Multitenant In-Network Acceleration with SwitchVM

Sajy Khashab
Technion

Alon Rashelbach
Technion

Mark Silberstein
Technion

Abstract

We propose a practical approach to implementing multitenancy on programmable network switches to make in-network acceleration accessible to cloud users. We introduce a *Switch Virtual Machine (SwitchVM)*, that is deployed on the switches and offers an expressive instruction set and program state abstractions. Tenant programs, called *Data-Plane Filters (*DPF*s)*, are executed on top of SwitchVM in a sandbox with memory, network and state isolation policies controlled by network operators. The packets that trigger DPF execution include the code to execute or a reference to the DPFs deployed in the switch. DPFs are Turing-complete, may maintain state in the packet and in switch virtual memory, may form a dynamic chain, and may steer packets to desired destinations, all while enforcing the operator's policies.

We demonstrate that this idea is practical by prototyping SwitchVM in P4 on Intel Tofino switches. We describe a variety of use cases that SwitchVM supports, and implement three complex applications from prior works – key-value store cache, load balancer and Paxos accelerator. We also show that SwitchVM provides strong performance isolation, zero-overhead runtime programmability, may hold two orders of magnitude more in-switch programs than existing techniques, and may support thousands of concurrent tenants each with its private state.

1 Introduction

Data-plane programmable PISA switches[1] transformed the field of in-network computing from dream into reality. By enabling stateful custom packet processing at a switch line-rate, they offer dramatic performance boost in infrastructure services such as telemetry [27, 34, 36] and congestion control [19, 35], and accelerate networking applications such as load balancers [25, 38, 60], distributed protocols [11, 12, 32], concurrency control [31, 55], aggregation [43, 44], storage [23, 24, 33], and more [26].

Unfortunately, the benefits of in-network application acceleration have only been accessible to data center operators. The vision of in-network programs being deployed in switches by *tenants* [29, 51] has so far remained quite far from materialization. The main obstacle is well-known: existing switches, such as Intel Tofino [20], lack the essential support for multitenancy as they do not guarantee fault, resource, and performance isolation of individual programs. Furthermore, they cannot enforce the program compliance with operator security policies, and require expensive reconfiguration each time a new program is installed, causing traffic disruption for all tenants. These limitations are further aggravated by the severe scarcity of on-switch hardware resources partitioned among co-resident data-plane applications.

Motivated by the need for in-network acceleration of tenants' applications, recent proposals tackle the challenge of switch virtualization. These solutions, however, still have some practical limitations. Novel switch hardware [16, 29, 45, 47, 50] offers isolation, but the perspectives of its adoption are unclear. Others propose to run P4-based hypervisors [18, 57] to guard the execution of application code, but high resource demands limit their application to FPGAs and software targets rather than ASIC switches such as Tofino. Another approach is to merge multiple P4 programs into a single one while enforcing isolation among them in software [42, 51, 59]. However, neither does this approach guarantee fault isolation nor does it scale beyond a handful of co-resident programs because the switch resources must be provisioned at compile-time for the aggregate of their hardware demands. Further, deploying a new program requires recompilation and reconfiguration that disrupts the traffic through the switch. Runtime programmability for partial reconfiguration alleviates the traffic disruption problem [54] but still requires an isolation-aware source code merging mechanism to allow multitenancy.

In summary, multitenancy poses two fundamental challenges: (1) how to execute code on physical network switches while constraining it to a *stringent security sandbox* guarded by network operators; (2) how to enable a concurrent deployment, execution and update of thousands of switch programs while assuring *strong isolation* among tenants.

We advocate for a *Language-Level Virtualization approach* to overcome these challenges. The switch runs a *Switch Virtual Machine, SwitchVM,* which executes *Data-Plane Filters, (*DPF*s),* written using a specialized, Turing-complete Instruction Set. SwitchVM dynamically loads a DPF specified by the tenant on a *per-packet basis*, either from pre-deployed in-switch libraries or from code located in the packets themselves. SwitchVM allows every packet to be processed by a different DPF, allocating switch hardware resources on demand per DPF while reusing the VM runtime. Thus, SwitchVM introduces a runtime interpreter for DPF

[1]We only discuss programmable switches, so we omit *programmable* in the rest for brevity.

code that translates it into switch operations using a shared dataplane runtime. This approach enables time-sharing of switch hardware across multiple DPFs, as opposed to static resource partitioning in alternative solutions, and is key to achieving superior multi-tenant scalability.

SwitchVM is inspired by software sandboxing techniques such as Berkeley Packet Filters (BPFs) [37]. DPFs are Turing-complete, yet they are less flexible than P4. Notably, DPFs cannot parse new protocol headers or define match-action tables with arbitrary keys and actions. Nevertheless, DPFs are powerful enough to implement a variety of sophisticated in-network applications. A DPF may perform arithmetic operations or hashes on packet header fields, update the in-switch program state, and modify the packet forwarding depending on the execution output, e.g., by choosing from a pre-defined set of possible destinations or by performing a multicast. We discuss many applications (§4), and fully implement a Key-Value cache [24], count-min sketch [10], Paxos accelerator [11], and several load balancers [41,60]. Furthermore, SwitchVM architecture is modular and can be easily extended to implement additional functionality, e.g., as new instructions.

DPFs of the same tenant may share a state in a switch. At the same time, SwitchVM ensures strict resource isolation between DPFs across tenants. A DPF uses virtual registers with load/store instructions to access the state in the packet, in-switch meta-data, or access *virtual* per-tenant space in the switch memory. In RMT switches, DPFs invoked by different packets will only observe consistent updates to the shared state thanks to the per-packet atomicity guarantees of the pipelined architecture. Proper placement of DPFs in multi-pipeline switches is crucial for achieving this consistency (§3.1). Furthermore, cross-tenant performance isolation is achieved thanks to the RMT line-rate throughput guarantees.

Crucially, SwitchVM restricts DPFs to a sandbox, giving fine-grain control over the data-plane functionality to operators on a per-tenant basis. SwitchVM enforces the sandbox policy at runtime because DPFs originate from untrusted tenants. Deployment of DPFs is performed via a control plane. Runtime reconfiguration by a tenant, i.e., code deployment and resource allocation, has no impact on the traffic of other tenants, as it is equivalent to adding/removing entries of match-action tables. Operators may deploy a per-tenant security policy, such as disallowing access to switch state, use of packet steering, or disabling DPF execution. The switches that are not authorized or unable to execute DPFs forward the packets as usual as they are compatible with network encapsulation protocols.

DPFs may reside in packets, similar to capsule-based active networks [14,22,46,48,49], and also pre-stored in a switch and invoked on-demand. In-switch deployment option is important because it eliminates the non-negligible bandwidth overheads of the DPF code header in a packet, and enables *code sharing* among tenants thus saving in-switch resources, yet without compromising inter-tenant state isolation.

Under the hood, SwitchVM implements a pipeline of generic Execution Units (EUs) that enable Multiple Instruction Multiple Data instruction execution, with several virtual registers and an extensive instruction set. Each DPF is spatially mapped onto the EUs. DPF chaining is supported by allowing a DPF to choose and invoke another DPF in the same or different switch to implement more complex logic.

We prototype SwitchVM on Intel Tofino-1. Implementing such a complex mechanism in P4 under tight hardware constraints is a formidable challenge. The key goal has been to fit as much logic as possible in a single hardware stage to increase the maximum instruction count of a single DPF. Higher instruction count allows DPF execution in a single pass through the switch, thereby eliminating bandwidth overheads associated with recirculation, and guaranteeing atomicity of accesses to a shared switch state, as well as strict performance isolation among DPFs.

We evaluate SwitchVM on a range of microbenchmarks and real-world applications of in-network acceleration which were originally developed in P4 in prior works. We demonstrate that performance overheads of SwitchVM compared to the P4 analogous code are negligible or none, whereas it allows concurrent execution of thousands of DPFs from different tenants, each with its own private state. We also show zero-overhead reconfiguration without packet loss, and strong performance isolation among different DPFs.

2 Motivation

The demand for in-network data-plane acceleration of network applications is steadily growing, and there is an abundance of proposals to use data-plane programmable switches for that purpose [11,23,24,26,41,60]. Such switches have become a commodity, with offerings from multiple vendors, most notably Intel [20], Broadcomm [6], Juniper [39], and NVIDIA [40].

In practice, in-switch execution has been accessible only to data center operators, and primarily used for acceleration of infrastructure workloads, such as congestion control [19,35] and telemetry [2,22]. On the other hand, a vast variety of use cases go beyond the infrastructural tasks. Key-Value Store (KVS) caching [24], consensus acceleration [11] and fast RPC load balancers [28] are just a few popular user applications shown to benefit from in-switch acceleration. Predeploying these as services by the data center is possible, but these applications might require fine-tuning for the needs of the specific tenant, i.e., custom-specific policies for the RPC scheduler, or non-standard key-value sizes and cache admission policies for KVS. Such fine-tuning is impractical on a per-tenant basis.

Building general in-switch acceleration services that can suit multiple use cases of different tenants is notoriously difficult due to the switch's hardware constraints. Satisfying these constraints is fundamental to achieving high performance on any architecture, as both Reconfigurable Match

Table switches (RMT), e.g., Intel Tofino, and disaggregated RMT (dRMT) switches, e.g., NVIDIA Spectrum, cannot guarantee line-rate throughput if attempted to execute a program that does not fit their hardware constraints.

Multitenancy in the switch would enable tenants to accelerate their own virtual networking infrastructure with custom network stacks and functions. Similarly, tenants might need to deploy custom in-network telemetry to debug their network performance. These services are difficult to offer as a general service since the internals of tenant virtual networks are usually not visible to the data center operators.

In summary, *switch multitenancy is not supported today, but could have significantly extended the range of applications accelerated on switches in data centers and clouds*.

2.1 Challenges

Present ASIC-based programmable switches lack the necessary mechanisms for resource, performance, and state isolation to support the concurrent execution of multiple applications from different tenants. Their programs expect to run assuming unmediated access to all hardware resources, and existing platforms do not provide any kind of privilege separation necessary to isolate tenant applications from the infrastructure packet processing logic. Any bug in a program may disrupt the stability of all the network traffic. Therefore, network operators might be reluctant to use the solutions that merge P4 programs of multiple clients into a single one to be executed on the switch [8,42,51,59].

Another challenge stems from the need to support thousands of tenants with their own in-switch programs. Existing compile-based solutions that merge multiple P4 programs, or allow runtime reconfigurability in hardware [16,50,54], allow co-residency of about a dozen of programs, which is a few orders of magnitude less than needed.

Last, allowing a program to be dynamically updated without disrupting the rest of the network traffic is difficult. Recent works have introduced runtime reconfigurability by extending the dRMT [54] and RMT [50] architectures. The dRMT extensions are indeed feasible on existing CPU-based switches, but the RMT modifications are more invasive as they require hardware changes. Our goal is to achieve runtime reconfigurability without architectural changes.

2.2 Packet Filters for Switches

The primary tenet of multi-tenancy is per-tenant isolation of state, faults, and performance. Language Virtual Machines (LVM), such as the Java Virtual Machine (JVM), can achieve the first two. Such virtual machines enable full sandboxing of the untrusted tenant code, with fine-grain control over the accessible hardware resources. Furthermore, it may help solve the current scalability issue by enabling *switch compute resource sharing* among the programs, by loading and unloading the relevant bytecode at runtime, without provisioning the actual hardware resources at compile time for all co-resident programs as has been done till now. Last, if such an LVM could run at line-rate on an RMT, switch, it would be possible to provide perfect performance isolation among the tenants, as the computing logic would not constitute the switch bottleneck.

Implementing a general-purpose high-performance VM in RMT systems is clearly unrealistic. Fortunately, however, many data-plane programs do not require such a VM, and involve relatively simple, more restricted logic. This observation is not new, as it served the designers of the popular Packet Filters [37] (BPF) virtual machine to allow the execution of untrusted code in the OS kernel. Drawing inspiration from BPFs, we seek to build an LVM to be executed on a switch, while potentially restricting the scope of programs that it can run efficiently and deviating from the standard P4 programming model.

Last, by implementing our design in P4, we intend it to be modular and flexible, serving as the framework for implementing a broader set of in-switch functions tailored to a particular network environment.

2.3 Target Switch Architectures

Switch architectures with data-plane programmability range from ASIC pipelines (i.e., Intel Tofino Reconfigurable Match-action Table, RMT) and FPGAs to manycore CPU processors (i.e., NVIDIA Spectrum-3 disaggregated RMT, dRMT), as well as a variety of hybrid designs. Among these, Tofino switches have been prominent in in-network computing research for the past few years. This success can be attributed to the ability to execute stateful packet processing logic, defined using the P4 language, while maintaining line-rate performance on a multi-Tbps switch. These capabilities lead us to focus on the Tofino RMT architecture as the target for SwitchVM design.

We believe, however, that the concept of running in-network programs in a Language Virtual Machine sandbox to offer scalable multi-tenancy in resource-constrained switches is not limited to RMT architecture. Yet, using other architectures may involve different design considerations. For instance, CPU-based switches may benefit from Just-in-Time compilation to reduce virtualization overhead. We leave the exploration of other architectures for future research.

3 Design

SwitchVM enables secure and isolated per-tenant execution of short programs, called *Data Plane Filters* (DPFs). DPFs are invoked for each packet arriving at the switch. The packets include the input and the reference/code of the DPF to invoke. DPFs are executed inside a sandboxed environment controlled by the operator. We first discuss the DPF programming model, and then explain the SwitchVM design.

Instructions	Description	Instructions	Description
Data Movement		ALU Operations	
`MOV`	R1=R2	`ADD SUB`	A = A op B
`LOAD_IMM`	B=imm	`AND OR XOR`	
`LOAD_CONST`	B=const_tbl[imm]	`LSH RSH NEG`	A = op(A)
`HASH`	A=hash(A)	`MIN MAX`	A = op(A, B)
Memory Operations		Control Flow	
`LOAD`	B = mem[A]	`HALT`	Goto END
`STORE`	mem[A] = B	`JMP`	Goto pc
`FAADD FAOR` `FAAND FAMAX`	t=Mem[A] Mem[A]=op(t, B) B=t	`BEQ BSET BLT` `BGT BLTS` `BGTS`	Goto A==B ? pc_taken : pc_not_taken
Prolog		Epilog	
`POP`	R=stack.pop()	`PUSH`	stack.push(R)
`PEEK`	R=stack.peek()	`FWD`	fwd according to R
`LOAD_MD`	R=MD[md_idx]	`NEXT_PC`	pkt.next_pc = R
`RAND`	R=rng.get()	`HDR_MOD`	drop DPF from pkt

Table 1: SwitchVM Instruction Set with representative instructions of each type. A/B are the register types, R can be either, other values are immediates.

3.1 Programming Model

A DPF comprises three sections. A *prolog* specifies the location of the input parameters and initializes the DPF execution *registers*. The registers can be initialized with the data from the packet, or with in-switch metadata, such as the size of the local queue or the packet timestamp. A *body* specifies the actual data-plane logic using the SwitchVM instruction set. An *epilog* determines what to do with the packet after the DPF completes, e.g., send it to a new destination based on the DPF output. It may also store the DPF outputs in the packet.

In-packet state. The data inside the packet is handled as a stack, similar to Tiny Packet Programs [22]. It can be read in the prolog and written in the epilog. This is convenient: when a packet traverses multiple switches, each pops the inputs and pushes the output back to the stack, simplifying the implementation of multi-switch applications. A variable-size stack allows each application to allocate precisely the required packet space, potentially growing or shrinking a packet during processing, using a single unified SwitchVM parser.

Body. The code is organized into a sequence of *Execution Stages* (ES), each comprising several instructions arranged in *lanes*. The instructions in an ES are executed in parallel, effectively implementing a Multiple-Instructions-Multiple-Data (MIMD) program. Intra-lane parallelism is also available: each lane may concurrently execute instructions of different types. There are three types of instructions: ALU operations, in-switch memory access and control flow branches.

Instructions use *virtual registers* shared across the execution stages. Thus, DPF computation can be seen as the processing of information in the registers while it is flowing through the pipeline of DPF ESes. There are two sets of registers, *A* and *B*, and certain restrictions apply to their usage, e.g., only *A* registers may be used as addresses for memory access instructions. Memory is *virtualized* to provide inter-tenant isolation.

DPFs inherit the constraints of the RMT pipeline. Therefore, a memory buffer is accessible only to a specific ES and lane, since a packet may access each memory location only once.

Instruction set. The SwitchVM instruction set is summarized in Table 1. Some instructions have restrictions. First, they might support specific register types (either *A*, *B*, or both). Second, some instructions are limited to the prolog/epilog DPF sections only: register initialization is restricted to the prolog, while packet steering, program-counter modification, and stack enlargement must be performed at the epilog. Other instructions are not limited to specific DPF sections.

Turing completeness. Turing completeness of a computational system implies that it is universal, i.e., it can perform arbitrary computations. A system is Turing-complete if it can access arbitrary memory locations and read/write any amount of data, and perform conditional jumps [17]. SwitchVM instruction set satisfies these requirements.

DPF size. The number of instructions in a DPF is constrained by the availability of hardware resources. If the application code does not fit in a single DPF, dynamic *chaining* can be used whereby one DPF may choose to invoke any other DPF, enabling them to be automatically invoked one after the other. The sequence of DPFs may run either on the same switch (via recirculation) or on another one. We note that recirculation might affect the performance isolation, so we strive to enable the execution of complex DPFs in a single switch pass. For non-RMT switches, DPFs can be sized for line-rate processing to achieve performance isolation in the common case. Then, instead of recirculation, we can gradually increase DPF sizes to accommodate larger applications, as long as performance isolation is maintained.

Atomicity execution semantics. DPFs naturally inherit the per-packet atomicity execution semantics offered by the RMT architecture. Specifically, the intermediate state updates done by a packet are not visible to other packets that invoke the same DPF on the same switch pipeline. Moreover, the state shared among multiple DPFs of the same tenant is also updated atomically, significantly simplifying the program development. Multi-pipe switches, such as Tofino, comprise multiple packet processing pipelines, each designated to serve a subset of ports. Switch allocation to run DPFs in a data center (§3.8) must consider the switch configuration to guarantee atomicity for a particular DPF.

Example: in-switch counter. Assume that several clients mark their outgoing packets with a unique application ID $\in \{1, 2, \ldots, N\}$, and wish to count the total number of packets in the network per ID. One possible implementation of this logic in a DPF is as follows. The DPF maintains one counter per ID allocated to lane 0 of EU 0. The ID is used as a memory address of the counter. The prolog `POP`-s the *ID* from the stack head in the packet into register A[0]. Then, EU 0 loads constant 1 (`LOAD_IMM`) into register B[0], and invokes

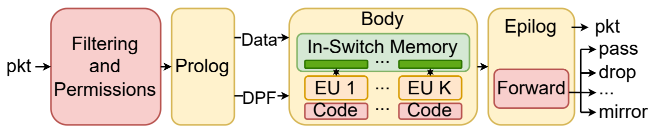


Figure 1: SwitchVM data-plane processing pipeline.

a *fetch-and-add* (FAADD) operation in that lane to update the counter. The epilog then can PUSH back the ID back onto the stack along with the counter value. Memory allocation for the counters, authorization to execute the DPF, and the actual placement of the DPF are executed by the control plane in the setup stage, as explained in §3.8.

3.2 SwitchVM Design

Figure 1 shows the high-level design of the SwitchVM processing pipeline. It comprises several fixed function modules: Filtering and Permissions, Prolog, and Epilog, as well as a variable number of *Execution Units, (EUs)*. Each EU has L lanes which is analogous to an execution unit of a Very Large Instruction Word (VLIW) processor. Each Execution Stage of a DPF is mapped onto its own EU. The instruction scheduling is static, i.e., a DPF specifies which instructions must be scheduled on which EU and each lane. The maximum number of EUs determines the maximum number of instructions in the DPF, and depends on the number of available hardware stages in the switch. The maximum number of lanes is determined by the number of match-action tables that can applied in parallel at a single hardware stage. For example, we can place 4 and 8 EUs in Tofino-1 and Tofino-2 respectively.

Packet processing flow. The packet is first passed to the Permissions module which determines whether a tenant may invoke a DPF. The Prolog then prepares the inputs of the first EU. The EUs then execute the DPF body. Finally, the Epilog prepares the output packet, and the Steering executes the forwarding logic.

3.3 Filtering and Permissions

The permission module is not visible to an executing DPF. It is managed by a privileged operator.

When a packet first enters the pipeline, its trusted tenant_id is matched against the permissions table. This table is programmed by the operator for each tenant that wishes to run a DPF. The table contains the action to perform on a packet that attempts to invoke a DPF without authorization (i.e., drop or pass), and also may specify the default DPF to invoke on any packet from that tenant. The table also stores a bit vector that controls tenant's access to certain functions such as access to the switch metadata.

In addition, a user may want to invoke only a subset of DPFs on a subset of switches. She specifies a special app_id token when asking for the DPF authorization. This token is stored in the Permission table as well. It must match the app_id token in the packet to execute a DPF. We use this functionality in our KVS cache application (See §4).

3.4 Program Loading and Initialization

The prolog module is responsible for initializing the virtual registers ($2L$, where L denotes the number of lanes) used as inputs for the first EU. The initial values can come from three sources: the in-packet stack (POP,PEEK), switch metadata (LOAD_MD), or from a random number generator (RAND). These instructions are the only ones available in Prolog. Initializing registers from the packet stack cannot lead to underflow since the data stack size is known. The switch metadata may include ingress/egress ports, timestamps, or a switch identifier. Execution of LOAD_MD can be restricted to specific tenants to keep the network topology hidden for the rest. Access to predefined protocol fields, e.g., TCP/IP headers of inner packet, could also be supported in the same manner, assuming it aligns with the operator's sandboxing policy.

The code for each segment (prolog, execution units, or epilog) may reside in a packet, or in the SwitchVM code memory in the switch. A *pointer* to the prolog is taken from the packet header, or is passed from the Permission module, effectively enforcing invocation of a default DPF for the current tenant on *all* the packets even without the SwitchVM header. The prolog transfers the execution to the first EU, by specifying the pointer to the first EU's instruction.

3.5 Epilog and Steering

The epilog pushes the results of the computations into the in-packet stack (up to $2L$ entries). It works similarly to Prolog. Specifically, the only operation that can be executed is PUSH, using any register as its input.

The epilog code defines how to steer the packet. This is determined by a dedicated immediate value in the epilog, or from any of the $2L$ registers. This value is then used to fetch a matching entry from a forwarding table defined for the specific tenant_id. We currently support the following actions (but more can be added): pass, drop, return-to-sender, forward, multicast, set_port, recirculate.

The default action is pass. The set_port and recirculate actions are reserved only for privileged tenants. Since the network uses encapsulation, some of these actions can also affect the headers of the encapsulated packet.

Updates to the steering table require the use of privileged control-plane requests to allow secure handling of virtual network address translations without disclosing the physical IPs to the tenant as we explain below.

A DPF may *drop* the code header after execution, saving bandwidth if there is no need to keep it.

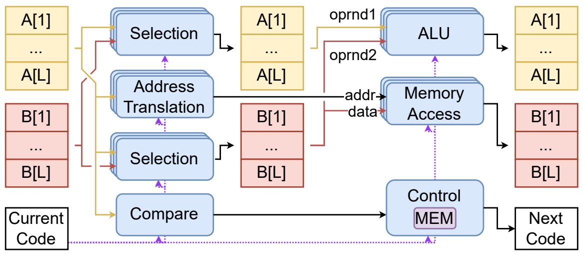


Figure 2: Execution Unit (EU) architecture.

3.6 Execution Unit

The architecture of a single EU is shown in Figure 2. An EU comprises L lanes. For each lane, each EU holds several fixed-function modules: Selectors that determine how to move data among registers, an ALU (one per lane), an in-switch memory access (one per lane), and a branching unit (one per EU) that determines the instruction pointer for the next EU. These modules are divided between two switch pipeline stages, and all modules in the same stage are executed in parallel.

Registers. An EU operates on $2L$ 32-bit registers, two for each lane. These are divided into 2 groups: $A[1..L]$ and $B[1..L]$. Due to resource constraints, the registers from different groups have minor usage restrictions. Specifically, only A registers are allowed to compute a hash of their previous value, and only B registers may store an immediate or a constant. These constraints are insignificant in practice but allow reducing required switch resources per EU, specifically the action instruction memory and hashing logic, as shown in §5.1.

The Selector module allows transferring data among arbitrary registers via `MOV` instruction. This is crucial because ALUs are constrained to execute operations only on the registers belonging to the same lane, as explained next.

ALU. Each ALU only performs 32-bit operations on registers from its lane, i.e., ALU i operates on $A[i]$ and $B[i]$. The supported set of operations can be easily extended (§3.7).

3.6.1 Memory Access

A DPF invokes `LOAD`, `STORE` and a few atomic operations to access in-switch memory buffers. These buffers are separately allocated for each EU, and each lane in the EU, without the ability to share them among EU instructions. As mentioned earlier, this limitation is due to Tofino RMT implementation (see §3.1).

DPFs may only access *virtual memory* to isolate memory between tenants. The memory is allocated by the control-plane API. Memory accesses involve virtual-to-physical address translation as we explain next. Once translated, the respective memory instruction is executed. The address operand in lane i is provided in $A[i]$, and $B[i]$ is used for the stored or loaded data.

Virtual address translation. SwitchVM supports two types of translation mechanisms. A segment-based memory translation allows allocating power-of-two-sized contiguous buffers. The allocation request specifies the requested virtual address and the buffer size. The control plane then allocates a contiguous range in physical memory, and uses TCAM to store the virtual address range as a key, and the numerical offset between the virtual and physical addresses as a value. The virtual address needs to be aligned to the segment's size.

When attempting to translate any address in this buffer, that address will match the allocated range and TCAM will retrieve the offset, allowing one to compute the physical address by adding that offset to the virtual address. For example, given the virtual address `0x4` and the request to allocate 4 bytes, the control plane may allocate a physical buffer of that size at physical `0x2`. The TCAM will store the range `0x4-0x7` as the key (using the mask 0xC), and the offset `-2` as the value. When accessing virtual `0x6`, the respective physical is computed as `0x6-2`. Note that if a virtual address is not mapped, it will not be found in the translation table, so no buffer overflows are possible.

However, for the cases where we need to store sparse data structures such as hash tables, the segmented allocation is wasteful. Using many small segments depletes the TCAM storage (about 1500 ranges per lane), whereas large ranges waste too much memory due to internal fragmentation.

In applications where the data to store is known in advance, one can create one-to-one virtual addresses mappings we call *direct*. These mappings are stored in a table with up-to 45K entries per lane. One notable example of using direct mappings is a KVS cache (see §4).

Memory Isolation. The entries in the translation tables are all amended with `tenant_id` as another key. Thus, each tenant may access only its own translation table.

3.6.2 Control Flow

Branching is essential to support realistic DPFs. Each EU contains a single Control Flow Unit (CFU) that implements different types of branches. Thus, DPFs specify the addresses of the instructions in *code memory*, or from the packet.

Each stage in a DPF, including prolog, EUs and epilog, may alter the control flow and choose the instructions invoked in the next stage. The CFU may use any register to compute the predicate and determine whether a branch is taken or not. The code specifies the instruction pointers for both taken and not-taken branches. SwitchVM supports multiple types of conditional and unconditional branches.

Code memory. To allow code sharing among DPFs, code memory is not virtualized and provides no memory protection. The security is not compromised, because the DPF memory isolation, permissions, and steering are enforced at runtime. Thus, even though a DPF code is shared, each DPF instance operates on a separate, per-tenant state.

The code addresses used in each stage point to code in the following stage. For example, the addresses in the Prolog are interpreted as addresses for the first EU, addresses in first EU point to instructions in the second EU, and so on.

The use of instruction pointers per stage is useful for other purposes too. First, it allows reusing code in switch memory by storing an `init_pc` that points to the prolog code that needs to be executed. This pointer is stored either in the packet (to invoke a specific DPF), or in the Permission table of a switch to serve as a default DPF. This enables atomic reconfiguration when replacing an in-switch DPF in-place, by replacing the prolog only after all other stages of a DPF have been stored in the switch. We demonstrate in-place replacement in §6.2. Second, in the epilog, the code has the ability to change the `init_pc` field in the SwitchVM header, which allows *service chaining* whereby one DPF invokes the next one in the next supporting switch. We show several examples of such applications in §4.

3.7 P4 Extensions

We envision SwitchVM to serve as an extensible framework that can be tailored to the needs of the specific environment by minor modifications to its P4 implementation. Thanks to its modular design, the modifications are localized. These additions will extend SwitchVM's capabilities and can be invoked from DPFs.

SwitchVM parser can be easily modified to expose new packet fields to a DPF and use them in the prolog to initialize DPF registers. Similarly, it is possible to expose additional switch metadata, such as port queue length. The steering unit can also be updated with new steering policies. Further, new instructions can be easily added to ALUs and memory units if the hardware support is available. For example, one may add support for bit slicing, or add new Read-Modify-Write operations. More elaborate changes may involve new fixed-function modules added to an EU. The code for these modules can be integrated into an EU, and contain the control signals needed to invoke them. For example, one may add a match-action table that can match on multiple registers and access memory from different lanes. Last, one may replace individual EUs with fix-function units, e.g., complex data-plane sketch.

3.8 Control Plane

Control plane is used for managing authorization, deployment, configuration, and memory allocation in switches. Our current design assumes that this functionality is implemented in a centralized controller managed by data center operators.

Authorization and Scheduling. DPF placement for in-network computing must be taken into account by the data center's resource management and scheduling system [3–5]. A tenant may ask to invoke specific DPFs (or a DPF chain) on certain switches along the data path of its virtual machines. To prevent exposing the physical network topology to tenants, authorization requests are expressed relative to a tenant's virtual network structure. Resource allocation should consider both resource utilization and the network topology in order to ensure consistent processing of tenant's DPFs across all packets, in accordance with the packet routing policies. While these aspects are beyond the scope of this paper, SwitchVM offers flexible *mechanisms* to control the DPF execution (§3.3).

In-switch memory management. The mechanisms are similar to the OS memory allocation. We implement a simple first-fit allocation policy. Unlike the traditional allocators, however, the tenant chooses the virtual addresses.

In-switch code deployment. Deploying a DPF involves several steps. When a DPF does not invoke the code from other DPFs (simplest case), the branch addresses it uses internally can be automatically inferred during the deployment. The physical address of the prolog is then reported to the user to be specified in the packets. However, reusing in-switch code requires knowledge of the addresses of the deployed code, requiring a process analogous to linking, which currently is performed manually.

Steering rules. The tenants use virtual networks with their private IP addresses, thus they are unaware of the physical addresses used to forward packets in a physical network. Thus, updates to DPF steering rules require translation from the virtual to physical IP addresses. The physical addresses are not visible to the DPF. DPF chaining is already considered at the authorization time, thus its effect on routing is acceptable to the operator.

Switch state migration and replication. In-switch state becomes an integral part of the tenant's application logic. Thus, migration of a VM to a different location in the network topology might require one to also migrate the state in the respective switches. Multi-path forwarding in modern networks might require maintaining consistent program state across multiple switches. Data-plane inter-switch replication solutions [56] can be deployed by data center operators to automatically synchronize the tenant's state across switches.

3.9 Security

Tenant identifier. Ensuring isolation among tenants relies on a trusted `tenant_id`, as it is used to distinguish tenants in all the security-critical units in SwitchVM, such as steering and virtual memory mappings. We use virtual network identifier fields, e.g., VID or VXLAN ID, common to encapsulation protocols, similar to prior research [29, 47, 50, 51].

DPF injection. Adding the DPF code or data to the packets requires privileged access to the encapsulation protocol headers via a hypervisor, which is expensive. One option is to add the DPF invocation requests in the control path (e.g., by attaching a function to an OS socket [15]), while adding the respective headers during encapsulation. Another option that we leave for future work is to modify the hypervisor networking in-

Application	DPF	Logic	Memory Mapping	Packet State	Steering	Chaining
KVS Cache [24]	Cache-put/get/update	Math	Direct,Segment	In/Out	Return-to-sender	✓
	Count-min sketch [10]	Hash,Math	Segment	In/Out	Optional	
Load Balancer	LB1 (Beamer [41])	Hash	Segment	Out	Dest. array	
	LB2 (Batch)	Math	Direct		Dest. array	
	LB3 (RackSched [60])	Rand	Segment		Dest. array	
Paxos [11]	Leader	Math	Direct	Out	Mulitcast	✓
	Acceptor	Math	Segment	In/Out	Mulitcast,Drop	✓

Table 2: Applications implemented using DPFs and the features they use.

terface to specify DPF invocation requests per packet. The in-packet data stack size needs to be validated at this point to prevent under-/overflow during packet parsing and deparsing.

Steering, chaining and routing. By default, DPFs cannot change packet steering nor routing, unless specifically authorized via dedicated steering rules installed by the control plane. These rules are tenant-specific, and they ensure that the tenant packets cannot escape the virtual network. The same mechanism applies to DPF chaining. In summary, neither steering nor chaining can affect the packet's original network path without control plane authorization, which only installs rules compliant with the network routing policies.

Unauthorized operations. Unauthorized access to memory and the attempt to invoke non-installed steering rules is identified by SwitchVM and can be reported to a control plane for further handling.

Denial-of-service with recirculation. Packet recirculation is not allowed by default, and can be limited to specific applications. Further, per-tenant rate limiting can be employed to guarantee performance isolation between recirculated packets of different tenants. In theory, packets can be recirculated indefinitely, as expected from a Turing-complete system. This can be prevented using per-packet recirculation counters, akin to the concept of *gas* in blockchain smart contracts [7, 53], or by using in-packet TTL counters.

4 Applications

We explain several complex applications we implement using DPFs (Table 2), and then discuss other use-cases supported by SwitchVM.

4.1 Key-Value Store Cache

We implement an in-switch KVS cache, similar to NetCache [24] with keys and values of 4B and 12B, respectively.

Frequently accessed keys are cached and returned upon GET requests, while PUT requests invalidate the respective cached entries. The installation and eviction of the entries are performed by the control plane and orchestrated by the auxiliary mapping data stored in the server using UPDATE operations. The cache evictions are guided by hit-counters and a count-min sketch [10] for frequently accessed keys.

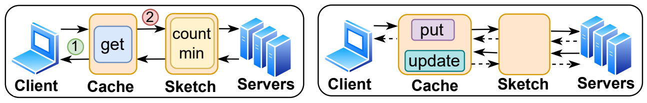


Figure 3: In-swtich key-value store cache implemented using SwitchVM. ① and ② show the flow of cache hit and miss respectively. Cache-get/put/update and count-min represent DPFs and the pipeline they are executed on.

Figure 3 summarizes the GET, PUT, and UPDATE operations. There are four DPFs (in boxes) in the application.

We implement a chain of two DPFs on the miss path. Specifically, cache-get invokes count-min in case of a miss. The DPFs are invoked in two pipes of the same switch, as our prototype only runs on the ingress pipe. These DPFs make use of most SwitchVM functions as explained next.

Data structures. Denote the cache size as k. We use direct mapping for k memory elements in EU0 for mapping keys to <valid-bit, index> tuples. The valid bit is reset when an entry is invalidated. The index represents the virtual address where the value for that key is stored. Additionally, a *null* segment mapping maps all misses to a single invalid tuple. Values are stored in 3 k-large segment-mapped arrays in EU2 using 4B per entry, and a k-large array in EU3 for the hit counters.

Cache-get. The prolog `POP`s the requested key from the stack and EU0 fetches the corresponding <valid-bit, index> tuple. A `BSET` (branch-if-set) in EU1 computes `AND` between an MSB one-hot mask (MSB set) and the tuple's valid-bit and accesses the tuple's index upon a hit. On a hit, EU2 `LOAD`s the 12B value, and EU3 atomically increments the hit counter using a `FA-ADD` instruction. The epilog `PUSH`s the key and value onto the stack, executes a *return-to-sender* steering policy, and drops the code section. On a miss, the epilog `PUSH`s the key onto the stack and changes the DPF pointer in the packet code header to point to the Count-Min DPF, and the packet continues to the next switch pipe.

Cache-put. The prolog `POP`s the requested key from the stack and EU0 performs a `FA-AND` operation with an MSB one-cold (all but LSB set) mask to mark the entry as invalid. Invalidating a non-cached key is okay, since it is mapped to an invalid entry anyway. The epilog `PUSH`s the key onto the packet stack, and changes the code pointer to a default *empty program* entry.

The packet then continues to the next pipe which passes it as is.

Cache-update. The server maintains the mappings of valid keys and their corresponding tuple indices in the switch cache. Upon the installation of a new key, the server sends the index of a free tuple to the switch, alongside the key and the 12B value. The prolog `POP`s these values from the stack, EU0 maps the key to the new tuple, and marks the tuple as valid using an `FA-OR` operation with an MSB one-hot mask. EU2 `STORE`s the 12B value and the epilog executes a *return-to-sender* operation and returns the packet as an ACK to the server.

Count-min sketch. The prolog extracts the key into first three lanes using two `PEEK`s and a single `POP` operations. EU0 `HASH`es these lanes using three different hashing polynomials, and `AND`s the results to acquire only the $\log k$ least significant bits. EU1 use the results as indices for atomically incrementing three counters using `FA-ADD` instructions. EU2 and EU3 extract the `MIN` value from all three counters. The epilog `PUSH`es the key and the frequency back into the packet.

Selective execution. This application demonstrates the use of `app_id` token for selective DPF execution. The server uses the **cache-update** DPF, which executes at the cache switch. The DPF needs to pass through the sketch switch, but it should not be executed there. However, the *tenant_id* is already configured there for allowing the *count-min* DPF to execute. Thus, we use a separate `app_id` token for the cache-update DPF and configure it only in the caching switch. We use a different `app_id` for all other DPFs.

Comparison to NetCache. The key/value bitwidth is limited in SwitchVM, due to the small number of lanes. Therefore, the amount of data that can be read from the packet is limited too. The number of entries in the cache is also limited because we divide the memory between VLIW lanes equally. Consequently, while NetCache supports up to 64K entries, 16B keys, and 128B values, SwitchVM can only support up to 45K entries, 4B keys, and 12B values.

4.2 Load Balancer

We implement three switch-accelerated load balancers.

Load-agnostic LBs. We consider two policies. LB1 splits the traffic by hashing a user-defined value, as previously suggested by Beamer [41]. LB2 schedules batches of 1024 packets to each server using round-robin.

Load-aware LB (LB3). Traffic is split according to load on the servers. Similar to RackSched [60], requests are forwarded to the least loaded server using an in-switch scheduler that implements a power-of-two-choices selection. Servers send their loads by piggybacking on existing traffic (Figure 4).

For lack of space, we only explain the details of LB3 DPFs.

LB3 setup. Denote the number of servers as 2^m. We store two copies of the server loads in two segment-mapped arrays of size 2^m, each assigned to a different lane of EU1. Updates

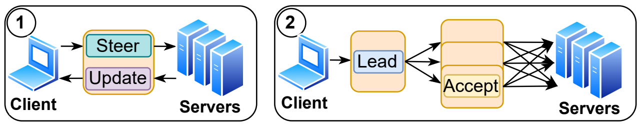


Figure 4: ① Load-aware load balancing. ② In-network Paxos acceleration with one leader switch and three acceptors.

are assured to be consistent between the two arrays due to the RMT per-packet atomicity guarantees.

Client requests. The prolog `RAND`-omizes two registers. EU0 `AND`-s each register with a mask to achieve the m least significant bits per register. The results indicate the indices of two servers. Next, EU1 `LOAD`-s the values of the server-reported loads, and performs a conditional branch using `BGT`. The selected packet steering policy is based on the branch result.

Server responses. The prolog extracts the server index and updates two registers using two `PEEK`s and two `POP`s. EU1 uses these indices to `STORE` the new load into the corresponding array entries. The packets are not modified.

Comparison to RackSched/Beamer. LB1 and LB3 functionalities are identical to those of Beamer [41] and RackSched [60]. RackSched uses a different memory for each server load, allowing it to use a single copy. This, however, limits the number of supported servers to a few dozens.

4.3 Paxos

Paxos [30] is a consensus protocol in which clients agree on one *proposed value*. Figure 4 shows how we use DPFs for an in-network acceleration of Paxos, similar to P4xos [11]. The leader is similar to NoPaxos [32]. Due to space limitations, we provide the DPF descriptions in Appendix A.

Comparison to P4xos and NoPaxos. The leader is functionally identical to NoPaxos. Constraints similar to the ones presented for the KVS application (§4.1) limit the values to 4B, compared to 32B in P4xos.

4.4 Other Applications

Tiny Packet Programs. SwitchVM extends Tiny Packet Programs (TPP) [22] with the ability to do complex computations instead of a handful of simple instructions. This allows, for example, aggregation of the maximal hop latency along the path instead of collecting the samples into the packet.

Distributed coordination. A small amount of stateful memory can be used for implementing several distributed latency-sensitive coordination tasks, including leader election, aggregation, and distributed barriers.

NetChain. NetChain [23] performs in-network coordination between switches. It requires similar functionalities as in Paxos and NetCache, both already implemented. Source routing can be performed using the in-packet data stack.

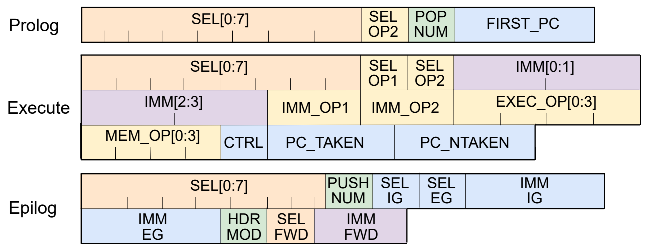


Figure 5: Program encoding for prolog, execution units and epilog.

Complex functions. Turing-completeness allows building arbitrary functions using packet recirculation and the code-pointer changing mechanism. In fact, SwitchVM's RISC-like ISA makes the implementation of data-plane programs simpler than in native P4. For example, one can implement bubble sort in a few DPFs, i.e., element swapping DPF, comparison DPF, and an iterator DPF.

Fast-path/slow-path. By adding mirroring capabilities to the steering module, complex data-plane operations can split the traffic to fast-/slow-paths. For example, all packets execute a DPF that selects which packets to mirror at egress while changing their code pointer to a slow-path DPF, which may be a complex Turing-complete program.

5 Implementation

Our prototype targets Intel Tofino [20] and has four EUs, each with four lanes. Each lane has up to 45K 32-bit memory entries and 1.5K segment mappings. The first lane of each EU has additional direct mappings, possibly to all registers. Overall, each DPF can execute up-to 16 ALU and 16 Memory operations. Permissions and forwarding tables have up to 32K entries each. It is possible to store up-to 4K in-switch instructions of each type. Assuming each DPF uses at most a single two-way branch, as is the case with our applications, the switch can store up-to 2K entirely distinct DPFs. These can be shared among tenants thanks to virtual memory and instruction pointers. We highlight only the most interesting aspects of the implementation for the lack of space.

Packet structure. DPFs are stored in an option field of the Geneve encapsulation protocol [21]. It includes parser hints, data used by SwitchVM, optional ingress and/or egress DPFs and a data stack. If the stack is small (up to 16 entries) it can be fully parsed with the encapsulated packet (up to layer 4).

Code header. The code header encodes the DPF program (Figure 5). All fields have a default *nop* operation to be ignored by the relevant unit. The *SEL* field determines where to read register values from. The *EXEC_OP* and *MEM_OP* fields dictate which operation should be executed by the ALU or Memory access. The *CTRL* field determines the DPF control flow. All SwitchVM components are implemented using

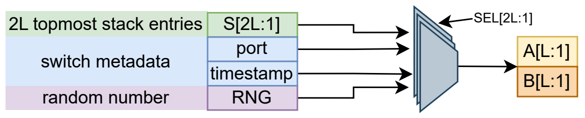


Figure 6: Prolog selection logic.

match-action tables. These tables effectively map the opcodes to the actual execution logic.

We use the prolog unit as an example to show how code in the header is translated into the actual execution. A unit with four lanes is built with 8 match-action tables we call selectors, and each initializes the value of a respective virtual register. The value *SEL[i]* in the prolog code section determines the source for initialization of the i-th register, i.e., the stack (`POP`,`PEEK`) the metadata (`LOAD_MD`) or `RAND`. These opcodes are used by the selector match-action table (Figure 6) to initialize the respective register. Semantically, the A registers are initialized first, and the registers with lower indices are initialized before those with higher ones.

Execution Unit. Each EU occupies two hardware pipeline stages (Figure 2). In the first it executes register selection, memory translation, and comparisons. The second runs ALU operations, memory access and branching. Selection is implemented similarly to the Prolog selection above, and it is used to shuffle the registers between different lanes, use them for hashing, and for loading immediate values. The ALUs are per-lane match-action tables with the key being the operation opcode, and the action simply performs the operation on the respective register pair.

Memory Translation Access. Direct and segmented mappings are done in first stage of the EU, for each lane. We match on both tables, but only at most one action is executed, giving priority to the direct mapping. In addition to the address at the lane's A registers, each table accepts the `tenant_id` as an additional matching key for isolation. The resulting physical address is used to access the per-lane Tofino register in the next stage, using the RegisterAction specified by the opcode.

Comparisons and Control Unit. In each EU, we can perform a single branch operation based on evaluating a predicate on two registers. In the first stage, we use comparison to compare two registers. In the second stage we use the result to select code for execution in the next EU. More details about the implementation of all branching operations are found in Appendix B.

5.1 Resource Usage

The key optimization goal is to reduce the number of hardware stages, hence our decision to use VLIW EUs. Permissions, prolog, epilog and steering each requires one hardware stage, whereas each execution unit takes up two stages. Since we have four execution units, SwitchVM uses all 12 stages of Tofino-1. We run a simple L3 switch, on the same pipeline,

Resource	Utilization
Exact match Xbar	24%
Ternary match Xbar	20%
TCAM	41%
SRAM	57%
Hash bit units	49%
Hash distribution units	72%
Gateways	17%
Action instructions	85%
Logical tables	73%
Normal PHV containers	64%

Table 3: SwitchVM resource utilization on Tofino-1 with 4 EUs and 4 lanes per EU.

for forwarding packets unaffected by SwitchVM.

Table 3 reports total resource consumption of the SwitchVM prototype. The primary constraint on expanding the EU size is the number of logical tables and available action instruction memory within each stage. Tofino's design is geared towards a small number of large tables capable of executing wide actions, i.e., ones that can modify many header fields simultaneously. In contrast, SwitchVM would benefit from a large number of small tables with narrow actions. Tofino-2 has 20 stages, which can be utilized for doubling the number of EUs from 4 to 8, implementing P4 extensions, or for handling standard data-center networking operations. We note that we were able to compile eight 2-way VLIW EUs for Tofino-2, but increasing the number of ways failed to compile due to a compiler bug.

As mentioned earlier, the switch can store a maximum of 2K distinct DPFs. These can be safely shared among tenants because they use virtual memory. However, certain resources that cannot be shared may limit the number of co-located tenants. These include the number of direct and segmented mappings in each EU, total memory per EU, and the overall number of forwarding rules.

Appendix C offers more details on the resource usage by the concrete DPFs we implemented in our prototype. The number of distinct co-resident DPFs of the same type depends on the resource usage of each DPF. For example, a single key-value cache can accommodate up to 45K keys, but the number of isolated key-value cache instances is capped at 2K, regardless of the cache size, because of the limit on the number segmented mappings. Similarly, the total memory capacity directly impacts the number of co-resident DPFs. On the other hand, some of the scalability limits can be increased via a more sophisticated implementation of SwitchVM, e.g., by implementing direct mappings on additional lanes.

This factor should be carefully considered, as highlighted in previous research [61].

Deployment. It is often desirable to combine the execution of SwitchVM with traditional networking functions, such as a regular switch. Indeed, we implement a simple L3 switch on the same pipeline. However, relatively high resource usage of SwitchVM may complicate colocation with resource-intensive P4 programs. To this end, we propose three practical deployment options: (1) Leveraging additional pipeline stages of Tofino-2, where 12 stages can be used for SwitchVM and the rest 8 stages for other data-plane functions. (2) In a multi-pipe switch, dedicating a separate pipe for SwitchVM at the cost of reduced bandwidth. (3) Employing SwitchVM as a discrete in-network computing appliance, in line with the existing proposals for stateful network function disaggregation [1].

5.2 Limitations

Design limitations. First, by choosing the language-level virtualization approach, SwitchVM introduces non-negligible resource overheads which increase the resource consumption and constrain the generality of the functions that can be implemented on top. We believe that this is a viable design point, however, as many in-network computing functions, e.g., eBPFs, require a fairly limited functionality. Second, SwitchVM is tailored for in-network computing and not for general data-plane programmability. This specialization facilitates implementing the security sandbox, which restricts the packet headers exposed to DPFs and constrains the effect of the DPFs on the packet network behavior. The downside of this design choice is that DPFs cannot be used to implement new network protocols, for example.

RMT-related constraints. First, to guarantee performance isolation, RMT-based switches require a DPF to fit within a single pipeline pass. This requirement places a hard upper limit on the DPF size. In contrast, CPU-based dRMT architectures could allow longer DPFs at the expense of gradual performance degradation. Second, RMT switches parse the entire packet prior to processing, necessitating the use of a uniform packet structure for all tenants. Since in SwitchVM the access to packet headers is virtualized, DPFs cannot access the headers, unless dedicated P4 extensions are incorporated during compile time, as discussed in §3.7.

Prototype. We do not yet have a compiler for DPFs, so they are implemented in assembly, using a Python framework for code construction. Further, SwitchVM cannot run on both egress and ingress pipelines due to the limitations of the egress parser in Tofino-1. Last, our control-plane APIs and the client API for using DPFs in applications are rather immature, and the Recirculation rate-limiting logic is not implemented.

6 Evaluation

We aim to highlight the following SwitchVM characteristics:

1. Strong performance isolation among tenants;
2. Runtime programmability behavior without interference when adding/removing tenants;
3. Low latency and bandwidth overheads;

4. End-to-end performance in applications equivalent to P4 baselines.

We emphasize that *SwitchVM was compiled once and never modified throughout all the experiments.*

6.1 Methodology

Setup. We use a dual-socket machine with Intel Xeon Silver 4216@2.1 GHz CPU with 188 GB of RAM and connect it to a 3.2Tbps Intel Tofino switch (EdgeCore Wedge 100BF-32X) via two two-port 100G NICS (Intel E810-C). Unless stated otherwise, the packet generator and the receiver server use different NIC ports and run on different sockets. Hyper-threading and power saving are disabled for consistent results. **Packet generator and receiver**. Microbenchmarks were performed using Cisco TRex traffic generator [9] (DPDK-based). End-to-end applications are implemented using DPDK. Unless stated otherwise, packets are encapsulated using the Geneve protocol [21] for simulating data center environments.

6.2 End-to-End Applications

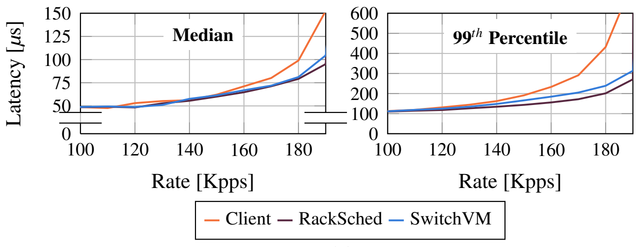
We show SwitchVM performance using the Load Balancer and KV store cache applications from §4.

6.2.1 Load-Aware Load Balancer

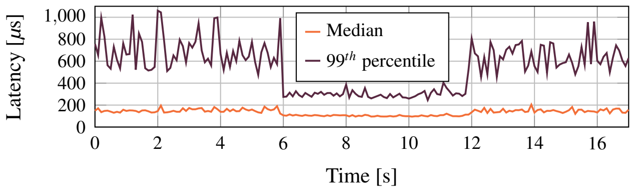
Our environment consists of two servers that execute the requests of a single client. We compare our implementation to the RackSched [60], a load-aware load balancer written in P4. We use the original RackSched implementation for clients, servers and the P4 scheduler. To run with DPFs we slightly modify the client and servers. Additionally, we add a dummy payload to RachSched original packets to mimic the bandwidth overheads of Geneve encapsulation protocol in SwitchVM, but without performing encapsulation itself to avoid intrusive P4 code changes.

As in RackSched, we run a synthetic workload with a bimodal request processing distribution (5μs and 50μs, for 90% and 10% of the requests respectively). We measure the median and the 99-percentile latency as a function of the load for three systems: the original RackSched, SwitchVM using in-switch DPF, and a client-based implementation that randomly selects a server for each request.

Figure 7a shows no latency difference compared to the P4-only baseline for up to 180K requests/s for the median latency, and up to 15% overheads in the 99% latency for higher rates. We believe, however, that the overhead is not related to the in-switch processing (which shows no sensitivity to load as we see in microbenchmarks), but is an artifact of the increased server load due to the Geneve encapsulation used in SwitchVM. Regardless of these artifacts, the use of DPFs improves the client-only implementation latency by 1.4× and 2.4× at maximum throughput.



(a)



(b)

Figure 7: A load-aware load balancer using SwitchVM. (a) SwitchVM demonstrates negligible end-to-end latency overheads compared to RackSched in P4 [60]. (b) SwitchVM changes the load-balancer scheduling policy at times $t = 6, 12$ without any request delays or packet drops.

In-place policy replacement. Unlike RackSched, SwitchVM can easily replace its scheduling policy without suffering from packet loss due to switch reconfiguration. We show this by dynamically modifying the scheduling policy from batch-round robin to the load-aware power-of-two-choices policy and back (§4). Figure 7b demonstrates the end-to-end median and the 99-th percentile latency as observed by the client. At $t = 6$ and $t = 12$ the policy is altered without the client having to stop. These results clearly demonstrate the power of SwitchVM to react to different network conditions with full application transparency.

6.2.2 In-Switch Cache for Key-Value Stores

We connect one client to a server and use two switch pipes, i.e., each pipe acts as an independent switch. SwitchVM runs a chain of two DPFs that act similarly to NetCache [24].

We demonstrate SwitchVM end-to-end latency for GET operations under abrupt changes in the key distribution (hot-in in [24]) and a constant TX throughput. The client sends 10K requests per-second (10 Kpps) while targeting a set of 128 keys that repeatedly changed every 10 seconds. The server periodically collects the most frequently accessed keys from a count-min sketch [10] implemented in DPF, then caches the 128 most frequent keys in the in-switch cache. Cache hits and sketch counters are cleared every 5 seconds.

Figure 8 reports the median and the 99th percentile of the requests' end-to-end latency. The periodic key changes induces cache misses that redirect the requests to the server, as evident from the momentary latency spikes.

This experiment shows the power of DPFs to implement

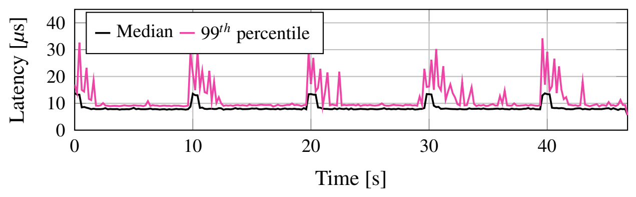


Figure 8: NetCache-like cache for key-value store implemented with DPFs. Latency spikes reflect cache misses due to an abrupt change in the hot-key working set. As the cache is re-populated with new hot keys the latency drops.

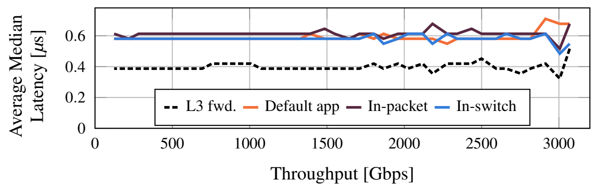


Figure 9: Throughput vs. per-port median latency using 1.5KB MTU-size packets. L3 fwd shows the smallest latency of a functional switch.

complex in-network accelerated applications.

6.3 Microbenchmarks

Similar to previous works [24], we simulate a fully loaded switch using a snake configuration, where two ports are connected to the traffic generator and to the receiver server while all remaining ports are connected to each other in pairs. Unless states otherwise, we use MTU-size packets (1500B).

Throughput vs. latency. We compare SwitchVM performance to a standard L3 forwarding application implemented in P4. SwitchVM invokes a *private counter* DPF that increments a counter stored in switch memory, similar to the leader DPF used in Paxos (§4).

We compare the performance of three ways to invoke a DPF: in-switch, in-packet, and in-switch default DPF that has no bandwidth overheads.

Direct measurements of per-port latency were too noisy due to sub-μs values. Instead, we calculate the average per-port latency as follows. First, we measure the median latency from the generator to the receiver after passing through the switch in the snake configuration. Next, we obtain the median latency from the generator to the receiver using a single pass, and subtract the two to cancel the latency of the server and client machines. We then divide the result by 31, which is the number of additional times the packet traverses the switch in the snake configuration.

Figure 9 shows that the switch approaches its maximum throughput (3.14Tbps) at about the same latency for all invocation techniques. As expected, L3 forwarding has lower latency, all three ways to invoke DPFs perform the same.

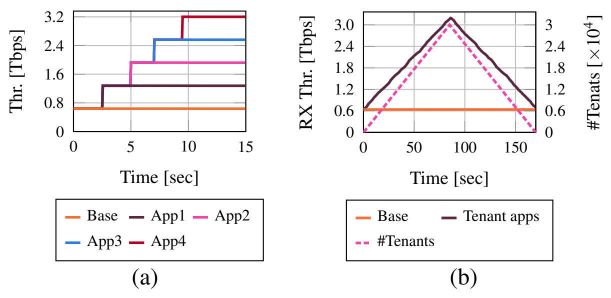


Figure 10: (a) SwitchVM demonstrates strong performance isolation between different DPFs and a base flow without DPF invocation. (b) Aggregate receive throughput vs. the number of tenants, each having its own DPF and a private switch state. Packets of unauthorized tenants are dropped. Base represents the background traffic not using DPFs.

Performance isolation. We demonstrate the performance isolation among tenants by running five network flows which together achieve the maximum aggregate switch bandwidth. Among these, there are four flows, each with its own DPF, and a base flow that does not invoke any. We measure the throughput per DPF at the receiver. We start with a single flow and gradually add more to see if there is any interference between them. No measurable interference was observed; neither among the DPFs nor with the other traffic (Figure 10a).

Scalability. We measure SwitchVM tenant reconfiguration performance by scaling the number of served tenants. We use the private counter DPF again, with in-packet code. Note that for a given amount of required resources, the DPF code has no effect on scalability, therefore this experiment is representative for the case when each tenant executes her own DPF (Figure 10a).

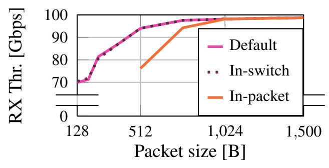
We start by dynamically adding tenants in batches of 350 tenants per second until reaching 30K tenants, then removing them at the same rate. The process invokes the control plane API to authorize execution of DPFs for the respective tenants, and allocates the private state in the switch (or deauthorizes and deallocates state upon tenant removal). For the purpose of this experiment, SwitchVM is configured to drop packets that attempt to run a DPF that has not been authorized.

The packet generator constantly sends packets for *all* the tenants at 80% line rate. The remaining bandwidth is occupied by packets that do not invoke any DPF. We measure the total bandwidth at the receiver. Initially, tenants are not authorized to run any DPF, so all the packets but those not invoking DPFs are dropped, as expected (Figure 10b). Once the tenants are authorized to run, the aggregate bandwidth grows until all the 30K tenants are authorized. Symmetrically, the tenants are gradually removed in the second part.

This experiment shows that SwitchVM scales to up to 30K tenants with strong performance isolation and without measurable interference. A similar experiment with in-switch code,

App	Header Size [B]
Default	0
In-switch	12
In-packet	192

(a)



(b)

Figure 11: (a) Packet overheads for DPFs. (b) Receive side throughput as a function of the packet size.

instead of in-packet, shows the same results. Note however that in this case the amount of *different* DPFs (shared across tenants) is limited to 2K, assuming each DPF has at most a single branch instruction. For comparison, compiler-based approaches for merging P4 programs [42,51,59], and runtime reconfigurable switches [54] can only scale to a handful of co-resident in-switch applications. Appendix C discusses the resource usage for the DPFs we implemented in more details.

DPF bandwidth overheads. Figure 11a reports the additional header size required for a representative DPF application. This overhead translates to a reduction in the effective bandwidth of the system (goodput) as a function of the packet size.

Figure 11b shows the maximal achievable throughput per packet size for different DPF invocation mechanisms. Naturally, the throughput is lower for smaller packets but eventually converges to 99% of the line-rate for all types, at MTU-size packets. Small in-packet DPFs impose packet parsing overheads that reduce the receive-side throughput.

7 Related Work

P4 virtualization. Several prior works focus on virtualization of the data-plane at the P4-level. HyPer4 [18] and HyperVDP [57] use a hypervisor P4 program that emulates other P4 programs, thus incurring significant overheads that limit these approaches to software and FPGA targets. SwitchVM offers a more efficient virtualized ISA that is optimized for in-network computing.

In-switch programming abstractions. Several works implement application-specific in-switch primitives that can be composed at runtime to implement certain in-switch applications. DIP [52] proposes in-switch primitives combined using in-packet recipes for implementing network layer protocols. NetRPC [58] offers primitives that can be composed via an in-switch recipe to implement a few popular in-network applications. Their primitives, however, are tailored to specific tasks. Moreover, their multitenancy is limited to a few dozens of concurrent applications of a few types. SwitchVM is more general and flexible, and scales to thousands of tenants executing arbitrary DPFs.

Compile-time merge. P4Visor [59], PRIME [42], SPEED [8] and [51] propose merging p4 programs before compilation. This has the potential of providing multitenancy, but is limited to only a few applications due to logic partitioning. In addition, changing the set of applications requires recompilation and switch reconfiguration and disrupts the switch traffic.

New architectures. Menshen [50] proposes an extension of the RMT architecture for data-plane multitenancy at the P4 level. Their solution adds a hardware indirection layer to allow running per-tenant packet processing logic on each packet, ensuring cross-tenant isolation using a specialized trusted compiler. SwitchVM also adds an indirection layer with its virtual ISA, with the key difference that it allows secure execution of untrusted DPFs thanks to runtime checks. P4VBox [45], MTPSA [47] and [29] propose new hardware architectures for data-plane virtualization. These architectures allow spatial partitioning of the resources between different hot-pluggable applications. However, they are currently limited to FPGA targets as they require hardware modifications, and they cannot scale beyond tens of programs. In contrast, SwitchVM runs on commodity switches and can scale to thousands of DPFs.

Runtime programmability. FlexCore [54] and IPSA [16], propose a runtime programmable dRMT architecture that allows changing functionality without causing traffic disruptions. These works do not discuss inter-tenant isolation. SwitchVM naturally offers runtime programmability since changing programs is equivalent to installing match-action entries. HW modification proposals [45,47,50] naturally integrate runtime programmability, a necessity for ensuring performance isolation across tenants.

In-network computing as a service. Several works discuss the isolation requirements [29,47,50,51] for allowing multi-tenancy of programmable switches in data centers. Runtime memory allocation was proposed in [51,61] to allow changing memory allocation between tenants. SwitchVM allows implementing such allocation policies due to the use of virtual memory.

Multi-core switches. Multicore switches, e.g., Juniper's Trio [39] and NVIDIA's Spectrum [40], follow a more conventional Von Neumann hardware architecture, rather than the dataflow type architecture of RMT switches. SwitchVM can be seen as a compatibility layer that allows architecture-independent network function development, but we leave adaptation of SwitchVM for such targets for future work. These architectures differ substantially from RMT, and therefore require different design considerations.

RMT switches feature a simple programming abstraction, where packets execute a sequence of match-action tables, such that each packet appears to execute to completion before the next packet processing starts. CPU-based switches on the other hand are more versatile, which potentially makes them better suited for complex in-network applications. In particular, they offer gradual performance degradation for longer programs, as opposed to the RMT pipelines that either fail to run or require coarse-grained packet recirculation. Additionally, CPU-based switches can potentially achieve higher scalability by utilizing a memory hierarchy, in contrast to the use of stage-local memory in RMT pipelines. However, achieving line-rate processing and performance isolation between co-resident programs on CPU-based switches is much more challenging, and cannot be generally guaranteed.

Deployment. Harmony [3] and HIRE [4] discuss the management implications and scheduling requirements of multitenant in-network computing. SwitchVM's scalability reduces the complexity of these management tasks as it makes migrating programs between switches more efficient.

Active Networks. The idea of embedding code into packets for in-network execution dates back to Active Networks [46,48,49]. Active networks allow users of a *shared* infrastructure to inject customized packet processing programs into network nodes, primarily driven by the desire to introduce new networking services on a per-user basis. SwitchVM can be viewed as a version of active networks for multi-tenant in-network computing within a data center. A key difference, however, is that SwitchVM establishes clear *privilege separation* between the network's operator and the network users, sidestepping the security and management issues associated with active networks. The in-packet state management we implement in SwitchVM is inspired by tiny packet programs [22]. TPPs do not consider multitenancy and isolation and are focused primarily on telemtery applications.

In a concurrent work, ActiveRMT [13, 14] proposes a capsule-based active networking approach with a shared runtime for interpreting small in-packet programs. Similarly to SwitchVM, ActiveRMT also enables per-packet reprogrammability, with the primary goal of improving in-switch memory utilization for multiple programs. Despite this similarity, ActiveRMT pursues different goals which further dictate different design choices, and make it less suitable for our purposes. First, the in-switch memory in ActiveRMT is segmented, rather than virtualized as in SwitchVM. As a result, in-switch program deployment and sharing available in SwitchVM cannot be easily supported, incurring bandwidth overheads due to in-packet code header and preventing service chaining natively supported in DPFs. Another implication is that, unlike in SwitchVM, in-switch memory reallocation requires updating all the tenants affected by it, impeding scalability in a multi-tenant system. Further, ActiveRMT design relies heavily on packet recirculations, which SwitchVM explicitly strives to minimize in order to enjoy atomic in-switch state update, reduce bandwidth overheads, and achieve strict performance isolation. Last, ActiveRMT does not support variable-size in-packet state, so it complicates its use in multi-switch applications such as in-network telemetry.

8 Conclusion

SwitchVM enables data center tenants to build in-network accelerated applications by securely deploying and executing Data Plane Filters (DPFs) on programmable switches. SwitchVM supports concurrent execution of thousands of DPFs while offering strong performance, state and fault isolation. We show SwitchVM's utility to build complex applications and experimentally demonstrate its performance and scalability. We envision that SwitchVM will open new opportunities for in-network accelerated applications in data centers.

Acknowledgments

TODO

References

[1] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating stateful network functions. In *USENIX NSDI*, 2023.

[2] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. PINT: probabilistic in-band network telemetry. In *ACM SIGCOMM*, 2020.

[3] Theophilus A. Benson. In-network compute: Considered armed and dangerous. In *ACM HotOS*, 2019.

[4] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. Switches for hire: Resource scheduling for data center in-network computing. In *ASPLOS*, 2021.

[5] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. Holistic resource scheduling for data center in-network computing. *IEEE/ACM TON*, 2022.

[6] Broadcom. Trident4 / BCM56880 series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, 2023.

[7] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[8] Xiang Chen, Hongyan Liu, Qun Huang, Peiqiao Wang, Dong Zhang, Haifeng Zhou, and Chunming Wu. SPEED: Resource-efficient and high-performance deployment for data plane programs. In *IEEE ICNP*, 2020.

[9] Cisco. TREX: Realistic traffic generator. https://trex-tgn.cisco.com/, 2023.

[10] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.

[11] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM TON*, 2020.

[12] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *ACM SOSR*, 2015.

[13] Rajdeep Das and Alex C. Snoeren. Enabling active networking on RMT hardware. In *ACM HotNets*, 2020.

[14] Rajdeep Das and Alex C Snoeren. Memory management in activermt: Towards runtime-programmable switches. In *ACM SIGCOMM*, 2023.

[15] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *USENIX ATC*, 2019.

[16] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *USENIX NSDI*, 2022.

[17] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science. Springer, 2010.

[18] David Hancock and Jacobus van der Merwe. HyPer4: Using P4 to virtualize the programmable data plane. In *ACM CoNEXT*, 2016.

[19] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*, 2017.

[20] Intel. Intel® tofino™ programmable ethernet switch asic. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, 2023.

[21] T. Sridhar J. Gross, I. Ganga. Geneve: Generic network virtualization encapsulation. https://www.rfc-editor.org/rfc/rfc8926.pdf, 2020.

[22] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *ACM SIGCOMM*, 2014.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-RTT coordination. USENIX NSDI, 2018.

[24] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. ACM SOSP, 2017.

[25] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable load balancing using programmable data planes. In *ACM SOSR*, 2016.

[26] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on P4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 2021.

[27] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[28] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: making RPCs first-class datacenter citizens. In *USENIX ATC*, 2019.

[29] Johannes Krude, Jaco Hofmann, Matthias Eichholz, Klaus Wehrle, Andreas Koch, and Mira Mezini. Online reprogrammable multi tenant switches. In *ACM ENCP*, 2019.

[30] Leslie Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, 2002.

[31] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using network multi-sequencing. In *ACM SOSP*, 2017.

[32] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*, 2016.

[33] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. USENIX OSDI, 2020.

[34] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better netflow for data centers. In *ACM NSDI*, 2016.

[35] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan

Yu. HPCC: High precision congestion control. In *ACM SIGCOMM*, 2019.

[36] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, 2016.

[37] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, 1993.

[38] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[39] Juniper Networks. MX series universal routing platforms. https://www.juniper.net/us/en/products/routers/mx-series.html, 2023.

[40] NVIDIA. NVIDIA spectrum-4. https://www.nvidia.com/en-us/networking/ethernet-switching, 2023.

[41] Vladimir Andrei Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *USENIX NSDI*, 2018.

[42] Ricardo Parizotto, Lucas Castanheira, Fernanda Bonetti, Anderson Santos, and Alberto Schaeffer-Filho. PRIME: Programming in-network modular extensions. In *IEEE NOMS*, 2020.

[43] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. ACM HotNets, 2017.

[44] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *USENIX NSDI*, 2021.

[45] Mateus Saquetti, Guilherme Bueno, Weverton Cordeiro, and Jose Rodrigo Azambuja. P4VBox: Enabling P4-based switch virtualization. *IEEE Communications Letters*, 2020.

[46] B. Schwartz, A.W. Jackson, W.T. Strayer, Wenyi Zhou, R.D. Rockwell, and C. Partridge. Smart packets for active networks. In *OPENARCH*, 1999.

[47] Radostin Stoyanov and Noa Zilberman. MTPSA: Multi-tenant programmable switches. In *ACM EuroP4*, 2020.

[48] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM CCR*, 2007.

[49] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. A survey of active network research. *IEEE Communications Magazine*, 1997.

[50] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation mechanisms for high-speed packet-processing pipelines. In *USENIX NSDI*, 2022.

[51] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan RK Ports, and Aurojit Panda. Multitenancy for fast and programmable networks in the cloud. In *USENIX HotCloud*, 2020.

[52] Ziqiang Wang, Zhuotao Liu, Xiaoliang Wang, Songtao Fu, and Ke Xu. DIP: Unifying network layer innovations using shared L3 core functions. In *ACM HotNets*, 2022.

[53] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[54] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *USENIX NSDI*, 2022.

[55] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, centralized lock management using programmable switches. ACM SIGCOMM, 2020.

[56] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *USENIX NSDI*, 2022.

[57] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. HyperVDP: High-performance virtualization of the programmable data plane. *IEEE JSAC*, 2019.

[58] Bohan Zhao, Wenfei Wu, and Wei Xu. NetRPC: Enabling in-network computation in remote procedure calls. In *USENIX NSDI*, 2023.

[59] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4Visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *ACM CoNEXT*, 2018.

[60] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. RackSched: A microsecond-scale scheduler for rack-scale computers. In *USENIX OSDI*, 2020.

[61] Hang Zhu, Tao Wang, Yi Hong, Dan R. K. Ports, Anirudh Sivaraman, and Xin Jin. NetVRM: Virtual register memory for programmable networks. In *USENIX NSDI*, 2022.

A The Paxos DPFs

We focus on two DPFs that correspond to Paxos's second phase messages.
Leader. This DPF requires a single directly mapped counter that can be placed in either lane of any EU. Upon packet arrival, the EU increments the counter using a FA-ADD instruction. Next, the epilog PUSHes the register value onto the packet stack alongside the Paxos round number, which is a constant zero for the fast-path phase of Paxos. The epilog also changes the program counter to the acceptor DPF and performs *multicast* steering action to send the packet to all acceptors.
Acceptor. Three n-sized arrays are allocated; one in EU0, two in EU2. The first array holds a list of Paxos round values per proposition. Similarly, the second and third arrays hold the most recent value and the Paxos *vround* value for each proposition, respectively.

The prolog POP-s the value proposition, its index i ($i < n$), and the round number from the packet stack. EU0 performs an atomic *fetch and max* (FA-MAX) between the locally saved i^{th} instance's round number and the one extracted from the packet. Next, EU1 compares these two values. If the local round number is larger than that of the packet, the packet is dropped. Otherwise, EU2 updates the i^{th} vround entry to hold the new round value and alters the i^{th} value to the one suggested by the packet. Then, the epilog *multicast*-s the packet to all learners.

B Branching Logic

We now describe how the control logic parses the DPF data fields and changes the execution flow. The DPF fields that are taken into account are CTRL_OP, PC_TAKEN, PC_NTAKEN, SEL_OP1, SEL_OP2, IMM_OP1 and IMM_OP2. We refer the reader to Figure 5 for a complete list of the DPF data fields. PC_TAKEN and PC_NTAKEN contain pointers to the next execution unit's instruction, or for the epilog code in the last EU. It can point to either an in-packet instruction, or an in-switch one.

Figure 12 provides an overview of the control logic. At first, the operands are acquired based on the SEL_OP1 and SEL_OP2 fields. Each of these fields may either represent a register index or signal that an immediate value should be loaded from one of the IMM_OP fields. After obtaining the operands they get compared using five operators (CMP component in Figure 12): bitwise-AND, bitwise-XOR, saturated-subtraction, sign-operand1, and sign-operand2. The outcome is a vector that holds the results of all operations.

Next, a branch resolution unit takes the CTRL_OP data-field for determining the next program-counter value. This is performed using a TCAM selection that finds the *first matching result* and acts according to the output action, effectively implementing a long if-else-if operation. The table is configured as follows (underscores represent don't-cares):

```
table branch_tbl {
    key = {
        ctrl_op : ternary;
        bitwise_and : ternary;
        bitwise_xor : ternary;
        sat_diff : ternary;
        oprnd1_sign : ternary;
        oprnd2_sign : ternary;
    }
    entries = {
        (HALT, _, _, _, _, _) : halt();
        (JMP,  _, _, _, _, _) : taken();
        (BSET, 0, _, _, _, _) : not_taken();
        (BSET, _, _, _, _, _) : taken();
        (BEQ,  _, 0, _, _, _) : taken();
        (BEQ,  _, _, _, _, _) : not_taken();
        (_,    _, 0, _, _, _) : not_taken();
        (BLT,  _, _, 0, _, _) : taken();
        (BGT,  _, _, 0, _, _) : not_taken();
        (BGT,  _, _, _, _, _) : taken();
        (BSLT, _, _, 0, 0, 0) : taken();
        (BSLT, _, _, 0, 1, 1) : taken();
        (BSLT, _, _, _, 1, 0) : taken();
        (BSGT, _, _, _, 0, 1) : taken();
        (BSGT, _, _, 0, 0, 0) : not_taken();
        (BSGT, _, _, _, 0, 0) : taken();
        (BSGT, _, _, 0, 1, 1) : not_taken();
        (BSGT, _, _, _, 1, 1) : taken();
        (_,    _, _, _, _, _) : not_taken();
    }
}
```

The following offers an alternate description for the outcome of branch resolution unit:

- HALT: Stop execution.
- JMP: Take PC_TAKEN unconditionally
- BEQ: If op1=op2 take PC_TAKEN, otherwise PC_NTAKEN
- BSET: If op1&op2 take PC_TAKEN, otherwise PC_NTAKEN
- BLT: If op1<op2 (unsigned) take PC_TAKEN, otherwise PC_NTAKEN
- BGT: If op1>op2 (unsigned) take PC_TAKEN, otherwise PC_NTAKEN
- BSLT: If op1<op2 (signed) take PC_TAKEN, otherwise PC_NTAKEN
- BSGT: If op1>op2 (signed) take PC_TAKEN, otherwise PC_NTAKEN

Since this is a two-way branch, we can achieve more functionality by swapping PC_TAKEN and PC_NTAKEN at compile time. For example, to implement a less-than-equal branch we can use a greater-than branch with swapped PC targets.

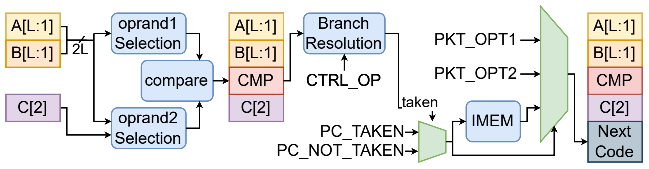


Figure 12: Overview of SwitchVM control unit.

App	EU0			EU1			EU2			EU3			FWD
	Dir	Seg	Mem	Dir	Seg	Mem	Dir	Seg	Mem	Dir	Seg	Mem	
Cache	k		k		3	$3k$		1	k				1
Sketch					3	$3k$							
LB1					3	$3k$				1		1	k
LB3					2	$2k$							k
Leader	1		1										1
Acceptor		1	k				1	2	$2k+1$				1
Available total	45K	6K	180K	45K	6K	180K	45K	6K	180K	45K	6K	180K	32K

Table 4: SwitchVM resource consumption for various DPFs.

C DPF Resource Usage

In our SwitchVM prototype, the switch can accommodate up to 4K in-switch instructions of each type. Assuming each DPF employs at most a single branch, as in our applications, the switch can store up to 2K completely distinct DPFs. These DPFs can be shared between tenants through the use of virtual memory.

Table 4 shows the resource consumption of various DPFs, specifically the direct and segmented mappings per EU, total memory usage per EU, and the total number of forwarding rules. These are the resource that *cannot* be shared by multiple tenants. In the table, k stands for the size of each DPF (top to bottom): number of cached keys, number of counters in sketch, number of servers, number of servers, N/A, number of value propositions.

Co-allocating different applications within the same switch enhances resource utilization, as different applications require memory allocation at different EUs. Capacity limitations for different instance sizes can be inferred from the table. For instance, we can have a single Cache with up to 45K keys, but the number of different instances is capped at 2K, irregardless of the cache size, due to the number of segmented mappings limitation at EU1.