

Sluice: A Network-Wide Model for Data Plane Programming

by

Vikas Natesh

Advisor : Dr. Anirudh Sivaraman

Examiner : Dr. Aurojit Panda

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

May 2019

Abstract

Networking devices have traditionally been thought of as fixed-function devices that perform the single task of packet switching very well. However, in recent years, researchers have developed network devices that can be explicitly programmed to perform complex packet processing tasks. These devices, which include routers, network interface cards (NICs), and middleboxes can support applications such as load balancing, heavy-hitter detection, and improved packet scheduling. While greater control over each device allows for better visibility into the network, it is still difficult to program the network as a whole. In the current state, network operators are required to program each individual device using a language such as P4 or Micro-C in order to express a network-wide computation. This thesis presents 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. We describe the design and language features of Sluice and compare it side-by-side to equivalent P4 translations. Finally, we run through several demos where Sluice is used to program a virtual network of programmable switches and hosts in a Mininet emulator.

Contents

1	Introduction	5
2	Sluice Overview	7
3	Design and Implementation	8
3.1	Imports	8
3.2	Packets	9
3.3	Snippets	11
3.3.1	Device Annotations	11
3.3.2	Variable Declarations	11
3.3.3	Control Blocks	14
3.4	Dependency DAG	20
4	Demonstrations	21
4.1	Sluice Simulator	21
4.2	Traffic Matrix	22
4.3	Stream processing	24
5	Future Work	27
5.1	An optimizing Sluice compiler.	27
5.2	Supporting multi-tenancy.	29
6	Appendix	30
6.1	Sluice grammar	30
6.2	Sluice Compiler Source Code	32

List of Figures

1	Figure 1 - Sluice Workflow	6
2	Figure x : Simple Arithmetic Example	16
3	H1 sends packet p with result initialized to 0 16	
4	H2 Receives p with a result header of 14	16
5	Figure x : Conditional Statement P4 Translation	19
6	Dependency DAG for Listing [cite]	21
7	Topology For Traffic Matrix Demo	22
8	CDF of Packet Rate on link s1-s3	23
9	Streaming example topology, data flow, and code placement on switches . . .	24

Listings

1	Sluice Parser Condition	10
2	P4 Ingress Control Block	10
	Name	16
	Name	17

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

While the benefits of programmability are clear, they are difficult to reap because programming the network as a whole remains challenging. 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 data plane functionality at the level of an entire network, without having to individually program each network device.

Prior work has looked at the problem of programming at the level of an entire network. In particular, Maple [cite] 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 [cite] is a more recent example of network-wide programming; unlike Maple, it additionally offloads stateful functionality to switches by leveraging stateful processing available in 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. In other words, Maple and SNAP generate highly optimized code, but are restricted in what they can express.

This thesis presents **Sluice**, a programming model that takes a network-wide specification

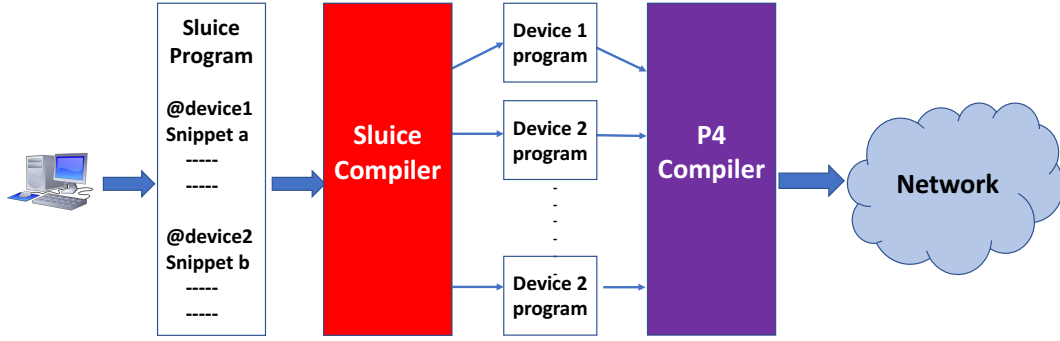


Figure 1 - Sluice Workflow

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 general, but potentially at the cost of quality of generated code. Sluice endows network operators with the ability to design and deploy large network programs for various functions such as scheduling, measurement, and application offloading. 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 (which we demonstrate later) translates into over 200 lines of P4 code.

2 Sluice Overview

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 headers, local variables, or stateful register arrays; and control flow statements. To handle computation on custom packet headers not supported by default (eth/ip/tcp/udp), users may define packet header declarations similar to C structs. An optional annotation in the declaration, the parser condition, automatically generates a header parser for these user-defined headers. 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, for example bmv2 or Tofino.¹

If successfully compiled into P4, Sluice programs are guaranteed to run at line rate on the switch. However, the need for line rate performance precludes the ability to perform looping in Sluice such as `for` and `while`. Although looping semantics can be implemented by recirculating packets back into the ingress pipeline of the switch (to repeat some computation)[cite p4-14 spec], this significantly reduces throughput. Nonetheless, fairly complex algorithms can still be expressed without loops. This is discussed further in the demonstrations section.

¹Currently we only support bmv2 but plan to support more devices.

3 Design and Implementation

```
program:
    imports packets snippets
    | packets snippets
```

The Sluice compiler is built using the Rust system programming language, chosen for its memory safety, variable lifetime management, and high performance. This section details the language specification. The following conventions are used:

- The GNU Bison grammar is used to display production rules. Boxes containing the grammar are shaded light-green.
- Terminal symbols are indicated in caps and constant string tokens are indicated in double quotes.
- A vertical bar, |, separates options from which exactly one must be selected.
- Curly braces, colons, and semicolons ending statements have been omitted for clarity.
- The 'Empty' keyword in the grammar indicates an empty string.
- Symbols marked as IDENTIFIER or *_ID represent names of files, variables, packets, snippets, packet fields, etc.

3.1 Imports

```
imports:
    imports import
    | import
    ;
import:
    "import device" IMPORT_ID
    ;
```

The `import_id` represents the name of an external file in which an operator may declare several target-specific fields. These fields can then be read and used within code snippets. For instance, the `bm2` target supports access to the enqueue depth of the virtual switches

in a topology. By default, Sluice supports the P4 portable switch architecture (PSA) fields, defined in the "psa.np" file.

```
device psa {
  timestamp_rx : bit <48>;
  timestamp_ingress : bit <48>;
  timestamp_egress : bit <48>;
  timestamp_tx : bit <48>;
  ingress_port : bit <9>;
  egress_port : bit <9>;
  packet_length : bit <32>;
  enq_qdepth : bit <32>;
  deq_qdepth : bit <32>;
}
```

3.2 Packets

```
packets:
    packets packet
    | packet
    ;
packet:
    "packet" PACKET_ID packet_base packet_parser_condition packet_fields
    ;
packet_base:
    "tcp" | "udp" | "ipv4" | "ethernet"
    ;
packet_fields:
    packet_fields packet_field
    | packet_field
    ;
packet_field:
    IDENTIFIER var_type type_qualifier
    ;
packet_parser_condition:
    IDENTIFIER value
    | Empty
    ;
```

Packets are declared similarly to C structs. Within a packet declaration, users can define custom packet headers which can be used for computation in code snippets. In addition, all (eth/ip/udp/tcp) header fields are supported by default. The **packet** keyword is followed by a `packet_base` and an optional `packet_parser_condition`. These two attributes help Sluice automatically generate a P4 packet header parser. The `packet_base` is a string of either *tcp*, *udp*, *ipv4*, or *ethernet* and represents the header type on which to parse the packet. Meanwhile, the parser condition defines the header field and value to match incoming packets

on. Packets that do not pass the parser condition are immediately sent to the ingress pipeline. However, to restrict the operation of Sluice snippet code to only those packets that passed the condition, a P4 `if` statement is automatically generated in the ingress control block. In this way, the parser condition allows the operator to conditionally perform computation on different flows.

In the example below, the packet p contains a single field $nhops$ that is used by the operator to maintain a hop count on each packet. It is parsed on the `udp srcPort` field. The corresponding P4 header parser and control block are also shown. In particular, the `parse_udp` parser shows that headers of packet p ($nhops$ in this example) are extracted only when `srcPort = 1234`. In the ingress block, the `if` statement on lines 2-4 ensures that P4 code corresponding to Sluice snippets are only executed on packets that have `udp.srcPort = 1234`. All other packets are sent to the routing table `ipv4_lpm`, provided that they have a valid `ipv4` header i.e. they are simply routed and do not affect router state.

Sluice Parser Condition

```
packet p: udp(srcPort:1234) {
  nhops : bit<32>;
}
```

P4 Ingress Control Block

```
1 control ingress {
2   if (udp.srcPort == 1234 &&
3     ethernet.etherType == ETHERTYPE_IPV4
4     && ipv4.protocol == IP_PROTOCOLS_UDP){
5     .
6     .
7     .
8     *** SNIPPET P4 CODE ***
9     .
10    .
11    .
12   }
13   if (valid(ipv4) and ipv4.ttl > 0) {
14     apply(ipv4_lpm);
15   }
```

P4 Parser

```
parser start {
  return parse_ethernet;
}
parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
    ETHERTYPE_IPV4 : parse_ipv4;
    default: ingress;
  }
}
parser parse_ipv4 {
  extract(ipv4);
  return select(latest.protocol) {
    IP_PROTOCOLS_TCP : parse_tcp;
    IP_PROTOCOLS_UDP : parse_udp;
    default: ingress;
  }
}
parser parse_tcp {
  extract(tcp);
  return ingress;
}
parser parse_udp {
  extract(udp);
  return select(latest.srcPort) {
    1234 : parse_p;
    default: ingress;
  }
}
parser parse_p {
  extract(p);
  return ingress;
}
```

3.3 Snippets

```
snippets:
    snippets snippet
    | snippet
    ;
snippet:
    device_annotation "snippet" SNIPPET_ID variable_decls ifblocks
    ;
```

Snippets define the logic in a Sluice program. They are blocks of code in which an operator may read or write router state, create and use local variables, and read or write packet headers. They are composed of a device annotation, variable declarations, and if-else control blocks.

3.3.1 Device Annotations

```
device_annotation:
    "@" DEVICE_TYPE device_vector
    ;
device_vector:
    device_vector "," IDENTIFIER
    | IDENTIFIER
    ;
```

Device annotations are indicated by an '@' symbol and are used to control the placement of snippets on the various devices of a topology. For instance, the annotation below indicates that the snippet *test* is to be translated into P4 for the *bmv2* device_type and that it should only run on switches *s1* and *s2*. After program compilation, this annotation is used by the control plane to automatically launch snippet P4 code on the annotated devices. By convention, if there is only a single snippet, Sluice installs that single snippet on all network devices.

```
@ bmv2 : s1, s2 ;
snippet test() {}
```

3.3.2 Variable Declarations

```
variable_decls:
    variable_decls variable_decl
    | variable_decl
```

```

;
variable_decl:
    IDENTIFIER initial_values var_type
;
initial_values:
    initial_values "," initial_value
    | initial_value
;
initial_value:
    value
;
var_type:
    var_info type_qualifier
;
var_info:
    bit_array | packet
;
bit_array:
    bit_width var_size
;
bit_width:
    digits
;
var_size:
    digits
;
type_qualifier:
    "Persistent" | "Transient" | "Field"
;
value:
    digits
;
digits:
    digit+
;
digit:
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

The first component of a snippet is the variable declaration section. Sluice supports two types of variables that can be declared and used within snippets: *transient* and *persistent* variables.

Transient variables are local variables initialized to zero by default but may be initialized by the operator. As packets flow through the switch, they can be updated using packet header values, other transient variables, or persistent variables. However, these variables are only valid for the current packet being handled by the switch. Once this packet exits and another packet enters the switch, all transient variables are reset to zero or their initial value. Transient variables are translated into P4 metadata by the Sluice compiler. It is important

to note that transient variables cannot be arrays like persistent variables as P4 itself does not support arrays in metadata.

Persistent variables, on the other hand, represent router state and translate into register arrays in P4. They are initialized to zero by default but can also be initialized by the operator. However, a write or update to a persistent variable is permanent i.e. the value stored in a persistent variable "persists" as new packets flow through the switch. Persistent variables can be updated using packet header values, transient variables, or other persistent variables, allowing them to keep track of various metrics in a network. Using our snippet *test* from earlier, the code below demonstrates the syntax for declaring transient and persistent variables.

```
packet p: udp(srcPort:1234) {
    nhops : bit<32>;
}

@ bmv2 : s1, s2 ;
snippet test() {
    transient a : bit<32> = 10;
    persistent reg1 : bit<32>[3] = {4, 1, 8};
    persistent reg2 : bit<32> = 0;
}
```

Persistent variable initial values are set by the control plane. Once the compiler translates the Sluice program into P4, it scans the program for initial values and generates commands that the control plane will execute to populate register arrays with these values, before any packets start flowing through the network. For example, *reg1* has an initial value of 8 at index 2 so the corresponding command would be `register_write reg1 2 8`. Transient variables are initialized via the `set_metadata` function in the P4 header parser. The code below shows the P4 representation for transient and persistent variables in the *test* snippet, initial value control plane table entries for persistent vars `reg1` and `reg2`, and the header parser where the transient var *a* is set to 10. If you recall from the discussion on packet parser conditions, the header parser `parse_p` is only applied to packets *p* in the flow `udp.srcPort = 1234`. This restricts transient var initialization to packets on this flow alone, preventing manipulation of router state (metadata) by other packets flowing through the network.

P4 Variable Declarations

```
header_type metadata_t {
    fields {
        a : 32;
        reg1 : 32;
        reg2 : 32;
    }
}
metadata metadata_t mdata;
register reg1 {
    width : 32;
    instance_count : 3;
}
register reg2 {
    width : 32;
    instance_count : 1;
}
```

Control Plane Initial Value Entries

```
register.write reg1 0 4
register.write reg1 1 1
register.write reg1 2 8
register.write reg2 0 0
```

Transient Var Metadata Initialization

```
parser parse_p {
    extract(p);
    set.metadata(mdata.a,10);
    return ingress;
}
```

Packet headers can be accessed, but not declared, in P4 snippets. The operator may read-write to any packet header field supported by default (eth/ip/tcp/udp) as well as any headers defined in packet declarations by the user. Header fields are accessed via dot notation separating the packet type and header field name, for example, `ipv4.srcAddr`. They will be used in several Sluice examples that follow.

3.3.3 Control Blocks

```
ifblocks:
    ifblocks ifblock
    | ifblock
    ;
ifblock:
    id condtype condition statements
    ;
id:
    digits
    ;
condtype:
    1 | 2 | 3
    ;
condition:
    expr
    ;
statements:
    statements statement
    | statement
    ;
statement:
    lvalue "=" expr
```

```

;
expr:
    operand expr_right
;
operand:
    lvalue | value
;
lvalue:
    scalar | array | field
;
scalar:
    IDENTIFIER
;
array:
    IDENTIFIER "[" operand "]"
;
field:
    IDENTIFIER "." IDENTIFIER
;
expr_right:
    BinOp | Cond | Empty
;
BinOp:
    "and" | "or" | "+" | "-" | ">>" | "<<" | "==" | "!=" | "<" | ">" | ">=" | "<="
;
Cond:
    operand "?" operand
;

```

There are 3 types of control blocks supported by Sluice: if blocks (condtype = 1), else blocks (condtype = 2), and unconditional blocks (condtype = 3). Within each block are assignment statements composed of an lvalue and a right hand side (RHS) expression. The lvalue may be a packet header, a transient variable, or a particular index of a persistent array. An array index may be a constant number, a packet header, or a transient variable. However, an array index cannot be an array lvalue i.e. a statement like `reg1[reg2[3]]` is illegal.

Ternary conditional assignment statements are also supported in Sluice. These are composed of an lvalue and a RHS *cond* expression. The RHS expression is further composed of a single bit condition variable and two operands, one of which will be assigned to the lvalue based on the value of the condition variable. Nested if-else syntax is not supported but equivalent semantics can be generated using sequences of ternary conditional statements.

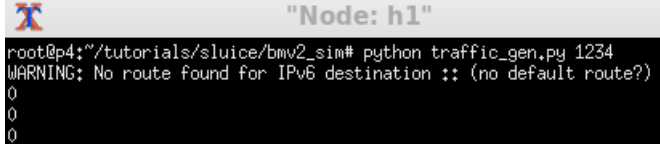
A variety of arithmetic, bitwise, and comparison operations are supported for compu-

Figure x : Simple Arithmetic Example

```

1 import device psa;
2
3 packet p: udp(srcPort:1234) {
4     result : bit<32>;
5 }
6
7 @ bmv2 : s1 ;
8 snippet simple_arithmetic() {
9     transient a : bit<32> = 2;
10    transient b : bit<32> = 6;
11    transient c : bit<1>;
12    persistent reg1 : bit<32>[3] = {4, 1, 8};
13    persistent reg2 : bit<32> = 1;
14    if(a > reg2) {
15        reg1[a] = reg2 + b;
16        reg2 = 7;
17    }
18    c = a > reg2;
19    b = reg1[a] + reg2;
20    p.result = c ? reg2 : b;
21 }

```

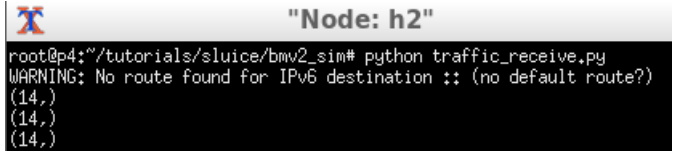


```

"Node: h1"
root@p4:~/tutorials/sluite/bmv2_sim# python traffic_gen.py 1234
WARNING: No route found for IPv6 destination :: (no default route?)
0
0
0

```

H1 sends packet p with result initialized to 0



```

"Node: h2"
root@p4:~/tutorials/sluite/bmv2_sim# python traffic_receive.py
WARNING: No route found for IPv6 destination :: (no default route?)
(14,)
(14,)
(14,)

```

H2 Receives p with a result header of 14

tation in statements including bitwise AND, bitwise OR, left shift, right shift, addition, subtraction, equal, not-equal, greater-than, less-than, LEQ, and GEQ. Although multiplication and division are not currently supported, strategies for dealing with them are discussed in future sections.

Through these features, statements allow the operator to perform computation via read-write to packet headers, transient variables, and persistent variables. The example below demonstrates a Sluite program where several transient and persistent variables are declared, some computation is performed, and either variable **b** or **reg2** is written to the packet header **result** depending on the value of **c**. A simple linear topology **h1-s1-h2** is used where the snippet **simple_arithmetic** is run on **s1**. The top terminal screen shows the packet **p** being sent from host h1 with header **result** initially set to 0. The bottom screen shows **p** being received at host h2 with the header result updated with the expected value of 14.

All statements in Sluite map to P4 tables and actions. For instance, for the unconditional statement in line 19 of the simple arithmetic example, several P4 actions are generated: one to read **reg2** into metadata, one to read **reg1** at index **a** into metadata, and one to add **reg1**

and **reg2** metadata and store the result in the transient var **b**. These actions are shown below.

Line 19 Actions

```

action action13 () {
    register_read(mdata.reg1, reg1, mdata.a);
}
action action14 () {
    register_read(mdata.reg2, reg2, 0);
}
action action15 () {
    add(mdata.b, mdata.reg1, mdata.reg2);
}

```

Conditional statement handling is slightly more complicated. Sluice programs containing if-else blocks undergo a **branch removal** step akin to a similar transformation used in the Domino compiler [cite]. Here, any statements inside an if-else block are converted into ternary conditionals.

```

1 import device psa;
2
3 packet p: udp(srcPort:1234) {
4     result : bit<32>;
5 }
6
7 @ bmv2 : s1, s2, ;
8 snippet simple_arithmetic() {
9     transient a : bit<32> = 2;
10    transient b : bit<32> = 6;
11    transient c : bit<1>;
12    persistent reg1 : bit<32>[3] = {4, 1, 8, };
13    persistent reg2 : bit<32> = 1;
14    transient if_block_tmp_1 : bit<1>;
15    transient tmp_0_if_1 : bit<32>;
16    if_block_tmp_1 = a > reg2;
17    tmp_0_if_1 = reg2 + b;
18    reg1[a] = if_block_tmp_1 ? tmp_0_if_1 : reg1[a];
19    reg2 = if_block_tmp_1 ? 7 : reg2;
20    c = a > reg2;
21    b = reg1[a] + reg2;
22    p.result = c ? reg2 : b;
23 }

```

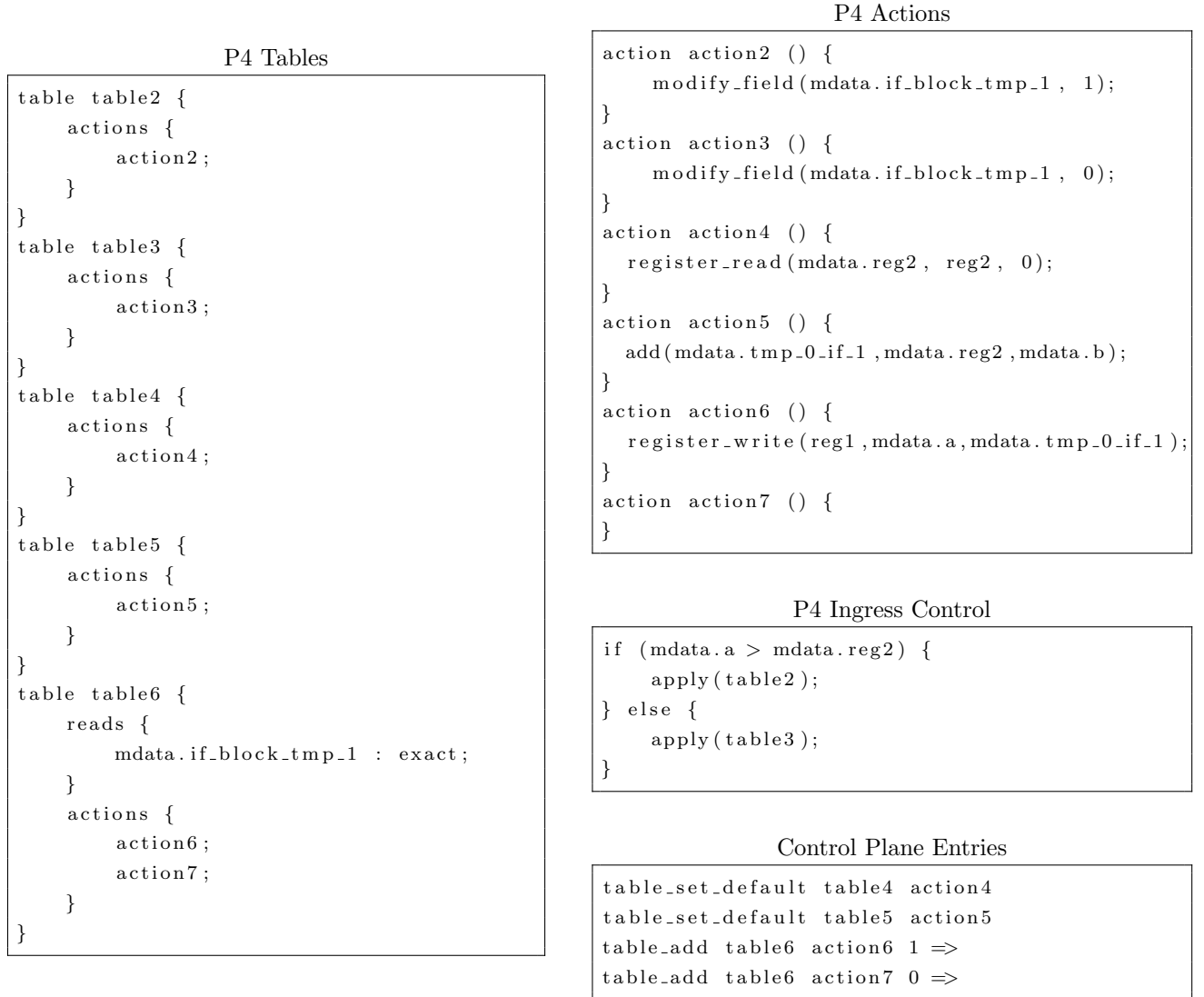
In Figure [cite] line 15 of the **simple_arithmetic** snippet, the assignment of **reg1[a]** to RHS **reg2 + b** is conditional on **a** being greater than **reg2**. Listing [cite] shows this same snippet with branches removed. To remove branches this, a new transient condition variable **if_block_tmp_1** is generated. It is then set to the single bit result of the original **if** statement

comparison `a > reg2`. Another transient variable `tmp_0_if_1` is used to store the value of the RHS, which is used only when the `if` condition is satisfied i.e. when `if_block_tmp_1 = 1`. The final conditional assignment to `reg1[a]` is expressed as a ternary conditional on line 18. If `if_block_tmp_1 = 1`, `reg1[a]` is assigned to the RHS, otherwise it is assigned to itself i.e. left alone. A similar ternary assignment based on the condition variable `if_block_tmp_1` is also created for `reg2` on line 19.

Branch removal leads to a sequence of ternary assignments which are in turn converted into P4 match-action tables by the Sluice compiler. Regular assignment statements are converted into a table with a single default action corresponding to the assignment. However, ternary conditional assignments are converted into a table with a match on the binary condition bit and the two corresponding actions. The first action represents assignment to the first ternary operand (in case the condition bit is true or 1,) while the second represents assignment to the second ternary operand (in case the condition bit is false or 0). Sluice populates the control plane with these table entries before packets start flowing through the network. The value of the condition bit is set in the P4 ingress control block Figure [cite] displays the compiler-generated P4 tables, actions, ingress control, and control plane entries corresponding to the Sluice code in lines 16-18 of listing [cite]. Note that the condition bit is only set to 1 (via `table2`, `action2`) if `mdata.a > mdata.reg2` in the ingress control block.

While branch removal does not improve nor significantly degrade performance, it transforms a snippet into straight-line code, greatly simplifying dependency analysis. This analysis is detailed more in next section.

Figure x : Conditional Statement P4 Translation

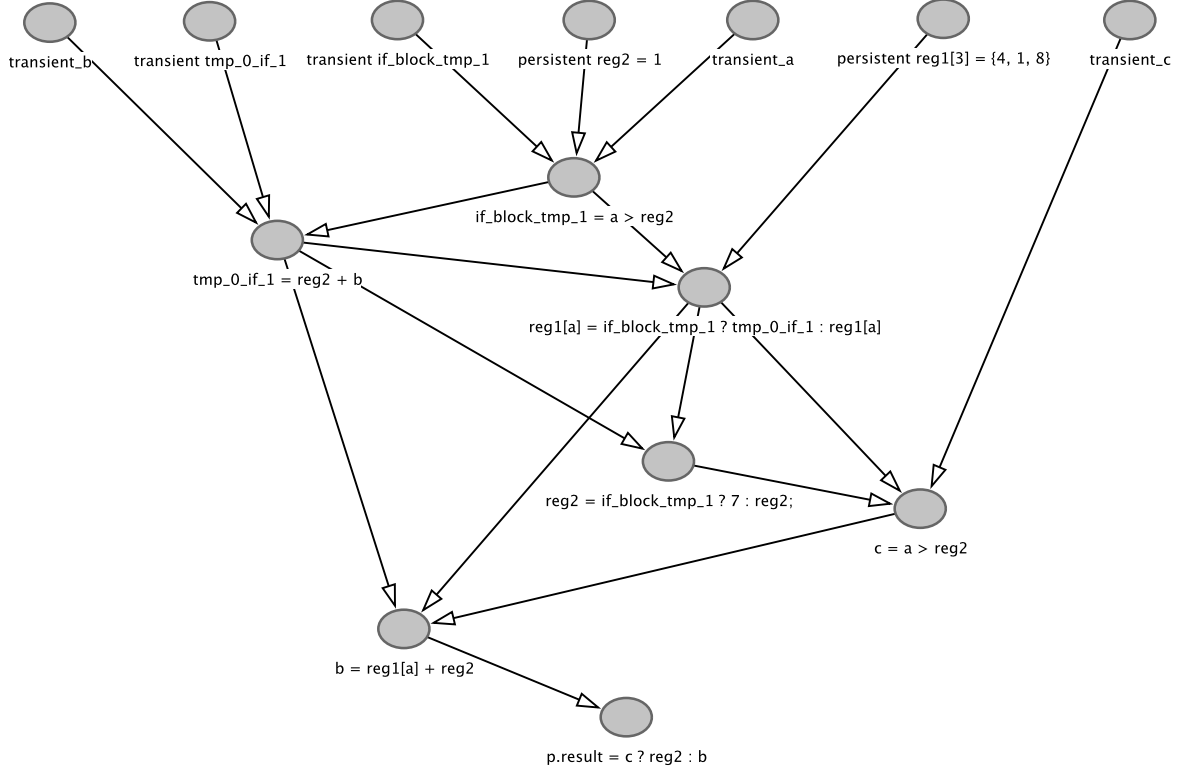


3.4 Dependency DAG

The Sluice model allows the possibility for statements to be moved between snippets i.e. between devices. However code movement is inherently dangerous as the semantics of the new program must be the same as those intended by the operator in the original snippet. Consider a simple unconditional assignment statement in snippet A that is moved to snippet B. If the lvalue variable is read again in snippet A (read after write RAW), movement of the statement onto snippet B may change future values of certain variables in snippet A. To keep track of these variable dependencies within a snippet, a snippet is decomposed into **atoms** and a directed acyclic graph (DAG) is constructed from these atoms (nodes in the DAG). An atom is a line or the smallest unit of a code in Sluice that needs to be translated to P4. An atom could be a variable declaration, conditional statement, or unconditional statement. A dependency between individual atoms could be RAW, write after write (WAW), or write after read (WAR). This DAG construction is necessary to maintain consistency when performing translation/movement, and to map the stages in switch-pipeline accordingly ². After the branch removal step, Sluice builds the DAG by maintaining the next and previous atoms for each atom in the snippet.

The dependency graph for snippet `simple_arithmetic` in listing [cite] is shown in Figure [cite]. The top layer of nodes represents the snippet variable declarations. Arrows represent dependencies between variables in the parent and child nodes. For example, we can trace the use of transient variable `a` in statements of the snippet. The dependency path for `a` touches the following nodes in order: `transient a` \rightarrow `if_block_tmp_1 = a > reg2` \rightarrow `reg1[a] = if_block_tmp_1 ? tmp_0_if_1 : reg1[a]` \rightarrow `c = a > reg2` \rightarrow `b = reg1[a] + reg2`.

²Pipeline stage mapping is relevant for the Tofino architecture, which is not currently supported.



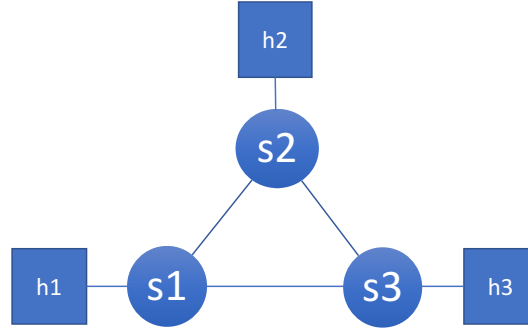
Dependency DAG for Listing [cite]

4 Demonstrations

In this section, we demonstrate Sluice’s functionality and ease of use via two examples: traffic matrix generation for network analysis and a streaming join-filter operation. Videos for these two demos can be found at [cite] and [cite] respectively. Before we describe them further, we provide a description of the Sluice simulator.

4.1 Sluice Simulator

The Sluice simulator is written in python and automates much of the aspects of network-wide programming. The operator need only provide a Sluice program and a topology json file outlining the layout of switches and hosts in the network topology. Given the topology file, the simulator generates arp requests for link layer addressing between hosts, determines the shortest routing paths between hosts, and installs forwarding rules on each switch. It uses



Topology For Traffic Matrix Demo

annotations in the Sluice program to create independent P4 programs for each snippet and run them on their respective devices. In the case that the operator intends to run multiple snippets on a single switch, these snippets will have to be merged to generate a single P4 program. While snippet merging is not currently supported, this is a desired feature and is discussed in the future work section.

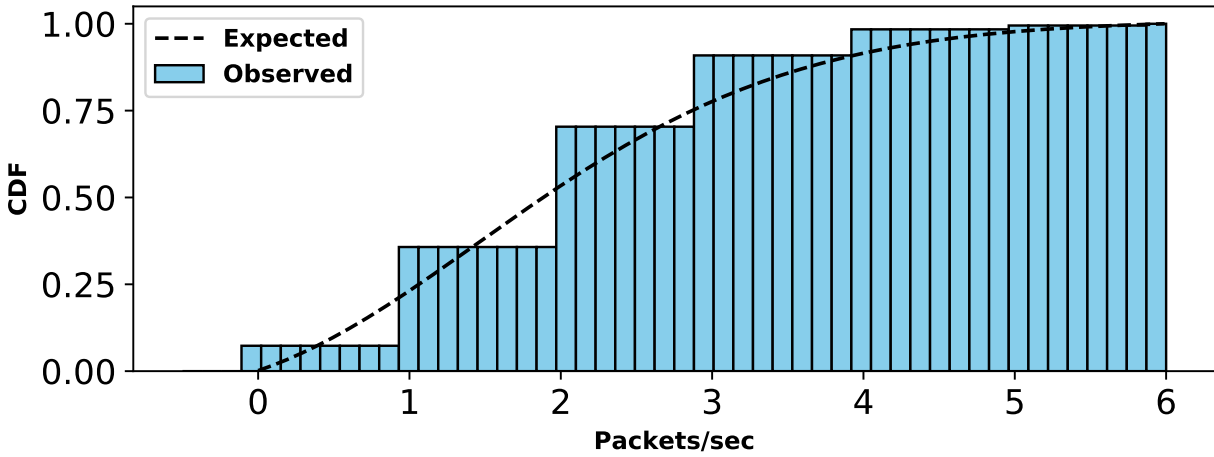
Persistent switch state in registers can be read/written to using the simulator. We provide an API that accesses registers from the control plane using an RPC call over the Apache thrift port corresponding to the switch. For instance, the API function call `register_read reg1` returns a list of all values in the array `reg1`. The simulator also automates the installing of control plane table entries for register initial values and statements in snippets.

4.2 Traffic Matrix

Figure 2 displays the Mininet network topology 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 codelet below is our 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.

Sluice Traffic.Example

```
import device psa;
```



CDF of Packet Rate on link s1-s3

```

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;

```

The codelet below shows the corresponding P4 actions generated by the Sluice compiler to execute the statements in snippet `traffic_example`.

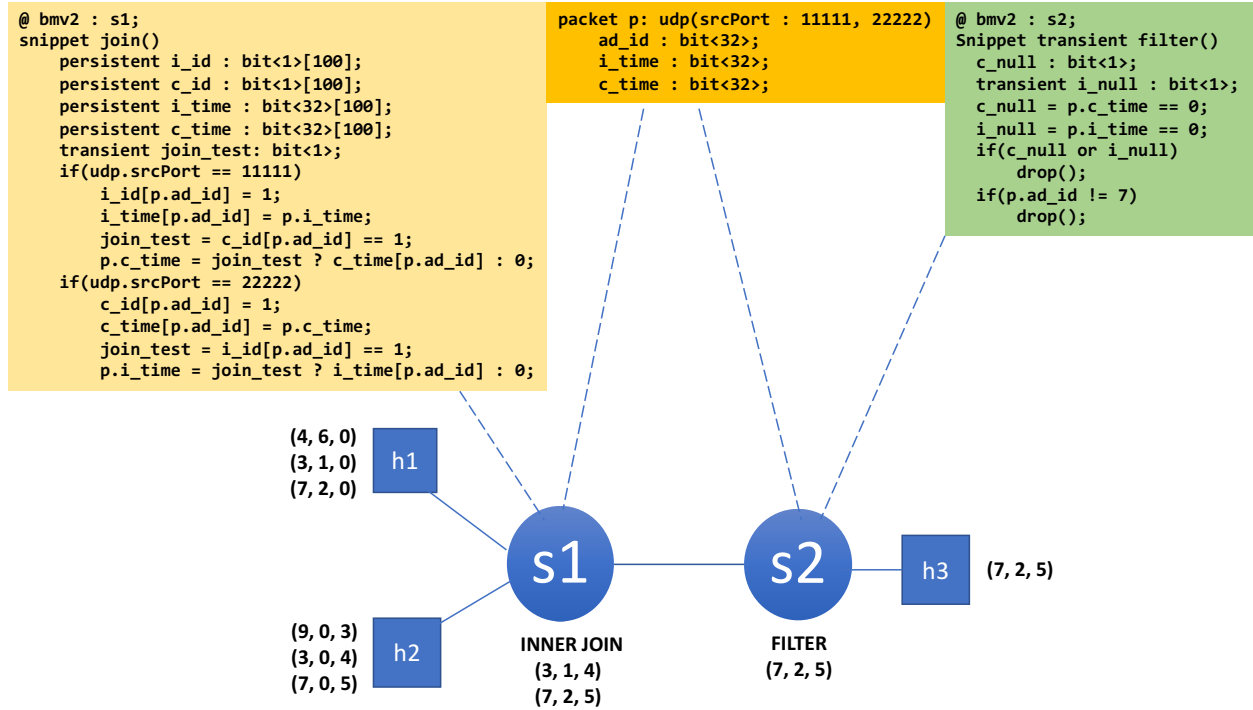
P4 Actions For Traffic_Example

```

action action1 () {
  register_read(mdata.cnt, cnt, standard_metadata.ingress_port);
}
action action2 () {
  add(mdata.cnt, mdata.cnt, 1);
  register_write(cnt, standard_metadata.ingress_port, mdata.cnt);
}
action action3 () {
  add(p.nhops, p.nhops, 1);
}

```

This demo shows how a simple Sluice program can be used to enable each switch to measure link usage for a specific flow of user-defined packets on UDP srcPort 1234. 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 switch ingress port, that tracks how many packets have entered through that ingress port. Aggregated over all switches, these



Streaming example topology, data flow, and code placement on switches

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 link-utilization 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 $\text{Poisson}(\mu = 2 \text{ packets/sec})$ is also plotted to confirm the accuracy of the Mininet emulation.

4.3 Stream processing

This example demonstrates a simple join-filter operation between two streams of tuples. Expressing join in Sluice requires only 31 lines whereas the equivalent P4 code is over 400 lines.

A stream is an unbounded table where a packet p represents a tuple of data (ad_id , impression_time , click_time) enclosed in a custom header. The operator wishes to correlate the two streams to determine the time lag between a user seeing the ad with $\text{ad_id} = 7$ and clicking that same ad. The topology in Figure [cite fig num] 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 with `udp.srcPort = 11111` while Host 2 sends a stream of ad clicks with `udp.srcPort = 22222`. The two streams are joined on the `ad_id` field at `s1` (which runs `snippet join`) and filtered at `s2` (which runs `snippet filter`) when `ad_id = 7`. The final result is sent to `h3`. The MySQL query corresponding to this example is shown below.

```
SELECT adId, impression_time, click_time
FROM impressions i
INNER JOIN clicks c ON i.adId = c.adId
WHERE i.adId = 7
```

Traditional join algorithms such as nested loop join, sort-merge join and hash join all require loops. On the other hand, Sluice is able to accomplish joins using loop-free semantics under several assumptions. Let us assume that the impression packet (3, 1, 0) arrives first at `s1`. The `ad_id` field with value 3 is used as an index into the `i_id` register and this register is set to 1 at the index 3. The `ad_id` field is also used as an index into the `i_time` register, which is updated at index 3 with the impression time of 1. In this way, the `i_id` register serves as a bit vector indicating whether or not the switch has seen particular ad impressions whereas the `i_time` register stores the impression times for the ad impression tuples the switch has seen so far.

Let us now assume that the click packet (3, 0, 4) arrives at `s1`. As in the case of impression packets, a pair of registers, `c_id` and `c_time`, are maintained to record arriving ad click packets. At this point, we check whether the switch has already seen an impression with an `ad_id` equal to that of the current click packet i.e. 3. If the value of bit `i_id[p.ad_id]` is 1, the impression has been seen and the corresponding impression time `i_time[p.ad_id]` is written to the packet header `p.i_time`. Since the impression packet (3, 1, 0) was already seen, the value of `i_id[3]` is indeed 1 and the impression time `i_time[3] = 1` is written to the click packet to yield (3, 1, 4).

The result of `s1`, (3, 1, 4), now flows to `s2` where it is filtered on `ad_id = 7`. Since the `ad_id` 3 is not equal to 7, the packet is simply dropped. Although not shown here, a similar analysis can be done when a click arrives before a matching impression.

It is important to note several caveats to this algorithm. For instance, if there are multiple

impressions for the same ad, only a single tuple will be stored on the switch at any time since the primary key `ad_id` is duplicated and the corresponding index gets overwritten. This is equivalent to an unresolved hash collision. In this case, it is simple to resolve by tracking impressions with an extra key, for example, an `impression_id` that is unique among impressions of a particular ad.

Another caveat is that the maximum number of unique tuples that can be stored for joining is limited by the total TCAM and SRAM memory of the target hardware. While a capacity of 100 unique `ad_ids` is suitable for this demo, whether it is suitable for a different application depends on how fast the streams are being sent. For example, suppose the switch only supports 30Mb of total memory. If impressions arrive at a rate of 30Mb/s and clicks arrive at a slower rate of 10Mb/s, then switch memory will be exhausted in 1 second with a majority of register entries storing tuples for impressions as opposed to clicks. Due to varying send rates, a large time lag ensues between possible impression/click join matches. In addition, since switch memory is quickly exhausted, many tuples are overwritten causing possible matches to not be recognized. Thus the operator may have to fine-tune sending rates to ensure that the probability of failed matches is minimized.

5 Future Work

5.1 An optimizing Sluice compiler.

We envision using the dependency DAG to provide several automatic optimizations or 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 data plane 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. Consider a case where a switch needs to carry out some computations based on packet headers alone without reading or writing switch-resident state. One example is taking the product of two headers and writing it into a third header without involving any switch state in the process. This pattern occurs in the congestion-control protocol XCP [cite 38], which needs to compute the product of the current RTT and the packet size and divide it by the current congestion window. All three of these variables are present in the packet and need not be computed on the switch. This makes it simpler to offload the computation to an end host where precise floating point division and multiplication are much easier to carry out [cite 36]

In certain situations, code movement cannot be performed even when floating point accuracy is desired. Consider the snippet below which computes an exponentially weighted moving average (EWMA with $\alpha = 0.2$) of queuing latencies and writes the EWMA to a packet header `result`.

```
import device psa;

packet p: udp(srcPort:1234)
  result : bit<32>;

@ bmv2 : ;
snippet ewma() {
  persistent avg : bit<32>;
  transient latency : bit<32>;
  avg = 8 * avg;
  avg = avg / 10;
```

```

latency = psa.timestamp_egress - psa.timestamp_ingress;
latency = 2 * latency;
latency = latency / 10;
avg = avg + latency;
p.result = avg;

```

The statements computing EWMA include division and multiplication and must accurate to several decimal places. The EWMA itself exists as a persistent variable `avg` on the switch which is updated with every packet `p` on the flow `udp.srcPort=1234`. In addition, it is written to the `result` header of each packet `p`. To match these stateful semantics, all statements must be run on the switch and cannot be moved onto the control plane or an end-host. The result must also be approximated with fixed point arithmetic since floating point is not supported in switch hardware.

Fortunately, it is straightforward to represent a multiplication using bit shifts and adds by writing out both numbers in binary. For instance, say we were multiplying 27 by 5. As shown below, this can be implemented by first shifting 27 left by 2 bits and then adding 27.

$$\begin{aligned}
& 27 * 5(1) \\
&= (11011) * (101) \\
&= (11011) * (1 * 2^2 + 0 * 2^1 + 1 * 2^0) \\
&= (11011) << 2 + (11011) << 0 \\
&= (11011) << 2 + (11011)
\end{aligned}$$

Similarly, we can approximate division by treating it as multiplication by the inverse and using right shifts instead of left shifts. Assuming a budget on N number of shifts (and hence $N - 1$ adds), we can use a lookup table to compute the shift coefficients for each value of the smaller of the two operands. For instance, in the example above where we multiply 27 by 5, this lookup table will tell us that to multiply by 5, we need to use the shift values 2 and 0 if we have a budget of two shifts and the shift value 2 alone if we have a budget of one shift. The more the budget on the number of shifts, the closer we can approximate the original multiplication or division. However, the smaller the budget on the number of shifts, the more we can compress the lookup table. For instance, if we are permitted only one shift, then the lookup table returns the nearest exponent of 2 for each value of the

smaller operand, allowing us to significantly compress the lookup table. As one example of compression, assuming 8-bit operands, the shift value for all operands from 17 to 31 will be 4 and can be represented by a single TCAM entry mapping the bit pattern 0001**** to the shift value 4.

5.2 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 [cite] provided for the control plane.

6 Appendix

6.1 Sluice grammar

```
program:
    imports packets snippets
    | packets snippets
    ;
imports:
    imports import
    | import
    ;
import:
    "import device" IMPORT_ID
    ;
packets:
    packets packet
    | packet
    ;
packet:
    "packet" PACKET_ID packet_base packet_parser_condition packet_fields
    ;
packet_base:
    "tcp" | "udp" | "ipv4" | "ethernet"
    ;
packet_fields:
    packet_fields packet_field
    | packet_field
    ;
packet_field:
    IDENTIFIER var_type type_qualifier
    ;
packet_parser_condition:
    IDENTIFIER value
    | Empty
    ;
snippets:
    snippets snippet
    | snippet
    ;
snippet:
    device_annotation "snippet" SNIPPET_ID variable_decls ifblocks
    ;
device_annotation:
    "@" DEVICE_TYPE device_vector
    ;
device_vector:
    device_vector "," IDENTIFIER
    | IDENTIFIER
    ;
variable_decls:
    variable_decls variable_decl
```

```

        | variable_decl
        ;
variable_decl:
    IDENTIFIER initial_values var_type
    ;
initial_values:
    initial_values "," initial_value
    | initial_value
    ;
initial_value:
    value
    ;
var_type:
    var_info type_qualifier
    ;
var_info:
    bit_array | packet
    ;
bit_array:
    bit_width var_size
    ;
bit_width:
    digits
    ;
var_size:
    digits
    ;
type_qualifier:
    "Persistent" | "Transient" | "Field"
    ;
value:
    digits
    ;
digits:
    digit+
    ;
digit:
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    ;
ifblocks:
    ifblocks ifblock
    | ifblock
    ;
ifblock:
    id condtype condition statements
    ;
id:
    digits
    ;
condtype:
    1 | 2 | 3
    ;
condition:
    expr
    ;

```



```

statements:
    statements statement
    | statement
    ;
statement:
    lvalue "=" expr
    ;
expr:
    operand expr_right
    ;
operand:
    lvalue | value
    ;
lvalue:
    scalar | array | field
    ;
scalar:
    IDENTIFIER
    ;
array:
    IDENTIFIER "[" operand "]"
    ;
field:
    IDENTIFIER "." IDENTIFIER
    ;
expr_right:
    BinOp | Cond | Empty
    ;
BinOp:
    "and" | "or" | "+" | "-" | ">>" | "<<" | "==" | "!=" | "<" | ">" | ">=" | "<="
    ;
Cond:
    operand "?" operand
    ;

```

6.2 Sluice Compiler Source Code

The Sluice compiler is written in rust while its simulator is written in python and bash. Source code can be found at [cite github]. The lines-of-source breakdown is shown below.

Language	files	blank	comment	code
Rust	15	689	419	4836
P4	58	110	0	603
Python	6	153	176	528
JSON	62	0	0	62
Bourne Shell	5	29	15	55
make	1	15	2	31
Markdown	3	5	0	18
TOML	1	1	0	9
SUM	151	1002	612	6142