Performance-Driven Internet Path Selection

Maria Apostolaki  
ETH Zürich

Ankit Singla  
ETH Zürich

Laurent Vanbever  
ETH Zürich

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.

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. 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, 59, 60]. Yet, despite several strong attempts [8–11, 54, 62], few practical progresses have 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, 65]. 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% [65]. Espresso explicitly pins these gains on being able to dynamically respond to performance variability across paths (rather than just average-case improvement from an 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 [27]. The importance of tail latency as opposed to mean latency is also demonstrated on 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 simply did not exist at the time of BGP’s first design iteration (1989). Further, if plans for satellite-based global Internet connectivity [18, 58] come to fruition, the performance gap across different paths will also increase. Two teams of researchers have separately argued in recent position papers [14, 43] 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.

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. 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 they are 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 nexthops for a large number of 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 [29].

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 conflicting decision-making 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;\(^2\) the majority of stubs are multi-homed and virtually all Internet traffic and 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 [46]. 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.

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.

\(^2\)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 [28].

### 2 Motivation

Performance-aware routing is an old problem [8,10,54,59,60], with several known solutions of varying ambition and complexity. Early work [31] 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 [40] and MATE [24] focused on intra-domain routing, splitting traffic across already setup tunnels. We would instead like to 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 the case for ROUTESCOUT.

**Active probing:** One can actively probe routes [21, 35]. While this approach can be effective in the intra-domain setting, where recent work [34, 42] used specially crafted probes to monitor performance, it is insufficient for our inter-domain context as 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 [23]. Several systems propose to address some of these issues by collecting and combining measurements from end-users [49, 57]. However, requiring large numbers of cooperative users makes bootstrapping hard.

**Passive sampling:** Gathering statistics on live traffic is possible using sampling with sFlow [50] or NetFlow [22]. 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 obviously captures the requisite information, it does not scale and is inflexible. 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 [68] and Stroboscope [61]. 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 1000 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], we find that 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 [65] 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.

Sketches: Sketches [41, 44, 45, 47, 64, 66] offer aggregate estimates for packet/flow counts and size distributions, but do not capture latency and loss across routes.

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 the control necessary for performance-aware routing. Blink [33] uses such switches to detect packet retransmissions. However, Blink can only detect large outages on a single path, not congestion and latency differences across multiple paths. Dapper [30], Lossradar [67], and In-band Network Telemetry [42] provide performance metrics, such as lost packets and queuing delays, but require bidirectional traffic or/and external mechanisms to aggregate performance markings. While Marple [48] could potentially be used to implement performance monitoring, it does 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.

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 [39]), 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 burstness.

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.

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 operators 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 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. 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 (a) to minimize the delay to both destinations; and (b) to load balance traffic across the next-hops, as long as delay is not increased by >10%. Note that ROUTESCOUT automatically learns the policy-compliant next-hops from BGP (R1).

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 . . . actsuates 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). ROUTESCOUT sensing and actuation operates at the granularity of a “slot”, which we define as a small amount of

---

4few hundreds (in expectation) accounting for most of the traffic [26, 53]

5adequately accurate estimates, are easy to obtain §6.1.
Figure 2: ROUTEScout is a closed-loop control system with sensing, analysis, actuation split across data and control planes.

Traffic to a particular prefix. 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 measurement 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 (§9).

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 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 (§4.2.3)) 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 3 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 allow 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
we hash their 5-tuples with multiple hash functions, thus gen-

To address this problem, we use a combination of two proba-

This component is responsible for accurately and scalably

While doing so means that R

In Fig. 4c, we illustrate the result of this process; observe that by $\oplus$-ing the timestamp in each of the

Selector will be deleted. The index 1 of the aggregator used
to store measurements for this prefix-nexthop pair can also be
reset and assigned to another one.

4.2 Measuring delays

This component is responsible for accurately and scalably
measuring the delay of any flow belonging to one of the mon-
itoring slots enforced by the Selector. It relies on a monitor
and an aggregator. The monitor estimates the delay observed
by each flow by tracking specific TCP metadata, 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 [38]). 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.

Recording timestamps at scale is challenging. Indeed, sim-
ply 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 proba-
bilistic 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
[25], while the Accumulator is similar to an Invertible Bloom Lookup Table [32]. We use XOR ($\oplus$) as sum operator
rather than a simple addition — while both + and $\oplus$ are re-
coverable (given A and A $\oplus$ B or A + B, one can recover B), $\oplus$
cannot cause overflows. Unlike previous works [44, 67] 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.

Example, Fig. 4: As SYNs of different flows arrive (Fig. 4a),
we hash their 5-tuples with multiple hash functions, thus gen-
erating 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 correspond-
ing 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 based on allocated memory, §7.2). Alternatively, the
filter can be reset periodically.

Aggregating statistics: The aggregator stores the delay me-
asurements per prefix-nexthop 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-nexthop 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 update the values in the index that is mapped to
its prefix and output port. The mapping between the prefix-
nexthop pair and the index in the aggregator is assigned by
the control plane and communicated via the monitoring Se-
lector. Thus, to monitor different prefixes or 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).
We note two things. First, such adversarial end points must be hosted within the stub AS, since RouteSCout optimizes exit traffic. Assumining basic anti-spoofing techniques are in place (e.g. [56]), each end point has a single IP address to 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 retransmit 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 will 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: Similarly 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 end-points or networks aim at faking signals in order to influence its decisions. While possible, and deserving a complete analysis in a follow-up work, we briefly argue why such attacks on RouteSCout are hard to perform.

In order to influence RouteSCout’s decisions, a malicious end-point could try to: (i) send repeated packets to fake retransmissions; or (ii) send fake pairs of SYNs and ACKs with a small/large timing differences to fool the delay monitor. We note two things. First, such adversarial end points must be hosted within the stub AS, since RouteSCout optimizes exit traffic. Assuming basic anti-spoofing techniques are in place (e.g. [56]), each end point 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
A packet can either check if it is expected or insert the next expected packet in the Loss monitor.

Similarly, a malicious transit network can: (i) drop packets to increase the loss rate; or (ii) drop/delay SYN, 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.

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. We have tested our design 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 [13,63]. 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 kind of 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, 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 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 ⊕ it to all (previous) stages. As an illustration, in Fig. 6a, the reversible index is in stage 3. At the time the packet reads it, it can no longer return to stages 1 and 2, and ⊕ 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 a re-transmission 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, 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 efficiently keep track of the number of packets.

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 a 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. 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 conflict-avoiding states by defining tolerance levels. Objectives with lower priorities will only be optimized if there are multiple equally-preferred solutions, namely solutions that differ from the operator’s objectives. To cover additional operational needs, this control plane can be extended for instance to strengthen stability guarantees as shown in [29].

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 [26, 53]. The traffic volumes to these prefixes can also be estimated accurately [37, 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 whether they want to: (i) optimize for 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 giving 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.

Problem statement: Let \(N\) be a set of next-hops and \(P\) the set of destination prefixes to optimize for. Let \(P_\text{o} \subseteq P \times N\) be the set of all pairs of destinations and equally-preferred next-hops (learned by BGP). The goal is to find a mapping \(F_i : P_\text{o} \rightarrow N\), 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 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 derived from real traces, and the original design as described in §4.2, §4.3 with 9 hash functions 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.

---

6 We chose 9 following the Bloom Filters heuristic [16].
7 More engineering effort might allow an implementation of more hashes.
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 [20] 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., SYNIs 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 which 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 ROUTE'SCOUT 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 SYNIs 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 SYNIs are followed by ACKs. Fig. 7a shows the median, max,
Despite these impairments, ROUTEscout is in practice, very accurate. Fig. 7b shows the (max, min, and median across all runs) 70\textsuperscript{th} percentile of difference across all flows between their estimated loss rate and the ground truth reported by tshark. We plot 70\textsuperscript{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, i.e., 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 the minimum traffic-shift granularity of 0.5% of the traffic per prefix). We report the median, 70\textsuperscript{th}, and 95\textsuperscript{th} percentile runtime as a function of each parameter in Fig. 8. We also group our experiments by objective and report median, 70\textsuperscript{th}, and 95\textsuperscript{th} percentile runtime in Table 7c.

**Key results:** Fig 8 shows that the 95\textsuperscript{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\textsubscript{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 $\frac{1}{7} - \frac{1}{32}$ 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 [36] client-servers pairs to generate s1→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 (due to hot potato routing), and therefore take different paths.

In particular, we measure the delay difference among paths with same pair of source-AS and destination IP but different first nexthop. We believe this measurement is a reasonable proxy for the RTT improvement achievable with ROUTEscout. Every 5 minutes,\footnote{The maximum probing frequency allowed by RIPE ATLAS.} 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.\footnote{The selection of ASes was done 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 nextHop\textsubscript{a} and nextHop\textsubscript{b} to reach the same destination prefix say p, which is among the 50 most popular Web destinations.} Fig. 10 shows the CDF of potential RTT improvement. Each line corresponds to a particular stub AS.
We implement ROUTE\textsc{scout} (\textsc{route} \textsc{scout}) to each of the destinations, resulting in 0
- 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 ROUTE\textsc{scout}'s operator is to minimize delay.

We use D-ITG \cite{17} to create 10 TCP flows of constant rate for 2 of them, RTT improvement would exceed 97%. For 6 ASes, RTT would improve by more than 21% in at least 20% of the cases, while than 35% of the cases by a 5–99% For 6 ASes, RTT would improve the latency of at least 20% of the cases by 12–99%. Should expect from delay-aware routing. 8 of the 10 ASes could improve their RTTs in more

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%.

8.3 ROUTE\textsc{scout} in a network

We implement ROUTE\textsc{scout} in the P4 behavioral model (BMV2) \cite{7} using \sim 900 lines of P4\textsubscript{16}. We emulate a network scenario with a stub that runs ROUTE\textsc{scout} 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 ROUTE\textsc{scout}'s operator is to minimize delay.

We use D-ITG \cite{17} to create 10 TCP flows of constant rate to each of the destinations, resulting in 0.2 Mbps of aggregated traffic. We configure ROUTE\textsc{scout} 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 ROUTE\textsc{scout} improves the delay in half of the cases by 32% or more.

9 Conclusion

ROUTE\textsc{scout} is a modern answer to the old problem of performance-aware Internet routing. Leveraging the capabilities of programmable switches, ROUTE\textsc{scout} continually and accurately monitors paths performance at scale with low compute, memory, and bandwidth footprints. Based on these measurements, ROUTE\textsc{scout} control plane then reroute traffic along policy-equivalent paths, fulfilling the operators' objectives. ROUTE\textsc{scout} is BGP-compatible, deployable without coordination across ASes and without network-wide updates, improving Internet routing one switch at a time.
References

[1] Caida Anonymized Internet Traces 2015. http://www.caida.org/data/passive/passive_2015_dataset.xml.

[2] CAIDA Macroscopic Internet Topology Data Kit. https://www.caida.org/data/internet-topology-data-kit/.

[3] Gurobi Solver. http://www.gurobi.com/.

[4] RIPE NCC. RIPE Atlas. https://atlas.ripe.net.

[5] Barefoot. Barefoot Tofino, World’s fastest P4-programmable Ethernet switch ASICs., 2018. https://barefootnetworks.com/products/brief-tofino/.

[6] The top 500 sites on the web., 2018. https://www.alexa.com/topsites.

[7] P4 behavioral model., 2019. https://github.com/p4lang/behavioral-model.

[8] Aditya Akella, Bruce Maggs, Srinivasan Seshan, and Anees Shaikh. On the performance benefits of multihoming route control. *IEEE/ACM Transactions on Networking (TON)*, 16(1):91–104, 2008.

[9] Aditya Akella, Bruce Maggs, Srinivasan Seshan, Anees Shaikh, and Ramesh Sitaraman. A measurement-based analysis of multihoming. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 353–364. ACM, 2003.

[10] Aditya Akella, Srinivasan Seshan, and Anees Shaikh. Multihoming Performance Benefits: An Experimental Evaluation of Practical Enterprise Strategies. In *USENIX Annual Technical Conference, General Track*, pages 113–126, 2004.

[11] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. *ACM SIGCOMM Computer Communication Review*, 32(1):66–66, 2002.

[12] Todd Arnold, Matt Calder, Italo Cunha, Arpit Gupta, Harsha V. Madhyastha, Michael Schapira, and Ethan Katz-Bassett. Beating BGP is harder than we thought. In *ACM HotNets*, 2019.

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

[14] Debopam Bhattacherjee, Waqar Aqeel, Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Gearing up for the 21st century space race. In *ACM HotNets*, 2018.

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

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

[17] Alessio Botta, Alberto Dainotti, and Antonio Pescapè. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15):3531–3547, 2012.

[18] Alan Boyle. Amazon to offer broadband access from orbit with 3,236-satellite ‘Project Kuiper’ constellation. https://www.geekwire.com/2019/amazon-project-kuiper-broadband-satellite/, 2019.

[19] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. *ACM SIGCOMM Computer Communication Review*, 45(4):167–181, 2015.

[20] Kenjiro Cho, Koushiro Mitsuya, and Akira Kato. Traffic data repository at the wide project. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’00, pages 51–51, Berkeley, CA, USA, 2000. USENIX Association.

[21] Cisco Performance Routing (PfR). https://www.cisco.com/c/en/us/products/ios-nx-os-software/performance-routing-pfr/index.html.

[22] Benoit Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004. http://www.ietf.org/rfc/rfc3954.txt.

[23] D. Clark, S. Bauer, K. Claffy, A. Dhamdhere, B. Hufaker, W. Lehr, and M. Luckie. Measurement and Analysis of Internet Interconnection and Congestion. In *Telecommunications Policy Research Conference (TPRC)*, Sep 2014.

[24] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. Mpl: Mpls adaptive traffic engineering. 2001.

[25] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.
[26] Wenjia Fang and Larry Peterson. Inter-as traffic patterns and their implications. In Seamless Interconnection for Universal Services. Global Telecommunications Conference. GLOBECOM’99. (Cat. No. 99CH37042), volume 3, pages 1859–1868. IEEE, 1999.

[27] Riot Games. Fixing the internet for real-time applications, 2016. https://engineering.riotgames.com/news/fixed-internet-real-time-applications-part-ii.

[28] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. IEEE/ACM Trans. Netw., 9:681–692, December 2001.

[29] Ruomei Gao, Constantinos Dovrolis, and Ellen W Zegura. Avoiding oscillations due to intelligent route control systems. In INFOCOM, 2006.

[30] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In Proceedings of the Symposium on SDN Research, SOSR ’17, pages 61–74, New York, NY, USA, 2017. ACM.

[31] David K Goldenberg, Lili Qiuy, Haiyong Xie, Yang Richard Yang, and Yin Zhang. Optimizing cost and performance for multihoming. In ACM SIGCOMM Computer Communication Review, volume 34, pages 79–92. ACM, 2004.

[32] Michael T. Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. CoRR, abs/1101.2245, 2011.

[33] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} ’19), pages 161–176, 2019.

[34] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. Contra: A programmable system for performance-aware routing. arXiv preprint arXiv:1902.00849, 2019.

[35] IP SLAs Configuration Guide. Cisco IOS. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipsla/configuration/15-mt/sla-15-mt-book/sla_icmp_echo.html.

[36] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

[37] Muhammad Faisal Iqbal, Muhammad Zahid, Durdana Habib, and Lizy Kurian John. Efficient prediction of network traffic for real-time applications. Journal of Computer Networks and Communications, 2019, 2019.

[38] Hao Jiang and Constantinos Dovrolis. Passive estimation of tcp round-trip times. SIGCOMM Comput. Commun. Rev., 32(3):75–88, July 2002.

[39] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP ’17, pages 121–136, New York, NY, USA, 2017. ACM.

[40] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In ACM SIGCOMM Computer Communication Review, volume 35, pages 253–264. ACM, 2005.

[41] Jorma Kilpi. Ip-availability and sla. In Proc. of the International Euro-NF Workshop on Traffic Management and Traffic Engineering for the Future Internet, 2008.

[42] Changhoon Kim, Anirudh Sivaraman, Naga Praveen Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. 2015.

[43] Tobias Klenze, Giacomo Giuliani, Christos Pappas, Adrian Perrig, and David Basin. Networking in Heaven as on Earth. In ACM HotNets, 2018.

[44] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A Better NetFlow for Data Centers. In NSDI, Santa Clara, CA, USA, March 2016. USENIX Association.

[45] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In Proceedings of the 2016 ACM SIGCOMM Conference, pages 101–114. ACM, 2016.

[46] Pascal Mérindol, Virginie Van den Schriek, Benoit Donnet, Olivier Bonaventure, and Jean-Jacques Pansiot. Quantifying ases multiconnectivity using multicast information. In Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement, IMC ’09, page 370–376, New York, NY, USA, 2009. Association for Computing Machinery.

[47] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Dream: dynamic resource allocation for software-defined measurement. In ACM SIGCOMM Computer Communication Review, volume 44, pages 419–430. ACM, 2014.
[48] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98. ACM, 2017.

[49] Venkata N Padmanabhan, Sriram Ramabhadran, and Jitendra Padhye. Netprofiler: Profiling wide-area networks using peer cooperation. In *International Workshop on Peer-to-Peer Systems*, pages 80–92. Springer, 2005.

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

[51] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006.

[52] Aimin Sang and San qi Li. 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)*, 1:342–351 vol.1, 2000.

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

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

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

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

[57] Srinivasan Seshan, Mark Stemm, and Randy H Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, pages 1–13, 1997.

[58] SpaceX Starlink. [https://www.spacex.com/webcast](https://www.spacex.com/webcast).

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

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

[61] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. Stroboscope: Declarative network monitoring on a budget. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018. USENIX Association.

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

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

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

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

[66] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 29–42, 2013.
[67] Li Yuliang, Miao Rui, Kim‡ Changhoon, and Yu Min-lan. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In CoNEXT, New York, NY, USA, December 2016. ACM.

[68] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM ’15, pages 479–491, New York, NY, USA, 2015. ACM.