SDN Programming Capacity Theorem on Realizing High-Level Programs on Low-Level Datapaths

Xin Wang[†‡], Christopher Leet[*], Y. Richard Yang[*†], James Aspnes[*], Changjun Jiang[†‡§]

[†] Department of Computer Science, Tongji University [*] Department of Computer Science, Yale University [‡] Key Laboratory of Embedded System and Service Computing, Ministry of Education, China [§] Department of Computer Science, Donghua University

Abstract

High-level programming and programmable data paths are two key capabilities of software-defined networking (SDN). A fundamental problem linking these two capabilities is whether a given high-level SDN program can be realized onto a given low-level SDN datapath structure. Considering all high-level programs that can be realized onto a given datapath as the programming capacity of the datapath, we refer to this problem as the *SDN datapath programming capacity problem*. In this paper, we conduct the first study on the SDN datapath programming capacity problem, in the general setting of *high-level, datapath oblivious, algorithmic SDN programs* and state-of-art *multi-table SDN datapath pipelines*. In particular, considering datapath-oblivious SDN programs as computations and datapath pipelines as computation capabilities, we introduce a novel framework called *SDN characterization functions*, to map both SDN programs and datapaths into a unifying space, deriving the first rigorous result on SDN datapath programming capacity. We not only prove our results but also conduct realistic evaluations to demonstrate the tightness of our analysis.

A preliminary version of this work appeared in [1].

I. Introduction

A major research direction of SDN is programmable, efficient datapaths (*e.g.*, OF1.3 [2], OF-DPA [3], P4 [4]). Only by being programmable can a given SDN datapath support diverse, ever evolving application scenarios. At the same time, it is crucial that datapaths be efficient, to be able to satisfy demanding requirements such as achieving high throughput and being cost effective. In the last few years, multi-table pipelines have emerged as a key structure of SDN datapaths (*e.g.*, Domino [5], Forwarding Metamorphosis [6]).

One problem of efficient datapaths, however, is that they must often be programmed at an inefficiently low level. For example, TCAM, which is essential to achieve high-throughput, does not support logical negation. Hence, a second major research direction of SDN is high-level, datapath path-oblivious programming, to provide abstractions to hide low-level datapath programming. To this end, in the last few years multiple high-level SDN programming models have emerged (*e.g.*, Frenetic [7], Maple [8]).

As both directions progress, a basic problem emerges: whether a given high-level program can be realized on a given low-level datapath. A good understanding of this problem can benefit both the design of high-level SDN programming and the design of datapaths. Given a fixed datapath (*e.g.*, a fixed pipeline architecture such as OF-DPA), the vendor of the datapath can provide guidelines on the high-level programs that can be realized. Given a set of high-level programs to be supported, one could use this understanding to design the most compact datapath supporting these programs. Even for reconfigurable datapaths (*e.g.*, P4), as reconfiguration can be expensive and time consuming, one can use this understanding to guide the design of a more robust datapath. Considering all high-level programs that can be realized onto a given programmable datapath as the capacity of the datapath, we define the basic problem as the *SDN datapath programming capacity problem*.

Solving the datapath capacity problem, however, is not trivial. Consider a simple datapath, named Simple-DP, shown in Fig. 1. It is among the simplest datapaths, consisting of three tables forming a pipeline, where the first table (`t1`) matches on source IP and may jump to one of the two following tables, which both match on destination IP.

Consider two simple high-level SDN programs below, both specified in the algorithmic, event-driven programming style to handle packet misses; see Sec. II for more details on the programming model. An interested reader can try to verify that the first program can be realized by Simple-DP, but the second cannot.

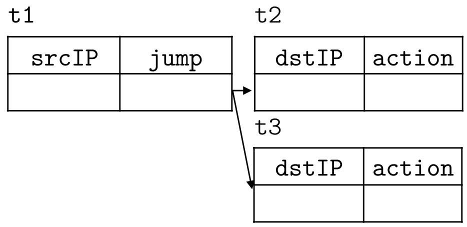


Fig. 1. A simple example datapath: Simple-DP.

```
//  Routing Function: secureL3Route
L0: secureL3Route(Addr srcIP, Addr dstIP):
L1:   if srcIP == 10.0.0.1:
L2:     return Forward(port=shortestPath(dstIP))
L3:   else:
L4:     return Drop();
```

```
//  Program: twoHostL3Route
L0: def twoHostL3Route(Addr srcIP, Addr dstIP):
L1:   if srcIP == 10.0.0.1:
L2:     return Forward(port=shortestPath(dstIP))
L3:   elif srcIP == 10.0.0.2:
L4:     return Forward(port=securePath(dstIP))
L5:   else:
L6:     return Drop();
```

Although the preceding datapath and high-level programs are among the simplest, they may already appear to be non-trivial for a reader to analyze. General datapath and high-level programs can be much more complex as multiple services need to be implemented and hence they can pose severe challenges in analysis. The goal of this paper is to develop the first systematic methodology to solve the SDN datapath programming capacity problem.

The contributions of this paper can be summarized as follows. First, we propose a unifying characteristic functional space to unify and extract the essence of programs and pipelines, removing complexities such as program structures and pipeline layouts. Second, we define a comparator in this functional space, which can be used to check whether a high-level program can be realized on a given pipeline.

The rest of the paper is organized as follows. We define our model precisely in Sec. II. The main results are given in Sec. III and the proofs are shown in Sec. IV. Sec. V shows our evaluation results. Finally, related work is provided in Sec. VI.

II. MODELS

We start by specifying the high-level SDN programs and low-level datapath models. Since the main focus of SDN is routing, we refer to a high-level SDN program as a routing function. Since multi-table pipelines are the state-of-art for SDN datapaths, we focus on pipelines as datapaths.

A. Routing Function Model

Routing function: We denote a routing function as f, and assume that it is a logically centralized, deterministic function written in a high level language logically executed by an SDN controller on every packet [8] entering that controller's network to determine network-wide routing for that packet.

Each execution of f on a packet reads a set of the packet's attributes (called match fields) $\mathcal{M} = \langle m_1, ..., m_n \rangle$ (*e.g.*, `<srcIP, dstIP, ...>`). We use M to denote a subset of packet match fields included in $\mathcal{M}$. Moreover, we denote $dom(M)$ as the domain of a set of match fields M. The execution of f returns a routing action from a set of valid actions $\mathcal{R}$ (*e.g.*, `Drop`, `Forward(port=2)`):

$$f : dom(\mathcal{M}) \rightarrow \mathcal{R}.$$

The space of such functions is denoted $\mathcal{F}$.

Example: We use the routing function `onPkt` below to illustrate key features of our routing function model.

```
\\  Routing function: onPkt
    Map hostTbl[key: dstIP, value: switch]
    Map condTbl[key: (dstIP, port), value: cond]
    Map routeTbl[key: (switch, cond), value: outPort]
L0: onPkt(Type ethType, Addr srcIP, Port srcPort, \
          Addr dstIP, Port dstPort):
L1: if (ethType != IPv4):
L2:   return Drop()
L3: if (verify(srcPort, srcIP)):
L4:   dstCond = condTbl[dstIP, dstPort]
L5:   dstSw = hostTbl[dstIP]
L6:   return Forward(port = routeTbl[dstCond, dstSw])
L7: return Drop()
```

Specifically, `onPkt` reads the match fields $\mathcal{M} = \langle \texttt{ethType}, \texttt{srcIP}, \texttt{srcPort}, \texttt{dstIP}, \texttt{dstPort} \rangle$ and maps each value in the domain of $\mathcal{M}$ to a routing action in $\mathcal{R} = \{\texttt{Drop()}, \texttt{Forward(port=x)}\}$. While we write `onPkt` as an imperative function, we emphasize that our model is fully generic and does not specify a programming paradigm.

Elaborating, `onPkt`'s first three lines declare key-value tables. Specifically, `hostTable` and `condTable` associate each IP address with an attachment switch and host condition (*e.g.*, authentication status) respectively, while `routeTable` maps a switch, condition pair to its forwarding port. Moving on to `onPkt`'s body, `L1` and `L2` detect and drop non-IPv4 traffic, while `L7` drops traffic from unverified endpoints. For verified packets, `L4` to `L6` further set `dstCond` and `dstSw` variables, and then return a routing action from `routeTbl` based on the two variables.

Routing function DFG: Since a generic routing function can have arbitrary, complex control structure, we transform a routing function into a dataflow graph (DFG) to better represent its structure. We denote an f's DFG as G_f.

Specifically, to compute G_f for f, we must remove all of f control follow dependencies. These dependencies are removed by the following transformations:

- We remove assignment statement order dependencies by converting f to single static assignment form (SSA).
- We remove branches by assigning their conditionals' values to guards, and appending dependencies on these guards to all statements in their `if` and `else` blocks.
- We remove program loops by converting them to black box functions which read all variables read by the loop and write all variables written by them.

For example, our example routing function `onPkt` is transformed as follows:

```
L0: onPkt(...):
L1: g0 = (ethType != IPv4)
L2: if  g0: return Drop()
L3: g1 = verify(srcPort, srcIP)):
L4: if  g1: dstCond = condTbl[dstIP, dstPort]
L5: if  g1: dstSw = hostTbl[dstIP]
L6: if  g1: return Forward(port = routeTbl[...])
L7: if !g1: return Drop()
```

Note that `onPkt`'s `if` statement at `L1`'s has been replaced by an assignment from its conditional to the guard `g0`. This guard is appended to `L2`, which was formally in the `if` statement's `if` block.

Given this transformation, we define G_f for f:

Definition 1. *A routing function f's' dataflow graph DFG $G_f = (V_f, E_f)$ is a vertex weighted dag generated from a transformed f such that:*

- *Each vertex v_f in V_f is a variable in f.*
- *A v_f's weight is its domain size.*
- *There is a directed edge in E_f between two variables if the source variable appears in the target variable's assignment.*

As an example, we give `onPkt`'s DFG below:

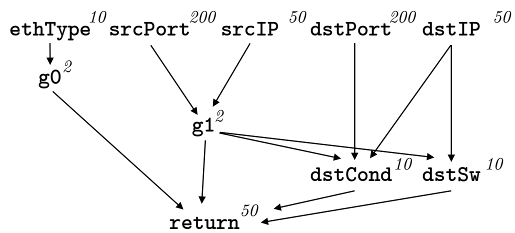


Fig. 2. The routing function `onPkt`'s DFG $G_{\texttt{onPkt}}$.

Observe that the vertex `dstSw` is descended from the two variables in its assignment, `g1` and `dstIP`. The vertex's weight, 10, indicates `dstSw`'s domain.

B. Pipeline Model

We focus on state-of-the-art datapaths: multi-table pipelines. We first model a table t in a pipeline p and then we give a clear definition for the pipeline.

Pipeline table: Each pipeline table $t \in p$ is a exact match match-action table. Each of t's actions is a routing action output, or a write to t's output register $r(t)$ followed by a hop to a subsequent table in p, or a simple jump action to a subsequent table in p. Not all t output routing actions, and we denote the t that do as an egress table.

Each t matches on a set of inputs $I(t)$ that contains packet match fields $m_i \in \mathcal{M}$ and preceding tables' output registers $r(t)$. Key limitations on a t are the maximum number of rules it can contain and $r(t)$'s bit length, which we denote $maxrules(t)$ and $bits(r(t))$ respectively.

Pipeline: A pipeline p is a singly rooted dag (directed acyclic graph) of tables $\{t_i\}$. An edge (t_i, t_j) in a p indicates that a packet arriving at t_i can jump to t_j.

Each packet passing through p starts at p's root and proceeds through p to an egress table. Therefore, each packet passing through p can map to a path in the p, along with a routing action for that packet from $\mathcal{R}$.

A packet's path through p and the action its egress table outputs are determined by the set of packet match fields $\mathcal{M}$ each $t_i \in p$ matches on. Given this, p may also be summarized as a mapping from $dom(\mathcal{M})$ to $\mathcal{R}$, which depends on p's contents.

We denote the space of all pipelines p as P.

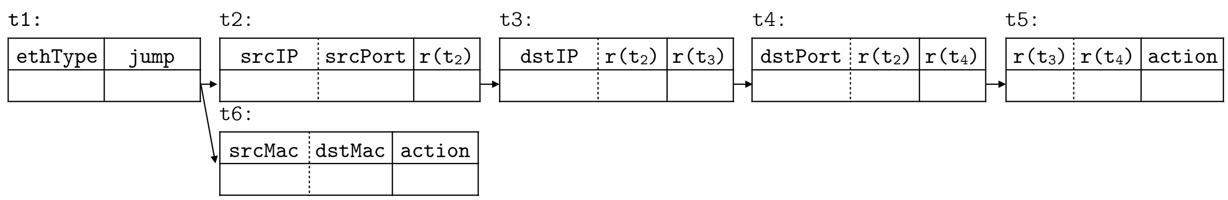


Fig. 3. Example datapath, `ExampleDP`

Example: We now give an example pipeline `ExampleDP`, shown in Fig 3, to illustrate our pipeline model. Note that in the example, a table matches on fields on its left-hand side, writes to a register on its right-hand side, and the field `output` of a table indicates the table contains output routing actions.

Narrowing our focus, consider `t2` $\in$ `ExampleDP`. `t2` is an exact match table whose inputs $I($`t2`$)$ are `srcIP` and `srcPort`, and whose output register is `r(t2)`.

Significant computation limits on `t2` are its maximum number of rules $maxrules($`t2`$))$ and the size of its output register $bits($`r(t2)`$)$.

III. Main Results

Given the function and pipeline models, we now present our main results, on whether a function f can be realized by a pipeline p.

To simplify the reading of our results, we put only the definitions and main results in the main text. The proofs of the results are in the appendix. To make it easier to follow the symbols, we collect key symbols in Table I for reference.

A. Overview

A main challenge in developing a systemic method to verify whether a routing function f can be realized by a pipeline p, which we denote as $f \Rightarrow p$, is that routing functions and pipelines

Symbol	Definition
Routing function symbols	
f	Routing function
$\mathcal{F}$	Routing function space
m_i	Packet match field
$\mathcal{M}$	Set of $\forall$ m_i
$dom(M)$	Domain of valid values of M
$\mathcal{R}$	Set of $\forall$ valid routing actions
Pipeline symbols	
p	Pipeline
$\mathcal{P}$	Pipeline function space
t_i	Pipeline table
$r(t_i)$	t_i's output register
$bits(r(t_i))$	t_i's output register bit length
$I(t_i)$	t_i's table inputs
$maxrules(t_i)$	Maximum # of rules t_i can contain

TABLE I
SYMBOL TABLE LISTING NOTATION IN OUR MAIN RESULTS.

are represented differently and both types of representations can have substantial complexities and variations. Consider each routing function f as a point in a functional space $\mathcal{F}$, and each pipeline p as a point in functional space $\mathcal{P}$.

Our main contribution is the introduction of a novel, unifying, normalization functional space $\mathcal{C}$ called the characteristic functions space. Each routing function f is mapped by the mapping τ to a characteristic function $\tau(f) \in \mathcal{C}$, characterizing the computational load of f. Each pipeline p, on the other hand, is mapped to a set $\kappa(p) \subset \mathcal{C}$ of characteristic functions, representing the set of computational capabilities of the pipeline. Fig 4 illustrates the mapping structure.

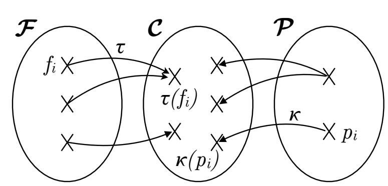


Fig. 4. The spaces $\mathcal{F}$, $\mathcal{P}$ and $\mathcal{C}$ and the mappings between them.

Since $\tau(f)$ and $\kappa(p)$ are defined in the the same space $\mathcal{C}$, as a point and as a set of points respectively, one can compare $\tau(f)$ with each element in $\kappa(p)$, to see if the load can be "covered" by a capability, resulting in our basic capacity theorem: that if $\exists\ c \in \kappa(p) \geqslant \tau(f), f \Rightarrow p$.

B. Characteristic Functions

We begin by defining a generic characteristic function c.

Definition 2. *A* characteristic function *c is a mapping from each subset M of a packet's match fields to a vector consisting of two components:*

$$c(M) \triangleq < scope(M),\ ec(M) > .$$

We refer to the two components of $c(M)$'s vector as $c(M)[scope]$ and $c(M)[ec]$ respectively. Given two characteristic functions, one can compare them.

Definition 3. *We define c_i* dominates *c_j, denoted as $c_i \geqslant c_j$ as follows:*

$$c_i \geqslant c_j \triangleq\ \forall\ n \in \{scope, ec\},$$
$$\forall\ M \in 2^{\mathcal{M}},\ c_i(M)[n] \geqslant c_j(M)[n].$$

To verify our capacity theorem, we need to compare a set of characteristic functions with a single characteristic function.

Definition 4. *A set of characteristic functions C_i* dominates *a characteristic function c_j, denoted as $C_i \trianglerighteq c_j$, if a $c_i \in C_i$ dominates c_j:*

$$C_i \trianglerighteq c_j \triangleq \exists\ c_i \in C_i : c_i \geqslant c_j.$$

C. Characterization of a Routing Function

Given the concept of characteristic functions, we now derive the characteristic function, denoted as $\tau(f)$, of a routing function f.

Definition 5. *The* scope *of the characteristic function of a routing function for a subset of packet match fields M is the size of the domain of valid values of M:*

$$\tau(f)(M)[scope] \triangleq dom(M)$$

$\tau(f)(M)[ec]$ is a property that we build from the concept of f-equivalence:

Definition 6. *We define* f-equivalence*, denoted as* $\sim_f$*, as a relationship between two values of* M*, which we write as* $v_i(M)$ *and* $v_j(M)$*, which denotes that these values cannot be distinguished by* f*:*

$$v_i(M) \sim_f v_j(M) \triangleq \forall\ v_k(\mathcal{M} - M) \in dom(\mathcal{M} - M),$$
$$f(v_i(M), v_k(\mathcal{M} - M)) = f(v_j(M), v_k(\mathcal{M} - M)).$$

Our definition of f-equivalence leads naturally to our definition of an f-equivalence class.

Definition 7. *An* f-equivalence class*, denoted as* $[v_i(M)]_f$*, is the set of all values f-equivalent to a given* M*'s value* $v_i(M)$*:*

$$[v_i(M)]_f \triangleq \{v_j(M) \in dom(M) : v_i(M) \sim_f v_j(M)\}.$$

Counting equivalence classes gives us the concept of f-equivalence class number.

Definition 8. *The* f-equivalence class number *of* M*, denoted as* $|dom(M)/ \sim_f|$*, is the cardinality of* M*'s set of f-equivalence classes.*

We now arrive at our definition of $\tau(f)(M)[ec]$.

Definition 9. *The* ec *of a routing function's characteristic function for an* M *is the cardinality of* M*'s set of f-equivalence classes:*

$$\tau(f)(M)[ec] \triangleq |dom(M)/ \sim_f|$$

Definition 10. The characteristic function $\tau(f)$ of a routing function *characterizes* f*'s computational load:*

$$\tau(f)(M) \triangleq (dom(M),\ |dom(M)/ \sim_f|).$$

While $\tau(f)$ is powerful it is impractical because f-equivalence class number is costly to directly calculate. Let us take the instruction, `dstCond = condTbl[dstIP, dstPort]`, in the `onPkt` as an example. The instruction takes two variables, `dstIP`, `dstPort`, as input, and computes `dstCond` by matching the table, `condTdl`. The DFG of the instruction which is a subgraph of `onPkt`'s DFG as shown in Fig. 5.

Using the definition of f-equivalence class, we can compute $|dom(M)/ \sim_f|$ equals 10 where $M = ($`dstIP`, `dstPort`$)$ from the DFG as shown in the Fig. 7. However, the computation of

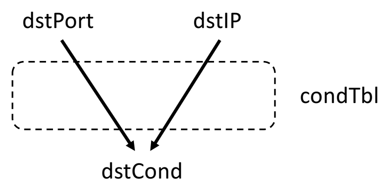


Fig. 5. A subgraph of `onPkt`'s DFG $G_{\texttt{onPkt}}$.

both $|dom(dstIP)/\sim_f|$ and $|dom(dstPort)/\sim_f|$ is hard. Consider an extreme case where the domain size of both `dstIP` and `dstPort` equals 100 and the domain size of `dstCond` equals 2 (*i.e.*, the output values of `condTbl` only include 0 and 1). Even for this simple case, we can construct the content of *condTbl* as shown in Fig. 6 to make $|dom(dstIP)/\sim_f|$ = $|dom(dstPort)/\sim_f|$ = 100 which is the domain size of `dstIP` and `dstPort`.

dstPort	dstIP	dstCond
10	10.0.0.0	1
10	10.0.0.1	0
…	…	…
10	10.0.0.99	0
11	10.0.0.0	0
11	10.0.0.1	1
…	…	…
11	10.0.0.99	0
…	…	…
199	10.0.0.0	0
199	10.0.0.1	0
…	…	…
199	10.0.0.99	1

Fig. 6. A table matching `dstIP` and `dstPort` outputs `dstCond`. For each group of table entries with the same `dstPort` field, there is only one entry whose `dstCond` field is 1 and the entry has different `dstIP` values for each group.

We therefore bound a $\tau(f)$ by defining the bounding characteristic function of a routing function $\tau_G(f)$ which is easily derivable from f's DFG. This function characterizes an upper bound on f's computation load: $\tau_G(f)$ dominates $\tau(f)$.

We find $\tau_G(f)[scope]$ as before. Instead of calculating $\tau_G(f)[ec]$, however, we determine an upper bound for with the value of specific vertex cut in G_f, f's DFG. We now construct this cut.

Definition 11. *Let $V_f(M)$ be the vertices of $m_i \in M$ in G_f, and $D_f(M)$ be the vertices in G_f descended from $V_f(M)$.*

The vertex-min-cut of M, $G_f.vertexMinCut(M)$*, is the product of the weights of the vertices in the minimum weight vertex cut severing* $V_f(M)$ *from* $D_f(\mathcal{M} - M)$*.*

Given this cut, we define $\tau_G(f)$ follows:

Definition 12. The characteristic function $\tau_G(f)$ of a routing function *characterizes an upper bound on* f*'s computational load;* $\tau_G(f)$ *dominates* $\tau(f)$*:*

$$\tau_G(f)(M) \triangleq (dom(M),\ G_f.vertexMinCut(M)).$$

Example: We illustrate these concepts with our example routing function `onPkt`.

Consider `onPkt`'s match fields `srcIP` and `srcPort`. Each are only read once: on `L3`, by the boolean function `isVerified`. Thus, while `srcIP` and `srcPort` may have many f-equivalence classes individually, (`srcIP`, `srcPort`) only has two: values that `isVerified` evaluates to 0, and values it evaluates as 1.

Suppose `onPkt` is a routing function for a small commercial network fronted by a NAT with 50 hosts each running a limited set of applications that only use 200 standard ports. Given this, $dom(\texttt{srcIP}, \texttt{srcPort}) = 10000$, and thus $\tau(\texttt{onPkt})(\texttt{srcIP}, \texttt{srcPort}) = (10000, 2)$.

While the equivalence class number of (`srcIP`, `srcPort`) was straightforwards, the equivalence class number of most other subsets of `onPkt`'s inputs is not so obvious. We therefore bound $\tau(\texttt{onPkt})$ with $\tau_G(\texttt{onPkt})$, which we calculate using `onPkt`'s DFG $G_{\texttt{onPkt}}$, shown in Fig. 7.

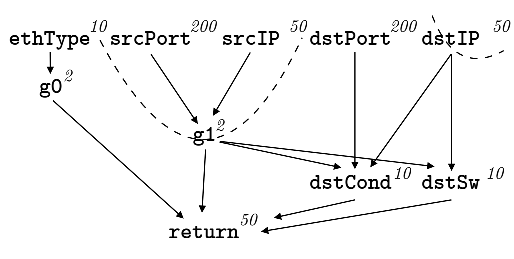


Fig. 7. The routing function `onPkt`'s DFG $G_{\texttt{onPkt}}$ and the cut $(\texttt{srcIP}, \texttt{srcPort}, \texttt{dstPort})$.

To bound, for example, the equivalence class number of `onPkt`'s inputs (`srcIP`, `srcPort`, `dstIP`) we take the vertex-min-cut in $G_{\texttt{onPkt}}$ between their vertices and every vertex descended from `onPkt`'s other inputs: (`ethType`, `dstPort`,`g0`, `dstCond`, `return`). This vertex-min-cut is indicated in Fig. 7 by a dotted line.

The vertices in this cut, $(\mathtt{g1}, \mathtt{dstIP})$ have weight 50 and 2, and thus $\tau_G(\mathtt{srcIP}, \mathtt{srcPort}, \mathtt{dstIP}) = (50000,\ 100)$.

D. Characterization of a Pipeline

We now define $\kappa(p)$, the set of characteristic functions of a pipeline p. We start by defining a path ρ through a pipeline p.

Definition 13. *A path, ρ, in a p is a path through p's dag $\langle t_1, ..., t_n\rangle$ such that t_1 is a root table and t_n an egress table in the p.*

As an example, `ExampleDP` contains two paths: $\langle \mathtt{t1}, \mathtt{t2}, \mathtt{t3}, \mathtt{t4}, \mathtt{t5}\rangle$, and $\langle \mathtt{t1}, \mathtt{t6}\rangle$, which we denote as ρ_{L2} and ρ_{L3} respectively.

We define, $\forall\ \rho \in p,\ \kappa_\rho(p)$ as the characteristic function of the a path through a pipeline.

Definition 14. The characteristic function set $\kappa(p)$ *of a p is the union of $\forall\ \rho \in p$'s characteristic functions:*

$$\kappa(p)(M) \triangleq \{c \in \mathcal{C} : c = \kappa_\rho(\rho)\ \forall\ \rho \in p\}.$$

We now construct the characteristic function of a path ρ by introducing the following definitions:

Definition 15. *The* input closure $\bar{M}_\rho(t_i)$ *of a table $t_i \in \rho$ is the set of inputs that t_i can obtain information about:*

$$\bar{M}_\rho(t_i) \triangleq \{m_i \in \mathcal{M} : m_i \in I(t_i)\ \vee$$
$$m_i \in \bar{M}_\rho(t_j)\ s.t.\ r(t_j) \in I(t_i)\}.$$

Definition 16. *The* closure set, $\bar{\bar{M}}_\rho(M)$ *of a ρ's M is the set of $t_i \in \rho$ with input closure M.*

$$\bar{\bar{M}}_\rho(M) \triangleq \{t_i \in \rho : \bar{M}_\rho(t_i) = M\}.$$

Using these definitions, we define the characteristic function of a ρ as:

Definition 17. The characteristic function $\kappa_\rho(\rho)$ of a ρ *characterizes the computational capacity of a ρ.*

$\kappa_\rho(\rho)[scope]$ is the maximum number of values of M that ρ can read and $\kappa_\rho(\rho)[ec]$ is the maximum number of equivalence classes of M that ρ can distinguish.

$$
\kappa_\rho(\rho)(M) \triangleq
\begin{cases}
\bar{\bar{M}}_\rho(M) \neq \varnothing & (min[maxrules(t_i) : t_i \in \bar{\bar{M}}_\rho(M)], \\
& \quad min[2^{bits(r(t_i))} : t_i \in \bar{\bar{M}}_\rho(M)]) \\
\bar{\bar{M}}_\rho(M) = \varnothing \wedge \exists\ m_i \in M : & \\
m_i \notin \bigcup_{t_i \in \rho} \bar{M}(t_i, \rho) & (1, 1) \\
otherwise, & (\top, \top).
\end{cases}
$$

Example: As before, we provide intuition into the characteristic functions of pipelines using our example pipeline `ExampleDP`.

Recall from our model that `ExampleDP` contains two ρ: ρ_{L2} and ρ_{L3}. Consider the table `t4`, only contained by ρ_{L3}. The input closure $\bar{M}_{\rho_{\text{L3}}}(\texttt{t4})$ is (`srcIP`, `srcPort`, `dstIP`) since `t4` reads `dstIP` and `r(t2)`, and `t2` in turn reads `srcIP` and `srcPort`. The closure set, $\bar{\bar{M}}_{\rho_{\text{L3}}}(\texttt{srcIP}, \texttt{srcPort}, \texttt{dstIP})$, of `t4`'s inputs in ρ_{L3} is $\{\texttt{t4}\}$: `t4`'s input closure is unique.

Thus, $\kappa_{\rho_{\text{L3}}}(\bar{M}_{\rho_{\text{L3}}}) = \kappa_{\rho_{\text{L3}}}(\texttt{srcIP}, \texttt{srcPort}, \texttt{dstIP}) = (maxRules(\texttt{t4}),\ 2^{bits(r(\texttt{t4}))})$. In the case that `t4` has 2^{20} rules and a 16 bit output register, $\kappa_{\rho_{\text{L3}}}(\bar{M}_{\rho_{\text{L3}}}) = (2^{20}, 2^{16})$.

Further, consider the subset of `ExampleDP`'s match fields (`srcMac`, `dstMac`). ρ_{L3} does not contain the inputs `srcMac` or `dstMac` and thus it can only realize functions that contain them in the unlikely event that all are constants. Constants have domain 1 and 1 equivalence class. Thus the value of $\kappa_{\rho_{\text{L3}}}$ for any set of outputs containing `srcMac` is $(1, 1)$.

Finally, consider the subset of `ExampleDP`'s match fields (`srcIP`, `srcPort`). `srcIP` and `srcPort` are both read by ρ_{L3}, but (`srcIP`, `srcPort`) is not an input closure of any $t_i \in \rho_{\text{L3}}$. In this case, it is not necessary to consider (`srcIP`, `srcPort`) to verify realizability, and thus $\kappa_{\rho_{\text{L3}}}(\texttt{srcIP}, \texttt{srcPort}) = (\top, \top)$, indicating that we can skip this field during comparison with a routing funciton's τ.

E. Datapath Programming Capacity Theorems

Combining the preceding definitions to characterize both routing functions and pipelines, we finally arrive at our central result: a sufficient condition for whether a given f can be realized in a given p.

Theorem 1 (Pipeline Realization Theorem)**.** *A routing function f can be realized by a pipeline p if $\kappa(p)$, the set of characteristic functions of p dominates $\tau(f)$, the characteristic function of f. Formally, we have:*

$$\kappa(p) \trianglerighteq \tau(f) \Rightarrow f \Rightarrow p.$$

As a corollary, because $\tau_G(f) > \tau(f)$, the Pipeline Realization Theorem extends to $\tau_G(f)$.

Example: We illustrate our Pipeline Realization Theorem using `onPkt` and `ExampleDP`. Specifically, our Pipeline Realization Theorem states that $\kappa(\texttt{ExampleDP}) \trianglerighteq \tau(\texttt{onPkt}) \Rightarrow \texttt{ExampleDP} \Rightarrow \texttt{onPkt}$.

Further, $\kappa(\texttt{ExampleDP}) \trianglerighteq \tau(\texttt{onPkt})$ is true if $\kappa_\rho(\rho_{\mathrm{L2}}) > \tau_G(\texttt{onPkt})$ or $\kappa_\rho(\rho_{\mathrm{L3}}) > \tau_G(\texttt{onPkt})$ We verify each conditional by comparing each component of each vector given by each pair of characteristic functions. For example, $\tau(\texttt{onPkt})(\texttt{srcIP}, \texttt{srcPort}, \texttt{dstIP}) = (50000,\ 100)$, $\kappa_\rho(\rho_{\mathrm{L3}})(\texttt{srcIP}, \texttt{srcPort}, \texttt{dstIP}) = (2^{20}, 2^{16})$, and thus the input set (`srcIP`, `srcPort`, `dstIP`) does not prevent `onPkt` from being realized in ρ_{L3}.

Tightness: Though the theorem provides only a sufficient condition, tighter results, in particular sufficient and necessary conditions, can be established in multiple settings. In particular, we have the following result:

Definition 18. *A branchless pipeline p is a p whose dag is a path from its root to its output node.*

Theorem 2 (Extension of Pipeline Realization Theorem)**.** *If p is a branchless pipeline, p's table size is large, and each match field $m_i \in \mathcal{M}$ appears in exactly one of p's tables, $\kappa(p) \trianglerighteq \tau_G(f) \Leftrightarrow f \Rightarrow p$.*

In Sec. IV, we provide the proofs of our capacity theorems.

IV. Proofs

A. Proof of Pipeline Realization Theorem

We now present proofs to verify our realization theorem. The structure of these proofs will be as follows. First, we define a mechanism to encode sufficient information about a given $M \in \mathcal{M}$ to fully execute a given f. Second, we show that a pipeline transmitting information internally using our encoding can realize an f in p given that $\kappa(p) \trianglerighteq \tau_G(f)$. Finally, we show that $\tau_G(f) > \tau(f)$, proving by extension that if $\kappa(p) \trianglerighteq \tau(f), f \Rightarrow p$. We omit a proof of Theorem 2 and the proofs of certain corollaries and lemmas due to space constraints. We will give these proofs in an extended report in an upcoming technical journal.

We base our summary on the vertices in the $G_f.vertexMinCut(M)$ of a f's G_f.

Definition 19. *The* min cut vertices $\mu_f(M)$ *are the vertices in an f's G_f cut by $G_f.vertexMinCut(M)$.*

Let a given value of $\mu_f(M)$ be $v_i(\mu_f(M))$ and the domain of values of $\mu_f(M)$ be $dom(\mu_f(M))$.

Lemma 1. *Given a $G_f.vertexMinCut(M)$, we can calculate f without knowing $v_i(M)$ given $v_i(\mu_f(M))$.*

Proof. We can calculate any DFG G's output given the values of all of its roots because every vertex in G must be descended from a subset of these roots.

Consider the subgraph of G_f, $G_{f,M}$, generated by removing every vertex in G_f that $\mu_f(M)$ separates from $D_f(M)$. By our definition of G_f, G_f's output node is descended from $\forall\ m_i \in \mathcal{M}$, and thus it is always in $G_{f,M}$.

Each of $G_{f,M}$'s roots is either a $m_j \in \mathcal{M} - M$ or some vertex in $\mu_f(M)$. Therefore, given $v_i(\mathcal{M} - M)$ and $v_j(\mu_f(M))$, we can calculate $G_{f,M}$'s output, and therefore G_f's output. By the definition of a DFG, G_f's output is f's output, and therefore we have shown that we can calculate f given $v_j(\mu_f(M))$ in lieu of $v_j(M)$. $\square$

Further, we have the bound of the number of equivalence classes of M.

Lemma 2. *The number of equivalence classes of M is bounded by $dom(\mu_f(M))$.*

Proof. Suppose, by way of contradiction, $\exists\ (f,\ M) : |dom(M)/ \sim_f\ | > dom(\mu_f(M)$. Each $v_i(M)$ in one of M's equivalence classes must generate a $v_i(\mu_f(M))$. By the pigeonhole principle, if M has more equivalence classes than $\mu_f(M)$, two values of M from different equivalence classes must generate the same value of $\mu_f(M)$. However, by Lemma 1, $\mu_f(M)$ contains sufficient information about M to fix f's outputs value, and thus these two bindings of M must be in the same equivalence class, which is a contradiction. $\square$

Then, we show that the $\mu_f(M)$ representation (*i.e.*, min cut) is a bound on the number of equivalence classes of M.

Proof. By Lemma 2, $|dom(M)/\sim_f\ | \leqslant |dom(\mu_f(M))/\sim_f\ |$. Further, $|dom(\mu_f(M))\ \sim_f\ | < |dom(\mu_f(M))|$. Finally, $|dom(\mu_f(M))| < \prod_{\mu_{f_i} \in \mu_f(M)} dom(\mu_{f_i})$, which is the value of $G_f.vertexMinCut(M)$. □

While $\mu_f(M)$ acts as an effective representation of values of M $v_i(M)$, we can compress it by introducing the concept of codewords, allowing us to maximize transmission through a pipeline.

Definition 20. *The* codewords $\chi_f(M)$ *of inputs* M *of a* f *are a set of integers that correspond to the f-equivalence classes of* M.

Receiving a codeword $\in \chi_f(M)$ is equivalent to receiving a value for M $v_i(M)$, since the codeword can be deterministically mapped back into a value from $v_i(M)$'s equivalence class. We now define the shorthand '*compute the codewords of* M' which we will use in our proofs:

Definition 21. *If we can* compute the codewords $\chi_f(M)$ *of* M*,* $\forall\ v_i(M) \in dom(M)$ *we can compute the codeword associated with the equivalence class of* $v_i(M)$.

Our codewords give us a bound on the transmission requirements of an M, given in Lemma 2.

Lemma 3. *A table* t_i *only requires* $log_2(\lceil |dom(\mu_f(M))| - 1 \rceil)$ *bits of information about* M *to execute* f *correctly.*

Proof. We can encode the value of any $v_i(M) \in dom(M)$ as a codeword in $\chi_f(M)$ and still convey sufficient information to compute f. If $\mu_f(M)$ can take $|dom(\mu_f(M))|$ distinct values, we can assign each value a unique codeword from the set [0, ..., $|dom(\mu_f(M))| - 1$], which take at most $log_2(\lceil |dom(\mu_f(M))| - 1 \rceil)$ bits to represent. □

Proving the realization theorem: Given our characterization of function transmission requirements, we can now embark on our proof of our realization theorem. First, we will give our key underlying lemma, lemma 3, from which our realization theorem follows naturally.

Lemma 4. *If* $\forall\ t_i \in \rho = \langle t_1, ..., t_n \rangle$ *have* $maxRules(t_i) > \tau_G(f)(\bar{M}_\rho(t_i))[dom]$*, and* $2^{r(t_i)} > \tau_G(f)(\bar{M}_\rho(t_i))[ec]$*, then* $\forall t_i \in \rho = \langle t_1, ..., t_n \rangle$ *can output* $\chi_f(\bar{M}_\rho(t_i))$ *to* $r(t_i)$.

Proof. We prove Lemma 4 by induction.

Base case: Assume ρ only includes one table t_m. Then, we have $m_i \in \bar{M}_\rho(t_m) \Leftrightarrow m_i \in I(t_m)$. Now we construct a table by assigning each value in [0, ..., $|dom(\mu_f(\bar{M}_\rho(t_m)))|-1$] to each match in $\bar{M}_\rho(t_m)$. Then, the table has that the number of rules equals $\tau_G(f)(\bar{M}_\rho(t_m))[dom]$ and the size of $r(t_m)$ equals $\tau_G(f)(\bar{M}_\rho(t_m))[ec]$. Therefore, if $maxRules(t_m) > \tau_G(f)(\bar{M}_\rho(t_m))[dom]$, and $2^{r(t_m)} > \tau_G(f)(\bar{M}_\rho(t_m))[ec]$, t_m can output $\chi_f(\bar{M}_\rho(t_m))$ to $r(t_i)$.

Inductive step: Assume Lemma 4 is true for $\langle t_1, ..., t_k \rangle$. We show Lemma 4 is true for $\langle t_1, ..., t_{k+1} \rangle$.

We prove the inductive case in two stages. First, we show that if $maxRules(t_1) > \tau_G(f)(\bar{M}_\rho(t_1))[dom]$, t_1 can compute $\chi_f(\bar{M}_\rho(t_1))$. Second, we show that given that t_1 can compute $\chi_f(\bar{M}_\rho(t_1))$, if $2^{r(t_1)} > \tau_G(f)(\bar{M}_\rho(t_1))[ec]$, $\chi_f(\bar{M}_\rho(t_1))$ can be output by t_1 to $r(t_1)$.

We start by proving stage 1. $m_i \in \bar{M}_\rho(t_{k+1}) \Leftrightarrow m_i \in I(t_{k+1}) \vee m_i \in \bar{M}_\rho(t_i) : r(t_i) \in I(t_{k+1})$. $r(t_i) \in I(t_{k+1}) \Leftrightarrow r(t_i) \in (r(t_1), ..., r(t_{k+1}))$.

By the inductive hypothesis, $r(t_i) \in (r(t_1), ..., r(t_{k+1})) \Rightarrow r(t_i)$ will contain $\chi_f(\bar{M}_\rho(t_i))$. Therefore, t_{k+1} can read $\forall\ m_i \in \bar{M}_\rho(t_{k+1})$ from $m_i \in I(t_{k+1}) \vee r(t_i) \in I(t_{k+1})$.

Now, as in the base case, if a table t_m is given $\forall\ m_i \in \bar{M}_\rho(t_{k+1})$, it can compute $\chi_f(\bar{M}_\rho(t_{k+1}))$ by matching on all $v_i(\bar{M}_\rho(t_{k+1})) \in dom(\bar{M}_\rho(t_{k+1}))$ if $maxRules(t_m) > dom(\bar{M}_\rho(t_{k+1}))$.

We now show how to transform t_m into t_{k+1} without increasing t_m's rule number. $\forall\ m_i \in \bar{M}_\rho(t_{k+1})$:

- If $m_i \in I(t_{k+1})$, we can leave m_i's column in t_m alone.
- If $m_i \notin I(t_{k+1}) \Rightarrow r(t_i) \in I(t_{k+1}) : r(t_i)$ contains $\chi_f(M_j) : m_i \in M_j$. Further, $M_j \subseteq \bar{M}_\rho(t_{k+1})$. Thus, each rule in t_m generates precisely one codeword in $\chi_f(M_j)$. We therefore replace t_m's match field header m_i with $\chi_f(M_j)$, and that each of that header's values in t_m's rules with that rule's codeword in $\chi_f(M_j)$.

This transformation does not increase rule number and results in a table that only matches on headers in $I(t_k)$. Thus we have proved stage 1.

We now proceed to proving stage 2. $\chi_f(M)$ only requires $\lceil log_2(|M/\sim_f|)\rceil$ bits to represent it. Therefore $2^{r(t_{k+1})} > |\bar{M}_\rho(t_{k+1})/\sim_f| \Rightarrow \chi_f(\bar{M}_\rho(t_{k+1}))$ can be placed in $r(t_{k+1})$. $\tau_G(f)(\bar{M}_\rho(t_{k+1}))[ec] \leqslant 2^{r(t_{k+1})}$. □

Given Lemma 4, we are now equipped to prove the realization theorem.

Proof. Given an f and p, we will prove that if $\kappa(p) \trianglerighteq tau(f)$, $f \Rightarrow p$. Consider a $\kappa_\rho(\rho) \in \kappa(p)$.

$\forall\ M \in \mathcal{M} : m_i \in M \rightarrow m_i \notin \bigcup_{t_i \in \rho} \bar{M}_\rho(t_i),\ \kappa_\rho(\rho)(M) = (1,\ 1)$. Therefore, if $\kappa_\rho(\rho) > \tau_G(f) \Rightarrow$ all m_i not read by ρ are treated as constants or not read at all by f, and thus f is effectively a mapping from $\bigcup_{t_i \in \rho} \bar{M}_\rho(t_i) \rightarrow \mathcal{R}$.

Further, given $\kappa_\rho(\rho) > \tau_G(f)\ \forall\ t_i \in \rho,\ maxRules(t_i) > \tau_G(f)(\bar{M}_\rho(t_i))[dom]$, and $2^{r(t_i)} > \tau_G(f)(\bar{M}_\rho(t_i))[ec]$, and thus by Lemma 4 t_n can calculate $\chi_f(\bar{M}_\rho(t_n))$.

Finally, consider that if a t_i can calculate $\chi_f(M_i)$, and an f is a mapping $dom(M_i) \rightarrow \mathcal{R}$, t_i can compute f's output $\forall\ v_j(M_i) \in dom(M_i)$ by mapping each codeword in $\chi_f(M_i)$ to the output of f it corresponds to.

Since t_n is ρ's only output, $\bar{M}_\rho(t_n) = \bigcup_{t_i \in \rho} \bar{M}_\rho(t_i)$. Thus, t_n can compute f's output. Further, since t_n is an egress table it can always pass this output back to the switch.

Therefore, if $\kappa_\rho(\rho) > \tau_G(f), f \Rightarrow \rho$. Since $\kappa_\rho(\rho) \in k(p)$ and $\rho \in p$, we have proved that if $\kappa(p) \trianglerighteq \tau_G(f),\ f \Rightarrow p$. $\square$

The last step required to prove our realization theorem is to show that $\tau_G(f) > \tau(f)$ and thus that $\kappa(p) \trianglerighteq \tau(f) \Rightarrow f \Rightarrow p$. The crux of this step is given in Lemma 2.

Corrollary 1. *The number of f-equivalence classes of any M is bounded by $G_f.vertexMinCut(M)$.*

Corrollary 2. *The characteristic function $\tau_G(f)$ dominates the characteristic function $\tau(f)$.*

We have therefore proven our realization theorem: that $\kappa(p) \trianglerighteq \tau_G(f) \Rightarrow f \Rightarrow p$.

B. Proof of Extension of Pipeline Realization Theorem

As our realization theorem, $\kappa(p) \trianglerighteq \tau_G(f) \Rightarrow f \Rightarrow p$, has been proved, now we give a proof for that if p is a branchless pipeline, p's table size is large, and each match field $m_i \in \mathcal{M}$ appears in exactly one of p's tables, $f \Rightarrow p \Rightarrow \kappa(p) \trianglerighteq \tau_G(f)$.

We consider the contradiction that $f \Rightarrow p$ but there exists M that $\kappa(p)(M)[ec] < \tau_G(f)(M)[ec]$ (Note that here we omit the domain size of M as the p's table size is large). As $f \Rightarrow p$ and p is a branchless pipeline and each match field $m_i \in \mathcal{M}$ appears in exactly one of p's tables, we can have $\kappa(p)(M)[ec] \geqslant \prod \kappa(p)(m_i)[ec]$ where $m_i \in M$. Then, we have $\kappa(p)(M)[ec] \geqslant \tau_G(f)(M)[ec]$ which has contradiction with $\kappa(p)(M)[ec] < \tau_G(f)(M)[ec]$. Therefore, we have the proof.

V. Evaluation

We now evaluate the tightness, time complexity and output size of routing function, pipeline characterization, and the main factors of realization numerically. All experiments are conducted on a 1.6 GHz Intel Core i5 with 4 GB RAM.

A. Routing Function Characterization Tightness

We demonstrate the tightness of characterization of a routing function by comparing the $ec(M)$, the number of equivalent classes for a set of matches M, from two mappings (*i.e.*, τ and τ_G) for several routing functions. The computation of $ec(M)$ for the mapping τ is based on the definition of f-equivalence which will get the exact value of the number of equivalent classes for a set of matches M for a particular routing function. The routing functions we use are shown as follows :

```
//  Routing function: simpleRoute
L0: def simpleRoute(Addr srcIP, Addr dstIP):
L1:   srcSw = hostTbl[srcIP]
L2:   dstSw = hostTbl[dstIP]
L3:   route = routeTbl[srcSw, dstSw]
L4:   return route
```

Our first function, `simpleRoute` maps a packet's `srcIP` and `dstIP` to their host packet switches `dstSw` and `srcSw` based on a key-value table `hostTbl`. After that, `simpleRoute` gets the route from another table, `routeTbl`, where the key is two internal variables, `srcSw` and `dstSw`.

```
//  Routing Function: condRoute
L0: condRoute(srcIP, dstIP):
L1:   srcSw = hostTbl[srcIP]
L2:   dstSw = hostTbl[dstIP]
L3:   routeCond = condTbl[srcIP, dstIP]
L4:   route = routeTbl[srcSw, dstSw, routeCond]
L5:   return route
```

Our second function `condRoute` extends `simpleRoute` by introducing a route condition variable (*i.e.*, `routeCond`) which is computed by a condition table, `condTbl`. After that, `condRoute` computes the route from three internal variables, `srcSw`, `dstSw`, and `routeCond`.

```
//  Routing Function: secureRoute
```

```
L0: secureRoute(Addr srcIP, Addr dstIP):
L1:   if (isFiltered(srcIP)):
L2:     return Drop()
L3:   else:
L4:     route = fwdTbl[dstIP]
L5:     return route
```

Our third function `secureRoute` drops all traffic from `srcIP`s on a filter list (*i.e.*, `isFiltered`) and forwards remaining traffic based on the table `fwdTbl` by only matching `dstIP`.

As our final function, we consider the example function `onPkt` as shown in Sec. II

Results: We present our results in Table II. Specifically, in Table II, column 2 defines the domain of `srcIP`, `dstIP`, columns 3-6 give the output ranges $O(tbls)$ of each table, and columns 7-10 give the values of selected fields in each function's $\tau(f)$ and $\tau_G(f)$.

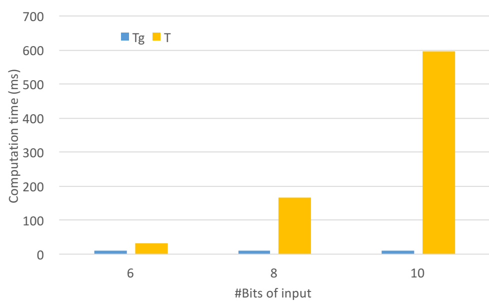
Note that the notation $b1(\texttt{scR})$ represents the branch 1 in `scR`, *i.e.*, L1→L2, while $b2(\texttt{scR})$ represents the branch 2 in `f3`, *i.e.*, L1→L3→L4→L5. We record N/A when a value is not applicable to a given function, and null for values of $\tau(f)$ where computation failed to halt. And we only evaluate the equivalent class value in the output vector of characterization of a function.

In our evaluation of `smplR` and `condR`, the results of τ and τ_G are almost identical in every case barring an extreme one where $O(\texttt{condTbl})$ = 1. Notably, our values of $\tau(f)$ and $\tau_G(f)$ are not influenced by the range of `routeTbl` (*i.e.*, $O(\texttt{routeTbl})$), which can be observed from the first three rows in the table. This implies there is no pattern between the allocation of routes to (`srcIP`, `dstIP`) pairs $\tau(f)$ can exploit to reduce its number of equivalence classes. Considering s_1 and s_2 as two values of `srcIP`, if `hostTbl`(s_1) is not equal to `hostTbl`(s_2), then the chance of $s_1 \sim_f s_2$ is very small.

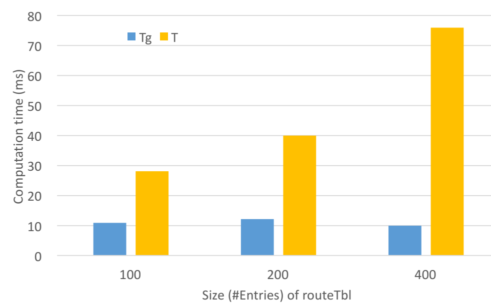
Further notice that our functions with control statements: `secureRoute` and `onPkt` have a large gap between τ and τ_G, suggesting τ_G's bound is loose on heavily branching programs. However, as rows b1(scR) and b2(scR) show, we can ameliorate this problem by calculating each route through a program characteristic function separately. For each branch, the result of τ and τ_G maintains the tightness, which is shown in the rows with $b1(\texttt{scR})$ and $b2(\texttt{scR})$ in Table II. Also, we evaluate the example routing function `onPkt`, and the result is shown in the last row in the table. Need to mention that, both $\tau(f)$(`srcIP`) and $\tau(f)$(`dstIP`) are null means that the computation time is too long to compute the result.

f	$bits(IP)$	O(hostTbl)	O(routeTbl)	O(condTbl)	O(fwdTbl)	τ(f)(srcIP)	τ(f)(dstIP)	τ_G(f)(srcIP)	τ_G(f)(dstIP)
smplR	10	100	2	N/A	N/A	100	100	100	100
smplR	10	100	30	N/A	N/A	100	100	100	100
smplR	10	100	5000	N/A	N/A	100	100	100	100
smplR	12	100	30	N/A	N/A	100	100	100	100
smplR	10	200	30	N/A	N/A	200	200	200	200
condR	10	100	30	50	N/A	1024	1024	1024	1024
condR	10	100	30	5	N/A	1024	1024	1024	1024
condR	10	100	30	1	N/A	100	100	1024	1024
scR	10	N/A	N/A	N/A	100	2	100	2	1024
b1(scR)	10	N/A	N/A	N/A	N/A	1	N/A	1	N/A
b2(scR)	10	N/A	N/A	N/A	100	1	100	1	100
onPkt	32	100	30	50	N/A	null	null	2^{32}	2^{32}

TABLE II
CHARACTERIZATION RESULTS OF ROUTING FUNCTIONS WITH DIFFERENT STATISTICS.



(a) Input size



(b) Table size

Fig. 8. Computation time required to generate τ(simpleRoute) and τ_G(simpleRoute) as input bit length varies and the size of table varies.

B. Routing Function Characterization Time Complexity

We now examine the computing complexity of characterization of a routing function by comparing the computation time of two mappings (*i.e.*, τ and τ_G) for the same function. As the same with the first evaluation, the computation of τ is based on the definition of f-equivalence. We run our tests using `simpleRoute` where the size of `hostTbl` = 100 with different size of domain of input (*i.e.*, the bits of input) and different size of `routeTbl`.

Results: Fig. 8 shows the scalability of τ_G as input size and `routeTbl` size grows. Specifically, Fig. 8(a) gives the result with fixed `routeTbl` size which equals 100, and Fig. 8(b) gives the result with fixed input size which equals 6. As shown in Fig. 8(a), as the bit length of `srcIP` and `dstIP` increases, τ_G's computation time remains constant while τ's computation time grows in exponential order. And as illustrated in Fig. 8(b), as `routeTbl` size grows, the τ_G's computation time still remains constant while τ's computation time grows in polynomial level. This is because the computation of τ_G only requires the cuts of DFG which can be done in a very fast way but the computation of τ requires the execution of function for every possible input.

Pipeline	#Paths	Time (ms)	#Valid M	$\#M$
`ExampleDP`	3	8	6	22
PicOS BR	4	13	19	$3 * (2^{24}) + 2^7$
PicOS OT	2	7	5	$2^{24} + 16$
Broadcom IPR	1	7	4	2^7
Broadcom PR	3	9	14	$2 * (2^{24}) + 2^7$

TABLE III
CHARACTERIZATION RESULTS OF PIPELINES.

C. Characterization of a Pipeline

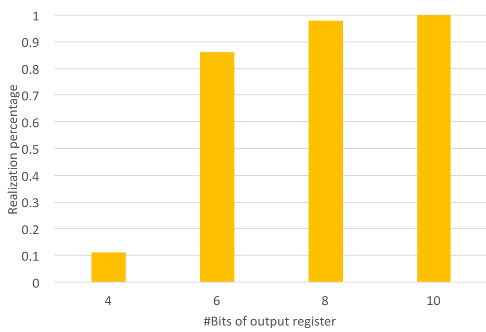
We now examine the characterization of a pipeline by evaluating the computation time and compactness of characterization functions of several pipelines. The compactness of a characterization function of a pipeline represents the output memory utilization of the characterization function compared with the size of all the possible M. We use the following pipelines in the evaluation:

1) **The OF-DPA Abstract Switch 2.0:** The OpenFlow Data Plane Abstraction Abstract Switch 2.0 (OF-DPA) is an abstract switch model based on the Open Flow 1.3.4 protocol designed to allow the programming of Broadcom-based switches under the OpenFlow protocol. We examine two OF-DPA flow table configurations: bridging and routing (BR), and data center overlay tunnel (OT), which contain 7, and 3 tables in 5, 3 stages respectively. [3]
2) **PicOS:** PicOS is a network operating system for white box switches that provides OF programmability across HP, Edgecore and Pica switches. We examine two fixed pipelines offered by PicOS as table type patterns: PicOS's IP routing pipeline (IPR) and Policy routing pipeline (PR), which contain 4 and 5 tables in 4 and 5 stages respectively. [9]

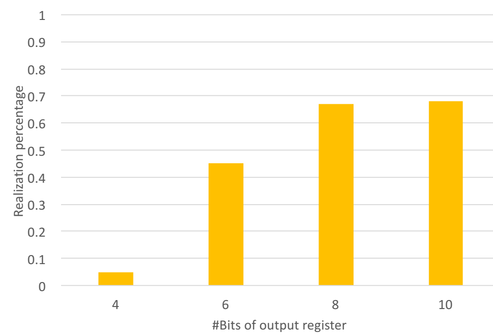
Results: Table III gives our characterization results for our evaluated pipelines. It shows the characterization results of several pipelines including four real pipeline structures and the example pipeline, `ExampleDP`. We say a set of packet match fields M is valid in a pipeline p means the value of $\kappa_\rho(\rho)(M)$ can be computed by the first formula in the definition of $\kappa_\rho(\rho)(M)$. The results show that despite the theoretically large number of subsets of $\mathcal{M}$ across evaluated pipelines, memory utilization and computation time are small.

D. Realization

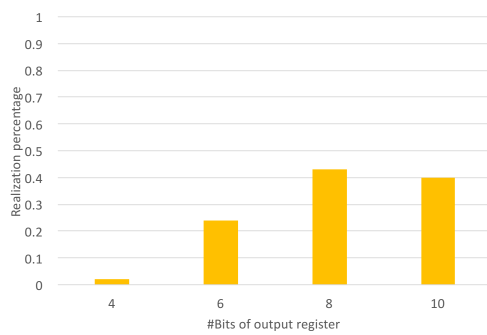
We now evaluate the percentage of successful realization of a function in a pipeline to see the factors of successful realization. We consider the example function `onPkt` with different



(a) #Tables = 2



(b) #Tables = 3



(c) #Tables = 4

Fig. 9. The percentage of realization for different number of tables and different length of registers.

content of tables. Specifically, for tables `condTbl` and `hostTbl`, we set the number of output values of tables ranging from 10 to 30 randomly, which means the domain size of `dstCond` and `dstSW` is from 10 to 30 (*i.e.*, the average domain size is 20). For the pipeline side, we randomly decide the number of tables (from 2 to 4) of a pipeline and the length of bits of registers (from 4 to 10) for each table. Also, we set the match fields of the generated pipeline must contain the five match fields required by `onPkt` and each match field can only appear in one table. Then, we compute the successful realization percentage of `onPkt` and the generated pipeline by using the realization theorem.

Results: The result is shown in Fig. 9. Specifically, Fig. 9(a) considers the pipeline with 2 tables; Fig. 9(b) considers the pipeline with 3 tables; Fig. 9(c) considers the pipeline with 4 tables. For each case, we compute the successful realization percentage with different length of bits of registers for each table. From the result of Fig. 9(a), we can find that a pipeline with more bits of registers can realize a function in a higher percentage. However, when the length is larger than a threshold, the successful realization percentage does not increase much. The threshold is determined by the size of domain of variables in the function. As shown in Fig. 9(a), the gap of successful realization percentage between 4 bits and 6 bits of length of registers (*i.e.*, the available size of equivalent classes from 16 to 64) can be explained by the fact that the average domain size of variables (*i.e.*, 20) is in the range between 16 and 64.

Furthermore, we can find the structure of pipelines also determines the realization percentage. A pipeline with 2 tables (*i.e.*, there is no branch in the pipeline) has a relatively high successful realization percentage, while a pipeline with 3 or 4 tables which may contain branches has a low realization percentage as the structure of the pipeline is not well organized (*i.e.*, the random mapping between matches fields and tables).

VI. Related Work

High level SDN Program Compilers: Multiple systems that allow programmers to write SDN programs in high level languages and then compile such programs to flow table pipelines have been proposed over the last several years. Such systems are related to our work in that they examine the transformation of policy programs into switch flow tables. We group these systems into two categories: *tier-less* and *split-level*.

Tier-less systems (*e.g.* SNAP [10], FML [11], FlowLog [12], Maple [8]), require programmers to specify forwarding behaviors as packet handling functions which are then used by the SDN controller to configure and update network state. Such systems pioneer our pipeline capacity theorem's notion of a *program function* and are able to compile such functions to single pipelines. These systems, however, are unable to verify that submitted functions can be written to a given pipeline without physically carrying out the time consuming process of compilation, and cannot write programs to multi-pipeline networks.

Split-level systems such as the Frentic family (*e.g.* Frenetic [7], Pyretic [13]) provide a two tiered programming model in which controller programs specify events of interest and then respond to these events when they occur by calculating new network policies. Again, such systems cannot verify that a given controller program's output can be written to the controller's switches' pipelines, although this paradigm falls outside of our pipeline capacity theorem's model as well.

Pipeline specification languages: There are some superficial similarities between pipeline specification languages (*e.g.* P4 [4] P5 [14], PISCES [15], Concurrent NetCore [16]) and our pipeline capacity theorem, such as the analysis and guarantees that such languages provide about pipeline behavior. For example, Concurrent NetCore's type system ensures that any program used to populate a pipeline has certain properties, such as determinism, whilst PISCES's switch specification allows compilers to analyze pipelines and optimize their performance. We contend, however, that our capacity theorem attacks an entirely different space in pipeline analysis - guaranteeing pipeline properties or improving performance is qualitatively different to verifying whether compilation is possible.

Pipeline design: Pipeline design schemes such as Jose et al.'s "Compiling Packet Programs to Reconfigurable Switches" [17], Sun et al.'s "Software-Defined Flow Table Pipeline" [18], FlowAdapter [19], and Domino [5] are clearly related to our pipeline capacity theorem in that they examine pipeline layout design under hardware constraints. Jose et al., Sun et al., and FlowAdapter however, focus on mapping logical lookup tables/flow table pipelines to physical tables whilst our pipeline capacity theorem focuses on generic programs, while Domino considers weaker hardware constraints (*e.g.* limits on stateful operations at line-rate) than our work does. Some other pipeline datapath design work (*e.g.* [20], [21], [22], [23], [24], [25]) focus more on the datapath configuration rather than the connection between high-level language and low-level datapath.

VII. Acknowledgement

The authors would like to thank Shenshen Chen who helped with initial discussions. The research was supported in part by NSFC grants NSFC #61672385, NSFC #61702373; Shanghai Key Project Grant #16511100900; NSF grants CC-IIE 1440745, CCF-1637385 and CCF-1650596; Google Research Award; and the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001.

References

[1] C. Leet, X. Wang, Y. R. Yang, and J. Aspnes, "Toward the first sdn programming capacity theorem on realizing high-level program on low-level datapaths," in *INFOCOM 2018 - IEEE Conference on Computer Communications, IEEE*, 2018.

[2] "OpenFlow Switch Specification Version 1.3.0," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf, ONF.

[3] (2014) OpenFlow Data Plane Abstraction (OF-DPA): Abstract Switch Specification Version 2.0. Broadcom. [Online]. Available: www.broadcom.com

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[5] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 15–28.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 99–110.

[7] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. [Online]. Available: http://doi.acm.org/10.1145/2034773.2034812

[8] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN programming using algorithmic policies," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. ACM, 2013, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486030

[9] (2015) Scaling up SDNs using TTPs (Table Type Patterns). Pica 8. [Online]. Available: www.pica8.com

[10] 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 Conference on ACM SIGCOMM 2016 Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934892

[11] T. Hinrichs, J. Mitchell, N. Gude, S. Shenker, and M. Casado, "Practical declarative network management," in *in ACM Workshop: Research on Enterprise Networking*, 2009.

[12] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 519–531. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616496

[13] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482629

[14] J. Lee, J. Lee, S. Banerjee, S. Banerjee, W. Wu, J. M. Kang, and A. Akella, "P5: Policy-driven optimization of p4 pipeline," in *Symposium on Sdn Research*, 2017, pp. 136–142.

[15] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "Pisces: A programmable, protocol-independent software switch," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 525–538.

[16] C. Schlesinger, M. Greenberg, and D. Walker, "Concurrent netcore: From policies to pipelines," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14. New York, NY, USA: ACM, 2014, pp. 11–24. [Online]. Available: http://doi.acm.org/10.1145/2628136.2628157

[17] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 103–115. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jose

[18] X. Sun, T. E. Ng, and G. Wang, "Software-Defined Flow Table Pipeline," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015, pp. 335–340.

[19] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie, "The FlowAdapter: Enable flexible multi-table processing on legacy hardware," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 85–90.

[20] H. Pan, G. Xie, Z. Li, P. He, and L. Mathy, "Flowconvertor: Enabling portability of sdn applications," in *INFOCOM 2017 - IEEE Conference on Computer Communications, IEEE*, 2017, pp. 1–9.

[21] H. Wang, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4," in *Symposium on Sdn Research*, 2017, pp. 122–135.

[22] P. G. K. Patra, F. E. R. Cesen, J. S. Mejia, and D. L. Feferman, "Towards a sweet spot of dataplane programmability, portability and performance: On the scalability of multi-architecture p4 pipelines," *IEEE Journal on Selected Areas in Communications*, 2018.

[23] R. Shah, A. Shirke, A. Trehan, M. Vutukuru, and P. Kulkarni, "pcube: Primitives for network data plane programming."

[24] W. L. da Costa Cordeiro, J. A. Marques, and L. P. Gaspary, "Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management," *Journal of Network and Systems Management*, vol. 25, no. 4, pp. 784–818, 2017.

[25] S. Pontarelli, M. Bonola, and G. Bianchi, "Smashing sdn" built-in" actions: Programmable data plane packet manipulation in hardware," in *Network Softwarization (NetSoft), 2017 IEEE Conference on*. IEEE, 2017, pp. 1–9.