Sluice: Network-Wide Data Plane Programming

Vikas Natesh[1] Pravein Govindan Kannan[2] Anirudh Sivaraman[1] Ravi Netravali[3]
[1]New York University [2]National University of Singapore [3]University of California, Los Angeles

CCS CONCEPTS

• **Networks** → **Programming interfaces**; **Programmable networks**; **Network management**.

KEYWORDS

Network-wide programming; data plane

1 INTRODUCTION

The last several years have seen the emergence of programmable network devices including both programmable switching chips and programmable network interface cards (NICs). Along with the rise of x86-based packet processing for middleboxes and virtual switches, these trends point towards a future where the entire network will be programmable. The benefits of network programmability range from commercial use cases such as network virtualization [9] implemented on the Open vSwitch platform [11] to recent research works that implement packet scheduling [12], measurement [10], and application offload of niche applications on programmable switches [7, 8].

While the benefits of programmability are clear, they are difficult to reap because programming the network as a whole remains a challenge. Current programming languages target individual network devices, e.g., P4 for the Tofino [1] programmable switching chip and the Netronome SmartNIC [3]. However, at present, there is no unified programming model to express and implement general data plane functionality at the level of an entire network, without having to individually program each network device.

Prior work has looked at programming an entire network. In particular, Maple [13] was an early example of a network-wide programming model designed for OpenFlow switches. Maple automatically divides functionality between a stateless component running on switches and a stateful component running on the network's controller. However, this creates overhead as packets requiring stateful processing must be forwarded to the controller. SNAP [6] is a more recent example of network-wide programming; unlike Maple, it offloads stateful functionality to switches by leveraging stateful processing available in programmable switches while providing the operator with a view of one-big-switch (OBS) of persistent arrays. This abstraction is good at expressing network-wide policies that do not require explicit placement of packet processing code on particular devices, e.g., DNS tunnel detection in a LAN. However, to develop applications that do require such specific placement, a more fine-grained programming model is necessary. For instance, an operator may wish to run active queue management (AQM), ECN, and DCTCP [5] on specific switches, but not all. This is hard to achieve using SNAP. Further, SNAP does not provide abstractions to express queue-based measurement e.g., tracking an EWMA of queueing latencies on a particular switch. To summarize, Maple and SNAP cannot express programmable switch functionality where the network operator requires specific placement of code on specific devices, e.g., packet scheduling, congestion control, and load balancing.

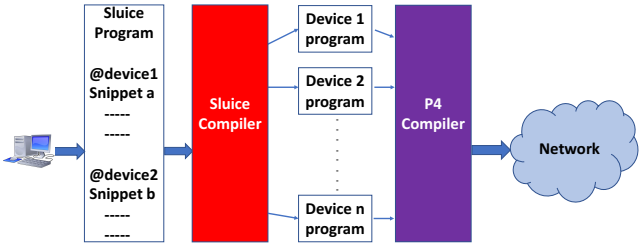

Figure 1: Sluice Workflow

This demo presents **Sluice**, a programming model that takes a network-wide specification of the data plane and compiles it into runnable code that can be launched directly on the programmable devices of a network. In contrast to prior network-wide programming models like SNAP and Maple that were focused on specific tasks (e.g., routing and security policies), Sluice aims to be more generic, but potentially at the cost of operator effort in specifying code placement. Sluice endows network operators with the ability to design and deploy large network programs for various functions such as scheduling, measurement, and in-network applications. The benefits of Sluice can be summarized as follows: (1) Sluice provides the same functionality as a per-device language like P4 but makes it easier to program the data plane of an entire network by abstracting device-specific architectural details like stateful ALUs, pipelines, etc., and (2) Sluice automatically reduces the amount of boilerplate code needed to write data plane functionality. For instance, the 8 line traffic matrix Sluice program we demonstrate translates into over 40 lines of P4 (excluding header/metadata/parser definitions and ipv4 forwarding P4 code). We demonstrate Sluice's functionality and ease of use via two examples: traffic matrix generation for network analysis and a streaming join-filter operation. Sluice is open-source and available at https://github.com/sluice-project/sluice.

2 SLUICE DESIGN

In the Sluice model, a network-wide program consists of high-level code *snippets* annotated by the operator to run on particular devices in a network. The code in each snippet is to be executed on packets arriving at its corresponding device. Snippets support a variety of operations: read-from/write-to packets; arithmetic using packet/meta data, local variables, or stateful register arrays; and control flow statements. To handle custom packet headers not supported by default (Ethernet/IP/UDP/TCP), users may define packet

SIGCOMM Posters and Demos '19, August 19–23, 2019, Beijing, China

ACM ISBN 978-1-4503-6886-5/19/08…$15.00
https://doi.org/10.1145/3342280.3342343

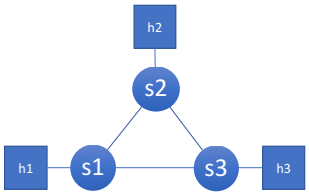

Figure 2: Topology For Traffic Matrix Demo

header declarations similar to C structs. An optional annotation in the packet declaration defines the parser condition for these user-defined headers (for example, see packet p in Figure 4). Sluice programs may also import device-specific variables/attributes for use in code snippets. Sluice also lets the programmer restrict snippets to operate on specific flows or IP address ranges.

Figure 1 describes the Sluice workflow. The compiler translates each snippet of a sluice program into a device-specific program. After initial parsing, lines of code in the snippet are decomposed into a directed acyclic graph (DAG) that maps dependencies between variables in each snippet. This graph is then passed to the backend of the compiler that generates the corresponding P4 program for that device, e.g., the P4 Behavioral Model [4] or Tofino [1].[1]

3 DEMONSTRATIONS

3.1 Traffic Matrix

Figure 2 displays the Mininet [2] network topology used for our traffic matrix demo. Packets are sent over UDP from each host to all other hosts according to a Poisson traffic model with mean inter-arrival time of 0.5 seconds. The code below is our Sluice program with a single snippet *traffic_example* that is launched on all switches of the network. To run the Mininet emulation, the user passes the Sluice program and network topology to the compiler. The compiler generates P4 code to run on each switch as well as control plane table entries for routing packets through the topology.

```
import device psa;
packet p: udp(srcPort:1234)
  nhops : bit<32>;


@ bmv2 : ;
snippet traffic_example()
  persistent cnt : bit<32>[10];
  cnt[psa.ingress_port] = cnt[psa.ingress_port] + 1;
  p.nhops = p.nhops + 1;
```

This demo shows how a simple Sluice program can be used to measure link usage for a specific UDP flow (srcport 1234) across the network. Each packet *p* contains a custom header *nhops* that is incremented each time the packet enters a switch to inform the receiving host of the number of hops the packet took. Each switch maintains a stateful register counter *cnt*, indexed by the switch ingress port, that tracks how many packets have entered through that ingress port. Aggregated over all switches, these counters represent a matrix measuring each link's usage in the network at a given time. This matrix (residing on the whole network) is then queried once every second from the control plane to generate time-series plots of packet rate for each link. Figure 3 displays the cumulative histogram of packet rates on link s1-s3 after collecting data for 15 minutes. The expected CDF of packet rates *Poisson*($\mu = 2$ packets/sec) is also plotted to validate the Sluice translation.

[1]Currently we only support the P4 Behavioral Model

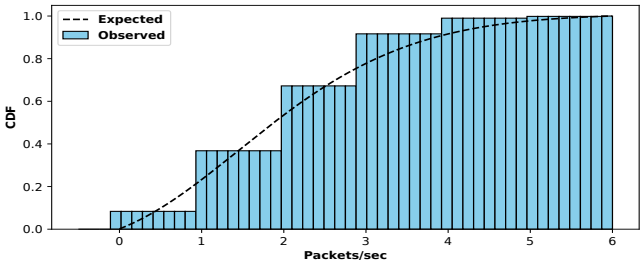

Figure 3: CDF of Packet Rate on link s1-s3

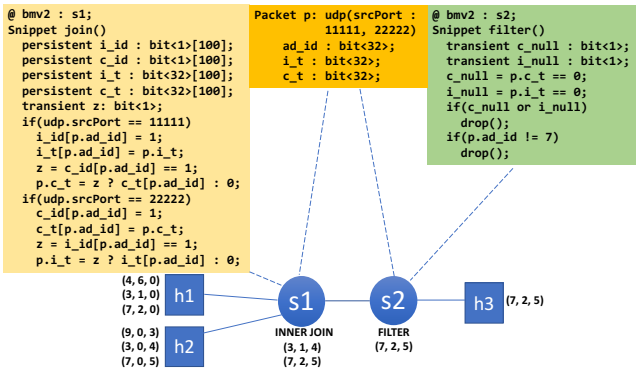

Figure 4: Streaming example topology, data flow, and code placement on switches

3.2 Stream processing

This example demonstrates a simple join-filter operation between two streams of tuples. A stream is an unbounded table where a packet represents a tuple of data (*ad_id, impression_time, click_time*) enclosed in a custom header. The topology in Figure 4 describes the data flow and shows how an operator query runs on the switches of the network. Host 1 sends a stream of ad impressions while Host 2 sends a stream of ad clicks. The two streams are joined on the *ad_id* field at s1 and filtered on the *ad_id* field at s2 and the result is sent to h3.

4 FUTURE WORK

An optimizing Sluice compiler. We envision using the dependency DAG (§2) to provide several automatic optimizations and code transformations. For example, it is possible that certain lines of code in a snippet cannot be run on the device annotated by the operator, e.g., programmable switching chips have limited support for floating point. or complex string operations. Code containing such features must be moved to the control plane or an end host while at the same time, preserving the original program semantics intended by the operator. Doing this automatically would free the Sluice programmer from reasoning about these semantics.

Supporting multi-tenancy. Another area of future work is allowing Sluice to support multiple tenants with their own Sluice programs running on their own virtual networks overlayed on the same physical topology. If each tenant wants to run their own network-wide program on their virtual topology, the network operator will need to merge all these into one data plane implementation that runs on the entire physical network. Extending Sluice to support this multi-tenancy use case would allow us to provide the same benefits to the data plane that multi-tenant network virtualization [9] provided for the control plane.

REFERENCES

[1] Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino. Accessed: 2019-07-02.

[2] Mininet. http://mininet.org/. Accessed: 2019-07-02.

[3] Netronome SmartNIC. https://www.netronome.com/products/smartnic/overview. Accessed: 2019-07-02.

[4] P4 Behavioral Model. https://github.com/p4lang/behavioral-model. Accessed: 2019-07-02.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[6] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM.

[7] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. Netchain: Scale-free sub-RTT Coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 35–49, Berkeley, CA, USA, 2018. USENIX Association.

[8] 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 *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[9] T. Koponen et al. Network Virtualization in Multi-tenant Datacenters. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 203–216, Berkeley, CA, USA, 2014. USENIX Association.

[10] 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.

[11] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association.

[12] 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 *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 44–57, New York, NY, USA, 2016. ACM.

[13] A. Voellmy et al. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *Proceedings of SIGCOMM*, 2013.