Online Reprogrammable Multi Tenant Switches

Johannes Krude
RWTH Aachen University
krude@comsys.rwth-aachen.de

Jaco Hofmann
Technische Universität Darmstadt
hofmann@esa.tu-darmstadt.de

Matthias Eichholz
Technische Universität Darmstadt
eichholz@cs.tu-darmstadt.de

Klaus Wehrle
RWTH Aachen University
wehrle@comsys.rwth-aachen.de

Andreas Koch
Technische Universität Darmstadt
koch@esa.tu-darmstadt.de

Mira Mezini
Technische Universität Darmstadt
mezini@cs.tu-darmstadt.de

ABSTRACT

Recent research shows many benefits for cloud workloads and network operations when putting software functionality onto switches. Sharing the physical resources of a programmable switch between multiple tenants and workloads enables the widespread deployment of on-switch software functionality. Currently, changing the program on a programmable switch incurs significant switch downtime, connectivity loss, and service interruption. We, therefore propose a modification to the common programmable switch architecture to enable hot-pluggability, the ability to insert, modify, and remove on-path software functionality without interrupting the network operation. With hot-pluggability, a programmable switch can be shared between applications of different on-switch lifetime and therefore also between different tenants. Such sharing requires performance and program isolation between different on-switch functions and tenants. Our proposal makes on-switch software functionality deployable within production networks and enables programmable switches to be offered as a service to multiple tenants within cloud and ISP networks.

CCS CONCEPTS

• **Networks** → **Programmable networks**; In-network processing; • **Hardware** → *Networking hardware*; • **Security and privacy** → *Systems security*.

KEYWORDS

programmable switches,hot-plugging,reconfiguration,multi-tenant

ACM Reference Format:
Johannes Krude, Jaco Hofmann, Matthias Eichholz, Klaus Wehrle, Andreas Koch, and Mira Mezini. 2019. Online Reprogrammable Multi Tenant Switches . In *1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms (ENCP '19), December 9, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3359993.3366643

1 INTRODUCTION

Research has provided a multitude of examples where individual applications benefit from moving some application logic to programmable switches: Offloading data aggregation and filtering to switches speeds up MapReduce, machine learning, massively parallel databases, and string searching [19, 24, 27, 34, 35]. On-switch support for consensus protocols greatly reduces coordination overhead [8, 10, 21, 25]. Adding key-value caches to switches increases throughput while also decreasing latency [22, 39]. A single stateful on-switch load balancer can replace hundreds of software-based load balancers [29]. Executing heavy hitter detection and monitoring on switches reduces communication overhead and increases accuracy [17, 26, 36]. Even control of industrial machinery benefits from in-network aggregation and decision making [14, 15, 33].

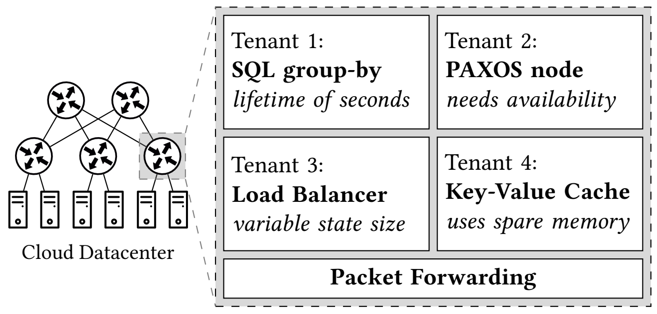

Figure 1: Multi-tenant programmable switch example.

Similar to current cloud computing, all the mentioned on-switch functions could either be provided by the network operator as application-specific services, or by allowing customers to put their *own* software into the network. Datacenter operators [7] as well as ISPs [31] are interested in integrating programmability to offer on-path software functionality in their networks. The virtual networks available in clouds may be extended with on-demand in-network computing as shown in Figure 1 or an ISPs may provide the ability to move functionality such as monitoring and DoS protection closer to the source. *We are convinced, that an extension of the current programmable switch architecture is needed to enable "Programmable Switches as a Service", where all these functions can be concurrently deployed in the same network.*

To support a diverse range of on-switch functions, the network must be able to concurrently execute a frequently changing set of multiple functions while providing performance and program isolation. However, existing programmable switches execute a single program that cannot be replaced while providing uninterrupted connectivity. Changing on-switch functions causes switch and network outages (up to 50 ms on the Barefoot Tofino [3]) while reprogramming the switch, which is a major obstacle to both running multiple functions on a single switch and to using the same switch concurrently for packet forwarding. For example, a downtime of 50 ms on a 64 port 100 GbE switch executing an aggregation step from a massively distributed database would lead to data loss of 37 GiB. *We, therefore propose to modify the common programmable switch architecture to allow for hot-pluggability, the ability to insert, modify, and remove on-switch functions without affecting other on-switch functions and packet forwarding.*

We believe, that hot-pluggable isolated on-path software functionality will bring forth many new research questions. To show the feasibility of our proposition, we describe three different approaches which require a varying amount of effort: *1.* A switch vendor can build a hot-pluggable switch using multiple current-generation switching ASICs. *2.* An FPGA based implementation can be realized by researchers and industry based on readily available hardware. *3.* Programmable switching ASICs can be extended to provide the primitives needed for hot-pluggable functions.

2 PROGRAMMABLE SWITCHES

Programmable switches, proposed as RMT (reconfigurable match tables) [6] and implemented in e.g., the Barefoot Tofino ASIC [23], emerged to solve two shortcomings of previous SDN switches: parsing of arbitrary headers and more universally programmable actions. In RMT, the parser maps header fields from variable packet offsets to fixed memory addresses in the header vector, which then passes through a fixed number of match-action stages, e.g., between 10 and 20 stages on the Barefoot Tofino [1], as illustrated in Figure 2. Each stage matches some fields of the header vector against tables to select actions that calculate a new header vector. Additionally, each stage has a small amount of stage-local registers, counters, and meters which can be accessed and modified by the action. Finally, the deparser recombines the header vector into a packet header.

On a programmable switching ASIC [6], each match-action stage processes one packet per cycle. This design gives a fixed latency and enables a packet rate equal to the clock frequency. Therefore, functionality which is expressible on such a programmable switch runs at line-rate with only small latency. However, some programmable switches [23] permit loops by recirculating processed packets back into the start of the processing pipeline at the cost of reduced throughput and increased latency. Additionally, packets can usually be diverted to a general-purpose CPU with varying latency.

The behavior of a programmable switch is programmed and configured through two different mechanisms. A program written in high-level languages such as P4 [5] describes the parser and deparser, how to match on the header vector, and a set of available actions. As illustrated in Figure 2, these can not be changed while processing packets. Unlike a CPU that distributes processing steps over a variable amount of time, a programmable switch performs spatial computation where individual processing steps are assigned to distinct areas on a switching ASIC. To fit a control program onto the fixed-size match-action pipeline, the compiler performs optimizations such as putting independent tables onto the same match-action stage and splitting too-large tables into multiple stages. Memory locations, such as in the header-vector, the match tables, and stateful stage memory are statically allocated by the compiler. Once a P4 program is loaded onto a switch, a protocol

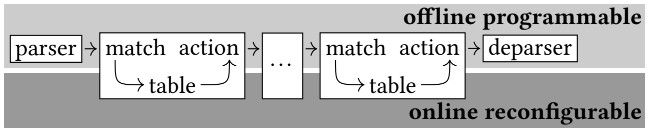

Figure 2: Current programmable switch pipeline.

like OpenFlow can be used to dynamically change the content of match tables while processing packets.

Although this architecture seems rather limited, a range of application functionality is expressible on these switches and can take advantage of the throughput and latency guarantees.

3 PROGR. SWITCHES AS A SERVICE

We envision, that hot-pluggable on-path software functionality allows a data center or network operator to efficiently manage their own on-switch functions, enables new network services, and is the necessary foundation to allow customers to execute their own functions in the provider's switching fabric. Especially with the ability to execute custom switch programs in someone else's network, there is no need to wait until the network operator offers a service for a particular application that enables rapid deployment of new on-switch functions. We see several scenarios benefiting from Programmable Switches as a Service.

On-switch functions in cloud networks. Most of the existing examples of on-switch functions focus on distributed data center applications typically found in the cloud such as massively parallel databases, key-value caches, and distributed consensus. Providing programmable switches as part of virtualized cloud networks enables the benefits of on-switch functions for cloud customers. For example, an aggregation operator in a database query could be offloaded by the cloud customer to a programmable switch operated by the cloud provider. Multi-tenancy on programmable switches requires isolation to provide performance guarantees and to restrict the execution and packet access of on-switch functions to the virtual network of the customer.

Fine-grained traffic filtering. IXPs are already experimenting with offering fine-grained traffic filtering to allow their customers to mitigate DoS attacks [11]. Any network could offer rich programmability to monitor and filter abnormal traffic by offering access to their hot-plugging enabled programmable switches. A victim of DoS attacks could deploy their own DoS protection into a network close to the attack source, e.g., at an ISP, IXP, or transit provider, by utilizing the network provider's programmable switches. The network provider then places the supplied functionality onto appropriate switches in their own network. However, the network provider should enforce restrictions such as executing a function only on packets destined for the owner of the function.

On-demand instantiation of a customer program on a switch is currently not possible without downtime and connectivity loss. Isolated hot-pluggability will enable the concurrent use of a single switch for multiple tenant functions and packet forwarding.

4 REASONS FOR HOT-PLUGGABILITY

Hot-pluggability enables the efficient use of switch resources by instantiating functions only for the time and place they are needed while sharing switches between multiple functions and tenants.

On-demand instantiation. Hot-pluggability enables efficient resource usage for short-running tasks and enables immediate deployment of customer programs. Putting some application functionality permanently onto switches prevents other applications from also utilizing on-switch resources.

As an example, NetAccel [24] offloads hash-join and group-by operators onto switches. Whenever no database query is currently processed, the resources for both operators remain unused, whenever the query planner decides to use only one of those operators on a particular switch, the resources solely used by the other operator still remain unused. Using on-demand instantiation, a database query planner can spawn operators on programmable switches whenever a new query arrives and immediately tear them down when processing is finished.

Switch Sharing. Software functionality that requires only little resources can share a switch with many other on-switch functionalities. For example, a single NOPaxos [25] instance compares a single sequence number, thereby requiring only very few switch resources. Using a separate switch for each NoPaxos instance leaves most switch resources unused. By sharing a switch between a NOPaxos instance and other functionality, switch resources can be much more efficiently used in comparison to having a dedicated switch for each function.

Programmable switches provide a guaranteed packet throughput thit is independent of the utilization of matches, table sizes, and actions. Sharing a switch between as many software functions as fit on the switch does therefore not influence the rate of processed packets. The combination of switch sharing and on-demand instantiation requires hot-pluggability since it enables to share a switch between functionality with different on-switch lifetime. A function can then be added, modified, and removed at any time without interfering with other functionality on the same switch.

Individual customization. Hot-pluggability enables offloading application functionality that is better tailored to the current needs instead of using a generalized variant. The fixed-size nature of on-switch data structures complicates the use of variable length values not only for aggregation [34] and caching [22]. When filtering data through string search with PPS [19], the offloaded program must be parameterized at compile time with the number of characters to compare in each pipeline stage. Comparing few characters in a single match-action stage allows for many short string patterns, whereas comparing many characters in each stage allows for only a few but long patterns. A generalized variant that is permanently on the switch only supports a single parameter, e.g., a fixed number of compared characters in each match-action stage. With hot-pluggability, the on-switch functionality can be customized for each individual task.

Adaptive placement. Hot-pluggability enables functionality to be moved to that switch where it is currently most useful. Aggregation [27] and filtering achieve better data reduction when being placed closer to the data sources. Monitoring approaches such as UnivMon [26] use integer-linear-programming to optimize the placement of sketches in the network, which may lead to network-wide placement changes for only small network topology changes. Through hot-pluggability, on-switch functionality can be frequently moved within the network.

Scaling. Hot-pluggability enables the dynamic allocation of resources according to the current demand. For example, the SilkRoad stateful load balancer [29] requires stateful switch memory for each connection. Programmable switches lack dynamic memory management with all memory allocations happening at compile time. Using hot-pluggability, a function can be replaced by a differently sized variant. Therefore, the flow table of a load balancer can be enlarged when many new flows are expected or shrunk when it is mostly empty. Replacing a stateful function with a differently sized variant requires migrating the state to the new variant. State migration could be implemented in the function or in the switch but is not yet provided by our approach.

Hot-pluggable on-switch functions provide many benefits but are not supported by current programmable switches. We continue with a description of requirements for supporting such functions.

5 ARCHITECTURE REQUIREMENTS

We propose a generic hot-pluggable switch architecture that captures the requirements for executing on-switch functions in a variety of scenarios. All on-switch function examples we found require executing at most one function for each packet. We, therefore divide the generic architecture into two parts as shown in Figure 3, the front-end part, shown in light gray, which selects a single switch-function for each packet from the hot-pluggable functions part, shown in dark gray.

Switch front-end. The switch front-end is the non-hot-pluggable part of the switch that provides regular non-function packet processing and is required to steer packets to the hot-pluggable functions. Whenever a packet arrives, the front-end inspects the outer packet headers and decides which function, if any, to execute. For example, in a public cloud data center, this decision could be based on the IP destination address, a VXLAN header, or some kind of application selection header. Although the front-end part of a switch program does not need to be hot-pluggable, some parts need to be online configurable. After adding or before removing on-switch functions, the function selection must be configured to appropriately steer packets through the switch.

The front-end may need to update the outer header based on the function result and decide on the output port. Additionally, regular packet processing, which may still happen on the switch, can be applied to the packets before or after executing a function. As shown in Figure 3, the front-end can be built upon programmable switch building blocks such as a parser, match-action stages, and a deparser.

Hot-Pluggable Functions. A major part of a hot-pluggable switch is the online reprogrammable functions. Since different application-specific functions require different headers, hot-pluggable functions not only require online reprogrammable match-action stages, but also an online reprogrammable parser and deparser. Some on-switch functions also require the help of an SDN controller [29], which can be virtualized [2] on the general-purpose CPU available to the switch.

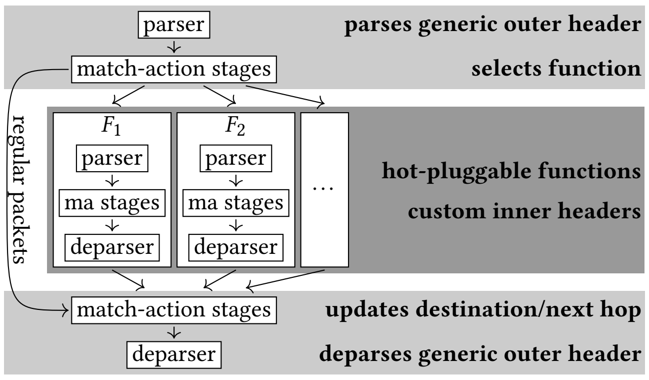

Figure 3: Generic hot-pluggable switch architecture.

Performance Isolation. When sharing a switch between multiple functions or tenants, the performance impact between the functions should be limited. Although the most important kind of performance isolation is the ability to hot-plug a function without interrupting everything else on the switch, some more switch resources need isolation. Since packet recirculation impairs the achievable packet rate, its use by on-switch functions should be either forbidden or limited, to give other functions the chance to process enough packets. The switch front-end should have the ability to override recirculating decisions to allow for dynamic accounting of recirculating bandwidth.

Program Isolation. When executing a function that cannot be fully trusted, its access to switch and packet memory should be restricted. For example in a multi-tenant cloud scenario, a customer's function should only have access to his customers packets. The switch execution model of executing the switch program on each individual packet fits well with restricting access to only selected packets. In our example, the cloud customer should additionally not be able to change the outer VXLAN identifier to ensure that packets do not leak into another virtual network. Access to the outer headers should therefore be limited by, e.g., giving only read access or no access to the VXLAN identifier.

Switch memory such as tables, registers, counters, and meters of one tenant or function should not be able accessible by other tenants or functions. A hot-pluggable switch architecture therefore needs to limit each hot-pluggable function to its own memory. For control-plane access to this memory an isolation layer similar to an SDN hypervisor [4] is needed.

A programmable hot-plugging enabled multi-tenant switch has several architectural requirements. How all these requirements are realizable is presented in the next section.

6 REALIZATION

The previous sections propose certain architectural traits that a new generation of programmable switches should support. Mainly the need for uninterrupted multi-application support requires decisive changes to the currently available hardware. This section discusses different possible implementations of a system that fulfills the requirements outlined previously. The focus is trifold: *1.* A solution based on a concatenation of multiple switching ASCIs to achieve interrupt free function switching. *2.* An FPGA based solution, which offers greater potential for reconfigurability at the cost of raw throughput and ease-of-use. *3.* Possible extensions to existing ASICs to make them next-generation ready.

6.1 Using multiple switching ASICs

Although a single programmable switch needs to be taken offline for reprogramming, interruption-free hot-pluggability can be implemented by using *two* ASICs for hot-pluggability and a third front-end ASIC that redirects packets from the physical switching ports to the active hot-pluggable ASIC. While one switching ASIC executes functions on packets, a second standby ASIC can be programmed with a different set of functions. Once reprogramming is finished, the front-end ASIC redirects packets to the newly programmed ASIC thereby swapping the roles of active and standby ASIC.

Program merging & isolation. This approach uses an ASIC that runs a *single* program to concurrently execute multiple functions. Therefore, a compiler is needed that not only merges multiple functions into a single program but also ensures isolation between the functions. The compiler therefore verifies that each function only accesses its own switch memory, adheres to the access restrictions on the outer headers in the packet memory, and limits its use of recirculation. To restrict control-plane access, the compiler provides a mapping from allocated tables and switch memory to individual functions.

State Migration. When moving from one ASIC to the other, functions that are present on both keep working without interruption. However, stateful functions also need to migrate their state to the new ASIC. Since atomically migrating such state between ASICs while redirecting packets is difficult, this approach is most useful for stateless functions. Integrating migration support within the function may allow the migration for at least some stateful functions.

Switch Ports. The front-end ASIC, which implements the front-end part from Section 5, connects to the physical switch ports and to the active and standby ASIC. Therefore, only some of its ports, e.g., a third, are available as physical switch ports. When using multiple front-end and active ASICs, the number of physical switch ports can be increased.

Although using multiple ASICs provides hot-pluggability for only some functions, its utilization of currently available ASICs makes it easily realizable. It allows instantiating on-switch functionality within the time it takes to compile the merged program, followed by reprogramming the standby ASIC and changing a forwarding rule on the front-end ASIC. On-switch functions are atomically swapped without interruption of packet processing by atomically changing the forwarding rules on the front-end ASIC. Switch resource usage is however doubled by having an standby ASIC that does not process packets.

The main limitation of this approach is the lack of switch support for state migration. Providing hot-pluggability through multiple switching ASICs is not suitable when executing stateful on-switch functions that cannot migrate their state by themself. We, therefore present two additional realization approaches that both do not require state migration.

6.2 Using FPGAs

FPGAs continue to gain traction in the data center world. They provide a good trade-off between performance and energy-efficiency offering highly specialized architectures for a given workload when the cost to produce a dedicated ASIC is too high. Another advantage that is heavily utilized is the ability of the FPGA to be reconfigured based on user-demand. For example, Microsoft uses FPGA based Smart NICs to enforce SDN policies in the Azure cloud [12].

Next-generation switch architectures can be explored in FPGAs. A wide spectrum of off-the-shelf boards is available [41, 45] that house varying sizes of FPGAs as well as connectors to communicate over 10 Gbit/s to 100 Gbit/s Ethernet. An FPGA can easily keep up with line rate for filtering tasks and even more complex network applications [9, 13, 30].

The flexibility of FPGAs comes at the price of increased development time and the need for specialized hardware knowledge. Tools such as Xilinx SDNet [42] alleviate some costs associated with FPGA development by translating P4 programs to a hardware description languages. In comparison to programmable switching ASICs, the flexibility of FPGAs does not require to fit a P4 program onto a fixed number of fixed-size stages and allows to extend a P4 program with custom logic expressed in a hardware description language.

A straightforward switch design for FPGA can use SDNet directly [18]. However, in this approach, hot-pluggability is only possible by replacing the complete design on the FPGA, which leads to packet loss as the FPGA cannot react on incoming packets during reconfiguration and leads to loss of state, which is problematic for stateful functions.

Dynamic Partial Reconfiguration. A more suitable approach utilizes the dynamic partial reconfiguration feature of modern FPGAs. This feature allows replacing only a part of the FPGAs fabric with new logic, leaving the rest of the FPGA fully operational. As illustrated in Figure 4, the FPGA is therefore divided into a fixed-function region, shown in light gray, and multiple dynamically reconfigurable areas, shown in dark gray. During reconfiguration of one region, the rest of the FPGA remains responsive to all requests that are not targeted at the specific region to be replaced.

Apart from the online reconfigurable regions that house the hot-pluggable functions, some additional infrastructure is needed to provide interfaces to the reconfigurable regions and to perform packet forwarding. Each packet is forwarded by the function selector, which can also be implemented in P4, to one of reconfigurable regions.

Reconfigurable Area Allocation. The fixed division of the FPGA into individually reprogrammable regions, imposes some limitations on functions put into these regions. To allow for a single large function to span across multiple regions, packet data can be forwarded between neighboring regions as shown in Figure 4. The compiler then needs to split a P4 program into multiple smaller programs that can be fit into individual regions.

Isolation. The reconfigurable regions isolate functions by only providing explicit interfaces between functions. Packet recirculation can be implemented within the regions of a single function, therefore not taking away bandwidth from other functions. Forwarding each packet to only the appropriate reconfigurable region limits packet access. The outer headers can be made read-only by keeping a copy in the front-end and can be made non-accessible by only selectively forwarding headers to reconfigurable regions.

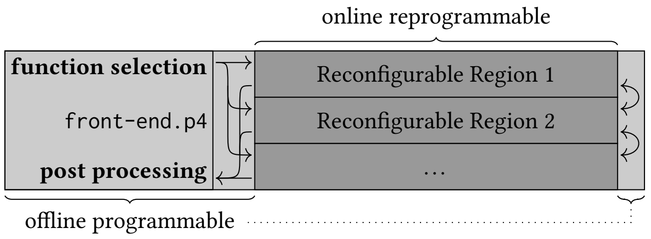

Figure 4: Using the partial reconfigurability of FPGAs to provide hot-pluggable on-path software functions.

Utilizing partial reconfiguration on FPGA can, in general, solve all the problems described in Section 5. However, some caveats apply. The size of the reconfigurable regions has to be predetermined and chosen wisely based on the expected user applications, since large regions waste unused FPGA area, whereas small regions introduce overhead when splitting functions into multiple small programs. Although a single reconfigurable region can be reprogrammed within a fraction of a ms, FPGA bitstream generation take minutes to hours to compile a function for a specific region. Additionally, the flexibility of FPGAs comes at a price of lower raw packet processing performance and fewer available Ethernet ports compared to dedicated switching ASICs.

6.3 An extension to current switching ASICs

Only minor modifications are necessary to make current switching ASICs hot-plugging enabled, since the biggest reconfigurable part of the ASIC, the table entries [6, 23], are already online reconfigurable.

Online reprogrammable pipeline. The parser behavior, match-table selection, actions, and the deparser are stored in the same kind of memory as the tables, namely TCAM and SRAM. Although TCAM and SRAM can easily be made online reconfigurable, it is important to avoid inconsistent states without interrupting packet processing.

The parser, as described by RMT [6], consists of TCAM that holds transitions of the parser state-machine. To remove a part of the parser during ongoing packet processing, the part must first be made unreachable by removing the transition that leads to the to be removed part. Once the part of the parser to be removed is no longer in use, all remaining entries can be removed. After the pipeline is drained of packets for a function, matches, actions, and match-table entries can also be removed. Similarly, when adding a function to the pipeline, all parts of this function should only be made reachable once they are fully configured. The use of a per entry valid-bit to allow for atomic insertion, deletion, and movement of TCAM entries is described in CoPTUA [40].

Program merging. When adding a function to an online reprogrammable pipeline, parts of the pipeline are already occupied by other functions. The P4 compiler must be made aware of the available resources in order to fit a function into the currently unoccupied parts of the pipeline. When allocating match-action stages

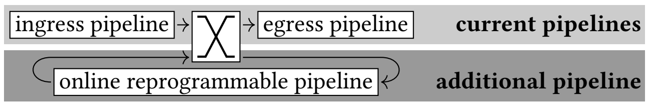

Figure 5: Adding a hot-plugging enabled pipeline.

and table memory, care must be taken to avoid fragmenting the unoccupied resources into many small ranges.
Isolation. When making the compiler aware of resources allocated to individual functions, the isolation procedures described in Section 6.1 can be applied.
Multiple pipelines. Current switches already include at least two pipelines, an ingress and an egress pipeline, which are executed before and after the selection of the output port. Using the ingress pipeline for both function selection and function execution is difficult. Either function selection must be implemented with the limited matching capabilities of the parser, or hot-pluggable functions lose the ability to parse their own custom headers. Executing hot-pluggable functions in the egress pipeline assigns the output port to each packet before executing functions, therefore removing the ability of hot-pluggable functions influence the output port. We propose to add a third pipeline for hot-pluggable functions as illustrated in Figure 5. In this way, function selection can be performed in the ingress pipeline before handing packets over to the custom parsers in the hot-pluggable pipeline while also providing the possibility to influence the output port from the hot-pluggable functions.

Adding a third pipeline is doable, since RMT [6] describes how to share most match-action resources between logically separate ingress and egress pipelines and some switches [1] already include additional but non online reprogrammable pipelines for extra processing.

Extending switching ASICs for hot-pluggability does enable on-demand instantiation of functions running at the full switch line-rate within a networks switching fabric. The exact cost of making a switching ASIC online reprogrammable is not known to us since we did not yet implement any of the presented approaches.

7 RELATED APPROACHES

Deployment problems for on-switch in-network computing have been widely recognized [19, 24, 32]. NetAccel [24] proposes to avoid on-demand instantiation by permanently putting generalized aggregation functionality onto switches, whereas PPS [19] proposes to use dedicated switches without forwarding duties as offloading appliances for a single application functionality. Tokusashi et al. [38] analyze that on-demand in-network computing can decrease power consumption but have not considered how to change switch programs. Ports et al. [32] provide guidelines for on-switch in-network computing, suggesting to put generalized functionality onto switches to avoid deployment problems.

Hot-pluggability is possible within an active network [37] with approaches such as tiny pack programs [20] where switch programs are embedded into the forwarded packets. Since each packet may carry a different program, this provides perhaps the highest degree of hot-pluggability.

Online reconfiguration of match-table entries is an integral part of SDN [28] and multiple controllers can be used with SDN hypervisors [4]. We want to go further in also enabling hot-pluggability for the underlying data-plane programs.

P4Visor [44] proposes A-B Testing of P4 programs by putting both programs onto the same switch while optimizing resource usage through merging similar program parts. Although they do not tackle the problem of interruption-free deployment, their merging algorithm could be useful when sharing a switch between multiple application functionality.

Hyper4 [16] and HyperVDP [43] emulate a programmable switch within a P4 program by encoding the switch program within match-action tables, thereby enabling hot-pluggability of P4 programs. They, however, focus on the composition and modification of virtual switches in virtual networks, while we want to improve the deployability of on-switch application functionality. Hyper4 uses massive packet recirculation to implement a hot-pluggable parser thereby dividing the achievable packet-rate by the number of parsed headers, whereas HyperVDP avoids recirculation by removing the programmable parser in only matching on fields at fixed packet offsets. Hyper4 and HyperVDP need 6-13 and 6 match-action stages to emulate a single match-action stage, which allows for only 1-3 emulated match-action stages on the 10-20 stages [1] available in the Barefoot Tofino. We propose to modify the programmable switch architecture to enable full hot-pluggability without excessive resource consumption. In comparison to Hyper4 and HyperVDP, we propose to keep the programmable parser, to not decrease the achievable packet rate, and to not decrease the number of available match-action stages.

8 CONCLUSION

Hot-pluggability brings the deployment of on-switch programmability to a new level, enabling cloud providers to offer switch programmability to customers and even enabling ISPs to offer some on-path edge computing. We describe architectural requirements as well as three approaches to realize our vision of hot-pluggable on-path software functionality, all of which atomically add and remove on-switch functions without interrupting packet processing. Putting together multiple switching ASICs is the most simple of the presented approaches at the cost of not supporting stateful functions. Our FPGA based proposal uses of the shelf hardware, but requires FPGA knowledge and supports only few switch ports. Modifying a switch ASIC may allow cheap hot-pluggability for networks and data-centers, but is difficult to implement as a researcher. Although we did not yet implement these three proposals, we believe that all of them are feasible. Hot-pluggability is the necessary foundation for Programmable Switches as a Service and raises many new questions, ranging from resource allocation, placement, and accounting, to state migration.

ACKNOWLEDGMENTS

The author would like to thank the numerous anonymous reviewers for their valuable comments and helpful suggestions.

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

REFERENCES

[1] ARISTA. 2018. Arista 7170 Multi-function Programmable Networking. https://www.arista.com/assets/data/pdf/Whitepapers/7170_White_Paper.pdf.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. *SIGOPS Operating Systems Review* 37, 5 (Oct. 2003), 164–177. https://doi.org/10.1145/1165389.945462

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

[4] Andreas Blenk, Arsany Basta, Martin Reisslein, and Wolfgang Kellerer. 2016. Survey on Network Virtualization Hypervisors for Software Defined Networking. *IEEE Communications Surveys & Tutorials* 18, 1 (First Quarter 2016), 655–685. https://doi.org/10.1109/COMST.2015.2489183

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[6] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the 2013 ACM SIGCOMM Conference (SIGCOMM '13)*. ACM, 99–110. https://doi.org/10.1145/2486001.2486011

[7] Adrian Caulfield, Paolo Costa, and Monia Ghobadi. 2018. Beyond SmartNICs: Towards a Fully Programmable Cloud. In *IEEE International Conference on High Performance Switching and Routing (HPSR)*. IEEE. https://doi.org/10.1109/HPSR.2018.8850757

[8] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos Made Switch-y. *ACM SIGCOMM Computer Communication Review* 46, 2 (May 2016), 18–24. https://doi.org/10.1145/2935634.2935638

[9] H. T. Dang, J. Hofmann, Y. Liu, M. Radi, D. Vucinic, R. Soulé, and F. Pedone. 2018. Consensus for Non-volatile Main Memory. In *26th International Conference on Network Protocols (ICNP)*. 406–411. https://doi.org/10.1109/ICNP.2018.00056

[10] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM. https://doi.org/10.1145/2774993.2774999

[11] Christoph Dietzel, Matthias Wichtlhuber, Georgios Smaragdakis, and Anja Feldmann. 2018. Stellar: Network Attack Mitigation Using Advanced Blackholing. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM, 152–164. https://doi.org/10.1145/3281411.3281413

[12] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*. USENIX Association, 51–66.

[13] Jeffrey Fong, Xiang Wang, Yaxuan Qi, Jun Li, and Weirong Jiang. 2012. ParaSplit: A scalable architecture on FPGA for terabit packet classification. In *20th Annual Symposium on High-Performance Interconnects*. IEEE, 1–8. https://doi.org/10.1109/HOTI.2012.17

[14] René Glebke, Martin Henze, Klaus Wehrle, Philipp Niemietz, Daniel Trauth, Patrick Mattfeld, and Thomas Bergs. 2019. A Case for Integrated Data Processing in Large-Scale Cyber-Physical Systems. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*. 7252–7261. https://doi.org/10.24251/HICSS.2019.871

[15] René Glebke, Johannes Krude, Ike Kunze, Jan Rüth, Felix Senger, and Klaus Wehrle. 2019. Towards Executing Computer Vision Functionality on Programmable Network Devices. In *1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms (ENCP '19)*. ACM. https://doi.org/10.1145/3359993.3366646

[16] David Hancock and Jacobus van der Merwe. 2016. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *Proceedings of the 12th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, 35–49. https://doi.org/10.1145/2999572.2999607

[17] Rob Harrison, Qizhe Cai, and Arpit Guptaand Jennifer Rexford. 2018. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. ACM. https://doi.org/10.1145/3185467.3185476

[18] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. ACM, 1–9. https://doi.org/10.1145/3289602.3293924

[19] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. 2019. Fast String Searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research (SOSR '19)*. ACM, 21–28. https://doi.org/10.1145/3314148.3314356

[20] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM '14)*. ACM, 3–14. https://doi.org/10.1145/2619239.2626292

[21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*. USENIX Association, 35–49.

[22] Xin Jin, Xiaozhou li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, 121–136. https://doi.org/10.1145/3132747.3132764

[23] Changhoon Kim. 2017. Why Data-plane Will Be Programmable: New Paradigms and Use Cases In Networking. https://conferences.sigcomm.org/events/apnet2017/slides/chang.pdf.

[24] Alberto Lernen, Rana Hussein, and Philippe Cudre-Maurox. 2019. The Case for Network-Accelerated Query Processing. In *9th Biennial Conference on Innovative Data Systems Research (CIDR '19)*.

[25] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, 467–483.

[26] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Validimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, 101–114. https://doi.org/10.1145/2934872.2934906

[27] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf Wolf. 2014. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. ACM, 249–261. https://doi.org/10.1145/2674005.2674996

[28] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (April 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[29] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM '17)*. ACM, 15–28. https://doi.org/10.1145/3098822.3098824

[30] Sascha Muhlbach and Andreas Koch. 2010. A dynamically reconfigured network platform for high-speed malware collection. In *2010 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 79–84. https://doi.org/10.1109/ReConFig.2010.41

[31] Larry Peterson, Tom Anderson, Sachin Katti, Nick McKeown, Guru Parulkar, Jennifer Rexford, Mahadev Satyanarayanan, Oguz Sunay, and Amin Vahdat. 2019. Democratizing the Network Edge. *ACM SIGCOMM Computer Communication Review* 49, 2 (April 2019), 31–36. https://doi.org/10.1145/3336937.3336942

[32] Dan R. K. Ports and Jacob Nelson. 2019. When Should The Network Be The Computer?. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM, 209–215. https://doi.org/10.1145/3317550.3321439

[33] Jan Rüth, René Glebke, Klaus Wehrle, Vedad Causevic, and Sandra Hirche. 2018. Towards In-Network Industrial Feedback Control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing (NetCompute '18)*. ACM, 14–19. https://doi.org/10.1145/3229591.3229592

[34] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets 2017)*. ACM, 150–156. https://doi.org/10.1145/3152434.3152461

[35] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. 2019. *Scaling Distributed Machine Learning with In-Network Aggregation*. Technical Report. KAUST. http://hdl.handle.net/10754/631179.

[36] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, 164–176. https://doi.org/10.1145/3050220.3063772

[37] David L. Tennenhouse and David J. Wetherall. 1996. Towards an Active Network Architecture. *ACM SIGCOMM Computer Communication Review* 26, 2 (April 1996), 5–17. https://doi.org/10.1145/231699.231701

[38] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, 21:1–21:16. https://doi.org/10.1145/3302424.3303979

[39] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. 2018. LaKe: The Power of In-Network Computing. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFil 2018)*. IEEE. https://doi.org/10.1109/RECONFIG.2018.8641696

[40] Zhijun Wang, Hao Che, Mohan Kumar, and Sajal K. Das. 2004. CoPTUA: Consistent Policy Table Update Algorithm for TCAM without locking. *IEEE Transactions on Computers* 53, 12 (Oct. 2004), 1602–1614. https://doi.org/10.1109/TC.2004.108

[41] Xilinx. [n.d.]. Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[42] Xilinx. [n.d.]. SDNet. https://xilinx.com/sdnet.

[43] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. 2019. HyperVDP: High-Performance Virtualization of the Programmable Data Plane. *IEEE Journal on Selected Areas in Communications* 37, 3 (March 2019), 556–569. https://doi.org/10.1109/JSAC.2019.2894308

[44] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM, 98–111. https://doi.org/10.1145/3281411.3281436

[45] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. 2014. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro* 34, 5 (July 2014), 32–41. https://doi.org/10.1109/MM.2014.61