STRIDE: Single-Trip-time based Reliable Data Transport Protocol for the Reconfigurable Cloud

[1]Wenwen Fu, [1]Tao Li, [2]Ahmed M. Abdelmoniem, and [1]Zhigang Sun
[1]School of Computer, National University of Defense Technology. China
[2]Faculty of Computers and Information, Assiut University, Egypt
{fuwenwen, taoli, sunzhigang}@nudt.edu.cn, , ahmedcs@aun.edu.eg

***Abstract*— In a recent development, reconfigurable clouds become a viable solution to overcome practical problems in clouds, such as scalability, delay, etc., by offloading computation tasks to reconfigurable hardware, FPGA. Several existing techniques, such as TCP/IP Offload Engine (TOE) and Lightweight Transport Layer (LTL), are still hard to be implemented in real-world deployment due to large overhead or stringent dependency of the underlying network. In this paper, we propose STRIDE, a novel inter-FPGA data communication protocol, to provide reliable end-to-end communication which addresses practical problems in deployment. In our design, STRIDE leverages FPGA's abilities through programming, such as precise timestamping, to make more accurate measurement on end-to-end queuing delay and deliver more precise control in managing traffic in cloud. We implement STRIDE on a FPGA-based network experimental platform and demonstrate that STRIDE reduces various hardware resources consumption by 36% to 49% compared to TOE. Additionally, it also improves flow completion time in comparison to TOE by 2.2X. We further demonstrate STRIDE outperforms DCTCP and TCP-Vegas in OMNET simulator by up to 1.8X and 2.3X on average and 99th percentile respectively in large scale setting.**

I. INTRODUCTION

Semiconductor industries have been struggling to keep up with the rapidly growing demand for computing power in cloud to support applications like big data and deep learning [1,2], and have reached their limit in overcoming practical problems such as scalability, delay, etc. This is because modern cloud architecture relies heavily on servers' CPU cores to run different network stack for different policies, such as virtual network, load balancing, firewall, security, etc. This may result in taking away processing power from VMs. Many existing ASIC (e.g. DPDK), multicore SoC, and RDMA based NICs allow the network stack to be offloaded to hardware. However, if there is a required feature that cannot be handled in the hardware, then the stack must be reverted to the software. This also means these technologies are not flexible to keep up with the dynamic policy change in cloud.

Reconfigurable computing offers a great potential to resolve the aforementioned problems through utilizing reconfigurable, Field-Programmable Gate Arrays (FPGA) [3]. It is a programmable hardware that allows full offload host networking to hardware, providing more flexibility in adapting to new solutions. Additionally, FPGAs can provide infrastructure operators with the capabilities to achieve high throughput, more predictable latency, lower power consumption. With this FPGAs, servers in cloud platforms can harvest the above benefits to improve applications' performance for the clients, as well as high system utilization for the operator (e.g. bandwidth, CPU, etc.). Another advantage is FPGAs can be quickly inserted into an existing system to enhance system performance without altering the system ("bump-in-the-wire") [3,4,5]. Nevertheless, despite all the previous benefits of FPGA devices, there is little study on how to effectively exploit these benefits in implementation.

One way to exploit FPGA is to offload computation task of the transport layer protocol (e.g. TCP) to FPGA. *TCP/IP Offload Engine* (TOE) [6,7], a pioneering work on offloading protocol to FPGA, allows applications to offload data without copying and performing interrupt handling in the kernel, reducing processing delay in TCP/IP stack [7]. Additionally, TOE is compatible to legacy TCP/IP and does not require switch support. However, due to implementation complexity and resource consumption, TOE does not fully offload TCP into FPGA, i.e. some parts of the processing are still done in the kernel [8]. Later in this paper, we provide a discussion on the implementation challenges in fully offloading TCP processing (and any existing transport layer protocols) to FPGA.

Another work of the same theme, *Lightweight Transport Layer* (LTL) [3] was proposed to provide low latency connection while avoiding packet drop in inter-FPGA end-to-end connection settings. To address reliability, the authors propose a low-layer level congestion control scheme, similar to DC-QCN [9]. LTL utilizes Priority Flow-Control (PFC) [10] to slow down incoming traffic to a switch when congestion is detected. LTL also employs ECN to signal end-hosts about a congestion. However, the authors do not provide detailed descriptions on how their congestion control scheme is implemented and offloaded into the FPGA device.

We find that there is design incompatibility between the Congestion Control (CC) schemes (such as DCQCN and TIMELY) and the corresponding hardware design. This incompatibility not only limits the full utilization of FPGA, it may also result in severe system performance degradation caused by the following phenomena occurring at the sender's side: (i) Multiplication and division operation required for computing congestion window (*cwnd*) takes up the majority of the processing resources in FPGA. (ii) The mechanism used to keep track of packet state (e.g. which packets has been ACKed and transmitted, or not, or retransmitted, and so on.)

occupies a large portion of on-board memory of the FPGA device. Moreover, memory allocated for multiplication and division operations further increases the memory cost. The details on these observations are discussed in Section II.

Therefore, the question is: *How to design a CC scheme that is compatible with hardware design such as FPGA*? To address this question, we first must consider a clean-slate solution that can be fully offloaded to FPGA. This is because in order for a solution to be deployment ready, first, the solution must be compatible with legacy commodity switches that are commonly used in cloud (or datacenter network). Second, datacenter network is typically made up of tens of thousands of switches and servers [11]. For these reasons, from network management and maintenance perspective, the solution must require minimal network reconfiguration. Driven by these two practical motivations, we consider delay-based scheme. Moreover, the scheme must be able to achieve low CPU, memory, and power consumption. At last, the design must be able to achieve basic CC mechanisms, such as end-to-end reliability, congestion window adaptation, and so on.

In this paper, we present a novel transport protocol, STRIDE (Single-TRIp-time-based reliable Data transport protocol for the rEconfigurable cloud) that is specifically designed for reconfigurable cloud. STRIDE is a lightweight reliable data transport protocol for inter-FPGA communication that imposes lower overhead on resource utilization and bandwidth. This is achieved by replacing the computationally heavy multiplication and division operation with a series of the simpler one-clock-cycle bit-shifting operation to compute and adjust transmission rate. This allows the entire computation to be done in FPGA, resulting in quicker responses to congestion.

STRIDE also takes advantage of the functionalities available in FPGA, such as packet timestamping in hardware and the high FPGA clock precision, which enables STRIDE to accurately measure end-to-end delay or *single trip time* (STT) delay between two end-hosts. Thus, utilizing STT as a congestion signal allows STRIDE to have better control on the transmission rate, which results in lower packet drops. Importantly, this makes STRIDE a plug-and-play solution without requiring additional reconfiguration in network. Finally, STRIDE also provides basic functionalities for end-to-end reliability, such as packet ACKing, retransmission, three way handshaking, and connection termination.

To evaluate STRIDE's performance, we build packet processing modules and implement an actual prototype using FPGA Accelerated Switches platform [15,16], which we then deploy in our testbed. In our testbed experiments, we compare STRIDE to the state of the art (TOE) and demonstrate that STRIDE improves flow completion time (FCT), or time required to complete a flow, by at least 2.2X. The detailed description of system implementation and experimental results are provided in the evaluation section. Additionally, STRIDE also reduces computational and memory cost by up to 49% and 38% respectively compared to TOE. In large scale scenario, we demonstrate STRIDE outperforms DCTCP and TCP-Vegas through OMNET simulator [28] by up to 1.8X and 2.3X on average and 99th percentile respectively.

II. Related Work

One of the early developments on TCP offloading technology for high-speed network is TCP offload engine (TOE). It allows hardware to offload the entire TCP/IP stack network controller, reducing the processing overhead in the CPU and server I/O. Such overheads include connection establishment and termination, TCP checksum, and sliding window for congestion control and reliability [6], which speeds up the processing speed. However, at implementation level, CPU is still required for TCP connection management function to process received and sent packets. For example, every socket created in application's user space corresponds to a socket structure in kernel space. Thus, every system call eventually has to invoke a function in the kernel. For these reasons, offloading TCP protocol from Linux system core function to a dedicated processing unit (hardware) can be very challenging and complex. This is because TCP connection management depends on functionalities residing in kernel space. There are a number of recent FPGA projects that focus on physical and data link layer [17], but they do not consider transport layer functions such as congestion control scheme due to implementation challenges described above.

LTL provides congestion management to avoid early packet drops [3] which incorporates ECN [18,24] based DC-QCN scheme [18] and Priority Flow Control (PFC) [10] in their implementation of inter-FPGA network protocol. However, ECN and PFC based schemes require switch support. Additionally, even with switches that support ECN and PFC, the scheme requires network operator to configure the switches in datacenter, which may not be efficient as a typical datacenter is made up of thousands of switches. Additionally, the utilization of PFC potentially may lead to deadlock caused by PDF pause frame [26] stalling the network from delivering data. Similarly, NDP [30] is another reconfigurable cloud solution that requires switch support (P4 Switches).

III. Background

Despite benefits that FPGA offers, we discover challenges of implementing a complete offloading congestion control scheme to FPGA. In our early designs, we first attempted to implement TCP and TIMELY in FPGA, and then performed simple experiments in our testbed with two servers connected through two serial switches. Through our endeavors, we learn that it is not only difficult to fully offload a protocol to FPGA, our experiments also show that the system performs poorly. The lessons we learned from implementing a fully offloaded FPGA presented in this section provides the insights to the design of our CC scheme for reliable end-to-end inter-FPGA data transfer.

Lesson 1: ***Computational cost***. Offloading the entire TCP stack to FPGA is computationally expensive, caused by the complex mechanism in calculating congestion window size in hardware. Generally, the high computation cost comes from multiplication and division operations. In multiplication operation, for every single partial product generated by multiplying two bits, it requires multiple rounds of bits shifting and addition operation. Moreover, since division involves several multiplication operations, the computation cost becomes even more expensive. This is because that

computation involves multiplying several factors by the divisor to obtain approximately 1, followed by multiplying the outcome from previous multiplication operations by the dividend. One of the main calculations in any CC scheme is the updates of the transmission rate which relies mainly on multiplications and divisions. Hence, it becomes computationally expensive to update the rate, especially for a scheme that requires many multiplication and division operations like TIMELY does. That cost increases proportionally with the number of flows sharing the NIC.

Lesson 2: ***Memory cost***. Typically, a CC scheme needs to keep track of some per-packet state such as SEQ/ACK numbers, inflight packets, packets that are ready to be transmitted, packets that are not ready for transmission, .., etc.; these information needs to be stored in the on-board memory inside the FPGA card. Moreover, the scheme also needs to allocate additional memory space to keep track of the partial product from multiplication and division operations. The amount of memory used may result in exhausting the memory.

Therefore, based on the above lessons, we ought to rethink how congestion control scheme should be re-designed to achieve the objective of complete offloading on the FPGA.

IV. STRIDE Overall Design

In this section, we provide a description of STRIDE framework design, discuss STT, and our CC scheme.

A. *STRIDE Framework*

STRIDE is a lightweight transport protocol that provides full-duplex and reliable delivery. We illustrate an overview of STRIDE's workflow Figure 1.

Packet format. STRIDE builds on top of UDP which provides the basic transport layer multiplexing function. STRIDE packet header (Figure 2) is encapsulated in UDP packet, allowing STRIDE to be more compatible with legacy system. Additionally, since UDP is a lightweight protocol, it keeps the packet processing cost low. Moreover, utilizing UDP also reduces the communication complexity caused by middle-box protocols [26] (e.g. NAT, Firewall, etc.).

Establishing and terminating a connection. To establish a connection between two hosts, STRIDE performs a three-way handshaking for connection establishment and connection tear-down functions similar to the ones of TCP.

Congestion Control. In our design, we first address the challenges described in section III. Then, we consider more practical and deployment aspects of the scheme, such as system configuration and scalability issues. We also consider the limitations of RTT delay measurements [13, 14]. To address the former, we design STRIDE as a pluggable system which requires minimal switch features. This differentiates STRIDE from ECN-based solutions (e.g. DCTCP [19], DCQCN [9], and ECN-RED [12].) which require configuring switches and may not be scalable in a large-scale datacenter environments. To address the latter limitation, STRIDE leverages hardware technology in FPGA such as timestamping to measure accurately the one-way end-to-end delay. This allows STRIDE to perform a more fine-grained control loop for datacenter congestion control, where a flow in datacenter only lasts for microseconds [13,19]. For this reason, STRIDE utilizes STT to detect congestion instead of RTT.

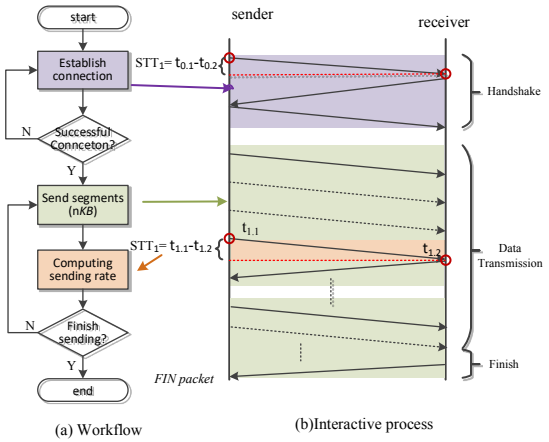

Fig. 1: Overview of STRIDE

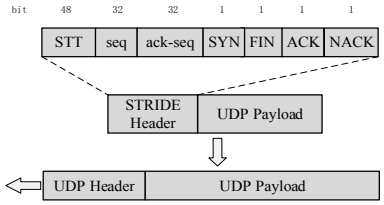

Fig. 2: STRIDE packet structure

B. *Single Trip Time (STT) Measurements*

Clock drift problem in NICs makes measuring STT very challenging. One possible solution is to synchronize on-board clock using Precision Time Protocol (PTP) [20]. This scheme can synchronize system clock down to sub-microseconds granularity. However, PTP is not scalable in a large-scale environments. As reported in [20], PTP requires all end-hosts and switches to be in the same multicast group. However, for scalability purposes, today's networks are divided into multiple smaller clusters for more effective and efficient large-scale network management. Additionally, PTP relies on centralized controller to keep track of the "right" clock. Nevertheless, the controller can become overloaded considering the scale and size of modern datacenter networks.

In contrast, FPGAs provides much more precise clock cycles and have lower time dilation among multiple cards [21], which makes this technology a more suitable solution for large-scale networks. To verify its clock precision, we perform testbed experiments with two servers connected by two serial 10GB switches. We generate a single flow traffic using iperf. Then, we measure the time difference Δt between two observed STTs at time t_i and t_{i+1} at both the sender and receiver, where, $\Delta t = \mathrm{STT}(t+1) - \mathrm{STT}(t)$. Next, we observe in multiple experiments running over 5 minutes interval that consistently the time difference is $\Delta t < 1\ nanoseconds$ for every 1 millisecond interval. The error Δt is sufficiently small for an accurate STT approximation in datacenter, where a flow typically completes in microseconds [13]. The precision comes from FPGA's crystal error that is within 0.0025%, which results in less than 1 nanosecond error margin. In other words, FPGA allows STRIDE to obtain STT at 10 nanoseconds accuracy. Moreover, this precision also allows multiple FPGA cards to be programmed to achieve the same cycle, so they can operate at the same frequency [22] resulting in a significantly more precise clock synchronization compared to NIC.

To measure STT, STRIDE uses the timestamping function available in FPGA by programming. We have $STT(t) = t_i^{RX} - t_i^{TX}$, for $t_i^{RX} > t_i^{TX} > 0$, where t_{TX}^i and t_{RX}^i denote timestamp at the sender (TX) and receiver (RX) on packet i. Here, $STT(t)$ can be interpreted as an estimation delay between TX and RX at time t. Additionally, $STT(t)$ is computed at RX upon receiving packet from TX, which carries t_i^{TX} in its STRIDE header.

C. Congestion Control

We describe our congestion control mechanism for inter-FPGA without in-network support. STRIDE is a rate-based scheme, whose congestion avoidance follows gradient-based algorithm [13]. In other words, rate is adjusted according to the derivative of the queueing delay with respect to time. This is realized by accurately measuring the STT on a per-packet basis. Hence, STRIDE can react to queue buildup without waiting for a standing queue to form, eventually allowing STRIDE to achieve low latency.

Algorithm 1 provides the description of our congestion control scheme. Generally, STRIDE maintains rate $r(t)$ per connection at time t and updates its rate every RTT by calculating delay gradient using two consecutive STT samples collected at the receiver. In our design, we employ two thresholds T_{low} and T_{high} to detect and respond when a link is underutilized or experiencing overly high latency respectively. T_{low} and T_{high} are related to min-STT, which is the transmission time without congestion. However, when $T_{low} < r(t) < T_{high}$, rate $r(t+1)$ is decided by normalized gradient using the concept proposed in [15]. If the normalized gradient ≤ 0, then it means there is room for a higher rate and $r(t+1)$ should be increased (line 20). Otherwise, $r(t+1)$ is decreased (line 22).

Algorithm 1: *STRIDE Congestion Control*

1. $STT_{new} = STT_{old} = \Delta STT = 0$ // Initialization
2. $STT_{min} = min\{S\}$ // S is a set of observed STTs
3. **Procedure** STRIDE (STT_{new})
4. $\Delta STT = (1 - \alpha) . \Delta STT + \alpha (STT_{new} - STT_{old})$
5. $STT_{old} = STT_{new}$
6. **if** $STT_{new} < T_{low}$
7. rate $r = r + \beta$ // increment with step size $\beta > 0$
8. *return*
9. **if** $STT_{new} \geq T_{high}$
10. Solve eq. (1) with $k = 3$
11. *return*
12. **if** $T_{high} > new_{stt} > T_{low}$
13. $\Delta = \Delta STT$
14. *normalized_gradient* = 0
15. **while** $\Delta > STT_{min}$ // solving eq. (2)
16. $normalized\_gradient$++
17. $\Delta = \Delta - STT_{min}$
18. **end while**
19. **if** $normalized_{gradient} \leq 0$
20. $r = r + \omega . \beta$ // increase rate with weighted β
21. **if** $normalized\_gradient > 0$
22. $r = r . (1 - normalized\_gradient)$
23. *return*
24. **end procedure**

D. Analysis

Low computational cost scheme. To achieve a lower computational cost, STRIDE uses bits shifting to calculate its transmission rate. To compute line 4, $\alpha \in \left\{\frac{1}{2^i}\right\}$ which allows the computation of $r(t+1)$ to be done by utilizing bit shifting together with addition and subtraction operations. To update transmission rate (line 10), $r(t+1)$ is adjusted with the following equation:

$$r(t+1) = r(t) \sum_{i=1}^{k} \frac{1}{2^i} = r(t) \left(1 - \frac{1}{2^k}\right), \quad (1)$$

where k is the number of shifts required. In our experiment, we have $k = 3$. In practice, k can be relatively small because it has diminishing returns property as $k \to \infty$. This means the impact of k diminishes as k grows larger. Similarly, to compute normalized gradient in eq. (2), we use subtraction operations and count the number of loops until the reminder is less than $STT_{min} = min\{S\}$, where S is a set of observed STTs (line 13 to18).

$$normalized\_gradient = \frac{\Delta STT}{STT_{min}}. \quad (2)$$

track of packet state (e.g. packet that has been ACKed, ready for transmission, needs to be transmitted, .., etc.), STRIDE allocates 64KB per packet in DDR SDRM at the sender. Tracking packets at different states is accomplished by maintaining a counter of the number of packets at different stages. This approach requires relatively smaller memory space compared to storing the entire packets. Memory space allocated to packets is released upon receiving its ACK. ACK generation is performed by the receiver and follows the same procedure of TCP acknowledgement scheme.

STRIDE Reliability. When the receiver detects a missing packet, it notifies the sender with Negative ACK along with the sequence number of the missing packet. The sender then resends the missing packet and the subsequent packets upon receiving NACK. This idea is inspired by the classic GO-back-N protocol [23,25]. However, the advantage of a single NACK packet over TCP's duplicate packets (typically 3 duplicate packets) is that a single NACK packet reduces computational cost and memory space for processing packets in FPGA. Additionally, single notification also speeds up retransmission process such that the sender can immediately resend the missing packet as soon as NACK is received, instead of waiting for multiple notifications like in TCP. Moreover, STRIDE also keeps a timer to detect packet drop. When a packet is dropped, STRIDE resends the packet that has not been ACKed.

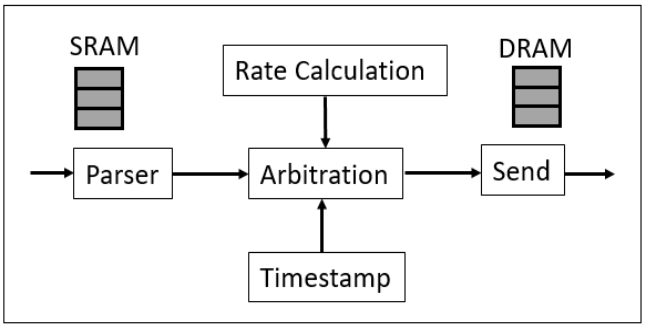

Fig. 3: STRIDE packet processing module.

V. IMPLEMENTATION

To evaluate STRIDE, we implement our solution in an open source FPGA Accelerated SwiTches (FAST) platform [15, 16], which provides multiple modular interfaces for FPGA developer. The FPGA board (iRouter board depicted in Figure 4) is equipped with Arria 5AGTMC3D3F31I5 and a flash memory to store file configuration of FPGA. The platform utilizes multi-core CPUs to allow software to communicate with FPGA through the PCIe bus. The line-card board provides eight 1-GigE Ethernet ports and two 10-GigE Ethernet ports.

Hardware module design. Here, we present the hardware design and implementation of STRIDE modules in FPGA. Figure 3 illustrates the steps required to process the packets. When a packet arrives at the FPGA (at the sender or receiver), the packet is placed in SRAM to be processed by a Parser module, which parses different headers (e.g. STRIDE, UDP, IP, and MAC header) in the packet. After parsing, Parser module passes the information in STRIDE header to Arbitration (AR) module to determine packet type (e.g. ACK, NACK, FIN, or SYN packet) which processes the packet according to its type. For instance, when the AR module identifies NACK bit is set to 1, STRIDE immediately retransmits the missing packet according to the sequence number stored in STRIDE header and the subsequent packets in DRAM.

AR module also counts the number of bits of an arriving packet and calls the timestamping module to timestamp the packet when the last segment of this packet arrived. After that, AR module creates STRIDE header and insert timestamping value into the header. Moreover, AR module fetches transmission rate from Rate Calculation (RC) module and relays it to the Send module. RC module employs algorithm 1 to determine its transmission rate. At the Send module, outgoing packets are stored in the DRAM and served in FIFO order onto the outgoing link.

In timestamping module, the timestamp value is inserted into the packet by assigning the timestamps bits in STT field in its STRIDE header. FPGA performs the same timestamping mechanism at both the sender and receiver. In the sender, FPGA processes packet coming from applications, whereas in the receiver, FPGA processes packets coming from network.

VI. EVALUATION

In this section, we provide evaluation of STRIDE through experiments in our testbed with 10GB port switches and large scale simulation involving 250 nodes.

A. Tested Experiment

Experiment setups. As depicted in Figure 4, Our network topology consists of two servers, two iRouters, two switches, and IXIA emulator to create background traffic. The two iRouters are used as the sender and receiver; and the two servers are used for initiating transport and gathering information from iRouters for analysis. We consider fixed rate, random rate, and linear increased rate to emulate background traffic in datacenter network using Spirent n12 emulators. Inspired by [13], STRIDE ACKs every 64 kb interval if there is no packet dropped, instead of ACK per packet (1500 byte). Through our testbed experiments, we observe that 64 kb provides STRIDE with the quickest time to adjust its transmission rate based on observed STT (Figure 5). Here, we refer 64 kB interval as data segment size in this paper. By doing so, STRIDE also cuts down the communication overhead for ACK.

Testbed experiments results. In our evaluation, STRIDE is compared to TOE, which is the state of the art FPGA based solution. Our experiments demonstrated that compared to TOE, as shown in Figure 6, STRIDE achieves significantly faster FCT across flow sizes (e.g. <10kb, 10kb-128kb, 128kb-256kb, 256kb-2048kb, 2048kb-10240kb), which translates to 2.2x improvement on average. Our improvement factor is the ratio of compared FCTs over STRIDE's FCT. These experiments demonstrate the benefits of fully offload CC scheme into FPGA, such that the processing speed is faster.

We conduct further investigation to understand how and why STRIDE obtains better performance. We first observe that STRIDE accomplishes lower number of packets dropped compared to TOE across different flow size, as depicted in Figure 7. By utilizing gradient based scheme, STRIDE up to3x less in the number of packets dropped compared to TOE, resulting in significantly lower packet retransmission caused by packet dropped. This way, STRIDE achieves lower average FCT.

Since STRIDE utilizes STT to determine its transmission rate, we investigate whether FPGA is also facing clock drift problem. To evaluate clock drift, we compare all STTs from a single flow to the observed first STT from the same flow. then, we measure how much the clock deviates since the connection is established. To ensure the measurement is not distorted by queuing delay in switches, we control the flow's transmission rate such that buffer in switches is empty. The performance matrix used in our evaluation is the ratio of each different STT over the first observed STT in *nanoseconds* (ns) scale. When the ratio is 1, both STT at time t and first observed STT is equal, such that $\mathrm{STT}(t) = \mathrm{STT}(1)$ for $t > 1$, which means FPGA does not experience clock drift problem. Otherwise, the clock deviates. As illustrated in figure 8.a, FPGA attains significantly small clock deviation since the establishment of the connection, where the ratio difference is at most 0.009. In other words, FPGA maintains very close clock approximation to the actual clock.

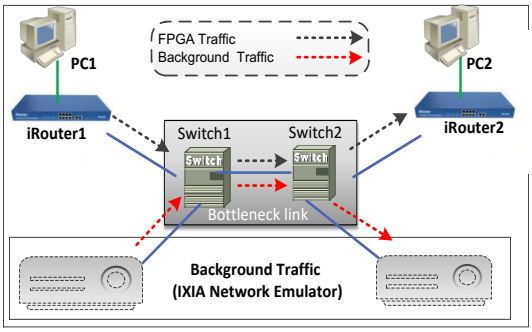

Fig. 4: Experiment topology

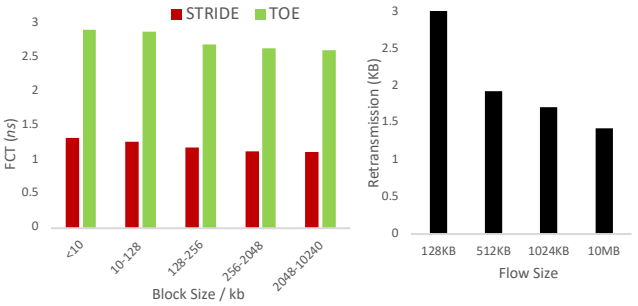

Fig. 6: Avg. time (per KiloByte) varying with block size

Fig. 7. Ratio Packet lost (TOE/STRIDE)

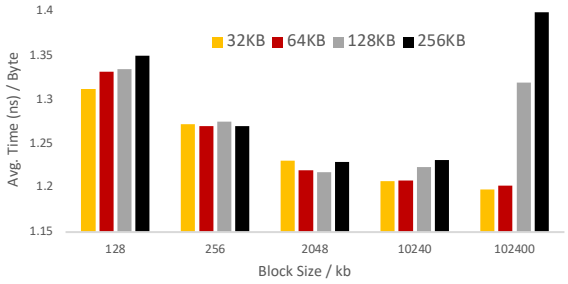

Fig. 5. Avg. time (per Byte) varying with segment size

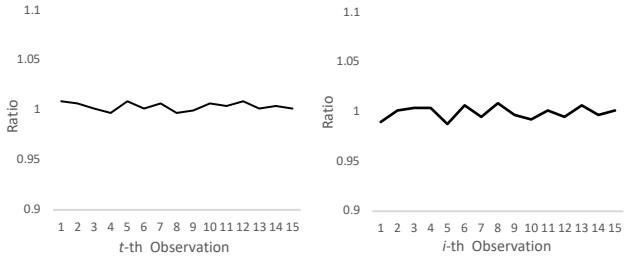

Fig. 8.a. FPGA Clock Drift

Fig. 8.b. FPGA Clock Drift

To further validate the FPGA clock precision, we compare $\mathrm{STT}(t) = \mathrm{STT}(t+1)$ for $t > 0$ and show that the ratio of $\mathrm{STT}(t)$ over $\mathrm{STT}(t+1)$ is very close to 1 (Figure 8.b). This also demonstrates that clock deviation between subsequent STTs is also significantly small. For these reasons, by keeping clock deviation minimal, FPGA provides much more precise STT approximation, allowing STRIDE to be more sensitive to queue buildup, resulting in quicker reaction to congestion without waiting for packet dropped. Hence, STRIDE achieves a lower average FCT compared to TOE.

Next, we investigate how much resources are required in FPGA to support STRIDE. Here, we measure the number of sets of arithmetic and logical operations in ALU, the amount of space required in memory (SRAM and DRAM), and the amount of block memory generator (BRAM) used in FPGA. In this experiment, we generate Iperf traffic in our testbed. As shown in table 1, STRIDE has better resource usage compared to TOE in all categories. This is because STRIDE achieves a lower processing overhead in FPGA by employing bits shifting technique and limiting the computation to just addition and subtraction operations (section IV). Thus, in achieving lower processing overhead, STRIDE has less information to store in the memory which results in lower space overhead. Moreover, by attaining a lower overhead in ALU and memory, STRIDE requires lower registration overhead in FPGA. Therefore, STRIDE achieves lower FCT.

With a lower overhead, STRIDE requires lower power consumption, leading to better energy saving, especially in large scale datacenter setting with hundred thousands of servers.

	ALUs	*Mem. Alloc.*	*BRAM*
TOE	12468	2.294MB	392
STRIDE	6339	1.4264MB	191
Improvement	49.15%	37.82%	36.75%

Table 1. Resource usage on Arria V.

Category	I	II	III	IV
Flow Size	<10b	10b-100kb	100b-1MB	>1MB

Table 2. Flow size categories.

B. Large Scale Simulation Experiment

Simulation setting: We employ OMNet++ simulator [28] to evaluate STRIDE at packet level and in larger scale scenario (256 hosts in FatTree based topology [27]). Here, STRIDE is compared to delay (TCP-Vegas [12]) and ECN (DCTCP [19]) based congestion control schemes with different traffic load described in table 2. The primary performance metric for comparison is the average FCT, and our performance factor is described as follows.

$$improvement = \frac{Compared\ FCTs}{STRIDE's\ FCTs}.$$

If the improvement is greater (smaller) than one, STRIDE is faster (slower).

Simulation results. Figure 9.a demonstrates that, on average, STRIDE achieves faster FCT across categories by up to 1.8x and 1.4x compared to TCP-Vegas and DCTCP respectively. Additionally, the outcome from average performance is also reflected in the 99th percentile, as illustrated in Figure 9.b.

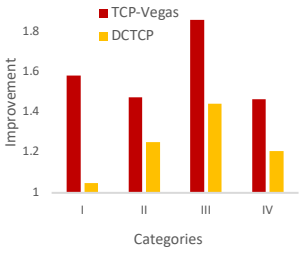

Fig.9.a Avg. FCT

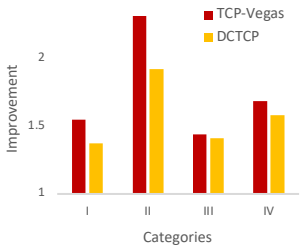

Fig.9.b 99th Percentile

The key observation on STRIDE's improvement over TCP-Vegas is that TCP-Vegas uses RTT to measure congestion. However, RTT is known for its imprecision due to its distorted reverse delay. For this reason, without precise measurement queue build up in switches, TCP-Vegas may react slower and less accurate in responding to congestion. In contrast, with more precise measurement, STRIDE is more sensitive to queue build up and can react quicker to congestion leading to lower packet drop, as demonstrated in our testbed experiments.

For DCTCP, it relies on fraction of packets marked by ECN to detect the congestion level. However, DCTCP requires one RTT interval to observe fraction of marked packet [19]. This also means DCTCP requires one RTT to adjust its transmission rate to achieve the desired rate. In comparison, STRIDE is able to adjust transmission rate to the desired rate as soon as it receives ACK packet. In other words, DCTCP reacts slower to congestion compared to STRIDE, which also results in slower FCTs for elephant flows in DCTCP. Moreover, as illustrated in Figure 9, smaller flows benefit from both DCTCP and STRIDE. This is because both schemes share similar objectives of keeping queue in switch buffer low, resulting in faster FCT for small (mice) flows.

VII. Conclusion

Although FPGA looks as a promising solution to improve system performance, full adoption of this technology is limited by high computational cost and space requirement to fully offload CC into FPGA. To resolve this problem, we propose a CC that utilizes lower computation resource and memory space. We also show the benefits and strengths of STRIDE through testbed and simulation experiments.

Acknowledgement. This research was supported in part by a research grant from Chinese National Prog. for Key Tech. of SDN, Supporting Resource Elasticity Scheduling, and Equipment Dev. (863 Programs) (No.2015AA016103), and National Natural Sci. Foundation: Synergy Research on CPU / FPGA Heterogeneous Network Processing System for Complex Network Applications (No.61702538). We thank Dr. Weichao Li for the discussion.

Reference

[1] M. Malik and H. Homayoun, "Big data on low power cores are low power embedded processors a good fit for the big data workloads," International Conference on Computer Design, 2015.

[2] M. Malik, et. al., "System and architecture level characterization of big data applications on big and little core server architectures," IEEE Int. Conference on Big Data. IEEE BigData, 2015.

[3] A. Caulfield, et. al., "A Cloud-Scale Acceleration Architecture". Annual IEEE/ACM Int. symp. on Microarchitecture (MICRO), 2016.

[4] Jian Ouyang, et. al.. "SDA: Software-Defined Accelerator for Large Scale DNN Systems. IEEE Hot Chips 26 Symposium (HCS), 2014.

[5] D. Sidler, et al., "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware," in FCCM, 2015.

[6] "TCP Offload Enginer", https://www.chelsio.com/nic/tcp-offload-engine/

[7] D. Sidler, et al., "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware," in FCCM, 15.

[8] D. Sidler, et. al.. Low-Latency TCP/IP Stack for Data Center Applications. Int. Conf. on Field Prog.Logic and App., 2016.

[9] Y. Zhu, et. al.. Congestion Control for Large-Scale RDMA Deployments. In SIGCOMM, 2015.

[10] "Priority Flow Control", https://1.ieee802.org/dcb/802-1qbb/

[11] A. Roy, et al, "Inside the Social Network's (Datacenter) Network," in ACM SIGCOMM 2015.

[12] J.Kurosa and k. Ross, "Computer Networking: A Top-Down Approach", Pearson Education, 2017

[13] R. Mittal, et. al.. TIMELY: RTT-based Congestion Control for the Datacenter. In SIGCOMM, 2015.

[14] C. Lee, et. al.. "DX: Latency-Based Congestion Control for Datacenters". In ACM Trans. On Networking, 2016.

[15] FAST project. https://github.com/FAST-Switch/fast.

[16] FAST Official website: http://www.fastswitch.org/

[17] S. Jun, et. al.. "A Transport-Layer Network for Distributed FPGA Platforms". Int. Conf. on Field Prog.Logic and App., 2015.

[18] Y. Zhu et. al.. "ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY". CoNEXT, 2016.

[19] M. Alizadeh, et. al.,"Data center TCP (DCTCP)," in SIGCOMM 2010.

[20] P. Estrela, et. al., Challenges Deplyoing PTPv2 in a Global Financial Company, in ISPCS, 2012.

[21] J. Casper and K. Olukotun, "Hardware acceleration of database operations," ACM/SIGDA Int. Symp. on FPGA, 2014.

[22] A. Rodionov and J. Rose, "Synchronization Constraints for Interconnect Synthesis", ACM/SIGDA Int. Symp. on FPGA, 2017.

[23] C. Guo, et. al.. "RDMA over Commodity Ethernet at Scale". In SIGCOMM, 2016.

[24] W. Bai, et. al.. Enabling ECN in Multi-Service Multi-Queue Data Centers. In NSDI, 2016.

[25] H. Saleh, et. al.. "Packet communication within a Go-Back-N ARQ system using Simulink" Crtl Eng. & Information Tech. , 2016.

[26] A. LangLey, et. al., "The QUIC Transport Protocol: Design and Internet-Scale Deplyoment", ACM SIGCOMM, 2017.

[27] M. Al-Fares. Et. al., "A Scalable, Commodity Data Center Network Architecture", ACM SIGCOMM, 2008.

[28] "OMNET++", https://www.omnetpp.org/

[29] D. Firestone, et. el. "Azure Accelrated Networking: SmartNICs in the Public Cloud", USENIX NSDI, 2018.

[30] M. Handley, et. al,"Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance", ACM SIGCOMM, 2017.