FlxVRM: Enabling Online Configuring Memory via Virtualization on Programmable Data Plane.

Mimi Qian, Lin Cui, *Member, IEEE,* Fung Po Tso, *Senior Member, IEEE*, Yuhui Deng, *Member, IEEE,* Zhen Zhang, and Weijia Jia, *Fellow, IEEE*

***Abstract*—Programmable data plane (PDP) has emerged as a powerful platform for line-rate packet processing, utilizing on-chip register memory to execute stateful applications. Yet most existing efforts concentrate on static approaches for allocating register memory, necessitating switch restarting and service interruption. Despite the availability of research on sharing memory for concurrent applications, the rigid requirement of limiting memory sharing to the same pipeline stages hampers application flexibility and poses scalability challenges. To address this limitation, we present *FlxVRM*, *a flexible register memory virtualization layer* for data plane P4 programs which supports high-flexibility sharing of register memory for concurrent applications on PDP. *FlxVRM* enables memory allocation at any stage and location of the pipeline on PDP for each application at run time. To reduce resource usage during virtualization in the data plane pipeline, *FlxVRM* further merges different tables and actions with similar structures within P4 programs. Additionally, *FlxVRM* provides a compiler to generate data plane programs for virtualization as well as the control plane API configuration. A prototype of *FlxVRM* is implemented based on P4 hardware switches with Intel Tofino ASIC. Our experiment results show that *FlxVRM* significantly improves the allocatable memory space for applications by up to 50%, while reducing the resource of the table up to 68%.**

***Index Terms*—Register memory sharing, P4, Programmable data plane.**

I. Introduction

Programmable data planes (PDPs) [1], exemplified by Intel Tofino [2], have emerged as highly promising alternatives to conventional data planes. PDPs are composed of multiple stages organized in a pipeline, with each stage housing a set of stateful elements known as registers, which are implemented using SRAM. Registers enable efficient read and write operations on packets' diverse states at line rate, thereby influencing subsequent packet processing behavior. Leveraging the inherent capabilities of register memory, numerous stateful applications have been designed to realize various functionalities, including network measurement [3], [4], caching [5], [6], network management [7], machine learning [8], [9], and consensus [10], [11], offering Quality of Service (QoS) assurance to various users.

Mimi Qian, Lin Cui, Yuhui Deng and Zhen Zhang are with the Department of Computer Science, Jinan University, Guangzhou, China. Email: qianmimi001@gmail.com, tcuilin@jnu.edu.cn, tyhdeng@email.jnu.edu.cn, zzhang@jnu.edu.cn.

Fung Po Tso is with the Department of Computer Science, Loughborough University, UK. Email: p.tso@lboro.ac.uk.

Weijia Jia is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University (BNU Zhuhai), and BNBU, Zhuhai, China. Email: jiawj@bnu.edu.cn.

Corresponding author: Lin Cui

However, the register memory on programmable switches is intrinsically constrained by the hardware architecture, including limited memory size (e.g., only a few MB in Intel Tofino ASIC [2]) and hardwired implementation (e.g., fixed or predetermined register memory size). These limitations present significant challenges in efficiently accommodating multiple concurrent stateful applications on a programmable network [12]. Some existing methods allocate static register memory for multiple applications [13]–[15] (Table I). Nonetheless, the current implementation of applications on programmable switches is hardwired, necessitating recompiling and reloading when reconfiguring register memory for changing network requirements, such as adjusting memory size for real-time traffic. This disrupts the network services and applications on switches.

One promising approach is to leverage the programmability of PDP to achieve register memory sharing and dynamic allocation at run time for multiple concurrent applications. Yet the existing methods either reallocate register memory at the cost of compromising line rates by recirculation [13], [16], or incur inflexibility and scalability challenges due to the restricted sharing requirement. For instance, some methods [17] enable concurrent applications to share memory within the same regions and stages when virtualizing register memory for cocurrent applications. However, the existence of inconsistent table dependencies across multiple P4 programs imposes constraints on the range of stages that can be shared. These limitations undermine both the scalability of programmable networks and the ability to provide reliable resource provisioning for modern service paradigms like multi-tenant services and NFV chains that require dynamic allocation and strict QoS guarantees.

Given the constraints inherent to RMT (Re-configurable Match Tables) based programmable switches, such as single-stage memory access and restricted number of operations, achieving flexible (i.e., at any stages and locations) and dynamic (i.e., at run time) resource allocation in the data plane is challenging. Another major challenge in facilitating concurrent applications to share register memory across multiple stages is the large resource overhead on the data plane. This is caused by the repeated definitions of register operations in different shared stages, leading to a linear growth in data plane resources such as tables, actions, and sALUs (Stateful Arithmetic and Logic Unit) with the number of shared stages (Figure 1). Such resource overhead diminishes the available resources for deploying other essential data plane functions, such as congestion control and caching, which rely on stateful operations and sALU. Moreover, hardware constraints (such as

Table I: Comparison with existing solutions.

Methods	Dynamic allocation	High flexibility	Low resource overhead
Static bind [13]–[15]	✗	✗	✓
Virtualizing of PDP [19]–[21]	✓	✓	✗
Virtualized memory [4], [13], [16], [17]	✓	✗	✓
FlxVRM	✓	✓	✓

Intel Tofino's sALU can only pre-load four operations [13], [18]) further restrict the applications that can be accommodated and ultimately severely undermine scalability.

In this paper, we propose *FlxVRM*, a register memory virtualization layer that enables dynamic multi-stage memory sharing for concurrent applications while minimizing resource overhead. *FlxVRM* decouples virtual and physical register memory by employing a *mapping table* that is implemented using a match-action table in the data plane without adding register-level overhead. *This virtualization design allows FlxVRM to support on-demand memory reallocation across any pipeline stage/location for each application without service-disrupting switch restarts.* This is crucial for dynamic network services such as NFV chains, which require agility aligned with Service-Oriented Architecture (SOA).

In order to optimize the capacity of sharing and concurrency for application deployment within limited resources of the data plane, *FlxVRM* formulates the problem of merging and mapping programs with virtualization to target stages as a *stage mapping* problem. This approach involves conducting an analysis on the metrics of register-related operations, including the dependencies of control flow and utilization of resources within P4 programs. Based on this analysis, *FlxVRM* develops an efficient *stage mapping* algorithm that efficiently deploys applications with virtualization, maximizing capacity while respecting all constraints imposed by the RMT switch. *FlxVRM* further merges different tables and actions with similar structures within the P4 program, which significantly reduces resource consumption, thereby surpassing the bottleneck of existing works that only support a small number of concurrent applications [17].

To facilitate the deployment of P4 applications with FlxVRM, we have developed a compiler of FlxVRM that enables users to input multiple P4 programs and output a merged P4 program that can be run through a standard PDP compiler. In summary, we make the following contributions:

1) We propose *FlxVRM*, a runtime-flexible register virtualization layer that enables multi-stage register memory sharing and dynamic allocation with minimal resource overhead for network concurrent applications.
2) We formulate the stage mapping problem and propose a novel merging optimization technique to mitigate resource footprints resulting from virtualization. We also provide a compiler for enabling the seamless implementation of sharing register memory for data-plane P4 programs, and control plane API for user configuration.
3) We implement *FlxVRM* prototype with Intel Tofino ASIC [2]. Testbed experiments on a variety of applications demonstrate that *FlxVRM* outperforms existing methods by significantly reducing table consumption by up to 68%, simultaneously providing the allocatable memory space for applications by up to 50%.

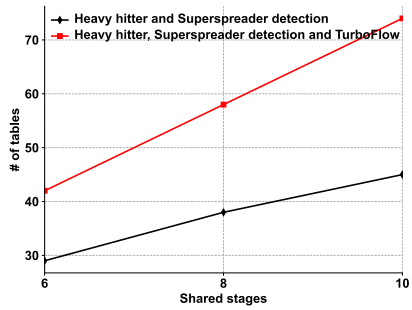


Figure 1: Comparison of table consumption for sharing different number of stages on programmable data plane between multiple applications.

The rest of paper is organized as follows. Section II discusses the background and motivations of *FlxVRM*. The overall design of *FlxVRM* is provided in Section III. Section IV evaluates *FlxVRM* on P4 hardware switches with Intel Tofino ASIC. The related works are presented in Section V. Section VI concludes this paper.

II. Background and Motivation

A. Background

The modern data center network incorporates a diverse array of network functions beyond simple packet forwarding including cache [5], load balancers [22], and network telemetry [3], [23]. Programmable switches (e.g., Tofino [2]) based on the RMT paradigm present an appealing alternative to commodity servers for implementing these stateful functions due to their high performance and programmability.

Programmable switches consist of several series of Match-Action Unit (MAU) stages arranged in data plane pipeline. The implementation of the application necessitates the utilization of one or several building blocks [24] on MAU stage. A sALU and corresponding SRAM can implement one of the building blocks. In RMT-based programmable switches, a sALU is bound to a fixed-size memory, collectively called a register [13]. Registers allow packets to read and write stateful states that involve register data, metadata and constant at line rate, subsequently determining the subsequent packet processing behavior. Registers are commonly structured as "register arrays". A register array is comprised of multiple register slots, all possessing an identical width, and can be accessed either through index-based direct mapping or hash-based mapping [17]. Stateful network applications using register memory are called reg-stateful applications, which enable many of the latest exciting researches [17], [25].

B. Motivations

Necessity of runtime dynamic allocation without recompiling. There are fundamental constraints of register memory on programmable switches. For example, only a few MB

in each stage within the Intel Tofino ASIC [2]. Additionally, previous studies have indicated that the utility of reg-stateful applications is influenced by both network traffic and the amount of allocated register memory. Consequently, the efficient allocation of the constrained register memory for multiple applications becomes imperative. One straightforward approach is to predict traffic and allocate the appropriate size of memory for application in advance. It may be easy to predict the number of flows in one hour, but hard at timescales of seconds or milliseconds [26].

Several existing works combine or merge multiple applications into one singular data plane program [14], [15], [19], [21] and allocate memory statically during compilation time. However, as network conditions fluctuate, applications need memory modifications, requiring system recompilation and reconfiguration to dynamically adapt. During this process, the switch must halt and restart, disrupting the function of all applications running on the switch, including fundamental tasks such as packet forwarding. Furthermore, this will also cause the switch to lose all state information, degrading network reliability and performance [1]. *To avoid interrupting applications, dynamically allocating register memory at run time without recompiling and reloading the program in the data plane is required.*

Multiple stages sharing *vs.* limitation of RMT Hardware. Several works [4], [9] allocate and release memory dynamically in a single stage for the multiplexing benefits by resubmitting packets back to the pipeline, leading to reduced throughput. Allowing current applications to share register memory in multiple stages represents a promising solution; nevertheless, achieving this objective poses significant challenges. First, register memory is distributed over multiple stages of pipeline, and each register can be accessed only from one stage. Second, the task implementations on programmable switches are hardwired at the compilation phase. The size and location of register memory can not be changed at run time. Most of the existing solutions propose the virtualization of PDP [19], [20] but overlook the practical hardware constraints (e.g., computational and memory resources). Thus, they are hard to implement in actual ASIC-based switch (e.g., Intel Tofino [2]).

Virtualizing register memory within multiple concurrent applications, like [17], requires repeated definitions for the same register operations at different stages. Hence, the resource usages, such as table, gateway, PHV, hash units[1] will grow linearly with the number of shared stages (as shown in Figure 1). Such resource overhead conflicts with the objective of maximizing concurrent applications within the data plane. In addition, due to the inherent hardware constraints [13], [18], each sALU can only support a limited set of operations and access limited fields from PHV containers (e.g., only four actions and a maximum of 2 fields in Tofino [2]). These restrictions severely affect the number of concurrent applications. *Therefore, being able to share register memory on multiple stages between concurrent applications with low resource consumption in data plane is highly desirable.*

[1]There are multiple types of hash resources (e.g., hash bits, hash calculation units, hash distribution units) in the data plane and usually the hash distribution units are the main bottleneck resources.

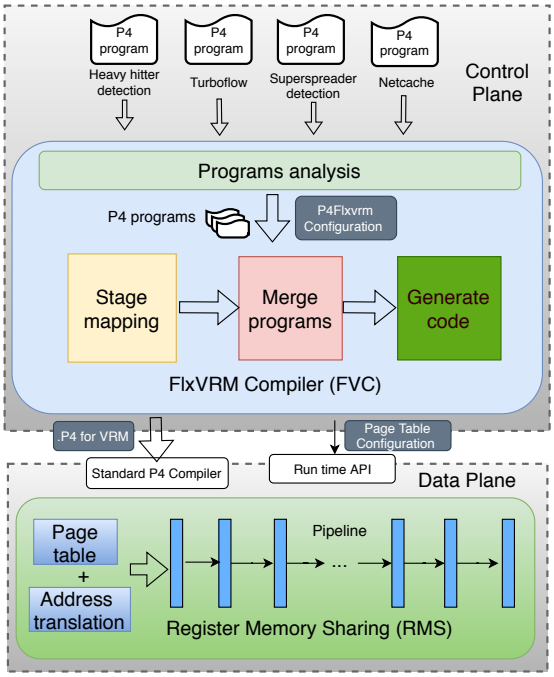


Figure 2: *FlxVRM* overview.

III. *FlxVRM* Design

A. Overview

FlxVRM is a flexible register memory virtualization layer for data-plane P4 programs that facilitates sharing and dynamic memory allocation for concurrent applications in a programmable network. Figure 2 shows the overview of *FlxVRM*. At a high-level, *FlxVRM* is composed of two components: the Register Memory Sharing (RMS) and the *FlxVRM* Compiler (FVC).

Register memory sharing (RMS). RMS runs on the data plane of PDP devices (e.g., the programming switch) and provides a virtual register memory abstraction to applications. The virtualized abstraction of RMS can hide the underlying implementation details of the physical register memory. When a packet accesses register memory, it only operates on a virtual address space, which will be translated to a physical address on the data plane by *mapping table* and *address translation* of RMS, rather than paying attention to complex physical addresses and operations.

***FlxVRM* compiler (FVC).** FVC runs on the control plane, which provides a compiler for developing concurrent applications to share register memory on data plane. FVC takes input P4 programs from operators and outputs a merged P4 program with virtualization in the control plane. The initial phase of FVC, called *program analysis*, first analyzes resource consumption, virtual memory size and control flow dependencies of each input P4 program, storing this information to a *P4Flxvrm* configuration file. The *stage mapping* then determines the deployment of the programs in the pipeline, aiming to maximize the number of concurrent stateful applications

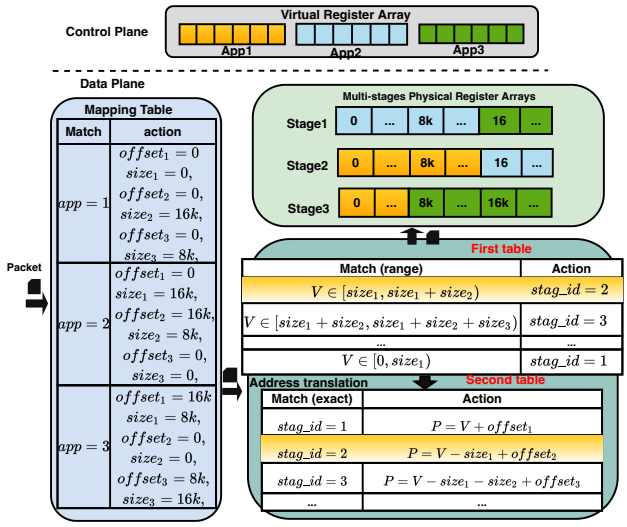


Figure 3: Register Memory Sharing.

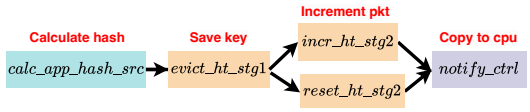


Figure 4: The control flow of heavy hitter detection, comprising two register operations: saving flow key (register *ht_stg1*) and increment packets (register *ht_stg2*).

and the range of shared stages based on this configuration, as well as the P4 programs.

To further reduce resource overhead with virtual register memory design, an optimization technique is proposed in *merge programs*, which involves merging tables and actions with similar structures in P4 programs. As a result of the merging process, the compiler creates an output merged P4 program, which can be compiled directly by a standard PDP compiler, and creates control plane APIs for dynamic memory configuration at run time.

B. Register Memory Sharing

The physical register memory in pipeline spans multiple stages, each accessed by tables and actions. To support configuration at run time, *FlxVRM* introduces virtualization of register memory in the data plane. Register memory is commonly abstracted as "register arrays". The register arrays of multiple stages are composed into *a large physical memory space* shared by multiple concurrent applications. *FlxVRM* maps the *virtual register array* of each application to different regions (e.g., variable sizes and locations) distributed in one or multiple stages of the large physical space. As shown in Figure 3, application 1 has a virtual array space with $24k$ slots, which are mapped to $[0, 16k)$ in stage 2 and $[0, 8k)$ in stage 3. *FlxVRM* uses the *mapping table* and *address translation* to perform the memory translation.

Design of memory translation. The mapping table is implemented by match-action table without introducing register memory overhead. As shown in Figure 3, the table matches application ID and identifies multiple regions with different locations (i.e., the start index) and sizes in each physical array (e.g., $offset_1$, $size_1$, $offset_2$, $size_2$, ..., $offset_h$, $size_h$), where h is the number of mapped stages (called *shared stages*). These parameters are configured by the control plane at run time during memory allocation. For example, the corresponding action parameters of application 1 are $(0, 0, 0, 16k, 0, 8k)$ (the occupied space in stage 1 is empty).

Let the size of a virtual register array for an application be N. A virtual address $V \in [0, N)$ is the index of the register slot in a virtual array. The address translation is implemented by two match-action tables shown in Figure 3. The first table matches the range of the V and identifies the stage k where the physical address is located (stage ID). The second table matches the stage ID and gets the physical address $P = V - (size_1 + size_2 + ... + size_{k-1}) + offset_k$. This translation process only requires addition and reduction operations, so it can be easily implemented on the pipeline of hardware switches.

Note that the design of mapping table also applies to the application that contains multiple register arrays. In such case, *FlxVRM* requires the register arrays of the same application to be mapped to the same location (i.e., offset in the mapping table) and size (i.e., size in the mapping table) in different regions. A key point of RMS is to guarantee that packets are processed in the correct sequence of Table Dependency Graph (TDG). To achieve this, *FlxVRM* requires that register arrays with dependencies in the application be mapped synchronously. For instance, Figure 4 illustrates a TDG of heavy hitters detection. If the mapped stages of *ht_stg1* encompasses stages 1 and 3, then the *ht_stg2* must be mapped synchronously to stages 2 and 4 (or greater). Otherwise, if *ht_stg2* is exclusively mapped to a single stage, such as stage 2, the table *incr_ht_stg2* or *reset_ht_stg2* could not be executed when the physical address of register *ht_stg1* is mapped to stage 3 for performing *evict_ht_stg1*. The reason is that the current programmable data plane does not support "backward" operations. Consequently, *FlxVRM* only needs to configure parameters for one of the multiple arrays within application, and the others will be automatically mapped to the same locations and sizes of the corresponding stages according to the dependencies in TDG. This process is executed automatically by the *stage mapping* of *FlxVRM* Compiler (details are discussed in Section III-C).

Some applications may contain multiple register operations that can be executed in parallel (i.e., no dependencies). In such case, *FlxVRM* combines multiple mapping tables to enable the address mapping for different register arrays, as each physical register array can only be accessed once by each packet due to the constraint of the hardware switch.

Benefits. The virtualization design of *FlxVRM* provides three notable enhancements in comparison to existing approaches [17], [20]. Firstly, *FlxVRM* enables dynamic allocation of register memory by the control plane, eliminating the static binding between applications and registers without the need for reloading or recompilation. Secondly, *FlxVRM* significantly enhances the sharing capability, including larger

physically accessible space for each application and more concurrent applications that can be accommodated. Since it allows each application to freely configure physical memory space at different stages of pipeline by the *mapping table*. This adaptability makes *FlxVRM* well-suited for most stateful applications without being constrained by the inconsistent table dependencies between applications. Thirdly, comparable to non-virtualized environments, *FlxVRM* enables fine-grained allocation at a 1-slot granularity level, achieving high space efficiency. Because *FlxVRM* uses exact and range match in match-action tables to implement address translation. This approach avoids computationally complex operations like division and modulo, which imposes constraints on the memory size, typically a power of 2 [27].

Implementation and challenges on RMT hardware. Packets in data plane are processed by a set of match tables and action processors in MAU stages for implementing network functions. One match-action table can only be referenced in a single stage for guaranteed line rates (Figure 5(a)). Figure 5(b) shows an example of the stateful operations on register memory, which presents the implementation of flow features collection in Turblflow [28] on the RMT-based switch. Specifically, a set of buckets in SRAM (e.g., register array) need to be allocated to store the packet features (e.g., packet size). We can input a flow key (e.g., five tuple) stored in PHV into the hash unit to get the address of a corresponding location in register array. Then, the sALU makes a conditional judgement and determines whether to perform add operation to update the bucket according to the hash address.

Despite the ability of the RMT hardware to facilitate user customization of stateful operations in multiple stages for virtualization, a straightforward implementation could easily lead to the underutilization of various hardware resources (e.g., table, hash). Since it always needs to instantiate register operations in all shared stages, enabling access to corresponding physical memory. However, the hardware resource in the data plane is very limited, for example, each stage only has 4 sALUs and each sALU can only preload 4 different operations in Tofino ASIC [2]. In the subsequent two subsections, we will first discuss how P4 programs with virtualization can be deployed in RMT switches, focusing on the efficient utilization of diverse hardware resources within the MAU stages, thereby maximizing the sharing and concurrency capacity of *FlxVRM*. Subsequently, we will present a merging optimization technique that significantly reduces the resource usage of virtual register memory in multiple stages.

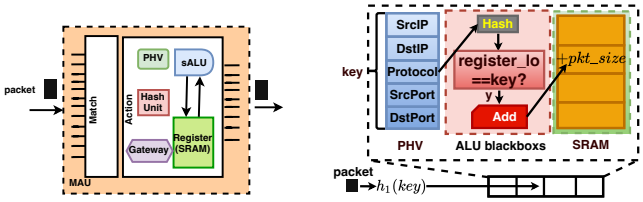
C. Stage Mapping Problem

The *FlxVRM* Compiler is used to implement the virtualization of register memory for concurrent reg-stateful applications. To maximize the capacity of RMS considering the constraints of the data plane, *FlxVRM* formulates the problem of mapping input P4 programs (table declarations and control flow) with virtualization to target stages within the pipeline as a *stage mapping* problem.

Objective function. We denote m as the number of shared stages that are available for virtual register memory within a pipeline. There are n stateful applications running in the network. A binary variable $v_{i,j}$ indicates whether the application i shares the register memory of stage j. Then, the stage mapping problem with two objectives is formulated as follows:

$$\max \sum_{i=1}^{n} L(\sum_{j=1}^{m} v_{i,j} \geq 2) \quad \textit{and} \quad \max \sum_{i=1}^{n} \sum_{j=1}^{m} v_{i,j}.$$

The objective is to maximize the capacity of concurrency and sharing of register memory virtualization, i.e., maximizing the number of concurrent applications and the range of shared stages, while respecting constraints of hardware and program dependency as described below.



(a) One match tables and action processors in pipeline.

(b) An example of the operations on register memory: collecting flow features in Turboflow.

Figure 5: Resource usage and implementation of register operation in RMT pipeline.

Table dependency. A P4 program can be abstracted by a TDG to investigate the dependencies among tables [29]. In TDG, tables are denoted as nodes, and their dependencies are represented by directed edges between two neighboring nodes. In TDG, if the result of table a affects the outcome of table b, we say that table b has a dependency on table a. Let $D_{a,b} = 1$ indicate the existence of this dependency between table b and table a, otherwise $D_{a,b} = 0$. The start and end stages for placing any given table l can be denoted as s_l and e_l, respectively (tables are allowed to span multiple stages in the pipeline). In order to ensure that the execution of table a is fully completed before table b begins, the end stage e_a of table a must precede to the start stage s_b of table b:

$$\forall D_{a,b} = 1 \colon e_a < s_b. \tag{1}$$

Shared stages constraint. A key point of register memory virtualization is to respect dependencies among nodes in TDG. Thus, the stage r_l of register array in reg-table[2] l that can be mapped in pipeline satisfies:

$$\forall l : e_l \leq r_l \leq m - p_l + e_l, \tag{2}$$

where m is the available number of shared stages in pipeline, and p_l denotes the position of table l within the longest path in TDG where it is located. For example, for a pipeline with 12 stages in Figure 6, the maximum range of register *SrcAddr* can be mapped to 1∼9. Exceeding this range, e.g., stage 10, will result in insufficient stage space to accommodate the subsequent tables (e.g., *tiIncrementPktCt* and *tiMcToCpu*).

We use the binary variable $x_l^j = 1$ to indicate the virtual register array in reg-table l is mapped to stage j, otherwise

[2]For convenience, we call the table in P4 program that operates on register memory as *reg-table*.

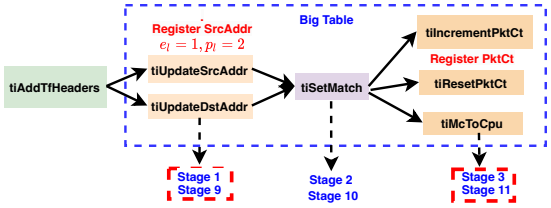


Figure 6: TDG of the *TurboFlow* [30] with 12 stages ($m = 12$) in pipeline.

$x_l^j = 0$. The table l_j is a *replica* of table l, enabling access to the physical register memory of the mapped stage j. It contains identical register operations to table l and must adhere to the dependencies specified in the TDG. Suppose table b has a dependency on table a (i.e., $D_{a,b} = 1$), then the end stage e_{a_j} of table a_j must precede to the start stage s_b of table b:

$$\forall x_a^j = 1 \colon e_{a_j} < s_b, j \in [e_a, m - p_a + e_a]. \tag{3}$$

Composition constraint. When multiple reg-tables in TDG exhibit dependencies, *FlxVRM* ensures synchronous mapping of the corresponding virtual register arrays, guaranteeing correct packet processing sequence in TDG. For instance, consider a scenario where the register *SrcAddr* in Figure 6 is mapped to stages 8 and 10, while the register *PktCt* is only mapped to stage 9. In such cases, the tables *tiIncrementPktCt* and *tiMcToCpu* cannot be executed if the physical address of the *SrcAddr* register is translated to stage 10 for performing table *tiUpdateSrcAddr* since data plane pipeline does not support "backward" operations.

To achieve this, we abstract all tables between two dependent reg-tables in TDG (the endpoints are inclusive) into a "Big Table" (shown in Figure 6). Suppose there are k tables inside a big table L and the binary variable $x_{L_k}^j$ indicates whether the register of k-th table in L is mapped to stage j. L is mapped to stage j that can only be deemed successful if every individual table within L is mapped successfully, hence *FlxVRM* requires:

$$\forall L : x_{L_1}^j \cdot x_{L_2}^{j+1} \cdot \ldots \cdot x_{L_k}^{j+k-1} = 1, j \in [0, m). \tag{4}$$

Capacity constraint in stage. All tables (including the replicas and big tables) must be assigned to a specific location within the pipeline. For each resource type, denoted by $t \in R = \{$*Logic Tables*, *Hash Units*, *sALU*, *SRAM*, *TCAM*, *Gateway*$\}$, the total memory assignment of table l in stage s, represented by $U_{s,l,t}$, should not exceed the physical capacity available in that stage, indicated by $R_{s,t}$. Consequently, the following condition must be satisfied:

$$\forall s, t : \sum_l U_{s,l,t} \leq R_{s,t}. \tag{5}$$

Capacity constraint in sALU. In RMT-based programmable switches, a singular register is assigned to one sALU. The sALU imposes constraints on the maximum number of pre-loaded stateful operations (e.g., actions), while limiting the number of fields that can be introduced for reserving states. Thus, in the case that multiple virtual register arrays share one physical register, the corresponding memory occupancies of table l in sALU u, denoted as $P_{t,l,u}$, should not surpass the physical capacity available in that sALU, represented by $G_{t,u}$, where t belongs to the resource set $G = \{$*PHV*, *action*$\}$:

$$\forall t, u : \sum_l P_{t,l,u} \leq G_{t,u}. \tag{6}$$

Complexity analysis of stage mapping. The stage mapping problem can be reduced from the 0-1 Knapsack Problem (KP), which has been proven to be NP-hard. In the following, we will show how to reduce a given 0-1 KP to the stage mapping problem.

Consider the following KP instance: for a set of items with different weights $Wt = \{w_1, w_2, ..., w_n\}$, and identical profits denoted as $Val = \{1, 1, ..., 1\}$, the knapsack capacity C, we construct an instance of the stage mapping problem. We consider n reg-stateful applications, where each application only contains one reg-table. And the replicas in these stateful applications can only be placed to a single stage, adhering to the Constraint (1) and (2). Consequently, the stage mapping problem becomes maximizing the number of replicas placed on a single stage[3] while satisfying with the resource constraints specified by (5) and (6). In this instance, each replica occupying the stage resource can be represented by the weight of an item. Each replica contributes to the objectives of *FlxVRM*, e.g., the number of concurrent stateful applications and the range of shared stages, corresponds to the profit value of 1. Additionally, a single stage resource is limited to a capacity of C.

It can be observed that the construction of the stage mapping instance from the KP can be accomplished in polynomial time, indicating that the KP problem, which is known to be NP-hard, can be efficiently reduced to the stage mapping problem. Therefore, it can be concluded that the stage mapping problem is also NP-hard.

D. Mapping Algorithm

We design an algorithm to solve the stage mapping problem, which consists of three major steps (as shown in Algorithm 1).

Firstly, *FlxVRM* merges TDGs from multiple P4 programs into one TDG, removing redundant nodes to reduce pipeline resource usage (line 1) via MergingRedundentNodesByLCS(). This process identifies TDGs with the largest Longest Common Subsequence (LCS) of mergeable MATs (e.g., identical match-action logic and dependencies) to minimize stage consumption while maintaining original logic and loop-free merged TDG, similar to prior studies [29], [31].

Secondly, *FlxVRM* utilizes a three-part partition strategy to process the TDG of each input P4 program based on the merged TDG. The first part includes tables executed prior to the reg-table of the P4 program, including the reg-table itself. The second part comprises tables that can be executed alongside the reg-table in parallel. The third part consists of tables executed after the reg-table (lines 2 $\sim$ 4). *FlxVRM* assigns the tables from the first and second parts to memory blocks

[3]We assume that all applications can be deployed within the data plane in the non-virtualized case.

Algorithm 1 Stage mapping algorithm

Input: A set of TDGs G_1, G_2, ..., G_n for input P4 programs.

1: $G \leftarrow$ *MergingRedundentNodesByLCS*$(G_1, G_2, .., G_n)$
2: **for** $i = 1, 2, ..., n$ **do**
3: $t_i \leftarrow$ Obtain the reg-table of G_i
4: $G_{i,1}, G_{i,2}, G_{i,3} \leftarrow$ *Partition*(G_i, t_i, G)
5: *PlaceTableFromFirstStageForward*$(G_{i,1}, G_{i,2})$
6: *PlaceTableFromLastStageBackward*$(G_{i,3})$
7: $S_i \leftarrow$*Obtain stages where register array in reg-table* t_i *can be mapped according to the Constraint* (2)
8: **for each** $j \in S_i$ **do**
9: R_j.*add*(*PreGenerateReplica*(t_i, j))
10: **for** $j = 1, 2, ..., m$ **do**
11: *SortOnMtrics*(R_j)
12: *Place the replicas* R_j *in order by a multi-metric heuristics*

in pipeline stages sequentially. The tables (including reg-tables) from all programs are placed in an order following the principle of previous approaches [32] that greedily minimizes the number of stages used (e.g., dependency chain length and resource type priority). If a table cannot fit in the first pipeline stage, *FlxVRM* searches for available memory blocks in the same stage or subsequent stages to accommodate it (line 5). For the third part, the placement strategy remains consistent, but *FlxVRM* searches for unoccupied memory blocks on the pipeline stages in a reverse order (line 6). In this step, the same tables are not placed repeatedly in different P4 programs.

Thirdly, *FlxVRM* identifies the stages where virtual registers of P4 programs can be mapped, e.g., determining the stages for placing the replicas of all reg-tables in the merged TDG (line 7). *FlxVRM* first pre-generates replicas for all reg-tables, including big tables, at each shareable stage determined by the Constraint (2) (lines 8∼9). These replicas are sorted within each stage (discussed in **Ordering tables** of this section). Then, according to the sorting of replicas, *FlxVRM* employs a multi-metric heuristic method (discussed in **Multi-metric heuristic**) to determine if each replica should actually be generated in its corresponding stage, until all replicas are successfully placed or resources are exhausted (lines 10∼12). No duplicate replicas are generated in different input programs.

Ordering tables. The effectiveness of the mapping process is significantly influenced by the sort order of the replicas. There are several sorting metrics that hold considerable importance in *FlxVRM*.

Virtual register memory: Applications with larger register memory requirements, i.e., the large slots of the virtual register array, should be prioritized to obtain larger physical space. Otherwise, there may be insufficient space for applications with large register memory requirements.

The length of table: We define the longest dependency chain of a big table as its length. The length of other tables in TDG, e.g., non-big tables, is set to 1. The sharing capacity of a table in the pipeline is, typically, inversely proportional to its length. Because a shorter table allows for a larger range of shared stages according to the Constraint (4), thereby the table of shorter length should be prioritized.

Resource consumption: There exist two distinct types of resource consumption that need to be considered: resources in shared sALU, such as actions, PHV, and stage-level resource consumption, e.g., gateway, tables with wide match or action words, etc. The replicas with less resource consumption should be placed first, because it leaves more room for more applications. Since the stage-level resources are generally more abundant than the resource of sALU in the RMT switch, the latter has a higher priority.

Multi-metric heuristics. The *FlxVRM* compiler provides program analysis component for analyzing the above key sort metrics for placing replicas in available pipeline stages. Building on program analysis, *FlxVRM* proposes a multi-metric heuristics algorithm that incorporates resource usage and the length of table, where the replicas with shorter lengths of dependency chains in tables with lower resource consumption are assigned earlier. If both metrics are the same, *FlxVRM* chooses the tables that have larger virtual register memory.

The *FlxVRM* compiler allows operators to customize the priority order for application placement, ensuring higher priority tables are placed before lower priority tables. This prevents starvation and adapts to changing demands, giving priority to applications that require more register memory, such as those with higher accuracy requirements.

E. Merging Optimization

One challenge introduced by virtualizing register memory is that it always needs to generate and deploy the replicas across multiple stages in order to enable access to the corresponding physical register memory. However, the available resource, such as hash and sALU, in the data plane's MAU stages is limited. To reduce the resource overhead in virtual register memory design, *FlxVRM* further proposes a novel optimization merging technique. This optimization focuses on merging different tables (including replicas) and actions with similar structures of the P4 programs with virtualization. *FlxVRM* introduces two methods: *TaA Merging* (merging table and action) and *TnA Merging* (merging table but not action).

TaA Merging. There are two cases for *TaA Merging*. The first case focuses on merging multiple tables and actions with identical register operations by making their names, including tables, actions and PHV, consistent in different programs. For instance, as shown in Figure 7, the *Superspreader* detection and *Turboflow* applications both store the *ipv4.srcAddr* from the packet header into a shared register. *FlxVRM* merges actions A and C by ensuring name consistency of both actions and PHV fields in "output_dst" of the sALU. One problem of this process is that the application deployment cannot dynamically set the key for arriving packets of different applications. Because the hash units in RMT hardware cannot change the input PHV fields to other fields at run time. Fortunately, an extra PHV filed named "Dynamic Key" [13] can be used. When a packet arrives, and according to the application to be performed, *FlxVRM* copies the original header fields to this "Dynamic Key". Then the hash unit calculates the dynamic key to get the corresponding memory address.

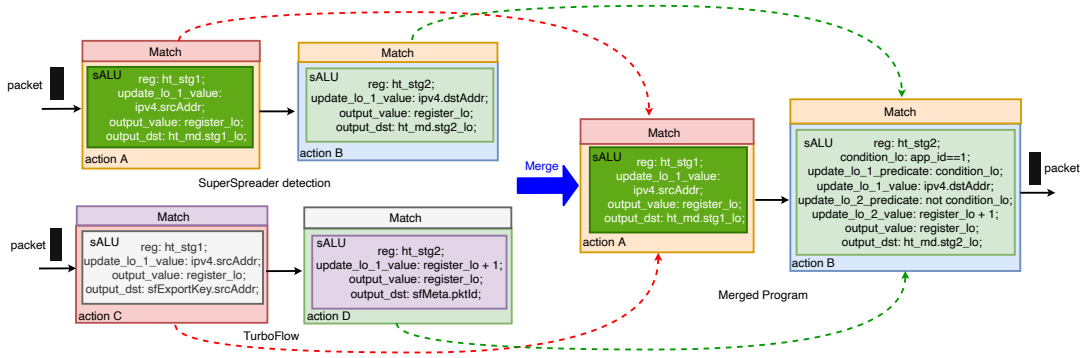


Figure 7: An example of *TaA Merging* on two applications: Superspreader detection and Turboflow.

In another case of *TaA Merging*, *FlxVRM* merges multiple tables and actions that employ distinct register operations. It capitalizes on the fact that the computational capabilities of sALU often surpass the demands of application algorithms. By using the condition judgment in sALU, *FlxVRM* merges diverse register operations to a single interface (action), thereby reducing the actions of interfaces consumed in sALU, subsequently decreasing the usage of both sALU and tables. The condition judgment is employed to determine the appropriate application to execute. For example, *Turboflow* and *Superspreader* detection perform add ("+1") and assignment (equal to *ipv4.dstAddr*) operations in register *ht_stg2*, respectively (actions B and D). As shown in Figure 7, *FlxVRM* combines these operations into a single blackbox through the condition judgment of app_id (*condition_lo* field).

TnA Merging. The *TnA Merging* technique in *FlxVRM* merges multiple tables but not actions, each performing different register operations on the same sALU, by introducing an exact match of the *app_id* field. The *app_id* is utilized to select different actions for various applications in the merged table. This approach can also be applied to tables in TDG that do not involve register operations such as field modification, shifting, hash operations, addition, and subtraction. Currently, *FlxVRM* only supports merging tables with exact matches or no matching fields. In future work, we will explore other match types like ternary and range. The experiment in Section IV validates the effectiveness of *FlxVRM*'s merging optimization in mitigating hardware resource constraints.

IV. Evaluation

We have implemented a prototype test-bed of *FlxVRM* on P4 hardware switches (equipped with Intel Tofino ASIC).

A. Experiment setup

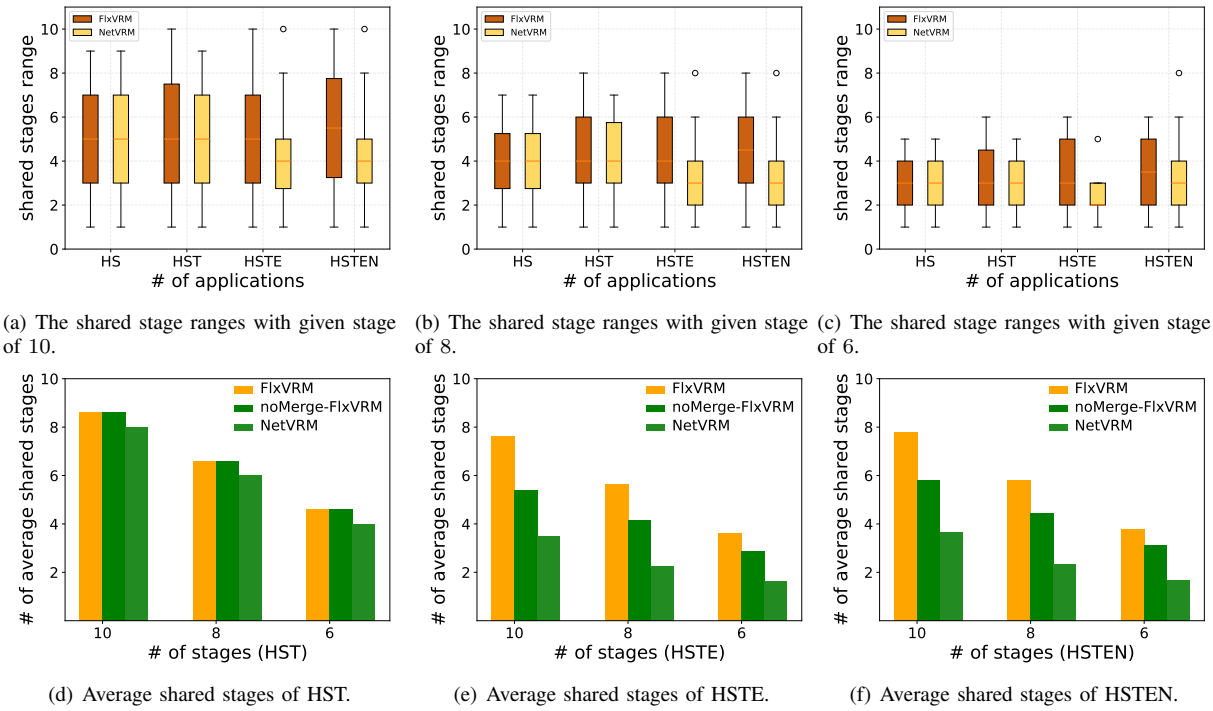
The test-bed for experiments consists of two servers (equipped with 8 Intel Core i7-4771 CPU @ 3.50GHz), each configured with 10/40Gbps NIC (i.e., offering maximum throughputs of 10 and 40 Gbps). A P4 hardware switch (using Intel Tofino ASIC) connects the servers, with all links utilizing 10/40Gbps fibers. The default deployment on Tofino switch leverages the $P4_{14}$, while the *FlxVRM* compiler is implemented in the Python language.

FlxVRM supports a wide range of network applications. We have implemented five common applications in our evaluations, including *Heavy hitter detection* (H) [30], *Superspreader detection* (S) [30], *Turboflow* (T) [28], *Elastic sketch* (E) [26] and *NetCache* (N) [5]. HS in our evaluation refers to both applications H and S are deployed, where flow keys (source or destination IP) and flow size are stored on the data plane. Upon hash collisions, this flow information is transferred to the control plane. HST expands the combination by including the T application, maintaining the same mechanism as H and S for handling hash collisions, but using different flow keys (5-tuple) and features (packet size). HSTE adds the E application, which handles elephant and mouse flows using a hash table and CM sketch respectively. Lastly, HSTEN incorporates the N application, which retains frequently accessed key-value pairs on the data plane for quick response.

In addition to *FlxVRM*, we also implemented NetVRM [17] for comparison. NetVRM implements virtual register memory in multiple stages by imposing all applications to share memory regions with consistent offset and size within the identical pipeline stages. We also compared with an alternative version of *FlxVRM* called *noMerge-FlxVRM*, which virtualizes P4 programs without employing merge optimization.

B. Performance of sharing register memory

Comparison of shared stages. To evaluate the sharing capability of *FlxVRM*, we conducted an experiment wherein varying applications (HS, HST, HSTE, and HSTEN) are deployed with different available number of stages (6, 8, and 10) for virtualization. The *shared stage range* represents the range of stages that can be mapped by the virtual register arrays of the deployed applications. A wider range of shared stages allows for larger memory space allocation for applications. Results in Figures 8(a), 8(b) and 8(c) demonstrate that *FlxVRM* enables applications to share register memory within a wider range of stages, enhancing up to 50% compared to NetVRM. Specifically, when *Heavy hitter detection* (H) and *Superspreader detection* (S) are deployed in the data plane, *FlxVRM* and NetVRM exhibit an identical shared stage range. This is because the TDG structure and location of the reg-tables in TDG are similar for both applications, leading to the same mapped stages of reg-table in both methods. As the number of applications increases, *FlxVRM* outperforms NetVRM. The



(a) The shared stage ranges with given stage of 10.

(b) The shared stage ranges with given stage of 8.

(c) The shared stage ranges with given stage of 6.

(d) Average shared stages of HST.

(e) Average shared stages of HSTE.

(f) Average shared stages of HSTEN.

Figure 8: Comparison of stages optimization among *FlxVRM*, *noMerge-FlxVRM* and NetVRM.

reason is that *FlxVRM* allows each application to share register memory in different stages based on their TDGs, unaffected by the varied table dependencies of applications. In contrast, NetVRM requires all applications to share memory within the minimum range of stages identified across applications.

The results in Figures 8(d), 8(e) and 8(f) reveal that the *noMerge-FlxVRM* exhibits a higher capacity than NetVRM but is inferior to the merged one (*FlxVRM*). The reason is that the flexible sharing of *FlxVRM* allows applications to be shared within larger stage ranges, but the high resource consumption in *noMerge-FlxVRM* and NetVRM leads to fewer replicas that can be placed for applications.

Comparison of tables. The results in Figure 9 demonstrate that *FlxVRM* exhibits considerably fewer table consumptions compared to *noMerge-FlxVRM* and NetVRM, with reductions of up to 75% and 68%, respectively. This is because the table consumption of *noMerge-FlxVRM* and NetVRM grows linearly with the number of shared stages, whereas *FlxVRM* effectively mitigates this issue through its merging optimization. *noMerge-FlxVRM* incurs higher table counts than NetVRM due to NetVRM's narrowed shared region (reducing table instances), whereas *noMerge-FlxVRM* covers a broader stage range (Figure 8(d)).

C. Performance of deployment

Comparison of resource usage. We evaluated hardware resource usage for *FlxVRM*, including stateful interfaces (which are used to read or write registers), meter-ALUs, hash dist unit, gateway and tables by implementing various applications in Tofino's switch with given stages of 6, 8, and 10. The results are presented in Tables II, III and IV. When the number of shared stages remains constant, it is observed that *FlxVRM* has lower consumption in most of resources compared to NetVRM and *noMerge-FlxVRM*. As the number of applications increases, *FlxVRM* reduces resource utilization in stateful interfaces by up to 60% and decreases table size by up to 50%. This advantage can be attributed to the effective mapping algorithm and merging optimization techniques in *FlxVRM*. These findings indicate that *FlxVRM* can accommodate a larger number of concurrent applications while maintaining optimal resource utilization.

Table II: Resource consumption of *FlxVRM* and NetVRM on Tofino[5].

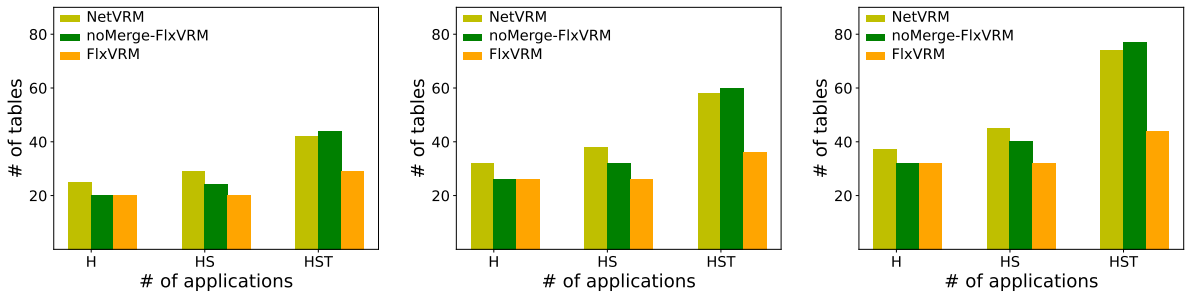
Per stage resource	*FlxVRM* (HS)			NetVRM (HS)		
	6(stg)	8(stg)	10(stg)	6(stg)	8(stg)	10(stg)
Stateful interfaces	37.5%	37.5%	37.5%	50%	50%	50%
Meter-ALUs	16.67%	26%	33.33%	20.83%	29.17%	37.5%
Gateway	15.62%	21.88%	28.12%	21.88%	29.17%	36.46%
Hash Dist Unit[6]	8.33%	11.11%	13.89%	9.72%	12.5%	15.28%
Pipeline resources						
Tables	20	26	32	29	38	45
Average shared stages	4	6	8	4	6	8

Execution time. We also evaluated the execution time of *FlxVRM* by conducting a comparative analysis between *FlxVRM* and the exhaustive baseline method (which evaluates all possible merging orders and placement strategies). We augment the experimental setup with six additional real-world

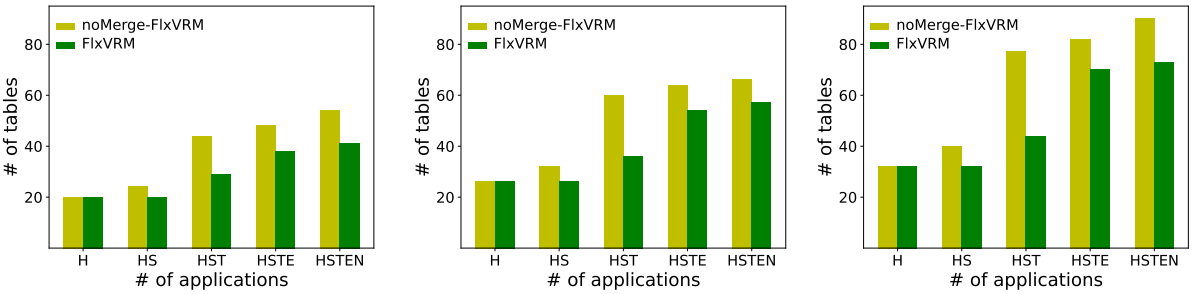
[4]NetVRM results for HSTE/HSTEN are omitted because its shared memory constraint causes resource exhaustion and deployment failures.

[5]NetVRM exhibits a notable reduction in the number of stages that can be shared, when the applications of HST, HSTE and HSTEN are deployed, thereby rendering comparisons of resource consumption infeasible.

[6]Current programmable switches use a hash (distribution) unit to access SRAM for sALU.



(a) Tables consumption in three methods with given stages of 6. (b) Tables consumption in three methods with given stages of 8. (c) Tables consumption in three methods with given stages of 10.



(d) Tables consumption in two methods with given stages of 6. (e) Tables consumption in two methods with given stages of 8. (f) Tables consumption in two methods with given stages of 10.

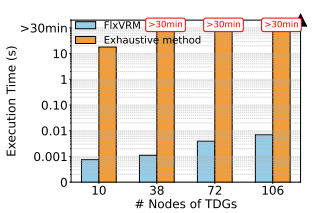
Figure 9: Comparison of tables consumption in different stages among *FlxVRM*, *noMerge-FlxVRM* and NetVRM.[4]

Table III: Resource consumption of *FlxVRM* and *noMerge-FlxVRM* on Tofino (HS and HST).

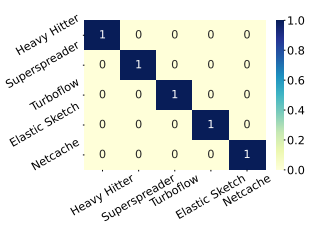
Per stage resource	HS		HST	
	FlxVRM	*noMerge-FlxVRM*	*FlxVRM*	*noMerge-FlxVRM*
Stateful interfaces	37.5%	50%	33.25%	66.7%
Meter-ALUs	16.67%	16.67%	31.25%	31.25%
Gateway	15.62%	17.71%	20.83%	35.42%
Hash Dist Unit	8.33%	8.33%	18.06%	27.78%
Pipeline resources				
Tables	20	24	29	44
Average shared stages	4	4	4.6	4.6

Table IV: Resource consumption of *FlxVRM* and *noMerge-FlxVRM* on Tofino (HSTE and HSTEN).

Per stage resource	HSTE		HSTEN	
	FlxVRM	*noMerge-FlxVRM*	*FlxVRM*	*noMerge-FlxVRM*
Stateful interfaces	37.5%	75%	43.75%	81.25%
Meter-ALUs	35.42%	33.33%	37.5%	39.68%
Gateway	26.04%	35.42%	28.12%	39.06%
Hash Dist Unit	23.61%	29.17%	26.39%	34.72%
Pipeline resources				
Tables	38	48	41	54
Average shared stages	4	3	4	3

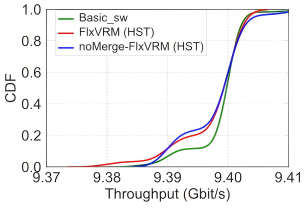


(a) Execution time.

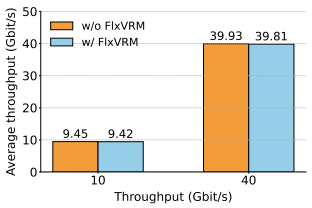


(b) Correctness.

Figure 10: Correctness and Scalability verification.



(a) Throughput under 10G traffic.



(b) 10Gbps *vs* 40Gbps.

Figure 11: Throughput on P4 hardware switches (Tofino ASIC).

programs that form large-scale TDGs. Each testing TDG has 5 $\sim$ 40 nodes and 3 $\sim$50 edges. The experimental result, as shown in Figure 10(a), demonstrates that *FlxVRM* completes the mapping process within 0.01 second, even when handling large-scale merged TDGs containing up to 100+ nodes, while the baseline's exponential time complexity exceeds 30 minutes at scale prohibiting practical deployment.

Correctness To evaluate *FlxVRM*'s ability to maintain correct packet sequencing for each application, a correctness verification was performed. We implement five applications in both *FlxVRM* and a non-virtualized deployment. During the evaluation, we examined their packets information received by the end-host, as well as monitoring the stateful information on the data plane. The result in Figure 10(b) shows that the packets from different applications do not interfere with each other, ensuring accurate processing.

Throughput. We evaluated throughput of *FlxVRM* and *noMerge-FlxVRM* on P4 hardware switches by deploying

different applications. A baseline program for packet forwarding (called "Basic_sw") is used for comparison. At 10Gbps, both variants matched baseline throughput (Figure 11(a)). To confirm that virtual register operations introduce no observable overhead with heavy traffic condition, we subjected *FlxVRM* to sustained 40Gbps traffic loads (see Figure 11(b)). The framework maintained an average throughput of 39.9 Gbps (99.7% of theoretical maximum), essentially matching the performance of native implementations without virtualization optimization and preserving line-rate performance.

V. Related works

There have already been many systems that support concurrent applications and resource sharing on programmable networks, but none of them can meet the goals of *FlxVRM*.

Static bind. Multiple existing studies [14], [15], [19], [21], [33] have explored the integration of multiple programs into a merged program during the compilation stage, while statically allocating register memory. Changing register memory requires reloading and recompilation for all applications in these systems, which would restart the switch disrupting all applications running on data plane. Furthermore, the switch loses all state information leading to performance degradation.

Virtualizing of PDP. The virtualization of PDP has been explored by several previous works, like HyPer4 [19], HyperVDP [20], and P4Visor [21]. However, they usually ignore the practical hardware constraints and do not apply to real ASIC switches (such as Intel Tofino [2]). For example, HyPer4 [19] and HyperVDP [20] realize the virtualization on software switches (e.g., BMv2, DPDK). They virtualize the data plane by running a P4 program emulating the behavior of multiple P4 programs through packet resubmission and recirculation. These approaches require significantly more hardware resources compared to the native P4 programs. Recent work ActiveRMT [34] achieves hardware-agnostic memory reallocation via capsule processing but enforces strict stage-order dependencies, preventing cross-stage sharing crucial for modern NFV pipelines.

Virtualized memory. Extensive research has been conducted on memory virtualization and allocation in both SDN [35], [36] and programmable data plane [4], [13], [16], [17]. Existing methods of virtualization based on programmable data planes either change register memory by recirculation at the cost of lower line rates [4], [16], or incur large resource overhead (e.g., tables or sALUs) by repeated definition in the data plane [13], [17]. For example, dDrops and *Flow [1], [4] achieve dynamic allocation and release memory in single stage by resubmitting packet back to pipeline leading to reduced throughput [4], [16]. NetVRM [17] virtualizes and shares register memory between concurrent applications, but the rigid requirement of sharing memory within the same stages for all applications severely limits the number of concurrent applications that can be accommodated.

VI. Conclusions

We present *FlxVRM*, a flexible register memory virtualization layer for data-plane P4 programs. It allows multiple applications to share register memory of multiple stages in programmable data plane, with dynamic allocation for each application. *FlxVRM* also provides a stage mapping algorithm and merging technique in a compiler to optimize the capacity of memory sharing and concurrency while minimizing resource usage.

Acknowledgments

This work is partially supported by National Natural Science Foundation of China (NSFC) under Grant 62172189; the Innovate UK grants 10098627, 10106629 and 600648.

References

[1] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, et al., Flow event telemetry on programmable data plane, in: *Proceedings of the ACM special interest group on data communication*, 2020.

[2] Intel Tofino ASIC, https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, Accessed on: May 27, 2025.

[3] In-band Network Telemetry (INT), https://github.com/p4lang/p4-applications/blob/master/docs/INT_latest.pdf, Accessed on: May 27, 2025.

[4] M. Qian, L. Cui, X. Zhang, F. P. Tso, Y. Deng, dDrops: Detecting silent packet drops on programmable data plane, in: *Computer Networks*, 2022.

[5] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, I. Stoica, Netcache: Balancing key-value stores with fast in-network caching, in: *Proceedings of the Symposium on Operating Systems Principles*, 2017.

[6] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, K. Atreya, Incbricks: Toward in-network computation with an in-network cache, in: *Architectural Support for Programming Languages and Operating Systems*, 2017.

[7] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, et al., HPCC: High Precision Congestion Control, in: *Proceedings of the ACM special interest group on data communication*, 2019.

[8] M. Zhang, L. Cui, F. P. Tso, Z. Zhang, Y. Deng, Z. Li, Quark: Implementing convolutional neural networks entirely on programmable data plane, in: *IEEE Conference on Computer Communications*, 2025.

[9] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, P. Richtárik, Scaling distributed machine learning with in-network aggregation, in: *USENIX symposium on networked systems design and implementation*, 2021.

[10] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, R. Soulé, Netpaxos: Consensus at network speed, in: *Symposium on Software Defined Networking Research*, 2015.

[11] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, I. Stoica, Netchain:scale-free sub-rtt coordination, in: *USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[12] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. Ports, A. Panda, Multitenancy for fast and programmable networks in the cloud, in: *Hot Topics in Cloud Computing*, 2020.

[13] H. Zheng, C. Tian, T. Yang, H. Lin, C. Liu, Z. Zhang, W. Dou, G. Chen, FlyMon: enabling on-the-fly task reconfiguration for network measurement, in: *Proceedings of the ACM special interest group on data communication*, 2022.

[14] H. Soni, T. Turletti, W. Dabbous, P4bricks: Enabling multiprocessing using linker-based network data plane architecture, in: *Conference on emerging Networking Experiments and Technologies*, 2018.

[15] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, J. Wu, Hyperv: A high performance hypervisor for virtualization of the programmable data plane, in: *International Conference on Computer Communication and Networks*, 2017.

[16] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, J. M. Smith, Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow, in: *USENIX Annual Technical Conference*, 2018.

[17] H. Zhu, T. Wang, Y. Hong, D. R. Ports, A. Sivaraman, X. Jin, NetVRM: Virtual register memory for programmable networks, in: *USENIX symposium on networked systems design and implementation*, 2022.

[18] Intel 2020 OpenTofino, https://github.com/barefootnetworks/Open-Tofino, Accessed on: May 27, 2025.

[19] D. Hancock, J. Van der Merwe, Hyper4: Using p4 to virtualize the programmable data plane, in: *Conference on emerging Networking EXperiments and Technologies* , 2016.

[20] C. Zhang, J. Bi, Y. Zhou, J. Wu, Hypervdp: High-performance virtualization of the programmable data plane, in: *IEEE Journal on Selected Areas in Communications*, 2019.

[21] P. Zheng, T. Benson, C. Hu, P4visor: Lightweight virtualization and composition primitives for building and testing modular programs, in: *Conference on emerging Networking EXperiments and Technologies*, 2018.

[22] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, J. Rexford, Clove: Congestion-aware load balancing at the virtual edge, in: *Conference on emerging Networking EXperiments and Technologies*, 2017.

[23] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, W. Willinger, Sonata: Query-driven streaming network telemetry, in: *Conference of the ACM Special Interest Group on Data Communication*, 2018.

[24] M. Yu, L. Jose, R. Miao, Software defined traffic measurement with openSketch, in: *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[25] L. Zeno, D. R. Ports, J. Nelson, D. Kim, S. Landau-Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, M. Silberstein, SwiSh: Distributed shared state abstractions for programmable switches, in: *USENIX Symposium on Networked Systems Design and Implementation*, 2022.

[26] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, S. Uhlig, Elastic Sketch: Adaptive and fast network-wide measurements, in: *Proceedings of the ACM special interest group on data communication*, 2018.

[27] S. Sheng, Q. Huang, P. P. Lee, DeltaINT: Toward general in-band network telemetry with extremely low bandwidth overhead, in: *IEEE International Conference on Network Protocols*, 2021.

[28] J. Sonchack, A. J. Aviv, E. Keller, J. M. Smith, Turboflow: Information rich flow record generation on commodity switches, in: *Proceedings of the EuroSys Conference*, 2018.

[29] X. Zhang, L. Cui, F. P. Tso, W. Jia, Compiling service function chains via fine-grained composition in the programmable data plane, in: *IEEE Transactions on Services Computing*, 2023.

[30] Turboflow code repository, https://github.com/jsonch/turboflow, Accessed on: May 27, 2025.

[31] X. Chen, H. Liu, Q. Huang, P. Wang, D. Zhang, H. Zhou, C. Wu, SPEED: Resource-efficient and high-performance deployment for data plane programs, in: *IEEE International Conference on Network Protocols*, 2021.

[32] L. Jose, L. Yan, G. Varghese, N. McKeown, Compiling packet programs to reconfigurable switches, in: *USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[33] M. Qian, L. Cui, X. Zhang, F. P. Tso, Y. Deng, Z. Li, W. Jia, DisPLOY: Target-Constrained Distributed Deployment for Network Measurement Tasks on Data Plane, in: IEEE Transactions on Parallel and Distributed Systems, 2025.

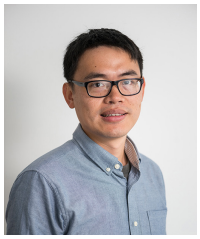
[34] R. Das, A. C. Snoeren, Memory management in ActiveRMT: Towards runtime-programmable switches, in: *Proceedings of the ACM special interest group on data communication*, 2023.

[35] X. Jin, J. Gossels, J. Rexford, D. Walker, CoVisor: a compositional hypervisor for software-defined networks, in: *USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[36] M. Moshref, M. Yu, R. Govindan, A. Vahdat, Scream: Sketch resource allocation for software-defined measurement, in: *Proceedings of the ACM special interest group on data communication*, 2015.

Mimi Qian is a postgraduate student in Jinan University. She received the B.E. degree in communication engineering from Hainan Tropical Ocean University, China, in 2014. Her current research interests include stateful data plane/programmable data plane, software-defined networking, and machine learning in computer networks.

Lin Cui is currently a professor in the Department of Computer Science at Jinan University, Guangzhou, China. He received the Ph.D. degree from City University of Hong Kong in 2013. He has broad interests in networking systems, with focus on software defined networking (SDN), programmable networking, NFV, cloud networking, distributed systems and so on.



Fung Po Tso received his B.Eng., M.Phil. and Ph.D. degrees from City University of Hong Kong in 2006, 2007 and 2011 respectively. He is currently a professor in the Department of Computer Science at the Loughborough University. His research interests include: Network resource management, edge computing, edge AI, In-network computing, digital twins, and cybersecurity.

Yuhui Deng received the Ph.D. degree from the Huazhong University of Science and Technology in 2004. He is currently a Professor with the Computer Science Department, Jinan University. He worked with the EMC Corporation as a Senior Research Scientist from 2008 to 2009. He worked as a Research Officer with Cranfield University, UK, from 2005 to 2008. His research interests cover green computing, cloud computing, information storage, computer architecture, and performance evaluation.

Zhen Zhang received the BS and MS degrees in computer science from Jilin University, China, in 1999 and 2003, and PhD degree in College of Computer Science and Engineering from South China University of Technology, China, in 2011. He is currently a professor at the Department of Computer Science of Jinan University. His research interests include graph theory, parallel and distributed processing and complex networks.

Weijia Jia is currently the director of Institute of Artificial Intelligence and Future Networking, Beijing Normal University, Zhuhai, also a chair professor with BNBU, Zhuhai, China. His research interests include smart cities, IoT, knowledge graph, multicast and anycast QoS routing protocols, wireless sensor networks, and distributed systems. He has over 500 publications in prestigious international journals/conferences and research books/chapters.