Fine-Grained Pipeline Parallelization for Network Function Programs

Seungbin Song
Yonsei University
Seoul, Republic of Korea
seungbin@yonsei.ac.kr

Heelim Choi
Yonsei University
Seoul, Republic of Korea
heelim@yonsei.ac.kr

Hanjun Kim
Yonsei University
Seoul, Republic of Korea
hanjun@yonsei.ac.kr

Abstract—**Network programming languages enable programmers to implement new network functions on various hardware and software stacks in the domain of Software Defined Networking (SDN). Although the languages extend the flexibility of network devices, existing compilers do not fully optimize the network programs due to their coarse-grained parallelization methods. The compilers consider each packet processing table that consists of match and action functions as a unit of tasks and parallelize the programs without decomposing match and action functions. This work proposes a new fine-grained pipeline parallelization compiler for network programming languages, named PSDN. First, the PSDN compiler decouples match and action functions from packet processing tables and analyzes dependencies among the matches and actions. While respecting the dependencies, the compiler efficiently schedules each match and action function into a pipeline with clock cycle estimation and fuses functions to reduce synchronization overheads. This work implements the PSDN compiler that translates a P4 network program to a Xilinx PX program, which is synthesizable to NetFPGA-SUME hardware. The proposed compiler reduces packet processing latency by 12.1% and utilization by 3.5% compared to previous work.**

Index Terms—**Pipeline Scheduling, Compilers, Networks**

I. Introduction

Recent network programming languages allow programmers to develop a network service on a network switch with multiple subdivided functional units. The OpenFlow specification [1] initially introduces a network switch architecture based on the multiple functional units, and the P4 programming language [2] introduces a programming model of the multiple functional units. The P4 language represents each functional unit as a *packet processing table* that consists of match and action functions. The *match* function compares packet header values with rules specified in a control plane. The *action* function modifies the packet header values or internal metadata according to the match comparison result. For example, a programmer can implement access control lists (ACLs), layer-2 (Ethernet) switching, layer-3 (IP) routing, and equal-cost multi-path (ECMP) routing as packet processing tables. The network service providers implement these various packet processing tables, combine them into a control flow, and execute the programmed network service on CPUs [3]–[5], FPGAs [6], or specialized packet processors [7]–[9].

Although the network programming languages like P4 [2] extend the flexibility of network devices, existing compilers [10]–[12] do not fully optimize the network programs. Since a P4 program contains matches and actions that read and update different packet header values, parts of the match and action functions can be executed in parallel. However, the compilers [10]–[12] consider each packet processing table as a unit of tasks and parallelize the programs without decomposing match and action functions. The compilers preserve data dependencies between tables in a coarse-grained manner and map them into a physical pipeline of packet processors. Therefore, the compilers lose fine-grained parallelism opportunities between match and action functions. To fully exploit the parallelism opportunities, a network programming compiler should disaggregate the packet processing tables into match and action functions and carefully allocate the functions instead of the tables into a pipeline.

This work proposes PSDN, a novel fine-grained pipeline parallelization compiler for a network function program written in the P4 language. The PSDN compiler consists of four parts: table decomposition, dependency analysis, pipeline scheduling, and code generation with function fusion optimization. First, the compiler decomposes a packet processing table into match and action functions and analyzes control and data dependencies among them. Then, the PSDN compiler estimates the processing latency of each function by reflecting execution behaviors and efficiently allocates the functions in a pipeline manner respecting the dependencies and the latency estimation. Here, to reduce the pipeline length, the compiler allocates independent functions into the same pipeline stage. The PSDN compiler finally generates a program written in PX language [13], which is synthesizable into FPGA-based network switches [6]. To simplify the network switch hardware generated from the PX program and reduce synchronization overheads among functions, the PSDN compiler additionally fuses concurrent functions in the same pipeline stage and consecutive functions in a pipeline.

This work evaluates the PSDN compiler using seven P4 programs [14] and synthesizes the programs to the NetFPGA-SUME board [6]. This work measures end-to-end packet processing latency by HDL simulation and resource usage of ALUs, registers, and memories by synthesizing the compiled program into the hardware. Compared to the previous work [12], the PSDN compiler reduces packet processing latency by 12.1% and resource utilization by 3.5%.

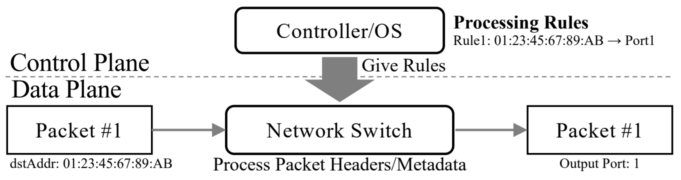


Fig. 1. Structure of Software Defined Networking

The contributions of this paper are:

- a new fine-grained pipeline parallelization compiler that transforms a P4 networking program into a PX program for Xilinx SDNet,
- a fine-grained dependency analysis and a pipeline scheduling scheme with clock cycle estimation,
- and a function fusion scheme that merges match and action functions to reduce latency and resource utilization.

II. Background & Motivation

A. Software-Defined Networking

Software-Defined Networking (SDN) [15] decouples control planes from data planes in network switches and enables controllable and programmable network switches. Fig. 1 describes a packet processing structure of SDN. In the control plane, network service providers describe rules about how a network switch processes packets depending on their packet headers using the built-in functions. In the data plane, a network switch receives packets and processes them according to rules defined in the control plane. OpenFlow [1] provides APIs between the data plane and the control plane, and ONOS [16] provides a control platform.

Thanks to the introduction of data plane languages and reconfigurable network switch architectures, the data plane becomes programmable. Instead of using the built-in functions, network service providers can program the switches to accept newly-defined protocols or execute multiple network functions in different protocols using data plane languages. One of the popular data plane languages is P4 [2]. For example, in-band Network Telemetry (INT) [17], load balancing [18], and in-network computation [19] are implemented in the P4 language.

B. Structure of Data Plane Language

P4 [2] is a domain-specific language that describes packet header processing. Although this work describes the P4 language, the structure of other languages like Huawei's Protocol-Oblivious Forwarding [20], [21] is similar to the structure of the P4 language. A data plane program consists of a parser, a table pipeline, and a deparser. The parser accepts packets and generates packet headers and metadata based on network protocols. With the parsed packet headers and metadata, the table pipeline modifies them based on the control plane rules. Finally, the deparser packages all the information and emits the modified packets.

Fig. 2 describes an example P4 pseudo program. This paper excludes header, metadata, parser, and deparser definitions to simplify the example program. A table pipeline contains action definitions, tables, and an apply function. An *action definition* is a function that modifies packet headers (`hdr`) or metadata (`meta`). Some actions require arguments as inputs (e.g., `set_output_port` and `set_broadcast`), whose actual values are given by the control plane. Besides, actions modify packet headers or metadata directly or by using external functions. A *table* definition contains *key*s that contain packet headers or metadata for match operations and *action*s that invoke action definitions when the keys are matched. An *apply* function is the main function that describes the control flow of tables. The apply function can contain conditional branches like `if` statements, but the P4 language does not support loops or iterations. Therefore, the control flow of the table pipeline is acyclic.

```
parser Parser(packet_in packet,
    out headers hdr, inout metadata meta) {...}

control TablePipeline(inout headers hdr,
    inout metadata meta) {
  action set_output_port(port_t port) {
    meta.dst_port = port;
  }
  table forward {
    key = { hdr.ethernet.dstAddr: exact; }
    actions = {
      set_output_port;
      NoAction;
    }
    default_action = NoAction;
  }
  action set_broadcast(port_t port) {
    meta.dst_port = port;
  }
  table broadcast {
    key = { meta.src_port: exact; }
    actions = {
        set_broadcast;
        NoAction;
    }
    default_action = NoAction;
  }
  apply {
    // forward based on destination Ethernet address
    if (!forward.apply().hit) {
      // miss, then broadcast
      broadcast.apply();
    }
  }
}

control Deparser(packet_out packet, in headers hdr) {...}

Switch(Parser(), TablePipeline(), Deparser()) main;
```

Fig. 2. An example P4 program

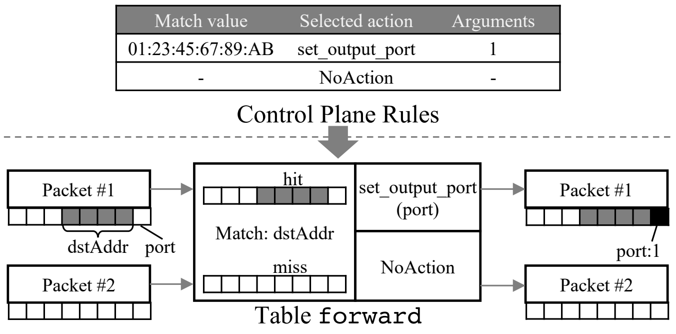
Fig. 3 describes a part of the P4 grammar about the table pipeline. `table_pipeline` contains action functions, table declarations, an apply function, and extern functions. The extern functions are implemented outside of the program like Verilog modules. The action function declarations and the apply function consist of statements such as assignment, if, and call statements. The table declaration consists of a list of keys and action names declared in `table_pipeline`. The keys contain the `id` of match variables and types of match.

```
   table_pipeline := control table_pipeline_name {
                         action_decl_list
                         table_decl_list
                         extern_decl_list
                         apply { stmt_list; } }
 action_decl_list := action_decl_list action_decl
                  := action_decl
      action_decl := action_name(args) { stmt_list; }
  table_decl_list := table_decl_list table_decl
                  := table_decl
       table_decl := table table_name {
                         key = { key_list; }
                         actions = { action_name_list; } }
         key_list := key_list; key
                  := key
              key := id : match_type
       match_type := exact
                  := ternary
                  := lpm
               id := packet header or metadata field
 action_name_list := action_name_list; action_name
                  := action_name
      extern_decl := extern extern_name (args);
        stmt_list := stmt_list; stmt
                  := stmt
             stmt := id = expr
                  := if(expr) {stmt_list}
                  := if(expr) {stmt_list} else {stmt_list}
                  := table_name.apply()
                  := extern_name(expr_list)
        expr_list := expr_list, expr
                  := expr
```

Fig. 3. The context-free grammar of a table pipeline in the P4 language

Fig. 4 shows the behavior of the table `forward` in Fig. 2. In the P4 program in Fig. 2, the table `forward` has Ethernet destination address (hdr.ethernet.dstAddr) with the `exact` option in the key (line 10) and two actions that invoke `set_output_port` and `NoAction` functions (line 11-14). Given the Ethernet destination address (`01:23:45:67:89:AB`) from network service providers in the control plane, the network switch compares the Ethernet destination address of a packet with the given value. If the address is the same as a given value, the switch sets the destination port number (meta.dst_port) as an argument (`1`) given by the control plane (`set_output_port`); otherwise, the switch passes the packet without modification (`NoAction`).

Although a data plane program consists of a parser, a table pipeline, and a deparser, this work focuses on optimizing the table pipeline because the data plane consumes most of its execution time in the table pipeline. While the parser extracts and the deparser packages the data, since the table pipeline reads/writes packet header fields or metadata, the table pipeline suffers from huge overheads. Furthermore, a recent publication [22] demonstrates that the latency of the table pipeline rapidly increases as the number of tables increases. Therefore, this work focuses on optimizing the table pipeline to minimize the packet processing latency of the programmable switch.



Match value	Selected action	Arguments
01:23:45:67:89:AB	set_output_port	1
-	NoAction	-

Fig. 4. The execution of table `forward`

$$
\begin{aligned}
\mathrm{U(stmt)} &:= \{\ \mathrm{id} \mid \mathrm{id} \in \mathrm{expr}\ \} \\
\mathrm{D(stmt)} &:= \mathrm{id} \\
\mathrm{U(stmt\_list)} &:= \mathrm{U(stmt\_list)} \cup (\mathrm{U(stmt)} \setminus \mathrm{D(stmt\_list)}) \\
\mathrm{D(stmt\_list)} &:= \mathrm{D(stmt\_list)} \cup \mathrm{D(stmt)} \\
\mathrm{U(keys)} &:= \{\ \mathrm{id} \mid \mathrm{id} \in \mathrm{keys}\ \} \\
\mathrm{D(keys)} &:= \emptyset \\
\mathrm{U(actions)} &:= \cup_{\mathrm{stmt\_list} \in \mathrm{action\_decl}} \mathrm{U(stmt\_list)} \\
\mathrm{D(actions)} &:= \cup_{\mathrm{stmt\_list} \in \mathrm{action\_decl}} \mathrm{D(stmt\_list)} \\
\mathrm{U(table)} &:= \mathrm{U(keys)} \cup \mathrm{U(actions)} \\
\mathrm{D(table)} &:= \mathrm{D(actions)}
\end{aligned}
$$

Fig. 5. The *use* and *def* of a P4 table. U(n) is a set of *use* of n, and D(n) is a set of *def* of n.

C. Limitation of Existing Compilers

Although P4 [2] extends the flexibility of network switches, existing P4 compilers [10]–[12] do not fully optimize the network programs because of the absence of fine-grained dependency analysis. Bosshart et al. [7] and Jose et al. [11] propose compilers that find data dependencies when a table modifies a packet header field and a subsequent table uses the field in its match (match dependency) or changes the field in its action (action dependency). However, they only focus on *table*-level data dependencies, so their pipeline scheduling becomes coarse-grained, losing possible parallelism opportunities among match and action operations.

The Table-level analysis considers a packet processing table as a unit of dependency analysis, thus ignoring the details of matches and actions. Fig. 5 shows the *use* and *def* of a table. Although match keys and actions have their uses and defs, and the existing compilers analyze the uses and defs of the match keys and actions, they only exploit the uses and defs in the table-granularity to find data dependencies. Therefore, to fully exploit parallelism opportunities in a finer-granularity, decoupling matches and actions from the tables is required in data dependency analysis.

Although decomposing tables into match functions and action functions is useful to find additional parallelism opportunities, the decomposition may increase computation and

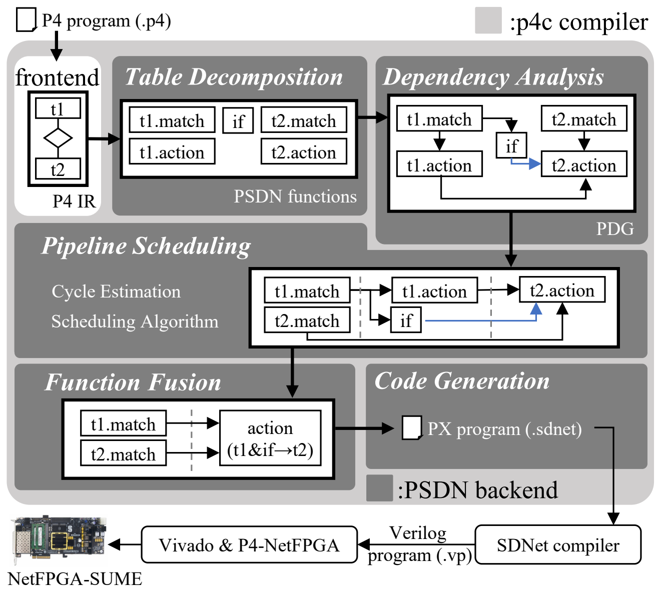


Fig. 6. Overall PSDN Compilation Process

area overheads of the synthesized hardware. Since the existing compilers [7], [11] allocate tables into hardware pipeline stages, their control flows are synthesized in a table-level granularity. Moreover, since match and action functions are synthesized together in one module, an optimization opportunity exists across match and action functions. However, since a fine-grained pipeline scheme that decomposes tables into match and action functions can allocate the match and action functions in the same table into different pipeline stages, the number of pipeline stages can increase, and the control flows become more complex. Since the increased pipeline stages and the complex control flows can cause unnecessary synchronization, the fine-grained pipeline scheme can increase the overall execution time even if each pipeline stage can be shorter. Therefore, reducing the number of pipeline stages and simplifying the control flows by decomposing only profitable tables are crucial for a P4 compiler to synthesize efficient switch hardware.

III. PSDN Compiler Design & Implementation

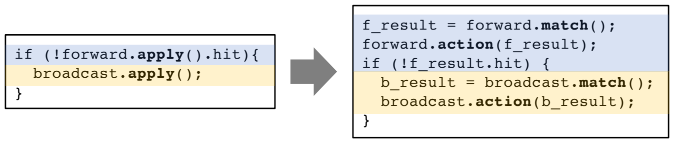
This work proposes the PSDN compiler to reduce latency and resource usage of network function programs. The compiler 1) decomposes packet processing tables into separated match functions and action functions, and 2) generates a program dependence graph, 3) efficiently allocates the functions in a pipeline manner respecting the dependencies and the latency estimation, and 4) finally generates a Xilinx PX program after fusing concurrent action functions and concatenating subsequent action functions.

A. Overview of Compilation Process

Fig. 6 describes an overview of the PSDN compiler. The frontend of the PSDN compiler is the p4c open-source compiler [23] that generates P4 intermediate representation (IR). Based on P4 IR, the PSDN compiler decomposes P4 tables into match functions and action functions (Section III-B). The PSDN compiler analyzes data and control dependencies among the functions and combines them into a program dependence graph (PDG) with pre-fetching the read-only functions (Section III-C). After the PDG is generated, the PSDN compiler performs cycle estimation and schedules the order of the functions into a pipeline (Section III-D). Here, the PSDN compiler allocates independent functions into the same pipeline stage to reduce the length of the pipeline. Finally, the PSDN compiler performs a function fusion scheme that merges adjacent action functions (Section III-E).

The PSDN compiler is in charge of translating a P4 IR to a PX program and optimizing the program. For the other parts of the compilation process, this work employs the p4c compiler as a frontend compiler and the SDNet compiler [24] as a backend compiler. This work implements the PSDN compiler on the top of p4c compiler, developing backend passes to translate P4 IR to an optimized PX program. After the PSDN compiler generates the optimized PX program, the SDNet compiler translates the program to a protected (encrypted)-Verilog program, which is synthesizable into FPGA hardware. Because the SDNet compiler is a commercial compiler and hides the translation details, this work cannot perform register transfer-level (RTL) optimizations. At the time of writing, this work follows the workflow of P4-NetFPGA-SUME [14] that includes the SDNet compiler. Still, we also consider developing an end-to-end compiler as future work to perform high-level synthesis and lift the pipeline limitations.

This work synthesizes the Verilog programs using Vivado 2018.2 and tests the programs on the NetFPGA-SUME board [6]. The evaluation infrastructure of this work is described in Section IV.


```
if (!forward.apply().hit){
  broadcast.apply();
}
```

```
f_result = forward.match();
forward.action(f_result);
if (!f_result.hit) {
  b_result = broadcast.match();
  broadcast.action(b_result);
}
```

Fig. 7. A table decomposition example of table pipeline in Line 28-34 of Fig. 2. The PSDN compiler decomposes the `apply` function into `match` and `action` functions.

B. Table Decomposition

A table decomposition scheme of the PSDN compiler decomposes match functions and action functions of P4 tables. Fig. 7 shows table decomposition of table pipeline (Line 28-34) in Fig. 2. The PSDN compiler first decomposes `forward.apply()` to `forward.match()` and `forward.action()`. The transformed code invokes `forward.match()` and saves its result in `f_result`. `forward.action()` performs actions with `f_result` as the argument. `if` statement needs the hit or miss result of the table `forward`, so the condition argument of the `if`

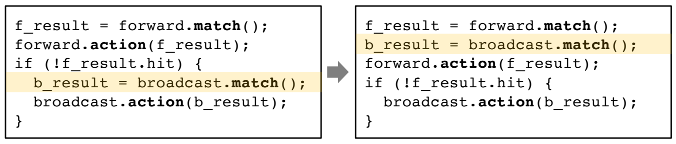
```
typedef enum { exact, lpm, ternary } LookupType;
typedef struct { bool hit; int action_id;
                 int* args; } Result;

LookupType f_type = LookupType.exact;
int f_keys[1] = [hdr.ethernet.dstAddr];

Result ForwardTable::match() {
  Result result = PERFORM_MATCH(f_type, f_keys);
  return result;
}

void ForwardTable::action(Result result) {
  if(result.hit) {
    switch(result.action_id){
      //case 1: set_output_port(), default: NoAction()
      case 1: meta.dst_port = result.args[0]; break;
      default: break;
    }
  }
}
```

Fig. 8. A match function and an action function of Table `forward` of Fig. 2



```
f_result = forward.match();
forward.action(f_result);
if (!f_result.hit) {
  b_result = broadcast.match();
  broadcast.action(b_result);
}
```

```
f_result = forward.match();
b_result = broadcast.match();
forward.action(f_result);
if (!f_result.hit) {
  broadcast.action(b_result);
}
```

Fig. 9. Code motion of stateless functions. Since the `match` function is stateless without modifying program states, the PDSN compiler moves `broadcast.match` outside of the `if` statement, thus allowing prefetching the `match` function and executing the `match` functions of `forward` and `broadcast` in parallel.

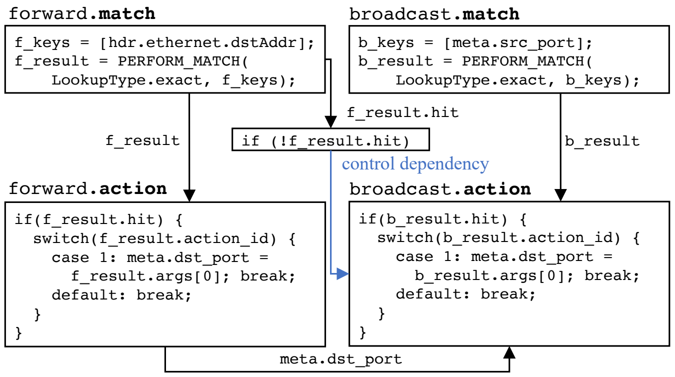
statement becomes `!f_result.hit`. The PSDN compiler decomposes the table `broadcast` in Fig. 2 in the same way.

Fig. 8 describes details of decoupled match and action function of table `forward` in Fig. 2. This paper barrows C++ semantics to describe the match and action functions. The match function compares keys with given values from the control plane rules and checks if they are matched or not. According to the matching result, the match function finds a corresponding P4 action and its arguments from the control plane rules. Finally, the match function returns the matching result (hit or miss), action type, and arguments. Given the return values from the match function, the action function decides the desired P4 action with the arguments. Here, since the PSDN compiler inlines P4 actions into one action function, the action function includes an `if` and a `switch` statement to select the correct P4 action among inlined actions.

P4 semantics allows custom instructions in the table pipeline as an `apply` function. For example, the example code in Fig. 2 defines an `apply` function that contains a conditional branch in Line 30. The PSDN compiler compiles these custom instructions into separate functions.

C. Dependency Analysis

Dependency analysis of the PSDN compiler follows traditional data and control dependency analysis [25], [26]. The PSDN compiler finds data dependencies with *use* and *def* information described in Fig. 5. The PSDN compiler draws



```
f_keys = [hdr.ethernet.dstAddr];
f_result = PERFORM_MATCH(
    LookupType.exact, f_keys);
```

```
b_keys = [meta.src_port];
b_result = PERFORM_MATCH(
    LookupType.exact, b_keys);
```

```
if(f_result.hit) {
  switch(f_result.action_id) {
    case 1: meta.dst_port =
      f_result.args[0]; break;
    default: break;
  }
}
```

```
if(b_result.hit) {
  switch(b_result.action_id) {
    case 1: meta.dst_port =
      b_result.args[0]; break;
    default: break;
  }
}
```

Fig. 10. Program Dependence Graph. A blue arrow is a control dependency, and black arrows are data dependencies.

a program dependence graph (PDG) by combining data and control dependencies. Here, the compiler redirects control dependencies from the table to the action function to prefetch read-only match functions.

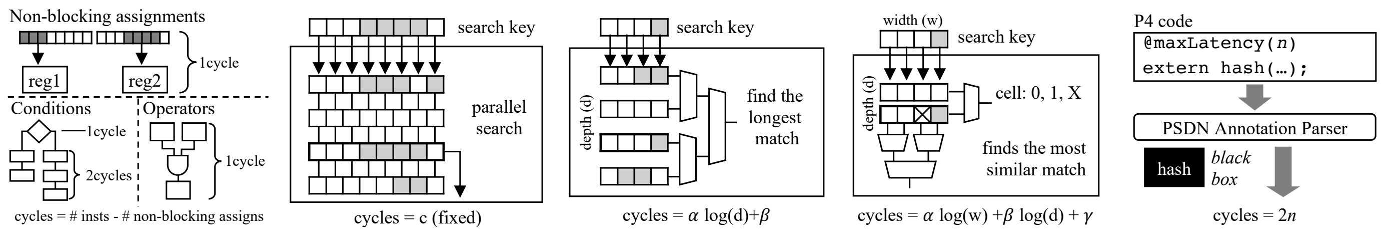
The match functions and some extern functions are *stateless*; the functions do neither modify the program's state nor depend on the control flow. Fig. 9 describes how the PSDN compiler manages the stateless functions. Since the `match` function only compares keys with control plane rules, the function only reads variables without modifying its program state. Some extern functions like register-read and hashing are also stateless if they do not modify the program state. The PSDN compiler moves invocations on the stateless functions outside the conditional branches, thus executing the stateless functions before the conditional branches. The prefetched execution increases parallelization opportunities like the `match` functions of `forward` and `broadcast` in Fig. 9 and reduces the execution time after the `if` statement is taken.

Fig. 10 shows the PDG of an example code in Fig. 2. The match function and the action function have data dependencies between each other. Also, `forward` and `broadcast` action functions have a data dependency because they modify the same metadata field (`meta.dst_port`). `if` statement requires `f_result.hit` and determines the execution of the `broadcast` table. The `broadcast` match function is stateless and independent from the `if` statement, so the PSDN compiler draws the control dependency from the if statement only to the `broadcast` action function.

D. Pipeline Scheduling

The PSDN compiler schedules the functions in PDG into a pipeline. The pipeline scheduler allocates independent functions in the same pipeline stage to reduce the length of the pipeline. The scheduler first estimates the latencies (clock cycles) of each function and allocates the functions with a greedy-based algorithm.

1) Clock Cycle Estimation: Clock cycle estimation of this work follows Fig. 11. For action functions, non-blocking assignments (which have no dependency between each other) take one cycle, and every blocking assignment and conditional



(a) Action function (b) CAM match function (c) LPM match function (d) TCAM match function (e) Extern function

Fig. 11. Cycle estimation of functions. Match function of type `exact`, `lpm`, `ternary` uses CAM, LPM, and TCAM match function, respectively.

Algorithm 1: Pipeline scheduling algorithm

```
Input   : A program dependence graph $G = (v, e)$
Output : A scheduled pipeline $P$ that is a list of pipeline stages
// PRED($v$): predecessors of $v$
// SUCC($v$): successors of $v$
$P \leftarrow \emptyset$
while $G \neq \emptyset$ do
    $I \leftarrow \{v \mid v \in G$ s.t. PRED(v) $= \emptyset\ \}$
    $N \leftarrow \cup_{v \in I}$ SUCC($v$)
    $R \leftarrow \cup_{v \in N}$ PRED($v$)
    $V \leftarrow$ Sort $v \in I \setminus R$ in assending order of Latency ($v$)
    for $v \in V$ do
        if Latency($v$) $\leq$ MaxLatency($R$) then
            $R \leftarrow R \cup \{v\}$
        else if MaxLatency($R$) $>$ MaxLatency($N$) then
            $R \leftarrow R \cup \{v\}$
        end
    end
    // $\oplus$: concatenation operator
    $P \leftarrow P \oplus R$
    $G \leftarrow G \setminus R$
end
```

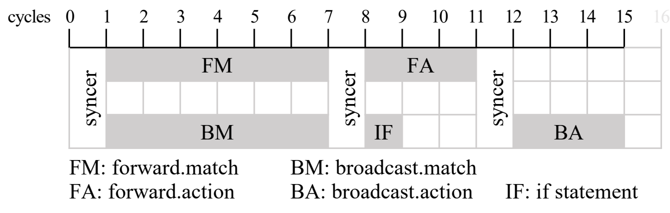


Fig. 12. A scheduled pipeline. Assume that a syncer logic takes one cycle.

statement takes one cycle each. For conditional branches, the estimator sums up the cycles of the longest branch among them. Therefore, the estimated latency of the action functions equals to the number of instructions in the longest branch minus the number of the non-blocking assignments (Fig. 11a).

The cycles of match functions depend on the types of match functions. The types of the match functions are exact match, ternary match, and longest prefix match. The cycle estimation formulae of match functions depend on their actual implementations, but this work abstracts the execution models of the match functions. A match function with type `exact` uses a content-addressable memory (CAM) that performs a parallel search to find an exact match with a key (Fig. 11b).

TABLE I
ESTIMATED CYCLES OF MATCH FUNCTIONS, ASSUMING THAT THE NUMBER OF ENTRIES $d = 256$.

Lookup type	Estimated cycles
exact	6
lpm	$20\ (= 2\log(256) + 4)$
ternary	$2\lfloor\log(\lceil w/40 \rceil)\rfloor + 6$

Therefore, the estimated latency of the CAM-based match function is constant.

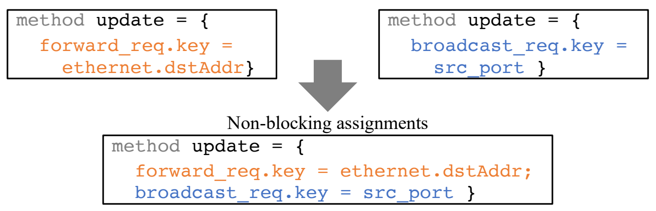
A match function with type `lpm` performs the longest prefix match (LPM) with search keys. Calculating the length of the prefix match is constant but finding the longest length depends on the number of entries (depth). Finding the longest match requires multiplexers. Therefore, the estimated latency of the LPM is a linear function of $\log(depth)$ (Fig. 11c).

A match function with type `ternary` uses a ternary-CAM (TCAM) module that allows don't-care terms (X) in entries. Finding the entries in TCAM depends on its implementation, but this work assumes that TCAM finds the most similar match. Therefore, TCAM requires multiplexers on calculating the match of each entry and finding the most similar match; the estimated latency is a linear function of $\log(width)$ and $\log(depth)$ (Fig. 11d). Table I shows the cycle estimation of the PSDN compiler uses. The cycle estimation of the match functions is architecture-specific, and this work uses these references [27]–[29].

The PSDN compiler considers the extern function as a *black box*. Because the compiler does not know the actual implementation of the extern function, the programmer should provide information about maximum latency with annotation. The PSDN compiler parses the annotation and determines the cycles of the extern function.

The backend FPGA hardware applies different clocks on action functions and match/extern functions. The clock period of the match functions and the extern functions is twice longer than the clock period of the action functions. This work also reflects the difference in clock periods when estimating the functions' cycles.

2) Pipeline Scheduling Algorithm: The pipeline scheduler allocates functions in PDG to a pipeline. Algorithm 1 describes the pipeline scheduling algorithm of this work. The algorithm

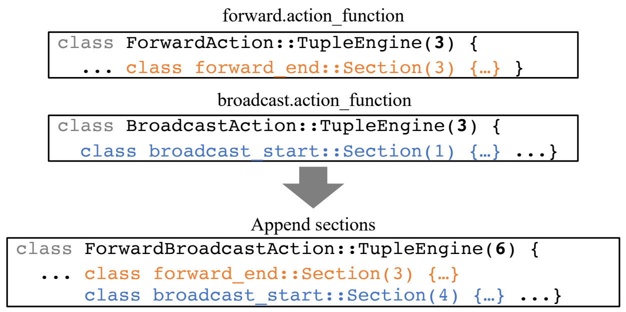


```
method update = {
  forward_req.key =
    ethernet.dstAddr}
```

```
method update = {
  broadcast_req.key =
    src_port }
```

```
method update = {
  forward_req.key = ethernet.dstAddr;
  broadcast_req.key = src_port }
```

(a) Merge concurrent action functions in the same pipeline stage



```
class ForwardAction::TupleEngine(3) {
  ... class forward_end::Section(3) {…} }
```

```
class BroadcastAction::TupleEngine(3) {
  class broadcast_start::Section(1) {…} ...}
```

```
class ForwardBroadcastAction::TupleEngine(6) {
  ... class forward_end::Section(3) {…}
      class broadcast_start::Section(4) {…} ...}
```

(b) Concatenate adjacent action functions

Fig. 13. Function fusion examples

first finds the independent functions (I) from the PDG and finds the next-independent functions (N) whose dependencies are resolved when allocating the functions in I. Next, the algorithm finds the required functions (R) necessary to resolve the dependencies of the functions in N.

The algorithm places all the functions in R in the pipeline stage and decides where to place the functions in $I \setminus R$. The pipeline stage of the function $v \in I \setminus R$ depends on $latency(v)$, $maxLatency(R)$, and $maxLatency(N)$. The scheduler chooses a stage that hides $latency(v)$ between the current stage (R) and the next stage (N). If $latency(v)$ is longer than $maxLatency(R)$ and $maxLatency(N)$, the scheduler chooses the stage that has longer latency. The algorithm repeats these steps until all the functions are allocated in the pipeline P.

Fig. 12 shows a pipeline allocation of PDG in Fig. 10. The functions allocated in the same pipeline stages will be executed in parallel. To synchronize the inputs and outputs of each pipeline stage, the PSDN compiler inserts a barrier named *syncer* between the pipeline stages. A syncer receives modified data from a previous stage, updates the global packet header values and metadata, and sends the packet headers and metadata to the next stage. Since the syncer requires additional clock cycles, inserting many syncers would increase the latencies and reduce the performance gain from the parallelization. Therefore, the PSDN compiler introduces the function fusion scheme to reduce the number of syncers described in the following section.

E. Function Fusion & Code Generation

To generate a PX program, the PSDN compiler adopts a similar way as the P4-SDNet compiler [12]. The PSDN compiler translates instructions in action functions into PX

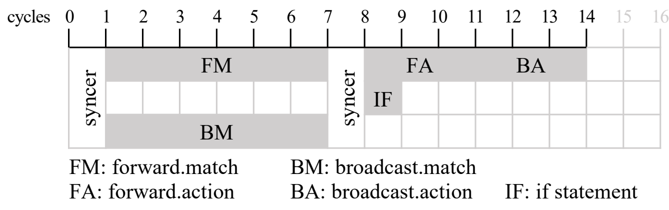


Fig. 14. A fused pipeline of the original pipeline in Fig. 12. Assume that a syncer logic takes one cycle.

instructions in tuple engines [13]. A tuple engine of PX language [13] has multiple *sections* that contain assignment statements (*update*) and a jump operation to the next section (*move_to_section*). The PSDN compiler puts the non-blocking assignment statements into the same section and translates conditional statements to the *move_to_section* operations.

While the PSDN compiler translates the functions, the PSDN compiler applies a function fusion scheme on action functions to reduce latency. First, the PSDN compiler *merges* the concurrent functions in the same pipeline stage into one tuple engine. For example, if the instructions in the action functions have no dependency between each other, the PSDN compiler places the instructions into one section and merges the functions (Fig. 13a). Second, the PSDN compiler *concatenates* an action function and the neighboring action function into one tuple engine. For example, the compiler places all the sections in the action functions into one tuple engine, redirects the `move_to_section` operation of the last section in the previous action function (`forward_end` in Fig. 13b) to point to the beginning section of the following action function (`broadcast_start` section in Fig. 13b), and increases the number of sections of the concatenated tuple engine. Note that the function fusion scheme is only applicable to action functions because match functions cannot be modified, and extern functions are considered black boxes.

Fig. 14 shows the fused pipeline. The PSDN compiler fuses the if statement, `forward` action function, and `broadcast` action function into one action function. The function fusion scheme merges the if statement in the last section of the `forward` action function. The compiler translates the if statement into a jump operation, so the if statement does not take additional cycles. Then, the function fusion scheme concatenates `forward` and `broadcast` action functions into one action function. The compiler finally removes the syncer between the two pipeline stages. Compared to Fig. 12, the fused pipeline takes a shorter processing time because of reducing syncers.

IV. Evaluation

A. Methods

This work evaluates how the PSDN compiler reduces packet processing latencies and resource utilization by HDL simulation and synthesis with P4 benchmarks from P4-NetFPGA GitHub [30]. The benchmark suite includes seven P4 programs: Learning Switch, In-Network Telemetry (INT), TCP

TABLE II
BENCHMARK DESCRIPTIONS AND SPECIFICATIONS. REG(REGISTER), HASH, AND TIMESTAMP ARE EXTERNS.

Program	**Description**	**# table**	**# reg**	**# hash**	**# time stamp**
Learning Switch	learns forwarding rules from packets	3	0	0	0
INT	collects network states in real-time	1	1	0	1
TCP Monitor	monitors TCP connections	1	2	1	0
Switch Calc.	performs basic arithmetic operations	1	1	0	0
Basic FRED	drops packets based on queing lengths	3	4	0	1
Heavy Hitter	finds over-flowed packets	2	3	0	1
Flow Rate	calculates flow rate of packets	2	7	0	1

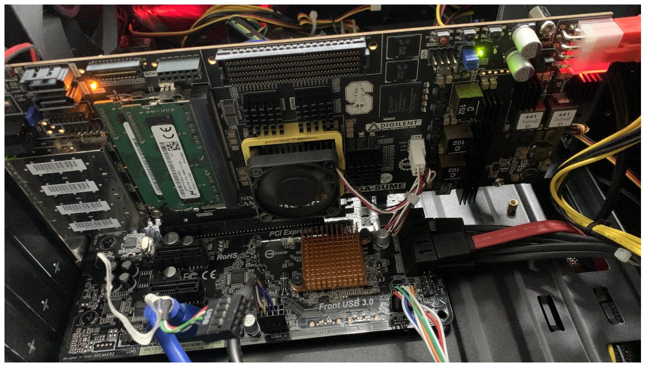


Fig. 15. NetFPGA-SUME board for hardware evaluation

Monitor, Switch Calculator, Basic Fair Random Early Detection (Basic FRED), Heavy Hitter, and Flow Rate. Table II shows the descriptions and specifications of each program. Some programs have independent and parallelizable functions, but the other programs have sequential structures to monitor and calculate packet information. The parallel or sequential structures of programs will determine the latency reduction of the PSDN compiler.

This work measures end-to-end packet processing latency by HDL simulation with Vivado 2018.2. This work also synthesizes the compiled programs to measure resource utilization. The testing hardware is a NetFPGA-SUME board [6] equipped with a Xilinx Vertex-7 FPGA module and four 10 Gbps network ports (Fig. 15).

To show that the PSDN compiler effectively reduces the packet processing latency and resource utilization of compiled programs, this work compares the proposed methods with previous work [12]:

- **Previous work**: performs table-level pipeline scheduling and optimizations.

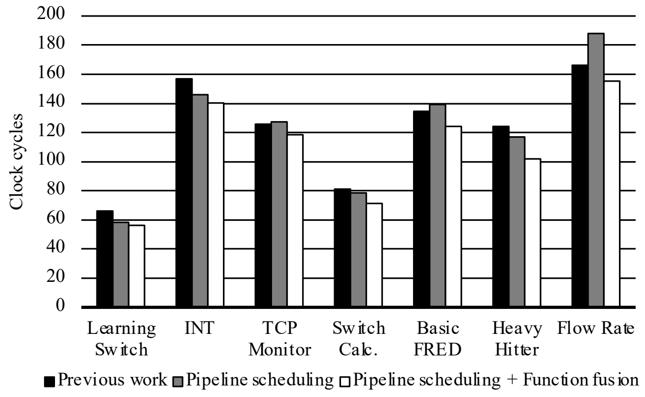


Fig. 16. Packet processing latency. 12.1% reduction in a geometric mean.

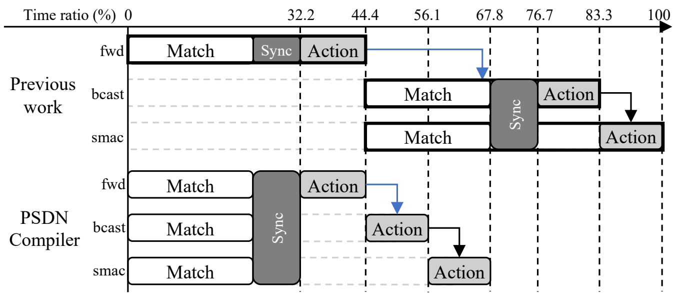


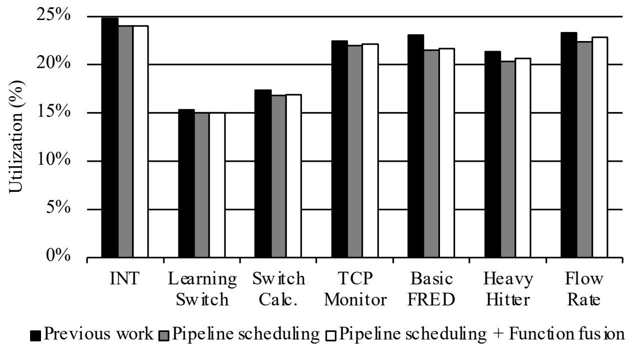
Fig. 17. Time ratio of functions in table pipeline of Learning Switch. Blue and black arrows are control and data dependencies. fwd, bcast, and smac are forward, broadcast, and smac tables, respectively.

- **Pipeline scheduling**: performs table decomposition and function-level pipeline scheduling without applying function fusion scheme.
- **Pipeline scheduling + Function fusion**: performs table decomposition, function-level pipeline scheduling, and function fusion scheme.

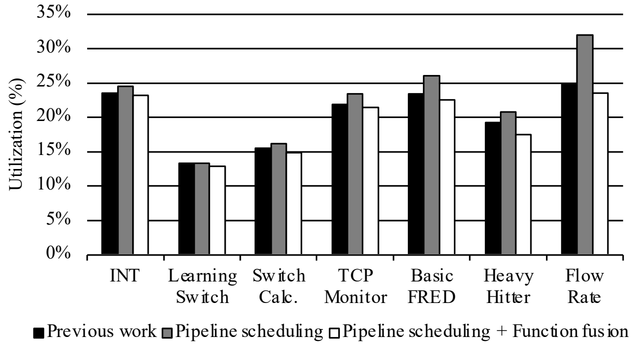
B. Results

Latency: This work measures the end-to-end packet processing latency of the compiled programs by HDL simulation. Fig. 16 shows clock cycles taken by the compiled programs. Compared to the previous work, the pipeline scheduling with function fusion reduces the latency by 12.1% in a geometric mean. Here, function-level pipeline scheduling does not reduce latencies of TCP Monitor, Basic FRED, and Flow Rate compared to the previous work because the three P4 programs have less parallelizable functions than the other programs. For the three programs, table decomposition on every table increases the number of functions and the pipeline length. By applying function fusion that merges the adjacent functions, the compiler reduces the number of action functions and synchronization overheads, exposing performance improvement of function-level parallelization.

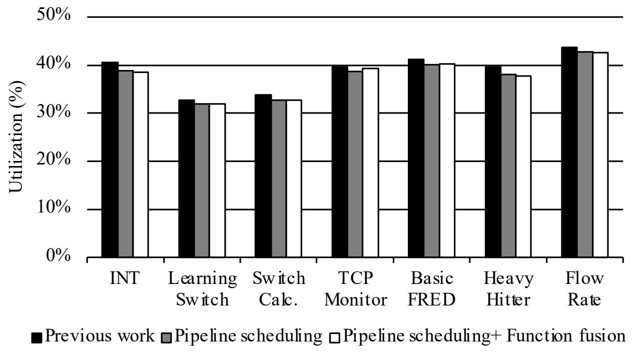
To deeply analyze how the PSDN compiler effectively reduces the latency, this work measures the cycles of functions in the table pipeline of the Learning Switch benchmark compared



(a) LUTs. 3.07% reduction in a geomean.



(b) Registers. 4.29% reduction in a geomean.



(c) Memory. 3.13% reduction in a geomean.

Fig. 18. Resource utilization (lower is better). 3.5% reduction in a geometric mean.

to the previous work (Fig. 17). The Learning Switch program has three tables: forward, broadcast, and smac. There exist a control dependency between forward and broadcast tables and a data dependency between actions of broadcast and smac tables. The previous work [12] considers the tables as execution units, so it sequentially allocates forward and broadcast tables. Because the previous work considers only match dependency or action dependency [7], it parallelizes only the match functions of broadcast and smac tables. On the other hand, the PSDN compiler decomposes the tables into match functions and action functions and redirects the control dependency to action functions to prefetch the read-only match functions. Therefore, the PDSN compiler can parallelize all the match functions of the three tables.

Resource utilization: This work measures the resource utilization of the synthesized programs in lookup tables (LUTs; representing combinational logics, not packet processing tables), registers, and memory. This work installs the synthesized programs into the NetFPGA-SUME board, which has Xilinx xc7vx690t FPGA module. Fig. 18 shows the resource utilization percentage of each unit. Compared to the previous work, the PSDN compiler reduces resource usage by 3.5% in a geometric mean.

In Fig. 18a, table decomposition and function-level pipeline scheduling reduce the usages of combinational logic compared to the previous work. The table decomposition reduces the number of conditional branches by placing separate action functions in parallel, thus reducing the combinational logic usage. Since the function fusion scheme merges functions with additional conditional branches, the LUT usage with the function fusion is slightly higher than the usage without the function fusion. Though the function function scheme increases the LUT usage, the PSDN compiler still reduces the LUT usage by 3.07% in a geometric mean compared to the previous work.

In terms of the register usage (Fig. 18b), table decomposition uses more registers than the previous work. A function needs to transfer all the data to the following functions in a pipeline, thus consuming registers for its data bits. Although making more functions increase register usages, by applying the function fusion scheme on the decomposed pipeline, the PSDN compiler reduces the register usage by 4.29% in a geometric mean compared to the previous work.

In Fig. 18c, pipeline scheduling reduces memory (BRAM) usage compared to the previous work. The memory utilization is related to the internal synchronization buffers for match functions and extern functions. By placing the independent match and extern functions into the same pipeline stage, pipeline scheduling reduces synchronization buffers. Compared to the previous work, the PSDN compiler reduces the memory resource usage by 3.13% in a geometric mean.

V. Related Work

Compilers and languages for network packet processing [31]–[33] provide programming tools for writing packet processing programs and optimize the programs. Aspen [31] is a language that supports the concurrency of network server applications. Code reuse on SDN programming [32], [33] simplifies network programming by providing reusable building blocks. This work adopts P4 language [2] as a frontend language, but the proposed parallelization method can be applied if the language has table pipeline semantics.

Previous work targeting the RISC-based network processors proposes compiler optimization techniques like register allocation [34]–[36], bit-level instruction partitioning [37], and resolving bank conflicts [38]. Although this work does not target RISC network processors, this work will improve the performance of the CPU-based multi-core packet processors with parallelization.

Some studies propose the compilers that optimize packet parsers [39]–[41]. The packet parser is a part of programmable data plane, but the parser's latency is shorter than the table pipeline. The reason is that the parser only reads packet header values to identify the protocol types of the packets while the table pipeline reads and modifies packet header values and metadata information. The table pipeline is the main bottleneck of the programmable data plane, so this work optimizes the pipeline to minimize overall latency.

To increase the throughput of packet processing applications, researchers have proposed compilers that exploit application partitioning algorithms [42]–[44]. The proposed implementations partition imperative packet processing programs into several basic blocks and make a pipeline to increase the throughput. This work also proposes the table decomposition in a similar way but exploits the characteristics of match and action functions.

Recent work by Jose et al. [11] adopts parallelization to optimize latency of P4 [2] packet processing applications. Jose et al. [11] propose a compiler that maps P4 logical tables into physical tables on packet processing architectures [7], [9]. The proposed compiler exploits a table dependency graph to find data dependencies and Integer Linear Programming (ILP) to map logical tables to physical tables. While the compiler by Jose et al. adopts pipeline scheduling on table-level, the compiler of this work decouples matches and actions from the tables and schedules the matches and the actions into the pipeline in a fine-grained manner.

P4FPGA [10] is a rapid prototyping compiler that translates P4 programs into Bluespec System Verilog [45], [46] programs. One of their approaches to reducing latency is co-locating independent instructions into the same pipeline stage. This work also performs a similar way in the function fusion scheme, but the main difference is that P4FPGA does not consider how to reorder and map the packet processing tables into the pipeline preserving table-level dependency. In other words, P4FPGA only supports intra-table optimization, but this work supports intra- and inter-table optimization.

Previous work on programmable ASIC compilers [47]–[49] primarily aims to overcome the limitations of chip-specific language and architecture. Gao et al. [47] leverage domain-specific synthesis techniques to accelerate compilation and target multiple backends using a pipeline description language. To support programming independent of the underlying hardware architecture, μP4 [48] increases the abstraction level in target-specific packet processing pipelines and configurations. Lyra [49] provides a one-big-pipeline abstraction to allow programmers to conveniently express their algorithms and generates chip-specific codes for distributed switches.

VI. Conclusion

This work proposes a new compiler that supports fine-grained pipeline scheduling and function fusion for network function programs. Previous compilers conduct packet processing table-level analysis and pipeline scheduling, but the compiler of this work decomposes the tables into subdivided functions and performs fine-grained static analysis and pipeline scheduling. This work also proposes function fusion that reduces the number of functions and synchronization overheads caused by the table decomposition. This work shows that the proposed schemes reduce packet processing latency by 12.1% and resource utilization by 3.5% compared to the previous work.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We also thank the CoreLab members for their support and feedback during this work. This work is supported by IITP-2017-0-00195, IITP-2018-0-01392, and IITP-2020-0-01847 through the Institute of Information and Communication Technology Planning and Evaluation (IITP) funded by the Ministry of Science and ICT. Hanjun Kim is the corresponding author of this paper.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] 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 Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[3] 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 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 525–538.

[4] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: ACM, 2018, pp. 54–66.

[5] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015, pp. 117–130.

[6] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep. 2014.

[7] 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 *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 99–110.

[8] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "dRMT: Disaggregated Programmable Switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 1–14.

[9] R. Ozdag, "Intel® Ethernet Switch FM6000 Series-Software Defined Networking," *Intel Cooperation*, 2012.

[10] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A Rapid Prototyping Framework for P4," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 122–135.

[11] 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.

[12] Xilinx, "P4-SDNet User Guide," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1252-p4-sdnet.pdf, 2018.

[13] G. Brebner and W. Jiang, "High-Speed Packet Processing using Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 8–18, Jan 2014.

[14] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4->NetFPGA Workflow for Line-Rate Packet Processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019, pp. 1–9.

[15] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," *ONF White Paper*, April 2012.

[16] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 1–6.

[17] J. Hyun, N. Van Tu, and J. W. Hong, "Towards knowledge-defined networking using in-band network telemetry," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, April 2018, pp. 1–7.

[18] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 15–28.

[19] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-Network Computation is a Dumb Idea Whose Time Has Come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XVI. New York, NY, USA: ACM, 2017, pp. 150–156.

[20] H. Song, "Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 127–132.

[21] S. Li, D. Hu, W. Fang, S. Ma, C. Chen, H. Huang, and Z. Zhu, "Protocol Oblivious Forwarding (POF): Software-Defined Networking with Enhanced Programmability," *IEEE Network*, vol. 31, no. 2, pp. 58–66, March 2017.

[22] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A P4 Language Benchmark Suite," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 95–101.

[23] Barefoot, "P4_16 reference compiler," https://github.com/p4lang/p4c, 2020.

[24] Xilinx, "SDNet Packet Processor User Guide," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1012-sdnet-packet-processor.pdf, 2018.

[25] R. Cytron, J. Ferrante, and V. Sarkar, "Compact representations for control dependence," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI '90. New York, NY, USA: ACM, 1990, pp. 337–351.

[26] W. Pugh, "The omega test: A fast and practical integer programming algorithm for dependence analysis," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 4–13.

[27] Xilinx, "Exact Match Binary CAM Search IP for SDNet," https://www.xilinx.com/support/documentation/ip_documentation/cam/pg189-cam.pdf, 2019.

[28] ——, "Ternary Content Addressable Memory (TCAM) Search IP for SDNet," https://www.xilinx.com/support/documentation/ip_documentation/tcam/pg190-tcam.pdf, 2017.

[29] ——, "Longest Prefix Match (LPM) Search IP for SDNet," https://www.xilinx.com/support/documentation/ip_documentation/lpm/pg191-lpm.pdf, 2017.

[30] NetFPGA Github Organization, "P4-NetFPGA-public," https://github.com/NetFPGA/P4-NetFPGA-public, 2018.

[31] G. Upadhyaya, V. S. Pai, and S. P. Midkiff, "Expressing and Exploiting Concurrency in Networked Applications with Aspen," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 13–23.

[32] H. Eran, L. Zeno, Z. Istvn, and M. Silberstein, "Design Patterns for Code Reuse in HLS Packet Processing Pipelines," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2019, pp. 208–217.

[33] 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. New York, NY, USA: ACM, 2013, pp. 87–98.

[34] J. Wagner and R. Leupers, "C Compiler Design for an Industrial Network Processor," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '01. New York, NY, USA: ACM, 2001, pp. 155–164.

[35] J. Kim, S. Jung, Y. Paek, and G.-R. Uh, "Experience with a Retargetable Compiler for a Commercial Network Processor," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '02. New York, NY, USA: ACM, 2002, pp. 178–187.

[36] L. George and M. Blume, "Taming the IXP Network Processor," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 26–37.

[37] S. Carr and P. Sweany, "Automatic Data Partitioning for the Agere Payload Plus Network Processor," in *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '04. New York, NY, USA: ACM, 2004, pp. 238–247.

[38] X. Zhuang and Santosh Pande, "Resolving register bank conflicts for a network processor," in *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2003, pp. 269–278.

[39] J. Santiago da Silva, F.-R. Boyer, and J. P. Langlois, "P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 147–152.

[40] P. Bencek, V. Pu, and H. Kubtov, "P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 148–155.

[41] A. Yazdinejad, A. Bohlooli, and K. Jamshidi, "P4 to SDNet: Automatic Generation of an Efficient Protocol-Independent Packet Parser on Reconfigurable Hardware," in *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*, Oct 2018, pp. 159–164.

[42] J. Dai, B. Huang, L. Li, and L. Harrison, "Automatically Partitioning Packet Processing Applications for Pipelined Architectures," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 237–248.

[43] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju, "Shangri-La: Achieving High Performance from Compiled Network Applications While Enabling Ease of Programming," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 224–236.

[44] 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 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 15–28.

[45] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, June 2004, pp. 69–70.

[46] R. S. Nikhil, *Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions*. Dordrecht: Springer Netherlands, 2008, pp. 129–146.

[47] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta, "Switch code generation using program synthesis," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 4461.

[48] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, "Composing dataplane programs with μP4," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer*

Communication, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 329343.

[49] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 435450.