electronics

MDPI

Article

Programming Protocol-Independent Packet Processors High-Level Programming (P4HLP): Towards Unified High-Level Programming for a Commodity Programmable Switch

Zijun Hang , Mei Wen *, Yang Shi and Chunyuan Zhang

School of Computer Science, National University of Defense Technology, Changsha 410073, China
* Correspondence: meiwen@nudt.edu.cn

Received: 27 July 2019; Accepted: 25 August 2019; Published: 29 August 2019

Abstract: Network algorithms are building blocks of network applications. They are inspired by emerging commodity programmable switches and the Programming Protocol-Independent Packet Processors (P4) language. P4 aims to provide target-independent programming neglecting the architecture of underlying infrastructure. However, commodity programmable switches have tight programming restrictions due to limited resources and latency. In addition, manufacturers tailor P4 according to their architecture, putting more restrictions on it. These intrinsic and extrinsic restrictions dilute the goal of P4. This paper proposes P4 high-level programming (P4HLP) framework, a suite of toolchains that simplifies P4 programming. The paper highlights three aspects: (i) E-Domino, a high-level programming language that defines both stateless and stateful processing of data plane in C-style codes; (ii) P4HLPc, a compiler that automatically generates P4 programs from E-Domino programs, which removes the barrier between high-level programming and low-level P4 primitives; (iii) modular programming that organizes programs into reusable modules, to enable fast reconfiguration of commodity switches. Results show that P4HLPc is efficient and robust, thus is suitable for data plane high-level programming. Compared with P4, E-Domino saves at least 5.5× codes to express the data plane algorithm. P4HLPc is robust to policy change and topology change. The generated P4 programs achieve line-rate processing.

Keywords: software-defined network; domain-specific language; P4; programmable switch

1. Introduction

A software-defined network (SDN) [1] stimulates the innovation of domain-specific languages (DSLs) [2–4] and switch architectures [5–8]. Programming Protocol-Independent Packet Processors (P4) is a DSL to program the data plane of programmable switches. P4 defines the data plane behavior of the programmable switches by primitives, regardless of the underlying architecture.

Although P4 (Programming Protocol-Independent Packet Processors) [9] provides abstractions for the data plane, switch chips put tight restrictions on P4 programming [10,11]. These restrictions can be on processing resources, storage resources, and latency; programmers can declare arbitrary numbers and combinations of the P4 hardware primitives; the programmers must handle read/write dependencies when using storage resources, and they must pay attention to latency restrictions to guarantee line-rate processing; in addition, manufacturers tailor P4 specifications for their architectures, putting even more restrictions on P4. Programmers also tail their programs for different architectures. These intrinsic and extrinsic restrictions dilute the goal of P4.

Network algorithms are the fundamental blocks of network applications. They provide a network-wide view via monitoring massive network traffic. Some algorithms were implemented in

software, which is weaker in speed and accuracy compared to programmable switches. Moving them to the data plane of the programmable switch is a better choice. In certain cases, we want to reproduce state-of-art hardware algorithms. Most importantly, we want to implement a new algorithm with ease. In this paper, we choose three types of network algorithms: (i) The basic algorithms which implement the single task and can be used as modules of other algorithms, e.g., the Bloom filter [12] and Flowlet [13]. (ii) The Heavy Hitter (HH) detection algorithms, e.g., SpaceSaving [14], HashPipe [15], Randomized Admission Policy (RAP) [16], and Probabilistic RECirculation admisSION (PRECISION) [10]. (iii) The network telemetry systems which are general-purpose algorithms, e.g., ElasticSketch [17] and SketchLearn [18].

However, programming these in P4 can be troublesome. Some algorithms are so complex that they augment the complexity of P4 programs. What is more, we have to write different P4 programs for different targets, as they use different architectural restrictions. Conventional wisdom concentrates on target-independent programming for control plane Application Programming Interfaces (APIs), i.e., P4Runtime APIs [19] which do not change with the P4 program. There is a gap between P4 data plane programming and the state-of-art commodity programmable switch. This paper aims to hide the restrictions on P4 data plane targets in order to provide unified high-level programming for commodity programmable switches, as follows.

- Inspired by the high-level synthesize [20] in Verilog programming and prior works [3,4,21], this paper proposes a domain-specific language (DSL), E-Domino, an enhanced version of Domino [4] that extends the grammar of Domino with loops, conditional statements, stateful atoms, and external functions. Also, E-Domino is C-like; thus, it is easy to use and lowers the learning curve.
- This paper devises P4HLPc (Programming Protocol-Independent Packet Processors High-Level Programming compiler), which automatically generates P4 programs from E-Domino programs. The generated P4 programs are eligible for programmable switches. P4HLPc removes the barrier between high-level programming and low-level primitives. P4HLPc comes with four fundamental techniques: loop unrolling, TBA (branch transform algorithm), atom template, and joint compiling.
- This paper proposes the thought of modular programming for commodity switches, which provides reusability for standard modules. Thus, P4 high-level programming (P4HLP) enables fast reconfiguration for commodity switches, while prior works [3,4,21] treat the programs as a whole.

We hope to support more hardware structures in future works. Currently, P4HLP targets the Barefoot Tofino ASIC (application specific integrated circuit) [5] which is under a RMT (reconfigurable match-action tables) structure. P4HLP explores enlarging the function of the RMT pipeline to implement network algorithms. This paper uses E-Domino to implement prior algorithms on a Barefoot Tofino programmable switch [5], evaluates the availability and functionality of P4 codes automatically generated by P4HLPc, compares E-Domino code with P4 code and evaluates P4HLPc under different conditions.

This paper is organized as follows: Section 2 presents background and some related works. Section 3 demonstrates our DSL, E-Domino, and an enhanced version of prior Domino. Section 4 illustrates P4HLPc design. Section 5 evaluates our design with existing algorithms. Section 6 presents some related works. Section 7 discusses our design and suggests some future works. Section 8 presents our materials and methods, and Section 9 concludes this paper.

2. Background

The Barefoot Tofino reconfigurable match-action tables (RMT) [5,22], Intel FlexPipe [6], Cavium XPliant Packet Architecture (XPA) [7], and Cisco Nexus [8] follow protocol-independent switch architecture (PISA), a flexible match-action pipeline that maintains comparable performance to fixed-function switches [23,24]. Yet to accommodate line-rate processing, the switches have complex constraints on their programmability.

2.1. Programmable Switches

A typical PISA switch (Figure 1) includes a programmable parser, ingress, queue, egress, and deparser. The parser and deparser are reconfigurable to support user-defined packet header formats. The ingress and egress pipelines process packets through match-action tables that are arranged in stages. Match-action tables match the header filled with pre-defined rules and performs the corresponding action on the packet. Actions use primitives to modify the non-persistent resources (headers or metadata) of each packet. In this paper, the word programmable switch denotes a PISA switch.

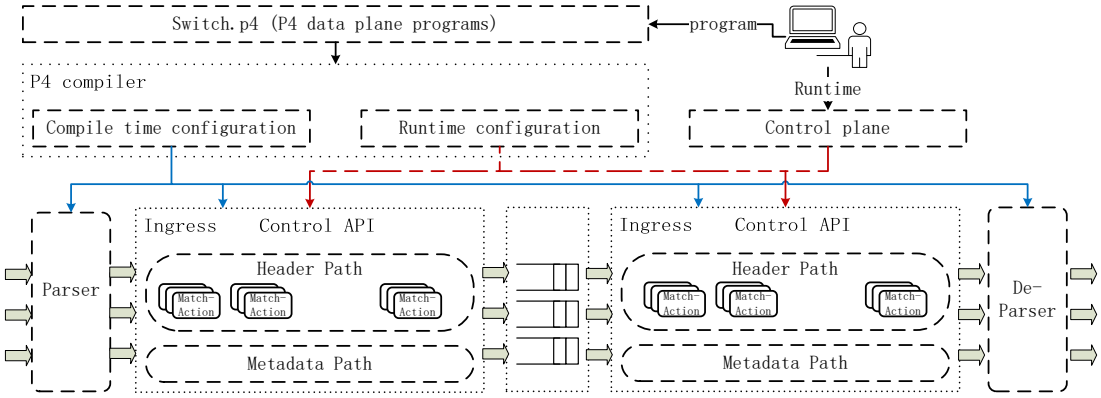

Figure 1. The architecture of a protocol-independent switch architecture (PISA) switch. The blue arrows show compiling time configuration, and the red arrows show runtime configurations. Programming Protocol-Independent Packet Processors (P4). Application Programming Interface (API).

Programmers configure switches by DSL [2–4] programs. P4 language describes the data plane of programmable switches and the interface by which the control plane and the data plane communicate [25].

The P4 compiler [26] generates target-specific data-plane configurations and P4Runtime APIs from P4 programs [27]. P4Runtime APIs are target-independent compiler outputs which define interfaces for controlling the data plane elements, to enable control plane functions. The control plane executes the PTF (packet test framework) [28] script to invoke APIs that are exposed over Thrift-RPC (Thrift Remote Procedure Call) [29], including port management, table entry modification, and register operation.

2.2. Restrictions

Processing packets at line-rate incurs rigid restrictions towards data plane programming, as summarized in our previous work [30]:

1. Simple operations. Each stage can only perform simple operations. Only primitive arithmetic is allowed in each stage, e.g., addition and subtraction; Branching on conditions of other registers is too costly to implement; There is no loop in P4 language.
2. Limited concurrent memory access. Each stage can access a few memory locations of different register arrays but only one location of a register array. For the SpaceSaving algorithm [14], finding the minimum value within an array is impossible.
   Also, two or more stages cannot access the same memory location. The packets are processed in different stages; read and write operations are paralleled. This restriction avoids the read after write and write after write hazards of different stages.
3. Limited stages. As the number of pipeline stages increases, the latency scales linearly and causes poor performance to real-time traffic. Thus, the number of stages in a pipeline is finite.

Programmers can not use arbitrary numbers and combinations of the P4 hardware primitives. Also, programmers must handle read/write dependencies when using storage resources. Programmers must pay attention to latency restrictions to guarantee line-rate processing. In addition, manufacturers

tailor P4 specifications for their architectures. Programmers have to tail their programs for different architectures.

For example, RMT uses a black box (Figure 2) to operate persistent resources on switch chips. A black box has two slices, the high slice, and the low slice. Each slice includes two SALUs (stateful arithmetic logic units) that perform different arithmetic operations, a conditional module that performs conditional judgments, and a three-state gate blocked by both *condition_hi* and *condition_lo*. While FlexPipe follows P4 specifications that use read and write primitives to operate the persistent resources. RMT can match on at most eight tables and modify at most eight fields in one stage, while FlexPipe has no analogous constraints.

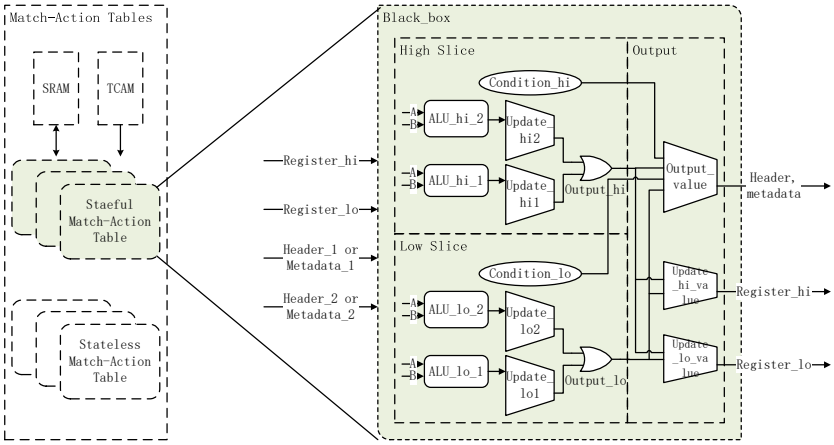

Figure 2. The structure of a black box. Arithmetic Logic Units (ALU).

P4HLP aims to hide the restrictions on P4 data plane architectures in order to provide unified high-level programming for commodity programmable switches. Currently we make a small step towards RMT architecture. The P4HLP and P4Runtime both aim to hide the target-specific properties in P4 programming. However, they are different, as the P4HLP targets data plane programming and P4Runtime targets control plane APIs. The P4HLP aims to be the front end of the P4 compiler through transforming high-level language to P4 programs, where the P4 compiler further compiles the P4 programs to data plane configurations and P4Runtime APIs.

3. E-Domino Design

3.1. E-Domino by Example

We present the design of E-Domino, taking the PRECISION [10] heavy hitter detection algorithm as an example, as shown in Algorithm 1. PRECISION measures the size of flows with the key-value pair register Key_i and Val_i. PRECISION consists of two paths: the regular path for measuring every packet (lines 1–14), and the recirculation path for updating statistics (lines 15–20). For each incoming packet whose flow key is *iKey*, PRECISION calculates the index l_i using the hash function h_i for $stage_i$ $(1 \leq i \leq d)$ (line 3). If *iKey* matches $Key_i[l_i]$, PRECISION sets the flag *matched* to true and adds one to flow size $Val_i[l_i]$ (lines 4–6). Otherwise PRECISION updates the minimum value *carry_min* and stage number *min_stage* (lines 7–9). If the packet has no matching key in any stages, PRECISION rounds the *carry_min* to the form of 2^k, i.e., the *new_val* which is $2^{\lceil log_2(carry\_min) \rceil}$ (line 11). Last, PRECISION recirculates the packet with the *new_val* and *min_stage* at a probability of $\frac{1}{new\_val}$ (lines 12–14). The recirculation path updates the minimum key and value according to *new_val* and *min_stage* in the recirculated packet (lines 15–19) and drops the cloned packet. (line 20).

We present E-Domino code in Figure 3. Combining the forwarding module and other transactions with the PRECISION algorithm into the ingress pipeline, we build a prototype of a switch.

Algorithm 1: PRECISION heavy hitter algorithm.

Input: Packet with iKey
Output: Packet Count

initialization: $carry\_min \leftarrow 0x7fffffff$;
for $i \leftarrow 1$ **to** d **do**
  $l_i \leftarrow h_i(iKey)$; ▷ for normal packets, find the minimum bucket
  if $Key_i[l_i] == iKey$ **then**
    $matched \leftarrow true$;
    $Val_i[l_i] \leftarrow Val_i[l_i] + 1$;
  else if $Val_i[l_i] < carry\_min$ **then**
    $carry\_min \leftarrow oval_i$;
    $min\_stage \leftarrow i$;
if $\neg matched$ **then**
  $new\_val \leftarrow 2^{\lceil log_2(carry\_min) \rceil}$; ▷ decide to replace or not
  $R \leftarrow random(0, new\_val)$;
  if $R = 0$ **then**
    clone and recirculate packet with new_val, min_stage;
if *packet is cloned* **then**
  $i \leftarrow min\_stage$; ▷ for cloned packets, update key and value
  $l_i \leftarrow h_i(iKey)$;
  $Key_i[l_i] \leftarrow iKey$;
  $Val_i[l_i] \leftarrow new\_val$;
  Drop the cloned copy;

```
#define MAX_STAGE 4
#define MAX_SIZE 65536

void precision(){
    reg_32 key[MAX_STAGE][MAX_SIZE];
    reg_32 val[MAX_STAGE][MAX_SIZE];
    matched = false;
    carry_min = 0x7fffffff;
    min_stage = -1;
    //for normal packets, find the minimum bucket
    if (p.resubmit == 0){
        for (i=0; i<MAX_STAGE; i++){
            l = hash(p.key);
            if (key[i][l] == p.key){
                matched = true;
                val[i][l] = val[i][l] + 1;
            }else if (val[i][l] < carry_min){
                carry_min = val[i][l];
                min_stage = i;}
        }
        if (!matched){ //decide replace or not
            new_val = roof(carry_min);
            R = random(0, new_val);
            if (R == 0) {resubmit(new_val, min_stage);}
        }
    }else{ //for cloned packets, update key and value
        l = hash(p.key);
        key[min_stage][l] = p.key;
        val[min_stage][l] = new_val;
        drop();
    }
}
```

(**a**)

```
void forward(){
    for (i=0; i<5; i++){
        if (p.ipv4.dstAddr & 0xff00 == i){
            metadata.egress_port = i;
        }
    }
}

void count(){
    reg_32 rx[MAX_SIZE];
    reg_32 tx[MAX_SIZE];
    rx[metadata.ingress_port] +=
        rx[metadata.ingress_port] + 1;
    tx[metadata.egress_port] =
        tx[metadata.egress_portt] + 1;
}

class MySwitch(){
    void parser(){}
    void ingress(){
        precision();
        forward();
        count();
    }
    void egress(){}
    void deparser(){}
}
```

(**b**)

Figure 3. E-Domino programs: (**a**) the PRECISION algorithm module; (**b**) other transactions.

3.2. E-Domino Grammar

Table 1 presents the grammar of E-Domino. An E-Domino program p consists of header declarations *hdr*, parser definitions *parser*, ingress pipeline i and egress pipeline e. We follow the P4 header and parser definitions and omit them for simplicity.

Table 1. Grammar of E-Domino language.

Derivations	Literals
$p \in$ programs ::= hdr; parser; i; e;	
$i \in$ ingress ::= ingress(v) $\{m\}$	$l \in literals$,
$e \in$ egress ::= egress(v) $\{m\}$	$v \in variables$,
$m \in$ modules ::= name(v) $\{s\}$ | $m; m$	$un\_op \in unary\ ops$,
$s \in$ statements ::= $s; s$ | $e = e$ | if(e) $\{s\}$ [;else $\{s\}$] | for $(e;\ e;\ e)$ $\{s\}$	$bin\_op \in binary\ ops$,
$e \in$ expressions ::= l | v | name.f | reg_name[e] |	$rel\_op \in relational\ ops$
$e\ bin\_op\ e$ | $un\_op\ e$ | $e\ rel\_op\ e$ |	
$f(e[, e, ...])$ | $valid$(name.f)	

Modules. The ingress and egress pipeline consists of modules m, which implement the data plane functions. We propose the notion, modular programming, that writes transactions in separate modules and organizes the modules into integrated data plane functions. The modules are reusable, e.g., we can write an L2 forwarding module and reuse it for any programs. Modular programming reduces the workload of repeatedly writing the same P4 programs.

P4 programs also contain several modules, i.e., parser, ingress, egress, and deparser, etc. However, P4 is coarse-grained in modularization. The ingress and egress pipeline includes several match-action tables that execute different tasks. Modular programming is fine-grained in that each module carries out a specific function. Thus, we can reuse the module to enable fast programming.

Statements. E-Domino has three types of statements, assignment statements, conditional statements, and loop statements. The assignment statements modify the header field, metadata field, or register values. The conditional statements make up control flows in the ingress or egress pipeline or register update conditions in register operation actions. The loop statements are the key to enabling fast programs for different stages of similar actions, e.g., to find the minimum value across all stages in the PRECISION algorithm. While in Domino, we have to write codes of the d stages repeatedly.

Expressions. E-Domino supports unary operations, binary operations, and relational operations in expressions. The switch does not directly support these operations. Thus, we transform them into hardware primitives.

We also need complex mathematical operations that are not provided by the data plane. We implement these complex mathematical operations in external functions. External does not mean that the functions rely on external processing and work through a slow path of a switch. Instead, external functions rely on match-action tables and work with other transactions in the pipeline of a programmable switch. For example, the ceil function (line 11 of Algorithm 1) approximates an integer to 2^i $(i \in \mathbb{N}^+)$ due to the restriction to the upper bound parameter of *random* primitives. The ceil function is organized as a match-action table with n entries. The largest possible value is 2^n. The ith$(i \in \mathbb{N}^+, i < n)$ entry is responsible for matching the number $carry\_min \in [2^{i-1}+1, 2^i]$ and outputs the approximate value 2^i.

Assuming $n = 8$ and the largest possible value is $2^8 = 256$, we will need eight entries as Table 2 shows. The entries are numbered from 1 to 8, where the priority decreases as the number is increasing. We use a ternary match for each entry, which is in the form of $value$ & $mask$. The mask bits which are zero indicate a wildcard, where the corresponding bits in value can be either 1 or 0; while the non-zero

mask bits indicate an exact match, where the corresponding bits in the value must be 0. Assuming $carry\_min = 6$, the third entry matches with the highest priority. Entries 4–8 also match, but they have lower priorities than entry 3. Thus, the input number 6 is approximated to 8. The ceil function causes one stage latency due to the match-action table.

Table 2. The match-action table to implement the ceil external function.

Entry No.	Match Field	Action Parameter
1	(0x00 & 0xfe)	2
2	(0x00 & 0xfc)	4
3	(0x00 & 0xf8)	8
4	(0x00 & 0xf0)	16
5	(0x00 & 0xe0)	32
6	(0x00 & 0xc0)	64
7	(0x00 & 0x80)	128
8	(0x00 & 0x00)	256

In all, the E-Domino is a modular top-down design language. Programmers write the high-level language in modules regardless of the underlying hardware restrictions, and the compiler is responsible for outputting the P4 programs for the programmable switch.

4. P4HLPc Design

This section presents P4HLPc design. The compiler handles loops, control flows, stateful processing, stateless processing, and external functions.

4.1. Unroll the Loops

The first step is to unroll all loops in the program. Loops have different varieties: One is for operations in different stages (e.g., the PRECISION algorithm module), and another is for different conditions (e.g., the forwarding module). It is easy to identify the two kinds of loops. The loops that operate register arrays belong to the former variety and must be arranged in sequential stages to obey restrictions. The loops that manipulate stateless fields belong to the latter variety, and they can be arranged in one single-stage match-action table. P4HLPc tags the type of loops with either stateful or stateless.

Note that the times of iterations must be fixed at compiling time for the former variety because the hardware pipeline must be fixed. We cannot modify the pipeline stages unless we halt the pipeline and apply a new P4 program. While for the latter variety, we can assume an upper limit of the times of iterations and set the match-action table size to that limit. We can add, delete and modify table entries which decide stateless processing at runtime.

4.2. Control Flows

After the unrolling, P4HLPc must identify all the possible paths and generate corresponding P4 control flows. The branch happens at each conditional statement. We use eFDDs (extended forwarding decision diagrams) for the intermediate representation of the E-Domino program.

The eFDDs are composed of nodes and directed edges. Each node denotes a set of expressions, and each edge indicates the dependency of two nodes. An eFDD acts like a binary decision diagram: Each intermediate node is a conditional expression, and each leaf node denotes actions. Each intermediate node has two successors: The left for the true condition, and the right for the false condition.

The conditional statement $if(t)\ \{e_1\}\ else\ \{e_2\}$ results in dependencies from t to e_1 and e_2. P4HLPc connects eFDDs by these dependencies.

4.3. Stateful Processing

SNAP (Stateful Network-Wide Abstractions for Packet Processing) [3] proposed SALU atoms for abstraction for stateful processing. These atoms include Read/Write, Read-Add-Write (RAW), Predicated Read-Add-Write (PRAW), IfElse Read-Add-Write (IfElseRAW), and Nested IfElse Read-Add-Write (Nested). We extend the atoms with Paired Updates (PU), and Paired Output (PO) (see Table 3).

Table 3. Stateful processing atoms. Read/Write, Read-Add-Write (RAW), Predicated Read-Add-Write (PRAW), IfElse Read-Add-Write (IfElseRAW), and Nested IfElse Read-Add-Write (Nested),Paired Updates (PU), and Paired Output (PO).

Type	Description	Resources
Read/Write	Read a single register to field/ write a field to a single register.	ALU_lo_1/ Update_lo_1_value
RAW	Add field to a single register.	ALU_lo_1, Update_lo_1_value
PRAW	Predicated RAW	ALU_lo_1, Condition_lo Update_lo_1, Update_lo_1_value
IfElseRAW	Two PRAWs, each conditional RAW according to predication is true or false.	Low Slice, Update_lo_value
Nested	Two PRAWs, each conditional RAW according to combination of two predications.	Condition_hi Low Slice, Update_lo_value
PU	Two Nested, and updates two registers.	entire black box
PO	Same as PU, but write two fields	entire black box, Low Slice of another black box

SALU supports conditional statements in the form of Equation (1).

$$\pm field \pm register + constant < rel\_op > 0. \quad (1)$$

The *field* denotes a packet header or metadata field, the *register* denotes the *register_lo* or *register_hi*, and the *constant* denotes a immediate value. The operations must be one of the atoms in Table 3. Besides altering register values, we can output either *register_lo* or *register_hi* to a header field. The SALU can operate a pair of registers but only can output to one of them. P4HLPc transforms stateful processing to atoms, by filing in predefined templates. Each atom forms an eFDD. If one of the conditions, operations, or output transforming fails, the operation is not supported by SALU, and P4HLPc reports an error.

The register operations also cause dependencies. Assuming a and b are two register slots, b depends on a if the program reads a and writes the value to b. We organize the eFDDs into strongly connected components (SCCs), according to the dependencies between eFDDs. We add an edge from one SCC to another, where the second one depends on the first one. Figure 4 presents the eFDDs and SCCs of the PRECISION module.

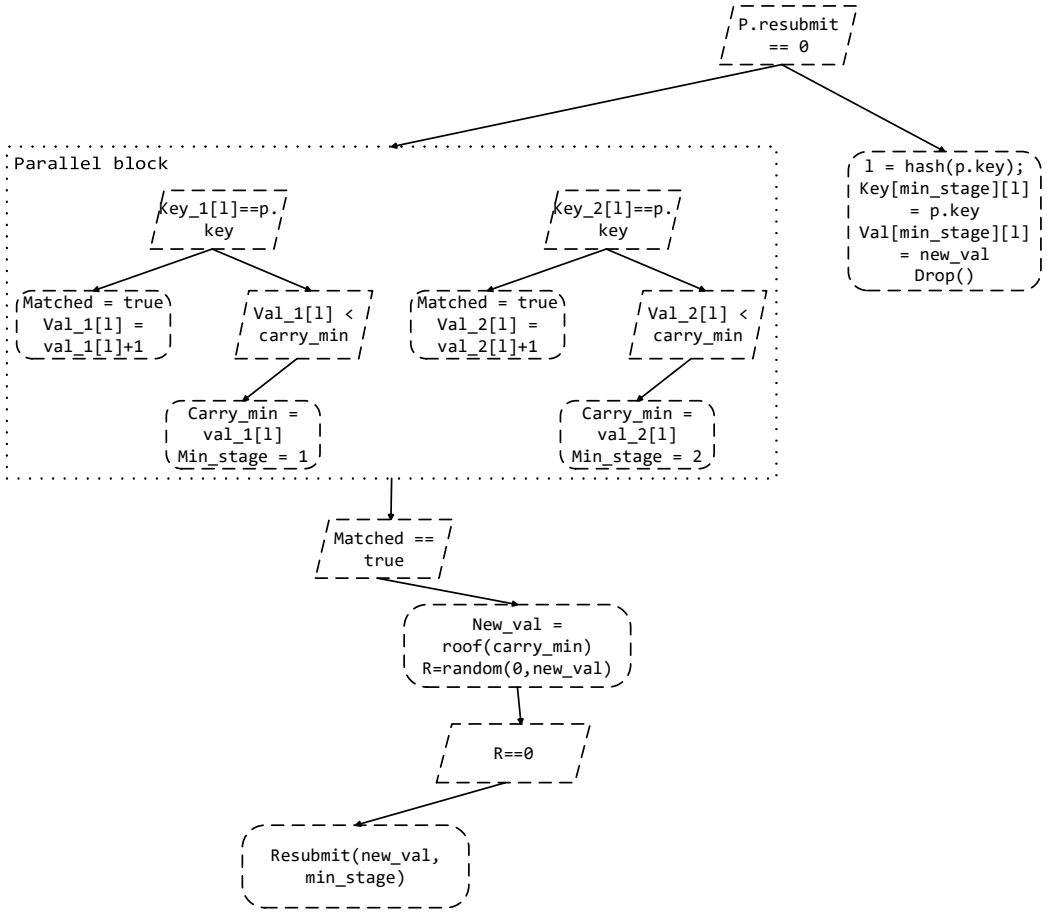

Figure 4. The strongly connected components (SCCs) of the PRECISION (Probabilistic RECirculation admisSION) module.

4.4. Stateless Processing

E-Domino uses mathematical symbols to represent stateless processing. The symbols exist in both statements and conditional expressions. For those in statements, we need to transform the mathematical symbols into the corresponding low-level primitives. For those in conditional expressions, we must further connect them by boolean operations. Due to this, we only elaborate the latter one.

Table 4 lists the detailed design. For example, we adopt the ternary match to implement the boolean operation "or". The ternary match only cares about the bits whose corresponding mask is 1. For example, the expression if (ethernet.dst==0xC0A80164 or ethernet.src==0xC0A80165) is true if the key $(ethernet.dst, ethernet.src)$ matches entry (0xC0A80164, *) or (*, 0xC0A80165). The "*" denotes a wildcard character. The table methods have no delay but consume some match-action tables. The primitive methods have a delay of 1–2 stages but consume no match-action tables. We use primitives instead of match-action tables to implement relational operations in order to trade-off resource consumption and latency.

Table 4. Stateless tests design: Transform a stateless branch into match-action tables or primitives.

Method	Type	E-Domino Program	Tofino Program	Delay
Tables	bool_op	if(ethernet.dst == 0xC0A80164 and ethernet.src== 0xC0A80165)	if(ethernet.dst == 0xC0A80164) if(ethernet.src== 0xC0A80165)	0
		if(ethernet.dst == 0xC0A80164 or ethernet.src== 0xC0A80165)	reads{ethernet.dst: ternary; ethernet.src: ternary;}	0
	rel_op	if (ipv4.ttl < metadata.ttl)	reads{ ipv4.ttl: exact; metadata.ttl: exact;}	0
		if (ipv4.ttl >= metadata.ttl)		0
		if (ipv4.ttl == metadata.ttl)		0
primi-tives	rel_op	if ($h_1 < h_2$)	subtract(tmp, h_1, h_2); if(tmp & 0x8000 !=0)	1
		if ($h_1 >= h_2$)	substrct(tmp, h_1, h_2); if(tmp & 0x8000 ==0)	1
		if ($h_1 == h_2$)	xor(tmp, h_1, h_2); if(tmp==0)	1
	bin_op	if ($h_1 - h_2$ rel_op value)	subtract($tmp, h_1, value$); if(tmp rel_op h_2);	2
		if (h_1^h_2 rel_op value)	bitxor($tmp, h_1, value$); if (tmp rel_op h_2)	2
		if ($h \ll value_1$ rel_op $value_2$)	shift_left($tmp, h, value_1$); if (tmp rel_op $value_2$)	2
	un_op	if ($-h_2$ rel_op value)	subtract($tmp, 0, h_2$); if(tmp rel_op value);	2
		if (^h_2 rel_op value)	bitxor($tmp, 0xFFFFFFFF, h_2$); if (tmp rel_op value)	2

We propose a branch transform algorithm (BTA, Algorithm 2) that transforms stateless conditional expressions. BTA consists of preprocessing, transforming, and reassembling.

Algorithm 2: Branch transform algorithm.

Input: E-Domino branch expression
Output: P4 branch program

```
set C = {C_i | C_i is conditional expression in control block};
set DNF = ∅;
foreach C_i ∈ C do
    replace and with ∧;
    replace or with ∨;
    DNF_i=DNF_ALGORITHM(C_i);
    foreach CC_j ∈ DNF_i do
        foreach Boolean_k ∈ CC_j do
            transform Boolean_k to RPN;
            transform RPN to primitives according to Table 4;
        form a nested If;
    form a match-action table;
```

Preprocessing. Extract all conditional expressions in the control blocks. For each conditional expression, replace "and" with a conjunction, replace "or" with a disjunction, and transform it into a disjunctive normal form (DNF) [31] by calling function DNF_ALGORITHM(C_i). We use DNF instead of CNF (conjunctive normal form) [32] for simplicity. (see Section 5.1) Split DNF by disjunction symbols into conjunctive clauses (CCs). For each CC in DNF, split it by conjunction, i.e., "and" in the conditional expression, into boolean expressions.

Transform. For each boolean expression, transform it into reverse Polish notation (RPN) [33]. Then transform the RPNs into primitives according to the rules in Table 4.

Reassemble. Reassemble boolean expressions according to conjunction and disjunction. Reassemble CCs first: For primitives in one CC, reassemble their results into a nested if. Then reassemble DNF: For CCs, reassemble their results into a match-action table.

4.5. External Functions

For the external functions, we adopt joint compiling to generate the match-action tables along with thrift codes to install table entries. So far we support three kinds of external functions: The math functions, such as the ceil function, that look up results in tables, the hash functions that calculate user-defined hash values, and the digest functions which generate a digest of packets for INT (in-band network telemetry).

4.6. Outputs

The programs may not be feasible to real machines, due to the atom and primitive restrictions. If any of the above steps fail, the compiling terminates, and P4HLPc throws compiling errors. Otherwise, P4HLPc generates the P4 programs from the E-Domino codes. Algorithm 3 summarizes the compiling steps.

Algorithm 3: Compiling algorithm.

Input: E-Domino programs
Output: P4 prorgams

```
unroll all loops (Section 4.1);
generate eFDDs to present control flows (Section 4.2);
generate SCCs according to dependencies;
foreach stf ∈ stateful processing do
    transform stf to atom (Section 4.3) ;          ▷ generate atoms in Table 3
foreach stl ∈ stateless processing do
    transform stl to primitives (Section 4.4) ;    ▷ generate primitives in Table 4
generate external functions (Section 4.5) ;                    ▷ Joint Compliling
```

5. Results

We implement algorithms in E-Domino, then compile the E-Domino programs in P4 code using P4HLPc. We evaluate the programs automatically generated by P4HLPc and compare the E-Domino code with the P4 code. We also evaluate P4HLPc under different conditions.

5.1. E-Domino

Table 5 presents the E-Domino experiment results. The E-Domino can implement various data plane algorithms. Except for the space saving algorithm, which is not feasible on the state-of-art hardware, we implement the other algorithms (Bloom filter [12], Flowlet [13], HashPipe [15], RAP [16], PRECISION [10], Elastic sketch [17] and SketchLearn [18]) in E-Domino and then transform them into P4 programs using P4HLPc.

Table 5. E-Domino implementation and comparison with P4.

Algorithm	Application	Atom	# of Stages, # of Atoms	E-Domino LOC	P4 LOC	Time (s)
Bloom filter	Distinct flows	Read/Write	1, 3	14	116	0.21
Flowlet	Flowlet Domain name system tunnel detection Time to live change tracking Phishing/spam detection	PU	1, 1	15	102	0.18
Space Saving	Heavy hitter detection Per-flow Frequency Cardinality estimation	-	not feasible			
HashPipe		PO	4, 8	25	371	0.33
RAP		IfElseRAW	4, 8	60	365	0.39
PRECISION		IfElseRAW	4, 8	60	365	0.39
Elastic sketch	Per-flow frequency Heavy hitter detection Heavy changers Cardinality estimation Frequency distribution and entropy	Nested	4, 4	44	241	0.42
SketchLearn		PRAW	8, 32	16	767	0.72

The P4 programs use different atoms. The flowlet and hashpipe use PU and PO respectively, to operate the key-value map. The new atoms (PU and PO) extend register operations of Domino, thus enabling more algorithms than Domino. Also, the P4 programs consume a various number of SALUs and stages. They use resources as defined in E-Domino. No resources are wasted.

The E-Domino is expressive for data plane functions. P4 uses 5.5× to 47.9× more code than E-Domino. We only consider the lines of codes of the core module. Most importantly, E-Domino is highlighting for writing programs which repeatedly carry out the same routine in different stages, e.g., the SketchLearn algorithm in E-domino is only 16 lines, while the number is 767 in P4, owing to the repeated register declarations and writing operations in P4.

5.2. P4HLPc

P4HLPc goes through the steps in Algorithm 3 for a new program, such as the cold start phases in Table 6. The cold start only happens once for each program, at the very beginning of deploying it. The cold start is accomplished in 1–3 s, and there is no need to halt the switch because all the compiling work is done offline in other machines. P4HLPc finishes the compiling steps for all algorithms in less than 0.72 s in Table 5. Once finished compiling, we can incrementally add new modules to the prior programs with no halt. Policy change needs steps 1–5 in Table 6 because it changes control flows and dependencies for the program. The external functions and routing stay unchanged. Thus we can reuse them. When the network topology changes, we only need to reconfigure the forward table entries, which is accomplished in milliseconds. P4HLPc induces no halting to the data plane when compiling the programs.

If any of the compiling steps fail, P4HLPc reports error(s), which is faster than the P4 compiler which transforms the P4 codes to RMT configurations. P4HLP enables programmers to reuse some standard modules from prior programs. P4HLP also supports fast reproducing prior works which are implemented in P4 simulators or other platforms.

It is reasonable to use primitives instead of match-action tables to implement relational operations in Section 4.4. Although match-action tables are faster than primitives, they need vast entries to match the keys, e.g., for expression $a \geq b$ ($a, b \in \mathbb{N}^+$), assuming the most significant value of a is A, we will need $1 + 2 + ... + (A + 1) = \frac{(A+1)(A+2)}{2}$ entries for match-action tables. Using primitives is a better trade off because the primitives consume fewer resources compared with table entries, while the latency is not remarkable for the overall 12 stages of the pipeline.

DNF is more space-efficient than CNF. Each disjunction clause (DC) will result in a match-action table, and each boolean expression in a DC needs an entry in the table. By contrast, each CC only needs

a few gates to implement nested ifs. Consider the expression $if((a == 1\ and\ b < 10)\ or\ c >= 0)$, CNF costs one gate and two match-action tables with six entries, doubling that of DNF, however, with no decrease in latency.

Table 6. P4HLPc (Programming Protocol-Independent Packet Processors High-Level Programming compiler) compiling phases for different scenarios.

No.	Phase		Cold Start	Policy Change	Topo Change
1	unroll loops		✓	✓	×
2	generate eFDDs		✓	✓	×
3	stateful processing		✓	✓	×
4	stateless processing		✓	✓	×
5	generate SCC		✓	✓	×
6	generate exernal functions	tables	✓	×	×
		entries	✓	✓	×
7	routing	tables	✓	×	×
		entries	✓	×	✓

5.3. Testbed Results

Table 7 shows the additional resource consumption of different algorithms on Tofino, normalized by the usage of the baseline program switch.p4, a P4 program that implements most networking features (L2/L3 forwarding, VLAN, QoS, etc.) for a typical switch. Bloom filter and Flowlet are lightweight algorithms that consume, on average 2.83% of all resources. HashPipe, RAP, and PRECISION are storage-intensive algorithms that consume 34.68%–46.67% SRAM (Static Random Access Memory) to store flow keys and values. They use 16.67%, i.e., eight out of all 48 SALUs to update the SRAM. Moreover, RAP and PRECISION consume much more TCAM (Ternary Content Addressable Memory) than HashPipe to implement the lookup table for the ceil function. Joint compiling automatically generates the table declarations and entries for the ceil function. We also test the controller python codes to dump the entries. The controller communicates with the data plane through thrift [29]. Elastic sketch and SketchLearn consume 75% and 66.67% SALUs, respectively. A SALU handles a register with one atom. We set the key length $k = 32$ bit. SketchLearn consumes one SALU for each bit of the key, i.e., 32 out of all 48 SALUs.

Table 7. Additional resource consumption, normalized by the usage of the baseline switch.P4. RAP (Randomized Admission Policy), SRAM (Static Random Access Memory), TCAM (Ternary Content Addressable Memory), and VLIW (Very long instruction word).

Resource	Base Line	Bloom Filter	Flowlet	HashPipe	RAP	PRECISION	Elestic Sketch	SketchLearn
Match Crossbar	50.13%	2.84%	1.47%	17.33%	12.60%	11.02%	5.87%	7.28%
Hash Bits	32.35%	14.21%	2.38%	37.25%	29.76%	27.30%	2.32%	7.14%
SRAM	29.79%	5.60%	2.26%	46.67%	35.41%	34.68%	12.51%	50.33%
TCAM	28.47%	0%	0%	2.84%	38.62%	37.83%	0%	1.31%
VLIW Actions	34.64%	3.48%	1.67%	13.96%	9.56%	7.59%	5.48%	11.31%
Stateful ALUs	15.63%	0.06%	0.04%	16.67%	16.67%	16.67%	75.00%	66.67%

We stress-test the throughput of the generated P4 programs on Tofino. Figure 5 shows the normalized throughput to the line-rate speed without measurement. The processing speed is preserved, and the variance is minimal. The external functions and branch transformation does not influence the throughput. The high performance comes from the character of pipeline stages that process multiple packets concurrently.

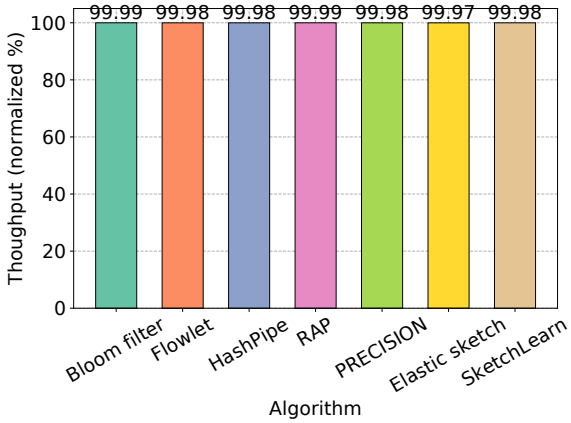

Figure 5. Throughput of P4 programs on Tofino.

6. Related Works

Many DSLs explore how to program the data plane. Click [34] is a configurable software switch architecture. A Click router consists of several packet processing modules, e.g., packet classification, queueing, and controlling modules. Click programs these modules in C++. Software switches are 10×–100× slower than hardware switches. As network traffic volume grows, the software can not process packets at line-rate. We must implement the data plane functions in hardware.

Jose et al. [35] designed a P4 compiler, in order to map P4 logical match-action tables to physical switching chips, while meeting the data and control dependence in the program. They adopt integer linear programming (ILP) and greedy algorithms to optimize latency, resource occupancy, and power consumption. They compile benchmarks to two commodity switches, RMT and Flexpipe. The compiler use abstractions to hide hardware details while capturing the essence required for mapping. The compiler focuses on stateless processing while P4HLPc also supports stateful processing.

Arashloo et al. [3] concentrated on sophisticated stateful processing in the data-plane and suggested SNAP, a more straightforward "centralized" stateful programming model. SNAP views the distributed stateful elements as centralized, relieving programmers of placing and optimizing access to these stateful arrays.

Instead of targeting specific targets, NetAsm [21] uses intermediate representation (IR) to program various devices, such as FPGAs (Field Programmable Gate Arrays) and programmable switches. These IRs remove target restrictions from consideration, much like P4. P4Visor [36] provides new primitives to replace the state-of-art data-plane primitives, compiler optimizations, and program analysis-based algorithms, which reduce the resource overheads. P4Visor supports rapid testing and deployment life-cycles.

Sivaraman et al. [4] proposed a new DSL Domino to program data planes in a C-like manner. They also devised a hardware machine model, Banzai, for programmable line-rate switches. The Domino compiler generates Banzai primitives from the C-like Domino programs. Domino concentrates on stateful processing, under different types of register operations in the Banzai model, neglecting the stateless processing part in control blocks. What is more, the commodity switches are more complicated in stateful processing than in Banzai, e.g., Tofino has a branch on condition of the register values, while Banzai only depends on header or metadata fields. Also, Domino does not support loops, which is not friendly to algorithms that repeatedly operate in the same routine. The proposed E-Domino language and P4HLPc support stateless processing and loops.

7. Discussion

P4HLP provides the ability to migrate software algorithms to hardware devices, reproduce prior works, and develop novel algorithms efficiently. However, some work must be done before we apply P4HLP to line-rate switches.

1. P4HLP defines programs in the high-level language E-Domino, which hides the hardware details of programmable switches. This abstraction relieves programmers from concerns about hardware restrictions but may contribute to infeasible programs on real machines or programs that are low-efficiency, despite being feasible. For example, P4HLP supports the hash parallel algorithm which needs to recirculate every packet in order to update the smallest register slot. Some algorithms may require too many specific hardware resources in one stage, making the other resources in the stage unavailable for other modules. This question may account for external functions that consume match resources. P4HLP warns programmer when algorithm halves the throughput or consumes too many resources.
2. P4HLP may fail to generate some programs that originally violate the RMT restrictions thus cannot be mapped to RMT architecture through optimizing techniques. We may find these algorithms and explore possible manual optimizing methods.
3. P4HLPc does not optimize P4 programs. Currently we leave optimization to a P4 compiler. Future works can be done in either P4HLPc or a P4 compiler to optimize the pipeline implementation. We may combine P4HLPc with a P4 compiler to generate the optimized hardware configuration directly from E-Domino language, compared with the current two step workflow: First, compile the E-Domino program to the P4 program, and then compile the P4 to the target.
4. P4HLP may support more emerging ASIC such as XPA that varies tiny in specific primitives, with minor modifications to E-Domino grammar.

8. Materials and Methods

All materials in this paper are available at Github [37]. We developed and tested the P4HLPc on Ubuntu 16.04. We tested all P4 programs on a Wedge 100BF-32X switch, which has a Tofino 3.2 Tbps chip. We connected the switch with two end hosts, using 40 Gbps QSFP links. Each of the end hosts was a DELL PowerEdge R820 Server, with an Intel XL710-QDA2 40 Gbps network interface card, two Intel Xeon E5-4603 CPUs and 256 GB memory. We installed MoonGen [38], a scriptable high-speed packet generator built on libmoon [39] and packet processing library DPDK (Data Plane Development Kit) [40], to send and receive packets on the hosts, achieving a stable speed of 38.04 Gbps.

We tested the cold start for P4HLPc using four steps. First, we implemented the algorithm in E-Domino language. Second, the P4HLPc transformed the E-Domino programs to Tofino P4 programs. Third, the P4 compiler compiled the P4 programs to Tofino configurations. Last, we configured the Tofino switch and dump table entries. We collected the resource utilization summary to analyze the quality of P4 programs. We also changed the policy and typology to test scalability and flexibility.

9. Conclusions

This paper proposes E-Domino language and P4HLPc, to support unified high-level programming for programmable switches. The P4HLP framework aims to hide restrictions on processing resources, storage resources, and latency.

E-Domino is a modular top-down design language, regardless of the underlying hardware restrictions. There are four aspects of the high-level language E-Domino that we highlight: (i) E-Domino extends Domino with loops, expediting the programming efficiency. (ii) E-Domino proposes arithmetic operations in the conditional statement of control flow. (iii) E-Domino extends stateful atoms with a Paired Update and Paired Output, which are widely used in key-value mapping programs. (iv) E-Domino supports external functions to extend the Tofino ASIC functions.

The P4HLPc transforms E-Domino programs to P4 programs with four fundamental techniques: loop unrolling, TBA, atom template, and joint compiling. (i) P4HLPc recognizes loops for stages

and table entries and unrolls them. (ii) The TBA handles conditional statements in control flows by generating a combination of primitives and match-action tables. (iii) P4HLPc generates SALU codes from an atom template. (iv) The joint compiling technique generates a match-table declaration along with entries to implement external functions. P4HLPc not only automatically transforms E-Domino programs to P4 implementations, but also fulfills switches with more functions, by TBA and joint compiling. This paper also proposes the modular programming thought, which organizes E-Domino transactions in different modules. Programmers can reuse the modules from prior works.

Results show, compared with P4, E-Domino is precise and expressive, thus is suitable for programming data planes of programmable switches. P4HLPc is robust to policy change and topology change, and thus is suitable for compiling E-Domino programs. Modular programming is fine-grained and enables fast reconfiguration of commodity switches. The generated P4 programs maintain line-rate speed on Tofino switches.

In all, our works make a step towards unified high-level programming for commodity programmable switches.

Author Contributions: Conceptualization, Z.H. and Y.S.; methodology, Z.H.; software, Z.H.; validation, Z.H. and Y.S.; formal analysis, Z.H. and Y.S.; investigation, M.W.; resources, Z.H.; data curation, Z.H.; writing–original draft preparation, Z.H.; writing–review and editing, Y.S.; visualization, Z.H.; supervision, M.W. and C.Z.; project administration, M.W. and C.Z.

Funding: Supported by National Key Research and Development program under No.2016YFB1000400, National Nature Science Foundation of China under NSFC No. 61872377, 61802417, 61802420 and 61502509.

Acknowledgments: First of all, I am very grateful to my mentor, Chunyuan Zhang, for his careful guidance of my graduation thesis in the past three months, which greatly improved my understanding of academic writing and taught me a lot of specific research skills; knowledge is a vast ocean, I am only one of the flat boats. I am grateful to the teachers who have given me selfless help in my two years of development.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

P4HLP	Programming Protocol-Independent Packet Processors High-Level Programming
SDN	software-defined network
P4	Programming Protocol-independent Packet Processors
P4HLP	P4 high-level programming
P4HLPc	P4 high-level programming compiler
DSL	domain-specific language
HH	heavy hitter
RAP	Randomized Admission Policy
PRECISION	Probabilistic RECirculation admisSION
APIs	Application Programming Interfaces
P4HLPc	Programming Protocol-Independent Packet Processors High-Level Programming compiler
Thrift-RPC	Thrift Remote Procedure Call
ALU	Arithmetic Logic Units
E-Domino	Enhanced version of Domino
BTA	branch transform algorithm
RMT	reconfigurable match-action tables
SALU	stateful arithmetic logic unit
ASIC	application specific integrated circuit
XPA	Cavium XPliant Packet Architecture
PISA	protocol-independent switch rchitecture
eFDD	extended forwarding decision diagram
SNAP	Stateful Network-Wide Abstractions for Packet Processing

RAW	Read-Add-Write
PRAW	Predicated Read-Add-Write
IfElseRAW	IfElse Read-Add-Write
Nested	Nested IfElse Read-Add-Write
PU	Paired Updates
PO	Paired Output
SCC	strongly connected component
DNF	disjunctive normal form
CC	conjunctive clauses
RPN	reverse Polish notation
CNF	conjunctive normal form
DC	disjunction clause
SRAM	Static Random Access Memory
TCAM	Ternary Content Addressable Memory
VLIW	Very long instruction word
FPGA	Field Programmable Gate Arrays
ILP	integer linear program
IR	intermediate representation
INT	in-band network telemetry
QSFP	quad small form-factor pluggable
DPDK	data plane development kit

References

1. ONF. *Software-Defined Networking (SDN) Definition*; ONF: Menlo Park, CA, USA, 2019.
2. P4 Language Consortium. 2019. Available online: https://p4.org/ (accessed on 15 June 2019).
3. Arashloo, M.T.; Koral, Y.; Greenberg, M.; Rexford, J.; Walker, D. SNAP: Stateful network-wide abstractions for packet processing. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, 22–26 August 2016; pp. 29–43.
4. Sivaraman, A.; Cheung, A.; Budiu, M.; Kim, C.; Alizadeh, M.; Balakrishnan, H.; Varghese, G.; McKeown, N.; Licking, S. Packet transactions: High-level programming for line-rate switches. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, 22–26 August 2016; pp. 15–28.
5. Barefoot. *Barefoot Tofino: World's Fastest P4-Programmable Ethernet Switch ASICs*; Barefoot: Santa Clara, CA, USA, 2019.
6. Intel. *Intel FlexPipe*; Intel: Santa Clara, CA, USA, 2019.
7. Cavium. *Cavium and XPliant Introduce a Fully Programmable Switch Silicon Family Scaling to 3.2 Terabits per Second*; Cavium: San Jose, CA, USA, 2019.
8. Cisco. 2019. Available online: https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/datasheet-c78-740836.html (accessed on 20 July 2019).
9. Bosshart, P.; Daly, D.; Gibb, G.; Izzard, M.; McKeown, N.; Rexford, J.; Schlesinger, C.; Talayco, D.; Vahdat, A.; Varghese, G.; et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 87–95. [CrossRef]
10. Ben-Basat, R.; Chen, X.; Einziger, G.; Rottenstreich, O. Efficient measurement on programmable switches using probabilistic recirculation. In Proceedings of the 2018 IEEE 26th International Conference on Network Protocols (ICNP), Cambridge, UK, 24–27 September 2018; pp. 313–323.
11. Sivaraman, A.; Subramanian, S.; Alizadeh, M.; Chole, S.; Chuang, S.T.; Agrawal, A.; Balakrishnan, H.; Edsall, T.; Katti, S.; McKeown, N. Programmable packet scheduling at line rate. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, 22–26 August 2016; pp. 44–57.
12. Chazelle, B.; Kilian, J.; Rubinfeld, R.; Tal, A. The Bloomier filter: An efficient data structure for static support lookup tables. In Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, New Orleans, LA, USA, 11–14 January 2004; pp. 30–39.
13. Sinha, S.; Kandula, S.; Katabi, D. Harnessing TCP's burstiness with flowlet switching. In Proceedings of the 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III), San Diego, CA, USA, 15–16 November 2004.

14. Metwally, A.; Agrawal, D.; El Abbadi, A. Efficient computation of frequent and top-k elements in data streams. In Proceedings of the International Conference on Database Theory, Edinburgh, UK, 5–7 January 2005; pp. 398–412.
15. Sivaraman, V.; Narayana, S.; Rottenstreich, O.; Muthukrishnan, S.; Rexford, J. Smoking out the heavy-hitter flows with hashpipe. *arXiv* **2016**, arXiv:1611.04825.
16. Basat, R.B.; Einziger, G.; Friedman, R.; Kassner, Y. Randomized admission policy for efficient top-k and frequency estimation. In Proceedings of the IEEE INFOCOM 2017-IEEE Conference on Computer Communications, Atlanta, GA, USA, 1–4 May 2017; pp. 1–9.
17. Yang, T.; Jiang, J.; Liu, P.; Huang, Q.; Gong, J.; Zhou, Y.; Miao, R.; Li, X.; Uhlig, S. Elastic sketch: Adaptive and fast network-wide measurements. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, Budapest, Hungary, 20–25 August 2018; pp. 561–575.
18. Huang, Q.; Lee, P.P.; Bao, Y. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, Budapest, Hungary, 20–25 August 2018; pp. 576–590.
19. P4 Group. *Specification for the P4Runtime Control-Plane API*; GitHub: San Francisco, CA, USA, 2019.
20. Xilinx. *Vivado Design Hub—High-Level Synthesis (C Based)*; Xilinx: San Jose, CA, USA, 2019.
21. Shahbaz, M.; Feamster, N. The case for an intermediate representation for programmable data planes. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, Santa Clara, CA, USA, 17–18 June 2015; p. 3.
22. Bosshart, P.; Gibb, G.; Kim, H.S.; Varghese, G.; McKeown, N.; Izzard, M.; Mujica, F.; Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 99–110. [CrossRef]
23. Cisco. *Cisco Data Center Switches*; Cisco: San Jose, CA, USA, 2019.
24. HUAWEI. *HUAWEI Data Center Switches*; HUAWEI: Shenzhen, China, 2019.
25. P4 Language Consortium. *P4-14 Specification*; ONF: Menlo Park, CA, USA, 2019.
26. P4 Group. *P4 Prototype Compiler*; GitHub: San Francisco, CA, USA, 2019.
27. ONF. *P4 and P4Runtime Technical Introduction and Use Cases*; ONF: Menlo Park, CA, USA, 2019.
28. P4 Group. *Packet Test Framework*; GitHub: San Francisco, CA, USA, 2019.
29. Apache. *Apache Thrift*; Apache: Wakefield, MA, USA, 2019.
30. Hang, Z.; Shi, Y.; Wen, M.; Quan, W.; Zhang, C. SWAP: A sliding window algorithm for in-network packet measurement. In Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications, Xi'an, China, 8–10 March 2019; pp. 84–89.
31. Wikipedia DNF. Disjunctive Normal Form. 2019. Available online: https://en.wikipedia.org/wiki/Disjunctive_normal_form (accessed on 20 July 2019).
32. Wikipedia CNF. Conjunctive Normal Form. 2019. Available online: https://en.wikipedia.org/wiki/Conjunctive_normal_form (accessed on 20 July 2019).
33. Wikipedia RPN. Reverse Polish Notation. 2019. Available online: https://en.wikipedia.org/wiki/Reverse_Polish_notation (accessed on 20 July 2019).
34. Kohler, E.; Morris, R.; Chen, B.; Jannotti, J.; Kaashoek, M.F. The Click modular router. *ACM Trans. Comput. Syst. (TOCS)* **2000**, *18*, 263–297. [CrossRef]
35. Jose, L.; Yan, L.; Varghese, G.; McKeown, N. Compiling packet programs to reconfigurable switches. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA USA, 4–6 May 2015; pp. 103–115.
36. Zheng, P.; Benson, T.; Hu, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, Heraklion, Greece, 4–7 December 2018; pp. 98–111.
37. Hang, Z. *P4HLP Repository*; GitHub: San Francisco, CA, USA, 2019.
38. Emmerich, P.; Gallenmüller, S.; Raumer, D.; Wohlfart, F.; Carle, G. Moongen: A scriptable high-speed packet generator. In Proceedings of the 2015 Internet Measurement Conference, Tokyo, Japan, 28–30 October 2015; pp. 275–287.

39. Libmoon. *Libmoonm, a Library for Fast and Flexible Packet Processing with DPDK and LuaJIT*; GitHub: San Francisco, CA, USA, 2019.
40. DPDK Project. *DPDK, the Data Plane Development Kit that Consists of Libraries to Accelerate Packet Processing Workloads Running on a Wide Variety of CPU Architectures*; DPDK: Dover, DE, USA, 2019.