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%.
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 programs that switch the control programs 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 keys that contain packet headers or metadata for match operations and actions 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.

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

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_list action_decl 
action_name := action_name(args) { stmt_list; } 
table_decl_list := table_decl_list table_decl 
table_decl := table_decl_list 
  key := { key_list; } 
  actions := { action_name_list; } 
key_list := key_list; key 
key := id ; match_type 
match_type := exact 
  := ternary 
  := lpm 
iid := 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. 4. The execution of table forward

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.

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.

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
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.

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;

5 LookupType f_type = LookupType.exact;
6 int f_keys[1] = [hdr.ethernet.dstAddr];
7 Result ForwardTable::match() {
8    Result result = PERFORM_MATCH(f_type, f_keys);
9    return result;
10 }
11 }
12 void ForwardTable::action(Result result) {
13    if(result.hit) {
14        switch(result.action_id) {
15            case 1: meta.dst_port = result.args[0]; break;
16            default: break;
17        }
18    }
19 }
20 }
21 }

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

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

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 not 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
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
// \( \text{PRED}(v) \): predecessors of \( v \)
// \( \text{SUCCE}(v) \): successors of \( v \)
\( P \leftarrow \emptyset \)
while \( G \neq \emptyset \) do
    \( I \leftarrow \{ v \mid v \in G \text{ s.t. } \text{PRED}(v) = \emptyset \} \)
    \( N \leftarrow \bigcup_{v \in I} \text{SUCCE}(v) \)
    \( R \leftarrow \bigcup_{v \in N} \text{PRED}(v) \)
    \( V \leftarrow \text{Sort } v \in I \cap R \text{ in asending order of } \text{Latency}(v) \)
    for \( v \in V \) do
        if \( \text{Latency}(v) \leq \text{MaxLatency}(R) \) then
            \( R \leftarrow R \cup \{ v \} \)
        else
            if \( \text{MaxLatency}(R) > \text{MaxLatency}(N) \) then
                \( R \leftarrow R \cup \{ v \} \)
            end
        end
    end
// \( \ominus \): concatenation operator
\( P \leftarrow P \ominus R \)
\( G \leftarrow G \setminus R \)
end
```

### TABLE I

<table>
<thead>
<tr>
<th>Lookup type</th>
<th>Estimated cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>exact</td>
<td>6</td>
</tr>
<tr>
<td>lpm</td>
<td>( 20 (= 2 \log(256) + 4) )</td>
</tr>
<tr>
<td>ternary</td>
<td>( 2 \log([w/40]) + 6 )</td>
</tr>
</tbody>
</table>

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

A match function with type \( \text{lpm} \) 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 \( \log(\text{depth}) \) (Fig. 11c).

A match function with type \( \text{ternary} \) 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 \( \log(\text{width}) \) and \( \log(\text{depth}) \) (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...
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 $\text{latency}(v)$, $\text{maxLatency}(R)$, and $\text{maxLatency}(N)$. The scheduler chooses a stage that hides $\text{latency}(v)$ between the current stage ($R$) and the next stage ($N$). If $\text{latency}(v)$ is longer than $\text{maxLatency}(R)$ and $\text{maxLatency}(N)$, 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 $\text{syncer}$ between the pipeline stages. A $\text{syncer}$ 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 $\text{syncer}$ requires additional clock cycles, inserting many $\text{syncers}$ 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 $\text{syncers}$ 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 instructions in tuple engines [13]. A tuple engine of PX language [13] has multiple sections that contain assignment statements ($\text{update}$) and a jump operation to the next section ($\text{move_to_section}$). The PSDN compiler puts the non-blocking assignment statements into the same section and translates conditional statements to the $\text{move_to_section}$ 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 $\text{move_to_section}$ operation of the last section in the previous action function ($\text{forward_end}$ in Fig. 13b) to point to the beginning section of the following action function ($\text{broadcast_start}$ 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, $\text{forward action}$ function, and $\text{broadcast action}$ function into one action function. The function fusion scheme merges the if statement in the last section of the $\text{forward action}$ 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 $\text{forward}$ and $\text{broadcast}$ action functions into one action function. The compiler finally removes the $\text{syncer}$ between the two pipeline stages. Compared to Fig. 12, the fused pipeline takes a shorter processing time because of reducing $\text{syncers}$.

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.

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

---

**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.

**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...
<table>
<thead>
<tr>
<th>Utilization (%)</th>
<th>Previous work</th>
<th>Pipeline scheduling</th>
<th>Pipeline scheduling + Function fusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) LUTs. 3.07% reduction in a geometric mean.

<table>
<thead>
<tr>
<th>Utilization (%)</th>
<th>Previous work</th>
<th>Pipeline scheduling</th>
<th>Pipeline scheduling + Function fusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Registers. 4.29% reduction in a geometric mean.

<table>
<thead>
<tr>
<th>Utilization (%)</th>
<th>Previous work</th>
<th>Pipeline scheduling</th>
<th>Pipeline scheduling + Function fusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) Memory. 3.13% reduction in a geometric mean.

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

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 fusion 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 collocating 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 sub-divided 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
