Accelerated Service Chaining on a Single Switch ASIC

Dingming Wu
Rice University & Alibaba Group
dingming.wu@rice.edu

Ang Chen
Rice University
angchen@rice.edu

T. S. Eugene Ng
Rice University
eugeneng@cs.rice.edu

Guohui Wang
Alibaba Group
g.wang@alibaba-inc.com

Haiyong Wang
Alibaba Group
haiyong.wang@alibaba-inc.com

ABSTRACT

Network functions and service function chaining are prevalent in cloud and ISP networks. In traditional software-based solutions, scaling up the capacity of these functions requires a large number of server cores. However, edge clouds are severely resource-constrained in terms of space, power, and budget, so traditional methods incur a high cost. We present Dejavu, a system that can offload a service chain to a programmable switch to achieve high performance and resource efficiency. Our system can compose multiple network functions into a single program that preserves the original chaining requirements, and exploit features of the switch ASIC to efficiently deploy the composed program on a single switch.

KEYWORDS

Service Function Chaining; Programmable Switch; Edge Cloud

ACM Reference Format:
Dingming Wu, Ang Chen, T. S. Eugene Ng, Guohui Wang, and Haiyong Wang. 2019. Accelerated Service Chaining on a Single Switch ASIC. In *The 18th ACM Workshop on Hot Topics in Networks (HotNets '19), November 13–15, 2019, Princeton, NJ, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3365609.3365849

1 INTRODUCTION

Today, network service function chaining (SFC) is a prevalent need in cloud and ISP networks [18]. SFC connects a sequence of network functions (NFs), e.g., load balancers, NATs, and provides it "as-a-service" to multiple tenants. Tenants' traffic is then steered through different NFs for processing. In response to the inflexibility and high cost of hardware-based middleboxes, a recent trend is to implement the services in software to run on baremetal servers or in VMs [12, 14, 17, 21, 24–27, 36, 38]. Software-based NFs reduce hardware costs and shorten the cycles for new service deployment, and they have gained popularity in large operational clouds [12, 16, 36].

However, software NFs are generally one or two orders of magnitude slower than their hardware counterparts [13, 28, 31], due to the huge performance gap between commodity CPUs and specialized switch hardware. While it requires multiple CPU cores to reach a packet processing speed of 10s of Gbps [30, 41], a single switch ASIC at a similar cost can easily process packets at several Tbps [1]. In order to handle the ever-increasing traffic volume, cloud networks need to deploy a large number of NF instances on dedicated servers [12, 30, 36]. This comes at the cost of a reduced amount of available resources for application-layer processing, which in turn translates to economic implications for the service provider.

Edge clouds are a place where this inefficiency is particularly problematic. Unlike regular cloud services that run on large-scale data centers, edge services need to be placed close to end users (e.g., mobile or IoT devices [40]), typically in leased facilities with tight power, space, and compute constraints. In such resource-constrained data centers, every available CPU cycle is precious. If a sizable amount of machines are enlisted just for packet processing, the capacity and benefit of edge clouds would considerably decrease.

Recent advances in programmable switch ASICs and data plane programming languages such as P4 [3] are quickly changing the landscape of NF implementation. Emerging switch ASICs, e.g., Intel FlexPipe [33] and Barefoot Tofino [1], are designed with a reconfigurable match-action architecture [4]. Switch programs written in P4 can be compiled to run on these switch ASICs, parse customized packet formats, and perform a rich set of operations over packet headers at Tbps speeds [1]. Recent work has shown that many NFs, such as layer-4 load balancing [31], caching [22] and security functions [32] can be implemented on programmable ASICs

HotNets '19, November 13–15, 2019, Princeton, NJ, USA

ACM ISBN 978-1-4503-7020-2/19/11…$15.00
https://doi.org/10.1145/3365609.3365849

at very high performance. This presents unprecedented opportunities for reducing the cost of network service functions at edge clouds.

Our work builds upon this trend, but takes it much further by exploring how we could deploy an *entire service chain* on a single programmable switch. If this is feasible, we could then build SFC solutions that are much more compact than software-based NFs, operating at much higher throughput and lower deployment footprints. One key challenge, however, is that the hardware resources needed by a service chain could easily exceed the capacity of any given switch ASIC pipeline. As a result, the possibility of supporting SFCs on a programmable switch remains an open problem today.

To bridge this gap, our project, called Dejavu, proposes a new model for utilizing programmable switch ASIC resources. This model does away with the static "ingress" or "egress" designation of processing pipelines, and explicitly enables packets to visit and revisit different processing logic nearly arbitrarily to efficiently realize SFCs. Dejavu leverages the multi-stage, multi-pipeline feature of the RMT architecture [1, 4], as well as the modularity of the recent P4 language [8] to connect multiple NFs and host them on a single switch ASIC. The individual functions are composed and placed on the ASIC in a manner that preserves the original functionality while maintaining high processing speeds.

The design of Dejavu addresses a set of challenges, such as merging the data plane programs, optimizing NF placements, and routing packet internally on the switch ASIC. In the rest of this paper, we present the technical challenges and our solution sketches. To obtain initial evidence of its feasibility, we have also developed a prototype of Dejavu, which supports a service chain of 5 NFs (traffic classifier, packet filtering firewall, virtualization gateway, L4 load balancer, and IP router) from a production edge cloud.

Although our current discussion focuses on single-switch scenarios, the techniques we develop can be naturally extended to multi-switch scenarios. We believe that the resource utilization model in Dejavu can enable more flexible ways to managing hardware resources—switch ASIC resources can be managed at the granularity of pipeline stages, network functions can be deployed at arbitrary pipeline locations, and flexible chaining policies can be constructed among functions within a switch or across switches to realize more advanced network services in the cloud.

2 BACKGROUND AND CHALLENGES

Switch ASIC Architecture. Fig. 1 shows the switch ASIC architecture based on the Portable Switch Architecture [9] from the P4 language consortium. In this architecture, a full packet processing pipeline contains an ingress pipe, an egress pipe, and a traffic manager. In an ingress/egress pipe

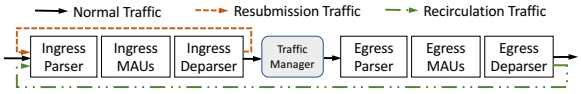

Figure 1: Switch pipeline architecture and packet paths.

(called a *pipelet* henceforth), packets are first parsed and then go through a series of match-action units (MAUs). Each MAU has a fixed amount of hardware resources (e.g., TCAM, SRAM, Crossbars, Gateways) for header and metadata processing. Packets are reassembled by the deparsers at the end of ingress pipe or egress pipe. A switch ASIC may have multiple such pipelines that share the same traffic manager. The traffic manager has packet buffers and can forward packets from any ingress pipe to any egress pipe. Pipelines are configured by data plane programming languages, such as P4 [7]. Existing work has demonstrated that many network service functions can be implemented in P4. However, the current P4 programming model does not support multi-programming—each NF must be implemented as a monolithic P4 program that is compiled and loaded onto the entire ASIC.

Recent advances in programmable switch ASIC architecture and the high-level languages (i.e., P4) present opportunities for SFC. First, the latest versions of the P4 language have made significant improvements for building modular NFs. For instance, they provide constructs (e.g., extern methods, parser, control and action blocks) for building and composing modular control blocks [8]. Furthermore, language primitives (e.g., Tofino Native Architecture [2]) are also available for independent pipeline configuration. These features provide building blocks for SFC to leverage the multi-pipeline switch architecture. Second, most switch ASICs support traffic loopback that allows packets to go through the ingress or egress pipe multiple times via packet resubmission and pipeline recirculation[1]. As shown in Fig. 1, a packet can be resubmitted to the same ingress parser after it finishes the ingress processing, or recirculated to the ingress parser after it completes the egress processing. Resubmitted or recirculated packets carry a special header that can be recognized by the ingress or egress parser and provides hints to the MAUs to perform different operations.

Challenges. The programmable ASIC architecture and the language features discussed above provide a useful starting point. However, the design of Dejavu needs to address several technical challenges. First, how should we implement and deploy multiple NFs on a single programmable ASIC? Since we can only load a monolithic P4 program on one ASIC, we need to merge or compose multiple NFs. Second, how should we chain multiple NFs based on the SFC policies? A flexible on-chip routing mechanism is needed

[1]The name of our work, Dejavu, captures the experience of packets traversing the same pipeline multiple times.

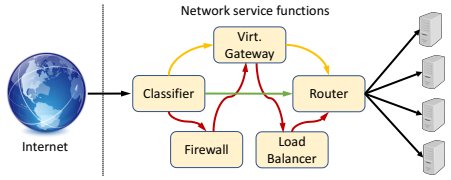

Figure 2: An example SFC in an edge cloud. Traffic may traverse the NFs in different orders. The Classifier and Router are supplied by the Dejavu framework for all SFC paths.

2 Bytes	1 Byte	4 Bytes	12 Bytes	1 Byte
Service path ID	Service index	Platform metadata	Context data	Next protocol

Figure 3: Network service header used by Dejavu.

to direct the traffic through NFs. Third, due to the architectural constraints on ASIC, different NF placements on the pipelines may require packets to travel through different physical paths. How should we optimize the placements to reduce the performance impact? Finally, the SFC policy may contain complex NFs in a long chain, whereas the packet processing pipeline usually has a small number of stages. How should we efficiently use the limited pipeline resources and compose more NFs?

3 SOLUTION SKETCH

Next, we sketch an initial design of Dejavu, using 5 NFs from a production edge cloud as an example (shown in Fig. 2). Depending on the SFC policies, traffic from the Internet may need to travel different paths: Classifier-FW-VGW-LB-Router (red arrows), Classifier-VGW-Router (orange arrows) and Classifier-Router (green arrows). Our goal is to place these 5 NFs onto a single multi-pipeline switch ASIC.

SFC Header Format. Dejavu uses a customized SFC header based on the IETF NSH proposal [37] (Fig. 3). In this header, a 2-byte service path ID and a 1-byte service index uniquely identify the next NF for a packet. The service index field will be modified after each NF. The following 4 bytes contain a copy of the platform-specific metadata, which in our current design includes 6 fields: *inPort*, *outPort*, *resubmissionFlag*, *recirculationFlag*, *dropFlag*, *mirrorFlag*, and *toCpuFlag*. *inPort* is the physical ingress port where the packet is received. *outPort* is the physical egress port that the packet needs to be forwarded to. The next 5 flag bits indicate whether a packet should be resubmitted, recirculated, dropped, mirrored or sent to CPU, respectively. Following the platform metadata, the SFC header provides 12 bytes of context header for NFs. This header allows a packet to carry context data, such as tenant ID, application ID and debugging info, along a service path so that NFs can perform policy decisions based on the context. The context header is formatted as key-value pairs, with 1-byte keys and 2-byte values. We embed the SFC header between Ethernet and IP headers and use a special EtherType to signify its existence. It is added by the Classifier module and removed by the Router module.

Generic Parser A P4 parser constructs a directed acyclic graph (DAG), where each vertex represents a header type at a particular location offset, and each edge represents a transition from one header to another. Note that the same header types appearing in different packet locations are represented by different vertices. The original parser of individual NFs may only include headers that it needs. To enable the co-location of multiple NFs, we merge the parsers of individual NFs and generate a generic parser. While DAG-merging algorithms exist [5], they cannot be directly applied to parser DAGs, as it is difficult to check the equivalence of two headers (vertices) in different parsers (DAGs)—a header may have different names or appears in different location offsets in different parsers. To solve the problem, we consider representing vertices in the DAG as *(header_type, offset)* tuples so that two vertices are equivalent only when their headers have the same type and appears at the same location offset. We create a lookup table that maps each such tuple to a global ID. The size of this table should be small as normal packets have limited header types and each header has limited possible offsets. Since parser merging requires the tuple IDs, we currently ask the NF programmer to supply this information in the program.

3.1 Control Block Programming Interface

Dejavu allows developers to write NFs in modular control blocks, and it hides the platform-specific metadata by providing a simple API with only one argument.

```
control XX_control (inout all_headers_t hdr);
```

The choice of function name ("XX") is up to the developer. The argument hdr is instantiated by the generic parser and provides all required protocol header fields, platform metadata and context metadata for the NFs. This API significantly reduces the complexity of developing NF programs as platform-specific features less relevant to the core NF logic (e.g., packet forwarding, ingress processing, egress processing, recirculation) are shielded by Dejavu.

Fig. 4 shows an example implementation of a L4 load balancer. It has only one match-action table *lb_session* that implements the core logic—replacing the destination IP of a packet with a selected server IP based on the hash of its 5-tuple, or sending the packet to CPU if there is a table miss. The 5-tuple header fields are read from the *hdr* argument (line 5). The session table *lb_session* stores the hash of all matched flows. If there is a hit on the hash code, action *modify_dstIp* will be called to modify the destination IP. If there is a miss, the default action *toCpu* will set the *toCpuFlag* in the SFC header. In the latter case, the Dejavu framework will check the flags and send the packet to the control plane.

```
control LB_control(inout all_header_t hdr){
    bit<32> sessionHash;
    Hash<bit<32>>(HashAlgorithm_t.CRC32) hasher;
    action computeFiveTupleHash(){
      sessionHash = hasher.get({hdr.ipv4.src_addr,
            hdr.ipv4.dst_addr, hdr.trans_prtcl,
            hdr.tcp.src_port, hdr.tcp.dst_port});
    }
    action modify_dstIp(bit<32>dip){hdr.ipv4.dst_addr = dip;}
    action toCpu(){hdr.sfc.toCpuFlag = true;}
    table lb_session{
        key = {sessionHash:exact;}
        actions = {modify_dstIp; toCpu;}
        const default_action = toCpu();
    }
    apply{computeFiveTupleHash(); lb_session.apply();}
}
```

Figure 4: An example implementation of a L4 load balancer using Dejavu's programming interface.

```
//sequential operator
control Ingress(inout all_header_t hdr, inout common_metadata_t
      md, inout standard_metadata_t standard_metadata){
  apply{
    if(check_nextNF.apply().LB){
      LB_control.apply(hdr);
      check_sfcFlags.apply(); }
    if(check_nextNF.apply().FW){
      FW_control.apply(hdr);
      check_sfcFlags.apply(); }
  }
}
//parallel opeartor
control Ingress(inout all_header_t hdr, inout common_metadata_t
      md, inout standard_metadata_t standard_metadata){
 apply{
    if(check_nextNF.apply().LB){LB_control.apply(hdr);}
    else if(check_nextNF.apply().FW)
      { FW_control.apply(hdr); }
    check_sfcFlags.apply();
 }
}
```

Figure 5: An example of the sequential and parallel compositions. Both invoke LB and FW on the same pipelet. Dejavu APIs are marked in blue.

Both actions will modify the corresponding fields in the *hdr* argument. Normally, the control plane will simply install a new session in the *lb_session* upon packet reception and reinject the packet into the data plane.

3.2 NF Composition

Dejavu has multiple ways to compose the NFs and generate a single multi-pipeline P4 program that can be compiled and loaded onto the physical pipelines. To reduce resource usage, Dejavu will pack multiple NFs onto the same pipelet whenever the hardware permits. Deciding whether two NFs can share the same pipelet requires the knowledge of the hardware resource usage of each NF. This information is usually available from the P4 compiler, which typically reports the exact amount of resource usage, e.g., MAU stages, SRAMs, TCAMs, of a P4 program[1, 23].

NFs can be composed either sequentially or parallelly to share the pipelet in different ways. Sequential composition places NFs on a pipelet back-to-back, and parallel composition places them side-by-side. Fig. 5 shows the implementations of these two composition approaches wrapped in an ingress processing block. The input arguments are based on an open-source switch target [10] and are similar for other targets such as Tofino [1]. The first implementation invokes LB and FW sequentially. After the packet passes an NF, Dejavu checks the SFC headers (using API *check-_sfcFlags()*), and translates any modification to the provided *hdr* argument to the corresponding platform metadata. In the second implementation, Dejavu invokes either LB or FW, but not both, and performs similar flag checks afterward.

The two composition approaches offer an efficiency and feasibility trade-off for NF placements. Parallel composition allows multiple NFs to share the same MAUs, and thus can pack more NFs on a pipelet. However, since packets can only traverse one branch on a pipelet, transitions from one branch to another require at least one resubmission (if on ingress pipe) or one recirculation (if on egress pipe). On the contrary, sequential composition places multiple NFs back-to-back on a pipelet and has no extra transition cost among those NFs. Nevertheless, sequential composition imposes an implicit dependency[2] between the chained NFs and enforces the P4 compiler to place the NFs in separate MAU stages, which may fail if the pipelet does not have enough stages.

3.3 NF Placement Optimization

Switch targets may have different constraints on packet resubmissions/recirculations. For example, Tofino [1] has the following constraints: a) packets can only be resubmitted (recirculated) after it is done with the ingress (egress) pipe; b) recirculation decisions must be made in the ingress by setting a packet's egress port to a loopback port; c) recirculation bandwidth is only supported at a granularity of Ethernet ports; and d) in a multi-pipe switch ASIC, packet resubmission and recirculation can only occur within a pipeline.

One implication of such constraints is that the NF locations have a large impact on the overall throughput and latency. Particularly, different NF placements will result in a different number of packet resubmission/recirculation—both will reduce the effective throughput and increase latency. If hardware permits, it is desirable to place two adjacent NFs on a single pipelet sequentially, or place the first NF on an ingress pipe and the second NF on an egress pipe. In both cases, packets transitions across these two NFs are cheap, as

[2]Multiple NFs may access the same data fields in argument hdr and thus incur different types of dependencies, e.g., match, action, or successor dependencies [23].

they require no resubmissions or recirculations. In all other cases, packets will need to be resubmitted/recirculated at least once to receive processing from both NFs.

One may imagine that this could be done by placing NFs one by one by order of their indexes, alternating between ingress and egress pipes. However, this naïve scheme usually results in sub-optimal placements. For instance, Fig. 6(a) shows an example of placement under the SFC policy A-B-C-D-E-F. Without loss of generality, we assume that packets should be eventually forwarded to a port on Egress 0 after finishing the service chain. In this example, both AB and EF are composed sequentially. According to Tofino's recirculation constraints, packets will have to traverse the NFs in the order of Ingress 0→Egress 0→Ingress 0→Egress 1→Ingress 1→Egress 1→Ingress 1→ Egress 0, causing three recirculations. Fig. 6(b) shows an improvement by exchanging the locations of C and EF. The traversal order then becomes Ingress 0→Egress 1→Ingress 1→Egress 0, requiring only one recirculation. Therefore, we believe that interesting optimization opportunities exist to reduce packet recirculations. In practice, there could be multiple chains. Each chain may require packets traversing the NFs in a different order, which adds another layer of complexity to NF placements.

Therefore, a general optimization model is needed to decide where each NF should be placed and when and how NFs should be composed to share the same pipelet. We envision that each SFC policy may carry a weight reflecting the percentage of traffic following that chaining policy. Our goal is then to minimize the weighted sum of the number of recirculations for all service chains. The input to this model would include the number of pipelines n, the number of NFs m, the set of chaining policy P, and the weight of each policy w. Developing such an optimization model is part of our ongoing work.

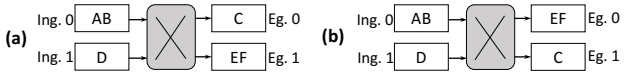

Figure 6: Two NF placement schemes under the SFC policy A-B-C-D-E-F.

3.4 On-Chip Packet Routing

Switch ASICs provide only basic recirculation/resubmission primitives and do not support flexible on-chip routing in the language interface. Dejavu builds on these primitives and employs an on-chip routing mechanism that faithfully routes packets through their desired NFs. To do this, we insert a branching table in the last MAU stage of all ingress pipelet, which directs packets to their next NFs based on the service path ID and index in the SFC header. If the *outPort* of a packet is already set, the branching table will directly

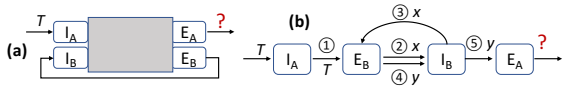

Figure 7: Illustration of the packet recirculation paths.

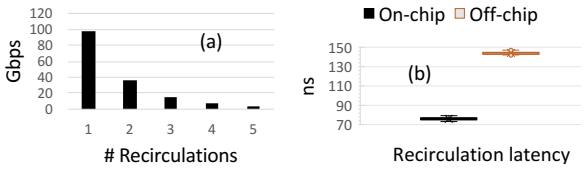

Figure 8: Throughput and latency impact of packet recirculations.

forward the packet to the port and bypass the egress pipe. Routing rules of this table can only be installed after NF placement. Depending on the next NF location, a packet can either be resubmitted to the same ingress pipe or switched to an egress pipe via the traffic manager. After the egress processing, a packet can be recirculated or sent out depending on whether the egress port is in loopback mode.

4 UNDERSTANDING RECIRCULATION

In this work, we advocate the use of packet recirculation as a "first class" functionality. It is necessary to understand the impact of recirculation on the service chaining performance. On Tofino, each pipeline provides 100 Gbps recirculation bandwidth for free via a dedicated recirculation port beside the normal Ethernet ports. However, this is not enough if a large fraction of the traffic (e.g., Tbps) needs recirculation. To provide extra bandwidth, we set a number of the Ethernet ports in loopback mode. On Tofino, a loopback port can no longer take external traffic and bounces all packets back into the ingress pipe.

Throughput. In general, we can analyze the throughput of recirculated packets using the following model. If m out of n Ethernet ports are in loopback mode, we can offer $\frac{n-m}{n}$ of the ASIC capacity for external traffic where $\min(1, \frac{m}{n-m})$ of the traffic can recirculate for once. However, this linear reduction of effective throughput will not hold if traffic needs multiple recirculations. Fig 7(a) shows an example of two Ethernet ports A and B, each with bandwidth T. Each also has an ingress (I_A/I_B) port and an egress port (E_A/E_B). We set Ethernet port B in loopback mode so that the switch can offer 50% of its original capacity. Packets coming from I_A will traverse different paths depending on its recirculation requirements. Under such setting, it can be seen that both the no-recirculation path (I_A–E_A) and 1-recirculation path (I_A–E_B–I_B–E_A) will have throughput T. However, when packets need multiple recirculations, the switch buffer will form a feedback queue where recirculated packets for the first time will compete for bandwidth with recirculated packets for

the second time on E_B, as shown in Fig 7(b). Assume the throughputs of steps 2 and 3 are x and y, respectively, then we have:

$$y + x = T; y = xT/(T + x)$$

where $T/(T+x)$ is the packet loss rate at E_B due to congestion. Solving the above equations gives us $x = 0.62\,T$. The effective throughput of step 5 is then $T - 0.62\,T = 0.38\,T$. Following the same feedback queue model, we can also obtain the effective throughput of the traffic throughput with 3-recirculation as $0.16\,T$.

We conduct testbed evaluation to confirm the throughput impact on a Tofino ASIC under the same setting as in Fig. 7(a). We use Tofino's internal packet generator to inject 100 Gbps traffic into one Ethernet port, which is then recirculated for several times before being sent out of the switch. Fig 8(a) shows the throughput reduction with an increasing number of recirculations. The results match our calculations well, and show that the effective throughput degrades super-linearly with the number of recirculations.

Latency. Recirculation adds extra latency for packet processing. This latency can be obtained by calculating the difference between the timestamps of packet leaving the egress pipeline and entering the ingress pipeline. On-chip packet recirculation is usually via dedicate circuitry on the chip without serialization/de-serialization, meaning that the extra latency should be very small. Fig 8(b) shows that this latency is ~75 ns, about 11.5% of the measured port-to-port latency without recirculation(~650 ns under idle switch buffer). We also measure the off-chip recirculation latency via a direct attach cable (1 m), which includes both the serialization/de-serialization delay and the propagation delay. As is shown, it is about 70 ns slower than on-chip recirculation.

Takeaways: Our analysis and benchmarks on the impact of recirculation indicate that: 1) recirculation does impact effective throughput significantly and an NF placement algorithm that minimizes the number of recirculations is critical for the overall SFC performance; 2) network operators can expect and calculate the throughput of their service chains after placement—the ASIC itself does not introduce any inefficiency on recirculation throughput; 3) recirculation latency is relatively small compared to the port-to-port hop latency and on-chip recirculation is 2× faster than off-chip recirculation.

5 INITIAL VALIDATION

We have built a preliminary Dejavu prototype using the SFC example in Fig. 2. Our testbed consists of a Wedge-100B 32X switch with a Tofino chip with 32×100 Gbps Ethernet ports and 2 physical pipelines (4 pipelets). Each pipeline has 16 hardwired Ethernet ports. We connect two servers to the switch via 25 Gbps breakout cables, one as the sender and the other as the receiver.

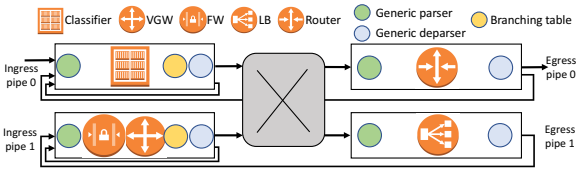

Figure 9: A prototype implementation of Dejavu SFC on aTofino ASIC with 2 pipelines and 4 pipelets.

Table 1: Resource overhead of Dejavu on Tofino.

Computational					Memory	
Stages	Table IDs	Gateways	Crossbars	VLIWs	SRAM	TCAM
20.8%	4.2%	2%	0.4%	1.5%	0.2%	0%

Placement. Fig. 9 shows the placement of NFs on 2 pipelines (4 pipelets) produced by Dejavu. Here, we put the 16 Ethernet ports of ingress 1 into loopback mode. Hence ingress pipe 1 takes no external traffic and is used only for recirculation, and traffic sent to egress pipe 1 will be automatically recirculated to ingress pipe 1. In this setting, our switch can provide 1.6 Tbps capacity and allow all the traffic recirculate on the ASIC for once. We test the input and output packets of multiple SFC paths using the Packet Test Framework [11] and have verified that the placement and routing logic in our example successfully achieve the original functionalities.

Resource overhead. There are three types of match-action tables: the branching table, the "check_next_hop" table and the "check_sfcFlags" table. The branching table and the "check_next_hop" table each has an entry for each (pathID, serviceIndex) pair, and the "check_sfcFlags" table has an entry for each field of the platform metadata. These tables are independent of the traffic and their sizes are determined at compile time. Dejavu tables are placed in separate MAU stages due to data dependencies on the platform metadata, and they consume 20.8% of the stages in total (shown in Table 1). However, Dejavu does not use the stages exclusively. NF tables that are independent of Dejavu tables can comfortably share the same stages with Dejavu. Due to the simple logic and bare-minimum table sizes, we observe negligible overheads for other types of resources.

6 RELATED WORK

Service Function Chaining. Most existing works on SFC consider a software-based architecture for chaining network functions, such as NetBricks [35], Click-OS [30], NetVM [21], and Hyper-Switch [39]. To increase performance under this software-based architecture, researchers

have proposed adding different kinds of hardware accelerators to servers, such as GPU [19], FPGA [28] and Network Processors [43]. In contrast, Dejavu uses programmable switches for high performance SFC. We provide a unified platform that shields the complexity of switch targets from NFs, as well as a programming model that can compose multiple NFs together.

Data Plane Multiplexing. Hyper4 [20] and HyperV [46] virtualize the switch data plane by running a general-purpose P4 program that can be configured to emulate the behavior of multiple P4 programs. However, these approaches require significantly more hardware resources (3-7×) compared to the native programs. P4Visor [47], P4Bricks [42] and P4SC [6] merge multiple P4 programs at the code level and achieve much higher resource efficiency. However, since they do not assume knowledge of the underlying hardware, they could either miss opportunities provided by new hardware features or generate programs that violate hardware constraints. Dejavu, on the other hand, can perform optimizations for program composition under hardware constraints.

7 DISCUSSION AND FUTURE WORK

Hardware restrictions. Dejavu relies on, and inherits a set of restrictions from the underlying hardware platform it runs on. First, the current generation of programmable ASICs cannot access packet payload easily. The implication is that NFs that require payload processing (e.g., DPI, WAN optimizer, NIDs) are not supported. Second, the NFs are subject to the inherent resource limitations of the hardware, such as table sizes, the number of MAUs, and others. These hardware restrictions will likely loosen over time with the advances of next-generation programmable ASICs. In the meantime, optimizations that can best leverage the on-chip hardware resource to implement more advanced NFs, and methods that can efficiently interpose Dejavu with other off-chip NFs are still active research directions.

Control plane merge. So far, we have mostly focused on merging the data plane programs to form a chain of functions. However, another important component would be to also merge the control plane logic for the respective data plane programs. Such logic is typically implemented in software. To merge the control plane software of different network services, a translation layer, as proposed in P4Visor [47], is likely needed to map the original control plane APIs to use the newly generated SFC APIs. So this should be achievable with a minimal amount of modification to the actual control plane source code.

Towards clusters of switch data planes. In this work, have we focused on running SFC on a single switch. However, there are certainly cases where a particular combination of NFs cannot fit in the pipelines of a single switch. We note that Dejavu can be readily extended to support NF placement across multiple switches. In the simplest case, multiple switches can be chained back-to-back to provide the same bandwidth of a single switch but with manyfold more MAU stages. This gives our placement algorithm high flexibility to accommodate more complicated NFs. Our off-chip recirculation latency in Fig 8(b) also reflects that the packet transition delay from one switch to another is low enough to be practical, although it is slightly higher than on-chip recirculations. Generally speaking, an extended placement algorithm that optimizes for throughput and latency needs to consider several factors: NF affinity, switch locality and cluster topology. Such an algorithm draws a parallel to a wide range of existing work on VM/container/NFV scheduling on compute clusters [15, 27, 29, 34, 44, 45], though the data plane programs have a much higher loading cost and should be operated at a relatively larger timescale.

Implications for hardware/compiler designers. Dejavu introduces a new resource utilization model, which sheds light on the design of future programmable switch ASICs. In particular, current ASICs can only support packet recirculation at per-port granularity. And recirculation decisions must be made in the ingress pipe. If recirculation decision can be done at per-packet granularity (e.g., packets can choose to be recirculated or be sent out of the switch after egress processing), we would not only have fine-grained control over the traffic that needs recirculation, but also more flexible function placement and potentially fewer recirculations in the pipelines. At the language level, the current P4 compilers do not provide direct support for multi-programming. Existing works advocating data plane multiplexing [42, 47, 48] have to merge multiple programs at the source code level to achieve more sophisticated functions. A language that can natively support multi-programming model will significantly improve the modularity and reusability of data plane programs for SFC.

Implications for network operation. Running a service function chain on a shared switch ASIC brings significant changes to how network operators manage and operate the network services. There are several interesting research problems in this space, such as service upgrade and expansion, failure handling, and troubleshooting. Solving these operational challenges can have significant impacts on the wider adoption of programmable network devices.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers of HotNets'19 for their thoughtful feedback. This research was sponsored by the NSF under CNS-1815525, CNS-1801884 and CNS-1718980.

REFERENCES

[1] Barefoot. 2019. Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino-2/. (2019).

[2] Barefoot. 2019. P4STUDIO Architecture. https://www.barefootnetworks.com/products/brief-p4-studio/. (2019).

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[4] 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 *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 99–110.

[5] Doruk Bozdag, Fusun Ozguner, and Umit V Catalyurek. 2009. Compaction of schedules and a two-stage approach for duplication-based DAG scheduling. *IEEE Transactions on Parallel and Distributed Systems* 20, 6 (2009), 857–871.

[6] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou. 2019. P4SC: Towards High-Performance Service Function Chain Implementation on the P4-Capable Device. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 1–9.

[7] P4 Language Consortium. 2019. P4 Language Specification. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf. (2019).

[8] P4 Language Consortium. 2019. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html. (2019).

[9] P4 Language Consortium. 2019. $P4_{16}$ Portable Switch Architecture. https://p4.org/p4-spec/docs/PSA-v1.1.0.html. (2019).

[10] P4 Language Consortium. 2019. P4_16 Prototype Compiler. https://github.com/p4lang/p4c. (2019).

[11] P4 Language Consortium. 2019. Packet Test Framework. https://github.com/p4lang/ptf. (2019).

[12] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer.. In *NSDI*. 523–535.

[13] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud Scale Load Balancing with Hardware and Software. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2619239.2626317

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling innovation in network function control. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 163–174.

[15] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 99–115. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog

[16] A Greenberg. 2014. Windows azure: Scaling SDN in the public cloud. *Open Networking Summit (ONS)* (2014).

[17] Adam Greenhalgh, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. 2009. Flow processing and the rise of commodity network hardware. *ACM SIGCOMM Computer Communication Review* 39, 2 (2009), 20–26.

[18] Joel Halpern and Carlos Pignataro. 2015. Service function chaining (sfc) architecture. *IETF, RFC7665* (2015).

[19] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. ACM, New York, NY, USA, 195–206. https://doi.org/10.1145/1851182.1851207

[20] David Hancock and Jacobus Van der Merwe. 2016. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 35–49.

[21] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 445–458. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang

[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*. ACM, 121–136.

[23] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 103–115.

[24] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless network functions: Breaking the tight coupling of state and processing. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 97–112.

[25] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. 2018. Metron:{NFV} Service Chains at the True Speed of the Underlying Hardware. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 171–186.

[26] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan J Jackson, et al. 2014. Network Virtualization in Multi-tenant Datacenters.. In *NSDI*, Vol. 14. 203–216.

[27] Sameer G Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, KK Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. 2017. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 71–84.

[28] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 1–14.

[29] Y. Li, L. T. X. Phan, and B. T. Loo. 2016. Network functions virtualization with soft real-time guarantees. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524563

[30] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 459–473.

[31] 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 Conference of the ACM Special Interest Group on Data Communication*. ACM, 15–28.

[32] Adam Morrison, Lei Xue, Ang Chen, and Xiapu Luo. 2018. Enforcing Context-Aware BYOD Policies with In-Network Security. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'18)*.

[33] Recep Ozdag. 2012. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. *See goo. gl/AnvOvX* (2012), 5.

[34] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/2815400.2815423

[35] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of {NFV}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 203–216.

[36] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/2486001.2486026

[37] Paul Quinn, Uri Elzur, and Carlos Pignataro. 2018. Network service header (nsh). *IETF, RFC 8300* (2018).

[38] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 227–240.

[39] Kaushik Kumar Ram, Alan L Cox, Mehul Chadha, and Scott Rixner. 2013. Hyper-switch: A scalable software virtual switching architecture. In *2013 USENIX Annual Technical Conference (ATC'13)*. 13–24.

[40] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 4 (2009), 14–23.

[41] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. 2012. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 24–24.

[42] Hardik Soni, Thierry Turletti, and Walid Dabbous. 2018. P4Bricks: Enabling multiprocessing using Linker-based network data plane architecture. (Feb. 2018). https://hal.inria.fr/hal-01632431 working paper or preprint.

[43] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. 2001. Building a robust software-based router using network processors. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 216–229.

[44] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.

[45] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.

[46] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. 2017. HyperV: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–9.

[47] 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

[48] Yu Zhou and Jun Bi. 2017. Clickp4: Towards modular programming of p4. In *Proceedings of the SIGCOMM Posters and Demos*. ACM, 100–102.