# Adaptive Switch: A Heterogeneous Switch Architecture for Network Centric Computing

17:02, February 28, 2020

***Abstract*—To support growing throughput, big data volume, and information complexity in data centers, processing and computing are being offloaded and disaggregated to network. The paradigm of such a network-centric computing was pioneered from SmartNIC by introducing adaptive intelligence at host network with customized processing and proprietary protocols. In this paper, we push adaptive intelligence further to network switches and propose a new hardware architecture, Adaptive Switch. Based on prototypes and use cases we implemented, we demonstrate the combined advantages of legacy Switching functions and FPGA to achieve high throughput and Agility simultaneously in the proposed architecture.**

***Index Terms*—Switch, FPGA, Programmable Data Plane, heterogeneous Architecture**

## I. Introduction

The need to transmit massive data under stringent latency requirement keeps growing and has pushed CPU to the limits of its scalability in modern data centers. As a result, In-Network Computing or Network-Centric Computing emerges, which offloads and disaggregates computing, storage and other functions through networks.

SmartNIC is known as the first attempt offloading computing from CPU to Network Interface Card (NIC), which has been increasingly deployed in data centers [1]. SmartNIC provides an more efficient processing model than CPU-attached acceleration boards by eliminating unnecessary dataflow copying between CPU and acceleration board.

In this paper, we extend the SmartNIC approach from the server network further to network core by introducing an Adaptive Switch architecture to support user specific processing as a cure to the scalability and flexibility challenges in Data Centers (DCs).

### A. Motivation

The pioneering work to introduce the programmability for a switch was OpenFlow switch [2], which was later developed to Protocol Switch Architecture (PSA) [3]. A PSA compatible target is complied from Protocol-independent Packet Processing (P4) language [4], which can flexibly define data plane for proprietary networking protocols and processing.

Our ambition is a much broader than the definition of PSA, which aims at a new switch architecture not only programs the network, but also supports disaggregation of computation/storage into the network. There are several limitations of PSA, which motivates our work in this paper.

First, PSA triggers processing only on packet arrivals or/and departures. It does not support algorithms and applications that run periodically or triggered by other events. NDP [10] is a particular motivating example, in which, an intermediate switch should react when its buffer is congested. And more detailed experiment results of the NDP case will be demonstrated in Section IV.

Second, P4 language defined over PSA is not rich enough to satisfy flexible computation requirement, even the complete support to algebraic calculation. As a result, many networking algorithms for packet scheduling and network measurement need to seek for new specialized hardware architecture. Here we use DISCO [9], a flow statistics approach, as a use case to evaluate Adaptive Switch in Section IV.

Third, PSA is designed for stateless processing, however, many networking protocols (*e.g.*, TCP) and applications (*e.g.*, firewall, instruction detection) require stateful processing depending on historical states. To better understand this motivation, a firewall [5] use case in Section IV is studied based on Adaptive Switch.

At the language level, P4 in its latest version has introduced P4_extern, which can be used to describe the unsupported features in standard P4; however, there is no unified switch architecture as "PSA_extern" to map processing defined by P4_extern. Without such a hardware architecture, we need to have a specialized hardware target each time when adding a new description of P4_extern, and such a way actually conflicts with the initial mindset of keeping programmable data plane unchanged.

Regarding to P4 targets, Barefoot Networks (recently acquired by Intel) releases a P4/PSA compliant ASIC named Tofino [8], which can achieve up to 12 Tbps throughput. It supports P4 programming to define the data plane, but is still not flexible enough to fit into the motivation examples mentioned before. Software-based P4 targets, *e.g.*, bmv2, would be much easier to add p4-extern features to the targets, as the specialized functions in data plane. However, it comes with an obvious disadvantage of low performance: typically, the throughput is at most 100Gpbs and the latency is 10X-100X more than switch ASIC. In addition, a pure Field-Programmable Gate Arrays (FPGA) PDP target cannot suffice the high throughput requirement for DCNs.

### B. Our Approach

In this paper, we innovate the switch architecture to have "unlimited" support to any P4_extern defined program, by a heterogeneous hardware architecture called Adaptive Switch consisting of a Switching System (SS) part and a Programmable Logic (PL) part. The SS part can be a traditional

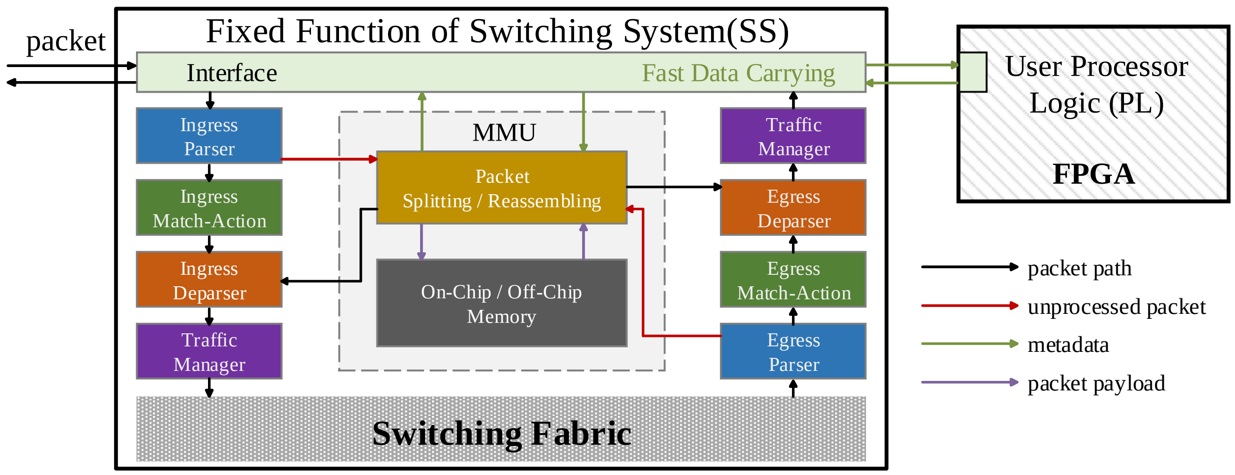

Fig. 1. Architectural Diagram of Adaptive Switch.

switch ASIC with or without P4 compliance, and the PL can be simply FPGA or MPSoC/ACAP integrating FPGA with other processors (e.g., Arm cores, GPU) in a single on-chip system. Each incoming packet enters SS first for processing, and only the one relying on the functions unsupported by SS, will be further sent to PL for further cooperative processing.

We have solved two technical challenges in the paper. The first one is described in section II, which is the architectural design of Adaptive Switch. The second challenge is how to develop and map programs to the Adaptive Switch target, which is elaborated in Section III. To evaluate the proposed switch architecture, we have implemented the three aforementioned motivating cases in Section IV. We conclude the paper in Section V.

## II. Architectural Design

Fig. 1 is the block diagram showing the hardware architecture of Adaptive Switch. It consists of a fixed Switching System (SS) part and a user Programmable Logic (PL) part. The SS part accommodates processing of a standard switch functions and customized processing would be programmed in PL. The two parts, SS and PL in the proposed architecture can be implemented with a **Single-Chip** solution, which integrates PL and SS, or with a **two-chip** approach having a Switching ASIC and a FPGA/MPSoC/ACAP chip separately. The single-chip solution is consistent with the technical strategy of MPSoC and ACAP, which improves the processing capacity and the power efficiency. SS and PL can be connected via a high bandwidth on-chip Bus (*e.g.*, AXI). For the two-chip solution, two physical chips are connected using PCIe or Ethernet interface.

As shown in the left side of Fig. 1, the core component of the switching system is the switching fabric, which usually adopts cross-bar for high speed switching from ingress ports to egress ports. Packets coming from a network interface traverse through an ingress pipeline before the switch fabric and an egress processing pipeline after the switch fabric. In each pipeline, there are usually parsers (extracting header fields of interest), flow tables (matching the extracted headers for actions executions), deparser (reassemble or/and manipulate packets), and traffic manager (traffic management processing like buffer scheduling, shaping, *etc*.). State-of-the-art switching ASIC usually has these components, but to form an adaptive switch depicted in Fig. 1, we need to have an additional Memory Management Unit (MMU), which is used to buffer the packet waiting for processing in PL. Only a packet seeking further processing in FPGA will trigger MMU functions including: 1) dynamically allocation of a memory block for keeping the packet payload; 2) writing the packet payload into memory; 3) reading out the packet payload from memory and assemble the packet with returned metadata from PL.

If an incoming packet needs further processing in PL, SS stores such a packet in on-chip/off-chip memory and only sends the metadata with extracted information from the packet to PL, where the metadata is customized for the processing needs in PL. After the specialized processing in PL, an updated metadata with results returns to SS. SS then combines the original packet and the returned metadata into a complete packet for actions like forwarding/dropping.

We design the Adaptive Switch architecture based on two observations or assumptions. First, not all traffic would require specialized processing in a switch. Otherwise, the related fucntions would become off-the-shelf with a switching ASIC or can be directly move the SS part of the proposed Adaptive Switch. Second, most processing is usually based on packet header only or the packet segment of header plus the first a few bytes in the payload. Only in very rear cases, a processing would look into the entire packet. It makes proper and efficient to only exchange metadata for the packets of interests, which guarantees a limited interconnect bandwidth consumption between SS and PL.

The metadata always includes a packet ID (PID) to identify each packet, a flow ID (FID) to identify flow, and packet header fields or/and the first $0 < x < L$ bytes of the payload ($L$ is the length of the payload). Since the average packet length is around 600 bytes, in the worst case of a 12 Tbps switching ASIC (the largest throughput of an off-the-shelf switching ASIC), a metadata of 64 bytes allows 20% of the traffic to be further handled in FPGA if interfacing through PCIe Gen4X16 without sacrificing port density. Besides PCIe, the physical connection between switching system and FPGA can be Ethernet ports/transceivers, which are able to support 100% original traffic to have a second FPGA processing stage, at the cost of reducing port density by about 10%, again in the condition that the average packet length is 600 Bytes and metadata is 64 bytes.

## III. User Specific processing in PL

### A. Basic Design Flow

PL is enabled by FPGA and its design flow is shown as the following.

1) Determine packet processing requirements and the dataflow model.

2) Write processing specific functions and processing. Either high level languages, *e.g.*, P4/P4extern, or hardware description languages, *e.g.*, Verilog HDL, can be used to program the user specific processing. In addition, a data plane builder generator, e.g., SDNet [6], [7] is also efficient to fast build the user specific processing on PL/FPGA.

3) Compiling to PL. With different ways to program the customized processing, the compiling procedures can be different.

We use Xilinx SDNet/P4-SDNet as a basis of the development flow, and showmore general compiling model in Fig. 2 for mapping user design to PL. SDNet [6] and P4-SDNet [7] are commercial off-the-shelf (COTS) products, which cover the compiling tool chain from P4 and SDNet Specification to Verilog Engines in FPGA based data plane. P4-SDNet supporting $P4_{16}$ provides two build-in P4 externs for manipulating packet data. Following the similar way, we can extend the development flow by updating compilers to support more externs: extend the front-end to recognize high level descriptions (annotations will be helpful for the compiler back-end to improve the performance), which will be converted to a proper Intermediate Representation (IR), *e.g.*., SDNet Specification; and lastly be mapped to PL/FPGA. In this paper, we add a new "mapping optimization" phase to the original back-end of P4-SDNet/SDNet to maximize the performance gain by mapping to parallel processing pipelines in PL.

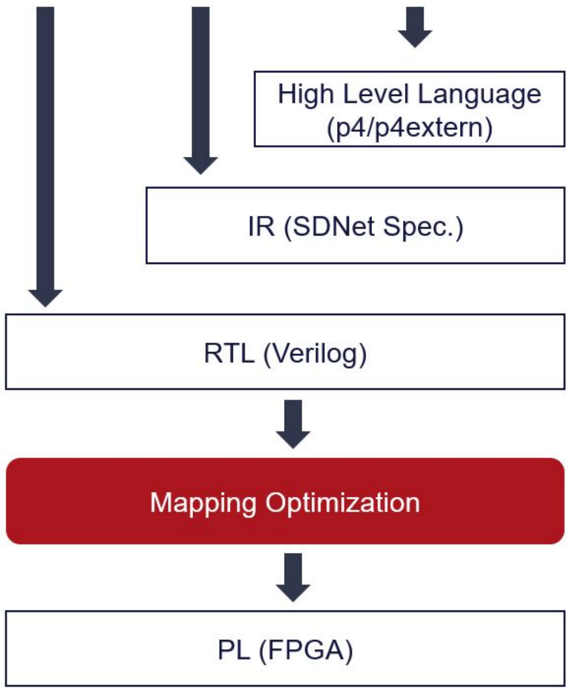

Fig. 2. Development Flow.

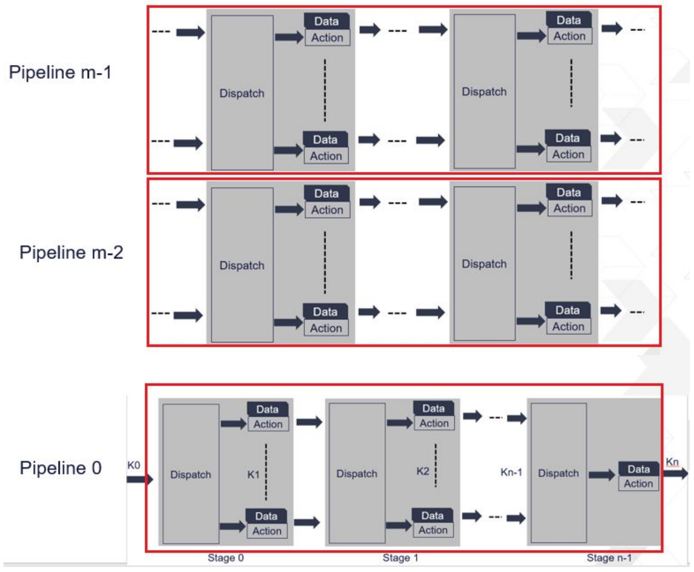

Fig. 3. A General Model of Parallel processing Pipeline in PL.

### B. Parallel Processing Optimization

FPGA based PL enables reconstructing micro-level parallel processing for user specific dataflow-based computing. We achieve efficient mapping to PL by 1) introducing an parallel processing architecture and 2) compiling optimization to reduce resource footprints.

The user specific processing is mapped into a parallel architecture as depicted in Fig. 3. From the top view, there are $m(m \geq 1)$ processing pipelines and $n(n \geq 1)$ stages in each pipeline. In addition, parallelism is also introduced into pipeline stages as shown in the shaded boxes, each of which we call a "Basic Processing Unit (BPU)". User specific processing is divided into different "execution engines", each of which is consist of a *Action* module operating on a set of *data* kept in memory. The number of inputted data-flows to a BPU is the same as the number of execution engines in its predecessor BPU, *i.e.*, in a generic model, we denote $K_i$ inputs in stage $i$ and $K_{i+1}$ outputs in stage $i$, and based on the needs, $K_i$ and $K_{i+1}$ are not necessarily the same. The *Dispatch* module in a BPU for stage $i$ is to distribute the $K_i$ data-flows into $K_{i+1}$ execution engines, as evenly as possible.

Combining the development flow in Section III-A and the parallel architecture in Fig 3, high level languages, SDNet Spec. and RTL languages can be used to define the execution unit and the parameters $(m, n, K_i)$. If we duplicate the entire data into each paralleled execution units and/or processing pipelines, we could simply achieve largest throughput by distributing the packet in a simple round-robin strategy, however, it obviously consumes too much memory, which limits the adoption to data-intensive processing.

The objective of the optimization is to balance the workload sending to each processing pipeline, while holding dataflow affinity and less data duplication. For an efficient mapping of user defined processing into the parallel architecture in PL, there are several aspects need to be considered when instantiating dataflow-based processing.

1) Processing replicas. Introducing more replicas for a single action/processing leads higher throughput and lower delay, requiring more logic resource in PL.

2) Data copies. Data dependent action/processing requires access of data in its replicated execution unit or/and pipeline. Introducing more data duplicates mitigates data access conflicts with the cost of more (on-chip) memory.

3) Dataflow affinity. All the packets from the same flow should be processed by the same execution unit and/or processing pipeline in order to keep the processing dependency and avoid the out-of-order problem. *Dispatch* module enables flow affinity distribution for packets over paths (execution units) based on a hash distribution.

Our idea is to split the data according to the dataflow affinity restriction. It is only needed to keep the data for processing the dataflow, which would be distributed to the execution engines. Small data redundant is also allowed among execution engines in order to make the dataflow distribution more balanced and keep flow affinity when using wildcard-based matching table (*e.g.*, longest prefix matching table, TCAM). In fact, this problem can be reduced from the classic "Subset-Sum" problem, which is with NP-Complete hardness. We use a simulated annealing based heuristic algorithm to find the data split solution.

### C. Runtime Reconfiguration on Dispatch

To maintain a high processing performance, it is important to always keep load balanced by the Dispatch module. In practice, the traffic distribution used as the input of the heuristic algorithm will change over time, so a remote controller is used to collect the workload variance among execute engines as feedback for runtime recalculation and configuration. The calculation time of the algorithm is evaluated in the next Section IV.

## IV. Evaluations

### A. Implementation and Evaluation Settings

We have implemented the proposed Adaptive Switch architecture on FPGA (4000+ lines of Verilog HDL code) including the fixed SS part and the user logic for three use cases in the PL part. We use ZC706 development board as the platform for our implementation and evaluations.

A packet generator is integrated in FPGA to inject input traffic based on 2 real traces from different sources including a **ISP** (WIND) trace, a trace collected from **LTE** base station, as well as 3 synthetic traces which obey **Exponential** distribution, **Pareto** distribution and **Uniform** distribution, respectively.

### B. Use Cases

To show the flexibility of the proposed adaptive switch, we further implemented three use cases on PL to echo the motivation listed in Section I-A.

1) **Congestion Control.** NDP is a data center traffic congestion control algorithm [10], which ensures a very low forwarding delay of small batch packets when congestion is detected in a switch. As we mentioned in Section I, NDP is an example of specific event driven processing, which is not well supported by exiting switches.

To deploy NDP to adaptive switch, We pre-allocate the logical output queue in the MMU of the SS part. A user specific logic is designed to detect the queue depth as the congestion trigger signal. Then, another use processing of NDP is to send a packet for notifying sender to adjust the sending packet size. Considering the delay to read queue length from SS by PL, our design will get a buffer size estimation error, which is less than 0.5% on average.

We have implemented the NDP processing using Verilog HDL language with 931 lines of code. The front-end compiler wrapped the user logic into the *Action* module for generating a BPU and processing pipelines. Xilinx Vivado is used to synthesize and the HDL code and generate a bitstream file.

2) **Network Measurement.** DISCO is a flow statistic algorithm for network measurement [9], which is based on complicated calculation like exponential functions, random selection, logarithmic operation, etc. It makes DISCO difficult to be implementing in a Commercial-Off-The-Shelve switches including P4 PDP compatible switches.

The input data of DISCO engine is the flow ID and the length of each incoming packet. The output data of DISCO is the counter value, which is kept in the on-chip storage with a memory like interface. We allocate $K$ DISCO engines in each paralleled processing path in FPGA. After a regular time period, the controller collects measurement results from all DISCO engines at the same time.

We have implemented the DISCO algorithm in PL using Verilog HDL language (1022 lines of code) as an P4_extern function to be called by p4 languages.

3) **Stateful Firewall.** With 584 lines of code, we have also implemented a stateful forwarding engine in the form of P4_extern function in Adaptive Switch. The firewall engine records the states of the connections for filtering packets. There are two hardware flow tables in the engine. One is used for fundamental Matching-Action operation, and the other stores a list of states for each corresponding flow. When every packet comes, it executes actions for according to the current state and the packet matching results, and meanwhile, it will updates the Matching-Action table to the next data state.

### C. Performance Results

We simplify the implementation of the parallel processing model in Fig. 3, with one pipeline, one stage, but multiple

TABLE I
THE IMPLEMENTING RESULT OF A SELECTED SET OF KEY ELEMENTS IN ADAPTIVE SWITCH

| | LUT | LUT RAM | FF | BRAM | DSP |
|---|---|---|---|---|---|
| Dispatch (with 4 layer3-parsers, 4 equalizers and one 4x4 crossbar) | 1380/0.64% | 216/0.28% | 3890/0.91% | 3/0.54% | 0 |
| Case: congestion control,8 priority queues for 4 output ports | 1844/0.83% | 1937/2.80% | 2621/0.59% | 47/8.68% | 0 |
| Case: Measurement (with 4 EM-based flow classifications and 4k counters) | 4492/2.07% | 2017/2.92% | 4089/0.95% | 47/8.68% | 4/0.44% |
| Case: stateful fierwall (with 4 Prog. states Trans. tables (each flow with 8 states)) | 908/0.43% | 1937/2.80% | 1725/0.43% | 43/7.92% | 0 |

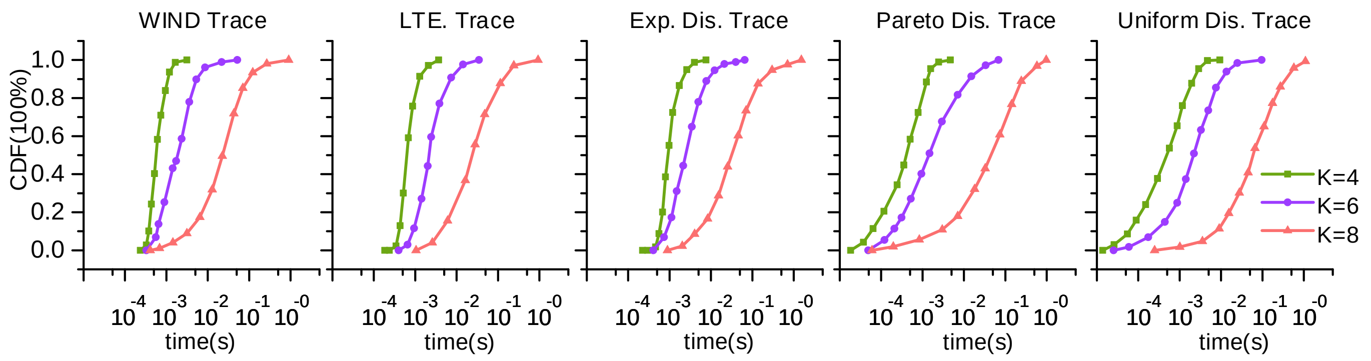

Fig. 4. The runtime performance of heuristic algorithm for load-balanced entry allocation. The performance is indicated by the calculation time(s).

execution engines. *i.e.*, our implementation of use cases using one BPU with different settings of the number of execution engines.

Table. I shows the PL resource consumption. The **Dispatch** row shows the resource for one Dispatch module in BPU, where we use SDNet generated parser/matching table and 4x4 cross-bar architecture to distribute dataflow. The rows list the results for different user cases, which all use four execution engines for user specific processing. The resource consumption is almost linear with the number of execution engines and number of entries adopted. From the results in the table, we observe that a typical FPGA is sufficient to have the three user cases deployed. In these three cases, each engine contains a EM-based matching table (1k entries) as data storage. Besides, we deploy 4k capacity counters for measurement case; in firewall case, we implement a programmable states transition table for each flow; and in congestion control (NDP) case, we build 8 priority queues for 4 output ports and the queue size for low priority queues is set to 64KB size, which requires quite large BRAMs.

In Fig. 5, we depict the trend of throughput when increasing the number of execution units. We calculate the throughput based on the maximum clock frequency, data bus width, and the average packet size (600 Bytes) learnt from real traces. The figure shows a curve which is very close to linear curve. As aforementioned, the resource utilization is not very high, so the timing constraints are not very hard to meet and maximum clock frequency does not change too much.

By looking into the cycles need for the processing, We get the PL processing delay report for the three use cases. There are 0.142$\mu s$, 0.130$\mu s$, 0.136$\mu s$ for the three use cases of network measurement, firewall and congestion control, respectively.

We have also implemented the PL mapping optimization

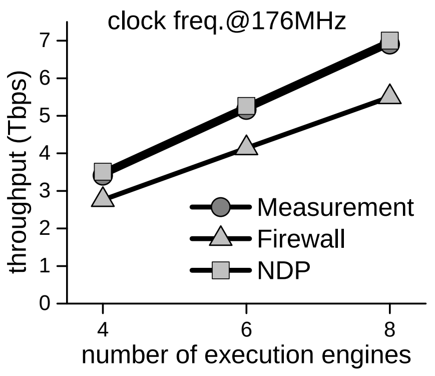

Fig. 5. Performance estimation in different number of execution engines. The maximum clock frequency is 176MHz, which is limited by the timing performance of matching tables.

algorithm for runtime configuration with 300+ lines of python code, which is deployed on a Dell R620 server (with 8G RAM and 2.80GHz Quad Core Intel CPU in Ubuntu 16.04 LTS OS). We use PyPy toolset to accelerate the program, benefiting from JIT (Just-in-Time) compiler in PyPy. Fig. 4 shows the Cumulative Distribution Function (CDF) of calculation time under five generated traffic patterns described in the experiment settings. The three curves in each of the figure represent the different settings of the number of execution engines in the processing pipeline, *i.e.*, 4, 6, 8 execution engines. It takes longer time for runtime mapping optimization for more complicated processing pipelines (more execution engines). The calculation time is less than 0.5s In 90% of the time, and less than 1.12s in 100% of the time.

## V. CONCLUSION

We present an effective Adaptive Switch architecture for network-centric computing, which obtains high performance and agility. The key insight behind Adaptive Switch is leveraging the switch fabric to provide high throughput while offloading the programmable processing to FPGA. The Adaptive Switch architecture adapts the external function that is limitedly supported by general programmable network data plane. We guarantee the resource consumption of parallelism under control, while making the processing throughput at T bps (the same magnitude with legacy switching ASIC). Although the processing throughput in PL part is still less than than throughput of switching ASIC, we argue that not a entire packet and not all dataflows need to be processed with specific processing in PL and it makes the overall throughput of an adaptive switch still comparable with COTS switch.

## References

[1] Lu, Guohan, et al. "Serverswitch: a programmable and high performance platform for data center networks." Nsdi. Vol. 11. 2011.

[2] McKeown, Nick, et al. "OpenFlow: enabling innovation in campus networks." ACM SIGCOMM Computer Communication Review 38.2 (2008): 69-74.

[3] The P4.org Architecture Working Group, Portable Switch Architecture (PSA), https://p4.org/p4-spec/docs/PSA-v1.1.0.html

[4] Bosshart, P., & Daly, D. (2014). P4: Programming Protocol-Independent Packet Processors. ACM SIGCOMM Computer Communication Review, 44, 18.

[5] Zerkane, Salaheddine, et al. "Software defined networking reactive stateful firewall." IFIP International Conference on ICT Systems Security and Privacy Protection. Springer, Cham, 2016.

[6] Xilinx. "SDNet Packet Processor." User Guide. (6-15-2017). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/UG1012-sdnet-packet-processor.pdf

[7] Xilinx. "P4-SDNet Translator." User Guide. (5-15-2017). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1252-p4-sdnet-translator.pdf

[8] Barefoot Networks. Barefoot tofino. https://barefootnetworks.com/products/breif-tofino/.

[9] Hu, Chengchen, et al. "Disco: Memory efficient and accurate flow statistics for network measurement." 2010 IEEE 30th International Conference on Distributed Computing Systems. IEEE, 2010.

[10] Handley, Mark, et al. "Re-architecting datacenter networks and stacks for low latency and high performance." Proceedings of the Conference of the ACM Special Interest Group on Data Communication. ACM, 2017.