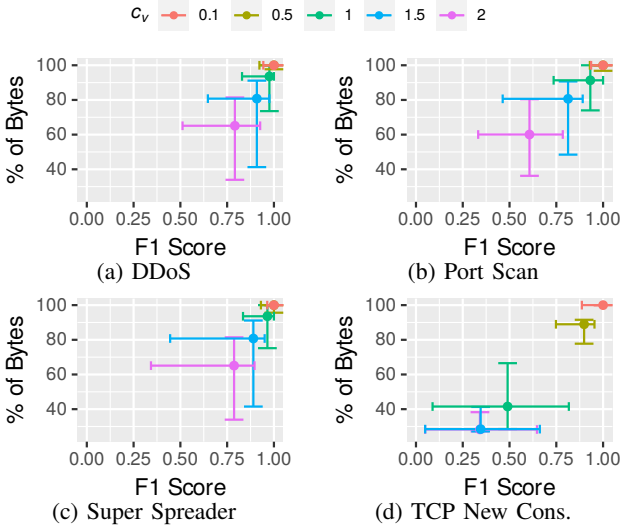
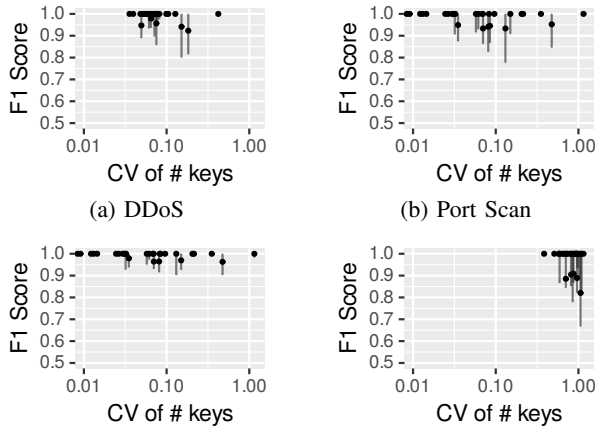


Fig. 10: Performance tradeoff for single queries over sample of 28 15-minute MAWILab traces for different target CV goals (colors show different target c_v).

Given that Figure 10 shows a relatively large range of F1 scores over all traces for each particular c_v, we further investigate how different properties of the underlying traces impact F1 score. In particular, we compute the CV of the number of keys in each epoch across each trace in our sample as a metric to summarise the trace's level of traffic composition dynamics.

Figure 11 compares the level of dynamics in each trace (x-axis) with the F1 score achieved by *DynATOS+* for a fixed setting of $c_v = 0.1$ (we observe qualitatively similar trends for other values of c_v). As expected, different queries observe different levels of traffic composition dynamics (*e.g.*, CV between 0.03 and 0.42 for DDoS compared to 0.38 and 1.15 for TCP new connections). We note that for most queries, F1 score is only weakly correlated with underlying trace dynamics demonstrating *DynATOS+'s* ability to provide consistent accuracy in the face of trace dynamics. The TCP new connections query is again a bit of an exception for



(a) DDoS (b) Port Scan

(c) Super Spreader (d) TCP New Cons.

Fig. 11: Performance for single queries at a fixed CV goal (0.1). Each point compares F1 score and CV of the number of keys per epoch (relative to each particular query type) for a single trace from our sample.

similar reasons as in Figure 10: since each key is typically only associated with a single SYN packet, when the number of keys varies, *DynATOS*+ has little opportunity to catch keys that arrived during un-sampled subepochs.

C. Impact of Query Workload Dynamics

To evaluate the impact of query workload dynamics on *DynATOS*+, we run dynamic query workloads generated by a fractional Poisson arrival process as described in § VI-A. To minimize the impact of trace dynamics, we use a single trace from our sample with relatively low CV across all query key types (in particular we use the trace from Feb. 22). We configure *DynATOS*+ to target $c_v = 0.1$ and compare against *DynATOS* with fixed σ set based on baseline measurements of observed standard deviation in the trace as in [10].

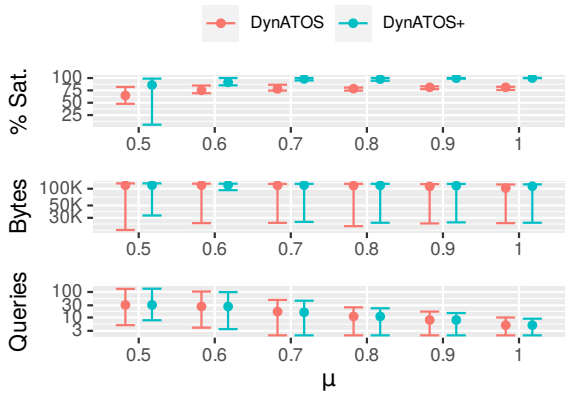


Fig. 12: Query satisfaction, bytes sent to collector each epoch, and number of queries each epoch as a function of workload burstiness (smaller μ indicates more bursty workloads) for a single trace showing the improvement of *DynATOS*+ over *DynATOS*.

Figure 12 shows query satisfaction (defined as the fraction of queries in the workload that achieve standard error less than σ) over 10 independent query workloads generated at each setting of arrival burstiness parameter μ. In addition to query workload satisfaction (top), we also plot the number of bytes returned to the collector (middle) and the number of queries executed per epoch (bottom). We observe that by automatically adjusting σ based on the target c_v, *DynATOS*+ finds a more optimal value for σ and is able to achieve consistently higher satisfaction compared to *DynATOS* (a median difference of 18% to 21% for all workload burstinesses) while inducing minimal increased load on collector (a median difference of less than 8 KB per epoch for all workload burstinesses). Note that the number of queries per epoch is larger for burstier workloads because we ignore epochs where no queries were run. Also, in some cases the query workload consists of a single burst of queries over the entire 15 m trace so that *DynATOS*+ does not have a sufficient number of epochs in which to tune the c_v to σ translation. This causes the lower error bar for *DynATOS*+ at $\mu = 0.5$ compared to *DynATOS* which uses a fixed (in this case higher) σ. In a real deployment where *DynATOS*+ is run for longer periods (*e.g.*, several hours), we expect query satisfaction would converge closer to the median in these plots.

D. Interaction Between Traffic and Query Workload Dynamics

To understand the interaction between traffic dynamics and query workload dynamics, we run the same query workloads used in Figure 12 over all 28 traces in our sample from MAWILab. Figure 13 summarises the results by showing the minimum and maximum median query satisfaction and bytes sent to the collector over all 28 traces for each workload burstiness setting μ. As in Figure 12, quantiles (shown here as different colors) are over the same 10 independently-generated workloads at each μ.

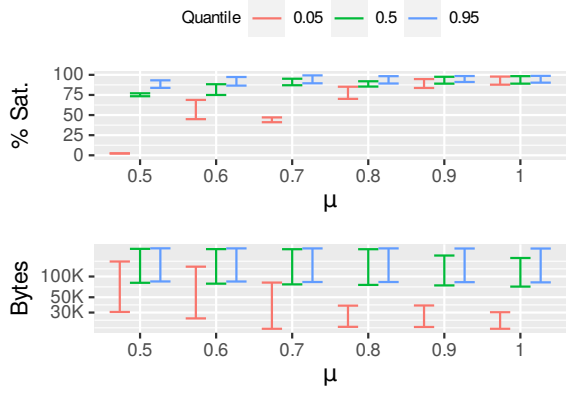


Fig. 13: Minimum and maximum over all traces of query satisfaction (for same quantiles as in Figure 12). The differences between different workload burstiness (across x-axis) is much larger compared the the differences among different traces (y-axis ranges) indicating workload burstiness has more impact on *DynATOS*+*'s* performance.

Similar to Figure 12, we observe that burstier workloads achieve lower satisfaction (∼73% for $\mu = 0.5$ compared to ∼89% for $\mu = 1.0$). On the other hand, the distance between minimum and maximum satisfaction across all traces (vertical height of the bars in Figure 13) is relatively consistent and small for all query workloads (an absolute difference of median from ∼4% to ∼13% for all workload burstinesses). The relatively larger range of bytes returned to the collector (∼68% difference in median across all workload burstinesses) demonstrates how *DynATOS+'s* sampling method automatically adjusts load on the collector to meet the different number of keys in different traces. Considering the fact that the per-epoch CV of number of keys considered ranges from ∼0.04 to ∼0.4 across the traces in our sample, these results indicate the methods in *DynATOS+* are more robust to dynamics of traffic composition compared to burstiness of query workloads. We envision developing new methods to improve *DynATOS+'s* handling of bursty query workloads as future work.

E. Sensitivity to Epoch and Subepoch Duration

The two most critical parameters in *DynATOS+* are the duration of epochs and the duration of subepochs. Since epoch duration determines the lower bound on how quickly query results can be delivered, we assume network administrators fix epoch duration based on their particular monitoring requirement. The key remaining parameter is then the duration of subepochs or equivalently the number of subepochs per epoch.

To understand the impact of subepoch duration on *DynATOS+'s* performance, we run *DynATOS+* over the same trace considered in § VI-C and the ten independent query workloads with $\mu = 1$. To expose the impact of query operation multiplexing, we limit the number of queries that can be assigned to run in a single epoch to twice the mean expected number of queries (*e.g.*, because the mean query arrival rate is 1 per second, for 8 s epochs we limit to 16 concurrent queries in each subepoch). Our query workload driver does not attempt to resubmit queries that are rejected when *DynATOS+* is in "fail-safe" mode leading to some fraction of queries going unanswered in some runs.

Figure 14 shows the resulting F1 score (top), bytes sent to the collector (middle), and fraction of queries submitted by the workload that actually receive answers, regardless of their accuracy goals (bottom). Note that due to the minimal progress constraint (C4 in Table III), we are limited to a minimum of two subepochs per epoch. Overall, F1 score is not significantly impacted by epoch duration indicating that network administrators can confidently choose the epoch duration best suited for their monitoring requirements without impacting *DynATOS+'s* performance.

Subepoch duration has a more complex effect on all metrics observed. First, longer subepochs tend to yield slightly higher accuracy (*e.g.*, median F1 scores increases from 90% to 94% for 8 s epochs) for queries that receive an answer. Second, shorter subepochs cause more bytes to be sent to the collector each epoch because intermediate results must be sent after each subepochs (*e.g.*, from 81K up to 192K

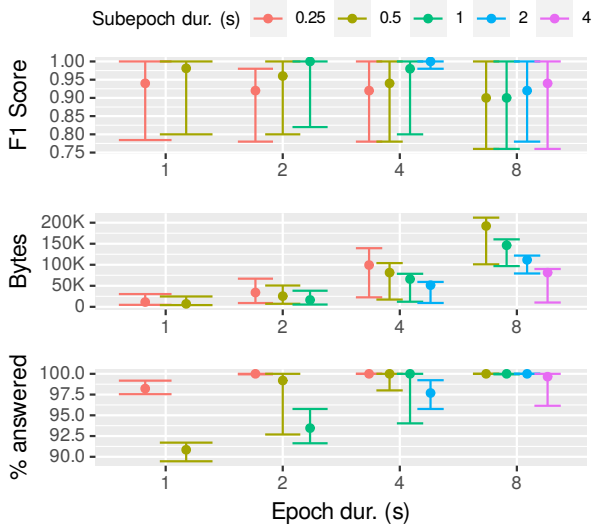


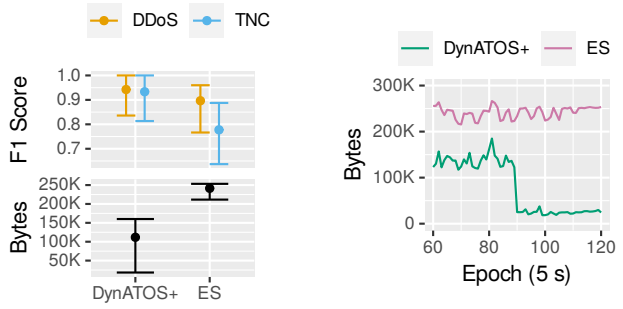
Fig. 14: Impact of epoch and subepoch duration on query accuracy, volume of traffic to collector each epoch, and fraction of queries answered.

for 8 s epochs). These two facts may seem to suggest that simply setting subepoch duration as long as possible yields optimal performance. However the bottom plot in Figure 14 exposes the cost of having fewer longer subepochs: due to the limited ability to multiplex query operations, a larger number of queries are rejected due to *DynATOS+* being in "fail-safe" mode. Moreover, fewer queries are answered for shorter epoch durations (*e.g.*, with two subepochs, a median of ∼90.8% for 1 s epochs compared to ∼99.9% for 8 s epochs) due to the fact that longer epoch durations smooth over bursts of queries that otherwise lead to infeasible scheduling problems at shorter epoch durations. In summary, we find that using between 4 and 8 subepochs exposes a sweet spot between *(i)* higher error and traffic to collector with more than 8 subepochs and *(ii)* reduced possibilities for multiplexing which cause more queries to go unanswered with fewer than 4 subepochs.

F. Comparison with Sketch Methods

ElasticSketch. To illustrate the challenge of switching between multiple queries with ElasticSketch [7] we consider a simple query workload which runs the DDoS query for 7.5 minutes, then switches to the TCP new connections query for the next 7.5 minutes. Note that this workload represents a wide range of scenarios where two or more queries with different filter conditions and different keys are required to run in sequence. We run two independent instances of ElasticSketch in parallel (one for each query). Since ElasticSketch doesn't support runtime reconfiguration on switch hardware, both instances run throughout the workload even though only one query output is used at a time. *DynATOS+*, on the other hand, can simply switch between queries after 7.5 minutes so only one query is run at a time.

We run this scenario over traffic from the May 29 trace in our sample since this trace has relatively dynamic composition (as measured by CV of number of keys per epoch). We set $c_v = 0.1$ and adjust the size of ElasticSketch to achieve



(a) Summary quantiles. (b) Load on collector

Fig. 15: Performance of *DynATOS+* and ElasticSketch on dynamic query scenario where the network administrator changes between DDoS and TCP New Connections queries. ElasticSketch requires sending >2× more bytes to the collector while achieving lower accuracy compared to *DynATOS+* because it cannot adapt to the change in queries.

slightly lower accuracy compared to *DynATOS+* as measured by F1 score. Figure 15a shows the actual F1 score achieved (top) as well as the number of bytes sent to the collector (bottom) over all epochs for the two methods described above. Even though it achieves lower accuracy for both queries, the overheads of maintaining parallel sketches in ElasticSketch require sending over 2× more bytes to the collector compared to *DynATOS+* (∼240KB compared to ∼110KB). As shown in Figure 15b, *DynATOS+* achieves lower load on collector by only sending results for one query at a time. In general, this example illustrates that even for simple dynamic query workloads, the ability to switch between queries at runtime leads to significantly lower overheads.

Newton. Newton [3] develops methods to dynamically change queries on-the-fly (similar to *DynATOS+*), but the Newton dataplane uses fixed-size sketch-based primitives which cannot adapt to changes in traffic composition.[5] To illustrate, we consider a scenario where the network administrator runs the Port Scan query over a trace from our MAWILab sample with pronounced changes in traffic composition (in particular Aug. 12). In particular, this trace has a relatively constant number of keys per epoch (∼18K) until the 135-th epoch when the number of keys spikes up by an order of magnitude (to ∼273K). We set the sketch sizes in Newton to achieve high accuracy on the first part of the trace and choose $c_v = 1.5$ so that *DynATOS+* achieves slightly lower accuracy compared to Newton for the first part of the trace.

Figure 16 shows, for each epoch of the trace (x-axis), the F1 score (top), fraction of bytes sent to collector compared to ground-truth (middle), and total number of keys in the underlying (ground-truth) traffic (bottom). Before the change in composition at epoch 135, Newton achieves high F1 score (median of 1.0) compared to *DynATOS+* (median of 0.8). However, when the number of keys changes, Newton's sketches become full leading to a significant reduction in F1 score (median of 0.24). The middle plot shows that the root cause of this is the relatively smaller number of bytes Newton sends to the collector compared to ground truth. *DynATOS+*, on the other hand, achieves consistent F1 score (median of 0.8) during the increase in number of keys by maintaining a consistent load on collector w.r.t. the total number of ground-truth keys that need to be reported.

[5] In particular, the Port Scan query considered here uses a Bloom filter [51] with a fixed number of bits to approximate the first "distinct" operator.

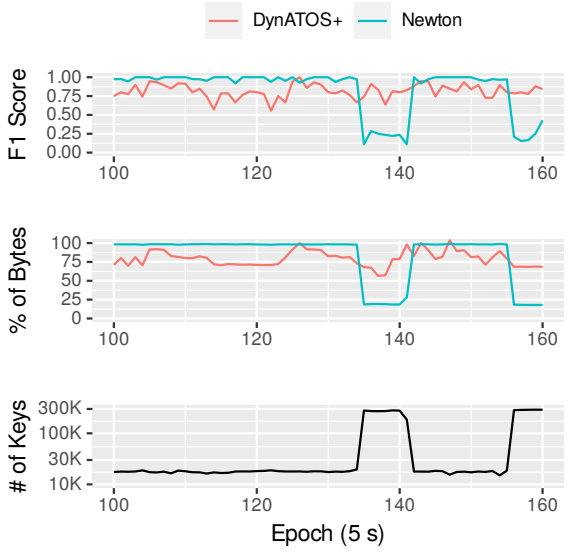


Fig. 16: Performance of *DynATOS+* and Newton running the Port Scan query on a excerpt from our sample of MAWILab traces with heavy traffic dynamics. Even though Newton is tuned for high accuracy, when the number of keys in the underlying traffic changes it suffers significant accuracy loss.

VII. Conclusion and Future Work

Current approaches to telemetry system design struggle to efficiently satisfy dynamism in query workloads and traffic workload composition. This work extends our previous effort on reframing telemetry systems as resource schedulers to develop efficient approximation and scheduling algorithms that expose accuracy and latency tradeoffs with respect to query execution to reduce hardware resource usage. In particular, this work adds a more intuitive form of target error to our scheduling method and presents an in-depth empirical evaluation of tradeoffs and system performance over a variety of traffic scenarios.

While we investigate the common sources of dynamics, both a *horizontal scheduling problem* (*i.e.*, how to design a scheduler to deal with those dynamics for multiple switch hardware stages or multiple distributed switches) and a *vertical scheduling problem* (*i.e.*, incorporation of computing resources, such as stream processing clusters and GPUs—both locally and at remote cloud data centers—into the pool of resources schedulable for telemetry tasks) remain. This opens up a wider question of *where*, not just *when* and *for how long*, telemetry queries should be executed. We plan to investigate this question as part of future work.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive feedback. We also thank Shahram Davari and Broadcom, Inc. for providing hardware and technical support for our testbed evaluation and Nima Nikkhah for assistance with ElasticSketch. This work is supported by the National Science Foundation through CNS-1850297, OAC-2126281, SaTC-2132651, and CNS-2212590, a Ripple faculty fellowship, and a Ripple graduate fellowship. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, Ripple, or Broadcom.

REFERENCES

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

[2] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018, pp. 357–371.

[3] Y. Zhou, D. Zhang, K. Gao, C. Sun, J. Cao, Y. Wang, M. Xu, and J. Wu, "Newton: Intent-driven network traffic monitoring," in *Proceedings of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2020, pp. 295–308.

[4] "BCM56870 series," https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series.

[5] "Intel Tofino," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

[6] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 29–42.

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

[8] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018, pp. 576–590.

[9] H. Zheng, C. Tian, T. Yang, H. Lin, C. Liu, Z. Zhang, W. Dou, and G. Chen, "Flymon: enabling on-the-fly task reconfiguration for network measurement," in *Proceedings of the conference of the ACM Special Interest Group on Data Communications (SIGCOMM)*, 2022, pp. 486–502.

[10] C. Misa, W. O'Connor, R. Durairajan, R. Rejaie, and W. Willinger, "Dynamic scheduling of approximate telemetry queries," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 701–717.

[11] S. Gangam, J. Chandrashekar, Í. Cunha, and J. Kurose, "Estimating TCP latency approximately with passive measurements," in *Proceedings of the International Conference on Passive and Active Measurement (PAM)*. Springer, 2013, pp. 83–93.

[12] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: Dynamic resource allocation for software-defined measurement," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 419–430, 2014.

[13] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. 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)*, 2021.

[14] "Intel ethernet switch FM6000 series product brief," https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[15] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[16] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2020, pp. 226–239.

[17] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen, "Runtime programmable switches," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 651–665.

[18] Y. Feng, Z. Chen, H. Song, W. Xu, J. Li, Z. Zhang, T. Yun, Y. Wan, and B. 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)*, 2022, pp. 635–649.

[19] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "SCREAM: Sketch resource allocation for software-defined measurement," in *Proceedings of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2015, p. 14.

[20] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[21] H. C. Zhao, A. Lall, M. Ogihara, O. Spatscheck, J. Wang, and J. Xu, "A data streaming algorithm for estimating entropies of OD flows," in *Proceedings of the ACM SIGCOMM conference on Internet measurement (IMC)*, 2007, pp. 279–290.

[22] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proceedings of the ACM SIGCOMM conference on Internet measurement (IMC)*, 2003, pp. 153–166.

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

[24] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016, pp. 129–143.

[25] Q. Huang, H. Sun, P. P. Lee, W. Bai, F. Zhu, and Y. Bao, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2020, pp. 404–421.

[26] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with SwitchPointer," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 453–456.

[27] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017, pp. 113–126.

[28] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019, pp. 334–350.

[29] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the Annual Middleware Conference*, 2015, pp. 149–161.

[30] T. Li, J. Tang, and J. Xu, "Performance modeling and predictive scheduling for distributed stream data processing," *IEEE Transactions on Big Data*, vol. 2, no. 4, pp. 353–364, 2016.

[31] A. Shukla and Y. Simmhan, "Model-driven scheduling for distributed stream processing systems," *Journal of Parallel and Distributed Computing*, vol. 117, pp. 98–114, 2018.

[32] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 168–178.

[33] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 377–392.

[34] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, "MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking," in *Proceedings of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2010.

[35] S. L. Lohr, *Sampling: Design and Analysis: Design And Analysis*. CRC Press, 2019.

[36] X. Zhu, G. Wu, H. Zhang, S. Wang, and B. Ma, "Dynamic count-min sketch for analytical queries over continuous data streams," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 2018, pp. 225–234.

[37] H. Zhu, Y. Zhang, L. Zhang, G. He, and L. Liu, "Cbfsketch: A scalable sketch framework for high speed network," in *2019 Seventh International Conference on Advanced Cloud and Big Data (CBD)*. IEEE, 2019, pp. 357–362.

[38] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich, "Catching the microburst culprits with snappy," in *Proceedings of the ACM Workshop on Self-Driving Networks*, 2018, pp. 22–28.

[39] S. Gangam, P. Sharma, and S. Fahmy, "Pegasus: Precision hunting for icebergs and anomalies in network flows," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2013, pp. 1420–1428.

[40] K. Borders, J. Springer, and M. Burnside, "Chimera: A declarative language for streaming network traffic analysis," in *Proceedings of the USENIX Security Symposium*, 2012, pp. 365–379.

[41] A. Chao and C.-W. Lin, "Nonparametric lower bounds for species richness and shared species richness under sampling without replacement," *Biometrics*, vol. 68, no. 3, pp. 912–921, 2012.

[42] A. Chao and C.-H. Chiu, "Species richness: estimation and comparison," *Wiley StatsRef: Statistics Reference Online*, pp. 1–26, 2014.

[43] L. Yuan, C.-N. Chuah, and P. Mohapatra, "ProgME: towards programmable network measurement," *IEEE/ACM Transactions on Networking*, vol. 19, no. 1, pp. 115–128, 2011.

[44] "COIN-OR Branch-and-cut MIP solver," https://zenodo.org/badge/latestdoi/30382416.

[45] "Trident3-X4 / BCM56470 Series," https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56470-series.

[46] "BCM56275 Gb/s Programmable Multilayer Switch Product Brief," https://docs.broadcom.com/doc/56275-PB.

[47] N. Laskin, "Fractional poisson process," *Communications in Nonlinear Science and Numerical Simulation*, vol. 8, no. 3-4, pp. 201–213, 2003.

[48] M. Politi, T. Kaizoji, and E. Scalas, "Full characterization of the fractional poisson process," *EPL (Europhysics Letters)*, vol. 96, no. 2, p. 20004, 2011.

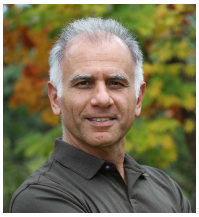
[49] "ONRG: DynATOS," https://onrg.gitlab.io/projects/dynatos/.

[50] "sonata-queries/sonata-queries," https://github.com/sonata-queries/sonata-queries, accessed: Nov. 2022.

[51] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

Chris Misa is a PhD student at the University of Oregon. His research interests include design and implementation of network traffic monitoring systems and statistical characterization of network traffic structure. Chris's work has been recognized by several research fellowships including a Ripple graduate fellowship.

Ramakrishnan Durairajan is an Associate Professor in the Department of Computer Science at the University of Oregon. His research has been recognized with multiple NSF awards, including NSF CAREER, Ripple faculty fellowship, UO faculty research award, several best paper awards, and has been covered in several fora.



Reza Rejaie Reza Rejaie is currently a Professor and head of Computer Science Department at the University of Oregon (UO). Before joining UO, he worked at AT&T Labs-Research in Menlo Park, California from 1999 to 2002. He received a NSF CAREER Award for his work on Peer-to-Peer streaming in 2005 and a European Union Marie Curie Fellowship in 2009. Reza has been a visiting professor at IMDEA Networks Institute, the Politecnico di Torino, and Sorbonne University. Reza is also a founding associated director of Oregon Cybersecurity Center of Excellence. Reza received his Ph.D. degree from the University of Southern California in 1999, and his B.S. degree in Electrical Engineering from the Sharif University of Technology in 1991. Reza is a Fellow of IEEE (2017) and a Distinguished member of the ACM (2022).

Walter Willinger is Chief Scientist at NIKSUN, Inc. Before joining NIKSUN, he worked at AT&T Labs-Research and at Bellcore Applied Research. He is a Fellow of ACM (2005), Fellow of IEEE (2005), AT&T Fellow (2007), and Fellow of SIAM (2009), co-recipient of the 1995 IEEE Communications Society W.R. Bennett Prize Paper Award and the 1996 IEEE W.R.G. Baker Prize Award, co-recipient of the 2005 and 2016 ACM/SIGCOMM Test-of-Time Paper Awards, and recipient of the 2024 IEEE Internet Award.