SPEED: Resource-Efficient and High-Performance Deployment for Data Plane Programs

Xiang Chen*†‡, Hongyan Liu§, Qun Huang*, Peiqiao Wang‡, Dong Zhang‡, Haifeng Zhou§††, Chunming Wu§††
*Peking University †Pengcheng Lab ‡Fuzhou University §Zhejiang University ††Zhejiang Lab

***Abstract*—Programmable switches allow network administrators to customize packet processing behaviors in data plane programs. However, existing solutions for program deployment fail to achieve resource efficiency and high packet processing performance. In this paper, we propose SPEED, a system that provides resource-efficient and high-performance deployment for data plane programs. For resource efficiency, SPEED merges input data plane programs by reducing program redundancy. Then it abstracts the substrate network into an one big switch (OBS), and deploys the merged program on the OBS while minimizing resource usage. For high performance, SPEED searches for the performance-optimal mapping between the OBS and the substrate network with respect to network-wide constraints. It also maintains program logics among different switches via inter-device packet scheduling. We have implemented SPEED on a Barefoot Tofino switch. The evaluation indicates that SPEED achieves resource-efficient and high-performance deployment for real data plane programs.**

I. INTRODUCTION

Recent advances in programmable switches (e.g., RMT [1]) allow network administrators to develop novel network protocols and functionalities (e.g., key-value cache [2] and sketches [3, 4]) in data plane programs. Through the switch compiler, administrators deploy these programs on programmable switches to change switch behaviors in accordance with the demands raised by network applications such as network monitoring [5] and traffic engineering [6]. In particular, network applications impose two requirements for the deployment of data plane programs. First, applications should achieve *resource efficiency*. In particular, the resources of programmable switches are scarce, thereby the program deployment should occupy as few resources as possible to support more programs. Second, applications expect *high performance*, including high throughput and low end-to-end latency.

However, to the extent of our knowledge, none of the existing solutions can satisfy the two requirements. Existing solutions [7, 8, 9, 10, 11] concentrate on the program deployment on a *single* programmable switch. In this case, it only needs to deal with the resource usage of the single switch, which is relatively simple. However, the scenario of multiple switches raises more challenges. For example, deploying multiple programs can easily exceed the capacity of available resources in a single switch. It needs to consolidate resources from multiple switches and allocate the resources coordinately. In addition, the performance across multiple switches is determined by various factors, including per-switch processing and traffic paths. Taking all issues makes the network-wide placement much more challenging.

In this paper, we propose SPEED, a system that offers resource-efficient and high-performance deployment for data plane programs. However, it is not trivial to design such a system due to program diversity, heterogeneous constraints, and the need to preserve program logics. In response, we design SPEED to address the above challenges. Specifically, SPEED first transforms each program into a table dependency graph (TDG) [12], which is a universal intermediate representation. Doing so tackles program diversity and eases further placement. Next, SPEED merges TDGs into a compound TDG. In this step, it identifies and reduces redundancy among TDGs to save switch resources and enhance resource efficiency. Thereafter, SPEED handles heterogeneous constraints by abstracting the substrate network into an one big switch (OBS), i.e., a virtual programmable switch that shields network-wide constraints. It places the compound TDG on the OBS with the objective of minimizing resource usage while taking switch resource restrictions into account. Then it searches for the performance-optimal mapping between the OBS and the substrate network with respect to network-wide constraints. Finally, it offers inter-device packet scheduling to maintain the original program logics among switches.

In summary, this paper makes the following contributions:

- We present our motivation of realizing resource-efficient and high-performance program deployment (§II). We identify the design challenges and propose SPEED to tackle these challenges (§III).
- We articulate the design of SPEED, including program merging that reduces program redundancy (§IV), TDG placement on the OBS that minimizes resource usage (§V), OBS placement on the substrate network that maximizes performance (§VI), and inter-device packet scheduling that preserves original program logics (§VII).
- We implement a SPEED prototype with a Barefoot Tofino switch [13] (§IX). Our evaluation indicates that for the ten tested data plane programs, SPEED achieves resource-efficient and high-performance program deployment (§X).

II. BACKGROUND AND MOTIVATION

Background. A data plane program comprises several match-action tables (MATs). An MAT matches packet headers and performs actions on packets based on the matching results.

Qun Huang is the corresponding author.

978-1-7281-6992-7/20/$31.00 ©2020 IEEE

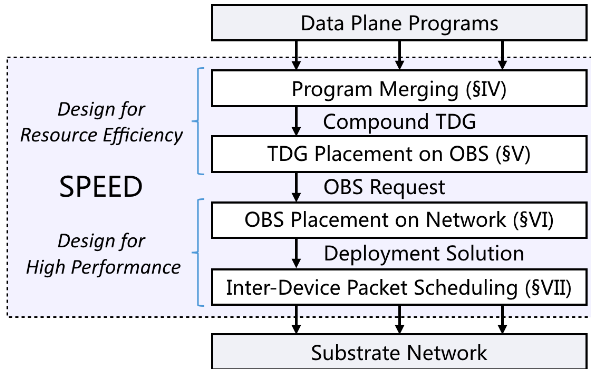


Fig. 1: Workflow of SPEED.

With MATs, administrators can develop novel network protocols and functions in a hardware-compatible manner. Moreover, programmable switches offer the customizable ASIC pipeline comprising a fixed number of stages [1, 7]. Each stage contains a region of SRAM and TCAM resources that support the deployment of data plane programs.

Goals. In this paper, we address the problem of deploying multiple programs on a network that contains several programmable switches. Our work is motivated by recent advances in the area of programmable networks, which raises the need of simultaneously deploying multiple data plane programs in a network. For example, software-defined measurement concurrently runs multiple measurement tasks to monitor various traffic statistics [14, 15]. In cloud environments, network operators permit multiple tenants' programs to share network resources [16]. In network function virtualization, administrators deploy several network functions (e.g., in-network caching [2]) to provide value-added services [11, 9].

Such deployment needs to address both *resource efficiency* and *high performance*.

- **Resource efficiency**. The resource capacity of the programmable switch is extremely limited. For instance, the memory of a programmable switch is typically tens of MB [1, 17]. Thus, we should make the best use of switch resources to reduce overall costs.
- **High performance**. Many applications impose strict requirements on the performance, especially the end-to-end latency [18, 19]. For example, adding 1 ms to the latency leads to ∼80% performance drop on the Memcached server [20]. Thus, we need to achieve high performance.

III. Design Overview of SPEED

In this section, we illustrate the design challenges and our corresponding design of SPEED.

Challenges. However, we face three design challenges in program deployment. First, we need to handle the diversity of multiple programs. For example, the program that realizes a Count-Min sketch [21] invokes MATs sequentially for packet processing, while the program for in-network caching [2] defines many branches, each of which invokes a portion of MATs, to conditionally process packets. Second, the constraints for resource efficiency and high performance are heterogeneous. For instance, the resource restrictions imposed by programmable switches are independent of end-to-end performance metrics. Consolidating them is non-trivial. Finally, a data plane program may be splitted across multiple switches after deployment. We need to preserve its packet processing logics. This requires careful program splitting and efficient inter-device coordination.

Design overview. To address the above challenges, we propose SPEED, a system that provides resource-efficient and high-performance program deployment, as shown in Figure 1. First, SPEED converts input programs into TDGs to address the challenge of program diversity. Then it offers an algorithm based on longest common subsequence (LCS) [22] that merges TDGs into a compound TDG with respect to the LCS of mergeable MATs. In particular, the LCS-based algorithm reduces program redundancy when merging TDGs so as to enhance resource efficiency. Second, to tackle the heterogeneous constraints, SPEED abstracts the substrate network into an OBS. It places the compound TDG on the OBS with respect to switch resource restrictions while minimizing the number of occupied stages. Next, it searches for the performance-optimal mapping between the OBS and the substrate network while taking network-wide constraints into account. Third, SPEED offers inter-device packet scheduling that correctly directs traffic to preserve the original program logics. It also enables the exchange of metadata fields among different switches.

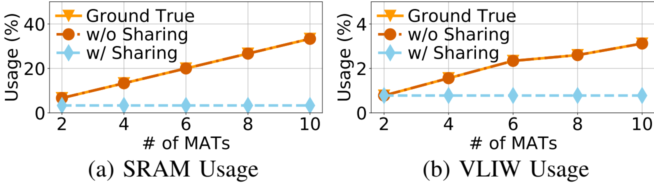
IV. Program Merging

In this section, we elaborate how SPEED merges input programs into a compound TDG. We illustrate the notion of TDG (§IV-A) and the workflow of program merging (§IV-B).

A. TDG Generation

Given a data plane program, SPEED generates the corresponding TDG. A TDG is a direct acyclic graph (TDG) $T = (V_T, E_T)$ [7]. The nodes in V_T represent the MATs defined in the program, and the directed edges in E_T represent the execution dependencies between MATs. Each MAT $u \in V_T$ has the following properties: (1) a set f_u^m recording the matching fields of u; (2) a set a_u recording the actions of u; (3) a set f_u^a recording the fields modified by the actions of u; (4) the match type, i.e., exact, longest prefix match, or wildcard; (5) the maximum number of rules e_u; and (6) the width (in bits) of each rule w_u. Each edge $(u, v, t_{u,v}) \in E_T$ indicates the dependency between two MATs, u and v, where v is a downstream MAT of u. The dependency type $t_{u,v}$ of an edge can be (1) *Match dependency* ($\mathbb{M}$): u modifies a field $f \in f_u^a \cap f_v^m$; (2) *Action dependency* ($\mathbb{A}$): both u and v modify the same field f, i.e., $f \in f_u^a \cap f_v^a$; (3) *Reverse match dependency* ($\mathbb{R}$): u matches a field $f \in f_u^m \cap f_v^a$; (4) *Successor dependency* ($\mathbb{S}$): whether to execute v depends on f_u^a; and (5) *No dependency* ($\mathbb{N}$): u and v are not interdependent.

To generate a TDG T, SPEED enumerates every MAT and obtains MAT properties to build TDG nodes. Then it identifies the dependency between each pair of MATs to construct TDG edges. Also, according to TDG edges, it creates a dependency



(a) SRAM Usage (b) VLIW Usage

Fig. 2: Impact of merging MATs on resource usage.

matrix D_T [10]: given an edge $(u, v, t_{u,v}) \in E_T$, $D_T[u, v] = 1$ if $t_{u,v} \in \{\mathbb{M}, \mathbb{A}, \mathbb{R}, \mathbb{S}\}$; $D_T[u, v] = 0$ if $t_{u,v} = \mathbb{N}$.

B. TDG Merging

Motivation. Our motivation of program merging comes from the need to reduce resource usage and the occurrence of redundant operations. For example, in software-defined measurement (SDM) [14, 15, 23, 24, 25, 26, 27], the applications (e.g., anomaly detection) require to measure various traffic statistics (e.g., per-flow count and flow cardinality). However, a measurement algorithm usually only measures partial statistics, which lacks of generality. Thus, SDM often simultaneously deploys multiple SDM algorithms (e.g., sketches [3, 4]) in the network for collecting multiple types of statistics, which highlights the need to optimize their overall resource consumption. Moreover, we observe that different SDM algorithms are likely to perform the same operations; for instance, all sketch algorithms need to compute several hash functions to locate the positions to store statistics. This raises the opportunity of program merging.

Observations. Our design of program merging is based on two observations. First, we observe that *merging MATs can reduce the overall resource usage if and only if MATs are mergeable*. We define that two MATs are *mergeable* if (1) their matching fields and actions are the same, and (2) their resources can be shared between them. To justify our observation, we study the impact of merging MATs on resource usage on a Barefoot Tofino switch [13]. Above all, we write a data plane program in P4 [12]. The program has n identical MATs, each of which matches five-tuples and executes a routing action with respect to 2^{16} rules. Thus, the required capacity of switch resources equals to $n \times 2^{16}$ rules. We vary n from 2 to 10 and measure the resource usage as the ground true. Next, we use two different methods to merge the n identical MATs, respectively. First, we merge the n identical MATs into a compound MAT without resource sharing ("w/o Sharing"), i.e., the compound MAT retains the same capacity of $n \times 2^{16}$ rules. Second, we merge the n identical MATs but permit resource sharing ("w/ Sharing"). In this context, the compound MAT uses a shared memory region, which has a capacity of 2^{16} rules. We measure the resource usage after merging MATs and compare it with the ground true. Figure 2 indicates the resource usage is reduced only when resource sharing is activated ("w/ Sharing"), which justifies our observation.

Second, we observe that in most cases, it is sufficient to only address the *default-only MATs*, i.e., those contain no matching fields or only one rule. Even though default-only MATs are simple, merging them still brings many benefits. For example, multiple SDM tasks usually execute some identical operations (e.g., hashing) via default-only MATs. Thus, merging these default-only MATs can reduce the overall usage of switch resources. For other MATs (referred as *normal MATs*), note that their rules are installed at runtime after the program is actually deployed. Since it is infeasible to obtain MAT rules in the deployment, we do not take normal MATs into account. Further, merging normal MATs brings limited gain. For instance, advanced load balancers require a capacity of millions of MAT rules to record connections [17, 28]; network functions need large lookup tables for handling different flows [29]; in-network caches store a large amount of key-value items in MATs [2]. Such rules are typically application-specific so that there are limited opportunities to merge them with others.

According to our observations, we design SPEED to *only identify and merge default-only MATs*. We leave the merging of application-specific normal MATs in our future work.

Problem statement. Given several TDGs as input, SPEED needs to merge these TDGs into a compound TDG T_m. It aims at minimizing the number of stages occupied by T_m, which is determined by both MAT dependencies and the resource usage of T_m [7]. In particular, the MAT dependencies are supposed to be preserved during TDG merging so as to maintain the original program logics. Thus, to minimize the number of stages, the objective of SPEED is to maximize the number of merged MATs in T_m. Here, two MATs u, v defined in input TDGs can be merged into a merged MAT $w \in V_{T_m}$ if and only if the following constraints are simultaneously satisfied.
(1) *Equivalence*. u and v use the same matching fields and actions: $f_u^m = f_v^m, a_u = a_v$.
(2) *Default-only*. u and v are default-only MATs, i.e., the MATs that have no matching fields or only execute a default action: $f_u^m = f_v^m = \phi$, or $e_u = e_v = 1$.
(3) *Correctness*. Merging u and v into w will not interrupt the original MAT dependencies in E_{T_1} and E_{T_2}:

$$\begin{cases} D_{T_m}[a, u] = D_{T_1}[a, u], \forall u, a \in V_{T_1} \\ D_{T_m}[u, b] = D_{T_1}[u, b], \forall u, b \in V_{T_1} \\ D_{T_m}[c, v] = D_{T_2}[c, v], \forall v, c \in V_{T_2} \\ D_{T_m}[v, d] = D_{T_2}[v, d], \forall v, d \in V_{T_2} \end{cases}$$

(4) *Loop-free*. The compound TDG T_m is loop-free:

$$D_{T_m}[a, b] \cdot D_{T_m}[b, a] = 0, \forall a, b \in V_{T_m}$$

Solution overview. We merge the TDGs iteratively. Given n input TDGs, we pick up and merge two of them in each iteration. After $n - 1$ iterations, there is only one TDG left, which is the resulting merged TDG. Thus, the problem is reduced to merge two TDGs with maximum merged nodes. This problem can be solved by maximizing the length of LCS between topological orderings of the two TDGs (proved by Theorem 3.3 in [30]). We propose an LCS-based algorithm (Algorithm 1) that runs in polynomial time as follows.

Algorithm. SPEED takes two TDGs, T_1 and T_2, as input. It first obtains their topological orderings (line 2). Then it enumerates every pair of MATs defined in the two TDGs to

Algorithm 1 Merging TDGs.

Input: TDG $T_1 = (V_{T_1}, E_{T_1})$, TDG $T_2 = (V_{T_2}, E_{T_2})$
Output: compound TDG T_m
Variables: set s of pairs of mergeable MATs

1: **function** MERGE_TDG(T_1, T_2)
2:     *order1* $\leftarrow$ TopoSort(T_1), *order2* $\leftarrow$ TopoSort(T_2)
3:     $s \leftarrow$ Get_Mergeable_MAT_Pairs(V_{T_1}, V_{T_2})
4:     **if** $s == \phi$ **then**
5:         $T_m \leftarrow ParallelMerge(T_1, T_2)$
6:         **return** T_m
7:     **else**
8:         $lcs \leftarrow LCS(odr1, odr2, s)$
9:         $V_{T_m} \leftarrow lcs \cup (V_{T_1} - lcs) \cup (V_{T_2} - lcs)$
10:        $E_{T_m} \leftarrow$ Deduplicate($E_{T_1} \cup E_{T_2}, lcs$)
11:        **return** $T_m = (V_{T_m}, E_{T_m})$
12:    **end if**
13: **end function**

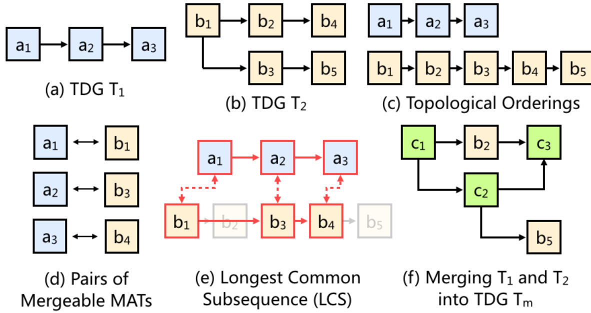


Fig. 3: Example of merging two TDGs.

obtain a set s that records the pairs of the MATs that can be merged, i.e., mergeable MATs (line 3). Specifically, for each pair (u, v) ($u \in V_{T_1}, v \in V_{T_2}$), it determines whether the two MATs satisfy the constraints as mentioned above. If so, u and v are mergeable, and (u, v) is added to s. Next, if s is empty, SPEED produces T_m by adding a pre-visiting MAT and parallelly connecting T_1 and T_2 to the pre-visiting MAT (lines 4-7). Otherwise, it inputs topological orderings and s into an LCS solver to obtain an LCS (line 8). The LCS solver uses dynamic programming for problem solving with respect to s. Note that topological orderings may correspond to multiple LCSes. We design the LCS solver to output the first LCS it solved since every LCS meets our objective. Finally, SPEED calculates E_{T_m} as the union of E_{T_1} and E_{T_2}. It also needs to deduplicate V_{T_m} based on lcs (line 10). This is because both E_{T_1} and E_{T_2} hold the same set of the dependencies of the MATs defined in lcs.

Example. SPEED first acquires the topological orderings of input TDGs, T_1 and T_2 (Figure 3(a)-(c)). It identifies three pairs of MATs that are mergeable (Figure 3(d)). Then it runs the LCS solver to obtain the LCS of topological orderings (Figure 3(e)). The LCS indicates that the maximum number of merged MATs is three. According to the LCS, SPEED merges V_{T_1} and V_{T_2} into V_{T_m}, and connects the MATs in V_{T_m} based on E_{T_1} and E_{T_2}. For instance, it merges $a_2 \in V_{T_1}$ and $b_3 \in V_{T_1}$ into $c_2 \in V_{T_m}$. It connects c_2 to b_5 based on the edge from b_3 to b_5, which originates from E_{T_2}.

V. TDG Placement on One Big Switch

In this section, we describe (1) how SPEED abstracts the substrate network into an OBS, and (2) how SPEED places the compound TDG T_m on the OBS with the objective of minimizing resource usage. Table I summarizes the notations.

TABLE I: Notation of symbols.

Symbol	Description
$\mathcal{N}$	Number of OBS stages.
V_Q	Set of virtual nodes in the linear OBS request.
E_Q	Set of virtual links in the linear OBS request.
V_G	Set of switches in the substrate network.
E_G	Set of physical links in the substrate network.
$P_{u,v}$	Set of paths between two programmable switches.
F_t	Throughput for packet processing.
F_l	Packet transmission latency.
p_s	Switch programmability.
b_s	Switch bandwidth.
l_s	Switch processing latency.
b_p	Path bandwidth.
l_p	Path transmission latency.
ρ	Number of stages per programmable switch.
$\mathcal{N}$	Number of stages in OBS.
M^S	Capacity of SRAM resources in a switch stage.
M^T	Capacity of TCAM resources in a switch stage.
R_S	SRAM resources used by an MAT.
R_T	TCAM resources used by an MAT.
S_S	Start stage ID of an MAT.
S_E	End stage ID of an MAT.
D	Variable denoting if two MATs should be separated.
x_i^u	Variable denoting if an MAT is placed on a stage.
$\alpha_{u_G}^{u_Q}$	Variable denoting if a virtual node is placed on a switch.
$\beta_p^{u_Q,v_Q}$	Variable denoting if a virtual link is placed on a path.

One big switch. SPEED first identifies both the number $\left|N^P\right|$ of programmable switches in the network and the number ρ of stages per programmable switch. Then it constructs a virtual programmable switch with $\mathcal{N} = \left|N^P\right| \times \rho$ stages as the OBS. Each OBS stage offers a region of memory resources for the placement of MATs. Here, memory resources include SRAM (denoted by M^S) and TCAM (denoted by M^T). The i-th OBS stage has a stage ID of i.

Problem statement. SPEED takes the compound TDG $T_m = (V_{T_m}, E_{T_m})$ and the OBS as input. It aims to place the MATs in V_{T_m} on the stages of OBS while respecting the MAT dependencies recorded in E_{T_m}. It produces a set of binary decision variables, $\{x_i^u\}, u \in V_{T_m}, i \in [1, \mathcal{N}]$: $x_i^u = 1$ if the MAT u is placed on the i-th stage of OBS; $x_i^u = 0$ otherwise.

Objective. SPEED aims to minimize the number of OBS stages occupied by T_m, which equals to minimizing the maximum stage ID λ occupied by the MATs in V_{T_m}:

$$\min \ \lambda \tag{1}$$

$$\lambda = \max_{\forall u \in V_{T_m}, \forall i \in [1, \mathcal{N}]} (x_i^u \cdot i) \tag{2}$$

NP hardness. However, the above problem is extremely hard. In fact, even a special case of this problem is NP-hard. In the special case, none of the MATs in V_{T_m} depends on other MATs, and each MAT can only be assigned to one stage. This special case equals the bin packing problem, which has been proved as NP-hard [31]. Thus, it follows that our problem is also NP-hard because our problem is more general.

Optimization framework. In essence, the above problem is a 0-1 integer programming problem since its output comprises

0-1 variables. Thus, SPEED offers an optimization framework that relaxes the NP-hard problem via linear programming relaxation to solve the problem in polynomial time. The framework encodes our objective and constraints, including both switch resource restrictions and MAT dependencies in E_{T_m}, and inputs them to a solver, Gurobi [32], to obtain the resource-optimal placement $\{x_j^{u_i}\}$. We illustrate how the framework encodes constraints as follows.

(1) *Switch resource restrictions*. The resources used by the MATs placed on a switch stage must not exceed the capacity of the stage. Therefore:

$$\sum_{u \in V_{T_m}} (x_i^u \cdot R_S(u)) \leq M^S, \ \forall i \in [1, \mathcal{N}] \tag{3}$$

$$\sum_{u \in V_{T_m}} (x_i^u \cdot R_T(u)) \leq M^T, \ \forall i \in [1, \mathcal{N}] \tag{4}$$

Here, the resource usage of each MAT can be calculated based on MAT properties. Given an MAT $u \in V_{T_m}$, its resource usage depends on its match type, number of rules e_u, and width of each rule w_u. If u only uses the match type *exact*, its usage of SRAM $R_S(u) = e_u \times w_u$ bits; otherwise, it occupies $R_T(u) = e_u \times w_u$ bits TCAM and does not use SRAM.

(2) *MAT dependencies*. The placement solution must respect the MAT dependencies indicated by E_{T_m}. Specifically, programmable switches prohibit the MATs with match/action dependencies from being placed on the same stages in order to maximize throughput [7]. Given two MATs $u, v \in V_{T_m}$, if v depends on u with a match/action dependency, the start stage occupied by v (denoted by $S_S(v)$) must occur after the end stage occupied by u (denoted by $S_E(u)$), i.e.

$$S_S(v) \geq S_E(u), \ \forall D(u, v) = 1 \tag{5}$$

Here, the boolean variable $D(u, v)$ indicates if u and v should be placed in different stages: $D(u, v) = 1$ if u and v should be placed in different stages; $D(u, v) = 0$ otherwise. For each edge $(u, v, t_{u,v}) \in E_T$: $D(u, v) = 1$ if $t_{u,v} \in \{\mathbb{M}, \mathbb{A}\}$; $D(u, v) = 0$ if $t_{u,v} \in \{\mathbb{R}, \mathbb{S}, \mathbb{N}\}$.

VI. One Big Switch Placement on Network

In this section, we first illustrate how SPEED places the OBS on the substrate network. Then we present the details of deploying the compound TDG on programmable switches with respect to placement results.

OBS request. SPEED first encodes the OBS into an OBS request $\mathcal{Q} = (V_Q, E_Q)$, where V_Q is the set of virtual nodes and E_Q is the set of virtual links. The i-th virtual node $u_Q \in V_Q$ corresponds to the sequence of ρ OBS stages, the IDs of which range from $(i-1) \cdot \rho$ to $i \cdot \rho$. For example, the left virtual node shown in the Step#3 of Figure 4 represents the sequence of the first two OBS stages. Each virtual link $(u_Q, v_Q) \in E_Q$ represents the link from the i-th virtual node u_Q to the $(i+1)$-th virtual node v_Q.

Substrate network. The substrate network can be represented by a directed graph $G = (V_G, E_G)$, where V_G and E_G denote the set of switches and that of physical links in the substrate network, respectively. Each switch $u_G \in V_G$ has three properties: (1) $p_s(u_G)$ indicates switch programmability. $p_s(u_G) = 1$ if u is programmable; $p_s(u_G) = 0$ otherwise; (2) $b_s(u_G)$ represents the switch bandwidth in Gbps; and (3) $l_s(u_G)$ denotes the processing latency in milliseconds. Each physical link $(u_G, v_G) \in E_G$ between two switches $u_G, v_G \in V_G$ is associated with two properties: (1) $b_l(u_G, v_G)$ represents the bandwidth of (u_G, v_G) in Gbps; and (2) $l_l(u_G, v_G)$ denotes the transmission latency of (u_G, v_G) in milliseconds.

In addition, for each pair of programmable switches denoted by (u_G, v_G) with $u_G, v_G \in V_G, p_s(u_G) \cdot p_s(v_G) = 1$, we search for the network paths from u_G to v_G and populate these paths to a set P_{u_G, v_G}. A network path is linear and may include several links and traditional switches. For each path $p \in P_{u_G, v_G}$, we measure its bandwidth b_p and latency l_p: b_p is the minimum bandwidth of the links and traditional switches resided in the path, while l_p is the sum of latency of the links and switches resided in the path. Such measurement can be easily done by means of a central SDN controller.

Problem statement. Given an OBS request $\mathcal{Q} = (V_Q, E_Q)$ and the substrate network $G = (V_G, E_G)$, SPEED needs to place $\mathcal{Q}$ on G via node mapping and link mapping. In *node mapping*, it needs to find a unique programmable switch in $u_G \in V_G$ for each virtual node $u_Q \in V_Q$. It produces a set of boolean variables, $\{\alpha_{u_G}^{u_Q}\}$, each of which indicates whether a virtual node u_Q is placed on a switch u_G: $\alpha_{u_G}^{u_Q} = 1$ if u_Q is placed on u_G; $\alpha_{u_G}^{u_Q} = 0$ otherwise. In *link mapping*, for the placement of virtual link $(u_Q, v_Q) \in E_Q$ between two virtual nodes $u_Q, v_Q \in V_Q$, it tries to find a path p between the two mapped programmable switches $u_G, v_G \in E_G$ from P_{u_G, v_G}. It outputs a set of boolean variables, $\{\beta_p^{u_Q, v_Q}\}$. If (u_Q, v_Q) is placed on p, $\beta_p^{u_Q, v_Q} = 1$; otherwise, $\beta_p^{u_Q, v_Q} = 0$. Note that our results can be easily extended to the one-to-many link mapping, where a single virtual link can be mapped onto several paths using the path-splitting technique [33].

Objective. SPEED aims to achieve maximum packet processing performance, i.e., maximizing throughput or minimizing per-packet processing latency, leading to a multi-objective optimization problem. For simplicity, SPEED transforms the problem into a single objective problem using the weighted sum method [34]. It defines the objective as a weighted sum:

$$\max \ (\omega \cdot F_t - (1 - \omega) \cdot F_l) \tag{6}$$

where F_t and F_l denote the throughput and packet transmission latency, respectively. Here, we ignore the in-switch processing latency because it is deterministic after TDG placement [7]. Moreover, ω is a user-configurable 0-1 weight: when $\omega = 0$, SPEED seeks for minimizing latency; when $\omega = 1$, it aims at maximizing throughput at runtime.

NP hardness. We show that even a special case of the above problem is NP-complete. In the special case, the mapping of nodes in V_Q has been provided, while SPEED only needs to place virtual links. However, even mapping links is hard. Previous studies [35, 36] have proved that the link mapping problem equals the unsplittable flow problem, which is NP-

complete. Since the special case is NP-complete, our original problem is NP-hard.

Optimization framework. SPEED relaxes the above problem in its optimization framework. Since the variables used by the above problem are restricted to be either zero or one, the framework utilizes linear programming relaxation and encodes network-wide constraints for problem solving. We elaborate network-wide constraints as follows.

(1) *Node assignment*. Each virtual node $u_Q \in V_Q$ can only be assigned to a programmable switch $u_G \in V_G$, i.e.

$$\sum_{\forall u_G \in V_G} \alpha_{u_G}^{u_Q} \cdot p_s(u_G) = 1,\ \forall u_Q \in V_Q \tag{7}$$

(2) *Link assignment*. Each virtual link $(u_Q, v_Q) \in V_Q$ can only be assigned to a path $p \in P_{u_G,v_G}$ between two mapped programmable switches $u_G, v_G \in V_G$, i.e.

$$\sum_{\forall p \in P_{u_G,v_G}} \alpha_{u_G}^{u_Q} \cdot \alpha_{v_G}^{v_Q} \cdot \beta_p^{u_Q,v_Q} = 1,\ \forall u_G, v_G \in V_G \tag{8}$$

(3) *Performance metrics*. The performance metrics include both throughput F_t and per-packet processing latency F_l. On the one hand, F_t is the minimum between the minimum bandwidth of mapped programmable switches (denoted by F_b^s) and the minimum bandwidth of mapped network paths (denoted by F_b^p), i.e.

$$F_t = \min\ (F_b^s, F_b^p) \tag{9}$$

Here, F_b^s is a constant implying the bandwidth capacity of a programmable switch, while F_b^p is the minimal bandwidth among selected paths, i.e.

$$F_b^p = \min\ \left(\alpha_{u_G}^{u_Q} \cdot \alpha_{v_G}^{v_Q} \cdot \sum_{\forall p \in P_{u_G,v_G}} \beta_p^{u_Q,v_Q} \cdot b_p\right) > 0, \quad \forall u_G, v_G \in V_G, \forall (u_Q, v_Q) \in E_Q \tag{10}$$

On the other hand, the latency F_l is calculated as the accumulation of latency of selected paths, i.e.

$$F_l = \sum_{\forall (u_Q,v_Q) \in E_Q} \sum_{\forall u_G,v_G \in V_G} \sum_{\forall p \in P_{u_G,v_G}} \alpha_{u_G}^{u_Q} \cdot \alpha_{v_G}^{v_Q} \cdot \beta_p^{u_Q,v_Q} \cdot l_p \tag{11}$$

Deployment. SPEED deploys the compound TDG T_m on the substrate network. It takes, as input, the placement results, including $\{x_i^u\}$ for the placement of T_m on the OBS, and $\left\{\alpha_{u_G}^{u_Q}\right\}$ for the placement of OBS on the substrate network. For each MAT $u \in V_{T_m}$, SPEED makes two decisions: (1) deploying u on the programmable switch $u_G \in V_G$; and (2) placing u on the k-th stage of u_G. For the first decision, SPEED first locates the virtual node $u_Q \in V_Q$ that includes the i-th OBS stage in which u resides (i.e., $x_i^u = 1$). Then it searches for the programmable switch $u_G \in V_G$, which sustains the virtual node u_Q (i.e., $\alpha_{u_G}^{u_Q} = 1$), as the target switch for the deployment of u. For the second decision, SPEED calculates k as follows.

$$k = i - \left\lfloor \frac{i-1}{\rho} \right\rfloor \cdot \rho$$

Next, for each programmable switch, SPEED inputs the MATs to be deployed on the switch and relevant MAT dependencies recorded in T_m to a target-specific compiler. The compiler generates corresponding switch configurations and installs these configurations on switches. After that, SPEED completes program deployment.

VII. Inter-Device Packet Scheduling

The placement decision may split a program on different programmable switches to fit resource limitations. However, this compromises the original MAT dependencies defined in the programs. First, the lack of packet scheduling between programmable switches leads to incomplete packet processing. Second, the MATs on a downstream switch cannot obtain the values of the fields modified by the MATs resided in upstream switches to continue packet processing.

To this end, SPEED provides inter-device packet scheduling comprising a four-step workflow: (1) It analyzes the input TDG T_m to identify the fields modified by MATs. It initials a specific packet header, i.e., SPEED header, to record these fields; (2) It inserts a specific routing module to the end of switch ASIC pipeline. The routing module is a default-only MAT that includes identified fields into the SPEED header. In the upstream switch, it *piggybacks* SPEED header on every incoming packet and delivers packets to the downstream switch; (3) It identifies the paths between the upstream switch and the downstream switch. It generates routing rules with respect to MAT dependencies and network paths, and populates these rules to the traditional switches resided in those paths; (4) The routing module of downstream switch parses packet headers and extracts fields from SPEED header. It delivers these fields to MATs for further processing and removes SPEED header from packets. In this way, SPEED guarantees the correctness of packet processing across different switches.

VIII. Case Study

We illustrate the workflow of SPEED via an example shown in Figure 4. In our example, there are two measurement programs to be simultaneously deployed. The two programs execute different sketch algorithms: one runs a Count-Min sketch [21] but the other runs an ElasticSketch [4]. Each program invokes three MATs. The first MAT hashes the source IP address of each packet. The second MAT maintains the sketch structure to record flow statistics. It uses the hashing result as the address to locate sketch counters, and updates corresponding counters accordingly. Finally, the third MAT matches the source IP address of the packet with its rules. For the matched packet, it determines which port to output the packet. Given the two programs as input, SPEED executes the following three steps to deploy them on the substrate network.

Step#1: program merging. SPEED first converts input programs into corresponding TDGs. Next, it identifies mergeable MATs among TDGs. Specifically, there are three pairs of

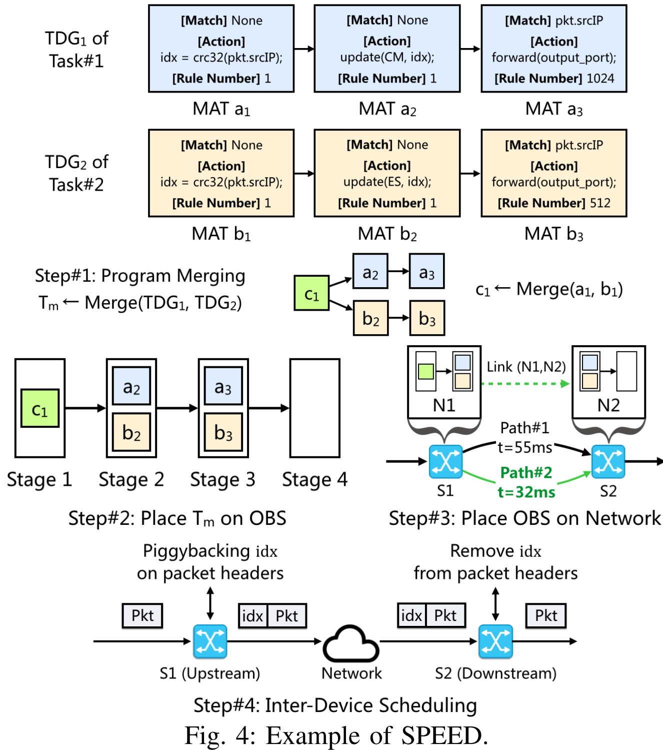


Fig. 4: Example of SPEED.

MATs that use the same matching fields, i.e., (a_1, b_1), (a_2, b_2), and (a_3, b_3). Here, only a_1, b_1 are mergeable MATs, while the remaining MATs are not. This is because: (1) a_1 and b_1 are default-only MATs and perform the same action; (2) a_2 and b_2 execute different actions (a_2 updates Count-Min sketch, while b_2 updates ElasticSketch); (3) although a_3 and b_3 perform the same action, they are not default-only MATs. Finally, SPEED calculates the LCS of mergeable MATs, which is a_1 in this example. According to the LCS, it merges the two TDGs into a compound TDG T_m.

Our example also illustrates why SPEED does not merge the two normal MATs a_3 and b_3. First, if SPEED merges a_3 and b_3 while preserving the total capacity of MAT rules (512+1024=1536 rules), the total resource usage will not be reduced as mentioned in §IV-B, such that merging a_3 and b_3 will not bring any benefits. Second, if SPEED merges a_3 and b_3 while allowing sharing resources among a_3 and b_3, the rule capacity of the merged MAT (denoted by c) must be less than the total capacity of a_3 and b_3, i.e., $c <$1536 rules. In this context, if administrators want to install more than c rules, they will encounter unexpected failures because the merged MAT does not provide sufficient space for MAT rules.

Step#2: TDG placement. SPEED abstracts the network into an OBS. Suppose that the number ρ of stages per programmable switch is two. SPEED consolidates the stages of all the programmable switches to create the OBS with four stages. It then places the MATs of T_m on the OBS with respect to MAT dependencies. Here, a strawman solution is to place each MAT on an individual stage, which occupies four stages. However, SPEED can find the resource-optimal placement that minimizes the number of occupied stages (i.e., three stages).

Step#3: OBS placement. SPEED partitions the OBS based on ρ, the number of stages per programmable switch. The partitioning generates two virtual nodes, N1 and N2. N1 corresponds to the first two stages in OBS, while N2 corresponds to the last two. There is also a virtual link $(N1, N2)$ from N1 to N2. The two virtual nodes and the virtual link are encoded as an OBS request. According to network-wide constraints, SPEED chooses programmable switches and network paths to place N1, N2, and $(N1, N2)$ with the objective of maximizing performance. In our example, N1 and N2 are placed in the two switches, S1 and S2, respectively. The virtual link $(N1, N2)$ is placed on Path#2 with minimal latency (i.e., 32 ms).

Step#4: Inter-device packet scheduling. SPEED installs routing modules to programmable switches, and populates routing rules to these modules. At runtime, the upstream switch (i.e., S1) looks up its routing rules and decides which next hop to forward packets. Before sending the packet out, it piggybacks the essential metadata fields (the index idx in this example) on packet headers. The downstream switch (i.e., S2) extracts the index from packet headers and feeds the index to its MATs for packet processing. It then removes the index from packet headers to recover the original packet structure.

In addition, SPEED can also enhance measurement accuracy by splitting a measurement program over multiple switches. For example, when using UnivMon [37] for heavy hitter detection, we can achieve an F1 score of 0.69 on a CAIDA 2018 trace [38] with 1.31M flows in one switch with 12 stages, while which SPEED will improve the F1 score to 1 by splitting the sketches over two switches of the same size.

IX. Implementation

Program merging. We implement the function of program merging in C++. It takes the data plane programs written in $P4_{14}$ or $P4_{16}$ as input. It translates input programs into high-level intermediate representations (HLIRs) [39] via the P4 compiler, P4C [40]. HLIR is a relatively straightforward translation of the input program that simplifies parsing. For each HLIR, SPEED generates the TDG by analyzing the properties of MATs defined in HLIR.

Program placement. We implement the optimization framework for TDG placement and OBS placement atop Gurobi [32]. SPEED also supports the incremental placement of data plane programs. Specifically, when new programs arrive, SPEED does not require any changes to existing deployed programs. Instead, it places new programs on the programmable switches whose resources are not exhausted yet. Doing so will minimize the interruption of the existing deployment. Also, during network upgrading, administrators can choose to jointly place the existing programs and new programs via SPEED so as to produce the resource-optimal deployment.

Runtime control. SPEED configures programmable switches via P4Runtime [41] and controls traditional switches via OpenFlow [42]. Also, it periodically measures the available bandwidth and latency of each path between programmable switches for making OBS placement decision.

TABLE II: Data plane programs used in our evaluation. "Dep." represents MAT dependencies.

Name	# MATs	# Dep.	Name	# MATs	# Dep.
CountMin (CM) [21]	6	3	Switch-V1 (V1) [44]	9	8
FlowRadar (FR) [45]	24	54	Switch-V2 (V2) [44]	10	10
UnivMon (UM) [37]	35	66	Switch-V3 (V3) [44]	20	20
SketchLearn (SL) [3]	35	66	Switch-V4 (V4) [44]	30	34
ElasticSketch (ES) [4]	21	66	Switch-V5 (V5) [44]	93	148

X. Evaluation

In this section, we perform experiments to evaluate SPEED. Our experimental results include:

- The program merging of SPEED reduces the number of occupied stages and runs faster than P4Visor (Exp#1).
- Compared to heuristics, the TDG placement of SPEED reduces the number of stages by up to 25% (Exp#2).
- Compared to heuristics, the OBS placement of SPEED achieves 14%∼59% latency reduction (Exp#3).
- For tested programs and topologies, SPEED completes its optimization in two seconds, which is acceptable for offline optimization (Exp#4).
- The inter-device packet scheduling of SPEED consumes less than 5% switch resources and has a small impact on packet processing performance (Exp#5).

A. Experimental Setup

Testbed. We build a testbed comprising a 32×100 Gbps Barefoot Tofino switches [13] and two servers. Each server has 2.30 GHz CPUs, 128 GB RAM, and a two-port 40 Gbps NIC. The testbed is organized as a sequential topologic. The Tofino switch is located in the middle of our testbed. The leftmost server and the rightmost server run a traffic sender and a traffic receiver based on MoonGen [43], respectively. These devices are connected via 40 Gbps links. Moreover, we employ another server that runs SPEED prototype and all the comparison methods as the control plane. This server is directly connected to the devices in our testbed. For each experiment, we present the average result after 100 runs.

Data plane programs. Table II presents the details of ten real data plane programs used in our evaluation. Here, the first five programs implement sketch algorithms and are used to measure flow statistics. Moreover, the remaining programs [44] are different versions (V1-V5) of switch.p4 [46], an open-source data plane program with 123 MATs. Specifically, V1 turns on l2 lookups and ACL; V2 realizes l2 lookups, ACL, and ECMP; V3 enables l2 lookups, ACL, ECMP, and link aggregation group (LAG); V4 supports l2 lookups, ACL, ECMP, LAG, and multicast; V5 retains most of the functions of switch.p4, but disables OpenFlow processing.

B. Heuristics for Comparison

We select two sets of heuristics that focus on TDG placement and OBS placement, respectively. These heuristics are used by representative research studies [7, 33, 47, 48]. Thus, we compare them with SPEED.

TDG placement. We implement two greedy heuristics, first fit by level (FFL) and first fit by level and size (FFLS) [7] for comparison. FFL and FFLS sort MATs and place MATs

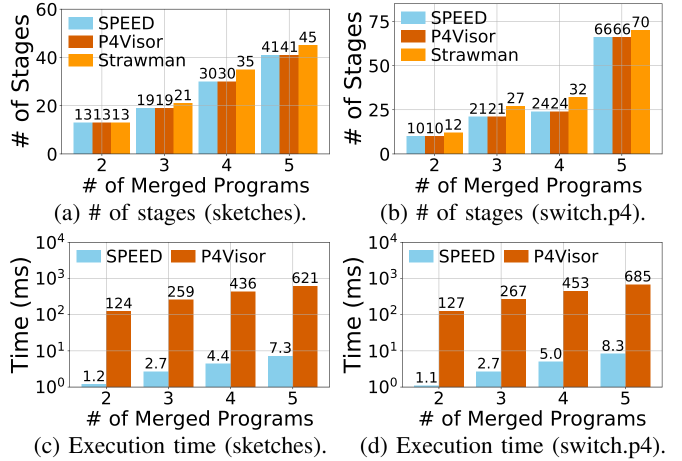


Fig. 5: (Exp#1) Impact of program merging.

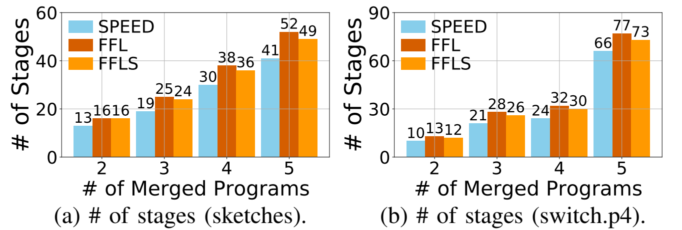


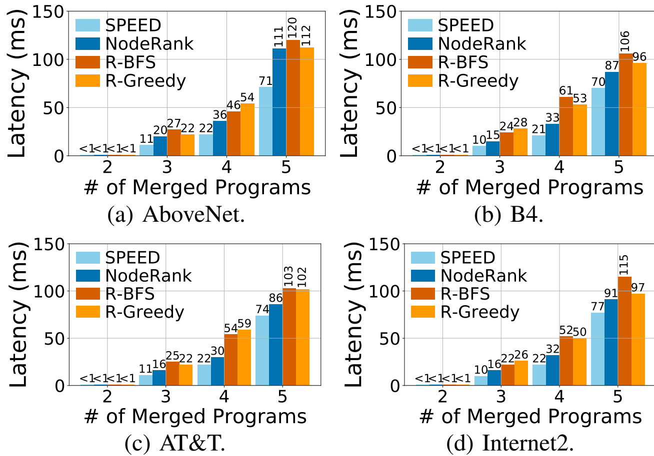
Fig. 6: (Exp#2) Effect of TDG placement.

on the OBS based on the order between them. Specifically, FFL sorts MATs based on MAT dependencies: the MAT with the highest *level* should be placed first. Here, the level of an MAT is defined as the number of dependencies in the longest path from the MAT to the end. Moreover, FFLS sorts MATs based on both dependencies and resource usage: the MAT with the highest level and the largest usage of SRAM and TCAM should be placed first.

OBS placement. We refer to virtual network embedding approaches [33, 47, 48] and design three heuristics for OBS placement: (1) *R-Greedy* randomly picks a programmable switch from the underlaying topologic. It uses a Greedy strategy that searches for an adjacent programmable switch with the minimal link latency. It repeats the procedure until finding enough switches for OBS placement; (2) *R-BFS* is similar to R-Greedy, but uses breadth-first search (BFS) to select programmable switches for OBS placement; (3) *NodeRank* first calculates a weight for every programmable switch based on the number of connections. For instance, a switch connected to three nodes will be given a weight of three. NodeRank picks the switch with the highest weight as root, and runs BFS to selects other switches for OBS placement.

C. Experimental Results

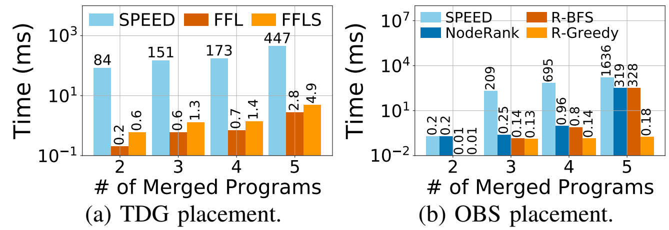
(Exp#1) Impact of program merging. We evaluate the impact of program merging on the usage of switch resources. We use two program merging methods, SPEED and P4Visor [10], to incrementally merge the programs in Table II, respectively. P4Visor is a virtualization platform that merges input programs into a compound one to concurrently run multiple programs on a single device. The difference between P4Visor



(a) AboveNet. (b) B4.

(c) AT&T. (d) Internet2.

Fig. 7: (Exp#3) Impact of OBS placement.



(a) TDG placement. (b) OBS placement.

Fig. 8: (Exp#4) Execution time of SPEED.

TABLE III: (Exp#5) Resource overhead of inter-device packet scheduling compared to the usage of switch.p4.

Name	PHV	SRAM	TCAM	VLIW	Stage
w/o SPEED	84.23%	29.58%	29.58%	36.72%	100%
w/ SPEED	89.41%	29.58%	29.58%	41.41%	100%
Overhead	4.91%	0%	0%	4.69%	0%

and SPEED is that P4Visor takes all types of MATs into account as merging TDGs. However, SPEED chooses to only merge default-only MATs as mentioned in §IV-B. Moreover, we also design a strawman method that directly combines input programs without program merging as the baseline. All the methods are running on the same environment.

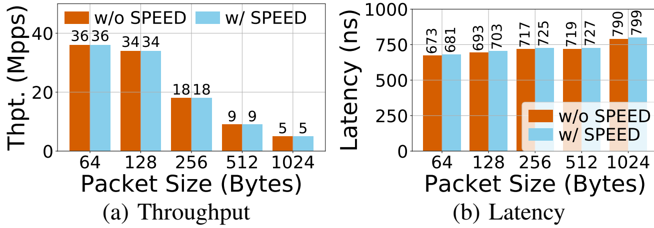
After program merging, we obtain eight merged TDGs, the first four of which are merged by the sketch-based programs, while the last four TDGs originate from the programs based on switch.p4. Then we leverage the TDG placement of SPEED to obtain the number of stages occupied by each merged TDG. We compare SPEED with P4Visor and the strawman method in two aspects: (1) the number of stages occupied by their merged TDGs, and (2) their execution time. Figure 5 indicates that (1) compared to the strawman method, SPEED achieves better placement that occupies a fewer number of stages; (2) the time of program merging in SPEED is less than 10 ms, while P4Visor spends hundreds of milliseconds. Note that our comparison is fair because both SPEED and P4Visor need to process all MATs (including default-only MATs and normal MATs). Thus, we evaluate the same functionality for the two systems. The only differences are their approaches of dealing with normal MATs: P4Visor does not classify the types of MATs, while SPEED identifies and discards the inefficient normal MATs to speed up the merging.

(Exp#2) Impact of TDG placement. We evaluate the impact of TDG placement on the usage of switch resources, in terms of the number of occupied stages. We use the same TDGs used by Exp#1 (merged by SPEED) as the input of TDG placement. We use SPEED, FFL, and FFLS to place these TDGs on an OBS with infinite stages, respectively. Figure 6 shows that SPEED consumes fewer stages and achieves high resource efficiency. Compared to FFL and FFLS, SPEED reduces the number of stages by up to 25% and 23.1%, respectively.

(Exp#3) Impact of OBS placement. We evaluate the impact of OBS placement in SPEED on the end-to-end performance of OBS. We select four real wide-area networks (WANs), including AboveNet [49], Google B4 [50], AT&T [51], and Internet2 [52]. We simulate these large-scale topologies in Mininet [53]. In each topologic, we randomly select half of devices as programmable switches, while the remaining devices are set to layer-3 routers. We set the latency of each link to be uniformly distributed from 10 ms to 50 ms via the APIs offered by Mininet, as suggested by the controller placement problem in WAN [54]. Next, we use the four TDGs evaluated in Figure 6(b) and generate four corresponding OBS requests. For each OBS request, we use SPEED and the three heuristics in §X-B to place the OBS on the substrate network, respectively. We set the objective to minimize the end-to-end latency. Figure 7 shows that the OBS placement of SPEED outperforms the three heuristics with 14%∼59% latency reduction. Note that the throughput results also follow the same trend, which is elided here.

(Exp#4) Execution time of SPEED. We measure the execution time of SPEED. We first invoke SPEED to place the four TDGs used by Exp#3 on the OBS, respectively, and generate corresponding OBS requests. Then we use SPEED to place the OBS requests on the Internet2 topologic. Also, we use the heuristics in §X-B to repeat the experiment, and compare their time with SPEED. As shown in Figure 8, the heuristics run faster since SPEED takes more time to optimize the placement. However, SPEED consumes at most 2.07s, which is acceptable for offline placement. Note that here we evaluate SPEED with a small number of programs. When the number of programs increases, the execution time of SPEED can reach several minutes (e.g., >5min when merging all the programs in Table II). We discuss this limitation in §XI.

(Exp#5) Overhead of inter-device packet scheduling. We measure the overheads incurred by inter-device packet scheduling. First, we quantify the resource overhead on a Barefoot switch. To do this, we deploy switch.p4 [46] on the switch and measure its resource usage as the baseline. Then we use SPEED to insert a routing module into switch.p4. The routing module initials a SPEED header and piggybacks all the metadata values used by the MATs of switch.p4 on incoming packets. As shown in Table III, SPEED consumes less than 5% switch resources. Second, we measure the performance overhead. We write a simple port-forwarding (i.e., directly forwarding packets from one port to another port) program as the baseline. Then we develop another port-forwarding program running with the routing module for switch.p4. We



(a) Throughput (b) Latency

Fig. 9: (Exp#5) Performance overhead of inter-device packet scheduling.

deploy the two programs on our testbed, respectively, and test their performance with 40 Gbps traffic. Figure 9 indicates that SPEED adds up to 10 ns to per-packet processing latency without dropping throughput, which is acceptable.

XI. Limitations

Assumptions. SPEED assumes that different programmable switches have the same number of stages (§V∼§VI). This assumption holds true in the production networks (e.g., Gateway [55], cloud systems [56], datacenter networks [2, 57]), which typically employ PISA-based switches [13, 1] that offer fixed number of stages. However, different types of programmable devices may contain different number of stages. To this end, we plan to extend SPEED to support program deployment on the network comprising various programmable devices.

Execution time of SPEED. The execution time of SPEED is long, which limits the scalability of SPEED. In particular, according to our preliminary evaluation, the execution time of SPEED can reach several minutes when there are a large number (>10) of programs to be deployed. In the future, we plan to develop efficient heuristics as the alternatives of our ILP-based placement to address this limitation.

Location-constrained deployment. SPEED jointly places the input programs on the network. However, some programs may impose the constraints that they have to be deployed on specific network locations. For instance, the program for Gateway should be placed on network boundaries [55]. We leave the exploration of this situation in our future work.

Overheads of inter-device packet scheduling. The inter-device packet scheduling consumes a portion of bandwidth resources due to packet piggybacking. However, according to our empirical study, it only requires a few additional bytes in packet headers. For example, it only uses additional 16 bytes in packet headers to deliver the metadata fields matched by the MATs of switch.p4 [46] between switches. Thus, compared to a general MTU of 1500 bytes, the bandwidth overhead of using a few additional bytes is relatively small (e.g., less than 1% when using 16 bytes). Also, since SPEED only occupies a few bytes in packet headers, the resulted overhead on packet processing performance is small (see Exp#5 in §X). Another limitation is that SPEED requires assistance from end-hosts to increase the MTU in order to enable inter-device packet scheduling. We plan to tackle this limitation in the future.

Impacts of network failures. Network failures such as link and device failures will affect the original packet processing of data plane programs. Currently, SPEED adopts a central controller to detect link and device failures. Once detecting a failure, the controller collects failure information and provisions them to administrators for further failure recovery. In the future, we will enhance the fault tolerance of SPEED by incorporating various state-of-the-art techniques. For example, we plan to leverage re-routing techniques (e.g, [58]) to search for backup paths after link failures.

XII. Related Work

Program merging. Many recent solutions [8, 9, 10, 11] leverage the techniques of program merging to reduce the usage of switch resources. However, these solutions only target a single device and ignore some realistic constraints imposed by programmable switches. Thus, their merged programs cannot be guaranteed to be deployable. Instead, SPEED carefully takes switch resource restrictions into account and deploys the merged program across multiple programmable switches, which ensures the practicality of program deployment. Moreover, P5 [59], MATReduce [60], and Precedence [61] are proposed to save switch resources at the code level. SPEED is orthogonal and complementary to these research efforts.

Program deployment. Recent works [7, 62, 63, 64] optimize the compilation of a data plane program on a single programmable device. SPEED takes a step further to investigate how to concurrently place several programs across multiple switches while achieving resource efficiency and high performance. In addition, prior solutions in other domains such as virtual network embedding [33, 65, 66, 48] optimize the placement of multiple requests on the substrate network. Different from them, SPEED addresses the specific challenges of achieving resource-efficient and high-performance program deployment in programmable networks.

XIII. Conclusion

We propose SPEED, a system that provides resource-efficient and high-performance program deployment. It first merges input programs into a compound TDG by reducing program redundancy. It then abstracts the substrate network as an OBS and places the compound TDG on the OBS while minimizing resource usage. Next, it offers the performance-optimal placement that maps the OBS to the substrate network. The experimental results indicate that SPEED achieves resource efficiency and high end-to-end performance.

Acknowledgement

We thank our shepherd Prof. Gabor Retvari, and the anonymous reviewers for their constructive comments. This work is supported by the National Key R&D Program of China (2018YFB1800601), the National Natural Science Foundation of China (61802365), FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (No. LZC0019), the Industrial Internet innovation and development project (No. TC190A449), the Key R&D Program of Zhejiang Province (2020C01021), and the Major Scientific Project of Zhejiang Lab (2018FD0ZX01).

REFERENCE

[1] 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," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[2] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *SOSP*. ACM, 2017, pp. 121–136.

[3] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference," in *SIGCOMM*. ACM, 2018, pp. 576–590.

[4] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *SIGCOMM*. ACM, 2018, pp. 561–575.

[5] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *SIGCOMM*. ACM, 2018, pp. 357–371.

[6] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *NSDI*, 2013, pp. 29–42.

[7] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches." in *NSDI*, 2015, pp. 103–115.

[8] D. Hancock and J. van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *CoNEXT*. ACM, 2016, pp. 35–49.

[9] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *ICCCN*. IEEE, 2017, pp. 1–9.

[10] P. Zheng, T. Benson, and C. Hu, "P4visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *CoNEXT*. ACM, 2018, pp. 98–111.

[11] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou, "P4sc: Towards high-performance service function chain implementation on the p4-capable device," in *IM*. IEEE, 2019, pp. 1–9.

[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[13] Barefoot Network. Barefoot Tofino. [Online]. Available: https://www.barefootnetworks.com/technology/#tofino

[14] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Scream: Sketch resource allocation for software-defined measurement," in *CoNEXT*. ACM, 2015, p. 14.

[15] ——, "Dream: dynamic resource allocation for software-defined measurement," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 419–430, 2015.

[16] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. Ports, and A. Panda, "Multitenancy for fast and programmable networks in the cloud," in *HotCloud*, 2020.

[17] 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 *SIGCOMM*. ACM, 2017, pp. 15–28.

[18] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *SIGCOMM*. ACM, 2016, pp. 1–14.

[19] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *SIGCOMM*. ACM, 2017, pp. 43–56.

[20] D. Popescu, N. Zilberman, and A. Moore, "Characterizing the impact of network latency on cloud-based applications performance," 2017.

[21] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[22] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM*, vol. 24, no. 4, pp. 664–675, 1977.

[23] Q. Huang and P. P. Lee, "Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams," in *INFOCOM*. IEEE, 2014, pp. 1420–1428.

[24] ——, "A hybrid local and distributed sketching design for accurate and scalable heavy key detection in network data streams," *Computer Networks*, vol. 91, pp. 298–315, 2015.

[25] L. Tang, Q. Huang, and P. P. Lee, "Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams," in *INFOCOM*. IEEE, 2019, pp. 2026–2034.

[26] ——, "Spreadsketch: Toward invertible and network-wide detection of superspreaders," in *INFOCOM*. IEEE, 2020, pp. 1608–1617.

[27] Q. Huang, H. Sun, P. P. Lee, W. Bai, F. Zhu, and Y. Bao, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *SIGCOMM*, 2020, pp. 404–421.

[28] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *SOSR*, 2016, pp. 1–12.

[29] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes," in *HotNet*, 2018.

[30] D. Saha, A. Samanta, and S. R. Sarangi, "Theoretical framework for eliminating redundancy in workflows," in *ICSC*. IEEE, 2009, pp. 41–48.

[31] E. C. man Jr, M. Garey, and D. Johnson, "Approximation algorithms for bin packing: A survey," *Approximation algorithms for NP-hard problems*, pp. 46–93, 1996.

[32] Gurobi optimizer. [Online]. Available: http://www.gurobi.com

[33] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking virtual network embedding: substrate support for path splitting and migration," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 17–29, 2008.

[34] R. T. Marler and J. S. Arora, "The weighted sum method for multi-objective optimization: new insights," *Structural and multidisciplinary optimization*, vol. 41, no. 6, pp. 853–862, 2010.

[35] J. M. Kleinberg, "Approximation algorithms for disjoint paths problems," Ph.D. dissertation, Massachusetts Institute of Technology, 1996.

[36] S. G. Kolliopoulos and C. Stein, "Improved approximation algorithms for unsplittable flow problems," in *Proceedings 38th Annual Symposium on Foundations of Computer Science*. IEEE, 1997, pp. 426–436.

[37] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *SIGCOMM*. ACM, 2016, pp. 101–114.

[38] The caida anonymized internet traces. [Online]. Available: http://www.caida.org/data/overview/

[39] P4 Language Consortium. P4-HLIR. [Online]. Available: https://github.com/p4lang/p4-hlir

[40] P4 Language Consortium. P4C. [Online]. Available: https://github.com/p4lang/p4c

[41] P4 Language Consortium. P4runtime: A control plane framework and tools for the p4 programming language. [Online]. Available: https://github.com/p4lang/pi

[42] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[43] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *IMC*. ACM, 2015, pp. 275–287.

[44] P4 language tests. [Online]. Available: https://github.com/jafingerhut/p4lang-tests/tree/master/v1.0.3

[45] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: a better netflow for data centers," in *NSDI*, 2016, pp. 311–324.

[46] P4 Language Consortium. switch.p4. [Online]. Available: https://github.com/p4lang/switch

[47] X. Cheng, S. Su, Z. Zhang, H. Wang, F. Yang, Y. Luo, and J. Wang, "Virtual network embedding through topology-aware node ranking," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 2, pp. 38–47, 2011.

[48] L. Gong, Y. Wen, Z. Zhu, and T. Lee, "Toward profit-seeking virtual network embedding algorithm via global resource capacity," in *INFOCOM*. IEEE, 2014, pp. 1–9.

[49] AboveNet. [Online]. Available: http://www.topology-zoo.org/maps/Abvt.jpg

[50] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[51] At&t north america topologic. [Online]. Available: http://www.topology-zoo.org/maps/AttMpls.jpg

[52] Internet2 topologic. [Online]. Available: https://www.internet2.edu/media/medialibrary/2018/07/16/Internet2-Network-Infrastructure-Topology-Layers-23.pdf

[53] Mininet. [Online]. Available: http://mininet.org/

[54] B. Heller, R. Sherwood, and N. McKeown, "The controller placement

problem," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 473–478, 2012.
[55] K. Qian, S. Ma, M. Miao, J. Lu, T. Zhang, P. Wang, C. Sun, and F. Ren, "Flexgate: High-performance heterogeneous gateway in data centers," in *APNet*, 2019, pp. 36–42.
[56] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *NSDI*, 2018, pp. 35–49.
[57] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *SIGCOMM*. ACM, 2017, pp. 85–98.
[58] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *NSDI*, 2019, pp. 161–176.
[59] A. Abhashkumar, J. Lee, J. Tourrilhes, S. Banerjee, W. Wu, J.-M. Kang, and A. Akella, "P5: Policy-driven optimization of p4 pipeline," in *SOSR*. ACM, 2017, pp. 136–142.
[60] X. Chen, D. Zhang, and H. Zhou, "Matreduce: Towards high-performance p4 pipeline by reducing duplicate match operations," in *GLOBECOM*. IEEE, 2018, pp. 1–7.
[61] C. Leet, S. Chen, K. Gao, and Y. R. Yang, "Precedence: Enabling compact program layout by table dependency resolution," in *SOSR*, 2019, pp. 1–7.
[62] 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 *SIGCOMM*, 2016, pp. 15–28.
[63] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy *et al.*, "drmt: Disaggregated programmable switching," in *SIGCOMM*. ACM, 2017, pp. 1–14.
[64] X. Gao, T. Kim, A. K. Varma, A. Sivaraman, and S. Narayana, "Autogenerating fast packet-processing code using program synthesis," in *HotNet*, 2019, pp. 150–160.
[65] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.
[66] M. Chowdhury, M. R. Rahman, and R. Boutaba, "Vineyard: Virtual network embedding algorithms with coordinated node and link mapping," *IEEE/ACM Transactions on networking*, vol. 20, no. 1, pp. 206–219, 2011.