NSF Workshop Report on Programmable Networks

Organizers:
Anirudh Sivaraman, Xin Jin,
Vladimir Braverman, and Mohammad Alizadeh

October 1, 2019

1 Introduction

Programmable networks have emerged to fundamentally change the way we design, build, and manage computer networks. Programmable networks have been a goal for networking researchers for a long time now, and many believe that the time has finally come true with programmable network devices (e.g., network interface cards and switches) released by vendors and new advanced network applications developed by researchers and network operators. Due to the requirement of providing line-rate packet processing, programmable network devices have a limited amount of high-speed memory and perform a small amount of computation on every packet. It is a grand challenge for the community to rethink the full software stack of programmable networks, from exploiting sublinear algorithms and compact data structures to efficiently realize network applications with limited resources, to developing high-level programming frameworks to simplify application development and management, to designing new hardware architectures to enable programmability. This requires the effort of multiple research communities: networking, algorithms, programming languages, and computer architecture.

The NSF workshop on programmable networks brought together these different research communities to survey the state of the art in programmable networks, to identify key research challenges, and to exchange ideas. Our primary goal was to stimulate discussion between different communities engaged in research on programmable networks and to broaden the research community working on programmable networks. This report summarizes the workshop's discussions and is organized by the major themes discussed at the workshop.

Workshop format. The workshop was held in the Computer Science Department of New York University. The participants included a mixture of senior graduate students, postdocs, junior and senior faculty members, industry

and defense experts on computer networks, and NSF program managers. The workshop included

1. keynotes from distinguished researchers,
2. breakout sessions between participants discussing emerging research challenges in programmable networks,
3. a panel discussion between academia, industry, and government,
4. and a poster session featuring early-stage research in programmable networks

The workshop was held on October 25 and 26 2018 at New York University for a day and a half. The workshop website with more information is available here: `https://sites.google.com/view/programmable-networks-workshop/`

Report format. This report is broadly organized by the four main themes in the workshop, each of which had a breakout session devoted to it

1. applications and use cases
2. algorithms and formal methods
3. programming language and hardware design
4. testbeds, infrastructure, and education support

We also include insights from the keynotes when discussing each of the themes.

2 Applications and use cases

We now discuss applications and use cases that arise from ubiquitous programmability in networks. We discuss opportunities in the context of three different sets of industry actors that could potentially benefit from network programmability: network operators (e.g., Microsoft, Google, Facebook, AT&T), network equipment manufacturers (e.g., Arista, Dell), and chip vendors (e.g., Broadcom, Barefoot Networks, Mellanox).

2.1 The network operator's perspective

Network telemetry Troubleshooting outages and performance anomalies remains problematic today despite continued research in these areas. Network programmability provides operators with the ability to instrument different network locations to track what is happening to a packet during the course of its lifetime from leaving the sender to being received at the receiver. For instance, the In-Band Network Telemetry specification [5] allows switches to add queueing-related information to packet headers (e.g., queue depths at enqueue and dequeue and queueing latencies) that can then be read at the end hosts.

This allows us to retrieve rich time series data of queueing behavior at different points in the network, which can then be analyzed for anomalous behavior. Beyond this, network telemetry provides the ability to use network programmability *passively* to observe packets, without the risk of actively modifying packets incorrectly. In this sense, it provides network operators with an easier pathway to migrate towards the full use of programmability in networks.

Traffic Engineering Data-plane load balancing and traffic engineering algorithms (e.g., CONGA [6] and HULA [15]) allow switches to dynamically react to changes in network conditions (e.g., link utilization fluctuations to new flows arriving and departing) at much faster time scales relative to invoking the control plane. The use of programmable switches can enable the deployment of such traffic engineering algorithms, with the potential for relieving hotspots in the network.

Application-specific routing Programmable networks, whether switches or network-interface cards, can be used to enable application-specific routing. For instance, video traffic could be routed over a preferred route relative to bulk data transfers. Similarly, in the context of a datacenter, packets belonging to latency-sensitive user-facing Web requests can be routed over more lightly loaded paths relative to packets from background data backup jobs. Similarly, packets belonging to an application that is less tolerant to jitter can be routed over a more trustworthy, low-variability path.

Application-specific scheduling Programmable scheduling within networks can allow network operators to prioritize data belonging to particular application classes when apportioning scarce network bandwidth. Further, after dividing applications into classes (based on either strict priorities or weighted fair queueing), a class-specific scheduling algorithm could be used within each class (e.g., FIFO for reducing tail latency, SRPT for reducing job completion time). Recent work on programmable scheduling designs (e.g., PIFO [23], Eiffel [18], and Loom [24]) has shown that it might be feasible to enable programmable scheduling at high packet-processing rates.

Network security A programmable network device can be used as a means of implementing monitors to detect anomalous patterns of network traffic (e.g., the Threshold Random Walk algorithm for portscan detection [13]). After detection, the same device can be used for enforcement to filter out relevant traffic by installing appropriate rules to drop traffic. While such network security functions are implemented on software platforms as part of the move to network function virtualization (NFV) [21], the rise of network programmability allows us to augment traditional NFV-based security with monitors and filters spread out through the network on programmable switches and programmable network-interface cards. It could also allow us to replace a cluster of commodity processors programmed to carry out NFV functionality with a smaller number

of programmable network devices for the same functionality, potentially saving money and energy in the process.

Network-assisted applications Programmability within networks can be used to accelerate distributed machine learning training by using the switch to aggregate model updates from several distributed workers [19]. A programmable switch can also serve as a load-balancing cache (e.g., NetCache [12]) to reduce the incidence of hotspots among backend servers. In other work, a programmable switch is used to build a lock service in the style of ZooKeeper [11], for fast string matching [9], or for high-speed stream processing [10]. A programmable switch could also be used to potentially filter and aggregate high-volume streams of sensor data before they hit backend servers. These systems point to the possibilities of using a programmable switch to directly speed up end-to-end applications—as an add-on I/O accelerator for servers.

Other applications In areas, where low latency is paramount, a programmable network device can be used to directly respond to market data and execute data. As one example, the Arista 7124 FX features an on-board FPGA to offload such applications for the express purpose of high-speed algorithmic trading. Recent work has also looked at programmable networks as a means of implementing consistent snapshots across the whole network [25] and for accurate time synchronization [14].

2.2 The equipment manufacturer and chip vendor perspective

Programmable switches have the potential to significantly reduce turnaround time for equipment manufacturers (e.g., Arista, Cisco) who build systems/boxes around merchant silicion switching chips. In such cases, the equipment manufacturers can respond to customer requests faster, fix bugs faster, and customize the same programmable switching chips to different market segments by simply changing the data plane program on these switching chips [2]. In addition, a programmable switching chip allows an equipment manufacturer to customize their switching chip to a particular use case and leave out everything else (e.g., use all memory on a switch for MAC forwarding or IPv4 lookups).

On the flip side, getting used to writing the data plane program in addition to control plane software for a programmable switch requires additional work from the equipment vendor. Some of this is a one-time cost and will likely be amortized in steady state. However, even in steady state, with a programmable switching chip, there are now two maintenance costs: (1) the data plane program (say in P4) describing the switch's functionality and (2) the usual control plane software for the switch CPU to interact with the data plane program. In return, however, there is a smaller time to market for many features—provided the feature can indeed be supported by the programmable switching chip.

From the chip vendor's (e.g., Broadcom, Mellanox, Barefoot, Intel) perspective, a programmable switching chip can help simplify hardware design. The chip vendor can worry less about whether every one of thousand features has been provided for. Instead, if the chip were programmable, the chip vendor can concentrate on designing and realizing in silicon a handful of highly reusable primitives (e.g., a match-action table, action units, a programmable scheduler, and a programmable parser) that can be repeated over the area of the chip. This has the potential to reduce labor costs in hardware design in steady state.

Programmable switching chips might also provide a way to experiment with features without committing to them, for chip vendors, equipment manufacturers and network operators. Later, once more experience has been gained with using these experimental features, a later generation of a switching chip could harden the features in silicon, instead of making it programmable.

At the beginning however, the hardware design effort is probably increased by the fact that the designer needs to design generic and programmable elements that need to be reusable for many as-yet-unknown needs in the future. Additionally, in steady state, a programmable switching chip might incur more labor cost in place and route or physical design. This is because programmability results in an increased number of wires because inputs to computational units can come from more sources. These new wires need to be routed in silicon, placing an increased burden in the place-and-route/physical design stage of hardware design.

In summary, programmable switches have the potential for significant benefits for both equipment vendors and chip manufacturers. Unlike the network operator use cases, however, these benefits have nothing to do with performance, reliability, or better monitoring. On the other hand, these benefits have much more to do with simplifying the engineering of a product whose feature set continuously changes with time. At some level, these engineering benefits have little to do with switches in specific. Many other domains have gone the way of programmability (e.g., GPUs, DSPs, smart phones are all programmable) for similar engineering reasons that allow system implementors and app developers to iterate quickly. Programmable switches and NICs bring the same kind of engineering benefits to high-speed networking.

3 Challenges

We now discuss challenges that must be resolved to effectively realize the opportunities in the previous section. First, in a network setting, both in a public and a private cloud, a programmable device needs to be shared across multiple different tenants. However, this poses both mechanism and policy challenges. On the mechanism front, programmable switches lack support for running multiple tenant programs on the switch simultaneously. On the policy front, even if the mechanism did exist, it is unclear how to split up multiple resources across different tenants in a manner that's fair to each tenant, but still allows the tenant to use the network effectively for their own needs. The policy problem

gets more complicated as we pool together the different resources (e.g., memory, ALUs) of multiple distinct programmable switches and present them all to the tenants as a single resource pool.

Modularity is another area where much more work needs to be done. For instance, we could imagine a program where a chip vendor sells a programmable switching chip to an equipment vendor. The equipment vendor customizes the chip according to their understanding of their customers' needs, say using a P4 program. Then the customer should also be free to modify the chip in some restricted ways according to their own needs. This scenario isn't atypial; the Amazon FPGA-as-a-service F1 offering [1] has a similar flavor. Amazon offers Xilinx's FPGAs as a service, with an Amazon shell written in Verilog providing a scaffolding around the F1's customer's own Verilog code. However, neither the programmable switch hardware nor the programming languages support this kind of modularity that allows customers to compose their P4 programs with a baseline P4 program provided by their equipment vendor [4].

Both the modularity and the multi-tenant scenario concerns are significantly alleviated in the case of programmable NICs because they feature a more familiar hardware architecture centered around processors. A processor-centric architecture makes it easier to support both modularity and multi tenancy.

The programming model for many of these devices is non standard and is likely to pose its own challenges. For instance, in a high-speed programmable switch, programs either compile and can run at up to the full line or programs do not compile at all. This is in contrast to a substrate like a CPU or GPU where a program can run at reduced performance in exchange for doing more work per packet. Switches don't allow this graceful tradeoff between performance and expressiveness. This leads to unintuitive behavior where a certain program compiles, but minor modifications to it do not compile. Providing guidance to the data-plane programmer under such circumstances will ease the transition to programmable network infrastructure. In addition, developing hardware architectures that allow a smoother tradeoff between performance and programmability (e.g., dRMT [8]) would also alleviate some of this concern of "falling off a cliff" once a program's complexity exceeds some threshold.

The presence of encryption poses challenges to packet processing that inspects payloads, e.g., deep packet inspection. It is possible that approximate inference using the packet's headers can make up for the lack of access to the packet payload [20], but this is far from a perfect solution. Further, if the proxy operations on the packet's headers turn out to be more involved than the original operations on the packet's payload, the tradeoff may not be worth it after all. Approaches based on specialized forms of fully homomorphic encryption [22] hold promise, but they currently slow down packet processing significantly for the benefit of being able to work directly on encrypted data.

Programmable network devices can also be challenging from a security perspective. They open up the possibility that the data-plane program running on the switch or NIC can be exploited by crafting specific packets. For instance, in a system like Marple [17] that proposes an in-switch key-value store as a cache for a backing store, an attacker could easily craft an adversarial traffic pattern

that causes repeated cache misses and an increased load on the backing store. If the backing store is not equipped to handle this increased load, it could easily be overwhelmed and become a point of failure.

A programmable network also introduces new management concerns because, in addition to the control plane software, the network operator now has to maintain the data plane program as well. This requires an additional code base, additional tests, additional qualification before release, and the potential for more bugs. One way to alleviate these concerns is to deploy programmable network devices in specific clusters alone or to use them initially for passive functionality alone (e.g., measurement) that doesn't modify the packet headers.

4 Algorithms and formal methods

4.1 Sublineaar algorithms and sketches

The gap between P4 and real hardware models. The open-source P4 behavioral model and the real hardware models are different. Sometimes the gap can be substantial. For example, the Barefoot Tofino switch, an example of a real hardware model, has the following hardware limitations that are not captured by the behavioral model.

- It has limited processing stages.
- It has limited per stage stateful memory.
- For algorithms, the number of per-packet operations should be constant and involve a constant number of memory accesses.
- There is no shared memory. One possible design is that P4 switches allow shared memory, for instance, by letting multiple pipeline stages share memory. But precisely defining and then achieving consistency on such shared memory is challenging.
- It is hard to reprogram the table rules from the data-plane.
- There are also memory bandwidth limitations on hardware.

As a concrete example, it is infeasible to store the flow keys on the data plane. For instance, in some cases, we might need to keep the top K flow keys on the data plane. To achieve this, we might need some priority queue data structures on hardware. But these data structures cannot be supported by current hardware chips. Priority queues need varying (non-constant) number of operations for each packet and need log(K) operations in the worst case, where K is the size of the priority queue. There may be some approximate priority queue structures that can be supported on hardware, but the number of entries has to be small. Current method reports the keys to the control plane and may not be ideal in some applications.

Some of these limitations may not be fundamental, and it is possible they might be addressed in future generations of programmable switch hardware. Demands from algorithms can push hardware design changes, and hardware design can push the implementation of algorithms.

New hardware and software models. New hardware models can inspire new algorithm designs and new applications. One recent example is that P4 switches can use RDMA to access DRAM as external memory, where the switch acts as a cache [16]. DRAM memory cannot be directly built into switches because DRAM cannot support line rate. Yet, the switch hardware design can be changed if line rate is not required. Combining switches with external hardware can be a direction for other applications.

There are also opportunities in new models for software switches. Sublinear algorithms can be used to optimize software switches. During the workshop, we discussed whether memory-saving is still relevant in software switches and how to optimize per-packet operations.

Use cases of algorithms. One example is correctness monitoring, which is about how to verify the correctness of a protocol based on the history of the packets sent and received by it. For example, in the FTP protocol, one can verify whether the ports activated by users are correct. To achieve this, Bloom filter and its variants like counting Bloom filter can be good candidates. But we need to handle membership deletions in these use cases. Many sketching techniques can not only support insertions, but also deletions. We possibly can adopt such sketching techniques for correctness monitoring.

In the sliding window stream processing model, the algorithms are more complicated with more number of operations. It is difficult to support them in programmable data planes.

In the distributed setting of multiple switches, there are new types of measurement tasks, e.g., most frequent flows on the critical path and traffic statistics between multiple paths. New sketching tools can be used. For example, "mergeable sketching" techniques provide sketch data structures that can be collected from a distributed setting and be accurately merged. For network monitoring and measurement, splitting the measurement tasks between end-hosts and switches is a potential option. For flow-level statistics, switches can be a good place to monitor. For event-based triggering, end-hosts can accurately capture them.

Algorithmic challenges There are many algorithmic challenges, such as supporting more complex tasks and measurements on multiple dimensions. Finding hierarchical heavy hitters on multiple dimensions is a hard problem. But the workshop participants agreed that we still needed to figure out real use cases for primitives such as hierarchical heavy hitters. TCAM usage on switches can inspire new algorithmic designs. TCAM can provide fast wildcard matching, and algorithm designs can make use of it for further acceleration.

4.2 Algorithms for dynamic networks

Algorithms in programmable networks We discussed the set of algorithms that are impacted the most in programmable networks (dynamic networks). Scheduling and load balancing are the most relevant applications. In addition to network sharing, resource sharing within switches is important in the context of programmable switches. We discussed the right level of abstraction for performance-related algorithms. For an algorithmic perspective, stateful processing (e.g., registers) is key. If you remove stateful processing, it is not clear if a programmable switch is any different from a traditional one. Programmable networks allow functions to be distributed across switches in a more fine-grained way. Programmable switches can simulate a Turing machine if we do not consider performance. Instead of restricting algorithms based on hardware constraints, we discussed that perhaps algorithms should drive hardware design.

Engaging the algorithm community The algorithm community likes formal clean problems. To engage them, the networking community should formalize when an algorithm can actually run in a programmable network in an abstract model. Organizing tutorials on network algorithms is an effective way to bridge the gap between the algorithm and networking communities.

New algorithmic frameworks We discuss the question of whether programmable networks can lead to new algorithmic frameworks. In traditional algorithms, memory and time are expensive, but accessing memory is no different than time. Programmable networks have a different computational model. It is not just about space and time, but also about the memory access constraints and number of operations (e.g., whether the switch can update the state once per packet). In programmable networks, streaming algorithms are triggered by packets. For online algorithms with recourse, it sometimes requires to set the state not once but twice. Oblivious routing routes traffic without knowing traffic demands. With the capability of programmable networks, it is possible to define a clear routing problem. We can borrow tools from optimization theory and decompose solutions under the requirements that are not aligned with per-packet constraints. The PODC community is actively working on algorithms for distributed computation, but there is the gap between the algorithm community and the practice. In networking, we focus on distributed congestion, instead of distributed computation. We discussed what is the computational model (or restrictions) of programmable networks, what are canonical problems that we need to solve in this computational model, and whether we can apply solutions on distributed computation to programmable networks. We also discussed whether machine learning can help network management. The machine learning community can be engaged by creating good benchmarks and "bake offs". The general trend is taking optimization problems and formulating them as learning problems. An important question is what is specific about networking beyond optimizing average case vs. worst case. Many networking problems

have high dimensions. The ability to consider high dimensional signals is an interesting benefit of machine learning. Competitive algorithms in the theory community is a useful framework to define problems in programmable networks. If the operators have no knowledge of the workload, competitive algorithms can be useful tools to provide some form of optimal guarantees, e.g., for congestion control. It is a difficult problem to deal with cascading effects and failures.

4.3 Security and privacy

Control plane denial of server attack We start with SDN security. The goal is to enforce network-side security for SDN applications. The security threat to SDN applications is that under adversarial workloads, there can be too many packets being sent to the controller, which is essentially a denial of service attack to the control plane. Specifically, if the attacker knows which packets will be redirected to the controller, the attacker can easily craft such malicious workloads. The channel between the control and data planes is PCIe, which provides 40 Gbps bandwidth. It is much slower than the data plane speed. While there are features on some switches that allow the data plane to batch updates to the control plane, the control plane bandwidth is still quite limited.

Programmable switches might help to address this problem, as they can look at more packet headers and make better decisions to prevent such attacks. A motivating example is that in an enterprise network, we can enforce a policy like "if no one is interacting with my user screen, and if someone suddenly starts sending traffic, it is an indication of malware". If the sensor information is compiled into the packet header, then the switches can look at the sensor information to make decisions to enforce such policies.

Memory and management network Another challenge for switch-based solutions is that it is hard to maintain per-connection state. Programmable switches provide the flexibility for applications to use on-chip memory for stateful processing, which can be exploited as a threat vector (e.g., exhaust switch memory). Currently, operators use a separate management network to load programs onto switches. If an attacker controls the management network, the attacker can even wipe out configurations.

Encryption Encryption is not supported in the switch data plane. If only a few packets are encrypted, these packets can be processed in the control plane. In that way, encryption and decryption are done in the switch CPU. This solution is limited, and is only suitable for control messages. It cannot be applied to IPSec or DPI where we need to encrypt and decrypt every packet. Today, AES encryption can be done very fast on x86 (even faster than FPGA), because there is a native instruction for AES encryption. But there are no native instructions in P4 for encryption. The use cases are important. We need to draw a line and not implement everything on a switch. The switch is not meant for general-purpose processing like encryption.

The switch can support XORs. But encryption requires loops and needs to scramble the data, making it hard to perform on a switch. One possibility is to use whitebox crypto, i.e., scrambling of the data is translated into indexing of the tables. If the theory of whitebox crypto is made practical, it is possible to offload whitebox crypto into a switch. While switching to a new crypto algorithm that is more amenable to P4 (e.g., whitebox crypto) may work, it might be hard to deploy because changing the crypto algorithm is a big problem in practice.

Another approach is to take existing algorithms and change them into table-based approaches. For example, taking an algorithm like AES and turning it into table-based approaches. It is also possible to implement crypto as an instruction in the ISA as opposed to a program in P4, which can be done using externs.

There is some debate on why we need hop-by-hop encryption if we can already do end-to-end encryption and why P4 is the right way to implement this. If we are doing encryption on the edge in an NF, it is unclear what is the benefit of adding it in P4 vs. doing it in x86. P4 by itself doesn't do encryption on its own. We need additional extern-like support in P4 to do that. We can also do it with additional hardware support on a NIC or in the wire. We need to figure out what are the use cases for 6 Tbps encryption. Some people use MACsec instead of IPSec.

In-network processing One use case is in-network processing. One of the assumptions for in-network processing is that the switch can actually see the data. It is not true with end-to-end cryptography. Maybe we move bits that needed to be computed into unencrypted headers. It is interesting to see if we can use a homomorphic encryption family of algorithms. End-to-end encryption is mostly used for the web. In-network processing is mostly used in datacenters, where everyone is within the same organization. As such, end-to-end encryption may not be needed in the first place. There are some use cases like multi-tenant datacenters in public clouds, which use end-to-end encryption to protect tenants from adversaries.

Some control plane functions can also be moved to the data plane, such as DHCP. These functions may use part of the data plane resources such as chip space and cycles. But the benefit is that the equipment is already there, and such offloading does not incur extra equipment cost.

5 Language and hardware design

The main question discussed is whether the hardware needs to change. Programmable data planes are currently based on the match-action model. But the match-action model may not be the right model for all packet processing. For example, some congestion-control protocols like HotCocoa [7] cannot be described by P4 and implemented with match-action processing. It is possible to implement them on an FPGA for a NIC, but it does not map to the language constructs in P4. Also, for many scenarios, per-RTT operations are important,

not per-packet operations. For these scenarios, the model should be changed to focus on per-RTT operations. There are some predictions on future network hardware: state sharing will get harder, bandwidth will continue to increase, scheduling will be hard, and instruction sets will probably change.

Today's switches are designed to achieve line-rate. Vendors perform quite a few optimizations to hit the line-rate tests (e.g., multicast). CPUs have a more graceful performance degradation curve. Switches are not designed in the same way as CPUs, because the traffic pattern is not known and the switch is expected to handle any traffic pattern. Also providing line rate for any traffic pattern allows switches to prevent DoS attacks. There is also a need to provide deterministic performance guarantees in some use cases, e.g., performance isolation in multi-tenant clouds. Classic solutions like TDM might be useful for these scenarios. Tenants need SLAs, and want to guarantee they have the capacity they have provisioned for.

Programming data planes is fundamentally hard. We need a better interface between the user and the compiler. There are currently some domain-specific languages (DSLs) like P4. But even with P4, it is still difficult to develop programs. Current programmable packet processing is made up of two components: packet parsing and forwarding. Develops need to program different independent components in three different sub-DSLs: parsing, forwarding, and traffic management. There are opportunities in developing new DSLs or sub DSLs to simplify data plane programming. Yet, multiple DSLs also pose new questions to users as they need to figure out which DSL is most suitable to use for their particular use cases. There is a distinction between what you want out of the hardware and what it actually provides. DSLs can help bridge the gap between them.

6 Testbeds, infrastructure, and education

Currently there are no testbeds available to the CISE community. One option is to invest in a new infrastructure similar to CloudLab or to add programmable network devices to CloudLab itself. The testbed can have a mix of switches and NetFPGAs. P4-NetFPGA is available with licenses from Xilinx. The reference designs are open source. The backend compiler is owned by Xilinx. The cost is about 2K per board. CloudLab has Intel SGX. Adding programmable switches and NetFPGAs can be done in a similar way. The low-hanging fruit is to articulate the form of a testbed. We might use leverage with companies to help. Some research groups have already built testbeds in the scale of multiple switches and a few to tens of servers.

Campus networks are interesting as mini-enterprise networks. They have real users and may allow researchers to perform more extensive experiments than enterprise networks. But it is hard to scale outside of a single institution. It is difficult to build a large distributed testbed like GENI. The PEERING infrastructure might be leveraged. It can provide real data (of BGP) and is lightweight.

There are also organizations like CAIDA [3]. But anonymization makes CAIDA's data less useful. Within the context of a campus network, it might be possible to do something less draconian. Future research will likely be data driven. Both static and dynamic data will be useful. We should consider sharing data between academic institutions and collaborating with industry. There are also opportunities in education. We should consider having students experiment with network programmability as part of both an undergrad and grad networks curriculum.

Appendices

A Posters

1. Praveen Tamanna: Distributed Network Monitoring and Debugging with SwitchPointer
2. Naveen Kumar Sharma: Approximating Fair Queueing on Reconfigurable Switches
3. Brent Stephens: Your Programmable NIC Should be a Programmable Switch
4. Steve Ibanez: Towards P4 Programmable Traffic Management
5. Marco Canini: Scaling Machine Learning with In-Network Aggregation
6. Jialin Li: Pegasus: Load-Aware Selective Replication with an In-Network Coherence Directory
7. Mina Tahmasbi Arashloo: Enabling Programmable Transport Protocols on High-Speed NICs
8. Muhammad Shahbaz: Elmo: Source-Routed Multicast for Public Clouds
9. Nate Foster: p4v: Practical Verification for Programmable Data Planes
10. Kausik Subramanian: Synthesizing Data and Control Planes for Multi-tenant Networks
11. Daehyeok Kim: Generic External Memory for Switch Data Planes

B Participants

1. Shir Landau Feibish, Princeton University
2. Justine Sherry, Carnegie Mellon University
3. Sangeetha Abdu Jyothi, University of Illinois at Urbana-Champaign

4. Naveen Kr. Sharma, University of Washington
5. Mario Baldi, Cisco
6. Radhika Mittal, University of Illinois at Urbana-Champaign/Massachusetts Institute of Technology
7. Lavanya Jose, Stanford
8. Gordon Brebner, Xilinx
9. Aurojit Panda, New York University
10. Kausik Subramanian, University of Wisconsin at Madison
11. Arpit Gupta, University of California at Santa Barbara/Columbia University
12. Costin Raiciu, Universitatea Politehnica Bucuresti, Romania
13. Manya Ghobadi, Massachusetts Institute of Technology
14. Shriram Krishnamurthi, Brown University
15. Jack Brassil, NSF
16. Walter Willinger, NIKSUN
17. Srinivas Narayana, Rutgers University
18. Vincent Liu, University of Pennsylvania
19. Radhika Niranjan Mysore, VMware Research
20. Hongqiang Harry Liu, Alibaba
21. Mina Tahmasbi Arashloo, Princeton University
22. Dan Ports, Microsoft Research
23. Brent Stephens, University of Illinois at Chicago
24. Muhammad Shahbaz, Stanford University
25. John Marshall, Cisco
26. Soudeh Ghorbani, Johns Hopkins University
27. Jeongkeun Lee, Barefoot Networks
28. Praveen Tammana, Princeton University
29. Mihai Budiu, VMware Research
30. Mosharaf Chowdhury, University of Michigan at Ann Arbor

31. Ang Chen, Rice University
32. Nate Foster, Cornell University
33. Marco Canini, King Abdullah University of Science and Technology (KAUST), KSA
34. Ryan Beckett, Microsoft Research
35. Y. Richard Yang, Yale University
36. Darleen Fisher, NSF
37. Debmalya Panigrahi, Duke University
38. Yifei Yuan, Carnegie Mellon University
39. Jialin Li, University of Washington
40. Wenchao Zhou, Georgetown University
41. Alex Sprintson, NSF
42. Tracy Kimbrel, NSF
43. Tom Anderson, University of Washington/MIT
44. David Stern, DISA
45. Ori Rottenstreich, Orbs
46. Stephen Ibanez, Stanford University
47. Daehyeok Kim, Carnegie Mellon University
48. Alan Liu, Carnegie Mellon University

References

[1] Amazon EC2 F1 instances. `https://aws.amazon.com/ec2/instance-types/f1/`.

[2] Building a product using P4. `https://p4.org/assets/P4WS_2019/p4workshop19-final5.pdf`.

[3] Center for Applied Internet Data Analysis (CAIDA. `http://www.caida.org/home/`.

[4] Exposing Data Plane Programmability on Turn-Key Network Devices. `https://p4.org/assets/P4WS_2018/Mario_Baldi.pdf`.

[5] In-band Network Telemetry (INT) Dataplane Specification. `https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf`.

[6] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM*, August 2014.

[7] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker. Hotcocoa: Hardware congestion control abstractions. In *USENIX NSDI*, 2017.

[8] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 1–14, New York, NY, USA, 2017. ACM.

[9] T. Jepsen, D. Alvarez, N. Foster, C. Kim, J. Lee, M. Moshref, and R. Soulé. Fast string searching on pisa. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, pages 21–28, New York, NY, USA, 2019. ACM.

[10] T. Jepsen, M. Moshref, A. Carzaniga, N. Foster, and R. Soulé. Life in the fast lane: A line-rate linear road. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 10:1–10:7, New York, NY, USA, 2018. ACM.

[11] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *USENIX NSDI*, 2018.

[12] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*, 2017.

[13] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, 2004.

[14] P. G. Kannan, R. Joshi, and M. C. Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, pages 8–20, New York, NY, USA, 2019. ACM.

[15] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *ACM SOSR*, March 2016.

[16] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan. Generic external memory for switch data planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, 2018.

[17] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM*

Special Interest Group on Data Communication, SIGCOMM '17, pages 85–98, New York, NY, USA, 2017. ACM.

[18] A. Saeed, Y. Zhao, N. Dukkipati, E. W. Zegura, M. H. Ammar, K. Harras, and A. Vahdat. Eiffel: Efficient and flexible software packet scheduling. In *USENIX NSDI*, 2019.

[19] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[20] P. Schmitt, F. Bronzino, R. Teixeira, T. Chattopadhyay, and N. Feamster. Enhancing Transparency: Internet Video Quality Inference from Network Traffic. In *The 46th Research Conference on Communication, Information and Internet Policy*, 2018.

[21] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *ACM SIGCOMM*, 2012.

[22] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 213–226, New York, NY, USA, 2015. ACM.

[23] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *ACM SIGCOMM*, 2016.

[24] B. Stephens, A. Akella, and M. Swift. Loom: Flexible and efficient NIC packet scheduling. In *USENIX NSDI*, 2019.

[25] N. Yaseen, J. Sonchack, and V. Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 402–416, New York, NY, USA, 2018. ACM.