Slicing Match-Action Pipeline Resources for Multitenancy on Programmable Switches

Johannes Krude, Felix Frei, Pedram Ahmadiyeh, René Glebke, Mirko Stoffers, Klaus Wehrle
RWTH Aachen University, Germany
{*lastname*}@comsys.rwth-aachen.de

***Abstract*—Match-action programmable switches enable high-performance custom packet processing without custom hardware. However, these switches are expensive and cannot currently be reasonably shared between multiple tenants since a switch provider cannot guarantee a share of the switch resources such as SRAM or TCAM for each tenant. Each match-action pipeline executes only a single program, and merging P4 source code from multiple tenants into a combined program gives no guarantee that the combined program fits onto the pipeline. We want to overcome this by showing how to divide the resources of an RMT-based match-action pipeline into slices which can be rented out individually. The resource usage of a program is checked if it matches the slice to enable safe composition with other programs. By giving shared access to most of the PHV, our approach allows for large numbers of programs on a shared pipeline. We implemented our approach for the Tofino programmable switch and successfully limited the resource usage of real P4 programs. Our evaluation shows that slicing often results in the same (and sometimes even higher) number of programs that can be accommodated on a switch, compared to merging P4 programs without resource guarantees. Additionally, our approach significantly reduces the computation time to determine that a composition does not fit onto a switch.**

***Index Terms*—match-action pipeline, multitenancy, slicing**

I. Introduction

Programmable switches bring the freedom of custom forwarding and packet manipulation while providing high throughput. Applications range from high-speed firewalls [1]–[3], custom load-balancing [4], and custom routing schemes [5] to offloading parts of application processing [6]–[8]. The match-action pipeline, introduced with reconfigurable match tables (RMT) [9] and available as Tofino, provides the needed throughput guarantees for such on-path packet processing.

Many applications require only a part of the resources, such as SRAM or TCAM, of the expensive match-action pipeline devices. They can be more economically used by putting multiple applications on a shared switch or by sharing a switch between multiple tenants, e.g., in the cloud or at an IXP. Sharing a programmable switch would make their benefits available without each tenant paying for their own programmable switch.

A match-action pipeline, such as shown in Figure 1, has multiple match-action stages, each consisting of several different components [9]. The individual components are configured to perform the same operation for each packet but can be skipped through branching (`if else`). Therefore, when configuring a match-action pipeline for multiple programs, each pipeline component can only be used by one of the programs.

Existing work for sharing switch resources either proposes a slightly modified hardware architecture [10]–[12] or merging individual P4 programs into a combined P4 program, which is then treated as a single program during compilation [13]–[16]. The difficulty in compiling P4 is fitting all program parts onto the pipeline components under the constraints imposed by the program structure [17]. In case a combined program does not fit onto the pipeline, no individual program can be blamed, and it is not obvious which tenant should modify its program.

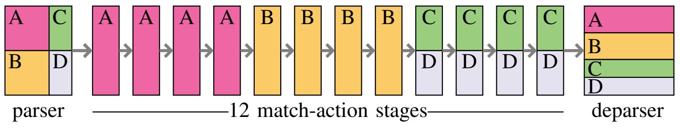


Fig. 1. A simplified example of dividing a match-action pipeline into two larger (A, B) and two smaller (C ,D) slices.

Consider two programs that require the same amount of memory but have a vastly different shape: (1) a *long program* with many small dependent tables that need to be executed in series and, therefore, need to be distributed to all stages of the pipeline, and (2) a *single-stage program* with a large table that cannot be split to multiple stages and, therefore, takes the entire memory of a single stage. These two programs cannot be placed together on the same pipeline, but each of them can be combined with many other programs. When compiling a combined P4 program without any resource guarantees, one tenant may submit one of those two programs to the switch provider, but if another tenant additionally submits the other program, the compilation of the combined program will fail.

Tenants should know in advance about the acceptable resource usage and program shape. Therefore, we propose to divide match-action pipelines into non-interfering slices, as exemplified in Figure 1. The tenant gets guaranteed match-action resources and then can choose and modify its program independently of other tenants as long as it stays within its limits. As shown in Figure 2, the switch provider divides the pipeline resources into slices and then checks if the tenant program stays within its slice. Since the resource usage of a program is decided during compilation, we propose to compile tenant programs individually and check the resource usage of compiled programs. Programs from multiple tenants that all stay within their slice can then safely be combined into a joint program that is guaranteed to fit onto the pipeline.

[1] main implemantation: https://github.com/johannes-krude/switch-slicing
modified p4c: https://github.com/johannes-krude/p4c-slicing
backup: https://zenodo.org/records/15188076

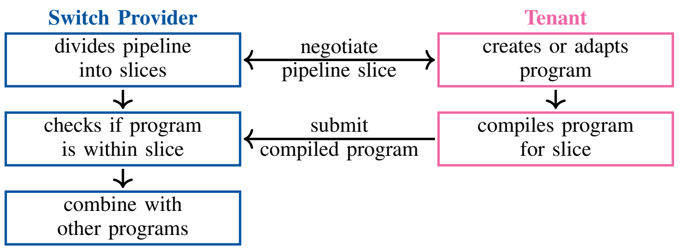

Switch Provider
Tenant
divides pipeline into slices
negotiate pipeline slice
creates or adapts program
checks if program is within slice
submit compiled program
compiles program for slice
combine with other programs

Fig. 2. The switch provider divides the pipeline into slices, checks compiled programs, and then combines them, whereas the tenant has to adapt and compile the program to fit into the slice.

To show that our approach is applicable to contemporary programmable switches, we analyze how an unmodified RMT-based [9] pipeline can be divided into slices. Although some switch resources (e.g., SRAM/TCAM) must be divided into non-overlapping parts, each assigned to a single slice, we show that this is not the case for all resources. In particular, since, at most, one tenant program is active for each packet, parts of the packet header vector (PHV) can be safely shared. We introduce a PHV sharing scheme that gives shared access to a huge part of the PHV and fits more programs onto a switch than the sum of the individual resource usage would suggest.

We implemented[1] our slicing approach to check and combine real programs on an unmodified Tofino switch and evaluate by comparing the number of coexisting programs to compiling an unchecked merged P4 program. For many programs, dividing the switch into equally-sized and -shaped slices allows to fit as many instances of the same program onto the pipeline as compiling a merged P4 program. For PHV-limited programs, our slicing approach even increases the number of coexisting programs since a huge part of the PHV can be used by all programs. However, programs that do not fit well with the chosen slice shapes result in fewer coexisting programs. Additionally, our approach significantly reduces compile time in case the program composition does not fit onto the switch since far fewer table placement variations must be assessed when compiling a single program for only a part of the switch.

Contributions.

- An analysis of the benefits and requirements for multitenancy resource isolation on programmable switches.
- A resource-slicing approach for match-action pipelines with resource guarantees while sharing most of the PHV.
- An evaluation on a real switch with realistic programs.

Structure. Section II discusses related work before we analyse multitenany in Section III. Then, Section IV explains slicing of individual pipeline resources, followed by a discussion in Section V on useful program and slice shapes. How slicing can be applied to a Tofino 1 is presented in Section VI followed by an evaluation in Section VII and a conclusion in Section VIII.

II. Related Work

Slicing the resources of existing pipelines has been deemed impossible [10] until now, but important steps have been made.

Resource Checking. Wang et al. [15] count the overall number of resources used by a tenant program without considering how resource usage is spread over the pipeline stages. The P4 source of checked programs is merged, and the combined program is again compiled. With two programs of an incompatible shape (e.g., a long program and a wide program), the resource checks would individually succeed, but compiling a combined program would fail. Our approach guarantees resources at individual pipeline stages and combines *compiled* programs to ensure resource usage does not change after checking.

Hardware Modification. In a subsequent work, Wang et al. [10] claim that resource isolation is impossible on existing hardware and propose a modified RMT architecture. Each packet is assigned a program identifier, and pipeline parts, such as match-crossbars and the deparser, execute different instructions for different program identifiers. This allows for full access to the PHV, which is usually too small to be effectively divided between tenants. MTPSA [11] and P4VBox [12] go even further by proposing a separate pipeline for each tenant program. We show that resource isolation is possible on existing RMT-based [9] hardware by introducing a PHV sharing scheme that still allows for separate deparser validity bits.

P4 based Approaches. P4Weaver [13], P4Bricks [16], and P4Visor [14] compose programs on a P4 level without giving resource guarantees. Hyper4 [18] and HyperVDP [19] emulate P4 within P4 with an impractically huge overhead, whereas ActiveRMT [20], SwitchVM [21], and P4runpro [22] emulate a simpler instruction set with a reduced expressiveness. With our approach, a program is natively executed on the hardware but gets access to only a subset of the pipeline.

III. Multitenancy on Match-Action Pipelines

Sharing a programmable switch may be beneficial at any networking infrastructure where multiple tenants are present, such as in the cloud or at an IXP. Through cooperation with DeCIX, we are familiar with a real-world use case that we use to illustrate the benefits and requirements of switch sharing.

Use Case: Virtual Edge Routers at an IXP. From a technical viewpoint, an IXP is essentially a big distributed switch that interconnects many edge routers from different IXP customers. Each customer places a physical edge router on the premise of the IXP and then connects it to an IXP switch. The edge router makes the routing decisions, encapsulates packets for the customer's internal network (e.g., MPLS), and filters unwanted packets. By moving from separate physical devices for each customer to *virtualized edge routers on a shared programmable switch*, IXP customers would save on their physical devices.

DeCIX proposed [5] to replace bloated fixed-function switches at IXPs with programmable switches to save space and energy. Since IXP packet forwarding is relatively simple, the switches have plenty of resources left that can be rented out to IXP customers. Virtualizing multiple edge routers on a shared IXP switch vastly reduces the number of devices and needed space. Sharing the costs of programmable switches between the IXP and its customers has the potential to benefit both sides. Instead of placing a physical router on the IXP premise, the IXP customer then uploads a switch program and becomes a tenant on the switch provided by the IXP.

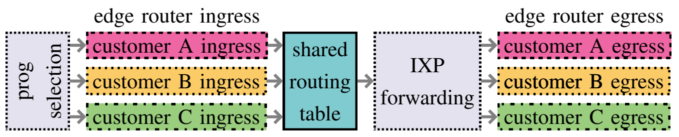


Fig. 3. A simplified program structure for virtualized edge routers at an IXP.

Sharing Common Routines. Although each IXP customer may need a different edge router adapted to the customer's network, some tasks are performed similarly by each virtualized edge router. Since all IXP customers can reach the same networks at the IXP, the content of their routing table is mostly the same, except when an IXP customer has some special routing policies for selected destinations. The routing table takes up a major part of the edge routers' memory and, therefore, provides an opportunity to reduce resource usage by sharing common entries. More generally, if the IXP provides common routines, edge routers only need to implement the functionality that differs between customers. *Providing common routines reduces the resource usage of tenant programs, but which common routines are suitable depends on the use case.*

Safely Composing Programs. Executing programs from multiple tenants on a shared switch bears the risk of unwanted or malicious interaction between the tenants. The switch provider should prevent access to other tenants' packets or memory and should ensure that tenant programs are isolated from each other. On a match-action pipeline, each packet is separately processed, and due to the lack of pointers, memory cannot be arbitrarily accessed. Therefore, as already proposed by others [13]–[16], access to packets and allocated memory can be isolated through control flow. With control flow starting in a *base program* from the switch provider, tenant programs can be selectively embedded at predefined extension points [13].

Figure 3 shows an exemplary program structure for safely composing IXP forwarding with virtual edge routers. For each incoming packet, the ingress part of only one virtual edge router is called, giving no access to this packet for other customers. Then, optionally, the shared routing table is applied, followed by the IXP forwarding the packet to the egress part of another edge router. Each edge router may arbitrarily parse customer-specific headers and perform customer-specific packet processing but only for packets from or to its own network.

Resource Isolation. The computational resources on a match-action pipeline, such as memory and action slots, are limited and distributed over the individual stages of the pipeline. These resources are allocated during compilation and cannot be changed while the switch processes packets. Resource allocation is local to stages, resulting in a compiled program having a shape, i.e., how resource usage is distributed over the stages. The shape of a program is not rigid since the compiler has some freedom in fitting program parts onto pipeline stages. When combining the base program with the tenant programs on a P4 source code level, as is done in related work [13]–[16], the resulting shape of individual programs depends on other tenants' programs since the compiler searches for a resource allocation that accommodates all parts of the combined program. In such a scenario, a tenant program can only be included in a combination if it is compatible with the sizes and shapes of all the other tenant programs. If a tenant has to know the other tenants' programs, it becomes difficult to construct a program.

We propose that each tenant should get a guaranteed slice of the pipeline resources. Resource isolation is provided by dividing the pipeline resources into non-interfering slices and checking whether tenant programs stay within their slice.

IV. Slicing Pipeline Resources

We want to isolate resource usage on programmable switches based on the RMT [9] architecture by slicing their match-action pipeline. The slices should enable to reason about the resource usage of each tenant program individually while allowing resource-safe composition of the tenant programs.

The Pipeline. As shown in Figure 4, each packet processed by the pipeline is first deconstructed by the parser, is then modified by multiple match-action stages, and finally reconstructed by the deparser. To accommodate packets with different headers, the parser is a finite state machine that acts upon previously extracted headers and stores the extracted headers into fields of the packet header vector (PHV). After being modified by the match-action stages, the PHV finally reaches the deparser, which concatenates a list of fields from the PHV to reconstruct the packet. Each packet passes through the pipeline twice, once to select the output port (ingress) and again after the output port is selected (egress).

Tables and Control-Flow. Each match-action stage is split into a limited number of logical tables executed in parallel. A table uses a subset of the crossbar to extract the match key from the PHV, optionally hashes the key, and matches the key against a subset of the SRAM or TCAM to select a single action, which modifies some PHV fields.

Most importantly, tables provide control flow through the *next-table* mechanism. The matched entry specifies the table to be executed next, and all tables between the current and the next table are skipped. The next table can be in the same stage as the current table, but only if the current action does not modify any PHV fields used in the next table's match key [17]. For additional control flow, each table can have a gateway that checks some PHV fields, similar to a restricted `if` condition, to select another next-table, thereby skipping the current table.

Partitioning Pipeline Resources. Most parts of the pipeline are configured to always perform the same operation and this configuration can only be changed with downtime. E.g., a crossbar byte always extracts the same PHV field for each packet, and an SRAM cell is always allocated to the same logical table even if the table is skipped for a packet. This kind of resource cannot be shared between multiple tenants and, therefore, needs to be partitioned between the slices.

As shown in the example in Figure 4, each tenant gets a subset of the parser transitions, crossbar bytes, SRAM and TCAM cells, action slots, and deparser entries. The partitioning can vary between stages, as in this example, the tenants get fewer resources in the first and last match-action stages since

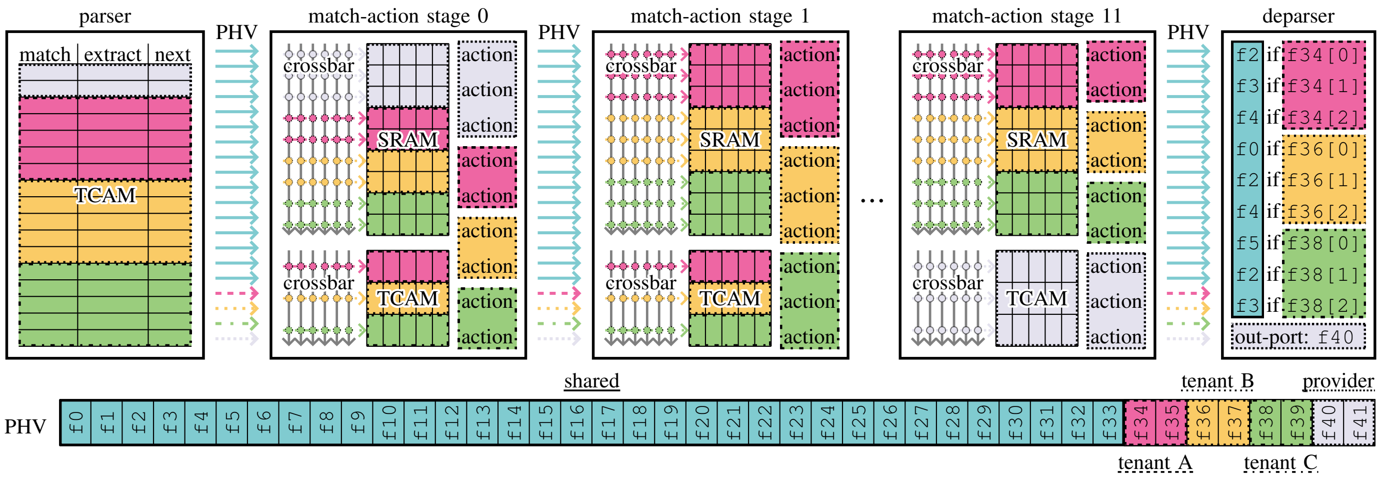


Fig. 4. Most resources are partitioned among tenants A, B, and C and the providers base program, but most of the PHV is sharedly usable by all tenants.

the provider's base program also needs resources to implement tables that pass control flow to the correct tenant and may provide common routines such as a shared routing table.

Shareable Pipeline Resources. Pipeline resources that have no configuration have the potential to be jointly used by multiple tenants. This includes buses that transport data within the pipeline, the most important of which is the packet header vector (PHV). Fields in the PHV are assigned by the parser and actions and are consumed by the match crossbar, actions, and deparser. If, at most, one tenant program is called for each packet, no conflict exists between multiple tenant programs reading and writing the same PHV fields.

PHV Validity Bits. However, the deparser is not part of the control flow provided by the next-table mechanism and always interprets all PHV validity bits for each packet, complicating PHV sharing. The deparser is configured with a list of PHV fields, which are concatenated to produce the output packet, with each entry being guarded by a bit from a PHV field called the validity bit. If one tenant program sets a PHV bit used as a validity bit in another program's deparser, the deparser will output fields from the wrong program. Therefore, PHV fields used for validity bits by one program cannot be used for headers or validity bits of other programs.

We propose to divide the PHV into a *shared* and an *exclusive part*. All tenants get access to all of the shared part, but the exclusive part is partitioned between the slices. Each tenant program can use both the shared part and its exclusive part of the PHV, but it is not allowed to use the shared part as validity bits since this would interfere with other programs' deparsers. The example in Figure 4 shows that multiple tenants have their header data in the same PHV fields for the ingress deparser, but the validity bits are exclusive.

Takeaway. *We have to partition the parser, deparser, and match-action stages since they store tenant-specific configuration. The packet header vector (PHV) can be mostly shared between tenants since it is only indirectly used by the other components. Only the validity bits in the PHV cannot be shared since this would break isolation between tenants' deparsers.*

V. Slice- and Program Shapes

By checking if each individual program stays within its slice, the switch provider can ensure that programs are composable without resource conflicts. Since resources are stage-local, the provider must check if the shape, i.e., how resource usage is distributed over stages, matches the allocated slice.

Compile Programs Individually. How a program is fitted to the stages is decided by the compiler, which respects the resource limits of individual stages and the dependencies between tables. Tenant programs, therefore, need to be individually compiled for their slice, and the provider checks and composes the compiled programs. Fitting a program into a small slice may require manual modifications, and the tenant can more freely change the program while preserving the intended functionality.

Choosing Slice Shapes. A program can only be fitted into a slice if the shape of the slice is compatible with the program. A slice with few large stages allows for large tables, but more smaller stages allow for longer dependency chains. Splitting the PHV into more or less shared and exclusive fields influences the amount of possible headers, metadata, and validity bits.

A switch provider may individually negotiate a slice shape with each tenant or divide a pipeline into identical slices. Offering only a few slice shape options may make it easier to rent out all resources and move tenant programs between switches but restricts the possible program shapes.

Reordering Undivided Stages. Slice shapes can be made more flexible by giving a tenant a mix of undivided and divided stages. Since an undivided stage is not used by any other tenant, it can be reordered to any position in the pipeline without affecting other tenants. Changing the order between undivided and divided stages in a slice allows tenants to use the larger undivided stage where it brings the most benefit. However, this reordering is possible only in the special case of undivided stages but not between differently divided stages since all other tenants using the same stages would also have to reorder their stage usage. E.g., if a tenant's slice contains

$^1/_2$ of a stage followed by $^1/_3$ of a stage and the tenant would reorder them, then other tenants would also have to reorder their stage usage since a table using $^1/_2$ of a stage cannot be relocated into $^1/_3$ of a stage.

Using Multiple Pipelines or Switches. To gain longer slices or increase flexibility, a slice could span over match-action stages from multiple pipelines or switches. As mentioned in related work [23]–[28], a program can be split between pipelines by serializing all PHV fields into a packet and reconstructing the PHV on another pipeline that continues processing with the next virtual match-action stage. Splitting a program oblivious to the tenant, however, requires solving new consistency problems for stateful programs, as it is now possible that packets get lost or reordered between match-action stages.

Takeaway. *The chosen slice shapes vastly influence which tenant programs are possible. Flexibility can be increased by reordering undivided stages or combining multiple pipelines.*

VI. Slicing the Tofino

We show that our approach is applicable to existing hardware by applying it to the pipeline of the most popular RMT-based [9] programmable switch, the Tofino 1. We analyzed how the pipeline components can be sliced and implemented an automated toolchain to check Barefoot Assembly (.bfa) files whether they stay within their slice and then compose them with other Assembly files for joint execution on real hardware.

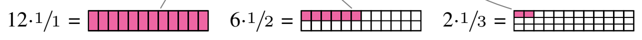
The Tofino Pipeline. A Tofino ASIC has up to four identical pipelines, each of which can execute a separate program. We, therefore, separately slice each pipeline. Each pipeline is connected to 16 ports of 100 GbE, which, in the IXP use case from Section III, could each be rented out to a different tenant if we achieve to divide the pipeline into 16 slices. We form functionally identical slices so that a tenant program can be compiled independently of the slot assigned for execution.

PHV. The Tofino 1 PHV consists of 224 fields of different sizes (64· 8 bit, 96· 16 bit, 64· 32 bit), which are organized into groups of 16 fields. Actions can only combine fields within a group, and a group can only be divided into halfs (8+8 fields) between ingress and egress processing, which is done in parallel on the same pipeline. When executing a single program on the pipeline, the PHV can be split between ingress and egress in a way suitable for this program. However, when combining multiple tenant programs, the PHV split for shared PHV fields has to be the same for all programs since the parallelly handled ingress and egress packets might belong to different tenants.

We slice the PHV by allocating most of the PHV half-groups to shared ingress or shared egress, with a few additional individual fields reserved for exclusive usage as validity bits. Therefore, the Tofino PHV can be divided into up to 48 equally-sized PHV slices with then each slice consisting of 16 ingress validity bits, 16 egress validity bits, and 128 shared PHV fields divided between ingress and egress. This slicing exceeds the goal of 16 tenant programs, and each tenant program can process up to 16 ingress and 16 egress headers, which fits all but one of our example programs (see Table II). With fewer number of slices, more per-slice validity bits are possible.

TABLE I
The number of slices when dividing the 12 match-action stages of the Tofino 1 into functionally identical slices. E.g., dividing into slices of 6·$^1/_2$ stages, results in 4 such slices.

		Number of Stages per Slice						
		12	6	4	3	2	1	
Stage Division	$^1/_1$	1	2	3	4	6	12	*Number of Slices*
	$^1/_2$	2	4	6	8	12	24	
	$^1/_3$	3	6	9	12	18	36	



Parser and Deparser. The 256 parser states and 192 deparser entries of the Tofino can be arbitrarily partitioned since there are no dependencies between parser states and between deparser entries. They do not limit the possible number and shapes of slices, as other switch resources are far more constrained.

Match-Action Stages. Each of the 12 identical match-action stages of the Tofino consists of many different resources. Memory is organized in rows, 8 SRAM rows and 12 TCAM rows, which can be independently allocated to different slices. The match crossbars for SRAM and TCAM are many bytes wide, but the individual bytes have slightly different constraints and cannot be arbitrarily exchanged for one another. We identified only three blocks of ternary crossbar bytes with identical constraints and, thus, can partition them into, at most, three interchangeable slices with ternary match capabilities. Hash distribution units can only be divided into two independent blocks, but most of the examined programs do not use hash distribution units. We divide the resources in a match-action stage of the Tofino 1 into functionally identical slices containing $^1/_2$ or $^1/_3$ of the resources, but some resources are lost in the division as they are not exactly dividable by two or three.

Dividing the 12 match-action stages of the Tofino 1 into *identically sized* slices with the *same amount of resources in each stage* results in slices spanning 12, 6, 4, 3, 2, or 1 stage(s) with $^1/_1$, $^1/_2$, or $^1/_3$ of the per-stage resources as shown in Table I. Up to 36 slices are possible, but each such slice consists of only one-third of a single stage. When targeting, for example, 4 slices, this can be achieved with slices of 6·$^1/_2$ stages, or 3·$^1/_1$ stages. Other combinations exist but result in uneven or non-interchangeable slices. Since all the non-match-action stage resources can be divided into more slices, *a Tofino 1 pipeline can be divided into up to 36 slices*.

A Minimal Base Program. We constructed a minimal base program for Tofino that passes control to placeholders for tenant programs but could be extended to, e.g., include a shared routing table or complex forwarding logic. Our minimal base program does not consume any bytes from the packet but only peeks at the input port and calls a different tenant program for each port. It also has no deparser or tables of its own and allows the tenant programs to set the egress port. To avoid an additional program selection table, an `if else` chain of gateways is used to pass control to the first table of tenant programs. This base program is lightweight as it consumes only 2 PHV fields and per-tenant additionally 2 parser transitions, 2 crossbar bytes, and 2 gateways, one for ingress and one for egress, respectively. This base program provides tenant programs with the same environment as if they were solely executed, but depending on the use case, more interaction between base and tenant programs could be implemented.

Checking and Composing Barefoot Assembly. Our automated resource checker and program composer implementation works on programs compiled to Barefoot Assembly files. Since the Tofino supports no indirection, such as pointers, all used resources are specified in the Barefoot Assembly file as concrete addresses. Resource checking is done by comparing the addresses to resources allocated to the slice, and the program can be relocated to another functionally identical slice by replacing all addresses with an equivalent address from the target slice. Well-formed programs that match their slice are safely composed by concatenating their parser, match-action tables, and deparser to the base program. Control flow in the base program's parser and match-action stages is adapted to point to the tenant programs instead of the placeholders. No adjustment to the deparser is needed, as it has no control flow.

Testing for Correctness. To check if the relocation or composition affects the correct behavior of programs, we executed some of the programs from Table II on a Tofino 1 switch. We generated test packets through symbolic execution on the P4 source code of the individual programs, similar to p4pktgen [29], and confirmed that the programs still behave the same in a composition on the switch.

Throughput, Control Plane, Stateful Programs. Several additional mechanisms would be necessary when using our implementation in a real deployment. The Tofino 1 guarantees to process packets at a fixed rate, but if a tenant manages to occupy a large portion of the guaranteed bandwidth with its own packets, other tenants won't receive many packets. The switch provider should, therefore, suppress excessive recirculation and routing loops through his routing mechanism [11], [15]. Our implementation only deals with the data plane, but programs are usually accompanied by a control plane program that reads and modifies the table entries stored in SRAM and TCAM. Control plane programs are executed on a general-purpose CPU, can be virtualized with well-established operating system virtualization techniques, and should be given access to only a tenant's own data plane tables, as described in [11], [15]. Loading a program onto the Tofino 1 incurs a downtime of up to 50 ms [30], and all stateful elements, such as counters and registers, lose their state [31], [32]. Several existing works [31]–[36] propose solutions for changing a single or multiple programs, and future work could investigate whether they are applicable to slicing.

VII. Evaluation

The evaluation investigates if slicing results in useful resource utilization on the Tofino 1. The main metrics are the size of the needed slices and how many programs can share a pipeline. Additionally, we compare compilation times.

Example Programs. We evaluate our approach by fitting the realistic P4 programs shown in Table II into a subset of the pipeline resources. When necessary, we ported the programs to the P4 Tofino Native Architecture and normalized them by putting implicit actions into tables without a match key. To show the different shapes of the programs, we list the number of tables and the minimum number of needed stages, which corresponds to the longest match-action dependency chain. The programs vary in their resource bottleneck, as can be seen by their most used non-PHV resource. We built the single_table program as an intentionally small program to showcase the capabilities of our approach. For the IXP use case, we implemented IXP_routing, which is the shared routing table from Figure 3 and two possible edge routers: MPLS and VXLAN. Two programs, chord and firewall_simple, were built by students in a P4 class. The remaining programs were taken from public repositories or published papers. We investigated some additional programs [17], [44]–[46] but excluded them from the evaluation since they use too many pipeline resources to be applicable to switch sharing.

TABLE II
The example P4 programs used in the evaluation.

Program	Lines	Tables	Min Stages	Validity Bits	Most Used Resource (except PHV)
single_table	99	1	1	0+0	table
IXP_routing	452	13	4	1+0	SRAM
MPLS	212	4	1	1+2	deparser
VXLAN	461	9	3	2+6	deparser
chord	419	11	4	19+0	parser
firewall_simple	245	4	1	4+0	deparser
accelerator [37]	477	16	4	1+1	table
aes [38]	722	30	5	7+0	checksum
conquest_baseline [39]	347	5	2	4+4	checksum
flowlet [40]	345	5	2	3+0	deparser
RTT [41]	779	20	10	5+0	hashing
speedtest [42]	895	1	1	5+0	deparser
advanced_tunnel [43]	261	2	2	3+0	registers
calc [43]	302	2	1	2+0	deparser
ecn [43]	228	3	1	2+2	deparser
firewall [43]	413	13	5	3+0	hashing
multicast [43]	212	2	1	2+1	deparser
qos [43]	314	3	1	2+0	deparser
source_routing [43]	245	4	1	11+0	deparser

Overview. At first, we evaluate resource checking and composition using a slice-unaware compiler to investigate the case of a compiled program not fitting into the slice. We then continue with several slice-aware compilers to asses how many program instances can be fitted onto a shared switch.

A. Resource Checking & Composing

Using the existing slice-unaware Tofino P4 compiler, we compile each example program independent of any slice and then check for each of the slicings from Table I if the resources used by the compiled program are within the slice or can be relocated into the slice. If the compiled program does fit into a slice, we then take multiple instances of the compiled program and combine them with the base program for joint execution.

Although a compiled program may use less than the number of resources allocated to a slice, the slice-unaware compiler

TABLE III
SLICES INTO WHICH THE EXAMPLE PROGRAMS WERE SUCCESSFULLY RELOCATED. FOR EACH PROGRAM, THE SLICE WITH THE FEWEST RESOURCES IS HIGHLIGHTED.

	Number of Stages		
Program	${}^{1}/_{1}$ Stages	${}^{1}/_{2}$ Stages	${}^{1}/_{3}$ Stages
single_table	1–12	1–12	**1**–12
IXP_routing	**12**		
MPLS	1–12	**1**–12	
VXLAN	3–12	**3**–12	
chord	**6**–12		
firewall_simple	**1**–12	**2**–12	
accelerator	**4**–12		
aes	**6**–12		
conquest_baseline	**2**–12	**4**–12	
flowlet	2–12	**2**–12	
RTT	**12**		
speedtest	**2**–12	**4**–12	**6**–12
advanced_tunnel	**2**–12		
calc	1–12	**1**–12	2–12
ecn	**1**–12	**2**–12	
firewall	**6**–12		
multicast	**1**–12	**2**–12	
qos	**1**–12	**2**–12	
source_routing	2–12	**3**–12	

speedtest fits into each of: $2\cdot{}^{1}/_{1}$, $3\cdot{}^{1}/_{1}$, $4\cdot{}^{1}/_{1}$, $6\cdot{}^{1}/_{1}$, $12\cdot{}^{1}/_{1}$, $4\cdot{}^{1}/_{2}$, $6\cdot{}^{1}/_{2}$, $12\cdot{}^{1}/_{2}$, $6\cdot{}^{1}/_{3}$, $12\cdot{}^{1}/_{3}$

does not respect slice boundaries and may use a mix of resources assigned to different slices. We, therefore, extended our implementation with heuristics that align a compiled program to a single slice: PHV fields are rearranged into the shared and exclusive part based on whether they are used as validity bits. Match-action resources (e.g., match crossbar bytes) are moved to entities with identical behavior.

Table III lists for each program all the slices into which we could relocate the compiled program. Whenever a program fitted into a slice, we also successfully composed the program with additional program instances, relocated to the remaining slices, without causing resource conflicts. This shows that slice checking, relocation, and composition work as intended.

For each program, the slice with the fewest number of resources is highlighted. If multiple slices are highlighted for a program, this program has its resource bottleneck not in the match-action stages but in the parser, deparser, or PHV. E.g., speedtest, needs a ${}^{1}/_{6}$th of the deparser entries and therefore only fits into slices that include at least that amount. Slices of $2\cdot{}^{1}/_{1}$, $4\cdot{}^{1}/_{2}$, and $6\cdot{}^{1}/_{3}$ stages all come with a ${}^{1}/_{6}$th of the deparser entries and are, therefore, the smallest possible slices for speedtest, although it actually uses only two stages.

Although we manually optimized the P4 source code of IXP_routing to fit into $4\cdot{}^{1}/_{1}$ stages, the slice-unaware compiler chooses a shape incompatible with relocation. Similarly, relocating programs into slices of ${}^{1}/_{3}$ stages failed for most programs. Our resource checker correctly identifies if a compiled program stays within a slice, but a slice-unaware compiler may choose a program shape that does not match the targeted slice.

Takeaway. *With our approach, checking programs for resource isolation and resource-safe composition is applicable to an unmodified Tofino 1 pipeline. However, the slice-unaware Tofino P4 compiler is not good at fitting programs into small slices.*

B. Program Density

Slicing guarantees tenants the availability of resources even when their program does not use them all. If a tenant program is not well-adapted to the provider-chosen slice shape, resources remain unused and cannot be reallocated to another tenant. To investigate the usefulness of slicing pipeline resources for switch sharing, we evaluate how many programs can share a sliced pipeline without manually optimizing the programs for the slices. We compare this number to merging programs on a P4 source code level, which is not restricted by the slice boundaries and can more freely allocate resources. Compiling a combined P4 program can result in more programs on a shared switch but gives no resource guarantees for any of the programs in case a single program changes.

Compiling for a Slice. For a useful comparison, we try to maximize resource utilization with a compiler that fits programs into small slices. Since the source code of the Tofino P4 compiler was recently made available [47], we modified the compiler to target a slice instead of the complete pipeline. We developed two additional approaches that utilize an unmodified compiler and allow for fitting into a wider range of slices.

1. Modifying the Tofino P4 Compiler (p4c). We made the Tofino P4 compiler slice-aware by making several resource constants runtime-configurable, such as the number of SRAM and TCAM rows, crossbar groups, and logical table ids. Since the compiler was not intended to be used for less than a complete pipeline, more complicated slice constraints can not be easily implemented. Instead of making the compiler aware of PHV shared and exclusive usage, we relocate PHV fields in the resulting Barefoot Assembly. Additionally, the compiler uses the same number of match-action resources for each stage, which makes this approach applicable to only uniform slices.

2. SMT-based `@stage` Annotations. As an alternative approach for non-uniform slices, we used the Z3 SMT solver to compute a placement of tables into match-action stages. The resulting table placement is then added to the P4 source as `@stage` annotations to instruct the Tofino P4 compiler to place the annotated tables into the specified stages. We generate SMT constraints similar to Jose et al. [17] and made it applicable to real hardware by respecting conditional dependencies to properly describe the next-table mechanism. This approach works with an unmodified compiler, can target non-uniform slices, and can reorder between undivided and divided stages.

3. Generate Additional Tables for Forbidden Resources. A third approach generates tables annotated to occupy the resources outside the slice. The compiler, therefore, has to find a fit where the tenant program uses the remaining resources, and our implementation afterward removes the generated tables from the Barefoot Assembly. This allows for non-uniform slices but does not support stage reordering. For some programs, this approach yields the smallest fit, perhaps because the heuristics of the Tofino P4 compiler are optimized for a complete pipeline.

Table IV shows for each example program how many program instances we could jointly execute on a Tofino 1 pipeline. The number of program instances in the Slicing

TABLE IV
COMPARISON OF HOW MANY PROGRAM INSTANCES FIT ONTO A TOFINO 1 PIPELINE WHEN COMPILING A COMBINED P4 PROGRAM OR INDIVIDUALLY COMPILING FOR SLICES.

	Max. Prog. Instances				Achieved By
Program	P4 Merge		**Slicing**	Smallest Slice(s)	Compiler
single_table	36	=	36	$1\cdot{}^{1}/_{3}$	p4c, SMT, gen
IXP_routing	3	=	3	$4\cdot{}^{1}/_{1}$	gen
MPLS	32	≈	24	$1\cdot{}^{1}/_{2}$	p4c, SMT, gen
VXLAN	6	<	**8**	$3\cdot{}^{1}/_{2}$	p4c, SMT, gen
chord	7	>	3	$4\cdot{}^{1}/_{1}$, $12\cdot{}^{1}/_{3}$	SMT, gen
firewall_simple	12	=	12	$1\cdot{}^{1}/_{1}$, $2\cdot{}^{1}/_{2}$, $3\cdot{}^{1}/_{3}$	p4c, SMT, gen
accelerator	9	>	6	$4\cdot{}^{1}/_{2}$	p4c
aes	3	=	3	$12\cdot{}^{1}/_{3}$	p4c
conquest_baseline	6	=	6	$2\cdot{}^{1}/_{1}$, $4\cdot{}^{1}/_{2}$	p4c, SMT, gen
flowlet	13	≈	12	$2\cdot{}^{1}/_{2}$	p4c, SMT
RTT	2	>	1	$12\cdot{}^{1}/_{1}$	p4c, SMT, gen
speedtest	6	=	6	$2\cdot{}^{1}/_{1}$, $4\cdot{}^{1}/_{2}$, $6\cdot{}^{1}/_{3}$	p4c, SMT, gen
advanced_tunnel	12	=	12	$2\cdot{}^{1}/_{2}$	p4c
calc	24	=	24	$1\cdot{}^{1}/_{2}$	p4c, SMT, gen
ecn	21	≈	18	$2\cdot{}^{1}/_{3}$	p4c, SMT
firewall	4	>	2	$6\cdot{}^{1}/_{1}$, $12\cdot{}^{1}/_{2}$	p4c, SMT, gen
multicast	15	<	**24**	$1\cdot{}^{1}/_{2}$	gen
qos	21	≈	18	$2\cdot{}^{1}/_{3}$	p4c, SMT
source_routing	10	≈	9	$4\cdot{}^{1}/_{3}$	SMT

< > =	More, less, or, the same number of program instances
≈	Same number impossible, achieves the next possible number

column results from the smallest slice we could fit the program into. For most programs, all three compiling approaches achieve the same number of program instances, but each compiling approach outperforms the others for at least one program. We compare this to naïvely merging the P4 source code of up to 36 program instances with the P4 base program and then compiling the combined program (P4 Merge column in Table IV). This number sometimes falls between the number of possible slices, e.g., for MPLS, the Tofino P4 compiler can fit 32 instances onto a pipeline, but we cannot divide the pipeline into any number between 24 and 36 equally-sized slices. The next possible number of slices for MPLS is, therefore, 24, and we can compile MPLS into a $1\cdot{}^{1}/_{2}$ stages slice, which results in the same number of 24 coexisting program instances. Slicing achieves the same or a higher number of instances for 10 of the 19 programs, and for 15 of the programs, it achieves the same or higher number of instances when rounding down to the next possible number.

Improved Density due to PHV Sharing. For two programs, VXLAN and multicast, slicing results in more programs on the pipeline than merging P4 programs. Both of these programs have a high PHV usage, and our PHV sharing approach makes a huge part of the PHV available to all tenants.The Tofino P4 compiler, without knowledge of the separation into individual programs, does not achieve the same amount of PHV sharing.

IXP Switch Densitiy. If we look at the programs for the IXP use case from Section III, we see that the IXP routing table fits into 4 stages, leaving 8 stages for the edge routers of the IXP customers. These 8 stages can accommodate 16 MPLS or 4 VXLAN instances, and maybe even more if a programmer manually adapts the program to the shape of the rented slice.

Non-Uniform Slice Shapes. To accommodate programs with a non-uniform shape, a pipeline can also be divided into slices that mix stages of different sizes. E.g., chord can be compiled into a slice of $1\cdot{}^{1}/_{1}$ + $4\cdot{}^{1}/_{2}$ stages using our SMT-based approach. Since the pipeline can be divided into 4 such slices, this is an improvement compared to the uniform slicing shown in Table IV, which only allows for 3 instances.

Takeaway. *Although resource guarantees may result in unused resources, for most of the example programs, our slicing approach achieves or exceeds the next-possible number of program instances when compared to naïvely merging P4 programs. For PHV-limited programs, our PHV-sharing approach can increase the number of coexisting programs on a shared switch. In a realistic scenario, a tenant may also adapt his program for the slice, resulting in fewer unused resources.*

C. Composition & Compile Time

Fitting a program onto a match-action pipeline is an NP-hard problem [48] and can, therefore, result in long compilation times. Especially when a program does not fit, the compiler exhaustively tries all table placement variations before it gives up. We observed up to 33.0 hours of compilation time on a Ryzen 7 5800X processor when trying to compile a combined P4 program consisting of 36 instances of the IXP_routing program. When compiling combined programs, a switch provider has to recompile everything whenever a new composition is needed, and it can take up to multiple hours until the switch provider gets feedback on whether this composition is possible. In contrast, with our approach, the switch provider only needs to check resource usage and compose compiled programs, and we observed a maximum total time for checking and composing of 6.4 seconds. Compiling programs individually for slices, therefore, not only provides resource isolation but also speeds up the composition task.

With our approach, the tenant program still has to be compiled, but this is done individually by each tenant. Since the tenant compiles only a single program for its slice, the problem solved by the compiler is vastly smaller, and we observed only up to 6.5 minutes for compiling a single program to a slice.

Takeaway. *Compiling programs individually deconstructs the compilation into smaller chunks, leading to faster compilation times. Most of the remaining compilation effort can be moved to tenants. The switch provider can, therefore, quickly compose programs within seconds.*

VIII. CONCLUSION

Slicing of match-action resources enables resource isolation since each tenant gets the guarantee that its data plane program can use all resources from its slice. A switch provider only checks whether a tenant program stays within its slice and then combines it with other tenant programs for joint execution on a switch. Although most pipeline resources have to be partitioned between slices, we show that large parts of the PHV can safely be shared. Our evaluation shows that a Tofino 1 pipeline can be divided into up to 36 slices, each holding a separate tenant program. We fitted 19 example programs into slices of varying sizes and showed that for most programs, the same program density can be reached when compared to compiling a combined P4 program without resource guarantees. For PHV-limited programs, we even improve the program density since our slicing approach improves the PHV sharing between mutually exclusive programs. Additionally, our approach significantly reduces the computation time to determine that a composition does not fit onto a switch.

Acknowledgments

This work has been funded by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 "MAKI – Multi-Mechanism-Adaption for the Future Internet".

References

[1] R. Datta, S. Choi, A. Chowdhary, and Y. Park, "P4guard: Designing p4 based firewall," in *MILCOM*. IEEE, 2018.

[2] J. Cao, J. Bi, Y. Zhou, and C. Zhang, "CoFilter: A High-Performance Switch-Assisted Stateful Packet Filter," in *SIGCOMM*. ACM, 2018.

[3] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo, "Programmable In-Network security for context-aware BYOD policies," in *USENIX Security*, 2020.

[4] 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.

[5] D. Wagner, M. Wichtlhuber, C. Dietzel, J. Blendin, and A. Feldmann, "P4IX: a concept for P4 programmable data planes at IXPs," in *FIRA*. ACM, 2022.

[6] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling Distributed Machine Learning with In-Network Aggregation," in *NSDI*, 2021.

[7] 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.

[8] A. Lernen, R. Hussein, and P. Cudre-Mauroux, "The Case for Network-Accelerated Query Processing," in *CIDR*, 2019.

[9] 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," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, 2013.

[10] T. Wang, X. Yang, G. Antichi, A. Sivaraman, and A. Panda, "Isolation Mechanisms for High-Speed Packet-Processing Pipelines," in *NSDI*, 2022.

[11] R. Stoyanov and N. Zilberman, "MTPSA: Multi-Tenant Programmable Switches," in *EuroP4*. ACM, 2020.

[12] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "P4VBox: Enabling P4-Based Switch Virtualization," *IEEE Communications Letters*, vol. 24, no. 1, 2020.

[13] A. Fattaholmanan, M. Baldi, A. Carzaniga, and R. Soulé, "P4 Weaver: Supporting Modular and Incremental Programming in P4," in *SOSR*. ACM, 2021.

[14] P. Zheng, T. Benson, and C. Hu, "P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs," in *CoNEXT*. ACM, 2018.

[15] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. K. Ports, and A. Panda, "Multitenancy for Fast and Programmable Networks in the Cloud," in *HotCloud*, 2020.

[16] H. Soni, T. Turletti, and W. Dabbous, "P4Bricks: Enabling multiprocessing using Linker-based network data plane architecture," 2018.

[17] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling Packet Programs to Reconfigurable Switches," in *NSDI*, 2015.

[18] D. Hancock and J. van der Merwe, "HyPer4: Using P4 to Virtualize the Programmable Data Plane," in *CoNEXT*. ACM, 2016.

[19] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "HyperVDP: High-Performance Virtualization of the Programmable Data Plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, 2019.

[20] R. Das and A. C. Snoeren, "Memory Management in ActiveRMT: Towards Runtime-programmable Switches," in *SIGCOMM*. ACM, 2023.

[21] S. Khashab, A. Rashelbach, and M. Silberstein, "Multitenant In-Network Acceleration with SwitchVM," in *NSDI*, 2024.

[22] Y. Yang, L. He, J. Zhou, X. Shi, J. Cao, and Y. Liu, "P4runpro: Enabling Runtime Programmability for RMT Programmable Switches," in *SIGCOMM*. ACM, 2024.

[23] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "Compiling Service Function Chains via Fine-Grained Composition in the Programmable Data Plane," *IEEE Transactions on Services Computing*, vol. 16, no. 4, 2023.

[24] W. Xu, Z. Zhang, Y. Feng, H. Song, Z. Chen, W. Wu, G. Liu, Y. Zhang, S. Liu, Z. Tian, and B. Liu, "ClickINC: In-network Computing as a Service in Heterogeneous Programmable Data-center Networks," in *SIGCOMM*. ACM, 2023.

[25] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs," in *SIGCOMM*. ACM, 2020.

[26] M. Blöcher, L. Wang, P. Eugster, and M. Schmidt, "Switches for HIRE: resource scheduling for data center in-network computing," in *ASPLOS*. ACM, 2021.

[27] X. Zhang, L. Cui, F. P. Tso, Z. Li, and W. Jia, "Dapper: Deploying Service Function Chains in the Programmable Data Plane Via Deep Reinforcement Learning," *IEEE Transactions on Services Computing*, vol. 16, no. 4, 2023.

[28] X. Jia, F. Li, S. Chen, C. Gao, P. Wang, and X. Wang, "RED: Distributed Program Deployment for Resource-aware Programmable Switches," in *INFOCOM*. IEEE, 2023.

[29] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "p4pktgen: Automated Test Case Generation for P4 Programs," in *SOSR*. ACM, 2018.

[30] A. Bas, "Leveraging Stratum and Tofino Fast Refresh for Software Upgrades," https://www.opennetworking.org/wp-content/uploads/2018/12/Tofino_Fast_Refresh.pdf, 2018.

[31] C. Ji and F. Kuipers, "State4: State-preserving Reconfiguration of P4-programmable Switches," in *NetSoft*. IEEE, 2023.

[32] J. Krude, J. Hofmann, M. Eichholz, K. Wehrle, A. Koch, and M. Mezini, "Online Reprogrammable Multi Tenant Switches," in *ENCP*. ACM, 2019.

[33] J. Xing, Y. Qiu, K.-F. Hsu, H. Liu, M. Kadosh, A. Lo, A. Akella, T. Anderson, A. Krishnamurthy, T. S. E. Ng, and A. Chen, "A vision for runtime programmable networks," in *HotNets*. ACM, 2021.

[34] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen, "Runtime Programmable Switches," in *NSDI*, 2022.

[35] Y. Qiu, R. Beckett, and A. Chen, "Synthesizing Runtime Programmable Switch Updates," in *NSDI*, 2023.

[36] Y. Feng, H. Song, J. Li, Z. Chen, W. Xu, and B. Liu, "In-situ Programmable Switching using rP4: Towards Runtime Data Plane Programmability," in *HotNets*. ACM, 2021.

[37] J. Krude, J. Rüth, D. Schemmel, F. Rath, I.-H. Folbort, and K. Wehrle, "Determination of throughput guarantees for processor-based SmartNICs," in *CoNEXT*. ACM, 2021.

[38] X. Chen, "Implementing AES Encryption on Programmable Switches via Scrambled Lookup Tables," in *SPIN*. ACM, 2020.

[39] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang, "Fine-grained queue measurement in the data plane," in *CoNEXT*. ACM, 2019.

[40] E. C. Molero, "P4 Learning," https://github.com/nsg-ethz/p4-learning.

[41] X. Chen, H. Kim, J. M. Aman, W. Chang, M. Lee, and J. Rexford, "Measuring TCP Round-Trip Time in the Data Plane," in *SPIN*. ACM, 2020.

[42] I. Kunze, R. Glebke, J. Scheiper, M. Bodenbenner, R. H. Schmitt, and K. Wehrle, "Investigating the Applicability of In-Network Computing to Industrial Scenarios," in *ICPS*. IEEE, 2021.

[43] p4.org, "P4 Tutorial," https://github.com/p4lang/tutorials.

[44] S. Yoo and X. Chen, "Secure Keyed Hashing on Programmable Switches," in *SPIN*. ACM, 2021.

[45] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *SOSR*. ACM, 2017.

[46] D. D. Robin and J. I. Khan, "An open-source P416 compiler backend for reconfigurable match-action table switches: Making networking innovation accessible," *Computer Networks*, vol. 242, Jul. 2024.

[47] "Intel's tofino p4 software is now open source," Jan. 2025. [Online]. Available: https://p4.org/intels-tofino-p4-software-is-now-open-source/

[48] B. Vass, E. Bérczi-Kovács, C. Raiciu, and G. Rétvári, "Compiling Packet Programs to Reconfigurable Switches: Theory and Algorithms," in *EuroP4*. ACM, 2020.