Declarative Network Path Queries

Srinivas Narayana

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

Adviser: Jennifer Rexford

May 2016

Abstract

Effective management of computer networks is crucial to ensure the availability and performance of "always online" Internet services that we depend on. Towards this goal, programmatic tools can remove slow and expensive human involvement in management. Recently, Software-Defined Networking (SDN) technology has eased programmatic *control* of networks, but there has been little attention on programmatic *measurement* of networks.

This thesis focuses on a broad class of measurement questions that analyze the flow of traffic along network paths. Today, network operators measure traffic flow by "synthesizing" multiple data streams—including updates to forwarding, traffic observations from packet samples, and changes in network topology. However, this approach has significant limitations: it makes measurements indirect for operators to express, and forces operators to make a difficult trade-off between measurement accuracy and overhead.

In this thesis, we approach network path measurement with two key principles: (1) Enable operators to specify the measurements they need in a declarative query language; and (2) Drive network measurement according to operator-specified queries. We realize these principles in three parts, as follows.

First, we present a declarative query language, that enables paths to be specified as regular expressions over predicates on packet locations and header values. The language also has SQL-like "groupby" constructs for aggregating results anywhere along a path. We show several realistic measurement queries corresponding to resource management, policy enforcement, and troubleshooting.

Second, we present a query run-time system that translates path queries into accurate measurement that runs on commodity switch hardware. The run-time first compiles queries into a deterministic finite automaton. The automaton's transition function is then partitioned, compiled into 'match-action' rules (that run on commodity hardware), and distributed over the switches. Storing the automaton state requires only a small amount of extra space (2-4 bytes) on packets.

Third, we present optimizations which address fundamental bottlenecks in compilation, caused by queries and forwarding policies requiring different actions on overlapping sets of packets. Experiments indicate that our run-time system can enable "interactive debugging," allowing an operator to compile multiple queries in a few seconds. Further, the generated switch rules fit comfortably in modern switch rule memories.

Acknowledgments

It is a position of privilege in one's life to be able to receive a doctorate, and I'm deeply thankful to several people to have gotten here.

My advisor, Jennifer Rexford, is one of the most kind and wise people I know. I've learned an immense amount both professionally and personally from her, on how to think, how to think about how to think, how to speak, how to write, how to abstract away irrelevant things, how to solve problems, how to break my own biases, and how to be nonchalant about it all. I will always cherish her support and mentorship through difficult times of my PhD research and life. Her versatility and speed in understanding and critiquing research—while being supportive of one's progress in a heartfelt way—constitute an art form, and she will be an inspiration to me always.

David Walker is a wonderful collaborator and mentor. Being exposed to his rigor in approaching problems, his concise and principled manner of speaking and writing, and his refreshingly free thought process on research problems, has been very edifying. I have learned much from his feedback on my writing, speaking and thinking, and I am very thankful to him for taking the time to impart his wisdom.

Sanjay Rao has been a great support through difficult times, providing very helpful guidance on research, future career paths, and "meta" thoughts on research methodologies. I often think of the discussions I have had with him, and his support to help me feel confident about my work. I am very thankful to him for being a reader on my thesis, and for his kind feedback.

I thank the other members of my thesis committee, Aarti Gupta and Nick Feamster, for their insightful feedback and discussions on related topics. I thank Nick for his comments on earlier drafts of my conference publication. My discussions with Aarti have been eye-opening. She has helped me understand the connections between my work and existing literature, and her class lectures got me interested in the very exciting area of formal verification that I was unfamiliar with. I appreciate her warmth and patience in answering my questions, and in discussing performance optimizations for my prototype.

I thank Mung Chiang, who was an early mentor and a very big influence on my research, as well as Moses Charikar for his brain-warping algorithms lectures. Mung and Moses gave me one of the greatest technical gifts I have received: the ability to put intuition before detail, whether it comes to thinking, understanding, or communicating. I also thank Michael Freedman who imparted many insights during discussions in the early days of my grad school.

Nick Turk-Browne (Princeton Psychology), Kai Li (Princeton CS), Sushant Jain (Google), Alok Kumar (Google), Narasimhan Venkataramaiah (Microsoft), Arunkumar Navasivasakthivelsamy (Microsoft) and

Navendu Jain (Microsoft) have all been influential in shaping my thought process through the years, whether at Princeton or during my internships at Google and Microsoft. I appreciate their time.

I am very fortunate to have close friends whom I have looked up to as mentors. Vimal Jeyakumar has been a great friend and support since my undergraduate days. I find his taste in research topics, scintillating thoughts, ideas and discussions, and sheer tenacity all very inspiring. He has been an invaluable bouncing board for my ideas and papers through these years. Anirudh Badam was a great mentor and role model through the early years at Princeton, and I will cherish the good times and discussions we shared. Laurent Vanbever's amazing work ethic and openness to new ideas was very inspiring, and I'm grateful for his good-natured and laid-back encouragement.

Sometimes one conversation can change your life. I had such a conversation with Emilie Danna at Google, who observed that intelligence in network algorithms often becomes unnecessary when high quality input data is available. I've reflected a lot on her observation since, and much of the technical motivation for this thesis came from a demand for precise inputs for network algorithms. I'm much grateful to her for so generously offering a seed of wisdom that blossomed into this thesis.

I'm really happy to have had some amazing collaborators through the years. Mina Tahmasbi is a hard-working collaborator and a sympathetic friend. She has been a lot of fun to discuss with, learn from, and work with. Her cheerfulness and love for dogs have brought much lightness and laughter to my days in Princeton. I thank Divjyot Sethi for helping me co-author my first ever peer-reviewed publication, resulting from just one summer's work together. I cherish our long and heartfelt conversations about life and research, and our love for food. I learned a lot from him: persisting with one's goals, the value of good ideas and good execution, and to never take good collaborators for granted. I also thank Nate Foster and Arjun Guha for their generous help in merging the NetKAT compiler with the Pyretic run-time system, and Joshua Reich for discussions during my initial days of familiarization with Pyretic. I'm also grateful to (Joe) Wenjie Jiang, Vimal Jeyakumar, Changhoon Kim, Sharad Malik, Mohammad Alizadeh, Ragavendran Gopalakrishnan and Abdul Kabbani for their time and the opportunity to learn together. I'm indebted to George Varghese and Anirudh Sivaraman for insightful and friendly research discussions. I also thank Gordon Stewart who provided a basis for my first implementation of derivative-based regular expression parsing.

I've enjoyed working with my junior colleagues: Violeta Ilieva, Anshuman Mohan, Michael Chang and Vibhaalakshmi Sivaraman. It's perhaps unsurprising that we end up learning a lot from the people we try to teach. I have benefited from Violeta's efforts in building a network evaluation tool, Michael's Pyretic run-time enhancements that illuminated the challenges in building a robust controller system, Anshuman's efforts on the initial setup to collect NetFlow records in an SDN, and Vibhaa's tireless enthusiasm in questioning

and understanding networking measurement and hardware. I especially thank Vibhaa and Michael for the fun times, and for their friendships.

Through my time at Princeton, I've been fortunate to have some great groupmates and colleagues. These include Eric Keller, Minlan Yu, Cole Schlesinger, Peng Sun, Mojgan Ghasemi, Xin Jin, Naga Katta, Nanxi Kang, Kelvin Zou, Jennifer Gossels, Mina Tahmasbi, Laurent Vanbever, Joshua Reich, Ronaldo Ferreira, Yaron Koral, and Ori Rottenstreich. I thank these guys for making the office such a great place to work in, for providing great feedback on talks and paper drafts, and for some very enjoyable times at work and outside. I'd like to especially thank Laurent, Cole, Ronaldo, Yaron and Naga for some great discussions; and additionally Xin, Mojgan and Kelvin for very thoughtful feedback on paper drafts. Jiasi Chen is a good friend with whom I share a lot of common interests at and outside of work, and I thank her for her smiles, fun road trips, and thoughtful feedback on talks and drafts. It was very exciting to be officemates with Wyatt Lloyd and Jeff Terrace in my early days at Princeton. They taught me a lot about the grad school experience, and ways to learn while having fun. I also thank Arpit Gupta for interesting discussions and insights on performance improvement for my run-time system. I go back a long way with Jude Nelson, with whom I share fun times in class projects and wisdom on computer programming and hackery.

I'm very grateful to the CS office staff at Princeton: Nicki Gotsis, Mitra Kelly, Melissa Lawson and Nicole Wagenblast. I'm especially thankful to Nicki for her utmost generosity and warmth during times of stress, and to Mitra for her help in catering for the networking reading group and general fiscal matters.

I get by with a lot of help from my friends. These guys have made my grad school worth it. I appreciate Bharath Balasubramanian for his contagious laughter, offering the most balanced viewpoints about life I have encountered, and his resolve to keep fighting and pursuing the endeavors he loves. Jahnavi Punekar has been a wonderful friend whose maturity and sheer willpower have been an inspiration. Her chai company, philosophical conversations and artistic tastes have energized me quite a bit. I treasure both Jahnavi and Bharath for their selfless emotional support during dark times, and I hope to cling on to them for a long time. I'm also very grateful to Prakash Prabhu, Anand Ashok, Divjyot Sethi, Raghunandan Keshavan, Kumar Appaiah, Shruti Patil, and Pramod Jamkhedkar for being inspiring and uplifting company. I'm especially thankful to Poornima Padmanabhan for shining light through my last winter at Princeton, with her unfathomable enthusiasm and love.

I'm thankful to Soumyadeep Ghosh and Stimit Shah for fun times and for pushing my general knowledge. I'm grateful for the time I have had for deep discussions with Abhinav Narain, Ravi Tandon, Chinmay Khandekar, Sathish Thiyagarajan, Nayana Prasad, and Tejal Bhamre. I thank Nayana and Tejal for letting me squat at their home several evenings, as we socialized over dinner, chai, games, and work. I will miss their smiles and company much. I appreciate Yogesh Goyal and Sneha Rath for providing a wider perspective

to life during my time here, and Debajit Bhattacharya, Jahnavi, Tejal and Sneha for our musical journeys which I hold dear. Nick Peng has been graceful in showcasing the human trials and tribulations of life, and I will cherish being at his “friends and family” piano concert at Princeton. I’m grateful to Aditya Bhaskara, Aravindan Vijayaraghavan, Rajsekar Manokaran, Vivek Seshadri and Vikram Seshadri for their very special company, and to Naveen Sharma and Ramya Vinayak for being energetic waves of happiness that wash ashore my generally tranquil life.

I couldn’t have gotten where I am without the steady and unconditional support of my family. Despite being so far away, they managed to keep in touch and hear me out almost every other day, and provided much needed moral support through the grad school years. My mother, Vasantha, has taught me the quality of persistence through difficult times by example of her own life, while my father, Narayan, taught me not to take myself too seriously. Along with their kindness, they have given me the most enduring gifts for life. I also thank Balaji and Sukanya for their love and support. My times at home with Arjun and their dog Argus, are etched in my memory.

I treasure Smruthi, who has fused her days with mine, despite the distance. She opened my eyes to a world of beauty, understanding and gratefulness. I thank her for believing in me, and for trusting me to trust in her.

To my parents, for their boundless love.

Contents

Chapter 1

Introduction

> "What is the arcane lore that gives you your power?"
> Master Foo said, "I have none. Nothing is hidden, nothing is revealed."
>
> *Eric S. Raymond, Unix Koans*

The Internet services that we enjoy, like search, social networking, and video, are made possible by a complex infrastructure of compute servers and data networks. These services operate on large amounts of input data, forcing them to run on a network of distributed computers that must communicate effectively. The results—that we see—then arrive at our devices through a chain of Internet Service Providers.

Such data networks face some difficult requirements. Being called on to support a diverse, evolving set of applications, the networks should provide good performance, be always available for users, and be inexpensive to run. *Operators* of such networks should be able to manage the infrastructure proficiently. But humans are slow in understanding and controlling complex systems, and cost a lot of money when tasked with high service standards.

Bringing forth simplicity through high-level abstractions is a recurring and fundamental idea in computing. Good abstractions lend themselves to programmatic control, which allow a system to grow larger, adapt faster, and become simpler and predictable for humans to understand. Such abstractions have touched data networks in the form of *Software-Defined Networking (SDN)*, which provides a way to program a set of devices distributed throughout a network with a logically centralized view.

Yet it is often challenging to *observe* what happens in a network. For example, packets may be dropped at congested links disrupting application performance, distributed denial-of-service attacks may impact victims,

and malicious communications may pass without inspection during transient network conditions. Observing such things is difficult for operators today—even though the fixes are reasonably straightforward after.

What are some good abstractions to simplify measurement for network operators? How should such abstractions be translated to efficient measurements? These are the key technical concerns of this thesis.

1.1 Need for High Performance Networking

Communication networks today support many important and useful applications. As we discuss below, such applications seek certain desirable characteristics from the network: high throughput, low loss, low latency, high availability, predictability, fast adaptation and elastic growth in capacity.

For example, Internet search applications (*e.g.,* `www.google.com`) serve user queries by indexing the content of the World Wide Web. Search operates on large amounts of data that make it prohibitively expensive to build powerful machines that process the data in entirety. Instead, it is more economical to use off-the-shelf, "commodity" machines to process smaller chunks of this data, and *aggregate* their results to construct a response [9]. This architecture, typically termed *scale out* [37,107], leverages inherent parallelism in compute workloads to distribute the computation and storage to numerous inexpensive machines.

We show the typical architecture of such a scale-out application in Fig. 1.1. An incoming user request is received by one machine, which partitions the query to multiple workers, which may themselves partition it to other workers. The results of the workers' computation is subsequently aggregated to construct a response for a user. The application has a small time budget to return an overall response, since a large user-perceived latency turns away users, leading to loss of revenue [39, 56]. Hence, the partition-aggregate pattern imposes significant performance requirements on the interconnecting network: transfers of intermediate results should be fast and lossless, and data should be transmitted at high rates—allowing the system to spend as much time as possible to generate *higher quality responses* for users. Further, the network itself must be highly available to connect the machines—outages and resulting poor performance can result in significant costs and reputation damage [8].

As another example, Internet Service Providers (ISPs) like AT&T and Verizon serve video traffic to users. Video is currently the most significant chunk of Internet traffic—an estimated 64% of global Internet consumer traffic in 2014 and growing at a high rate [17]. User studies show that the percentage of time spent in buffering videos, and the average video bitrate, both have significant impact on user engagement [26]. Further, poor predictability of network conditions can lead to poor user experience [44]. Hence, both ISPs and the infrastructure hosting video content must ensure that the networks servicing video traffic provide high throughput, low loss, and low variance in network conditions over time (*i.e.*, predictability).

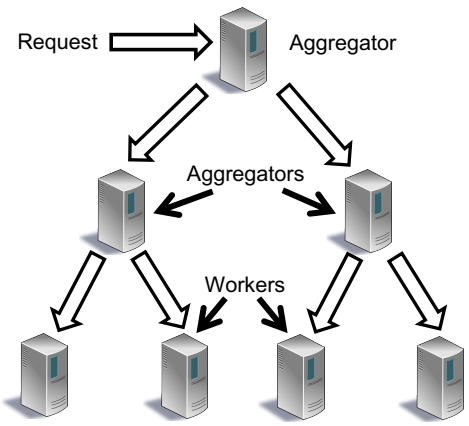


Figure 1.1: Architecture of a typical "scale out" application.

Computation on increasing data volumes in upcoming machine learning and scientific computing applications may similarly require high speed and low delay interconnecting networks. An emergent feature of such largely distributed applications is that often the overall performance and quality of the system depends on its *worst* performing components. Then, it is necessary to engineer networks to maintain such high operating standards—but it turns out that today this is very challenging.

1.2 Complexity of Network Management

What does operating a large network entail? At a high level, operators must configure servers and networking components, ensure that applications are highly available, monitor application performance and optimize the infrastructure to meet Service-Level Agreements (SLA). Operators must ensure that the infrastructure is utilized efficiently, troubleshoot it when things (inevitably) don't work, and ensure that it is secure. Managing today's networks is daunting.

Further, management practices must hit a moving target: both infrastructure and applications change often. New and heterogeneous devices are added all the time, and applications and their traffic patterns change. These changes force both application writers and infrastructure operators to regularly revise their assumptions about what the other can do. Outages often force operators to take actions that aren't normally planned (and hence not readily "scripted"), like routing traffic differently [79] or dropping traffic entirely [29].

Today, most networks are managed through extensive human involvement, which is too slow and expensive. As the scale of the network reaches a few hundreds of devices, it becomes untenable for such manual processes to scale: for example, during a network-wide change, an operator needs to compute the configuration of each network device, log into each device using a command-line interface (CLI) to perform the reconfiguration [18, 80, 117], while remembering the transitory states of the network (as she may only

configure a few devices at a time) and retaining context of changes that should be effected in future. Such *low-level* processes are complex, error-prone, and time-consuming for operators to execute, and there are both pragmatic and philosophical problems with this state of affairs.

The efficiency and costs at which network operators can maintain networks with current practice—especially to fix problems—leaves much to be desired. It can take *hours* for teams of operators to fix outages [36, 122] which impact businesses significantly. According to some studies, an unplanned network outage lasts for around 200 minutes, while costing $5600 per minute on average [8, 28]. An average large corporation experiences 87 hours of downtime every year! Recruiting more humans alone won't solve the problem, because that also costs organizations significant money: as of March 2016, a network administrator in San Jose, California is paid $68,000 per year[1] on average [94].

Further, the existing low-level interfaces to the network force operators to become "masters of complexity" [92]—it is highly indirect to express a desired global outcome by computing the configuration of each device separately. For instance, operators compute routing configurations to 'coax' a distributed protocol like Open Shortest Path First (OSPF) to achieve specific link utilization and resiliency goals [33, 34]. And operators must do so within the confines of the network protocols that deal with several issues irrelevant to those goals: to be concrete, the OSPF RFC [64] is over 200 pages long, with details on protocol message formats, primitives for state distribution and maintenance and so on, with only 20 pages discussing the construction of the routing table—which is the only thing the operator cares about at that moment!

Software-Defined Networking. Given the complexity of managing networks, there is a clear need to simplify it through *high-level abstractions* for operators and their networks. Recently, *Software-Defined Networking* (SDN [60]) technologies have sought to introduce good abstractions and software control into network management. SDN encompasses at least two key principles: (1) unify the control logic of the network (which is currently distributed among network devices) into a logically *centralized* entity called *the controller*; (2) *program* packet forwarding primitives implemented efficiently on switches using general purpose programs running on the controller. Such an architecture enables an operator to reason directly about the global state of the network—implementing complex network policies in software—and to programmatically push the desired switch-level behaviors into the network.

In today's switches the exposed forwarding model is often the so-called *match-action flow tables*, into which an SDN controller programs packet-processing *rules* (Fig. 1.2). Match-action is a model for switch lookup tables [12, 60]: the lookup table consists of several rules, each with three parts:

1. a match, which denotes a ternary or exact pattern specifying one or more packet header fields;

[1] With a 40 hour work week, this comes to about $32 an hour. Human time is about two orders of magnitude more expensive than compute time: for example, Amazon EC2's *largest* on-demand compute instance costs $0.532 per hour [6], as of March 2016.

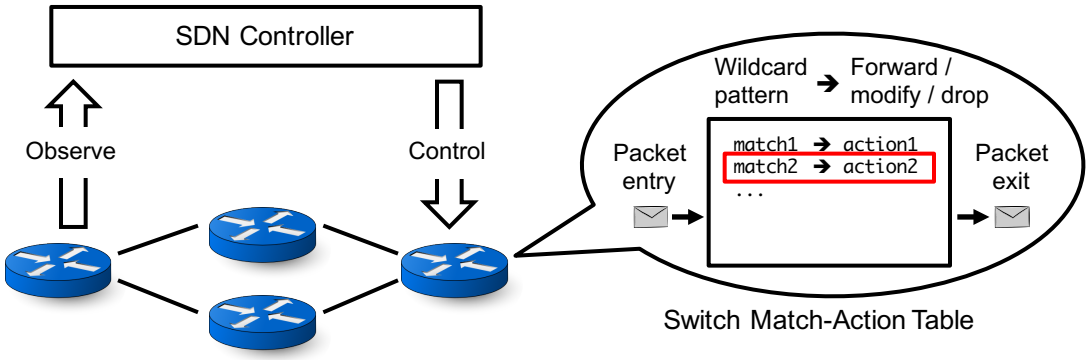


Figure 1.2: Architecture of a Software-Defined Network (SDN). An SDN controller programs a network of switches, each of which contains one or more flow tables with match-action rules. As packets come into a switch, they are matched against patterns in the flow table, and actions corresponding to the matching pattern are carried out on the packet.

2. a set of actions, which may include forwarding the packet out of one or more ports (possibly to an SDN controller), modifying packet header fields to a fixed value, or dropping the packet entirely; and
3. a priority, that disambiguates switch actions when more than one pattern match a packet. The actions of the matching higher priority rule are applied to the packet.

The match-action model is widely applicable to existing switch hardware—a number of popular protocols to programmatically control switches support it [71, 72, 75]. A switch could either have a single match-action table that processes packets, or several pipelined tables which a packet passes through one by one.

Research efforts in SDN have taken strides towards controlling networks with high-level programmatic abstractions, *e.g.,* [7, 35, 63, 67, 110]. However, as we discuss in the next section, knowing what's happening in the network can often be more challenging than controlling the network based on that knowledge. In other words, if we view network management as a *feedback loop of measurement and control* (Fig. 1.2), the control abstractions have come a long way, but the measurement abstractions have not.

1.3 Complexity of Network Measurement

Unfortunately, while network measurement is *possible* with tools existing today, is it not *easy.* For example, the network operator's staple measurement toolkit is well-suited to monitoring traffic at a single location (*e.g.,* SNMP/RMON, NetFlow, and wireshark), or probing an end-to-end path at a given time (*e.g.,* ping and traceroute). However, operators often need to ask questions involving packets that traverse specific *paths, over time*: for example, to measure the traffic matrix [32], to resolve congestion or a denial-of-service attack by determining the ingress locations directing traffic over a specific link [31, 90], to localize a faulty device by tracking how far packets get before being dropped, and to take corrective action when packets evade a scrubbing device (even if transiently).

Answering such questions requires measurement tools that can analyze packets based both on their *location* and *headers*, attributes which may change as the packets flow through the network. The key measurement challenge is that, *in general*, it is hard to determine a packet's upstream or downstream path or headers. Current approaches either require inferring flow statistics by "joining" traffic data with snapshots of the forwarding policy, or answer only a small set of predetermined questions, or collect much more data than necessary (§1.5). Below, we consider examples of measurement tasks that operators care about today, but find it challenging to get right.

1.3.1 Example: Debugging Packet Loss

Almost every network operator would have asked the question "where are packets getting dropped?" at some point in their professional life. Packet loss is problematic because it increases the completion time of short flows, which are crucial for scale-out applications (§1.1).

Consider the scenario in Fig. 1.3, where among 100 packets sent from A, only 80 are received at B. A network operator may wish to localize the switch or interface in the network which is dropping packets. One way to do so is to "follow" the path of the packets through the network, and determine where traffic disappears. However, this isn't simple. Traffic between A and B may be forwarded through multiple paths in the network shown, and an operator needs to observe traffic on multiple paths to know where the $A \rightarrow B$ traffic was dropped. But even that isn't sufficient: operators should understand how traffic *should have* flown through the network if everything was working correctly.

Standard packet capture tools like wireshark [112] enable operators to direct packets to the switch's control plane for inspection. However, using software packet-capture tools is restrictive: the switch's (software) control plane can typically only process packets at a much lower rate than its (hardware) data plane. There is also a deeper issue: since only a subset of traffic should be captured, an operator must write *filters* to match the traffic (*e.g.,* `srcip==A and dstip==B`) as shown in the figure. But those filters must change based on how traffic is forwarded: if a device somewhere in the network rewrites destination addresses of the packets to `B'`, then the filter must correspondingly reflect that.

Another option is to use switch *counters* that passively count the traffic going through the device. For example, an operator can install a rule that matches and counts the interesting traffic, *i.e.*, $A \rightarrow B$. While counters don't incur the same overheads as software packet capture, packets can typically only match one rule on every device (§1.2): hence if there is an existing rule that matches some overlapping traffic, *e.g.,* all B packets, then the operator must carefully tease out the old and new rules to ensure the intentions of all the rules are enacted correctly.

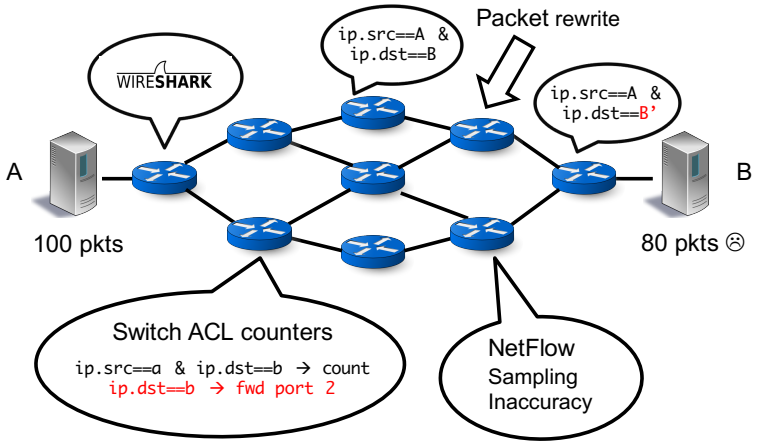


Figure 1.3: Debugging packet loss (§1.3.1) requires joining the forwarding policy with traffic observations in the network.

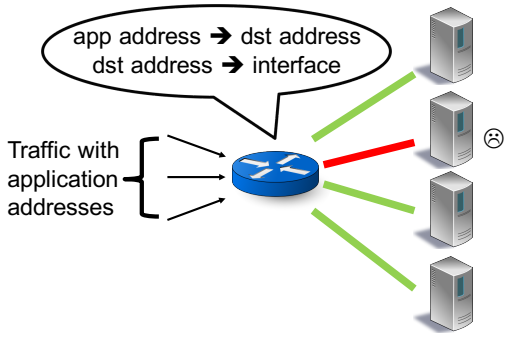


Figure 1.4: Diagnosing uneven link load requires relating incoming and outgoing traffic flows as packets are rewritten.

Yet another possibility is to *sample* from all the packets going through the device, using widely-deployed packet sampling technologies like NetFlow [1] and sFlow [3] on switches. Unfortunately, the samples might totally miss the packets that are interesting, since they are not specific to the traffic that the operator cares about at the moment.

Despite the fact that loss localization is challenging, note that the corrective action is relatively more straightforward: the operator could simply reroute traffic around the affected device or interface.

1.3.2 Example: Debugging Load Balancing

Operators often strive to balance conflicting goals of minimizing congestion and efficiently utilizing the network resources. One concrete example of such a tension is understanding why traffic load balancing actions on switches fail to keep link loads evenly balanced. The resulting imbalance may cause application servers to be overloaded or packets to be dropped due to link congestion.

Consider the situation in Fig. 1.4. A load balancing switch moves incoming request traffic to a server pool that hosts a set of applications. Let us suppose that the incoming client requests are sent to an externally visible *application address*, which is rewritten to one of the server IP addresses by the load balancing switch. A choice of a server also implies the selection of a switch interface to send the traffic, since servers are not all connected to one switch interface. The load balancing policy itself is typically determined by switch vendor implementations that use hash functions to keep a connection's packets going to the same server in the pool. Further, the policy may be dynamic and take network conditions into account, for *e.g.,* avoiding sending clients to failed servers, and balancing the load on the servers and the network.

Suppose an operator finds the second link (shown in red) overloaded relative to the other links in the system. There are at least two reasons why this could happen: there could be too many connections going to the same server, or a set of connections going to the respective server may be "heavy," contributing a lot of traffic.

However, disambiguating between the two possibilities itself is not easy. It would be ideal to have a table listing each incoming connection against its traffic volume, and the server and interface that was chosen to route it, but this is only possible by measuring load of *incoming* traffic from clients (*e.g.,* through switch counters, or samples), and relating it to the *outgoing* interface of the packets. Hence, the operator must relate incoming to outgoing traffic through the load balancing policy—which may be dynamic and unknown to the operator.[2] To have access to this policy, an operator may either use existing command-line interfaces to pull the forwarding information base (that maps incoming addresses to outgoing addresses), or measure the client traffic at the servers to know incoming addresses. Either way, there is no getting around joining forwarding policy with traffic measurements.

Once the operator knows the root cause, she could either rework the switch hash functions to balance client connections more evenly, or provision more machines in the server pool to handle heavy clients. Knowing what to do about the load peak is easy once we know why the load peaked in the first place.

Hence, existing network measurement tools force operators to choose one among two difficult options: either perform a complex and potentially inaccurate join between traffic and forwarding, or collect a lot of traffic everywhere—including traffic that isn't necessary to solve the problem—at high overhead. Both choices are difficult for network operators to accept.

[2]Vendors often do not expose hash functions used for load balancing, which is another issue in practice [64].

1.4 Goals

We believe that better abstractions for network measurement can significantly help operators: when operators want to measure *path-level* flows in an network, they should be able to specify concise, network-wide *declarative queries* that are

1. independent of the forwarding policy,
2. independent of other concurrent measurements, and
3. independent of the specifics of network hardware.

The measurements themselves should be carried out by a *run-time system*, that enables operators to

4. get accurate measurements directly, without having to "infer" results by joining multiple datasets,
5. have direct control over measurement overhead, and
6. use commodity match-action switch hardware [12, 60].

A *programmatic* and *high-level* interface would enable operators to focus on details *relevant* to the current management task without worrying about unnecessary details, as operators do today (§1.3). Further, a specification of operator *intent*—artfully constrained by such an interface—enables the implementation of highly accurate measurements at reasonable overheads. Software automation stemming from the use of such high-level, programmatic interfaces could enable faster adaptation of the infrastructure, faster growth, simplicity and predictable behavior of infrastructure [18]. Good abstraction and subsequent software control have already supported substantial growth in scale and complexity in the IT industry—which has developed mature software tools [15, 42, 57, 105] to make routine tasks much simpler, allowing operators to focus on strategic, revenue-generating, tasks.

By focussing on a broad class of questions, *i.e.*, path-level measurements, the *run-time system* can implement measurement techniques once and implement it well, allowing a diverse set of operator queries to reuse the same implementation. Next, we provide some background on the core technical problem that the run-time should solve: observing packet paths.

1.5 Background on Packet Path Measurement

How do we know which path a packet took through the network? How do we collect or count all packets going through a specific path? A number of prior approaches [1, 27, 40, 53, 54, 83, 90, 96, 102, 118, 122] aim to answer these questions. Table 1.1 provides a taxonomy around these approaches. There are at least three desirable characteristics for measurement techniques:

Approach	Expressiveness	Sources of inaccuracy	Sources of overhead
Policy checking (§1.5.1)			
Header space analysis [52, 53]	Locations and headers	No actual packets Only control plane view	Policy analysis
Out-of-band approaches (§1.5.2)			
Infer using traffic matrix [32, 119]	Switch-level paths	Forwarding dynamism Downstream packet drop Opaque multipath routing	Load collection [21] Traffic collection [1, 14, 87]
Upstream inference [53, 121]	Locations and headers	Ambiguous upstream path Packet modification	Traffic collection [1, 14, 87] Policy analysis
Join per-hop info [27, 40, 96, 122]	Locations and headers	Ambiguous packet joins	Packet digests (every hop) Topological sort
In-band approaches (§1.5.3)			
Record interfaces [83, 90]	Interface-level paths	Record few interfaces	Packet space for interfaces
Path tracing [102, 118]	Interface-level paths	Strong assumptions	Packet space for interfaces Data plane rules
Our approach (§1.6)			
DFA on packet state [65, 66]	Locations and headers	*None*	Packet space for DFA state Data plane rules Query compile time

Table 1.1: Qualitative comparison of approaches to collect path-level flow information (§1.5).

1. *High expressiveness*, to enable posing questions in a broad class, *e.g.,* to specify packet attributes including both locations and headers;
2. *High accuracy*, to be able to capture exactly the packets which traverse paths according to the operator question; and
3. *Low overhead*, to avoid overwhelming network resources such as data collection bandwidth, compute, switch rule space, and packet header space.

We begin surveying the literature with *policy analysis* (§1.5.1), an approach that serves as a building block to others that follow, by analyzing the forwarding policy to determine paths that packets *would* take (§1.5.1). Then we describe how it is possible to observe packets at various locations in the network and infer their trajectories *out of band* (§1.5.2). Finally, we discuss how packets can themselves carry information about their paths to enable switches to perform path measurement efficiently *in band* (§1.5.3).

1.5.1 Policy Analysis

What packets *would* take a specific path in a network, given the network's forwarding policy? Prior works answer specific instances of this question by analyzing the network's forwarding policy, and checking which packets satisfy the property of interest. For example, approaches like Header Space Analysis [52, 53] and VeriFlow [54] can determine if reachability constraints are met (*e.g.,* tenant A can talk to tenant B), if the network has loops (*i.e.*, packets are processed by the same switch more than once in their path, causing

bandwidth to be wasted), or if the network has blackholes (*i.e.*, packets passing through certain interfaces may be silently dropped), among other things. The algorithms used by these approaches efficiently evaluate the action of the network on packets using the *symbolic* version of their contents, instead of testing each *possible* packet one by one.

Could these approaches be used to understand which packets go through specific paths of interest? Unfortunately, requiring *accurate forwarding policies* as input data—typically obtained from control plane mechanisms—makes it challenging. Forwarding policies are typically extracted from dumps of the forwarding tables on switches (*e.g.,* Cisco configurations [53]) or from forwarding updates sent out by an SDN controller. This indirect source of truth about the network, *i.e.*, the control plane, hides visibility into at least three important aspects of the network's data plane, as we discuss below.

First, control plane policies don't consider the *actual packets in the data plane* that are processed by these policies. This is necessary to measure resource utilization, for example: an operator must know *how much* traffic corresponding to some demand goes through a congested link, in addition to the fact that the demand itself uses the link. Second, it is challenging to have a consistent view of the data plane's behavior at small time scales—for instance, due to fast failover mechanisms (*e.g.,* fast reroute [76]) or data plane load balancing mechanisms [5,51] that operate entirely in the data plane without control plane involvement. Finally, the data plane behavior may be unknown to the control plane, in situations like network congestion (*i.e.*, packets are dropped due to lack of capacity, even if the forwarding policy requires them to be processed) and data plane faults (*e.g.,* so-called "silent blackholes" [122]). Ultimately, policy-analysis approaches cannot directly track path-level flows in the data plane, since actual data-plane behavior can be unknown or unexpected.

1.5.2 Out-of-band Path Measurement

It is possible to collect observations of packets from multiple points in the network, and put those observations together to infer packet paths. This has the benefit of observing the traffic (unlike policy analysis approaches §1.5.1). The packet observations can come from independent per-packet sampling, *e.g.,* NetFlow [1], packet mirroring [14, 87], hash-based sampling [27], "postcards" or packet digests [40], or matching and mirroring specific packets [112,113,122]. After collecting packet observations, it is possible to run offline queries either by loading them onto a database, or by using purpose-built systems like Gigascope [23] or NetSight [40].

Broadly, such path measurement approaches can be divided into two sub-categories: those that join the forwarding policy with packet observations to infer paths, and those that directly join packet observations with each other. We describe these two methods separately, as they differ both in their workings and their inherent challenges.

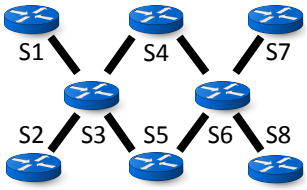


Figure 1.5: Inferring device-level flows using a traffic matrix. We can compute the flow through the path `S3 - S4 - S6` by adding the demands `S1` $\rightarrow$ `S8` and `S2` $\rightarrow$ `S7`, which route through the aforementioned path.

Joining Traffic Observations with Forwarding

One way to measure path-level flows is to capture packets as they enter the network, and estimate their path ahead using the forwarding policy of the network. Viewed in this way, such estimations naturally extend the policy-analysis approaches (§1.5.1) by including information about actual packets in the network.

How do we put together forwarding policies and packet observations to measure path-level traffic flows? One answer to this question—which is a special case of a more general approach—is to infer paths using a *switch-level traffic matrix*: a matrix of size $M \times N$ for a network with M ingress and N egress switches, and entry(i, j) of the matrix refers to the demand between ingress i and egress j. In a network whose switch-level paths for traffic is entirely determined by the combination of the ingress and egress devices,[3] operators can use this path measurement to estimate the traffic on *any* switch-level path in the network.

For this reason, several research works have studied the problem of estimating traffic matrices accurately [61, 108, 119, 120] from coarse-grained network data, such as flow-level samples (*i.e.*, NetFlow [1]) or aggregated link load counters (*i.e.*, by querying devices using Simple Network Management Protocol (SNMP)).

Given a switch-level traffic matrix, we can estimate the traffic on any switch-level path as follows. Consider the network shown in Fig. 1.5. Let's suppose that the network has ingress points $S1$ and $S2$, egress points $S7$ and $S8$, and that the forwarding policy routes demands between the ingresses and egresses as follows:

```
S1 -> S7: S1, S3, S5, S6, S7
S1 -> S8: S1, S3, S4, S6, S7
S2 -> S7: S1, S3, S4, S6, S7
S2 -> S8: S1, S3, S5, S6, S8
```

Now using this traffic matrix, it is possible to compute the volume of traffic on any switch-level path to a first approximation. First, we determine the ingress-egress demands which use a path of interest, say $S3 - S4 - S6$.

[3] Conventionally, transit ISP networks have had this model, since they use iBGP to pick an egress device for any transit traffic, and then use an intra-domain routing protocol like OSPF to pick the route within the ISP [32].

In this example, the corresponding demands are $S1-S8$ and $S2-S7$. Then, we just add the traffic volumes of those demands by looking up the corresponding entries of the traffic matrix.

A more general version of such traffic volume estimation uses the notion of "Forwarding Equivalence Classes" or FECs [52, 54], which are sets of packets that start out at a given ingress point, and are routed exactly in the same way at every hop on their path until they egress the network. A monitoring system may measure the traffic volume for every FEC at every ingress of the network, and use such an "FEC traffic matrix" for estimation (along with the forwarding policy).

Unfortunately, "joining" such traffic matrices with forwarding policies for path-level measurements is a process that is fraught, for the reasons we discuss below.

Pre-determined measurement granularity. The granularity at which a traffic matrix is measured, *e.g.,* switch-level, or FEC-level, immediately restricts the granularity at which questions about packets can be asked. In the example in Fig. 1.5, we couldn't ask for traffic volumes of packets through switches $S3-S4-S6$ *and* destination IP address `10.0.1.10`, simply because traffic volume isn't available at that granularity. It *is* possible to measure traffic volumes at finer granularities, for *e.g.,* an IP-subnet or IP address-level traffic matrix, but at high overheads. Yet, inference using finer-grained traffic matrices will still suffer from the rest of the problems we describe below.

Dynamic forwarding policies. Packet forwarding in a network changes often due to topology changes, failover mechanisms (*e.g.,* MPLS fast re-route [76]), and traffic engineering [5, 51]. Further, today's devices do not provide the timestamps at which the forwarding tables were updated, so it is difficult to reconcile packet-forwarding state with collected traffic data. However, such timestamp alignment would be really quite useful, especially to troubleshoot problems due to network forwarding changes, for example to decide whether a demand change or a routing change led to link congestion in the network [103].

Packets dropped in flight. It is tricky to estimate actual packet paths even when packet forwarding is static. Packets may be dropped downstream from where they are observed, *e.g.,* due to congestion or faulty equipment, so it is difficult to know if a packet actually *completed* its inferred downstream trajectory.

Opaque multi-path routing. Switch features like Equal Cost Multi-Path (ECMP) routing are currently implemented through hardware hash functions which are closed source and vendor-specific. This confounds techniques that attempt to infer downstream paths for packets. This is not a fundamental limitation (*e.g.,* some vendors may expose hash functions), but a pragmatic one.

Given the limitations of estimating packet paths ahead of the observation point, an alternative is observing traffic *further downstream* along packet paths—as opposed to the ingress—and estimating the packet paths *before* the observation point using the network's forwarding policy. For example, Header space analysis

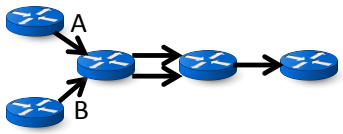


Figure 1.6: An example of ambiguous upstream paths: Identical packets arriving from A and B cannot be distinguished once they reach a common downstream point along the path. So, it is unclear whether the packet from A or B got dropped.

and SDN traceback [53, 121] provide recipes to estimate paths upstream from a given observation point. Unfortunately, they suffer from some significant fundamental limitations:

Ambiguous upstream path. Observations of traffic on internal links of interest cannot always tell where the traffic entered. For example, packets with identical header fields may arrive at multiple ingress points, *e.g.,* when packet headers are spoofed as in a DDoS attack, or when two ISPs peer at multiple points. Consider the scenario in Fig. 1.6: packets from devices A and B follow different paths eventually merging on the same set of downstream interfaces. Disambiguating them at that point is impossible.

Packets modified in flight. Compounding the difficulty, network devices may modify the header fields of packets in flight. For example, a Network Address Translation (NAT) device may rewrite the IP addresses of packets [69]. "Inverting" packet modifications to compute the upstream trajectory is inherently ambiguous, as the upstream packet could have contained arbitrary values on the rewritten fields. Computing all possibilities is computationally difficult [121]. Further, packet modifications thwart schemes like trajectory sampling [27] that hash on header fields to sample a packet at each hop on its path.

Joining Traffic Observations With Each Other

Is it possible to reconstruct any path-level traffic flow we desire by only collecting packet observations from the network? If we wish to avoid relying on the forwarding policy (as in §1.5.2), one option is to *collect packet observations at every hop.* Prior approaches such as NetSight [40], trajectory sampling [27] and hash-based IP traceback [96] collect digests of packets at every hop of their trajectory—with the former two sending the digests to collectors, and the latter storing them locally on the switch for a short while. By querying this (abstract) database of packet observations, it is possible to reconstruct packet paths, by assembling the observations of each unique packet from various switches in one place, and ordering the observations topologically [40] or using timestamps at which they were received [122]. There are still fundamental difficulties in getting accurate answers with this process, however.

Ambiguous packet "joins." By observing packets at every hop, the approaches mentioned above get around some of the inaccuracies of using the forwarding policy that we saw in §1.5.2; however, they cannot get

around all of them. Inaccuracies due to ambiguous upstream paths and packet modifications will still hamper inference of complete paths from per-hop packet observations (*e.g.,* see Fig. 1.6).

Determining paths from packet observations is akin to a *database join:* each switch represents one table of records which are packet observations, and the tables are joined on a key which is a combination of packet headers, corresponding to the *same packet* at different switches. The trouble is that the key, *i.e.*, set of packet header fields, is not fundamentally required to be unique at any given switch or across switches. Further, the header fields of the same packet may change from switch to switch! Examples of path ambiguities resulting from such joins have been noted in prior literature, for *e.g.,* see [27, Fig. 3], [96, Sec 2.2]. The join process may result in packet *trees* instead of packet paths, due to the ambiguities.

High data collection overhead. Further, running taps at every point in the network and collecting all traffic is infeasible due to the bandwidth and data storage overheads. Even targeted data collection using wireshark [112] or match-and-mirror solutions [113, 122] cannot sustain the bandwidth and storage overheads to collect all traffic affected by a problem. Sampling the packets at low rates [27] would make such overheads manageable, but at the expense of losing visibility into the (majority) unsampled traffic. Such lack of visibility may prove to be an unwanted obstacle to the diagnosis process when an operator is looking for specific traffic (*e.g.,* a small set of TCP connections) that the sampling missed.

1.5.3 In-band Path Measurement

What if switches could write all the necessary information about a packet's path into the packet itself? That would remove any ambiguity about the prior locations and header values of the packets, assuming that we trust the switches in the network.[4] Hence, it would be possible to "tag" packets with metadata that enable switches to *directly* identify packet paths [48, 55, 65, 90, 102, 118] in the data plane! Unfortunately, current approaches have some drawbacks as we discuss below.

Limited expressiveness. Approaches like IP record route [83], traceback [90] and path tracing [102, 118] can identify the network interfaces traversed by packets. They do so by attaching information at each hop corresponding to the interface that was just visited (for *e.g.,* an interface IP address). However, this is often insufficient: operators also care about packet *headers*, including modifications to header fields in flight—*e.g.,* to localize a switch that violates a network slice isolation property [53].

High packet overhead. Further, the accuracy and overhead of these approaches cannot be customized according to requirement: traceback can only accurately record a few waypoints, because of the limited packet

[4]This assumption is somewhat implicit in our setting: we look at networks within a single autonomous domain. Conceptually, one could still consider a model with untrusted devices within a single domain, but it is outside the scope of this thesis.

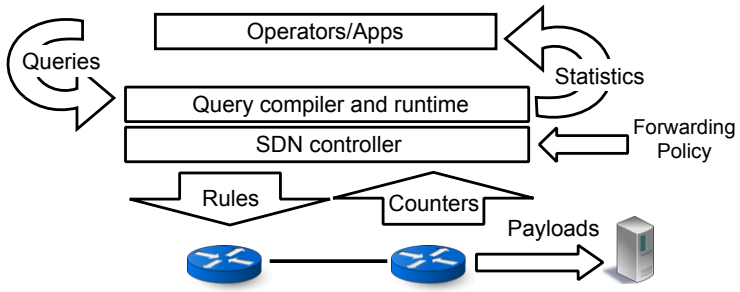


Figure 1.7: Path Query System.

space that is available to write this information. On the other hand, path tracing approaches use up packet space to record the entire path regardless of what the information is used for. For instance, operators may only care about very specific information, like whether packets traversed a firewall in the network; yet these schemes would use up packet space to record the entire path.

Strong assumptions about the operation of the network. Current approaches require that some strong assumptions hold in the network: *e.g.,* symmetric topology [102], no loops [102, 118], stable paths to a destination [90], or requiring that packets reach the end hosts [48, 55]. Such assumptions help these techniques compress necessary path information into limited per-packet space; however they also restrict the applicability of the schemes. Unfortunately, an operator may be debugging the network exactly when the assumptions above *do not* hold.

In the next section, we discuss how we solve the problems with prior work outlined here.

1.6 Contributions

We design an accurate "in-band" path measurement system without the limitations of prior solutions. Operators specify measurements declaratively in the form of *queries*, and leverage a *run-time system* that compiles independently specified queries and forwarding behaviors to packet-processing rules on switches. We discuss the query language and the run-time system below.

A Path Query Language. We have developed a *query language* where operators specify regular expressions over boolean conditions on packet location and header contents. To allow concise queries over disjoint subsets of packets, the language includes an SQL-like "groupby" construct that aggregates query results anywhere along a path. Different actions can be taken on a packet when it satisfies a query, such as incrementing counters, directing traffic to a mirroring port or controller, or sampling at a given rate. These actions may be applied either before or after the packets traverse the matching trajectory.

The Run-time System. To implement a path query, the run-time system programs the switches to record path information in each packet as it flows through the data plane. While prior approaches have tracked packet paths this way [48, 83, 90], a naive encoding of every detail of the path—location and headers—would incur significant overheads. For example, encoding a packet's source and destination MAC addresses, and connection 5-tuple (24 bytes) at each hop incurs more than a 10% space overhead on a 1500-byte packet, if the packet takes six hops.

Instead, we *customize packet path information to the input queries.* More specifically, the run-time system compiles queries into a deterministic finite automaton (DFA), whose implementation is then distributed across the switches. The state of the DFA is stored in each packet as updated as it traverses the network. Upon receiving a packet, the switch reads the current DFA state, checks conditions implied by the query, writes a new DFA state on to the packet, executes actions associated with forwarding policy, and sends the packet on its way. Further, if a packet reaches an accepting state of the DFA, the actions associated with the accepting state are triggered. Hence, if the action associated with an accepting state is to send the packet to a collector, only packets actually matching a query are ever sent to a collector.

The mechanism we propose has an attractive "pay for what you query" cost model. Intuitively, our technique acts as an application-specific compression scheme for packet content and paths: rather than encoding every detail of the packet trajectory, *only the information necessary to answer queries* is represented in the automaton state. When a packet hits an accepting state, all user-requested information about the packet path can be reconstructed.

Our system completely avoids the ambiguities of out-of-band approaches that utilize the forwarding policy (§1.5.1, §1.5.2), by carrying path-identifying information directly on the packet itself, and processing this information in the switch. Further, unlike other in-band path measurement schemes (§1.5.3), our system *customizes* packet state to the operator queries, by employing a compression scheme (*i.e.*, using deterministic automata) that results in a small amount of packet state.

A key enabler to building this system is the ability to *modify packet processing based on measurement requirements*, by running on top of an SDN controller and installing measurement rules on switches. However, that brings its own challenges:

(1) Composing measurement and forwarding rules. Switches must identify packets on all operator-specified paths—with some packets possibly on multiple queried paths simultaneously. The switch rules that match and modify packet trajectory metadata should not affect regular packet forwarding in the network, even when operators specify that packets matching the queries be handled differently than the regular traffic. Further, we must ensure that no unecessary traffic is sent over the network. To achieve these goals, we show how we correctly combine sets of query and forwarding rules into a unified set of match-action rules.

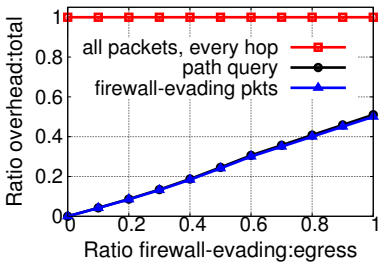


Figure 1.8: Overheads are limited to traffic matching a query.

(2) Resource constraints. The run-time system installs new packet-processing rules on switches, whose rule table space is limited [24]. Hence we must generate a compact rule set. Further, the space to carry packet trajectory metadata is limited, as packets must fit within the network's Maximum Transmission Unit (MTU). Finally, to be usable for operator problem diagnosis, the system should compile queries in an acceptable amount of time. We achieve these goals through several compiler optimizations in our implementation.

Prototype Implementation and Evaluation. We have implemented a prototype of our query system on the Pyretic SDN controller [63] with the NetKAT compiler [95]. Our compilation algorithms generate rules both for single and multi-stage match-action tables, which we have tested with Open vSwitch [73], a popular software switch. We implemented several compiler optimizations that reduce rule-space overhead and query compile time significantly with multi-stage tables. Our system design satisfies requirements (1)-(6) outlined earlier (§1.4). On an emulated campus network topology, our prototype can compile several queries we tested (together) in under 10 seconds. We believe such compile times can enable "interactive" network debugging by human operators. The amount of packet state is less than two bytes, and fits in standard fields like VLAN or MPLS headers. Further, the emitted data plane rules—numbering a few hundreds—fit comfortably in the TCAM memory available on modern switches [12, 24, 43]. As a demonstration of our query system, Fig. 1.8 shows that *only* those packets evading a firewall switch in the network core are collected at the network egress, on an emulated Stanford campus topology [2]. In comparison, common alternatives like wireshark will need to collect all network traffic to reliably catch such packets.

Practically, our query system is complementary to other measurement tools which are "always on" at low overheads, *e.g.,* [1, 87, 122], as opposed to completely replacing those tools. Instead, our query system enables operators to focus their attention and the network's limited resources on clearly-articulated tasks during-the-fact. In summary, this thesis contributes:

1. the design of a query language that allows users to identify packets traversing given paths (§2.1), and an evaluation of query expressiveness through examples (§2.2-§2.3),
2. a run-time system that compiles queries to switch rules that emulate a distributed DFA (§3),

3. a set of optimizations that reduce query compile time by several orders of magnitude (§4.1), and an evaluation of the prototype with the Pyretic SDN controller on Open vSwitch (§4.2).

We have open-sourced our prototype [104] and it is publicly available. Our technical paper [66] on the subject matter of this thesis has been accepted for publication at the Symposium on Networked Systems Design and Implementation (NSDI), 2016. In the following chapters, we delve into the details.

Chapter 2

Path Query Language

> The limits of my language are the limits of my mind. All I know is what I have words for.
>
> *Ludwig Wittgenstein, Logico-Philosophical Treatise*

A path query identifies the set of packets in the data plane with particular header values and that traverse particular locations. Such queries can identify packets moving through a network with possibly changing header values at each location. When the system recognizes that a packet has satisfied a query, any user-specified action may be applied to that packet. In our path query language (§2.1), we provide various constructs to *test* boolean conditions on packets, *combine* such predicates using regular expression operators to form path specifications, *aggregate* the results on headers or locations anywhere along a path, *capture* either at the beginning, end, or middle of a path, and *return* the results as packet payloads, counter statistics, or packet samples. We provide detailed examples of using this language to write queries in §2.2, and dive deep into a network debugging example using a succession of queries in §2.3. In §2.4 we discuss prior work on query languages.

2.1 Language Constructs

What language primitives does an operator need to specify interesting packet paths for network measurement? To answer this question, we determine common patterns that arise repeatedly in operators' questions about the network, and present such language constructs in Fig. 2.1. In what follows, we explain the details through small examples.

```
field ::= location | header
location ::= switch | port
header ::= srcmac | dstmac | srcip | dstip | ...

pred ::= true | false | field=value
         | pred & pred | (pred | pred) | ~pred
         | ingress() | egress()

atom ::= in_atom(pred) | out_atom(pred)
         | in_out_atom(pred, pred)
         | in_group(pred, [header])
         | out_group(pred, [header])
         | in_out_group(pred, [header],
                        pred, [header])

path ::= atom | path ^ path | (path | path)
         | path* | path & path | ~path
```

Figure 2.1: Syntax of path queries.

Packet Predicates and Simple Atoms. One of the basic building blocks in a path query is a *boolean predicate* (`pred`) that matches a packet at a single location. Predicates may match on standard header fields, such as:

```
srcip=10.0.0.1
```

as well as packet's location (a switch or an interface), such as:

```
switch=S10
```

The predicates `true` and `false` match all packets, and no packets, respectively. Conjunction (&), disjunction (|), and negation (~) can be used to put predicates together to construct bigger predicates. For example,

```
srcip=10.0.1.10 & port=3
```

matches packets at port 3 with the specified source IP address.

The language also provides syntactic sugar for predicates that depend on topology, such as `ingress()`, which matches all packets that enter the network at some *ingress* interface, *i.e.*, an interface attached to a host or a device in another administrative domain. Similarly, `egress()` matches all packets that exit the network at some *egress* interface.

Atoms further refine the meaning of predicates, and form the "alphabet" for the language of path queries. The simplest kind of atom is an `in_atom` that tests a packet as it *enters* a switch (*i.e.*, before forwarding actions). Analogously, an `out_atom` tests a packet as it *exits* the switch (*i.e.*, after forwarding actions). The set of packets matching a given predicate at switch entry and exit may be different from each other, since a switch may rewrite packet headers, multicast through several ports, or drop the packet entirely. For example, to capture all packets that enter a device `S1` with a destination IP address (say `192.168.1.10`), we write:

```
in_atom(switch=S1 & dstip=192.168.1.10)
```

It is also possible to combine those ideas, testing packet properties on both "sides" of the forwarding policy on a switch. More specifically, the `in_out_atom` tests one predicate as a packet enters a switch, and another as the packet exits it. For example, to capture all packets that enter a NAT switch with the virtual destination IP address `192.168.1.10` and exit with a private IP address `10.0.1.10`, we would write:

```
in_out_atom(switch=NAT & dstip=192.168.1.10, dstip=10.0.1.10)
```

Partitioning and Indexing Sets of Packets. It is often useful to specify groups of related packets concisely in one query. We introduce *group atoms*—akin to SQL `groupby` clauses—that aggregate results by packet location or header field. These group atoms provide a concise notation for partitioning a set of packets that match a predicate into subsets based on the value of a particular packet attribute. More specifically, `in_group(pred, [h1,h2,...,hn])` collects packets that match the predicate `pred` at switch entry, and then divides those packets into separate sets, one for each combination of the values of the headers `h1, h2, ..., hn`. For example,

```
in_group(switch=10, [port])
```

captures all packets that enter switch 10, and organizes them into sets according to the `port` at which traffic entered the switch. Such a groupby query is equivalent to writing a series of queries, one per entry `port`. The path query system conveniently expands `groupbys` for the user and returns a table indexed by the `port`. The `out_group` atom is very similar to `in_group`: naturally, it matches predicates and partitions packets at the exit point of switches.

The `in_out_group` atom generalizes both the `in_group` and the `out_group`. For example,

```
in_out_group(switch=2, true, [port], [port])
```

captures all packets that enter `switch=2`, and exit it (*i.e.*, not dropped), and groups the results by the combination of input and output ports. This single query is shorthand for an `in_out_atom` for each pair of ports `i, j` on switch 2, *e.g.,* to compute a port-level traffic matrix.

Querying Paths. Full paths through a network may be described by combining atoms using the regular path combinators: concatenation (`^`), alternation (|), repetition ($*$), intersection (&), and negation ($\sim$). The most interesting combinator is concatenation: Given two path queries `p1` and `p2`, the query `p1 ^ p2` specifies a path that satisfies `p1`, takes a hop to the next switch, and then satisfies `p2` from that point on. The interpretation of the other operators is natural: `p1 | p2` specifies paths that satisfy *either* `p1` *or* `p2`; `p1*` specifies paths that are zero or more repetitions of paths satisfying `p1`; `p1 & p2` specifies paths that satisfy *both* `p1` and `p2`,

and ∼p1 specifies paths that do *not* satisfy p1. We defer the examples of queries with path combinators to next section (§2.2).

Query Actions. An application can specify what to do with packets that match a query. For example, packets can be counted (*e.g.,* on switch counters), be sent out a specific port (*e.g.,* towards a collector), sent to the SDN controller, or extracted from sampling mechanisms (*e.g.,* sFlow [3]). Below, we show sample code for various use cases. Suppose that p is a path query defined according to the language (Fig. 2.1). Packets can be sent to abstract locations that "store" packets, called *buckets.* There are three types of buckets: *count buckets*, *packet buckets*, and *sampling buckets.* A count bucket is an abstraction that allows the application to count the packets going into it. Packets are not literally forwarded and held in controller data structures. In fact, the information content is stored in counters on switches. Below we illustrate the simplicity of the programming model.

```
cb = count_bucket() // create count bucket
cb.register(f)      // process counts by callback f
p.set_bucket(cb)    // direct packets matching p
...                 // into bucket cb
cb.pull_stats()     // get counters from switches
```

Packets can be sent to the controller, using the packet buckets and an equally straightforward programming idiom. Similarly, packets can also be *sampled* using technologies like NetFlow [1] or sFlow [3] on switches.

In general, an application can ask packets matching path queries to be processed by an arbitrary *NetKAT* policy, *i.e.*, any forwarding policy that is a mathematical function from a packet to a set of packets [7,63]. The output packet set can be empty (*e.g.,* for dropped packets), or contain multiple packets (*e.g.,* for multicasted packets). For instance, packets matching a path query p can be forwarded out a specific mirroring port mp:

```
p.set_policy(fwd(mp)) // forward out mirror port
```

An arbitrarily complex NetKAT policy pol can be used instead of fwd above by writing p.set_policy(pol).

Query Capture Locations. The operator can specify where along a path to capture a packet that satisfies a query: either *downstream*—after it has traversed a queried trajectory, *upstream*—right as it enters the network, or *spliced*—somewhere in the middle. The difference between these three scenarios is illustrated in Fig. 2.2. The packets captured for the same query may differ at the three locations, because the network's forwarding policy may change as packets are in flight, or packets may be lost downstream due to congestion. For query p, the operator writes p.down() to ask matching packets to be captured downstream, p.up() to be captured

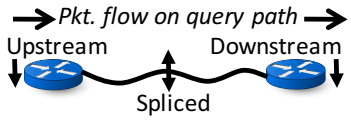


Figure 2.2: Query Capture Locations.

upstream, `p.updown()` to be captured at both locations, and `splice(p1,p2)` to be captured between two sub-paths `p1, p2` such that `p = p1 ^ p2`.

Sometimes, we wish to collect packets at many or even all points on a path rather than just one or two. The convenience function `stitch(A,B,n)` returns a set of queries by concatenating its first argument (*e.g.,* an `in_atom`) with k copies of its second argument (*e.g.,* an `in_group`), returning one query for each `k` in `0...n`. For example, `stitch(A,B,2) = {A, A^B, A^B^B}`.

The capabilities described above allow the implementation of a *network-wide packet capture tool*. Drawing on wireshark terminology, an operator is now able to write global, *path-based capture filters* to collect exactly the packets matching a query.

2.2 Example Applications

In this section, we illustrate the utility of the path query language by walking through example applications of the query language shown in Table 2.1. Path queries enable novel capabilities (*e.g.,* localizing packet loss using just a few queries), significantly reduce operator labor (*e.g.,* measuring an accurate switch-level traffic matrix), and check policy invariants (*e.g.,* slice isolation) in the data plane.

2.2.1 Waypoint Conditions

It is often useful to set up a "live monitor" that checks packets for violations of important policies on waypoints that the packets must (or must not) traverse in a network. The path query system allows collecting only those packets which violate the invariants on the data plane, saving bandwidth on the most common case where packets satisfy the invariants. Let's look at some examples below.

Firewall Evasion. An operator may choose to be alerted whenever packets leave a network without going through a firewall switch `FW`. To do so, he can write the query

```
in_atom(ingress()) ^ (in_atom(~switch=FW))* ^ out_atom(egress())
```

The query as written assumes that the firewall isn't the ingress or egress hop for any packets in the network, but it is easy to relax this assumption by an alternation of paths that allow packets to visit the firewall at the network ingress (or egress) hop.

Example	Query code	Description
A simple path	`in_atom(switch=S1) ^ in_atom(switch=S4)`	Packets going from switch `S1` to `S4` in the network.

Firewall evasion	`in_atom(ingress())`
	`^ (in_atom(∼switch=FW))*`
	`^ out_atom(egress())`
		Catch packets evading a firewall device `FW` when moving from any network ingress to egress interface.

Slice isolation	`true* ^ (in_out_atom(slice1, ∼slice1) |` `in_out_atom(slice2, ∼slice2))`	Packets going out of a network slice (*e.g.,* from `slice1` to `slice2`, or vice versa) on any switch.

Middlebox order	`(true* ^ in_atom(switch=FW) ^ true*) &`
	`(true* ^ in_atom(switch=P) ^ true*) &`
	`(true* ^ in_atom(switch=IDS) ^ true*) &`
	`∼(in_atom(ingress()) ^ true*`
	`in_atom(switch=FW) ^ true*`
	`in_atom(switch=P) ^ true*`
	`in_atom(switch=IDS) ^ true*)`
		Packets that traverse a firewall `FW`, proxy `P` and intrusion detection device `IDS`, but do so in an undesirable order.

Loop detection	`p = in_group(true, [switch, port]);` `p ^ true* ^ p`	Detect packets that visit any fixed switch and port twice in their trajectory.
Switch-level traffic matrix	`in_group(ingress(), [switch]) ^ true*` `^ out_group(egress(), [switch])`	Count packets from any ingress to any egress switch, with results grouped by (ingress, egress) switch pair.

Congested link diagnosis	`in_group(ingress(), [switch]) ^ true*`
	`^ out_atom(switch=sc)`
	`^ in_atom(switch=dc) ^ true*`
	`^ out_group(egress(), [switch])`
		Determine flows (switch sources → sinks) utilizing a congested link (from switch `sc` to switch `dc`), to help reroute traffic around the congested link.

DDoS sources	`in_group(ingress(), [switch]) ^ true*` `^ out_atom(egress(), switch=vic)`	Determine traffic contribution by volume from all ingress switches reaching a DDoS victim switch `vic`.
Port-to-port traffic matrix	`in_out_group(switch=s, true,` `[port], [port])`	Count traffic between any two ports of switch `s`, group-ing the results by the ingress and egress interface.
NAT debugging	`in_out_atom(switch=NAT & dstip=10.0.1.2,` `dstip=192.168.1.2)`	Catch packets entering a `NAT` with destination IP `10.0.1.2` and leaving with the (modified) destination IP `192.168.1.2`.

ECMP debugging	`in_out_group(switch=S1 & ecmp_pred,`
	`port=1 | port=2 | port=3 | port=4,`
	`[], [port])`
		Measure ECMP traffic splitting on switch `S1` for a small portion of traffic (predicate `ecmp_pred`), over ports 1 through 4.

Hop-by-hop debugging	`in_atom(switch=S1 & probe_pred) ^ true*`	Get notified at each hop as probe packets (satisfying `probe_pred`) starting at `S1` make progress through the network.

Packet loss localization	`in_atom(srcip=H1) ^ in_group(true,`
	`[switch]) ^ in_group(true, [switch])`
	`^ out_atom(dstip=H2)`
		Localize packet loss by measuring per-path traffic flow along each 4-hop path between hosts `H1` and `H2`.

Table 2.1: Some example path query applications.

Slice Isolation. Suppose an operator has defined exclusive *slices* of resources in a network, such as host IP addresses, corresponding to multiple tenants in a multi-tenant compute cluster. She may require that packets within one such slice do not go into another. For simplicity, let us suppose that there are two slices whose IP addresses are denoted by predicates `slice1` and `slice2`. To catch packets moving from one slice to another, or alternatively, moving out of any one slice, she can install the query

```
true* ^ (in_out_atom(slice1, ∼slice1) | in_out_atom(slice2, ∼slice2))
```

Middlebox Traversal Order. Correct and efficient operation of middleboxes may require that all packets in the network traverse the middleboxes in a specific order [85]. For example, the operator may state that every packet must go through a firewall `FW`, a proxy `P` and an intrusion detection system `IDS` in that specific order:

```
p1 = ∼(in_atom(ingress()) ^ true*
        ^ in_atom(switch=FW) ^ true*
        ^ in_atom(switch=P) ^ true*
        ^ in_atom(switch=IDS) ^ true*
        ^ out_atom(egress()))
```

Note that this query also catches packets which skip any of the middleboxes, in addition to those that traverse all of them but in an undesirable order. To further constrain the query to only those packets that do traverse each of the middleboxes, we can intersect `p1` with other paths as shown below.

```
p = (true* ^ in_atom(switch=FW) ^ true*) &
    (true* ^ in_atom(switch=P) ^ true*) &
    (true* ^ in_atom(switch=IDS) ^ true*) &
     p1
```

Loop Detection in Data Plane. When packets loop around in the data plane, it wastes bandwidth, and delays (or in the worst case prevents) the destination from receiving packets. Loop detection is an interesting waypoint query that chooses waypoints on the fly—we wish that a packet visiting *some* interface should never visit *the same interface* again. To specify this, we could write a query for each interface in the network, as shown below:

```
for (sw, intf) in network_interfaces:
  q = in_atom(switch=sw & port=intf) ^ true* ^ in_atom(switch=sw & port=intf)
```

Alternatively, we could also use a group atom to generate a query for each network interface:

```
some_port = in_group(true, [switch, port])
p = some_port ^ true* ^ some_port
```

2.2.2 Ingress-Egress Traffic Flow

Network operators often find it useful to track the volume of traffic flowing through a cross-section of the network, or the entire network, as we discuss in the examples below.

Switch-level Traffic Matrix. A switch-level traffic matrix is an $M \times N$ matrix for a network with M ingress and N egress switches, and the entry (i, j) corresponds to the traffic demand from ingress switch i to egress switch j. The traffic matrix is a useful input for traffic engineering [32] and anomaly detection [58]. It is possible to write one query to capture each ingress and egress switch demand by hand, but the `groupby` constructs allow an operator to write this query very concisely by aggregating on *both* the ingress and egress switch of any traffic:

```
in_group(ingress(), [switch]) ^ true* ^ out_group(egress(), [switch])
```

Congested Link Diagnosis. When a link in a network becomes congested, it is useful for operators to know which sources are utilizing the congested link, and which downstream destinations are affected by it. This enables corrective actions such as re-routing demands from some ingress points to some "heavy" destinations, to avoid congesting the link downstream. Suppose the operator is interested in the demands utilizing a congested link between switches `Si` and `Sj`. Then she writes the query

```
in_group(ingress(), [switch]) ^ true*
  ^ out_atom(switch=Si) ^ in_atom(switch=Sj)
  ^ true* ^ out_group(egress(), [switch])
```

to capture switch-level demands on the congested link. It is also possible to write a query to capture destination IP-level flows, since routing is typically destination IP-based:

```
in_group(ingress(), [switch]) ^ true*
  ^ out_atom(switch=Si) ^ in_atom(switch=Sj)
  ^ true* ^ out_group(egress(), [dstip])
```

DDoS Sources. If a machine is under a distributed denial-of-service (DDoS) attack in a network, it is helpful to understand the traffic contribution by volume from all ingress (source) locations to the victim. Let the victim destination switch be `S_vic`. The following query measures traffic volumes from every ingress to `S_vic`:

```
in_group(ingress(), [switch]) ^ true* ^ out_atom(egress() & switch=S_vic)
```

A similar query can be used to measure the spread of traffic originating from some location in a network.

Port-to-port Traffic Matrix. As mentioned in §2.1, a single query can capture the traffic volumes between every two ports on a switch:

```
in_out_group(switch=S, true, [port], [port])
```

2.2.3 Troubleshooting

It is often useful for operators to inspect small portions of traffic for troubleshooting purposes, to determine how the network handles specific subsets of traffic. We discuss below some examples of how path queries can help in such scenarios.

Detecting Incorrect NAT Operation. Devices that rewrite packets can be an obstacle to troubleshooting a network because they hide the previous values of the header fields that were rewritten. For example, if a Network Address Translation (NAT) device [69] is faulty, an operator can write a query to inspect packets with a specific input-output relationship on the switch which rewrites packets:

```
in_out_atom(switch=NAT & dstip=10.0.1/24, dstip=192.168.1/24)
```

Measuring Load Balancing. Balancing traffic across a number of output interfaces of a switch is a very useful mechanism in a network, enabling operators to push more traffic demand with less link congestion, *e.g.,* [5, 46]. Unfortunately, the exact load balancing primitives (*e.g.,* hash functions) that a switch uses are specific to the device vendor [64]. As a result, when a link congestion occurs, it is typically unclear whether it is due to imbalance in demands of the flows routed through the link (*e.g.,* due to so-called "elephant flows" which have higher volume relative to all other flows [4]), or an imbalance in the number of flows routed through each interface. The same question can also be posed for a subset of traffic, *e.g.,* is traffic from an IP subnet balanced evenly across the available paths?

Suppose the operator is interested in determining the fractions of traffic matching the predicate `ecmp_pred` routed through ports 1 through 4 of switch `S1`. She can install the following query:

```
in_out_group(switch=S1 & ecmp_pred,
             port=1 | port=2 | port=3 | port=4,
             [], [port])
```

Hop-by-hop Debugging. It is possible to create a "packet interpreter" for a very small subset of traffic *on-the-fly* using a path query. This enables each packet to be inspected at a controller, as the packet makes its way through the network. Suppose that such packets match the predicate `probe_pred`, and the operator wishes to inspect them starting from switch `S1`. The query

```
in_atom(switch=S1 & probe_pred) ^ true*
```

would send the corresponding packets to the controller at each hop, processing it by the forwarding policy otherwise.

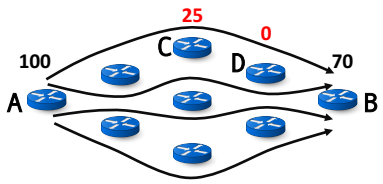


Figure 2.3: An example debugging scenario (§2.3).

These examples illustrate the breadth of applicability of the path query language. In the next section, we dive deep into a specific troubleshooting scenario showing how a sequence of queries helps an operator quickly localize the source of a network problem.

2.3 Interactive Debugging with Path Queries

Packet loss in a network is very problematic for network operators running high-performance applications, as they significantly increase the completion time for short and revenue-generating (§1) flows. Unfortunately in a large network, it is quite challenging to localize the point where packets are dropped. Consider an example scenario shown in Fig. 2.3, where an operator is tasked with diagnosing a tenant's performance problem in a large compute cluster, where the connections between two groups of tenant virtual machines A and B suffer from poor performance with low throughput. The $A \rightarrow B$ traffic is routed along the four paths shown.

Such performance problems do occur in practice [122], yet are very challenging to diagnose, as none of the conventional techniques really help. Getting information from the end hosts' networking stack [99,114] is difficult in virtualized environments. Coarse-grained packet sampling (NetFlow [1], [27]) may miss collecting the traffic relevant to diagnosis, *i.e.*, A and B traffic. Interface-level counters from the device may mask the problem entirely, as the issue occurs with just one portion of the traffic. It is possible to run wireshark [112] on switch CPUs; however this can easily impact switch performance and is very restrictive in its application [22]. Network operators may instead mirror a problematic subset of the traffic in the data plane through ACLs, *i.e.*, "match and mirror" [122]. However, this process is tedious and error-prone. The new monitoring rules must incorporate the results of packet modification in flight (*e.g.,* NATs and load balancers [77]), and touch several devices because of multi-path forwarding. The new rules must also be reconciled with overlapping existing rules to avoid disruption of regular packet forwarding. Ultimately, mirroring will incur large bandwidth and data collection overheads, corresponding to all mirrored traffic.

In contrast, we show the ease with which the path query system allows an operator to determine the root cause of the performance problem. In fact, the operator can perform *efficient* diagnosis using just switch counters—without mirroring any packets.

As a first step, the operator determines whether the end host or the network is problematic, by issuing a query counting all traffic that enters the network from A destined to B. She writes the query `p1` below:

```
p1 = in_atom(srcip=vm_a, switch=s_a) ^ true* ^ out_atom(dstip=vm_b, switch=s_b)
p1.updown()
```

The run-time then provides statistics for $A \rightarrow B$ traffic, measured at network ingress (upstream) and egress (downstream) points. By comparing these two statistics, the operator can determine whether packets never left the host interface card, or were lost in the network.

Suppose the operator discovers a large loss rate in the network, as query `p1` returns values 100 and 70 as shown in Fig. 2.3. Her next step is to localize the interface where most drops happen, using a downstream query `p2`:

```
probe_pred = switch=s_a & srcip=vm_a & dstip=vm_b
p2 = stitch(in_atom(probe_pred), in_group(true, [switch]), 4)
```

These queries count $A \rightarrow B$ traffic on each switch-level path (and its prefix) from A to B. Suppose the run-time returns, among statistics for other paths, the packet counts 25 and 0 shown in red in Fig. 2.3. The operator concludes that link $C \rightarrow D$ along the first path has a high packet drop rate (all 25 packets dropped). Such packet drops may be due to persistent congestion or a faulty interface, affecting all traffic on the interface, or faulty rules in the switch (*e.g.,* a "silent blackhole" [122]) which affect just $A \rightarrow B$ traffic. To distinguish the two cases, the operator writes two queries measured midstream and downstream (each). Here are the midstream queries:

```
probe_pred = switch=s_a & srcip=vm_a & dstip=vm_b
p3 = splice(in_atom(probe_pred) ^ true* ^ in_atom(switch=s_c), in_atom(switch=s_d))
p4 = splice(true* ^ in_atom(switch=s_c), in_atom(switch=s_d))
```

These queries determine the traffic loss rate on the $C \rightarrow D$ link, for all traffic traversing the link, as well as specifically the $A \rightarrow B$ traffic. By comparing these two loss rates, the operator can rule out certain root causes in favor of others. For example, if the loss rate for $A \rightarrow B$ traffic is particularly high relative to the overall loss rate, it means that that just the $A \rightarrow B$ traffic is silently dropped. On the other hand, if both loss rates are comparable, it could be a sporadic issue with the switch interface itself.

We believe that our query abstractions that enable operators to unearth relevant information from a live network—especially in an iterative fashion—can significantly simplify troubleshooting practice.

Primitive	Description	Prior Work	Differences
Atomic predicates	Boolean tests on located packets	Frenetic [35], Pyretic [63]	Match at specific switch locations, *e.g.,* input/output interface
Packet paths	Regular expressions on atomic predicates	NetSight [40]	More regular expression operators (&, $\sim$)
Result aggregation	Distinguish combinations of specific locations and headers	Frenetic [35], SQL `groupby`	Group anywhere and at multiple hops along path
Capture location	Packet captures before, in the middle of, or after queried path	*None*	*Not applicable*
Capture result	Actions on packets satisfying queries	Pyretic [63]	Additional actions: sampling [1, 3] and path-based forwarding

Table 2.2: Prior query languages and differences from our path query language.

2.4 Prior Query Languages

We wrap up this chapter with a brief discussion of existing work on query languages in networking literature. The most closely related works to our language design are Frenetic [35], Pyretic [63] and NetSight [40]. We borrow useful primitives from them; however by themselves these languages are insufficient to express concise path queries, as we discuss below.

First, Pyretic and Frenetic support specification of packet capture *at a single location*—combinations of which captures are neither expressive nor accurate for path measurement (§1.5.2). Further, the Frenetic and Pyretic (*i.e.*, `groupby`) constructs are similarly constrained: they don't allow aggregation by locations or headers *across multiple hops.* Finally, while NetSight enables operators to write regular expression path specifications, it does not consider any operator inputs on how and where packets should be captured, missing crucial opportunities to control measurement overhead.

For each primitive in our language, we summarize the chief related works and differences in Table 2.2. We discuss other related works below.

Data-plane query systems. Several query languages have been proposed for performing analytics over streams of packet data [11, 23, 35, 106]. Unlike these works, we address the collection of *path-level* traffic flows, *i.e.*, observations of the same packets *across* space and time, which cannot be expressed concisely or achieved by (merely) asking for single-point packet observations.

Control-plane query systems. NetQuery [93] and other prior systems [16, 19, 45] allow operators to query information (*e.g.,* next hop for forwarding, attack fingerprints, *etc.*) from tuples stored on network nodes. As such, these works do not query the data plane. SIMON [68] and ndb [59] share our vision of interactive debugging, but focus on isolating control plane bugs.

Programming traffic flow along paths. Several prior languages [7, 30, 47, 88, 97] specify how to *forward* packets along policy-compliant paths. However, language constructs for aggregation, capture results and capture location are unique to measurement. Further, our language specifies how to *measure* traffic along operator-specified paths, while the usual forwarding policy continues to handle traffic.

So far, we have described language constructs to write concise packet path specifications, and discussed several applications. In the next chapter, we explain how we implement such specifications on switches.

Chapter 3

Path Query Compilation

> To understand a program you must become both the machine and the program.
>
> *Alan Perlis, Epigrams of Programming*

How are path queries translated to switch rules that recognize *exactly* the packets that match the queries? This is the subject of this chapter. Query compilation translates a collection of independently specified queries, along with the forwarding policy, into a unified set of data-plane rules that perform both path measurement and forwarding. We describe *downstream* query compilation, *i.e.*, recognizing packets after they traverse a queried path, in §3.1, and then *upstream* compilation in §3.2.

3.1 Downstream Query Compilation

Downstream query compilation aims to capture packets after they traverse paths satisfying regular expressions on predicates. A simple strawman solution is to append information about the current location and headers of a packet at every hop of the packet's path—every downstream switch then has sufficient information to determine if the packet has gone through a path that satisfies the query. However, there are two problems with this approach: there isn't enough space on the packet to put all this information, and match-action devices can't match regular expression patterns on packets.

Our key insight is that we can simplify the packet state by using the structure of the queries, *i.e.*, regular expressions. From formal language theory, we observe that the computation of recognizing whether a string satisfies a regular expression can be encoded as a finite state automaton [49]. A deterministic[1] finite state automaton is an abstract machine that models computation, wherein the machine can be in one of a finite

[1] Finite state automata can also be non-deterministic, with the machine existing in a *set* of states at any given time.

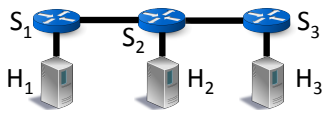


Figure 3.1: Topology for the running example (§3).

number of abstract states, starting from a known *initial state.* By reading input symbols, the machine can *transition* from one state to another. The machine *accepts* an input if the automaton is in a fixed subset of its states known as *accepting states,* after reading the input. Otherwise, the automaton *rejects* the input. By encoding the state of the automaton corresponding to the path queries into packets, we can process each packet according to conditions it encounters on its path, and "accept" a packet if the corresponding automaton accepts one or more queries. We observe that the automaton transition function can be written down as match-action rules, and the amount of state needed is just a few bytes (actual experiments are in §4.2).

Now we work out the details of this intuition. Downstream query compilation proceeds in three main stages:

1. We convert the regular expressions corresponding to the path queries into a deterministic finite automaton (DFA) (§3.1.1).
2. Using the DFA as an intermediate representation, we generate state-transitioning (*i.e.*, *tagging*) and accepting (*i.e.*, *capture*) data-plane rules. These allow switches to match packets based on the state value, rewrite state, and capture packets which satisfy one or more queries (§3.1.2).
3. Finally, the run-time combines the query-related packet-processing actions with the regular forwarding actions specified by other controller applications. This is necessary because the state match and rewrite actions happen on the *same* packets that are forwarded by the switches (§3.1.3).

The run-time first expands group atoms into the corresponding basic atoms by a pre-processing pass over the queries which we describe in §3.1.4. But first, we focus on the remainder of the compilation process, assuming that the queries only contain `in`, `out`, and `in_out` atoms. We use the following example queries running on the network shown in Fig. 3.1 to illustrate the compilation algorithms.

```
p1 = in_atom(srcip=H1 & switch=1) ^ out_atom(switch=2 & dstip=H2)
p2 = in_atom(switch=1) ^ in_out_atom(true, switch=2)
```

3.1.1 From Path Queries to DFAs

We first compile the path queries—which look like regular expressions—into an equivalent DFA. We could conceivably use a Non-deterministic Finite Automaton (NFA) instead of a DFA, to produce fewer states.

However, using an NFA would require each packet to store all the possible states that it might inhabit at a given time, and require switches to have a rule for each subset of states—leading to a large number of rules. Hence, we compile our path queries to a DFA. We do this in three steps, as follows.

Rewriting atoms to in-out-atoms. The first step is quite straightforward. For instance, the path query `p1` is rewritten to the following:

```
in_out_atom(srcip=H1 & switch=1, true) ^ in_out_atom(true, switch=2 & dstip=H2)
```

Explicitly writing down the predicates matched both at switch entry and exit as above allows the forthcoming algorithms to analyze overlaps between different query atoms.

Converting queries to regular expressions. In the second step, we convert the path queries into string regular expressions, by replacing each predicate by a character literal. However, this step is tricky: a key constraint is that different characters of the regular expressions cannot represent overlapping predicates (*i.e.*, predicates that can match the same packet). If they do, we may inadvertently generate an NFA (*i.e.*, a single packet might match two or more outgoing edges in the automaton). To ensure that characters represent non-overlapping predicates, we devise an algorithm that takes an input set of predicates P, and produces the smallest orthogonal set of predicates S that matches all packets matching P. The key intuition is as follows. For each new predicate `new_pred` in P, the algorithm iterates over the current predicates `pred` in S, teasing out new disjoint predicates and adding them to S:

```
int_pred = pred & new_pred
new_pred = new_pred & ∼int_pred
pred = pred & ∼int_pred
```

Finally, the predicates in S are each assigned a unique character.

Now we describe the full algorithm. Algorithm 1 takes an input set of predicates `P`. The partitioned set `S` is initialized to a null set (line 2). We iterate over the predicates in `P`, teasing out overlaps with existing predicates in `S`. If the current input predicate `new_pred` already exists in `S`, we move on to the next input (lines 5-6). If a predicate `pred` in `S` is a superset of `new_pred`, we split `pred` into two parts, corresponding to the parts that do and don't overlap with `new_pred` (lines 7-11). Then we move to the next input predicate. The procedure is symmetrically applied when `pred` is a subset of `new_pred` (lines 12-13), except that we continue looking for more predicates in `S` that may overlap with `new_pred`. Finally, if `pred` and `new_pred` merely intersect (but neither is a superset of the other), we create three different predicates in `S` according to three different combinations of overlap between the two predicates (lines 14-20). Finally, any remaining pieces of `new_pred` are added to the partitioned set `S`. In each case in the algorithm above, we also keep track

Algorithm 1 Predicate partitioning (§3.1.1).

```
P = set_of_predicates
S = ∅
for new_pred ∈ P do
    for pred ∈ S do
        if pred is equal to new_pred then
            continue the outer loop
        else if pred is a superset of new_pred then
            difference = pred & ∼ new_pred
            S ← S ∪ {difference, new_pred}
            S ← S \ {pred}
            continue the outer loop
        else if pred is a subset of new_pred then
            new_pred ← new_pred & ∼ pred
        else if intersect then
            inter1 = pred & ∼ new_pred
            inter2 = ∼ pred & new_pred
            inter3 = pred & new_pred
            S ← S ∪ {inter1, inter2, inter3}
            S ← S \ {pred}
            new_pred ← new_pred & ∼ pred
        end if
    end for
    S ← S ∪ {new_pred}
end for
```

of the predicates in the partitioned set `S` with which the input predicate in `P` overlaps. Further, note that we can run the partitioning algorithm separately for the switch ingress and egress predicates since they are never simultaneously matched on a packet.

Our usage of packet predicates and partitioning to form an orthogonal basis are closely related to symbolic automata and minterms [25, 116]. Effectively, we reduce DFA construction over predicates to the standard version for character literals.

For the running example, Fig. 3.2 shows the emitted characters (for the partitioned predicates) and regular expressions (for input predicates not in the partitioned set). Notice in particular that the `true` predicate coming in to a switch is represented not as a single character but as an alternation of three characters. Likewise with `switch=1`, `switch=2`, and `true` (out). The final regular expressions for the queries `p1` and `p2` are:

```
p1: a^(c|e|g)^(a|d|f)^c
```

```
p2: (a|d)^(c|e|g)^(a|d|f)^(c|e)
```

Constructing the query DFA. Finally, we construct the DFA for `p1` and `p2` together using standard techniques. The DFA is shown in Fig. 3.3. For clarity, state transitions that reject packets from both queries are not shown.

Predicate	Regex	Predicate	Regex
switch=1 & srcip=H1	a	∼switch=1	f
switch=1 & ∼srcip=H1	d	∼switch=2	g
switch=2 & dstip=H2	c	switch=1	a|d
switch=2 & ∼dstip=H2	e	switch=2	c|e
true (in)	a|d|f	true (out)	c|e|g

Figure 3.2: Strings emitted for the running example (§3.1.1).

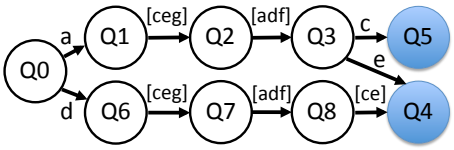


Figure 3.3: Automaton for p1 and p2 together. State Q4 accepts p2, while Q5 accepts both p1 and p2.

3.1.2 From DFA to Tagging/Capture Rules

The next step is to emit policies that implement the DFA. Conceptually, we have two goals. First, for each packet, a switch must read the DFA state, identify the appropriate transition, and rewrite the DFA state. This action must be done once at switch ingress and egress. Second, if the packet's new DFA state satisfies one or more queries, we must perform the corresponding query actions, *e.g.,* increment packet or byte counts.

State transitioning policies. The high-level idea here is to construct a "test" corresponding to each DFA transition, and rewrite the packet DFA state to the destination of the transition if the packet passes the test. This is akin to a string-matching automaton checking if an input symbol matches an outgoing edge from a given state.

To make this concrete, we show the intermediate steps of constructing the transitioning policy in Pyretic code. But first, we briefly introduce the notions of *parallel and sequential composition,* and *network policies.* Pyretic [63] and NetKAT [7] define a network policy as a mathematical function from a packet to a set of packets. For example, a *match* `srcip=10.0.0.1` is a function that returns the singleton set of its input packet if the packet's source IP address is `10.0.0.1`, and an empty set otherwise. Similarly, a *modification* `port←2` is a function that changes the "port" field of its input packet to 2. Given two policies `f` and g—two functions on packets to sets of packets—the *parallel composition* of these two policies is defined as:

`(f + g)(pkt)` $=$ `f(pkt)` $\cup$ `g(pkt)`

The *sequential composition* of policies is defined as:

`(f >> g)(pkt)` $= \cup_{pkt' \in f(pkt)}$ `g(pkt')`

For example, the policy

```
(srcip=10.0.0.1 + dstip=10.0.0.2) >> (port←2)
```

Concept	Example	Description
Modification	`port←2`	Rewrites a packet field
Match	`switch=2`	Filters packets
Parallel composition	`monitor + route`	The union of results from two policies.
Sequential composition	`balance >> route`	Pipe the output from the first in to the second
Edge predicate	`pred_of(c)`	Get predicate of symbol
Path policy	`p.policy()`	Policy to process packets accepted by query `p`.

Figure 3.4: Syntactic Constructs in Query Compilation.

selects packets with either `srcip 10.0.0.1` or `dstip 10.0.0.2` and forwards them out of port 2 of a switch. A NetKAT compiler [95] can translate a network policy consisting of basic elements like `match` and `modify`, and the composition operators (`>> and +`) into a set of match-action rules targeting a single-stage match-action table—a fact we will use later (§3.1.3).

Now we produce a policy fragment for each edge of the DFA. Suppose the helper function `pred_of` takes in a character input `c` and produces the corresponding predicate. For each edge from state `s` to state `t` that reads character `c`, we construct the fragment

```
state=s & pred_of(c) >> state←t
```

We combine these fragments through parallel composition, which joins the tests and actions of multiple edges:

```
tagging = frag_1 + frag_2 + ... + frag_n
```

We produce *two* state transitioning policies, one each for ingress and egress actions. Each edge fragment belongs to exactly one of the two policies, and it is possible to know which one since we generate disjoint characters for these two sets of predicates. For example, here is part of the ingress transitioning policy for the DFA in Fig. 3.3:

```
in_tagging =
  state=Q0 & switch=1 & srcip=H1 >> state←Q2 +
  state=Q0 & switch=1 & ∼srcip=H1 >> state←Q6 +
   ... +
  state=Q7 & ∼switch=1 >> state←Q8
```

Accepting policies. The accepting policy is akin to the *accepting* action of a DFA: a packet that "reaches" an accepting state has traversed a path that satisfies some query; hence the packet must be processed by the

actions requested by applications. We construct the accepting policy by combining edge fragments which move packets to accepting states. We construct the fragment

```
state=s & pred_of(c) >> p.policy()
```

for each DFA edge from state `s` to `t` through character `c`, where `t` is a state accepting query `p`. Here `p.policy()` produces the action that is applied to packets matching query `p`. Next we construct the *accepting* policy by a parallel composition of each such fragment:

```
capture = frag_1 + frag_2 + ... + frag_n
```

Similar to the transitioning policies, we construct two accepting policies corresponding to switch ingress and egress predicates. For example, for the DFA in Fig. 3.3, the accepting policy looks as follows:

```
state=Q3 & switch=2 & dstip=H2 >> p1.policy()
+ state=Q3 & switch=2 & dstip=H2 >> p2.policy()
+ state=Q3 & switch=2 & ∼dstip=H2 >> p2.policy()
+ state=Q8 & switch=2 & dstip=H2 >> p2.policy()
+ state=Q8 & switch=2 & ∼dstip=H2 >> p2.policy()
```

Ingress tagging and Egress un-tagging. The run-time ensures that packets entering a network are tagged with the initial DFA state `Q0`. Symmetrically, packets leaving the network are stripped of their tags. We use the VLAN header to tag packets, but other mechanisms are possible.

3.1.3 Composing Queries and Forwarding

The run-time system needs to combine the packet-processing actions from the transitioning and accepting policies with the forwarding policy. However, this requires some thought, as all of these actions affect the *same* packets. Concretely, we require that:

1. packets are forwarded through the network normally, independent of the existence of queries,
2. packet tags are manipulated according to the DFA,
3. packets matching path queries are processed correctly by the application-programmed actions, and
4. no unnecessary[2] duplicate packets are generated.

To achieve these goals, the run-time system combines the constituent policies as follows:

```
(in_tagging >> forwarding >> out_tagging)
+ (in_capture)
+ (in_tagging >> forwarding >> out_capture)
```

[2]By "unnecessary," we mean cases where no forwarding or query action required the packet duplication.

The first sequential composition (involving the two `tagging` policies and the `forwarding`) ensures both that forwarding continues normally (goal 1) as well as that DFA actions are carried out (goal 2). This works because `tagging` policies do not drop packets, and the `forwarding` does not modify the DFA state.[3] The remaining two parts of the top-level parallel composition (involving the two `capture` policies) ensure that packets reaching accepting states are processed by the corresponding query actions (goal 3). Finally, since each parallelly-composed fragment either forwards packets normally or captures it for the accepted query, no unnecessary extra packets are produced (goal 4).

The run-time system hands off the joint policy above to Pyretic, which by default compiles it down to a single match-action table [35, 63].

3.1.4 Expanding Group Atoms

We now describe how a query with group atoms, *i.e.*, `in, out` and `in_out_group`, is expanded to a set of queries which only contain *simple* atoms, *i.e.*, `in, out` and `in_out_atoms`. We perform this expansion proactively before the other stages of the compilation.[4] Conceptually, there are two steps here. First, we enumerate the set of possible simple atoms for each *unique* group atom in the queries. Then, for each combination of the simple atoms, we generate a new query that is structurally similar to the original, but with the simple atoms dropped in place of the original group atoms. We illustrate how this works with an example on the network shown in Fig. 3.1:

```
portx = in_group(switch=1, [port])
hop = out_group(true, [switch])
p = portx ^ hop ^ portx
```

This query captures all packet *loops* starting from a port on switch 1 and returning to the same port after one hop. Further, the query also distinguishes results by the other switch that was visited.

Enumerating simple atoms for group atoms. Consider an atom `in_group(pred, [field1, ..., fieldN])` that requires packets matching `pred` to be grouped by `field1` through `fieldN`. We consider all possible values of the aggregation fields,[5] and generate a new predicate corresponding to each combination of possible values. For one such combination, say `value1, ..., valueN`, we would construct the simple atom

```
in_atom(pred & field1=value1 & ... & fieldN=valueN)
```

[3] The run-time ensures this by constructing `tagging` policies with a virtual header field [63] that regular `forwarding` policies do not use.

[4] Reactive expansion of group atoms according to network traffic is also possible.

[5] For topology fields, these values are currently inferred by the run-time system. For packet header fields, we require a list of values to be initialized separately.

For our example query, we enumerate these simple atoms corresponding to the group atoms:

```
portx: in_atom(switch=1 & port=1); in_atom(switch=1 & port=2)
hop: out_atom(true & switch=1); out_atom(true & switch=2); out_atom(true & switch=3)
```

The procedure for `out_group` and `in_out_group` are similar.

Inserting simple atoms in place. For each combination of assignments from group atoms to simple atoms, we now create a new query through simple substitution. One such query in our example would be

```
p' = in_atom(switch=1 & port=1) ^ out_atom(true & switch=2)
       ^ in_atom(switch=1 & port=1)
```

Note that this is just one query among $2 \times 3 = 6$ queries possible from as many substitutions for group atoms.

3.2 Upstream Query Compilation

Upstream query compilation finds those packets at network ingress that *would* match a path, based on the current forwarding policy—assuming that packets are not dropped (due to congestion) or diverted (due to updates to the forwarding policy while the packets are in flight). We compile upstream queries in three steps, as follows.

Compiling using downstream algorithms. The first step is straightforward. We use algorithms described in sections §3.1.1-§3.1.3 to compile the set of upstream queries using *downstream* compilation. The output of this step is the *effective* forwarding policy of the network incorporating the behavior both of forwarding and queries. Note that we do *not* install the resulting rules on the switches.

Reachability testing for accepted packets. In the second step, we cast the upstream query compilation problem as a standard network *reachability test* [53, 54]. Specifically, we ask: when processed by the *effective* forwarding policy from the previous step, which packets—starting from the network ingress—eventually reach an accepting DFA state? We leverage *header space analysis* [53] to efficiently compute the full set of packet headers that match the query. We briefly review header space analysis and reachability testing below.

Header space analysis (HSA) models packets as objects existing in an L-dimensional geometric space: a packet is a flat vector of 0s and 1s of length L, which is the upper bound on the number of packet header bits. Sets of packet *headers*, h, are hence *regions* in the space of $\{0,1\}^L$. A switch T is modeled as a *transfer function* that maps headers arriving on a port, to set(s) of headers and ports: $T(h,p) : (h,p) \rightarrow \{(h_1,p_1),(h_2,p_2),\ldots\}$. The action of the network topology is also modeled similarly: if p_1 and p_2 are ports

on the opposite ends of a link, then $T(h, p_1) = \{(h, p_2)\}$. The inverse of a switch or topology transfer function is well-defined [53].

Reachability testing asks which of all possible packet headers at a source can reach a destination header space and port. To compute this, HSA takes the set of all headers at a source port, and applies the network's transfer function iteratively until the header spaces reach the destination port. If the final header space at the destination port is empty, the source cannot send any packets that reach the destination port. Otherwise, the *inverses* of the transfer functions are iteratively applied to the headers at the destination port to compute the *full set of packets* that the source can send to reach the destination.

Now we simply ask which packets at network ingress, when forwarded by the *effective* policy above, *reach* header spaces corresponding to accepting states for query `p`. We call this packet match `upstream(p)`.

Capturing upstream. The final step is to process the resulting packet headers from reachability testing with application-specified actions for each query. We generate an *upstream capture* policy for queries `p1, ..., pn`:

```
(upstream(p1) >> p1.policy()) + ...
+ (upstream(pn) >> pn.policy())
```

We make a few remarks. First, without downstream query compilation, a straightforward application of reachability testing or HSA will not return the packets matching a regular expression query. Further, we can implement complex applications of HSA like loop detection and slice leakage detection [53, §5] simply by compiling the corresponding upstream path query (Table 2.1). In general, reachability testing does not restrict the paths taken to reach the destination—however, perhaps surprisingly, we are able to use the packet DFA state to do exactly that. Finally, we can compile spliced queries, *e.g.,* `splice(p1,p2)`, by asking for packets that *reach* `p2`-accepting states starting out as packets accepted by `p1` at any network port.

In this chapter, we have shown algorithms that take a path specification written in our query language (§2) and translate it to an accurate path measurement using a distributed DFA running on switches. In the next chapter, we discuss how we make this system practical.

Chapter 4

Optimizing Path Queries

> In computing, turning the obvious into the useful is a living definition of the word "frustration."
>
> *Alan Perlis, Epigrams of Programming*

Is it possible to make network measurement an interactive process for operators, at practical overheads? Could operators debug their networks in an iterative fashion akin to general-purpose program code?

While the compilation algorithms in §3 produce data plane rules to measure paths *accurately*, they fail to do so *efficiently.* For example, we will see later in this chapter that it takes more than a couple of hours to compile a mix of queries to a campus network topology, using over tens of thousands of switch rules! We implemented several key optimizations in our prototype to reduce query compile time and data-plane rule space (§4.1). Our performance evaluation shows that our optimizations bring down compile time and rule space by at least three orders of magnitude (§4.2), helping us achieve interactive time scales for measurement.

4.1 Compiler Optimizations

We first describe a fundamental challenge—the "cross-product explosion" problem—that results in large compilation times and rule-sets when compiling the policies resulting from algorithms in §3.

Cross-product explosion. The output of policy compilation is simply a prioritized list of match-action rules, which we call a *classifier.* When two classifiers C_1 and C_2 are composed—using parallel or sequential composition (§3.1.2, Fig. 3.4)—the compiler must consider the effect of every rule in C_1 on every rule in C_2. If

the classifiers have N_1 and N_2 rules (resp.), this results in a $\Theta(N_1 \times N_2)$ operations. To see this, consider two classifiers `C1, C2`, and their sequential composition below:

```
C1 =                                        C2 =

    state=Q0 & srcip=A ⇒ state←Q1               dstip=C ⇒ fwd(1)

    state=Q1 & srcip=B ⇒ state←Q2               dstip=D ⇒ fwd(2)


C1 >> C2 =

          state=Q0 & srcip=A & dstip=C ⇒ state←Q1; fwd(1)

          state=Q0 & srcip=A & dstip=D ⇒ state←Q1; fwd(2)

          state=Q1 & srcip=B & dstip=C ⇒ state←Q2; fwd(1)

          state=Q1 & srcip=B & dstip=D ⇒ state←Q2; fwd(2)
```

A similar problem arises when predicates are partitioned during DFA generation (§3.1.1). For instance, suppose a query contains `N` predicates each matching on a different field, *i.e.*, `field_i=val_i` for `i` in `1...N`. Notice that each predicate intersects with every other, since there are packets which satisfy `field_i=val_i` and `field_j=val_j` for every `i, j` in `1...N`. Then, the total number of orthogonal predicates generated by partitioning them (§3.1.1, Alg. 1) is exponential in `N`.

Several prior works have observed similar problems [25, 38, 95, 100, 115, 116]. Our optimizations reduce compile time and rule set size through domain-specific techniques, and by leveraging modern switch hardware that can support several match-action flow tables (§1.2, [12, 73]).

(A) Separating Forwarding and Measurement

We observe that the compilation of the joint "measure and forward" policy in §3.1.3 to single-stage rule tables is quite expensive because of the numerous parallel and sequential compositions involved:

```
(in_tagging >> forwarding >> out_tagging)
+ (in_capture)
+ (in_tagging >> forwarding >> out_capture)
```

However, notice that we can rewrite this final policy as follows:

```
(in_tagging + in_capture)
 >> forwarding
 >> (out_tagging + out_capture)
```

This construction preserves the semantics of the original policy provided `in_capture` policies do not forward packets onward through the data plane.[1] This new representation decomposes the complex compositional policy into a *sequential pipeline* of three smaller policies—which can be independently compiled and installed to separate stages of a multi-stage match-action table. Further, separating the forwarding and measurement rules into different tables enables *decoupling the updates* to query and forwarding policies, allowing them to evolve independently in the data plane.

(B) Optimizing Conditional Policies

The policy generated from the state machine (§3.1.2) has a very special structure, namely one that looks like a conditional statement: *if state=s1 then ... else if state=s2 then ... else if ....* A natural way to compile this down is through the parallel composition of policies that look like

```
state=s_i >> state_policy_i
```

This composition is expensive, because the classifiers of `state_policy_i` for all `i`, $\{C_i\}_i$, must be composed parallelly. We avoid computing these cross-product rule compositions as follows: If we ensure that each rule of C_i is specialized to match on packets disjoint from those of C_j—by matching on state `s_i`—then it is enough to simply *append* the classifiers C_i and C_j. This brings down the running time from $\Theta(N_i \times N_j)$ to $\Theta(N_i + N_j)$. We further compact each classifier C_i: we only add transitions to non-dead DFA states into `state_policy_i`, and instead add a default dead-state transition wherever a C_i rule drops the packets.

Now we describe the details of this optimization. First, we introduce a *switch case* syntactic structure in the Pyretic intermediate language:

```
switch (field):
  case v0 ⇒ p0
  ...
  case vM ⇒ pM
  default ⇒ actions
```

This means that the policy `p_i` is applied when the value of the field `field` is `v_i`, and if no values match, the default actions are applied. Since no two policies `p_i, p_j` act on the same packet (*i.e.*, the packet must differ in the value of `field`), this policy structure can be compiled as follows. Each policy `p_i` is independently compiled, and each rule of C_i is specialized by a match on the `field` value. The classifiers are concatenated with each other, and finally with a single rule that performs the default actions, to produce the

[1] We believe that this is not a strong restriction. None of our examples forwarded packets onward in the `in_capture` policy. If however `in_capture` outputs packets in the data plane, they would also be processed by the `forwarding`, `out_tagging` and `out_capture` parts of the rewritten policy.

final classifier. By getting rid of the cross-product of classifiers $C_0, C_1, ...C_M$, policy compilation now takes $N_0 + ... + N_M$ time instead of $N_0 \times ... \times N_M$.

Second, we introduce the policy structure `p else a`, where `p` is any policy and `a` is an unconditional list of primitive actions (*e.g.,* field modifications). The policy `p else a` takes an input packet and evaluates it by policy `p` first. If `p` does not drop the packet, the output packet of `p else a` is the same as that of `p`. If `p` drops the packet, then actions `a` are applied on the input packet. This structure is compiled by first compiling `p`, and checking each rule in the classifier to determine if it drops the packet. If so, the classifier is modified so the actions `a` are executed instead. This produces a classifier in just $N_p + 1$ time, processing any packets that `p` drops.

Now we compile the policy

```
switch (state):
  case s0 ⇒ transition from s0 to live states
  ...
  case sM ⇒ transition from sM to live states
  default ⇒ transition to dead state
else transition to dead state
```

which ensures that each state classifier is independently constructed (no cross-producting across states), and that dead state transitions are automatically included on any packets not processed by the switch case policy.

(C) Integrating Tagging and Capture Policies

The `tagging` and `capture` policies have similar conditional structure:

```
tagging =                capture =
  (cond1 >>  a1) +         (cond1 >>  b1) +
  (cond2 >>  a2) + ...     (cond2 >>  b2) + ...
```

Notice that the `tagging` and `capture` policies use the same set of conditions `cond_i:` a match on the DFA state and an edge predicate (§3.1.2). Rather than supplying Pyretic with the policy `tagging + capture`, which will generate a large cross-product, we construct a simpler equivalent policy:

```
combined =
  (cond1 >>  (a1 + b1)) +
  (cond2 >>  (a2 + b2)) + ...
```

(D) Pre-partitioning Predicates by Flow Space

In many queries, we observe that most input predicates are disjoint with each other, but predicate partitioning (§3.1.1) checks overlaps between them anyway. We avoid these checks by pre-partitioning the input predicates into disjoint flow spaces, and only running the partitioning (Alg. 1) within each flow space. For example, suppose in a network with `n` switches, we define `n` disjoint flow spaces `switch=1, ..., switch=n`. When a new predicate `pred` is added, we check if `pred & switch=i` is nonempty, and then *only* check overlaps with predicates intersecting the `switch=i` flow space. We have implemented pre-partitioning with switch-wise flow spaces, but other pre-partitions are possible. Note that since each disjoint flow space partitions the predicates independently, overlap detection can be easily parallelized.

(E) Caching Predicate Overlap Decisions

We avoid redundant checks for predicate overlaps by caching the latest overlap results for all input predicates. We index this cache by a hash on the string representation of the predicate's abstract syntax tree, and execute the predicate partitioning algorithm only on a cache miss. The cached overlap results for a predicate are reset to the latest values each time the predicate is broken up due to overlap with new incoming predicates (Alg. 1). Hence, explicit cache invalidation and re-population are unnecessary.

(F) Decomposing Query-Matching into Multiple Stages

Often the input query predicates may have significant overlaps: for instance, one query may count packets on `M` source IP addresses, while another counts packets on `N` destination IP addresses. By installing these predicates on a single table stage, it is impossible to avoid using up `M`×`N` rules, as there are packets that may satisfy any of the `M`×`N` source-destination combinations. However, by installing the `M` source matches in one table and `N` destination matches in another, we can reduce the total rule count to `M+N`. Grouping queries into separate stages allows us to emulate the action of *multiple disjoint DFAs* on the same packet—which can be independently and parallelly compiled to match-action rules in the usual way (§3.1.1-§3.1.3). These *logical* tables may then be mapped to *physical* tables on hardware [50, 91].

The key intuition of this optimization is to spread queries matching on *dissimilar* header fields into different DFAs, and hence different table stages. However, we should run as few DFAs as possible, so that we can reap the benefits of sharing match-action rules and packet DFA states among multiple *similar* queries. Further, the queries must fit into the limited number of table stages available on switches, and the limited capacity of rules available at each stage. To capture these tensions and find a "sweet spot" while spreading queries across stages, we write down the following optimization problem:

minimize: S = $\sum_j y_j$

variables: $q_{ij} \in \{0,1\}, y_j \in \{0,1\}$

subject to:

$\forall j:$ cost($\{q_{ij} : q_{ij} = 1\}$) $\leq$ rulelimit $* y_j$

$\forall i:$ $\sum_j q_{ij} = 1$

$S \leq$ stagelimit

Here the variable q_{ij} is assigned a value 1 if query `i` is assigned to stage `j`, and 0 otherwise. The variable y_j is 1 if stage `j` is used by at least one query and 0 otherwise. The constraints ensure, respectively, that (i) queries in a given stage are within the rule space available in that stage, (ii) every query is assigned exactly one table stage, and that (iii) the total number of stages is within the number of maximum stages supported by the switch. This integer optimization problem minimizes the number of used table stages, which is a measure of the latency and complexity of the packet-processing pipeline. Other optimization formulations are possible, *e.g.,* minimizing the total number of rules given a constraint on the number of stages.

Now we elaborate on the cost function in the first constraint of the optimization problem. It is in general difficult to know the exact rule space incurred by a set of queries together without actually doing the partitioning in Alg. 1. Instead, we *estimate* the *worst case* cost of grouping a set of queries in one stage, as follows. We define the *type* and *count* for each query as the set of header fields the query matches on, and the number of total matches respectively. In the preceding example with `M` source IP addresses and `N` destination IP addresses, the query types and counts would be

```
q1: ([srcip], M)
q2: ([dstip], N)
```

We write down a worst-case rule space estimate when combining *two* queries in one table stage:

```
cost ((type1, count1), (type2, count2)) :=
  case type1 == φ:
    count2 + 1
  case type1 == type2:
    count1 + count2
  case type1 ⊂ type2:
    count1 + count2
  case type1 ∩ type2 == φ:
    (count1 + 1) * (count2 + 1) - 1
```

```
  case default:
    (count1 + 1) * (count2 + 1) - 1
```

Further, notice that the type of the resulting query is `type1` $\cup$ `type2`, as predicate partitioning (Alg. 1) creates matches involving the union of the matched header fields in the two queries. Given the type and count of the result from grouping *two* queries, it is easy to generalize to *multiple* queries by iteratively applying this procedure to the queries in order, *e.g.,*

```
cost((type1, count1), (type2, count2), (type3, count3))
  = cost((type1 ∪ type2,
          cost((type1, count1), (type2, count2))),
         (type3, count3))
```

Readers may identify similarities between our optimization problem and formulations in prior compilers [50, 91] that map packet-processing programs to complex match-action pipelines. However, there are key differences in the formulation. First, our rule space cost function explicitly favors queries matching similar headers in one table, while penalizing groups of queries with very different header matches. In contrast, in [91] the table sizes are always multiplied independent of headers that the policies in the table match on. Second, the policies in prior works have pre-existing dependency structures, *e.g.,* control flow graphs, but there are no dependencies between groups of queries. Hence, our formulation must explore more possibilities than these prior works. These prior compilers could treat the output of our optimization as logical tables, and translate them to physical tables on switch hardware.

Reduction of bin-packing to rule-packing. Unfortunately, the optimization problem as posed above is NP-hard. It is straightforward to show that a version of the *bin-packing problem*, *i.e.*, minimizing the number of bins B of capacity V while packing n items of sizes $a_1, a_2, \cdots, a_n$, can be solved through a specific instance of the rule packing problem above. We construct n queries of the same type, with rule counts $a_1, \cdots, a_n$ respectively. We set the `rulelimit` to the size of the bins V, and `stagelimit` to the number of maximum bins allowed in the bin packing problem (typically n). Since all queries are of the same type, the rule space cost function is just the sum of the rule counts of the queries at a given stage. It is then easy to see that the original bin-packing problem is solved by this instance of the rule-packing problem.

First-fit heuristic. The first-fit heuristic we use is directly derived from the corresponding heuristic for bin-packing. We fit a query into the first stage that allows the worst-case rule space blowup to stay within the pre-specified per-stage `rulelimit`. The cost function above is used to compute the rule space cost of including a new query in a stage. We use a maximum of 10 logical stages in our experiments, with a 2000 rule limit per stage in the worst-case.

(G) Detecting Overlaps using Forwarding Decision Diagrams (FDDs)

To make intersection between predicates efficient, we implement a recently introduced data structure called *Forwarding Decision Diagram* (FDD) [95]. An FDD is a binary decision tree in which each non-leaf node is a test on a packet header field, with two outgoing edges corresponding to true and false. Each path from the root to a leaf corresponds to a unique predicate which is the intersection of all tests along the path. Inserting a new predicate into the FDD only requires checking overlaps along the FDD paths *which the new predicate intersects,* speeding up predicate overlap detection.

FDDs are variants of Binary Decision Diagrams (BDDs [13]), which are representations of boolean predicates which enable efficient predicate union and intersection operations. Several prior works have already used BDDs or their variants to reduce the time complexity of predicate intersection operations [95, 100, 115, 116]. However, most of our optimizations make predicate intersections *unnecessary*, by leveraging opportunities for pipelined processing of (possibly intersecting) packet predicates with multi-stage tables.

4.2 Performance Evaluation

Now we quantitatively evaluate how well the compiler optimizations and the system perform.

We implemented the query language, compilation algorithms and our optimizations (§2, §3, §4.1) on the Pyretic SDN controller [63] and NetKAT compiler [95]. The query language is embedded in Python, and the run-time system is a library on top of Pyretic. To build upstream query compilation, we use hassel-C [82], a library that implements header space analysis. It is possible to sample queried packets from switches using NetFlow; the samples are processed with `nfdump` [70]. The Pyretic run-time system sends switch rules to Open vSwitch [73] through OpenFlow 1.0 for single-stage flow tables, and the Nicira extensions [74] for multi-stage flow tables. We use `Ragel` [20] to compile string regular expressions, and evaluate our system using the `PyPy` compiler [84]. The source code of our implementation is freely and publicly available online [104], as are instructions to reproduce our numerical results [78].

Metrics. A path-query system should be efficient along the following dimensions:

1. Query compile time: Can a new query be processed at a "human debugging" time scale?
2. Rule set size: Can the emitted match-action rules fit into modern switches?
3. Tag set size: Can the number of distinct DFA states be encoded into existing tag fields?

There are other performance metrics which we do not report. Additional query rules that fit in the switch hardware tables do not adversely impact packet processing throughput or latency, because hardware is typ-

ically designed for deterministic forwarding performance.[2] The same principle applies to packet mirroring [81]. The time to install data plane rules varies widely depending on the switch used—prior literature reports between 1-20 milliseconds per flow setup [41]. Our compiler produces small rule sets that can be installed in a few seconds.

Experiment Setup. We pick a combination of queries from Table 2.1, including switch-to-switch traffic matrix, congested link diagnosis, DDoS source detection, counting packet loss per-hop per-path,[3] slice isolation between two IP prefix slices, and firewall evasion. These queries involve broad scenarios such as resource accounting, network debugging, and enforcing security policy. We run our single-threaded prototype on an Intel Xeon E3 server with 3.70 GHz CPU (8 cores) and 32GB memory. We first evaluate the benefits of our optimizations, and then show benchmarks with all optimizations enabled. Further, since it is possible to parallelize the compilation of policies in different match-action stages (*i.e.*, optimizations (A) and (F)), we report on the *maximum* compile time across stages.

Benefit of Optimizations

We evaluate our system on an emulated Stanford campus topology [2], which contains 16 backbone routers, and over 100 network ingress interfaces. We measure the benefit of the optimizations when compiling *all of the queries listed above* together—collecting over 550 statistics from the network. By injecting traffic into the network, we tested that our system collects the right packets (Fig. 1.8), extracts the right switch counters, and produces no duplicate packets.

The results are summarized in Table 4.1. Some trials did not finish as they ran out of memory, and are labeled "DNF." Each finished trial shown is an average of five runs. The rows are keyed by optimizations—whose letter labels (A)-(F) are listed in paragraph headings in §4.1. We enable the optimizations one by one, and show the *cumulative* impact of all enabled optimizations in each row. The columns show statistics of interest—compile time (absolute value and factor reduction from the unoptimized case), *maximum* number of table rules (ingress and egress separately) on any network switch, and required packet DFA bits.

The cumulative compile-time reduction with all optimizations (last row) constitutes three orders of magnitude: reducing the compile time to about 5 seconds, suitable for "interactive debugging" by a human operator [62, topic 11]. We enable (G) only for larger networks, where the time to set up the data structure is offset by fast predicate intersection. In this experiment, enabling FDDs increased the compile time to 98s.

Further, in the most optimized case, notice that the maximum number of rules required on any one switch fits comfortably in modern switch memory capacities, typically 2-4K rules [12,24,43]. We could not succeed

[2] Navindra Yadav. Personal communication, January 2016.
[3] We use the version of this query from §2.3, see `p2` there.

Enabled Opts.	Compile Time		Max # Rules		# State Bits
	Abs. (s)	Rel. (X)	In	Out	
None	> 7900	*baseline*	DNF	DNF	DNF
(A) only	> 4920	1.606	DNF	DNF	DNF
(A)-(B)	> 4080	1.936	DNF	DNF	DNF
(A)-(C)	2991	2.641	2596	1722	10
(A)-(D)	56.19	140.6	1846	1711	10
(A)-(E)	35.13	224.9	1846	1711	10
(A)-(F)	5.467	1445	260	389	16

Table 4.1: Benefit of optimizations on queries running on the Stanford campus topology. "DNF" means "Did Not Finish."

Network	# Nodes	Compile Time (s)	Max # Rules		# State Bits
			In	Out	
Berkeley	25	10.7	58	160	6.0
Purdue	98	14.9	148	236	22.5
RF1755	87	6.6	150	194	16.8
RF3257	161	44.1	590	675	32.3
RF6461	138	21.4	343	419	29.2

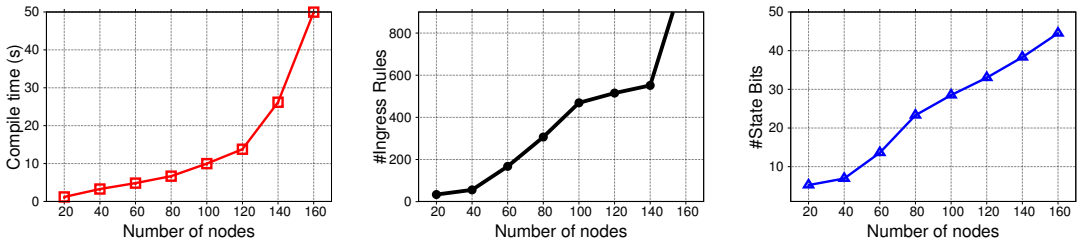
Table 4.2: Performance on enterprise and ISP (L3) network topologies when all optimizations are enabled.

in measuring the rule space usage of the unoptimized prototype as the run did not complete. However, we could *estimate* the rule space usage using the fact that the existence of overlapping rules in composed policies results in multiplication of rule set sizes of the constituent policies during composition (§4.1). Unfortunately, this *worst case* explosion applies to composing forwarding and query rules, because the Stanford forwarding rules match on destination IPs (the forwarding policy contains on average 240 rules per switch), but almost none of the queries do. Hence, disabling optimization (A) alone would lead to 240 * 260 * 389 rules per switch (maximum over the network). The overall estimated benefit in rule space usage from enabling all (A)-(F) optimizations is at least (240 * 260 * 389) / (240 + 260 + 389) $\approx$ 27300, or roughly five orders of magnitude.

The DFA state fits within tag fields like VLANs in all cases—it is 2 bytes at most. Notice that multi-stage query decomposition (F) increases the number of state bits since each DFA uses a disjoint set of packet bits, but significantly reduces rule space usage.

Performance on Enterprise and ISP Networks

We evaluate our prototype on real enterprise and inferred ISP networks, namely: UC Berkeley [10], Purdue University [101], and Rocketfuel (RF) topologies for ASes 1755, 3257 and 6461 [89, 98]. All optimizations are enabled. For each network, we report averages from 30 runs (five runs each of six queries). The results are summarized in Table 4.2. The average compile time is under 20 seconds in all cases but one; rule counts are within modern switch TCAM capacities; and DFA bits fit in an MPLS header.



(a) Compile time (b) Ingress rule-space (c) Number of DFA packet bits

Figure 4.1: Scalability trends on synthetic ISP topologies.

Scalability Trends

We evaluate how performance scales with network size, on a mix of five synthetic ISP topologies generated from Waxman graphs [111] and IGen, a heuristic ISP topology generator [86]. The edge probability between vertices u, v in a Waxman graph is given by the relationship $P(u,v) = \alpha * exp(-d/L\beta)$, where d is the distance between u and v, and L is a scale parameter. We generated connected Waxman graphs with four sets of parameters: $(\alpha,\beta) = (0.2,0.4),(0.3,0.3),(0.4,0.2),(0.5,0.15)$, at each network size. We further generated a fifth topology at each network size using IGen. We report average metrics from 30 runs, *i.e.*, six queries compiled to five networks of each size. The trends are shown in Fig. 4.1. The average compile time (see Fig. 4.1 (a)) is under ≈ 25 seconds until a network size of 140 nodes. In the same size range, the ingress table rule counts (see Fig. 4.1 (b)) as well as the egress (not shown) are each under 700—which together can fit in modern switch TCAM memories. DFA packet bits (see Fig. 4.1 (c)) fit in an MPLS header until 120 nodes. For networks of about 140 nodes or smaller, our query system supports interactive debugging—continuing to provide useful results beyond for non-interactive tasks. Among large ISP topologies mapped out in literature [98], each ISP can be supported in the interactive regime for queries at the granularity of 'points of presence' (PoPs), which are typically metropolitan locations.

We believe that our system significantly advances the state of "human time scale" network debugging. Operators today are often involved for hours in fixing network failures [36, 122], but iteratively debugging a network by issuing queries that install in a few seconds can substantially expedite this process. Yet there are also interesting future directions, as we discuss in the next chapter.

Chapter 5

Conclusion

> In programming, everything we do is a special case of something more general, and often we know it too quickly.
>
> *Alan Perlis, Epigrams of Programming*

We conclude this thesis with a summary of its contributions, future directions, and some final remarks.

5.1 Contributions

This thesis contributes by proposing a high-level abstraction and an efficient implementation to measure traffic flows over network paths—which we identify to be a common primitive underlying several measurement tasks.

The key challenge in measuring path-level traffic flow is that a switch processing a packet cannot in general know the prior or future path of the packet. This is because packets can be dropped or routed through unexpected paths downstream, and can appear at a given location through multiple prior paths. Observing a packet at every hop in its path by collecting it from switches is too expensive. And even if we pay the expense, joining traffic and packet forwarding information is fundamentally inaccurate. We address these challenges in three parts.

Path query language. We allow network operators to declaratively specify traffic flows of interest. Users can write regular expressions involving boolean packet predicates on packet locations and headers. Related sets of paths can concisely expressed as one regular expression aggregated by location or header fields of packets anywhere along the path, using a `groupby` construct, akin to a SQL groupby clause. Operators can ask for query results to be delivered as packet payloads, counts, or sampled records, and request to capture packets

either before, in the middle of, or after they traverse a queried path. We show how this language allows network operators to write several useful measurement queries, and show a detailed example of its utility in troubleshooting networks (§2).

Run-time system. We measure packet paths by having packets carry information about their own prior paths. Hence, when switches inspect packets, they already have accurate information about their prior paths. However, it is necessary to keep the amount of information on the packet small, because of bandwidth overheads, network-dictated packet size limits (MTUs), and restrictions on fast packet processing in switches (*i.e.*, match-action processing). Our key insight is to customize the path information according to the queries, and to exploit the regular expression structure of the queries. We encode the state of a Deterministic Finite Automaton (DFA) based on the queries into the packet. Further, switches inspect the DFA state, check conditions implied by the query, rewrite the packet state, and forward it in the usual way. When packets hit the accepting states of the DFA, only then are they captured, counted, or sampled (§3).

Compiler optimizations. A fundamental challenge is that multiple queries and forwarding rules may process the same set of packets, resulting in large switch rule sets and compile times. We built a number of compiler optimizations that leverage key domain-specific insights—especially significant opportunities for parallelism and pipelined processing on multiple switch tables—to reduce the number of switch rules and compile time to a practical realm. On an implementation on the Pyretic SDN controller and NetKAT compiler, our system compiles multiple queries in a few seconds, starting from an unoptimized implementation that takes more than two hours (§4).

Our prototype is open source and available online [104], and instructions are provided to reproduce our experimental results [78].

5.2 Future Directions

Identifying packet paths of interest directly in the data plane allows many new possibilities.

Network Performance Queries

While path queries allow declarative specifications of packet paths in networks, the general class of questions that operators would like to ask of their networks is much broader. In particular, there are generalizations of path queries to *performance* considerations of packets at small time scales. We discuss some examples below.

First, identifying packets traversing paths of interest allows us to expose *path performance information* for those paths. It is possible to leverage emerging switch hardware technologies such as In-band Network Telemetry (INT [55]), to enable switches to write switch-local state such as instantaneous queue sizes and processing timestamps into data plane packets. Hence, to understand path performance, we could use the path query and INT mechanisms—which are orthogonal—to mirror statistics of interest on packets corresponding only to the paths of interest.

Secondly, providing query interfaces to diagnose performance disruptions that occur at small time scales in low latency networks like data centers may be an important area for future work. For example, multiple communication flows may contend for the same switch port over a short, bursty period, and impact applications significantly: flows may suffer high completion times, and short flows may experience high packet losses [109]. Such events are challenging to diagnose with coarse metrics from switches available today, like link utilization averaged over a few minutes [21]. Is it possible to query fine-granularity performance criteria in a network using a high-level language? What primitives must be implemented in switch hardware to make it possible to implement such abstractions efficiently?

Improving Compiler Performance

Improving the performance of the compiler to scale to large networks, *e.g.,* data centers, is a ripe area for future work. We have a few proposals under this theme.

First, it is possible to exploit parallelism in compiler stages which are naturally parallel—such as detecting predicate overlaps in distinct partitions (parallel across partitions), compiling the conditional `switch` policy (parallel across DFA states) and compiling groups of queries running on separate DFAs (parallel across the flow table stages).

Secondly, the run-time system may choose to compile fewer queries than those supplied by the operator, based on network topology and traffic conditions. To be concrete, let us consider queries with `group` atoms.

Recall that the `group` atoms result in proactive substitutions of all combinations of grouped location or header fields, resulting in several "basic" queries (§3.1.4). However, some of these combinations may never be satisfied by any packet because of the network topology: for example, `ingress() ^ egress()` can only be satisfied by certain pairs of network interfaces. Similarly, if switches `S1` and `S3` are never connected, we don't need to install the basic query `in_atom(switch=S1) ^ in_atom(switch=S3)` resulting from the query `in_group(true, [switch]) ^ in_group(true, [switch])`.

In a similar vein, it is also possible to *reactively* expand `groupby` according to network traffic. For example, `in_group(switch=S1, [srcip])` may be expanded to only include queries for the source IP

addresses actually observed in the network. This is similar to the notions of *reactive specialization* in prior work [35], but here we generalize it to paths. Such pre-selection of queries will reduce the rule footprint on switches as well as the number of states required by the query DFAs.

Finally, we may be able to further optimize the FDDs we use [95] to evaluate predicate overlaps, using innovations to reduce Binary Decision Diagram (BDD) sizes and manipulation time. This could enable much faster predicate overlap computation, which constitutes a significant fraction of overall compile time.

Path-based Forwarding

Several applications that implement policies on packet paths [30, 47, 88, 97] may find it useful to flexibly forward packets based on prior path in the data plane. For example, an upstream failure that causes a backup path to be chosen (among others) may imply a specific downstream path (among others) [88]. As another example, forwarding packets through sequences of middleboxes may require forwarding packets based on the *prior* middleboxes traversed by the packet. Currently such applications pre-compute rules for each switch based on a static path-based global policy. Might it be useful to consider scenarios where traffic is dynamically routed according to network conditions, say attacks, congestion, or failures? How would one write path-based forwarding policies at a given switch, and have them co-exist with a "default" forwarding policy?

5.3 Final Remarks

Data networks should be managed well to sustain the Internet applications we love and enjoy today. It is essential for network operators to have fine-grained visibility into the underlying networks, so that they can plan capacity, troubleshoot problems when they occur, and enforce policies to secure the network. Such improved visibility into networks becomes increasingly important over time as Internet applications and user expectations grow.

We believe that high-level abstractions for networking are here to stay. As of this writing, there is very positive industry and academic outlook on building systems to measure and understand network data. We may well be on the brink of a shift in management practices for large networks, where operators are freely able to get network data they need to efficiently manage their networks, by writing concise programs in a high-level language.

Bibliography

[1] Sampled Netflow, 2003. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html.

[2] Mini-Stanford backbone topology, 2014. [Online, Retrieved February 17, 2016] https://bitbucket.org/peymank/hassel-public/wiki/Mini-Stanford.

[3] sFlow, 2016. sflow.org.

[4] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. H edera: Dynamic flow scheduling for data center networks. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2010).

[5] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proc. ACM SIGCOMM* (2014).

[6] AMAZON EC2 PRICING. [Online, Retrieved March 22, 2016] http://aws.amazon.com/ec2/pricing/.

[7] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic foundations for networks. In *Proc. ACM Symposium on Principles of Programming Languages* (2014).

[8] ANDREW LERNER. The cost of downtime, 2014. [Online, Retrieved February 17, 2016] http://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/.

[9] BARROSO, L. A., DEAN, J., AND HOLZLE, U. Web search for a planet: The google cluster architecture. *IEEE Micro* (2003).

[10] BERKELEY INFORMATION SERVICES AND TECHNOLOGY. UCB network topology. [Online, Retrieved February 17, 2016] http://www.net.berkeley.edu/netinfo/newmaps/campus-topology.pdf.

[11] BORDERS, K., SPRINGER, J., AND BURNSIDE, M. Chimera: A declarative language for streaming network traffic analysis. In *Proc. USENIX Security Symposium* (2012).

[12] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proc. ACM SIGCOMM* (2013).

[13] BRYANT, R. E. Graph-based algorithms for boolean function manipulation. *Proc. IEEE Transactions on Computers* (1986).

[14] CATALYST SWITCHED PORT ANALYZER (SPAN) CONFIGURATION EXAMPLE. [Online, Retrieved March 24, 2016] http://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/10570-41.html.

[15] CHEF SOFTWARE, INC. Chef. [Online, Retrieved March 15, 2016] https://www.chef.io/.

[16] CHEN, X., MAO, Y., MAO, Z. M., AND VAN DER MERWE, J. Decor: Declarative network management and operation. *ACM SIGCOMM Computer Communication Review* (2010).

[17] CISCO. Cisco visual networking index: Forecast and methodology, 2014-2019 white paper. [Online, Retrieved April 7, 2016] http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html.

[18] CLARK, D., REXFORD, J., AND VAHDAT, A. A purpose-built global network: Google's move to SDN. *Communications of the ACM* (2016).

[19] COHEN, J., REPANTIS, T., MCDERMOTT, S., SMITH, S., AND WEIN, J. Keeping track of 70,000+ servers: The Akamai query system. In *Proc. Large Installation System Administration Conference, LISA* (2010).

[20] COLM NETWORKS. Ragel state machine compiler. [Online, Retrieved February 17, 2016] http://www.colm.net/open-source/ragel/.

[21] CONFIGURING SNMP SUPPORT. [Online, Retrieved March 25, 2016] http://www.cisco.com/c/en/us/td/docs/ios/12_2/configfun/configuration/guide/ffun_c/fcf014.html.

[22] CONFIGURING WIRESHARK ON THE CATALYST 3850 SWITCH. [Online, Retrieved January 25, 2016] http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst3850/software/release/3se/network_management/configuration_guide/b_nm_3se_3850_cg/b_nm_3se_3850_cg_chapter_01000.html#concept_8EBE76A3D6DB46B79E7F0B6CFFE9FDF9.

[23] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: A stream database for network applications. In *Proc. ACM SIGMOD* (2003).

[24] CURTIS, A. R., MOGUL, J. C., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., AND BANERJEE, S. DevoFlow: Scaling flow management for high-performance networks. In *Proc. ACM SIGCOMM* (2011).

[25] D'ANTONI, L., AND VEANES, M. Minimization of symbolic automata. In *Proc. ACM Symposium on Principles of Programming Languages* (2014).

[26] DOBRIAN, F., SEKAR, V., AWAN, A., STOICA, I., JOSEPH, D., GANJAM, A., ZHAN, J., AND ZHANG, H. Understanding the impact of video quality on user engagement. In *Proc. ACM SIGCOMM* (2011).

[27] DUFFIELD, N. G., AND GROSSGLAUSER, M. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Networking* (June 2001).

[28] EVOLVEN. Downtime, outages and failures: Understanding their true costs, 2012. [Online, Retrieved March 22, 2016] http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html.

[29] FACEBOOK ENGINEERING BLOG. More details on today's outage. [Online, Retrieved March 21, 2016] https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919/.

[30] FAYAZBAKHSH, S. K., SEKAR, V., YU, M., AND MOGUL, J. C. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2014).

[31] FELDMANN, A., GREENBERG, A., LUND, C., REINGOLD, N., AND REXFORD, J. NetScope: Traffic engineering for IP networks. *IEEE Network* (2000).

[32] FELDMANN, A., GREENBERG, A., LUND, C., REINGOLD, N., REXFORD, J., AND TRUE, F. Deriving traffic demands for operational IP networks: Methodology and experience. *IEEE/ACM Trans. Networking* (June 2001).

[33] FORTZ, B., REXFORD, J., AND THORUP, M. Traffic engineering with traditional IP routing protocols. *IEEE Communications Magazine* (2002).

[34] FORTZ, B., AND THORUP, M. Internet traffic engineering by optimizing OSPF weights. In *Proc. IEEE INFOCOM* (2000).

[35] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A network programming language. In *Proc. ACM International Conference on Functional Programming* (2011).

[36] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proc. ACM SIGCOMM* (2011).

[37] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A scalable and flexible data center network. In *Proc. ACM SIGCOMM* (2009).

[38] GUPTA, A., VANBEVER, L., SHAHBAZ, M., DONOVAN, S. P., SCHLINKER, B., FEAMSTER, N., REXFORD, J., SHENKER, S., CLARK, R., AND KATZ-BASSETT, E. SDX: A software defined Internet exchange. In *Proc. ACM SIGCOMM* (2014).

[39] HAMILTON, J. The cost of latency. [Online, Retrieved April 7, 2016] `http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx`.

[40] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., MAZIÉRES, D., AND MCKEOWN, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2014).

[41] HE, K., KHALID, J., DAS, S., GEMBER-JACOBSON, A., PRAKASH, C., AKELLA, A., LI, L. E., AND THOTTAN, M. Latency in software defined networks: Measurements and mitigation techniques. In *Proc. ACM SIGMETRICS* (2015).

[42] HP SERVER AUTOMATION SOFTWARE. [Online, Retrieved March 23, 2016] `http://www8.hp.com/us/en/software-solutions/server-automation-software/`.

[43] HUANG, D. Y., YOCUM, K., AND SNOEREN, A. C. High-fidelity switch models for software-defined network emulation. In *Proc. Hot Topics in Software Defined Networks* (2013).

[44] HUANG, T.-Y., HANDIGOL, N., HELLER, B., MCKEOWN, N., AND JOHARI, R. Confused, timid, and unstable: Picking a video streaming rate is hard. In *Proc. Internet Measurement Conference* (2012).

[45] HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., SHENKER, S., AND STOICA, I. Querying the Internet with PIER. In *Proc. International Conference on Very Large Data Bases* (2003).

[46] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a globally deployed software defined WAN. In *Proc. ACM SIGCOMM* (2013).

[47] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a globally-deployed software defined WAN. In *Proc. ACM SIGCOMM* (2013).

[48] JEYAKUMAR, V., ALIZADEH, M., GENG, Y., KIM, C., AND MAZIÈRES, D. Millions of little minions: Using packets for low latency network programming and visibility. In *Proc. ACM SIGCOMM* (2014).

[49] JOHN HOPCROFT AND RAJEEV MOTWANI AND JEFFREY ULLMAN. *Introduction to automata theory: Languages and computation*. Pearson, 2001.

[50] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling packet programs to reconfigurable switches. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2015).

[51] KATTA, N., HIRA, M., KIM, C., SIVARAMAN, A., AND REXFORD, J. HULA: Scalable load balancing using programmable data-planes. In *Proc. ACM Symposium on SDN Research* (2016).

[52] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using Header Space Analysis. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2013).

[53] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2012).

[54] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2013).

[55] KIM, C., SIVARAMAN, A., KATTA, N., BAS, A., DIXIT, A., AND WOBKER, L. J. In-band network telemetry via programmable dataplanes, June 2015. Demo at Symposium on SDN Research, http://opennetsummit.org/wp-content/themes/ONS/files/sosr-demos/sosr-demos15-final17.pdf.

[56] KOHAVI, R., HENNE, R. M., AND SOMMERFIELD, D. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proc. ACM SIGKDD* (2007).

[57] LABS, P. Puppet enterprise. [Online, Retrieved March 15, 2016] https://puppetlabs.com/puppet/puppet-enterprise.

[58] LAKHINA, A., CROVELLA, M., AND DIOT, C. Characterization of network-wide anomalies in traffic flows. In *Proc. Internet Measurement Conference* (2004).

[59] LIN, C.-C., CAESAR, M., AND VAN DER MERWE, K. Toward interactive debugging for ISP networks. In *Proc. ACM Workshop on Hot Topics in Networking* (2009).

[60] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* (2008).

[61] MEDINA, A., TAFT, N., SALAMATIAN, K., BHATTACHARYYA, S., AND DIOT, C. Traffic matrix estimation: Existing techniques and new directions. In *Proc. ACM SIGCOMM* (2002).

[62] MILLER, R. B. Response time in man-computer conversational transactions. In *Proc. Fall Joint Computer Conference, Part I* (1968).

[63] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing software-defined networks. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2013).

[64] MOY, J. OSPF version 2. Internet Requests for Comments, 1998.

[65] NARAYANA, S., REXFORD, J., AND WALKER, D. Compiling path queries in software-defined networks. In *Proc. Hot Topics in Software Defined Networks* (2014).

[66] NARAYANA, S., TAHMASBI, M., REXFORD, J., AND WALKER, D. Compiling path queries. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2016).

[67] NELSON, T., FERGUSON, A. D., SCHEER, M. J., AND KRISHNAMURTHI, S. Tierless programming and reasoning for software-defined networks. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2014).

[68] NELSON, TIM AND YU, DA AND LI, YIMING AND FONSECA, RODRIGO AND KRISHNAMURTHI, SHRIRAM. Simon: Scriptable interactive monitoring for SDNs. In *Proc. ACM Symposium on SDN Research* (2015).

[69] NETWORK ADDRESS TRANSLATION (FAQ). [Online, Retrieved March 24, 2016] `http://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/26704-nat-faq-00.html`.

[70] NFDUMP TOOL SUITE. [Online, Retrieved February 17, 2016] `http://nfdump.sourceforge.net/`.

[71] OPENFLOW V1.0 SPECIFICATION. [Online, Retrieved March 30, 2016] `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf`.

[72] OPENFLOW V1.3 SPECIFICATION. [Online, Retrieved February 17, 2016] `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf`.

[73] OPENVSWITCH. [Online, Retrieved April 6, 2016] `openvswitch.org`.

[74] OPENVSWITCH NICIRA EXTENSIONS. [Online, Retrieved February 17, 2016] `http://git.openvswitch.org/cgi-bin/gitweb.cgi?p=openvswitch;a=blob;f=include/openflow/nicira-ext.h`.

[75] P4 LANGUAGE CONSORTIUM. [Online, Retrieved March 30, 2016] `http://p4.org/wp-content/uploads/2015/04/p4-latest.pdf`.

[76] PAN, P., SWALLOW, G., AND ATLAS, A. Fast reroute extensions to rsvp-te for lsp tunnels, 2005. RFC 4090 [Online, Retrieved April 6, 2016] `http://www.ietf.org/rfc/rfc4090.txt`.

[77] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud scale load balancing. In *Proc. ACM SIGCOMM* (2013).

[78] PATH QUERIES FOR INTERACTIVE NETWORK DEBUGGING. [Online, Retrieved February 17, 2016] http://www.cs.princeton.edu/~narayana/pathqueries.

[79] PC WORLD. Google says outage caused by traffic routing error. [Online, Retrieved March 21, 2016] http://www.pcworld.com/article/164904/Google_Says_Outage_Caused_By_Traffic_Routing_Error.html.

[80] PELLE, I., LÉVAI, T., NÉMETH, F., AND GULYÁS, A. One tool to rule them all: A modular troubleshooting framework for SDN (and other) networks. In *Proc. ACM Symposium on SDN Research* (2015).

[81] PERFORMANCE IMPACT OF SPAN ON THE DIFFERENT CATALYST PLATFORMS. [Online, Retrieved January 21, 2016] http://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/10570-41.html#anc48.

[82] PEYMAN KAZEMIAN. Hassel: Header space library. [Online, Retrieved February 17, 2016] https://bitbucket.org/peymank/hassel-public/wiki/Home.

[83] POSTEL, J. IP record route (Internet Protocol), 1981. RFC 791 [Online, Retrieved February 17, 2016] http://www.ietf.org/rfc/rfc791.txt.

[84] PYPY. [Online, Retrieved February 17, 2016] http://pypy.org.

[85] QAZI, Z. A., TU, C.-C., CHIANG, L., MIAO, R., SEKAR, V., AND YU, M. Simple-fying middlebox policy enforcement using sdn. In *Proc. ACM SIGCOMM* (2013).

[86] QUOITIN, B., VAN DEN SCHRIECK, V., FRANÇOIS, P., AND BONAVENTURE, O. IGen: Generation of router-level Internet topologies through network design heuristics. In *Proc. International Teletraffic Congress* (2009).

[87] RASLEY, J., STEPHENS, B., DIXON, C., ROZNER, E., FELTER, W., AGARWAL, K., CARTER, J., AND FONSECA, R. Planck: Millisecond-scale monitoring and control for commodity networks. In *Proc. ACM SIGCOMM* (2014).

[88] REITBLATT, M., CANINI, M., GUHA, A., AND FOSTER, N. FatTire: Declarative fault tolerance for software-defined networks. In *Proc. Hot Topics in Software Defined Networks* (2013).

[89] ROCKETFUEL: AN ISP TOPOLOGY MAPPING ENGINE. [Online, Retrieved February 17, 2016] http://research.cs.washington.edu/networking/rocketfuel/interactive/.

[90] SAVAGE, S., WETHERALL, D., KARLIN, A., AND ANDERSON, T. Practical network support for IP traceback. In *Proc. ACM SIGCOMM* (2000).

[91] SCHLESINGER, C., GREENBERG, M., AND WALKER, D. Concurrent NetCore: From policies to pipelines. In *Proc. ACM International Conference on Functional Programming* (2014).

[92] SCOTT SHENKER. The future of networking, and the past of protocols, 2011. [Online, Retrieved March 22, 2016] `http://opennetsummit.org/archives/oct11/shenker-tue.pdf`.

[93] SHIEH, A., SIRER, E. G., AND SCHNEIDER, F. B. NetQuery: A knowledge plane for reasoning about network properties. In *Proc. ACM SIGCOMM* (2011).

[94] SIMPLYHIRED. Network administrator salaries in San Jose, CA, 2016. [Online, Retrieved March 22, 2016] `http://www.simplyhired.com/salaries-k-network-administrator-l-san-jose-ca-jobs.html`.

[95] SMOLKA, S., ELIOPOULOS, S., FOSTER, N., AND GUHA, A. A fast compiler for NetKAT. In *Proc. ACM International Conference on Functional Programming* (2015).

[96] SNOEREN, A. C., PARTRIDGE, C., SANCHEZ, L. A., JONES, C. E., TCHAKOUNTIO, F., KENT, S. T., AND STRAYER, W. T. Hash-based IP traceback. In *Proc. ACM SIGCOMM* (2001).

[97] SOULÉ, R., BASU, S., MARANDI, P. J., PEDONE, F., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Merlin: A language for provisioning network resources. In *Proc. ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (Dec. 2014).

[98] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP topologies with Rocketfuel. In *Proc. ACM SIGCOMM* (2002).

[99] SUN, P., YU, M., FREEDMAN, M. J., AND REXFORD, J. Identifying performance bottlenecks in CDNs through TCP-level monitoring. In *Proc. of the First ACM SIGCOMM Workshop on Measurements Up the Stack* (2011).

[100] SUNG, Y.-W. E., LUND, C., LYN, M., RAO, S. G., AND SEN, S. Modeling and understanding end-to-end class of service policies in operational networks. In *Proc. ACM SIGCOMM* (2009).

[101] SUNG, Y.-W. E., RAO, S. G., XIE, G. G., AND MALTZ, D. A. Towards systematic design of enterprise networks. In *Proc. ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2008).

[102] TAMMANA, P., AGARWAL, R., AND LEE, M. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *Proc. ACM Symposium on SDN Research* (2015).

[103] TEIXEIRA, R., DUFFIELD, N., REXFORD, J., AND ROUGHAN, M. Traffic matrix reloaded: Impact of routing changes. In *Proc. Passive and Active Measurement Workshop* (2005).

[104] THE PYRETIC LANGUAGE AND RUN-TIME SYSTEM. [Online, Retrieved February 17, 2016] `https://github.com/frenetic-lang/pyretic`.

[105] TOMA, F. 48 best cloud tools for infrastructure automation. [Online, Retrieved March 15, 2016] `https://blog.profitbricks.com/48-best-cloud-tools-for-infrastructure-automation/`.

[106] UDDIN, M. Real-time search in large networks and clouds, 2013.

[107] VAHDAT, A., AL-FARES, M., FARRINGTON, N., MYSORE, R. N., PORTER, G., AND RADHAKRISHNAN, S. Scale-out networking in the data center. *IEEE Micro* (2010).

[108] VARDI, Y. Network tomography: Estimating source-destination traffic intensities from link data. *Journal of the American Statistical Association* (1996).

[109] VASUDEVAN, V., PHANISHAYEE, A., SHAH, H., KREVAT, E., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND MUELLER, B. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Proc. ACM SIGCOMM* (2009).

[110] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: Simplifying SDN programming using algorithmic policies. In *Proc. ACM SIGCOMM* (2013).

[111] WAXMAN, B. Routing of multipoint connections. *IEEE J. on Selected Areas in Communications* (1988).

[112] WIRESHARK. [Online, Retrieved February 17, 2016] `https://www.wireshark.org`.

[113] WU, W., WANG, G., AKELLA, A., AND SHAIKH, A. Virtual network diagnosis as a service. In *Proc. Symposium of Cloud Computing* (2013).

[114] YU, M., GREENBERG, A. G., MALTZ, D. A., REXFORD, J., YUAN, L., KANDULA, S., AND KIM, C. Profiling network performance for multi-tier data center applications. In *Proc. USENIX Symposium on Networked Systems Design and Implementation* (2011).

[115] YUAN, L., CHEN, H., MAI, J., CHUAH, C.-N., SU, Z., AND MOHAPATRA, P. Fireman: a toolkit for firewall modeling and analysis. In *Proc. IEEE Symposium on Security and Privacy* (2006).

[116] YUAN, L., CHUAH, C.-N., AND MOHAPATRA, P. ProgME: Towards programmable network measurement. In *Proc. ACM SIGCOMM* (2007).

[117] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic test packet generation. In *Proc. ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2012).

[118] ZHANG, H., LUMEZANU, C., RHEE, J., ARORA, N., XU, Q., AND JIANG, G. Enabling layer 2 pathlet tracing through context encoding in software-defined networking. In *Proc. Hot Topics in Software Defined Networks* (2014).

[119] ZHANG, Y., ROUGHAN, M., DUFFIELD, N., AND GREENBERG, A. Fast accurate computation of large-scale IP traffic matrices from link loads. In *Proc. ACM SIGMETRICS* (2003).

[120] ZHANG, Y., ROUGHAN, M., LUND, C., AND DONOHO, D. An information-theoretic approach to traffic matrix estimation. In *Proc. ACM SIGCOMM* (2003).

[121] ZHANG, HARVEST AND REICH, JOSHUA AND REXFORD, JENNIFER. Packet traceback for software-defined networks. Tech. rep., Princeton University, 2015. [Online, Retrieved April 6, 2016] `https://www.cs.princeton.edu/research/techreps/TR-978-15`.

[122] ZHU, Y., KANG, N., CAO, J., GREENBERG, A., LU, G., MAHAJAN, R., MALTZ, D., YUAN, L., ZHANG, M., ZHAO, B., AND ZHENG, H. Packet-level telemetry in large data-center networks. In *Proc. ACM SIGCOMM* (2015).