Performance-Driven Internet Path Selection

Maria Apostolaki
ETH Zürich
Switzerland
apmaria@ethz.ch

Ankit Singla
ETH Zürich
Switzerland
ankit.singla@inf.ethz.ch

Laurent Vanbever
ETH Zürich
Switzerland
lvanbever@ethz.ch

ABSTRACT

Internet routing can often be sub-optimal, with the chosen routes providing worse performance than other available policy-compliant routes. This stems from the lack of visibility into route performance at the network layer. While this is an old problem, we argue that recent advances in programmable hardware finally open up the possibility of performance-aware routing in a deployable, BGP-compatible manner.

We introduce ROUTESCOUT, a hybrid hardware/software system supporting performance-based routing at ISP scale. In the data plane, ROUTESCOUT leverages P4-enabled hardware to monitor performance across policy-compliant route choices for each destination, at line-rate and with a small memory footprint. ROUTESCOUT's control plane then asynchronously pulls aggregated performance metrics to synthesize a performance-aware forwarding policy.

We show that ROUTESCOUT can monitor performance across most of an ISP's traffic, using only 4 MB of memory. Further, its control can flexibly satisfy a variety of operator objectives, with sub-second operating times.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **Routing protocols**; **Control path algorithms**; **Network performance evaluation**; **Public Internet**; **Network dynamics**.

ACM Reference Format:
Maria Apostolaki, Ankit Singla, and Laurent Vanbever. 2021. Performance-Driven Internet Path Selection. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21), October 11–12, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3482898.3483357

1 INTRODUCTION

Internet routing uses cost-driven policies to select *one* interdomain path per destination along which to direct traffic. To select one path amongst multiple policy-compliant ones, the Internet's Border Gateway Protocol (BGP) uses particularly crude criteria rather than dynamically optimizing for performance. For instance, BGP will favor paths crossing fewer networks or paths crossing networks whose identifiers are smaller.[1] As a result, BGP selects routes that are often suboptimal in terms of throughput, latency, and reliability.

This problem is far from new and the sub-optimality of Internet routing is long-established [54, 58, 59]. Yet, despite several strong attempts [8–11, 54, 61], limited progress has been made. The problem is that enabling performance-aware routing is particularly challenging, requiring: scalable monitoring of path performance, handling path dynamics, stability and correctness of routing, and insurmountable resistance to any approach incompatible with BGP.

Despite the problem's difficulty and its long history, we posit its time to revisit this problem for three reasons.

First, Internet application requirements have evolved, with a sharper focus on *reliably high* network performance. For hyperscale Web services with numerous well-connected points-of-presence across the globe, BGP is, in fact, good enough *most of the time* [12]. However, even in these best-case environments, the benefits of reducing tail latency and performance variability in response to transient congestion are valuable enough for providers like Google and Facebook to invest in performance-aware routing [55, 64]. Google's Espresso showed that being able to dynamically reroute around transient congestion improved mean time between rebuffers in their video service by 35–170% [64]. Espresso explicitly pins these gains on being able to dynamically respond to performance *variability* across paths (rather than just average-case improvement from a one-time evaluation), thus underscoring the need for making path decisions based on *continuous assessments of the changing performance* of paths. Beyond Web services, other applications are even more demanding: in gaming, even small latency overheads can put players at a disadvantage [28]. The importance of tail latency as opposed to mean latency is also demonstrated in CDN's efforts to improve latency of the worst-performing clients[19]. Thus, *if* performance-aware routing were practical, the benefits would justify significant design effort.

Second, the available paths are increasingly diverse due to increased peering and the establishment of Internet Exchange Points (IXPs), which did not exist at BGP's first design iteration (1989). Further, if plans for satellite-based global Internet connectivity [18, 57] come to fruition, the performance gap across different paths will also increase. Two teams of researchers have separately argued in recent position papers [14, 44] that these satellite systems exhibit continuous changes in both the performance and availability of routes, and thus, will pose challenges to the performance-oblivious and slow-to-converge BGP routing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSR '21, October 11–12, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9084-2/21/10…$15.00
https://doi.org/10.1145/3482898.3483357

[1] One of BGP tie-breaking criteria is indeed to prefer routes announced by the router with the smallest IP address [51].

Third, the recent development of programmable switches that allow line-rate, per-packet data plane operations enables new design primitives. These heretofore unavailable primitives, as we shall show, drastically improve our ability to both evaluate and control multiple candidate routes.

Motivated by the above factors, we present **ROUTESCOUT**, a novel software-hardware co-design for performance-aware routing that runs at each edge of the network and independently controls the paths of the traffic exiting. ROUTESCOUT's data plane estimates loss and delay along different policy-compliant next-hop routes for different destinations. It leverages probabilistic data structures in programmable switches to aggregate delay and loss measurements on a per-destination-next-hop granularity. This in-data-plane aggregation eliminates the necessity of mirroring traffic to more powerful general-purpose hardware, thus alleviating: (a) bandwidth and compute overheads; and (b) deterioration in monitoring capabilities when most needed, under congestion. Past methods (§2) are incapable of producing such accurate, high-coverage, real-time, and low-overhead performance measurements for multiple candidate next-hops for many destinations.

The succinct measurements allow ROUTESCOUT's control plane to evaluate multiple policy-compliant candidate paths by measuring their performance systematically for small slices of live traffic. ROUTESCOUT then encodes the best path choices in the data plane using a small memory footprint. ROUTESCOUT enforces those choices gradually while continually monitoring performance to avoid self-induced congestion and, therefore, oscillations [30].

While ROUTESCOUT could be used by any Autonomous System (AS), for tractability of control, we trim the problem's scope: we take the perspective of a stub AS, which offers no transit services to other ASes. This eliminates the risk of multiple parties concurrently sensing and independently modifying the same end-to-end path leading to transient loops and instability. We humbly suggest that this "relaxation" still leads to a highly non-trivial and useful setting: stubs comprise 85% of all ASes;[2] and the majority of stubs are multi-homed and virtually all Internet traffic originates from some stub. In addition, despite sitting at the edge of the Internet, stubs often know several paths to reach each destination: our measurements on CAIDA AS-level topologies [2] reveal that the majority of them (55%) can use at least two equally-preferred paths for at least 80% of the destinations.[3] Stubs also tend to connect with their neighbors via redundant links, further increasing path diversity [47]. Finally, while ROUTESCOUT can only control paths *from* the stub, not towards it, the resulting reductions in round-trip time, and being able to avoid congestion/failures at least in one direction, are still valuable improvements.

ROUTESCOUT is carefully designed to run on available programmable switches, respecting constraints on memory, operations per packet, memory accesses per packet, and constraints on accesses to memory blocks across pipeline stages. It requires no coordination across ASes and works over unmodified BGP. Within an AS, it yields benefits starting with only one programmable switch deployed at the edge.

[2] A likely low estimate, computed from CAIDA's AS-level topology [2].

[3] For each stub we calculated the number of BGP-equivalent paths for 1000 randomly selected destination prefixes, following [29].

Our main **contributions** are the following:

- ROUTESCOUT, a system capable of rerouting traffic to test the performance of alternative routes to each destination prefix in a controlled and automated manner.
- Methods to compute delay and loss rates across different paths that are accurate and effective while respecting the constraints of data-plane hardware.
- Efficient interconnection between the control and data plane that allows: (a) fast, fine-grained, and asynchronous changes in the forwarding and monitoring policy; (b) fast, fine-grained, and low-bandwidth retrieval of statistics.
- An implementation of ROUTESCOUT on a Barefoot Tofino switch [5], with an evaluation of its control- and data-plane.

2 MOTIVATION

Performance-aware routing is an old problem [8, 10, 54, 58, 59], with several known solutions of varying ambition and complexity. Early work [32] narrowly targeted multi-homed end-users with perfect visibility over their performance, cost being their first priority, and direct links the only possible bottleneck. TeXCP [41] and MATE [25] focused on intra-domain routing, splitting traffic across already setup tunnels. We instead tackle the problem from the perspective of an AS picking routes to external destinations, with no end-host control and only observing its own traffic. In this setting, we discuss several alternatives for monitoring path performance, whose limitations make a case for ROUTESCOUT.

Active probing: One can actively probe routes with known tools [22, 36]. Yet, probes may not be representative of real traffic's performance — the volume of probing traffic is likely orders of magnitude less than the actual traffic, and some ISPs are known to treat probing traffic preferentially [24]. Most importantly, low-volume probing is inadequate for accurately measuring loss rate, while high-volume probing on all destinations via multiple paths would lead to significant extra load. Probing has been successfully used for performance-driven *intra-domain* routing in the past *e.g.,* Contra [35]. Yet, such techniques are impractical in our inter-domain context as they would require one *Vantage Point at each destination network*.

Passive sampling: Gathering statistics on live traffic is possible using sampling with sFlow [50] or NetFlow [23]. However, sampling simply does not capture performance — measuring these metrics requires capturing state across particular packets per flow (§4.2, §4.3), not arbitrary random samples.

Mirroring: While mirroring captures the requisite information, it does not scale and is inflexible [49]. To avoid congestion from mirrored traffic, one can rate-limit it, but this has limitations similar to sampling: naive rate-limiting will discard arbitrary packets across flows, impairing loss and delay estimation. Alternatively, one can target mirroring more narrowly, with systems like Everflow [67] and Stroboscope [60]. However, for continuous, high-coverage monitoring across Internet prefixes and potential next-hops, such methods would require a large and constantly changing set of monitoring rules in network devices. Further, even if we could dynamically match on a given number of flows per prefix and mirror only those (*e.g.,* with programmable switches to store flow identifiers), the mirrored traffic will still be burdensome.

As an illustration, consider an operator who wants to monitor the performance for traffic sent to $1K$ destinations over only 2 alternative next-hops and by mirroring only 50 flows per destination-next-hop pair. At the mean flow rate observed in CAIDA traces [1], such a design would require mirroring 25.7 Gbps of traffic. In contrast, by aggregating measurements directly in the data plane, RouteScout generates 108.4 kbps in performance reports, *i.e.,* at 287,000× higher efficiency.

End-system monitoring: Google [64] and Facebook [55] have recently shared their solutions for path-aware routing. These approaches leverage their unique control: one end of the monitored connections terminates at their own powerful servers, and the other at a client application that also supplies performance data. This is obviously infeasible for ASes.

Performance monitoring with programmable switches: RouteScout exploits programmable switches that open up avenues unavailable to past efforts. To the best of our knowledge, no prior work leveraging programmable switches fully addresses either the sensing/monitoring or flexible reroutes needed for performance-aware routing. Blink [34] detects outages, exploiting a failure-specific property: *failed* paths deterministically drop every retransmission of a packet. This property simplifies Blink's design but doesn't hold for congested paths. Observe that Blink can only detect the second retransmission of a packet, thus cannot measure loss rate or delay. Lossradar [66] detects losses between pairs of *deployed VPs*. Measuring *per-path loss-rate* though requires significant additional effort *i.e.,* adding per-path synchronized counters and mapping each lost packet to a path. Also, Lossradar does not measure delay. In-band Network Telemetry [43] provides *intra-domain* performance metrics. Yet, similarly to Lossradar, INT requires control over multiple VPs (one per destination). Dapper [31] detects performance problems using one VP but requires bidirectional traffic, which is unrealistic considering asymmetric routing. Sketches [42, 45, 46, 48, 63, 65] offer aggregate estimates for packet/flow counts and size distributions, but do not capture latency and loss across routes. Finally, Marple [49] could be used to implement performance monitoring. Yet, implementing two levels of aggregating (per flow and & per prefix and next hop) is not straightforward. Even assuming that is possible, such a solution would not run in today's programmable switches and does not provide flexible rerouting.

2.1 Design constraints

The following constraints drive RouteScout's design: :

- **R1 Respect routing policies:** By default, RouteScout must select amongst equally-preferred routes, replacing arbitrary tie-breaks in BGP, and hot-potato routing.
- **R2 Ensure correctness and stability:** RouteScout must prevent loops and oscillatory behavior.
- **R3 Deployability:** RouteScout should not require any coordination between ASes. A single AS deploying RouteScout should also benefit from it without upgrading its entire network.
- **R4 Support asymmetric routing:** Due to asymmetric routing, a RouteScout switch may not see both directions of traffic, it must, therefore, be able to estimate and improve performance from one-way traffic.

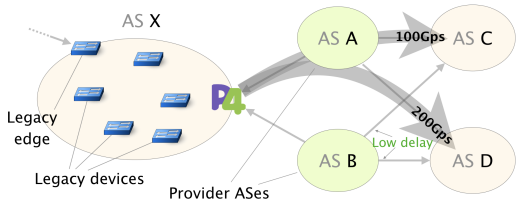


Figure 1: *ASA* and *ASB* are providers for the other three ASes. *ASX* has several legacy switches and a RouteScout-capable switch; not all edge switches in *ASX* run RouteScout; no coordination among RouteScout-capable switches and/or legacy switches is required.

- **R5 Respect flow affinity:** To avoid performance degradation due to reordering of packets that could result from sending packets of the same flow across different paths, RouteScout must enforce flow-path affinity.
- **R6 Fit today's switches:** RouteScout should fit within the scarce memory (dozens of MB at best [40]), restricted operations set (e.g., no floating points) and parallel memory accesses available to existing programmable network hardware.
- **R7 Limit bandwidth usage:** RouteScout must limit bandwidth usage between the data and control planes, regardless of the traffic rate and burstiness.

3 OVERVIEW

RouteScout is a closed-loop control system that dynamically adapts how a stub AS forwards its outgoing traffic across multiple policy-compliant routes according to observed performance and operator's objectives.

We illustrate RouteScout operations on a simple running example (Fig. 1) in which a stub network, *ASX*, routes traffic to multiple destinations, among which are *ASC* and *ASD*. *ASX* knows two equally-preferred paths to reach both destinations through its providers, *ASA* and *ASB*, with whom *ASX* has 250 Gbps links. BGP's arbitrary tie-breaking selects *ASA* as the next-hop for traffic to *ASC* and *ASB* for traffic to *ASD*. Unbeknownst to *ASX*, the path via *ASB* has a much lower delay to *ASC* and a slightly lower delay to *ASD*. Only one (edge) devices of *ASX* is programmable (**R3**).

Inputs To use RouteScout, the operator first specifies the **prefixes** of interest[4], together with their typical traffic **demands**.[5] In our example, *ASX*'s operator wants RouteScout to optimize for destinations *ASC* and *ASD*, which drive 100 and 200 Gbps of traffic respectively. Then, the operator specifies her **objectives** which in our example are to (a) minimize the delay to both destinations; and (b) load balance traffic across the next-hops, as long as the delay is not increased by >10%. Note that RouteScout automatically learns the policy-compliant next-hops from BGP (**R1**). RouteScout runs independently on a single edge device [6] and does not need to coordinate with other devices in or outside *ASX*.

[4] few hundreds accounting for most of the traffic volume [27, 53]
[5] adequately accurate estimates, are easy to obtain §6.1.
[6] or multiple if there are multiple edges

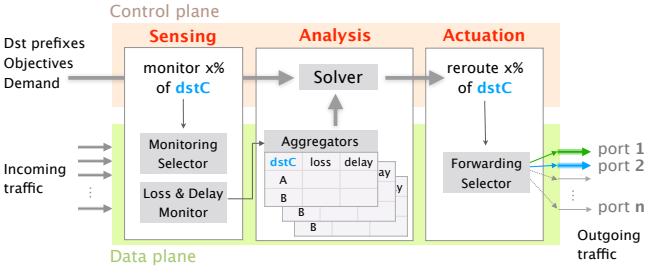


Figure 2: RouteScout is a closed-loop control system with sensing, analysis, actuation split across data and control planes.

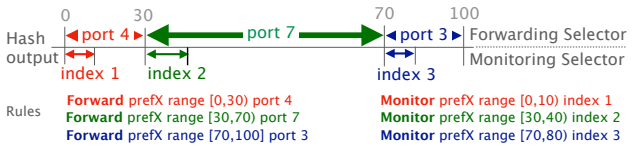


Figure 3: RouteScout uses two match-action tables to flexibly forward the traffic according to the control-plane decisions and flexibly monitor a given fraction of traffic per next hop.

System To satisfy the operator's objectives, RouteScout implements a control loop which...

- ... directs traffic to alternative next-hops
- ... monitors performance across prefix-nexthop pairs
- ... computes an optimized traffic allocation to next-hops
- ... actuates appropriate traffic shifts in the data plane

RouteScout splits the above functions across its control- and data-planes (Fig 2). The data plane collects and aggregates measurements for the control plane to analyze (**sensing**). The control plane decides which traffic to monitor and which traffic to reroute to which next-hops (**analysis**). The data-plane receives and enforces these decisions (**actuation**). Sensing and actuation operate at the granularity of a "slot", which we define as a small amount of traffic to a particular prefix. The number of slots pertaining to each prefix is determined by the proportion of its traffic volume. Operating at a per-slot granularity provides measurement efficiency, improved stability and better resource utilization. For instance, slot-based routing enables RouteScout to use paths that can not support all the traffic for a given prefix. Coming back to our example, *ASD* receives twice the traffic as *ASC*. Assuming a total of 3,000 slots, RouteScout allocates 1,000 slots to *ASC*, and 2,000 slots to *ASD*, with each slot carrying around 0.1 Gbps of traffic.

Data plane: RouteScout data plane enforces the per-slot monitoring and forwarding decisions made by the control plane. To scalably monitor effectively satisfying **R6**, RouteScout exploits TCP's semantics together with probabilistic data structures to analyze the relevant packets, aggregate the measurements (**R7**), and actuate the corresponding forwarding decisions (§4). Note that, while RouteScout relies on TCP, it only requires some TCP flows to exist per prefix, meaning it can still be useful even in QUIC-dominated Internet. To flexibly forward, RouteScout uses two match-action tables and a novel memory mapping scheme (§4.1), that allows it to seamlessly adapt to BGP updates, prefix or policy changes, consistently satisfying **R1**.

In our example, RouteScout reroutes 1 slot of traffic to each destination via the alternative next-hop, namely *ASB* (as decided by the control plane) and monitors 4 slots one for each destination, next-hop pair. As a result, aggregated loss and delay measurements for each pair will be available to the control plane.

Control plane: RouteScout control plane pulls aggregated data-plane measurements and computes a new forwarding state based on these and the operator objectives (§6.2) by formulating and solving a linear optimization program(§6.2).

The main challenge in computing a new forwarding state is the conflicting objectives that the operators often have. In our example, the operator wants low delay (primary) and balanced load (secondary). These cannot be satisfied together as *ASB* offers lower delay for both destinations. This is a deliberately simple example: since performance for *ASC* improves more, *ASD*'s traffic should be load balanced. But the problem becomes more complex as the number of prefixes, next-hops, and objectives grows.

RouteScout moves to the computed forwarding state on a slot-by-slot basis while tracking and reactive any performance degradation to avoid heavily congesting remote bottlenecks potentially violating **R2**. Slot-by-slot traffic shifts also reduce the risk of oscillations, even when multiple RouteScout systems co-exist by adding randomness and therefore avoiding synchronization [30].

4 ROUTESCOUT DATA PLANE

RouteScout's data plane uses compact data structures and efficient algorithms to flexibly forward traffic (§4.1) and accurately measure delay (§4.2) and loss (§4.3). We also discuss the impact of adversarial inputs and defenses (§4.4).

4.1 Selector stage

The *Selector* enforces the forwarding and monitoring decisions communicated by the control plane (§3) on a per-prefix basis. The forwarding decisions correspond to the number of slots to forward to given next hops, while the monitoring decisions correspond to the number of slots to collect statistics for on given next hops.

The *Selector* implements slot-based forwarding and monitoring by first hashing each incoming packet to a range $[0, k]$ and then using two match-action tables to identify sub-ranges $[i, j)$ of of the range $[0, k]$ that need to be monitored or forwarded to a given port. The two tables, *forwarding Selector* and *monitoring Selector*, use the same type of keys composed of: (i) a prefix; and (ii) a range $[i, j)$ which identifies a subset of traffic. In the *forwarding Selector* table, each key maps to a next-hop. In the *monitoring Selector* table, each key maps to the index of a memory block of a table (*aggregator* (§4.2-4.3)) in which the corresponding aggregated statistics will be

stored. By adapting the contents of each table, the controller can flexibly adapt the forwarding and monitoring behavior.

Example: Fig. 3 shows an example with a hash range of 0-100, and three rules in each table. The rules are such that, in expectation, 30% of packets (subrange 0–30) to prefix 'prefX' will be forwarded to port 4. Additionally, 1/3 of these packets (subrange 0–10) will be monitored before being forwarded, with the monitoring results stored in index 1 of the *aggregator*. Observe that the flexible design of the *monitoring Selector* table allows seamless adaptation to the system's dynamics. For example, if the BGP peer at port 4 withdraws prefX, then the range of the green (second) rule in the *forwarding Selector* could be expanded to include hash outputs 0-30, and the red (first) rules in both the *forwarding Selector* and *monitoring Selector* will be deleted. The index 1 of the *aggregator* used to store measurements for this prefix-next-hop pair can also be reset and assigned to another one.

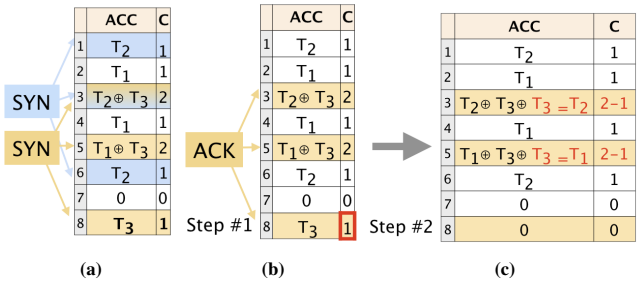


Figure 4: *Delay monitor*:(a) SYNs of different flows (blue/above & yellow/below) increment different indexes; (b) The first ACK of the yellow flow checks that all its indexes (3,5,8) are set, and reads the timestamp of the yellow SYN from the reversible index 8; (c) The same ACK removes the footprint of the yellow flow by XOR-ing T3 to the indexes of (3,5,8), and decrementing their counters.

4.2 Measuring delays

This component is responsible for accurately and scalably measuring the delay of any flow belonging to one of the monitoring slots enforced by the *Selector*. It relies upon a *monitor* and an *aggregator*. The *monitor* estimates the delay observed by each flow by tracking specific TCP metadata,[7] while the *aggregator* accumulates these statistics, which are eventually pulled by the control plane.

Estimating delay: To estimate the delay of a given flow in the presence of asymmetric routing, the *Delay monitor* computes the time elapsed between its TCP SYN and the first ACK (similarly to [39]). While doing so means that RouteScout only measures delay at connection setup, it also minimizes the noise from application-level effects, which are likely to be more significant for later packets. Moreover, using only SYN and ACK packets allows the *Delay monitor* to operate with unidirectional traffic, thus accounting for asymmetric routing (unlike prior work [20]). Observe that mirroring SYN and ACK packets to a software component is unrealistically costly, as *all ACKs* would need to be mirrored and matched to SYNs. Moreover, measuring the delay between SYN and SYNACK would require bidirectional traffic effectively violating **R4**.

Recording timestamps at scale is challenging. Indeed, simply storing the SYN timestamp and the 5-tuple in a hash table does not scale since it requires >100 bits per measurement. To address this problem, we use a combination of two probabilistic data structures: an *Accumulator*, for storing sums of timestamps at each index, and a *Counter* for counting how many timestamps are in each sum in the *Accumulator*. In essence, the *Counter* can be seen as a Counting Bloom Filter [26], while the *Accumulator* is similar to an Invertible Bloom Lookup Table [33]. We use XOR ($\oplus$) as the sum operator rather than a simple addition — while both + and $\oplus$ are recoverable (given A and $A \oplus B$ or $A + B$, one can recover B), $\oplus$ cannot cause overflows. Unlike previous works [45, 66] that send their full Bloom filters to the controller to be decoded (incurring both compute and bandwidth expense), we measure entirely in the data plane and only expose aggregated statistics to the control plane, which can pull them asynchronously.[8]

Example, Fig. 4: As SYNs of different flows arrive (Fig. 4a), we hash their 5-tuples with multiple hash functions, thus generating multiple indexes. Here the yellow (lower) flow is hashed to $(3, 5, 8)$, and the blue (upper) flow to $(1, 3, 6)$. Each entry of the *Accumulator* in those indexes is $\oplus$-ed with the timestamp of the SYN. Additionally, the *Counter* of each entry is incremented. Different SYNs can end updating the same index, *e.g.,* index 3 in Fig. 4a.

On receiving an ACK, we first compute the corresponding indexes using the same hash functions. If all the corresponding *Counter* values are non-zero, then we know that the SYN timestamp is contained in the *Counter*. In Fig. 4b, the ACK of the yellow flow arrives and finds its indexes set. To get the timestamp of its corresponding SYN, we need to find one index among the indexes to which the ACK is hashed, whose value in the *Counter* is one. We will call this index reversible. The same index in the *Accumulator* yields the timestamp for this flow's SYN, thus allowing us to compute its delay. In Fig. 4b, the ACK finds a value equal to 1 in the index 8, namely the third of the three indexes it is hashed to. Thus, the timestamp of the SYN is at index 8 in the *Accumulator*.

To erase the footprint of a SYN from the *Delay monitor*, we decrement each of the hashed indexes in the *Counter*, and $\oplus$ the recovered timestamp with the sums at these indexes in the *Accumulator*. In Fig. 4c, we illustrate the result of this process; observe that by $\oplus$-ing the timestamp in each of the hashed indexes, the effect of the yellow SYN vanishes.

Keeping the *Delay monitor* healthy: In the common case, the *Delay monitor* stores some per-flow state only during the handshake as an ACK removes the memory footprint created by the corresponding SYN. This allows the *Delay monitor* to scale with the number of flows regardless of their rate and duration. Still, a large number of SYNs not followed by corresponding ACKs can pollute the *Delay monitor*. This challenge can be easily addressed by keeping track of the number of SYNs in the *Delay monitor* and not add new ones if the filter has exceeded its capacity (number of elements it can store

[7] While RouteScout relies on TCP, it only requires some TCP flows to exist per prefix for measuring the path's performance. Yet, RouteScout's decisions will also benefit QUIC/UDP traffic.

[8] Also observe that lossradar [66] cannot measure loss from a single VP as we explain in see § 2

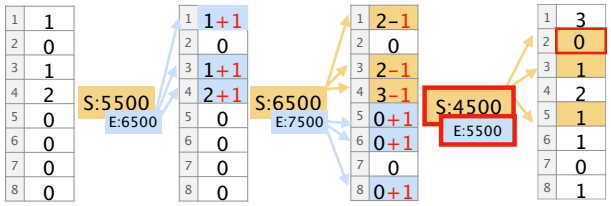


Figure 5: Here the *Loss monitor* sees three packet arrivals, 2 in-order and 1 retransmit. The first, with sequence number S:5500 has the next expected sequence number E:6500, and inserts the latter into the CBF by incrementing the indexes corresponding to the E:6500 (blue indexes, 1, 3, and 4). The second packets finds its indexes (now yellow, 1, 3, and 4) non-zero, thus knows it was expected. It cleans itself out, and inserts the next expected packet (blue indexes, lower) . The third one, a retransmit, finds one of its indexes (2) unset.

based on allocated memory, §7.2). Alternatively, the filter can be reset periodically.

Aggregating statistics: The *aggregator* stores the delay measurements per prefix-next-hop pair in an array with two values per index: one for storing the sum of the delays and one for storing the number of delay measurements contained in the former. The control plane can pull the measurements for a prefix-next-hop pair or for all pairs at once and calculate the mean delay. For example, in Fig. 4c, once the ACK has read the timestamp of its SYN it calculates the time elapsed since then and updates the values in the index that is mapped to its prefix and output port. The mapping between the prefix-next-hop pair and the index in the *aggregator* is assigned by the control plane and communicated via the *monitoring Selector*. Thus, to monitor different prefixes or a different number of next hops for some prefixes, one just changes this mapping instead of re-allocating memory and needing recompilation (see example in §4.1).

4.3 Measuring loss rates

The design and challenges of the loss measurement component are similar to those for the delay, with some key distinctions. In particular, to measure the loss rate, the *monitor* tracks the number of retransmitted and regular packets, while the *aggregator* accumulates the counts for each category. Similar to the *Delay monitor*, the *monitor* only needs to observe one direction of each monitored flow and only a few TCP flows to monitor.

Estimating loss rate: Measuring retransmissions at scale is challenging since one cannot simply store every packet and compare new arrivals against the history to identify duplicates. Our solution, somewhat surprisingly, requires only a few bits per flow at the cost of one minor compromise: the inability to distinguish reordering from retransmissions. Given that reordering also hurts TCP [15], mistakenly accounting for it as loss is not a significant downside if it is one at all.

Our solution keeps only one element per flow by exploiting TCP semantics and the fact that, given a TCP packet p, one can compute the *next* expected sequence number based on p sequence number and payload length. By storing this expected sequence number, we can check whether the next packet is either a retransmitted or an out-of-order packet. Instead of storing a 32-bit (expected) sequence number, e, we can insert it into a counting bloom filter (CBF), *i.e.,* the same data structure as our *Counter* for delay estimation. Since packets across flows can share sequence numbers, we insert the concatenation of the 5-tuple with the sequence number instead. Increasing the length of the inserted value is immaterial, as the length of the hash output is the same.

Whenever a packet with sequence number s arrives, we check the CBF for <5-tuple, s>: If the entry does not exist, the packet is out-of-order or a retransmit. If the entry exists, the packet is in order and we delete it from our filter by decrementing all the indexes <5-tuple, s> hashes to. We then insert the *next* expected packet by incrementing all the indexes that <5-tuple, $s + tcp.len$> hashes to.

Not all packets carry information regarding previous segments. For instance, an ACK that does not carry any TCP data will be followed by a packet of the same sequence number regardless of whether the former was lost or not. Similarly, KEEPALIVE messages (commonly used in Web traffic) contain an "unexpected" sequence number: one byte less than the previously sent sequence number. To avoid these issues, we only use packets with TCP payload. This does not disrupt functionality, as for every non-zero-payload packet whose subsequent sequence number we store, there will be a non-zero-payload packet that can remove it, even if it comes after multiple zero-payload ACKS.

Example, Fig. 5: In this example, we illustrate how 3 packets (the last one being a retransmitted one) of a flow update the CBF. The yellow (upper) box contains their sequence number, and the blue box (lower), the sequence number of the expected packet. The first packet inserts the fingerprint of the expected (second) one by incrementing the values stored in the indexes that the expected sequence number (concatenated with the 5-tuple of the flow) hashes to (blue indexes). Thus, when the second packet arrives, it will find all hashed indexes of its sequence number set (yellow indexes) and consider itself expected. This is not true for the third packet, whose indexes are not all set and is a retransmit.

Keeping the *monitor* healthy: Similar to the *Delay monitor*, the *Loss monitor* contains one item per flow regardless of its rate as the structure "cleans itself" with incoming packets. In particular, once a flow terminates, the corresponding RST or a FIN removes the flow permanently. Still, out-of-order and lost packets will, in most cases, cause some packets to stay in the filter. However, this represents a very small fraction of packets, as we discuss in §7.3. To avoid overflowing the *monitor*, a counter in the data plane can keep track of the number of flows using it. If the filter's capacity is exceeded, insertions are stalled until some of the flows terminate. Alternatively, the filter can be reset periodically, as we show in §7.3.

Aggregating statistics: Similarly to the *Delay aggregator*, the *aggregator* stores the number of expected and unexpected packets observed per prefix and next hop.

4.4 Dealing with adversarial inputs

Like any data-driven system, RouteScout is prone to attacks in which malicious endpoints or networks aim at faking signals in order to influence its decisions. While possible and deserving a complete

analysis in follow-up work, we briefly argue why such attacks on RouteScout are hard to perform.

In order to influence RouteScout's decisions, a malicious endpoint could try to: (i) send repeated packets to fake retransmissions; (ii) send fake pairs of SYNs and ACKs with small/large timing differences to fool the delay monitor or (iii) send fake FIN or RST packets to prevent the loss monitor from measuring loss rates of certain flows. We note two things. First, such adversarial endpoints must be hosted within the stub AS, since RouteScout optimizes exit traffic. Assuming basic anti-spoofing techniques are in place (e.g. [56]), each endpoint has a single IP address to source traffic from. As such, limiting the number of flows tracked per IP would be sufficient to mitigate the attack. Second, RouteScout randomly associates a flow to a next hop, depending on a hash function. As such, the attacker is equally likely to add noise to measurements of *all* next hops, making targeting one next-hop difficult. RouteScout can also defeat attempts to use traceroutes for probing such decisions by randomly forwarding traceroutes to next hops.

Similarly, a malicious transit network can: (i) drop packets to increase the loss rate; or (ii) drop/delay SYNs, SYN/ACK, or ACKs to fool the delay monitor. While this is possible, we note that, by doing so, malicious networks can only make their performance worse, not better. As such, malicious networks can only push away traffic, not attract more. Observe that an attacker cannot craft a SYN/ACK packet for every SYN it receives to fake low latency as she does not know the sequence number that the receiver will use until the actual SYN/ACK packet is received.

Finally, attackers can also attempt to pollute RouteScout's data structures. An efficient way to mitigate such pollution is to periodically reset the data structures, as we discuss in §7.

Fault tolerance In case of a data-plane failure RouteScout will rebuild the monitoring state (kept in the monitors and aggregators) from scratch and will retrieve the forwarding state (kept in the *Selector*) from the control plane. Thus, during the rebuilt RouteScout will not be able to respond to new performance opportunities but will instead use its previous decisions.

5 HARDWARE DESIGN

Our design needs modification to fit a real Protocol Independent Switch Architecture (PISA) switch. We briefly explain the key constraints imposed by PISA and how we adapted the Delay and Loss monitors accordingly. Our design takes up 10 physical pipeline stages , and we have fully implemented it in a Barefoot Tofino Wedge 100BF-32X.

PISA constraints: A packet traversing a PISA switch goes through a pipeline of stages. Besides the limited memory and instruction set, which our design already addresses, there are constraints on the sequence of memory accesses [13, 62]. First, a packet cannot read or write multiple memory addresses in the same memory block. Second, memory blocks are tied to a single stage in the pipeline and can only be accessed in it. This is to avoid contention from stages processing different packets simultaneously. Similarly, accessing stages in a different order or multiple times per packet is not possible.

Delay Monitor modifications: To access any Bloom Filter, including those in the Delay Monitor, we need to access multiple indexes, each corresponding to the output of a hash. For instance, in Fig. 4a,

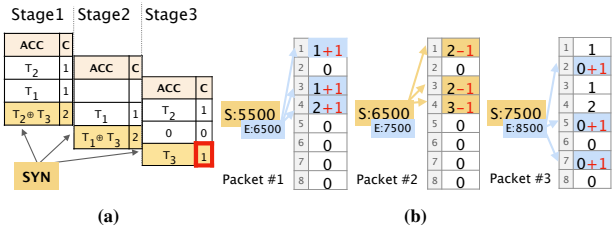


Figure 6: (a) We implement the *Delay monitor* as a series of arrays; (b) A packet can either check if it is expected or insert the next expected packet in the *Loss monitor*.

the yellow SYN would need to access three indexes corresponding to the yellow indexes. In PISA, though, one cannot concurrently access multiple indexes of the same memory block. We thus divide the two tables of the *monitor* into smaller chunks and constrain each hash to index a single chunk as seen in Fig. 6a. Now, chunks reside in different stages of the pipeline and can be accessed serially.

Serializing accesses creates another issue. Particularly, when an ACK arrives, the *monitor* first needs to find out if it corresponds to the first ACK of a flow whose SYN is in the *Accumulator* (Fig. 4b), and if so, decrement all corresponding indexes in the *Counter*. For this, the SYN will need to traverse all three pipeline stages in Fig. 6a to check whether all corresponding indexes of the *Counter* are non-zero. But after doing so, the packet cannot return to stage 1 and decrease their values in the *Counter*. To address this, the monitor recirculates packets corresponding to the first ACKs. Observe that even if we could rely on SYNACK, which is impractical due to asymmetric routing, we would still not be able to avoid recirculation. Indeed, even if an incoming ACK knew upon arrival that the timestamp of the corresponding SYN is in the structure, it will still need to find a reversible index to read this timestamp and then $\oplus$ it to all (previous) stages. As an illustration, in Fig. 6a, the reversible index is in stage 3. When the packet reads it, it can no longer return to stages 1 and 2, and $\oplus$ it to the corresponding indexes.

Loss Monitor modifications: Similarly here we need to split the CBF into multiple chunks and stages. Recall that every incoming packet needs to check if it is expected, remove itself, and insert the next expected packet in the CBF. This results in two violations of the PISA constraints.

First, a packet needs to access each memory chunk (in each stage) in two different indexes, one corresponding to the output of itself, whose value it needs to decrement, and one corresponding to the next expected packet, whose value it needs to increase. Second, the former access is conditioned on whether the packet is expected or retransmission, something which will only be known after the packet traversed all stages.

To address the first violation, we allow each packet one of the two operations, either to remove itself if it is expected or to insert the next expected one iteratively. To achieve this, we keep track of the number of packets seen by each flow. Particularly, when a packet arrives, it checks the number of non-zero-payload packets its flow has already sent. If this number is even, as for S:5500 and S:7500 in Fig. 6b, then the packet will insert the next expected one in the CBF. If the number is odd, as for S:6500 in Fig. 6b, the packet will

try to find its footprint in the CBF and remove it. We use a counting bloom filter to keep track of the number of packets efficiently.

To address the second violation, we assume all packets to be expected and recirculate packets that violate this assumption. In more detail, on arrival, a packet whose flow has sent an odd number of packets reads and decrements the indexes corresponding to it in the CBF. If the packet was indeed expected, *i.e.,* all read values are non-zero (as for S:6500 in Fig. 6b), the packet increments the *Accumulator* and leaves the device. If the packet was retransmission, it is recirculated to re-increment the indexes it wrongly decremented.

6 ROUTESCOUT CONTROL PLANE

In this section, we describe ROUTESCOUT's control plane and how it leverages measurements from the data plane to improve forwarding decisions. This is a challenging problem as due to bandwidth limitations, load-balancing preferences, or stability concerns traffic cannot always be forwarded to the most performant route. We start by describing the control-plane inputs (§6.1). We then explain how it solves the induced optimization problem (§6.2).

We describe the simplest version of the control plane that would enable performance-driven routing and support conflicting operator objectives. To cover additional operational needs, this control plane can be extended, for instance, to strengthen stability guarantees as shown in [30].

6.1 Inputs

ROUTESCOUT triggers the *Solver* periodically giving as input a description of the environment, a set of objectives, and optionally, some additional constraints for each prefix, together with fresh performance statistics.

Environment: The network environment includes topological, traffic, and routing information. The former two are provided by the operator and the latter by BGP. Topological information corresponds to the set of direct next-hops and their link capacities. Traffic information consists of the set of prefixes that ROUTESCOUT should optimize for, together with the volumes they drive. Routing information corresponds to the set of next-hops that ROUTESCOUT can use to route each prefix (obtained from routing tables and BGP policies).

Expecting traffic information is reasonable as important prefixes are few and stable over time [27, 53]. The traffic volumes to these prefixes can also be estimated accurately [38, 52]. Note that inaccurate traffic volumes won't affect ROUTESCOUT's performance if the direct links are not running at full capacity, which is true in most stub ISPs. If that's not the case, ROUTESCOUT might indeed not find the optimal solution but will never deteriorate the performance by moving traffic to a worse next hop.

Objectives: The operator can decide for each destination prefix whether they want to: (i) optimize for the delay and/or loss; (ii) minimize the number of traffic shifts necessary to meet the requirements; or (iii) load-balance traffic by minimizing the difference between the most- and the least-used next-hop. Linear combinations of these or similar other objectives are easily implementable.

ROUTESCOUT also allows multiple objectives to be flexibly implemented. To do so, the operator needs to express how important each objective is by defining priorities and how valuable are the differences among alternative forwarding states by defining tolerance levels. Objectives with lower priority will only be optimized if there are multiple equally-preferred solutions, namely solutions that differ from the optimal by no more than the tolerance level. For example, an operator might want to balance the load across the next-hops, as long as the delay difference between the best- and the used next-hop is lower than 10%. The operator can communicate this to ROUTESCOUT by assigning a high priority to delay with 10% tolerance and a lower priority to load-balancing.

Operational constraints: ROUTESCOUT admits constraints of two types: (i) those that limit the number of next-hops traffic can be spread on; and (ii) those that define performance constraints. Constraining the maximum number of next-hops per destination might be useful, for instance, to ease debugging. Performance constraints are maximum loss/delay values that traffic for a certain destination should experience. Defining such objectives is useful for meeting Service Level Agreements (SLAs) or particular application requirements.

Data plane statistics: ROUTESCOUT periodically pulls measurements of loss and delay aggregated per prefix and next-hop from the respective *aggregators*.

6.2 Solver

The solver is responsible for synthesizing a forwarding state. To do so, it formulates each of the operator's inputs into a constraint or an objective, creating a linear optimization problem. Since some variables are integer, *e.g.,* number of slots per prefix, our problem is an Integer Linear Problem.

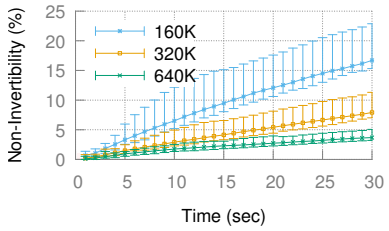
Problem statement: Let N be a set of next-hops and P_r the set of destination prefixes to optimize for. Let $P_a \subseteq P_r \times N$ be the set of all pairs of destinations and equally-preferred next-hops (learned by BGP). The goal is to find a mapping $F_t : P_a \rightarrow \mathbb{N}$, namely the number of slots allocated to each pair (prefix, next-hop) at time t such that it optimizes the operator's objectives while adhering to the environmental and operational constraints. We implement the *Solver* using Gurobi [3].

7 EVALUATION

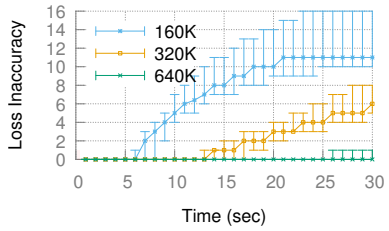
We evaluate ROUTESCOUT's *Delay monitor* (§7.2), *Loss monitor* (§7.3) and *Solver* (§7.4). For the monitors, we investigate the trade-off between accuracy and memory footprint using real traffic traces and our practical hardware design (§5). We find that, with 1 MB of memory, the *Delay monitor* can accurately measure the delay of hundreds of thousands of flows/sec. Moreover, the *Loss monitor* can accurately measure the loss rate of $36K$ flows/sec with as little as 312KB of memory. For the *Solver*, we focus on runtime and show that it computes forwarding states for thousands of destinations, across tens of next hops and for various objectives, in less than a second.

7.1 Methodology

To evaluate ROUTESCOUT's monitors, we estimate the memory they use as a function of their accuracy via both theoretical and practical means. For the theoretical analysis, we assume perfectly behaved TCP traffic (in-order, with expected semantics), with flow rates



(a) The probability of an ACK to decode its SYN's timestamp is >95% with a 1MB (640K elems) *Delay monitor* of 2 hashes.



(b) Using 625KB (640K elems) and 2 hashes, the *Loss monitor* calculates the loss rate with high accuracy.

Target	*50th*	*70th*	*95th*
# moves	0.02	0.03	0.29
balance	0.03	0.04	0.4
performance	0.08	0.3	1.09
combined	0.03	0.05	1.23

(c) Runtime percentiles in seconds depend on the complexity of the objective.

Figure 7

derived from real traces, and the original design as described in §4.2, §4.3 with 9 hash functions[9] and without any additional hardware limitations. For the practical analysis, we use real traffic traces and our hardware design for Tofino, with only 2 hash functions[10].

For the theoretical analysis, we use two different directions of CAIDA traces (CAIDA.A, CAIDA.B) collected at the Equinix-Chicago monitor in March 2018 [1], and one from MAWI [21] from January 2018. Together, these contain ~6 billion packets with an average rate ranging from 240-3200 Mbps. For the practical analysis, we use the CAIDA.A trace, which is the noisiest, and feed it to the monitors in 100 chunks of 30 seconds. While none of those traces are from a stub network, this has no impact on our analysis, as we are only interested in estimating accuracy and resource usage.

7.2 *Delay monitor*

Accuracy metric: We calculate the *invertibility*, namely the probability of a successfully computed delay. The delay between a SYN and its corresponding ACK can be successfully computed *if* upon arrival of the ACK, there is at least one index that contains *only* the timestamp of the SYN. Other than the memory used, invertibility depends on the number of concurrent delay measurements, the number of hash functions used, and the pollution of the structure due to traffic noise, *e.g.,* SYNs that are not followed by ACKs.

Theoretical analysis: In theory, invertibility is the inverse of the probability of false positive in a regular Bloom Filter: the probability of a SYN being ⊕ed to indexes that all contain other timestamps is the same as finding all hash outputs set in a regular Bloom Filter during a lookup. We calculate the memory requirements for an invertibility of 99.9% (false positive rate in BF of 0.1%) using the analytical formula for optimal Bloom Filter design [16]. For these calculations, we assume that each handshake completes in <1 sec, and that RouteScout needs to monitor all flows in each trace. The results are summarized in Table 1. The *Delay monitor* would need 12.9K-781.5K elements, corresponding to 6KB–381KB memory assuming an implementation over an array of 16-bit values using 9 hash functions.

Practical analysis: In practice, the filter is gradually polluted by SYNs that are not followed by ACKs. This can happen, *e.g.,* under SYN attacks, or when hosts try to reach an offline server. Such noise is common in our traces: in the noisiest trace (which we use for this evaluation), only 40% of the SYNs are followed by ACKs. Fig. 7a shows the median, max, and min non-invertibility probability as a function of time using {160K, 320K, 640K} elements in the data structure. As expected, the failure probability increases with time as the filter gets polluted. Still, RouteScout is very efficient. Indeed, a *Delay monitor* with only 320K elements has an invertibility of >90%. Another interesting insight is that we can do this with less memory if we periodically reset our *Delay monitor*, *e.g.,* with only 160K elements (312KB), we get the same >90% invertibility if we reset it every 15 seconds.

Trace	*SYNs/s*	*Elements*	*Delay M*	*Flows/s*	*Elements*	*Loss M*
CAIDA.A	3.8K	54.2K	26KB	36.8K	529.1K	1MB
CAIDA.B	54.4K	781.5K	381KB	233.8K	3361.3K	6MB
MAWI	899	12.9K	6KB	3.3K	47.8K	93KB

Table 1: *Delay monitor* and *Loss monitor* would combined need 6.4M to monitor as many flows/s as there are in the CAIDA.B trace.

7.3 *Loss monitor*

Accuracy metric: We compare the measured loss per flow to its actual loss rate. RouteScout's accuracy is affected by false positives: a retransmitted packet can be considered expected (instead of correctly being assessed as unexpected) and thus not counted towards loss, if all the indexes it hashes to are set. As the *Loss monitor* is a CBF, its false-positive rate depends on the memory and the number of hashes used.

Theoretical analysis: We use the same method as for the *Delay monitor*, to calculate memory requirements for achieving a false positive rate of <0.1%. The results are summarized in Table 1. The *Loss monitor* would need 47.8K–3.4M elements depending on the number of flows/sec in the trace. This corresponds to 93K–6M memory if the *Loss monitor* is implemented as an array of 4-bit values with 9 hash functions.

Practical analysis: In practice, the *Loss monitor*'s accuracy is deteriorated by three more factors. First, out-of-order packets are not only classified as losses but also pollute the structure as explained in §4.3. Second, flows terminating unexpectedly (*i.e.,* without FIN/RST) remain in the monitor until it is reset, decreasing its effective capacity.

[9] We chose 9 following the Bloom Filters heuristic [16].
[10] More engineering effort might allow implementation of more hashes.

Third, the *Loss monitor* can miss some loss events due to the compromise for PISA constraints: it only checks whether every other non-zero-payload packet has the right sequence number.

Despite these impairments, RouteScout is, in practice, very accurate. Fig. 7b shows the (max, min, and median across all runs) 70^{th} percentile of difference across all flows between their estimated loss rate and the ground truth reported by tshark. We plot 70^{th} as lower percentiles have zero error and thus unsuitable for studying the memory trade-off. We find that a *Loss monitor* with only 640K elements (625KB assuming 4bits/element) is almost perfect for 30 sec. Like the *Delay monitor*, resetting every 15 sec would allow smaller implementations to be similarly accurate.

7.4 *Solver* runtime

We investigate the influence of each parameter of the operational environment (§6.2) on the *Solver*'s runtime.

Methodology: We evaluate runtime: the time the *Solver* takes to compute a forwarding state; across several scenarios with different numbers of prefixes, next-hops, and slots. For each scenario, we run >5500 experiments with four different objectives: performance, balance across next-hops, minimal number of steps, and all of these combined. We fix all but one of the three parameters (*i.e.,* prefixes, next-hops, and slots) to default values. By default, we set the number of prefixes to 800 (corresponding to 80% of the traffic of CAIDA.A); the number of next-hops to 3, and the number of slots to be 200 (corresponding to a minimum traffic-shift granularity of 0.5% of the traffic per prefix). We report the median, 70^{th}, and 95^{th} percentile runtime as a function of each parameter in Fig. 8. We also group our experiments by objective and report median, 70^{th}, and 95^{th} percentile runtime in Table 7c.

Key results: Fig 8 shows that the 95^{th}-percentile runtime is 0.25 sec for 22 slots per prefix (left), 0.1 sec for 10 next-hops per prefix (center), and 0.05 sec for 2K prefixes (right). As Table 7c shows, the runtime also depends on the complexity of the objective. The most efficient objective to solve for is minimizing the number of shifted slots, while the least efficient one, unsurprisingly, is the combination of all objectives. In nearly all cases, the *Solver* finishes in under one second.

8 CASE STUDIES

We validate RouteScout's practicality and effectiveness in three steps. First, we prove that it is deployable by running it on a real testbed composed of Barefoot Tofino [5] switches. We then measure the benefits of running RouteScout for 10 stub ASes. Finally, we highlight the effectiveness of RouteScout in a larger testbed using $P4_{16}$.

8.1 Hardware testbed

We implement our hardware design (§5) on a Barefoot Tofino Wedge 100BF-32X in which a control process pulls statistics every 1 second and updates routing accordingly.

Our testbed (Fig. 9a) has two Tofinos ($SW1$ and $SW2$) and two servers ($s1$ and $s2$). $SW1$–$SW2$ are connected to each other with two links via ports 1 and 2, creating two $s1$–$s2$ paths. $SW1$ runs RouteScout and splits traffic to $s2$ across the two links. $SW2$ randomly drops a configurable portion of incoming packets matching on a specified ingress port.

We partition traffic to $s2$ into 16 slots. Thus, the minimum portion of traffic RouteScout can reroute/monitor is $1/16$ in this configuration. (More generally, anything from $\frac{1}{2} - \frac{1}{2^{32}}$ is feasible.) We assume the operator wants to minimize loss for traffic to $s2$. We also assume that the default next-hop for traffic to $s1$ is port 1, *i.e.,* the green (top) path. RouteScout thus routes most traffic $(15/16)$ on it, using one slot to probe the other path. We use 81 iperf [37] client-servers pairs to generate $s1 \rightarrow s2$ traffic. At time $t_1 = 7$ sec, we introduce 0.8% loss on the top path using $SW2$.

Fig. 9b and Fig. 9c show how the flow-count and traffic at each port evolve. Initially, port 1 sees 76 flows (4.3 Gbps) while port 2 sees only 5 flows (0.4 Gbps). At t_1, loss starts, and bandwidth across the green path drops as TCP reacts. This is quickly detected ($< 2sec$) by RouteScout, which installs new rules to shift almost all the traffic to port 2. RouteScout could be made faster by (for instance) increasing the polling rate for statistics. A pure data-plane system that forgoes a controller will, of course, be even faster but lose RouteScout's flexibility in terms of optimization goals and its stability.

8.2 Achievable gains in the wild

Quantifying the gains provided by RouteScout is challenging for three main reasons: *(i)* one needs to control egress routing of the tested stub AS; *(ii)* multiple stub ASes need to be tested for the results to be meaningful; *(iii)* running the full system using previously collected traces is problematic as the traffic is not responsive to RouteScout's operations (e.g. a lost packet will not be retransmitted).

To circumvent those limitations, we leverage *(i)* the RIPE ATLAS platform [4] which gives us access to multiple measurement probes in many stub ASes all over the world; and *(ii)* the fact that some stubs host multiple probes whose traffic exits via different next-hops ASes due to hot potato routing, therefore taking different paths.

In particular, we measure the delay difference among paths with the same pair of source-AS and destination IP but different first next-hop. We believe this measurement is a reasonable proxy for the RTT improvement achievable with RouteScout. Every 5 minutes,[11] we perform 2 concurrent traceroutes from 2 probes in the same AS, to each of the top-50 Alexa [6] destinations and report the difference in median delay observed by the two probes per pair of destination and 5-min interval iff they used a different next hop. We perform this experiment for 24 hours and repeat it for 10 stub ASes.[12] Fig. 10 shows the CDF of potential RTT improvement. Each line corresponds to a particular stub AS.

We find that 9/10 ASes could improve their RTTs in more than 35% of the cases by a 5–99% For 6 ASes, RTT would improve by more than 21% in at least 20% of the cases, while for 2 of them, RTT improvement would exceed 97%. Observe that the benefits shown in Fig. 10 can *only* be achieved by RouteScout but not by CDNs who can only optimize paths from their PoPs to users/stubs.

[11] The maximum probing frequency allowed by RIPE ATLAS.

[12] The selection of ASes was made such that there is at least one pair of probes a, b in ASX; which are geographically close to each other; and use different ASes, say $nextHopA$ and $nextHopB$ to reach the same destination prefix say p, which is among the 50 most popular Web destinations.

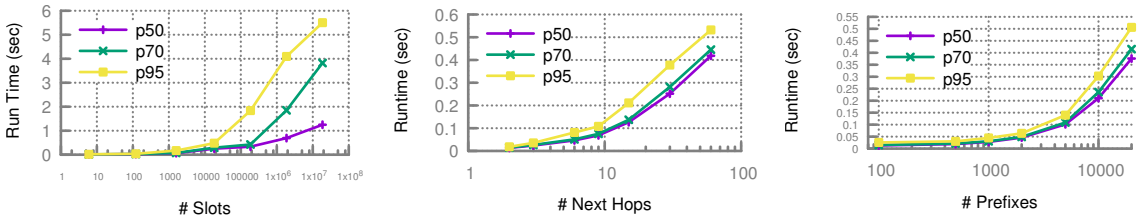
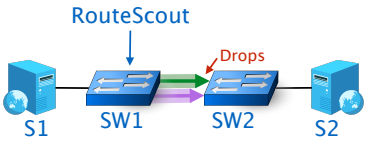
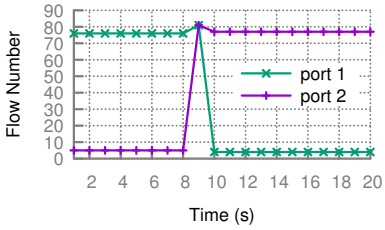


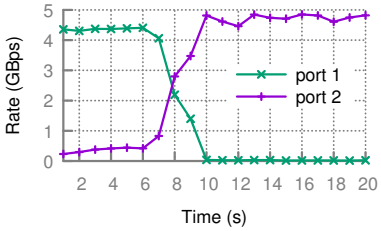
Figure 8: RouteScout is fast even when run with an increasing number of slots, next hops and destinations.



(a) Two Tofinos set up two $s1$-$s2$ paths. $SW1$ runs RouteScout and $SW2$ introduces loss in between the experiment.



(b) Number of flows routed via each alternative port changes after the increased loss is detected. Traffic shift takes <$2sec$.



(c) Bandwidth drop in port 1 is visible immediately after the loss is introduced and is clearer after RouteScout reroutes traffic.

Figure 9

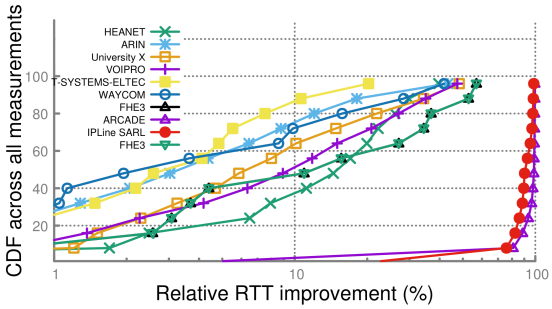


Figure 10: CDF of the relative RTT improvement each source AS should expect from delay-aware routing. 8 of the 10 ASes could improve the latency of at least 20% of the cases by 12–99%.

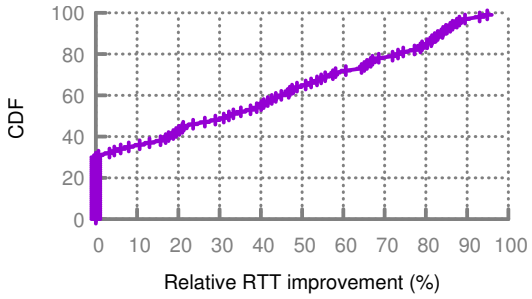


Figure 11: CDF of % delay improvement with RouteScout.

8.3 RouteScout in a network

We implement RouteScout in the P4 behavioral model (BMV2) [7] using ~900 lines of $P4_{16}$. We emulate a network scenario with a stub that runs RouteScout and 10 destination networks towards each of which it has 3 next-hops. The network scenario has 14 ASes, and 33 10 Mbps AS-to-AS links. The end-end delays are configured based on the latency differences observed in our RIPE experiments (§8.2). We assume that BGP has selected the first next hop for all prefixes. The goal of RouteScout's operator is to minimize the delay.

We use D-ITG [17] to create 10 TCP flows of constant rate to each of the destinations, resulting in 0.2 Mbps of aggregated traffic. We configure RouteScout to use 50 slots in total; as all prefixes drive the same traffic volume, each gets 5 slots. We run the experiment 10 times and report (Fig. 11) the CDF of improvement on the average end-end delay compared with the initial state. We see that RouteScout improves the delay in half of the cases by 32% or more.

9 CONCLUSION

RouteScout is a modern answer to the old problem of performance-aware Internet routing. Leveraging the capabilities of programmable switches, RouteScout continually and accurately monitors path performance at scale with low compute, memory, and bandwidth footprints. Based on these measurements, RouteScout control plane then reroutes traffic along policy-equivalent paths, fulfilling

the operators' objectives. ROUTESCOUT is BGP-compatible, deployable without coordination across ASes and without network-wide updates, improving Internet routing one switch at a time.

10 ACKNOWLEDGMENTS

We thank the NSG Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This work was supported by a Swiss National Science Foundation Grant ("Data-Driven Internet Routing", #200021- 175525).

REFERENCES

[1] [n.d.]. Caida Anonymized Internet Traces 2015. http://www.caida.org/data/passive/passive_2015_dataset.xml.
[2] [n.d.]. CAIDA Macroscopic Internet Topology Data Kit. https://www.caida.org/data/internet-topology-data-kit/.
[3] [n.d.]. Gurobi Solver. http://www.gurobi.com/.
[4] [n.d.]. RIPE NCC. RIPE Atlas. https://atlas.ripe.net.
[5] 2018. Barefoot. Barefoot Tofino, World's fastest P4-programmable Ether- net switch ASICs. https://barefootnetworks.com/products/brief-tofino/.
[6] 2018. The top 500 sites on the web. https://www.alexa.com/topsites.
[7] 2019. P4 behavioral model. https://github.com/p4lang/behavioral-model.
[8] Aditya Akella, Bruce Maggs, Srinivasan Seshan, and Anees Shaikh. 2008. On the performance benefits of multihoming route control. *IEEE/ACM Transactions on Networking (TON)* 16, 1 (2008), 91–104.
[9] Aditya Akella, Bruce Maggs, Srinivasan Seshan, Anees Shaikh, and Ramesh Sitaraman. 2003. A measurement-based analysis of multihoming. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 353–364.
[10] Aditya Akella, Srinivasan Seshan, and Anees Shaikh. 2004. Multihoming Performance Benefits: An Experimental Evaluation of Practical Enterprise Strategies.. In *USENIX Annual Technical Conference, General Track*. 113–126.
[11] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. 2002. Resilient overlay networks. *ACM SIGCOMM Computer Communication Review* 32, 1 (2002), 66–66.
[12] Todd Arnold, Matt Calder, Italo Cunha, Arpit Gupta, Harsha V. Madhyastha, Michael Schapira, and Ethan Katz-Bassett. 2019. Beating BGP is Harder than We Thought. In *ACM HotNets*.
[13] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (Sep 2018). https://doi.org/10.1109/icnp.2018.00047
[14] Debopam Bhattacherjee, Waqar Aqeel, Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. 2018. Gearing up for the 21st century space race. In *ACM HotNets*.
[15] Ethan Blanton and Mark Allman. 2002. On making TCP more robust to packet reordering. *ACM SIGCOMM Computer Communication Review* 32, 1 (2002), 20–30.
[16] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692
[17] Alessio Botta, Alberto Dainotti, and Antonio Pescapè. 2012. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks* 56, 15 (2012), 3531–3547.
[18] Alan Boyle. 2019. Amazon to offer broadband access from orbit with 3,236-satellite 'Project Kuiper' constellation. https://www.geekwire.com/2019/amazon-project-kuiper-broadband-satellite/.
[19] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. 2015. End-user mapping: Next generation request routing for content delivery. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 167–181.
[20] Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, and Jennifer Rexford. 2020. Measuring tcp round-trip time in the data plane. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*. 35–41.
[21] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. 2000. Traffic Data Repository at the WIDE Project. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (San Diego, California) *(ATEC '00)*. USENIX Association, Berkeley, CA, USA, 51–51. http://dl.acm.org/citation.cfm?id=1267724.1267775
[22] Cisco Performance Routing (PfR). [n.d.]. https://www.cisco.com/c/en/us/products/ios-nx-os-software/performance-routing-pfr/index.html.
[23] Benoit Claise. 2004. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational). http://www.ietf.org/rfc/rfc3954.txt.
[24] D. Clark, S. Bauer, K. Claffy, A. Dhamdhere, B. Huffaker, W. Lehr, and M. Luckie. 2014. Measurement and Analysis of Internet Interconnection and Congestion. In *Telecommunications Policy Research Conference (TPRC)*.
[25] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. 2001. MATE: MPLS adaptive traffic engineering. (2001).
[26] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.* 8, 3 (June 2000), 281–293. https://doi.org/10.1109/90.851975
[27] Wenjia Fang and Larry Peterson. 1999. Inter-AS traffic patterns and their implications. In *Seamless Interconnection for Universal Services. Global Telecommunications Conference. GLOBECOM'99.(Cat. No. 99CH37042)*, Vol. 3. IEEE, 1859–1868.
[28] Riot Games. 2016. Fixing the Internet for Real-time Applications. https://engineering.riotgames.com/news/fixing-internet-real-time-applications-part-ii.
[29] Lixin Gao and Jennifer Rexford. 2001. Stable internet routing without global coordination. *IEEE/ACM Trans. Netw.* 9 (December 2001), 681–692. Issue 6. https://doi.org/10.1109/90.974523
[30] Ruomei Gao, Constantinos Dovrolis, and Ellen W Zegura. 2006. Avoiding Oscillations Due to Intelligent Route Control Systems.. In *INFOCOM*.
[31] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data Plane Performance Diagnosis of TCP. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '17)*. ACM, New York, NY, USA, 61–74. https://doi.org/10.1145/3050220.3050228
[32] David K Goldenberg, Lili Qiuy, Haiyong Xie, Yang Richard Yang, and Yin Zhang. 2004. Optimizing cost and performance for multihoming. In *ACM SIGCOMM Computer Communication Review*, Vol. 34. ACM, 79–92.
[33] Michael T. Goodrich and Michael Mitzenmacher. 2011. Invertible Bloom Lookup Tables. *CoRR* abs/1101.2245 (2011). http://arxiv.org/abs/1101.2245
[34] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 161–176.
[35] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A Programmable System for Performance-aware Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 701–721. https://www.usenix.org/conference/nsdi20/presentation/hsu
[36] IP SLAs Configuration Guide. Cisco IOS. [n.d.]. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipsla/configuration/15-mt/sla-15-mt-book/sla_icmp_echo.html.
[37] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. [n.d.]. https://iperf.fr/.
[38] Muhammad Faisal Iqbal, Muhammad Zahid, Durdana Habib, and Lizy Kurian John. 2019. Efficient Prediction of Network Traffic for Real-Time Applications. *Journal of Computer Networks and Communications* 2019 (2019).
[39] Hao Jiang and Constantinos Dovrolis. 2002. Passive Estimation of TCP Round-trip Times. *SIGCOMM Comput. Commun. Rev.* 32, 3 (July 2002), 75–88. https://doi.org/10.1145/571697.571725
[40] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764
[41] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. 2005. Walking the tightrope: Responsive yet stable traffic engineering. In *ACM SIGCOMM Computer Communication Review*, Vol. 35. ACM, 253–264.
[42] Jorma Kilpi. 2008. IP-availability and SLA. In *Proc. of the International Euro-NF Workshop on Traffic Management and Traffic Engineering for the Future Internet*.
[43] Changhoon Kim, Anirudh Sivaraman, Naga Praveen Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band Network Telemetry via Programmable Dataplanes.
[44] Tobias Klenze, Giacomo Giuliari, Christos Pappas, Adrian Perrig, and David Basin. 2018. Networking in Heaven as on Earth. In *ACM HotNets*.
[45] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *NSDI*. USENIX Association, Santa Clara, CA, USA. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-yuliang
[46] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 101–114.
[47] Pascal Mérindol, Virginie Van den Schrieck, Benoit Donnet, Olivier Bonaventure, and Jean-Jacques Pansiot. 2009. Quantifying Ases Multiconnectivity Using Multicast Information. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement* (Chicago, Illinois, USA) *(IMC '09)*. Association for Computing Machinery, New York, NY, USA, 370–376. https://doi.org/10.1145/1644893.1644937
[48] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2014. DREAM: dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 419–430.

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

[50] Peter Phaal, Sonia Panchen, and Neil McKee. 2001. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176 (Informational). http://www.ietf.org/rfc/rfc3176.txt.

[51] Y. Rekhter, T. Li, and S. Hares. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard). http://www.ietf.org/rfc/rfc4271.txt

[52] Aimin Sang and San qi Li. 2000. A predictability analysis of network traffic. *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)* 1 (2000), 342–351 vol.1.

[53] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. 2012. Leveraging Zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review* 42, 1 (2012), 16–22.

[54] Stefan Savage, Thomas Anderson, Amit Aggarwal, David Becker, Neal Cardwell, Andy Collins, Eric Hoffman, John Snell, Amin Vahdat, Geoff Voelker, and John Zahorjan. 1999. Detour: Informed Internet routing and transport. *IEEE Micro* (1999).

[55] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. 2017. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *ACM SIGCOMM*.

[56] Stephanie AC Schuckers. 2002. Spoofing and anti-spoofing measures. *Information Security technical report* 7, 4 (2002), 56–62.

[57] SpaceX Starlink. [n.d.]. https://www.spacex.com/webcast.

[58] Neil Spring, Ratul Mahajan, and Thomas Anderson. 2003. The causes of path inflation. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 113–124.

[59] Hongsuda Tangmunarunkit, Ramesh Govindan, and Scott Shenker. 2001. Internet path inflation due to policy routing. In *ITCom 2001: International Symposium on the Convergence of IT and Communications*. International Society for Optics and Photonics, 188–195.

[60] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. 2018. Stroboscope: Declarative Network Monitoring on a Budget. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/nsdi18/presentation/tilmans

[61] Vytautas Valancius, Bharath Ravi, Nick Feamster, and Alex C Snoeren. 2013. Quantifying the benefits of joint content and network routing. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 41. ACM, 243–254.

[62] Chen Xiaoqi, Feibish Shir, Landau, Rexford Yaron, Koral andJ ennifer, and Rottenstreich Ori. 2018. Catching the Microburst Culprits with Snappy. https://www.cs.princeton.edu/~jrex/papers/snappy18.pdf.

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

[64] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2017. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *ACM SIGCOMM*.

[65] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 29–42.

[66] Li Yuliang, Miao Rui, Kim‡ Changhoon, and Yu Minlan. 2016. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *CoNEXT* (Irvine, California, USA). ACM, New York, NY, USA, 15 pages.

[67] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-Level Telemetry in Large Datacenter Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. ACM, New York, NY, USA, 479–491. https://doi.org/10.1145/2785956.2787483