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

Siyi Qiao, Chengchen Hu, Gordon Brebner, Jianhua Zou and Xiaohong Guan

Abstract—**Network-centric computing offloads and disaggregates computing and data processing from CPU to network in order to support growing throughput, big data volume, and information complexity in data centers. An emerging paradigm is employing SmartNIC for network-centric computing, which introduces user-specific processing at a host's network interface. In this paper, we take this initiative further to tackle current proprietary processing and computing issues in the network core, in switches. We propose a new hardware architecture called *Adaptive Switch*, combining legacy switching functions and FPGA flexibility. Based on testing of a prototype, with three use cases running above the prototype, we demonstrate that both high transmission throughput and processing flexibility can be achieved simultaneously on *Adaptive Switch*.**

Index Terms—**Network-centric computing, programmable data plane, Switch Architecture, FPGA, P4.**

Introduction

THE need to transmit massive data under stringent latency requirements keeps growing and has pushed CPU to its limits of scalability in modern data centers. Offloading networking stack and computing from CPU to Network Interface Card (NIC) has been increasingly deployed in data centers [1]. This approach is called SmartNIC, which leads people to further embrace In-Network Computing or Network-Centric Computing to offload and disaggregate computing, storage and other functions through networks. In this paper, we propose an *Adaptive Switch* architecture to support user-specific processing and proprietary protocols in network cores, to complement the existing SmartNICs in host networks.

The pioneering work to introduce user-defined actions on traffic traversing a switch was the OpenFlow switch [2]. This fairly inflexible data plane was later developed to become the Portable Switch Architecture (PSA) [3]. A PSA-compatible data plane target can be flexibly defined and compiled from the P4 (Programming Protocol-independent Packet Processors) language [4]. There are several limitations of PSA, which motivate the work in this paper.

First, PSA triggers processing only on packet arrivals and/or departures, leaving actions for other events unsupported. For example, in PSA it is hard to enable the control in the NDP [10] proposal, where a switch should react when buffer is congested. A use case of NDP in our experiment is presented later with more details.

Second, PSA is deficient in computational operations (*e.g.*, algebraic calculation), which prevents many algorithms being deployed on a PSA compatible switch. There is a concrete example discussed later on a flow statistics approach (DISCO [9]) leveraging complex calculation.

Third, PSA is originally designed for stateless processing on network packets. Stateful protocol processing and/or stateful computing based on historical states is usually hard with PSA. To better understand this, a firewall [5] use case will be discussed later.

PSA is very suitable for stateless and forwarding-based data plane, but our ambition is much broader than the scope of PSA – we aim at designing a switch architecture to offload and disaggregate computation into a network. At the language level, P4 in its latest version (P4_16) has introduced the notion of the P4_extern to describe any features unsupported in the language's standard form; however, there is no flexible switch architecture with a matching "PSA_extern" for processing defined by P4_extern. A solution is always to have a new specialized hardware target when adding a new P4_extern. However, this conflicts with the initial mindset of PSA as a portable programmable data plane and should be avoided.

S. Qiao, J. Zou and X. Guan are with the Faculty of Electronic and Information Engineering, Xi'an Jiaotong University. S. Qiao was working on this work when he was an intern with Xilinx.

C. Hu and G. Brebner are with Xilinx.

We innovate by introducing a heterogeneous hardware switch architecture to support any possible P4_extern defined processing. We name this architecture as *Adaptive Switch* and we have solved two technical challenges. The first one described in the next section is the architectural design of *Adaptive Switch*. And the second one is how to develop and map programs to the targets based on *Adaptive Switch*, which is elaborated later. To evaluate our proposal, we have implemented three use cases on a prototype of *Adaptive Switch*. Finally, we conclude the work.

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 facilitates the standard switching functions and the PL part deploys FPGA for programming the customized processing. The two parts, SS and PL in the proposed architecture can be implemented by a **two-chip** approach, in which SS can be a traditional switching ASIC with or without P4 compliance, and PL can be an FPGA chip or an MPSoC/ACAP chip integrating FPGA with other processors (*e.g.*, ARM cores) in a single system on chip. In the two-chip solution, the two physically separated chips are connected using a PCIe or Ethernet interface or transceiver. Alternatively, we can also build *Adaptive Switch* with a single-chip solution, connecting PL and SS via a high-bandwidth on-chip bus (*e.g.*, AXI).

As shown in the left side of Fig. 1, switching fabric is the core of SS, which usually adopts cross-bar for high speed switching between ingress and egress ports. Packets coming from a network interface traverse through an ingress/egress pipeline before/after the switch fabric. In an ingress or egress 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 (buffer management, packet scheduling, shaping, *etc.*).

All incoming packets enter SS first, and most of them are completely handled in SS. Only the ones relying on the functions unsupported by SS will be further sent to PL for cooperative processing. In the case that a packet triggers processing in PL, SS stores the packet in on-chip or off-chip memory and sends specialized metadata to PL. The metadata is customized to carry the extracted information from the packet that the processing in PL needs. The processing in PL updates the metadata and returns it to SS. SS combines the original packet with the returned metadata into a complete packet for forwarding, or simply drops the packet.

We introduce an additional Memory Management Unit (MMU), as shown in Fig. 1. MMU manages the memory to buffer packets waiting for updated metadata from PL. More specifically, it includes three main functions: a) dynamically allocation of a memory block for packet storage. b) writing packet data into memory. c) reading the stored packet data back from memory and assembling the output packet using the returned metadata from PL.

We design the *Adaptive Switch* architecture based on two observations or assumptions. First, most processing in PL is usually based on packet header only or/and the packet segment of the first a few bytes in payload. Our design exchanging only metadata, which can be flexibly defined, guarantees a limited interconnect bandwidth consumption between SS and PL. In a rear case, a processing, which looks into the entire packet, will fill the metadata with the whole packet. Second, not all traffic would require processing in PL; otherwise, the related functions would become a part of an off-the-shelf switching ASIC or can be directly moved to the SS part in the *Adaptive Switch* architecture.

Considering the average packet length to be around 600 bytes and assuming metadata size to be 64 Bytes, it allows 20% of the traffic to be further handled in PL if interfacing through PCIe Gen4 x16 without sacrificing port density, in the worst case of using the largest switching AISC with 12 Tbps as SS. Allowing 10% port density to connect SS and PL in the two-chip solution, 100% original traffic can have further processing stages in PL, again in the condition that the average packet length is 600 Bytes and metadata is 64 bytes.

User Specific Processing in PL

Development Flow. The basic design flow of FPGA-based PL includes several steps: a) Determine packet processing requirements and the dataflow model. b) Write processing specific functions and processing. Either high level languages,

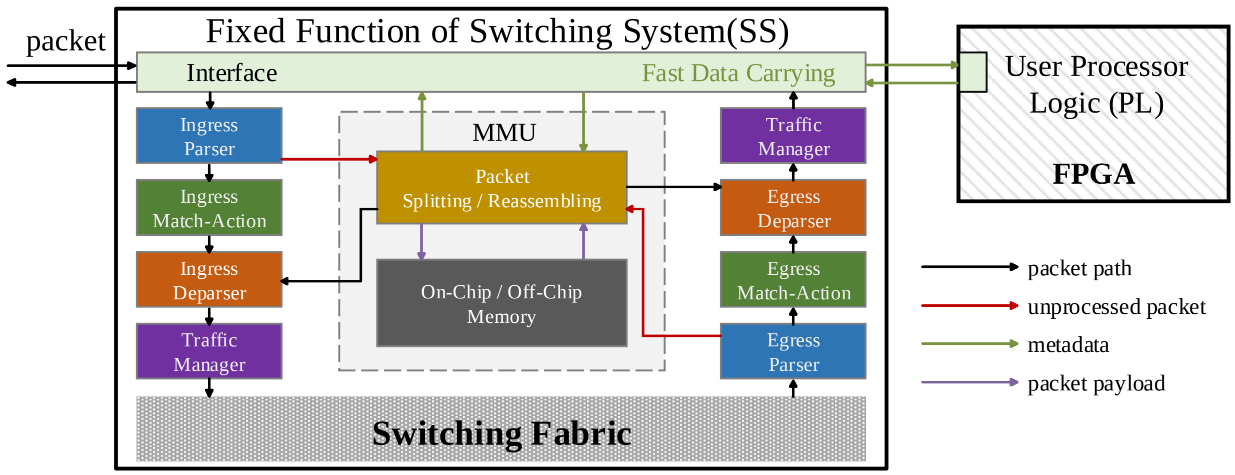

Fig. 1. Architectural Diagram of Adaptive Switch.

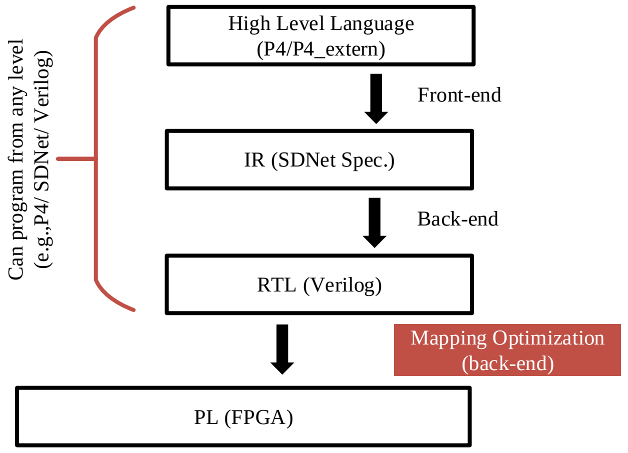
Can program from any level (e.g.,P4/ SDNet/ Verilog)
High Level Language (P4/P4_extern)
Front-end
IR (SDNet Spec.)
Back-end
RTL (Verilog)
Mapping Optimization (back-end)
PL (FPGA)

Fig. 2. Development Flow.

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], is also efficient to fast build processing logic. c) 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 to build our prototype and show a more general 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 built-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 based PL enables reconstructing micro-level parallel processing for user specific dataflow-based computing. We achieve efficient mapping to PL by introducing: a) a parallel processing architecture and b) compiling optimization to reduce resource footprints.

The parallel architecture for mapping user specific processing is depicted in Fig. 3. From the top view, there are $m(m \geq 1)$ processing pipelines. In each pipeline, there are $n(n \geq 1)$ stage. We frame the abstraction of processing stage as a “Basic Processing Unit (BPU)” colored as shaded areas in Fig. 3. Parallelism is also introduced to BPU by dispatching input traffic load to multiple “execution engines”, each of which is consist of an *Action* module operating on a set of *data* kept in memory. In a generic model, the number of inputs (dataflows) to a BPU is the same as the number of execution engines of its predecessor but it is not necessarily the same for number of inputs and output in a BPU. Assume K_i inputs and K_{i+1} outputs in stage i. The *Dispatch* module in stage i of a BPU is to distribute the K_i inputted data-flows into K_{i+1} execution engines, as evenly as possible. High level languages, SDNet Spec. and RTL languages can be used to define *Action* modules in BPUs and the parameters (m, n, K_i, *etc.*). *Dispatch* module enables flow affinity distribution for packets over paths (execution units) based on a hash distribution.

It is non-trivial to have an optimization phase to split and allocate the processing of related data in BPUs (*i.e.*, decide what *data* to keep in each execution engine). If we duplicate the complete data in each paralleled execution engine and/or processing pipelines, we simply achieve the largest throughput by distributing the packet in a simple round-robin strategy, however, PL cannot afford the memory cost of this solution, especially for data-intensive processing.

We introduce an optimization where the objective is to balance the workload handled in each processing pipeline and each execution engine, while maintaining dataflow affinity and reducing data duplication. We formulate the optimization problem by considering:

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

b) Data copies. Data-dependent action/processing requires accessing its duplicated data in execution engines or/and pipelines. Introducing more data duplicates mitigates data access conflicts at the cost of more (on-chip) memory. We allow small data redundancy among execution engines so as to make the dataflow distribution more balanced between execution engines.

c) Dataflow affinity. All the packets from the same flow should be processed by the same execution engine of a processing pipeline in order to maintain processing dependencies and avoid the out-of-order problem. The flow affinity can be kept by using a wildcard-based matching table (*e.g.*, longest prefix matching table, TCAM, *etc.*).

The optimization problem of data splitting can be reduced from the classic “Subset-Sum” problem, which is known to be NP-Hard. We invented a simulated annealing-based heuristic algorithm to explore the solution. To always have high processing performance, it is important to keep the load balanced by the *Dispatch* module. In practice, the traffic distribution used as the input of the heuristic algorithm changes over time, and so a remote controller is used to collect the workload variance among execution engines as feedback for

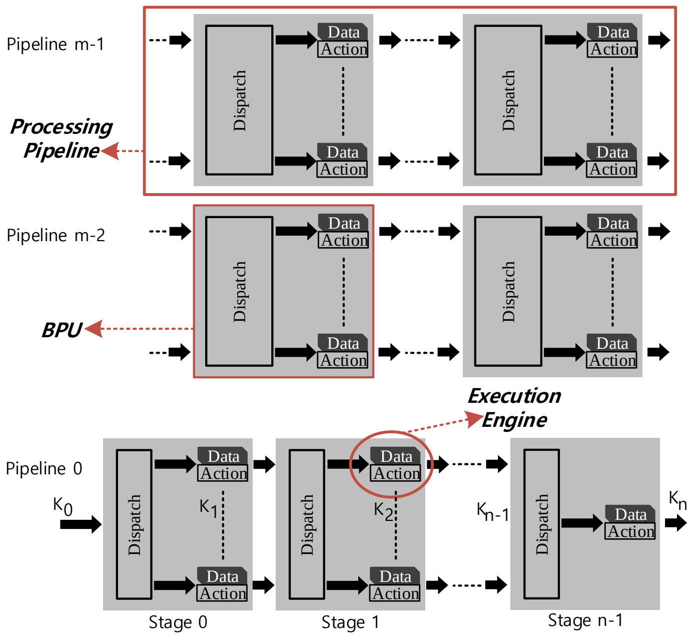

Fig. 3. A General Model of Parallel processing Pipeline in PL.

run-time recalculation and configuration. The calculation time of the algorithm is evaluated below.

EVALUATION

As a proof of concept, we implemented a prototype of *Adaptive Switch* on a Xilinx ZC706 development board, with 4000+ lines of code including the SS functions and three use cases in PL. A packet generator was integrated in the *Adaptive Switch* prototype to inject input traffic. We utilized five traces: two real traces collected from an ISP backbone (**WIND** trace) and an LTE base station (**LTE** trace), and three synthetic traces following **Exponential** distribution, **Pareto** distribution and **Uniform** distributions, respectively.

Use Cases

Three use cases were implemented to show architectural flexibility.

a) **Congestion Control.** NDP [10] ensures low forwarding delay of small-batch packets when detecting congestion in a switch. NDP is an example of event-driven processing, which is not well supported by exiting switches. To deploy NDP in *Adaptive Switch*, we allocate the logical output queue in the MMU of the SS part. Two NDP actions are implemented in PL: one monitors the queue depth as the congestion trigger signal, and the other

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 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.State.Tras. tables (8 states per flow))	908/0.43%	1937/2.80%	1725/0.43%	43/7.92%	0

sends explicit notifications to adjust packet size. The front-end compiler wraps the user logic into the *Action* module for generating a BPU and processing pipelines.

b) **Network Measurement.** DISCO [9] is an efficient flow statistics algorithm, but it is challenging to deploy DISCO in (P4 or non-P4) COTS switches due to the lack of exponential and logarithmic calculation support. In our DISCO implementation on *Adaptive Switch*, the input metadata to PL includes the flow ID and the length of each incoming packet, while the output is the flow statistics counter value kept in on-chip memory. We used Verilog HDL to implement P4_extern functions for DISCO.

c) **Stateful Firewall.** We have also facilitated a stateful firewall forwarding engine in the form of P4 extern functions supported by *Adaptive Switch*, which is a challenge with commodity switches. The implemented engine records the states of the connections for filtering packets. And there are two hardware flow tables implemented: one is used for fundamental Match-Action operations, and the other stores a list of states for each corresponding flow. When a packet from a flow arrives, it executes actions according to the current flow state and the packet fields of interest. It also updates the Match-Action table to indicate the next state.

Performance Results

In Table. I, we quantify the resource consumption of a simplified model of Fig. 3. The row illustrating the resource for one *Dispatch* module in BPU consists of the consumption for an SDNet-generated parser/matching table and a 4x4 crossbar architecture to distribute dataflow. For the three cases illustrated in the other rows, execution engines in each case contains an exact-match table (1K entries) as data storage. More memory is required by each case with different usage. In the congestion control case, there are eight priority queues for four output ports, each of which has queue size of 64KB. In addition, we deploy 4K counters for the measurement case, and a programmable state transition table for each flow in the firewall case. The resource consumption almost linearly increases with the increasing number of execution engines, stages, pipelines and entries used. Due to page limitations, we only demonstrate here the results for each use case implemented with four execution engines (one stage and single pipeline). From the results, we observe that a typical FPGA is sufficient to have the three user cases deployed.

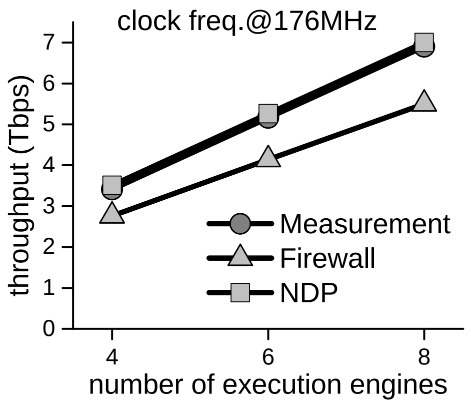

Fig. 4. Performance estimation of different number of execution engines. The maximum clock frequency is 176MHz, which is limited by the timing performance of the match tables.

In Fig. 4, we depict the throughput trend when increasing the number of execution engines. We calculate the throughput based on the maximum clock frequency, data bus width, and the average packet size (600 Bytes). The figure shows a curve that is very close to a linear function.

By looking into the cycles need by each use case, we established the PL processing delay as $0.136\mu s$, $0.142\mu s$, $0.130\mu s$, for network measurement, firewall, and congestion control, respectively.

We have also implemented the PL mapping optimization presented earlier for run-time configuration with 300+ lines of Python code. It was deployed on

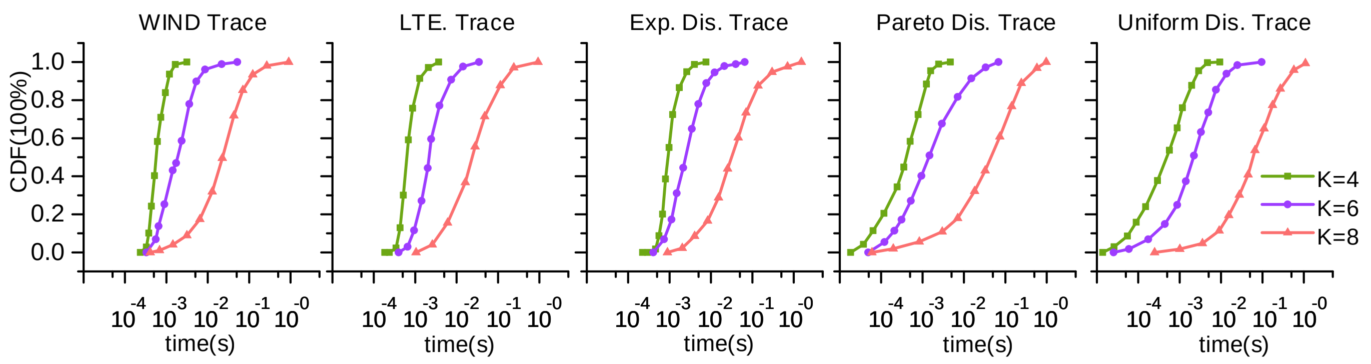

Fig. 5. The runtime performance of the heuristic algorithm for load-balanced entry allocation. The performance is indicated by the calculation time(s).

a Dell R620 server (with 8G RAM and 2.80GHz Quad Core Intel CPU running Ubuntu 16.04 LTS OS), using PyPy toolset to accelerate the Python program and benefiting from the JIT (Just-in-Time) compiler in PyPy. Fig. 5 depicts the Cumulative Distribution Function (CDF) of the calculation time under the five traffic patterns described earlier in this section. There are three curves in each subfigure representing the different settings for the number of execution engines (with one stage and one single processing pipeline), *i.e.*, 4, 6, 8 execution engines. It takes a longer time for run-time mapping optimization on more execution engines. The calculation time in our experiment settings was less than 0.5s in 90% of the testing, providing a 100% guarantee of less than 1.12s.

Conclusion

We have presented the *Adaptive Switch* architecture for network-centric computing. The key insight behind *Adaptive Switch* is leveraging the switch fabric to provide high throughput while offloading hardware programmable processing to FPGA. We have implemented a prototype and three use cases for it. The three user cases demonstrate that the flexibility of the proposed architecture is superior to fixed-function state-of-the-art switches. In addition, the evaluation results show that the implementation controls well the resource consumption in providing processing throughput at terabit-per-second rates. Although the processing throughput in PL part is still less than the throughput of the switching ASIC, we argue that not entire packets and not all dataflows need to be handled with user defined processing in PL, and as a result, *Adaptive Switch* architecture has overall throughput compatible with a COTS switch.

Acknowledgment

This research was supported in part by the National Natural Science Foundation of China under grants (U1736205).

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. (1-20-2020) https://barefootnetworks.com/products/brief-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.

BIOGRAPHIES

Siyi Qiao (qiaosiyi.900305@stu.xjtu.edu.cn) is pursuing a Ph.D. degree at the Xi'an Jiaotong University. His research interests include computer networks and field programmable technology.

Chengchen Hu [STM'03, M'09] (chengchen.hu@xilinx.com) is a Principal Engineer at Xilinx Inc. and is the founding director of Xilinx Labs Asia Pacific in Singapore. Prior to joining Xilinx, he was a Professor and the Department Head at the Department of Computer Science and Technology, Xi'an Jiaotong University in P. R. China. He is recipient of the New Century Excellent Talents in University award from Ministry of Education, China, a fellowship from the European Research Consortium for Informatics and Mathematics (ERCIM), a fellowship of Microsoft "Star-Track" Young Faculty. His research theme is to monitor, diagnose and manage networking and the distributed computing through hardware optimized and software defined systematical approaches.

Gordon Brebner (gbj@xilinx.com) is a Fellow at Xilinx, Inc., the technology leader in highly flexible and adaptive processing platforms. He works in Xilinx Labs, leading an international group researching issues surrounding networked and trusted processing systems of the future. His main personal research interests concern dynamically reconfigurable architectures, domain-specific languages with highly concurrent implementations, and high performance networking and telecommunications. He holds around 40 patents, and has published around 60 papers, in the general area of networking with FPGAs. Prior to joining Xilinx in 2002, he was the Professor of Computer Systems and Head of the Department of Computer Science at the University of Edinburgh, and remains an Honorary Professor of Informatics there.

Jianhua Zou [M'10] (jhzou@xjtu.edu.cn) received the Ph.D. degree from the Institute of Plasma Physics, Chinese Academy of Sciences, Beijing, China, in 1991. He conducted the Postdoctoral Research with the Huazhong University of Science and Technology, Wuhan, China, and Xi'an Jiaotong University in 1991 to 1993 and 1993 to 1995, respectively. He became a Professor with the System Engineering Institute, Xi'an Jiaotong University. He was the Principle Investigator of two projects funded by the National Science Foundation of China, about ten key projects supported by industry and military. As a Senior Visiting Scholar, he has established partnerships with Eindhoven University, Eindhoven, The Netherlands, and Philips Company, Amsterdam, The Netherlands. He has published more than 60 academic articles. His current research interests include intelligent control, computer vision and pattern recognition, data mining, and knowledge discovery.

Xiaohong Guan [M'93,SM'95,F'07] (xhguan@xjtu.edu.cn) received his B.S. and M.S. degrees in Control Engineering from Tsinghua University, Beijing, China, in 1982 and 1985, and his Ph.D. degree in Electrical and Systems Engineering from the University of Connecticut in 1993. From 1985 to 1988 and since 1995 he has been with the Systems Engineering Institute at Xian Jiaotong University, Xian, China, and Dean of School of Electronic and Information Engineering since 2008. From 2001 he has also been with the Center for Intelligent and Networked Systems, Tsinghua University. He is a member of Chinese Academy of Science and IEEE Fellow. He has been serving the Editor of IEEE Transactions on Smart Grid since 2014. His research interests include economics and security of networked systems, optimization based planning and scheduling of electrical power and energy systems, manufacturing systems, etc., and cyber-physical systems including smart grid, sensor networks, etc.