Sluice: A Network-Wide Model for Programmable Networks

Vikas Natesh[1] Anirudh Sivaraman[1] Ravi Netravali[2]
[1]New York University [2]University of California, Los Angeles

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 implemented on Open vSwitch to more recent projects that implement packet scheduling, measurement, and application offload of niche applications on programmable switches.

While the benefits of programmability are clear, it is still difficult to program the network as a whole. Current programming languages target individual network devices, e.g., P4 for the Tofino programmable switching chip and the Netronome programmable NIC. However, at present, there is no unified programming model to express and implement general functionality at the level of an entire network, without having to individually program each network device. As an analogy, data scientists prefer using a cluster-wide framework such as Apache Spark or Apache Hadoop that automatically handles scheduling, fault tolerance, and communication between different cluster machines, instead of a framework like MPI that requires the programmer to explicitly program each machine?s computation and communication.

Maple was an early example of a network-wide programming model designed for OpenFlow switches. Maple automatically divided functionality between a stateless component running on switches and a stateful component running on the network?s controller. SNAP is a more recent example of network-wide programming; unlike Maple, it additionally offloads stateful functionality to switches by leveraging stateful processing available in several programmable switches. However, both Maple and SNAP cannot express programmable-switch functionality that affects network performance at fine time scales, e.g., packet scheduling, congestion control, fine-grained measurement of microbursts, and load balancing. For these reasons we have developed Sluice, a programming model that takes a high-level specification of a network program and compiles it into runnable code that can be launched on the programmable devices of network.

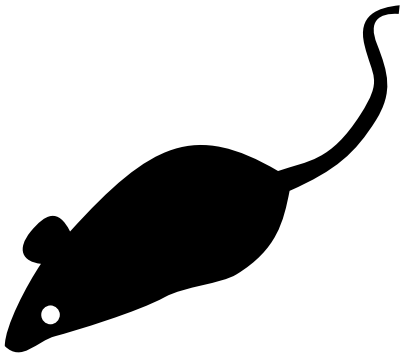

Figure 1: test

Sluice endows network operators with the ability to seamlessly design and deploy large network programs for various functions such as scheduling, measurement, and application offloading. We demonstrate Sluice's functionality and simplicity of use via two examples: traffic matrix generation for network analysis and a streaming join-filter operation.

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 data, local variables, or stateful register arrays; control flow statements. To handle computation on custom packet headers not supported by default (ip/tcp/udp/eth), users may define packet declarations similar to C structs. An optional attribute in the declaration, the parser condition, allows the user to restrict snippets to operate on specific flows or IP address ranges. Sluice programs may also import device-specific variables/attributes for use in code snippets. Overall, Sluice provides equivalent functionality of P4 but with an ease of use amenable to programming a whole network.

The compiler translates each snippet of a sluice program into a device-specific program. After initial parsing, a snippet is decomposed into a directed acyclic graph (DAG) that maps temporal dependency relations among all variables used in that snippet. This graph is then passed to a backend that

generates the corresponding P4 program. Our ongoing work has focused on using the dependency DAG to provide several optimizations. For example, it is possible that certain lines of code in a snippet cannot be run on the device annotated by the operator. Since programmable switching chips do not support floating point operations, code containing floating point ops must be moved to the control plane or end host while preserving the original program semantics intended by the operator.

3 DEMONSTRATIONS

3.1 Traffic Matrix

```
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;
```

Figure 2 displays the Mininet network topology and simulation components. Packets flow over UDP between virtual hosts, through a network of virtual programmable switches. The codelet in Figure 3 is a Sluice program with a single snippet 'traffic_example' that is launched on all switches of the network. To run the simulation, 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.

This demo shows how a very simple Sluice can be used to program each switch to measure link usage and hop counts for a specific flow of user-defined packets on UDP srcPort 1234. Each packet 'p' contains a custom header 'nhops' that will be used by the host to maintain a time-series of hop counts. Specifically, nhops is incremented each time the packet enters a switch, i.e. on each hop. Each switch maintains a stateful register counter 'cnt', indexed by switch ingress port, that tracks how many packets, on the particular UDP flow 1234, have entered through that ingress port. Aggregated over all switches, these counters data represent a matrix measuring each link's usage in the network. This matrix (residing on the whole network) is then queried regularly from the control plane to generate time-series plots of link-utilization and queue depth.

3.2 Streaming Algorithms

This example demonstrates a simple join-filter operation between two streams. 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.

```
packet p: udp(srcPort : 11111, 22222)
  ad_id : bit<32>;
  i_time : bit<32>;
  c_time : bit<32>;

@ bmv2 : s1;
snippet join()
        persistent impr_adId : bit<1>[100];
        persistent click_adId : bit<1>[100];
        persistent impr_time : bit<32>[100];
        persistent click_time : bit<32>[100];
        transient join_test: bit<1>;
        if(udp.srcPort == 11111)
                impr_adId[p.ad_id] = 1;
                impr_adId[p.ad_id] = p.i_time;
                join_test = click_adId[p.ad_id] == 1;
                p.c_time = join_test ? click_time[p.ad_id] : 0;
        if(udp.srcPort == 22222)   // check if stream is clicks
                click_adId[p.ad_id] = 1;
                click_adId[p.ad_id] = p.c_time;
                join_test = impr_adId[p.ad_id] == 1;
                p.i_time = join_test ? impr_time[p.ad_id] : 0;

@ bmv2 : s2;
snippet filter()
        transient click_invalid : bit<1>;
        transient impr_invalid : bit<1>;
        click_invalid = p.c_time == 0;
        impr_invalid = p.i_time == 0;
        if(click_valid or impr_valid)
                drop();
        if(p.ad_id != 10)
                drop();
```

The MySQL query corresponding to this example is shown below.

```
SELECT ad_id, impr_time, click_time
FROM impressions i
INNER JOIN clicks c ON i.adId = c.adId
WHERE ad_id = 10;
```

4 REFERENCES