electronics

MDPI

Article

Design and Implementation of CPU & FPGA Co-Design Tester for SDN Switches

Yue Jiang, Hongyi Chen *, Xiangrui Yang, Zhigang Sun and Wei Quan

Computer College, National University of Defense Technology, Changsha 410073, China
* Correspondence: chenhongyi@nudt.edu.cn

Received: 4 August 2019; Accepted: 25 August 2019; Published: 28 August 2019

Abstract: The southbound protocol of Software Defined Networking (SDN) enables the direct access into SDN switches which accelerates the innovation and deployment of network functions in the data plane. Correspondingly, SDN switches that support the new southbound protocol and provide high performance are developed continuously. Therefore, there is an increasing need for testing tools to test such equipment in terms of protocol correctness and performance. However, existing tools have deficiencies in flexibility for verifying the novel southbound protocol, time synchronization between the two planes, and supporting more testing functions with less resource consumption. In this paper, we present the concept of CPU & FPGA co-design Tester (CFT) for SDN switches, which provides flexible APIs for test cases of the control plane and high performance for testing functions in the data plane. We put forward an efficient scheduling algorithm to integrate the control plane and the data plane into a single pipeline which fundamentally solves the time asynchronization between these two planes. Due to the reconfigurable feature of our proposed pipeline, it becomes possible to perform different testing functions in one pipeline. Through a prototype implementation and evaluation, we reveal that the proposed CFT can verify the protocol correctness of SDN switches on the control plane while providing no-worse performance for tests on the data plane compared with commercial testers.

Keywords: SDN switch; testing tool; CPU & FPGA co-design; reconfigurable pipeline

1. Introduction

Software-Defined Networking (SDN), decoupling the control plane from the data plane, enables the invocation and evolution in the development of the network. It makes the network open by allowing operators to deploy novel protocols (OpenFlow [1], Netconf [2], etc.) and network services (Network Management [3], Traffic Engineering [4,5], etc.). Through southbound protocols and provided programming interfaces between the control plane and the data plane, it is possible to determine how to process the packets on SDN switches. Operators can manage the network through this method flexibly. As a result, both industry [6,7] and academia [8–10] are incredibly interested in the design and implementation of the prototypes of SDN switches. However, the misunderstanding of SDN related protocols and the inconsistent implementation of SDN switches increase the risk of introducing errors, for example, packet loss or incorrect forwarding on SDN switches [11,12]. For example, according to the OpenFlow specification, the *Barrier* message is a way to notify that a set of OpenFlow operations have been completed [13]. Further, the SDN switch has to complete a set of operations issued prior to the *Barrier* before executing any further operation. If the SDN switch complies with the specification, we expect to receive a *Barrier* notification for a flow modification once the SDN switch has updated the flow table, implying that the change could be seen from the data plane. However, some SDN switches return a *Barrier* notification message as soon as they receive a *Barrier* request message from the controller, rather than after all flow table rules have taken

effect [9]. This inconsistent implementation will mislead the controller to produce erroneous processing behaviors, including packet loss, incorrect packet forwarding, performance reducing, and so on.

To evaluate the protocol correctness and performance of SDN switches, testing is a direct way to find out the problems in implementation. By constructing a group of specific input packets and determining whether the behavior of the SDN switch meets the expected settings, vendors or researchers can verify the protocol correctness and the performance of SDN switches. However, because of the characteristic of proprietary and limited flexibility, most existing commercial network testers, which are developed based on ASIC, could not fully support the lastest SDN southbound protocols or even an updated version of a specific southbound protocol. Due to the feature of closed-source, fully supporting user-defined test cases or test functions for SDN switches is still impossible for commercial network testers [9,10]. Besides, most commercial network testers only support passive statistical mode, that is, the testers only receive and analyze input packets for more test information but do not record specific behaviors of switches such as the forwarding behavior of packets [14,15]. The above limitations make commercial testers unable to test SDN switches.

Researchers in academia have proposed several open-source test tools for SDN switches which are made up of several different functional components, including the controller, the packet generator, the packet capture, and so on [15–17]. These functional components are implemented on different devices, including CPU, FPGA or ASIC. However, this brings the problem of clock asynchronization because the clock reference used in the control plane and the data plane are different, which introduces errors in time-related tests such as the correctness detection on the *Barrier* message. Based on these inaccurate testing results, researchers or vendors are hard to determine whether SDN switches meet the expected requirements. Taking full advantage of the flexibility and agility of the software, most test tools are implemented in software on PC. However, they could not provide accurate timestamps and stable rates for traffic generation because of the hardware limitation of PC and the scheduling process of the operating system. For some specific scenarios which need massive and stable network flow for performance test, it will not provide enough test support [18]. For some time-related test, there will be some errors and inaccuracies in the test. The above shortcomings limit the ability to thoroughly test SDN switches in terms of correctness and performance [10,13]. Correspondingly, how to achieve clock synchronization between the control plane and the data plane and how to ensure performance and accuracy are urgent problems to be solved.

In this paper, to overcome the problems mentioned above, we present a CPU & FPGA co-design Tester (CFT), which is a software and hardware co-design testing tool for SDN switches regarding protocol correctness and performance. The design of CFT is based on the following principles:

- To eliminate clock asynchronization, the hardware should be able to process both control plane packets and data plane packets at the same time. Therefore, we combine the control plane and the data plane into a single hardware pipeline and obtain the timestamps for different packets based on the same clock.
- To provide precise timestamps and specific traffic rates for some particular test scenarios, CFT takes full advantage of the hardware which behaves well in performance and accuracy. CFT uses FPGA as the traffic generation and capture engine and enables the functions of traffic generation and monitoring in a single hardware pipeline, also allows control plane packets to travel through the pipeline. In our pipeline, due to the reuse of hardware function modules, CFT could be able to support more features with as little hardware resources as possible.
- To support more user-defined test cases, CFT takes advantage of the flexibility and scalability of software to provide high-level and user-friendly APIs for users. Through these APIs, the functions supported by the hardware can be called by the software and the corresponding hardware registers or memory locations can be accessed directly. Therefore, users can develop their test cases according to the requirements they need based on the APIs provided by CFT.

There are many benefits of the proposed CFT. Firstly, CFT creatively combines the control plane and the data plane into the same hardware pipeline, which solves the previous problem of clock

asynchronization. Therefore, there is no additional overhead for clock synchronization. Secondly, CFT takes full advantage of the high performance and accuracy of the hardware and the flexibility and scalability of the software, which can satisfy any test scenarios for SDN switches. Thirdly, CFT adopts module multiplexing method in the hardware pipeline, which reduces the consumption of valuable hardware resources. Users are able to create new hardware function modules by following the specifications of interfaces between hardware modules, supporting more testing functions for SDN switches. Meanwhile, users are also allowed to develop their test cases or functions through provided APIs, which expands the coverage of the test. Fourth, CFT can support the test requirements of high performance and accuracy and correctness detection for SDN switches, which is beneficial to the development and deployment of SDN switches.

To evaluate our work, we conducted experiments to compare the abilities between the CFT, the commercial network tester and the open-source network tester. The experiments revealed that with CFT, users are able to develop a wide range of test cases through provided APIs regarding the correctness and the performance while keeping no-worse accuracy compared to commercial testers and similar flexibility in comparison with software-based test tools. At the same time, experiments have shown that the hardware pipeline we designed can support the fast insertion of hardware function modules and can consume as little hardware resources as possible.

We make the following contributions:

- We propose the CFT, a CPU & FPGA co-design Tester, which takes full advantage of hardware and software to enable precise and flexible test for SDN switches.
- We solve the problem of clock asynchronization by combining the control plane and the data plane into the same hardware pipeline so that timestamps for different packets are based on the same clock. To ensure normal communication of the control plane, we put forward a scheduling algorithm for arranging the output of control packets and data packets.
- We present a hardware architecture which allows traffic generation and traffic monitoring with high performance and precision in the same pipeline. Due to the reuse of modules and the specifications between modules, users can develop more functions with less consumption of hardware resources.
- We provide specific APIs for users to develop their test cases, which could cover a wide range of test for SDN switches.

The rest of the paper is organized as follows. Section 2 shows the overview of CFT. Section 3 presents the design of CFT in detail. In Section 4, we introduce the key technologies of CFT. We evaluate CFT in terms of the performance, correctness detection, and resource consumption in Section 5. Section 6 outlines the related work. Finally, we conclude the paper and point out the future work in Section 7.

2. Overview of CFT for SDN Switches

This section shows the basic idea of CFT and the hardware and software design of CFT. Besides, we present the workflow of CFT in this section.

2.1. Basic Idea of CFT

The basic idea of CFT is to adopt the method of software and hardware co-design, as shown in Figure 1. It takes full advantage of high flexibility and scalability of the software as well as high performance and accuracy of the hardware.

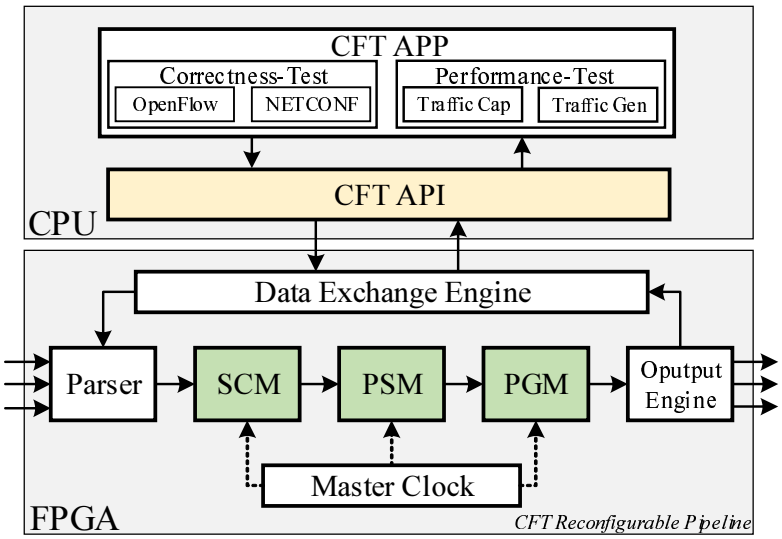

Figure 1. The architecture of the co-design tester (CFT).

The hardware of CFT is implemented on Field Programmable Gate Array (FPGA), which is widely used in network domain innovation and new function implementation [19,20]. We implement the functions of high-speed packets generation and high-precise packets capture, making full use of the advantages of hardware in performance and accuracy. At the same time, CFT also supports nanosecond-level timestamps for some time-related tests like latency and throughput. To eliminate clock asynchronization between the control plane and the data plane, CFT novelly allows the control packets to be transmitted in FPGA to record timestamps instead of NIC, which means we combine the control plane and the data plane into FPGA. The above functions appear in the form of functional modules in a hardware pipeline in FPGA. The hardware pipeline we design uses a module reuse approach to support more test functions while consuming as little hardware resources as possible. Standard interface specifications between hardware modules help users quickly insert user-defined modules to extend new test functions. We will present more details in Section 4.

Upon the hardware, there is the software of CFT, which is implemented on a standard Linux host. The software of CFT provides four kinds of APIs for users to develop customized test cases for different test scenarios. For example, to check the correctness of different southbound protocols, we should deliver protocol-related packets to tested SDN switches and response packets from switches. Therefore, we need to develop new functions, such as protocol-related packets construction and specific packets parsing, to satisfy the requirements of the test. By providing these software interfaces of reading and writing hardware registers or RAMs, CFT can hide the details of the underlying implementation and simplify the deployment of test cases, which extends the range of tests for SDN switches.

2.2. The Hardware Design of CFT

Here we introduce the core idea of the hardware design of CFT. As shown in Figure 1, CFT contains a single hardware pipeline in FPGA (we will discuss more in Section 4) which is the core of the CFT hardware. Due to the feature of the hardware, the CFT hardware pipeline provides the functions of high-performance traffic generation and high accuracy traffic monitoring. The pipeline also supports the nanosecond-level timestamps recording for packets. At the same time, to eliminate clock synchronization between the control plane and the data plane and to ensure that the upper CFT software part can parse and respond to the control plane packets from SDN switches to ensure the correctness of the test correctly and timely, control packets are allowed to travel through the pipeline.

In the stage of traffic generation, before generating, a template packet for the test will be sent from the software to PGM (Packet Generation Module) and PGM uses a block RAM to store the template packet. When starting to generate data plane traffic, PGM will read the packet from RAM and send it

to Output Engine continuously in a configured manner (more details in Section 3). Output Engine will forward these packets to the dedicated port.

In the stage of traffic monitoring, when the received packets enter the pipeline, they will firstly be parsed by Parser to extract key fields of the packet to compose the MetaData for further processing (we will discuss more details about MetaData in Section 4). According to the MetaData, CFT could determine which packet should be passed to the next module (e.g., a probe packet sent by CFT or a control packet from SDN switch) and which packet should be dropped (e.g., an LLDP packet). Followed by is SCM (Statistics Collection Module), which records specific packets and updates the statistic accordingly. Only control plane packets could be passed to the next module by SCM which means other packets will be dropped so that there will be no influence on the traffic generation.

To allow the control plane packets to travel through the pipeline without being dropped or making any influence on the data plane packets, a scheduling algorithm is implemented in PSM (Packet Schedule Module) to solve the output conflicts between the control plane packets and the data plane packets. We will discuss more about the algorithm in Section 3.

The CFT pipeline is composed of general modules as well as crucial modules. General modules (Parser, Output Engine, etc.) provide familiar network testing functions, including header parsing, packet forwarding and so on. We provide users with these general modules to reduce the workload of implementing new testing functions. The key hardware functional modules of CFT pipeline (green modules in Figure 1) are as follows:

- ***Statistics Collection Module (SCM).*** SCM is responsible for recording statistic (total number of bytes, etc.) and sampling specific packets stored into a block RAM. SCM supports the nanosecond-level timestamps for input packets to calculate time-related information, such as forwarding delay.
- ***Packet Schedule Module (PSM).*** PSM is used to assign priorities for different packets and schedule packets output according to the requirements of test scenarios. We implement a scheduling algorithm in PSM to schedule the output of the control plane packets and the data packets for the correctness of testing.
- ***Packet Generation Module (PGM).*** PGM is responsible for generating flow according to the pre-defined packet and parameters, including generating rate, time interval and so on. PGM also supports the nanosecond-level timestamps for output packets.

2.3. The Software Design of CFT

The core idea of the CFT software is to provide flexible and unified APIs with different functions for users to develop their test cases which are related to specific test scenarios. As shown in Figure 1, the CFT software is mainly composed of CFT API Layer and CFT APP, which are listed as follows:

- ***CFT API Layer.*** Based on CFT Lib Layer, CFT API Layer abstracts four types of programming interfaces, including *pkt_gen*, *pkt_cap*, *reg_wr* and *reg_rd*. The first two programming interfaces support the functions of packets generation and packets capture, which is used in traffic generation and monitoring. The latter two programming interfaces are responsible for reading and writing registers to set parameters or obtain statistical data. According to the above APIs provided in CFT API Layer, development of different test cases can be realized, covering various test scenarios for different test purposes. We will discuss more details about APIs in Section 3.
- ***CFT APP Layer.*** CFT APP Layer is composed of multiple test cases which are developed based on the APIs provided in CFT API Layer. Users could build their test cases according to the requirements of specific test scenarios. For example, to verify whether the flow table rules are valid, users can construct corresponding packets for different flow table rules and deliver them to verify the validity of flow table rules. Also, users can simulate a controller that supports a specific southbound protocol (e.g., OpenFlow, OF-CONF) to test the protocol correctness of SDN switches with related-protocol.

2.4. The Workflow of CFT

Tests for SDN switches include the performance test and the correctness test. As shown in Figure 2, the workflow of these two kinds of tests is as follows:

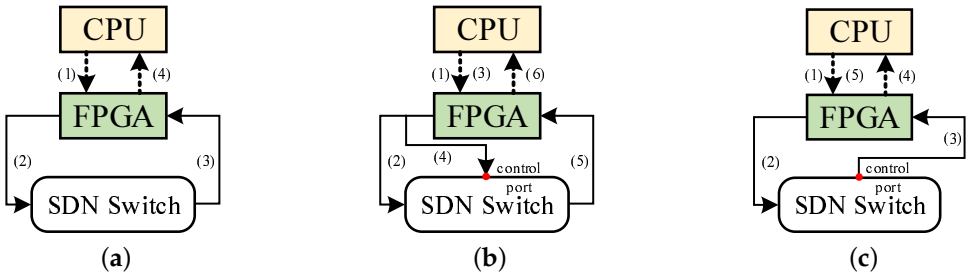

Figure 2. The workflow of the CFT; (**a**) performance, (**b**) correctness (actively), (**c**) correctness (passively).

2.4.1. The Performance Test

As shown in Figure 2a, the whole workflow of the performance test contains the following four stages:

(1) *Configuration Download.* When users want to conduct a performance test on SDN switches, template packets should be constructed first. Users can define the type, size and content of packets which are used for traffic generation or to use the packets already provided by the software. At the same time, users should set the parameters of traffic generation, including the sending rate, the sending time, the number of packets sent and the sending port. Parameters will be configured into the FPGA by calling APIs of *pkt_gen* and *reg_wr*.

(2) *Traffic Generation.* The template packets constructed by users are stored in the block RAMs of the FPGA. Test-related parameters are written into specific hardware registers of the FPGA. When receiving the start signal of traffic generation, the PGM reads packets from the block RAM and send them out continuously according to the parameters. The traffic will be generated at a pre-defined rate and forwarded through the specified port to the tested SDN switch. Each sent packet will be recorded with an output timestamp for further calculating in PGM. When the sending time expires or the number of packets sent is exceeded, traffic generation is terminated.

(3) *Traffic Monitoring.* When the CFT hardware receives packets from the SDN switch, the SCM will collect the test-related data, such as the number of received packets, bytes of received traffic, and so on. SCM also records an input timestamp for each received packet and will calculate the processing latency (input timestamp minus output timestamp) of the SDN switch. These test-related data will be saved into specific registers which could be obtained by CFT software test cases through CFT APIs we provided. Meanwhile, the SCM will sample some received packets and upload them to software test cases for further analysis.

(4) *Statistics Upload.* When the test is finished, the CFT software will read the registers and RAMs from the hardware FPGA to obtain the statistic through the APIs of *pkt_cap* and *reg_rd*. The software will calculate performance-related test results, such as throughput, processing latency and packet loss rate, based on the statistic. The test results will be presented to users in the form of text and charts.

2.4.2. The Correctness Test

For the test of protocol correctness, the CFT should be able to simulate the role of a controller. CFT needs to send the control plane packets (e.g., *flow_mod* message) to the tested SDN switch actively and respond to the control plane packets (e.g., *packet_in* message) from the tested SDN switch passively.

Send Control Packets Actively

As shown in Figure 2b, the whole workflow contains the following six stages:

(1) *Rule Setting.* To verify that the tested SDN switch complies with the protocol specification, users will construct the corresponding flow table rules, which are written into the SDN switch. Taking the verification of OpenFlow protocol as an example, users construct the *flow_mod* messages which contain the rules for the switch.
(2) *Rule Writing.* The *flow_mod* messages are forwarded to the tested SDN switch through the port directly connected to the switch management port.
(3) *Configuration Download.* The CFT needs to generate data plane traffic to verify that the rules in the tested SDN switch are valid. Therefore, users should construct specific packets for generating flow, which can hit the flow table rules. The operations are the same as the performance test.
(4) *Traffic Generation.* According to the constructed packets and pre-defined parameters, the PGM generates the data plane flow for the tested SDN switch. The operations are the same as the performance test.
(5) *Traffic Monitoring.* The SCM captures the packets from the specified port of the tested SDN switch and stores them for further analysis.
(6) *Rule Verification.* The software determines whether the flow table rules written into the tested SDN switch take effect by determining whether packets are captured from the specified switch port (the flow table rules decide) and whether packets are the constructed data plane packets.

Respond Control Packets Passively

As shown in Figure 2c, the whole workflow contains the following five stages:

(1) *Configuration Download.* Users need to construct a unique flow that can not be processed by the tested SDN switch, causing request messages (e.g., *packet_in* messages) from the switch. The operations are the same as the performance test.
(2) *Traffic Generation.* According to the constructed packets and pre-defined parameters, the PGM generates the data plane flow for the tested SDN switch. The operations are the same as the performance test.
(3) *Request Capture.* When the tested SDN switch can not process the received flow, it will send request messages to the CFT through the management port. The request messages, acting as control plane packets, will travel through the pipeline and be scheduled by the PSM.
(4) *Request Upload.* After receiving the packets, the CFT software parses the packets and extract the key fields of the packets.
(5) *Response Feedback.* According to the key fields of the packets, the CFT software constructs the respond messages like *flow_mod* or *packet_out* messages for the tested SDN switches. We will verify whether the response messages from the switch have taken effect by checking the correctness of operations (such as forwarding behavior) on the data plane packets.

3. CFT Design Details

Thus far, we have shown the CFT architecture. In this section, we proposed CFT design details.

3.1. Statistics Collection Module (SCM)

The main objective of the SCM is to collect test-related statistic and sample part of the packets according to the pre-defined sampling conditions for further analysis. As shown in Figure 3, we propose an FPGA-based statistic collection module, which contains:

(1) ***Input Parser.*** For each input packet, the Input Parser unit will parse the contents of the packet and extract the key fields of the packet. Based on the information obtained above, the Input Parser unit will determine whether the input packet is a data plane packet or a control plane packet. If the packet is a control plane packet, it will be sent to the Forwarding Engine unit. Otherwise, it will be sent to the next functional unit.

(2) ***TS Recorder.*** The TS Recorder unit is responsible for recording an input timestamp for each received data plane packet. The input timestamp will be used to calculate the time-related test items, such as processing latency. After recording the timestamp, the packet will be forwarded to the next functional unit.
(3) ***Statistics Engine.*** The Statistics Engine unit consists of registers for saving statistical data and block RAMs for storing sampled packets. When the data plane packet enters, the registers, which store the number of packets, bytes and so on, are updated. Sampled packets are written into the block RAMs according to pre-configured sampling rules. Unsampled packets are discarded after the relevant registers are updated with values.
(4) ***Forwarding Engine.*** For control planes packets and software-defined configuration packets, the Forwarding Engine unit forwards these packets directly to the next hardware functional module without any additional operations.

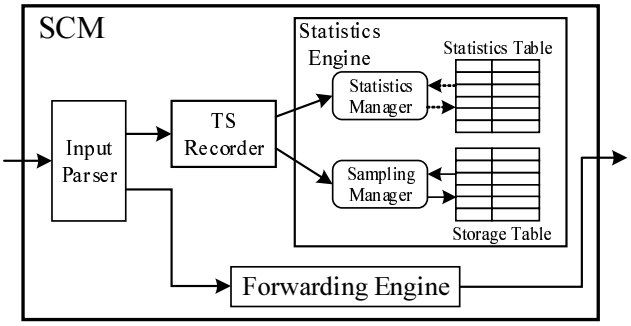

Figure 3. The design of Statistics Collection Module.

The SCM allows the CFT software to configure sampling rules for the operations of packet sampling. At the same time, the software can obtain the sampled packets stored in the block RAMs directly to analyze the internal process of the SDN switch further, such as recurring the packets forwarding behavior or the process of interactions with the controller. We call this function flow replay. The software can directly access the statistical registers in the SCM to calculate test-related items such as throughput and packet loss rate.

3.2. Packet Schedule Module (PSM)

The PSM is responsible for assigning different priority values to the control plane packets and scheduling the output of the control plane packets and the data plane packets according to the proposed scheduling algorithm. As shown in Figure 4, we put forward an FPGA-based packet schedule module, which contains:

(1) ***Input Parser.*** The Input Parser unit is used to distinguish whether it is a control plane packet or a software-defined configuration packet and forwards them to the corresponding functional units.
(2) ***Priority Management.*** The Priority Management unit is responsible for assigning different priority values to different control plane packets by looking up the Classification Table according to the extracted key fields. After assigning priorities, the Priority Management forwards the packets to the Forwarding Engine. At the same time, the Priority Management unit informs the Selection Engine unit in the PGM that the control hardware packets will be output through the hardware pipeline.
(3) ***Forwarding Engine.*** The Forwarding Engine unit forwards the software-defined configuration packets and the control plane packets with assigned priorities to the next functional unit.

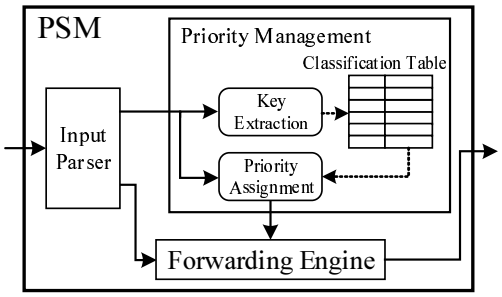

Figure 4. The design of Packet Schedule Module.

Supporting the control plane and the data plane in the same single pipeline will cause output conflicts between the control plane packets and the data plane packets. These conflicts can affect the test correctness of the CFT. Taking the test of the protocol correctness of the OpenFlow as an example, *echo* message in OpenFlow is used to interact between the SDN controller and the SDN switch to confirm that both are active. The frequency of interaction is generally from a few seconds to a dozen seconds. If there is no *echo* message interaction for some time, the connection between the two sides will be disconnected and the SDN switch enters "emergency" mode and clears all installed flow table rules. When the output port is used to generate the data plane packets for a long time, the control plane packets are blocked. The connection between the two sides will be broken and the process of the test will be terminated. The *packet_in* message in OpenFlow is used to upload a query message to the SDN controller when the SDN switch is not able to provide a flow table rule for the input flow. For these unmatched packets, some switches will temporarily store them in the cache while others encapsulate these packets into *packet_in* packets and upload them to the SDN controller. If this kind of control plane packets are blocked by the data plane packets and can not reach the controller, the switch will constantly upload *packet_in* message which will result in the loss of packets and the full filling of the pipeline, making it impossible to carry on other test operations.

Therefore, to solve the above problems, we propose and achieve the Priority-based Packets Scheduling (PPS) algorithm to schedule the output of different types of packets in the Priority Management unit of the PSM.

As shown in Algorithm 1, for each incoming packet, we distinguish the type of the packet according to the data structure *MD* (line 2 of the pseudo-code) which contains the relevant information needed for packets to be transmitted in the pipeline (we will discuss more in Section 4). In Table 1, we classify the control plane packets of the OpenFlow protocol based on their contents. According to the classification, we assign different priorities to different control plane packets and pass the signal of the control packets to the next hardware functional module (line 4 to line 7 of the pseudo-code). Thereby, when there are conflicts of output in the pipeline, the control plane packets with high priorities would be scheduled preferentially to ensure the normal interactions between the CFT and the SDN switch.

Algorithm 1 Priority-based Packets Scheduling (PPS)

Input: CFT_Pkt: The control plane packet from the previous module SCM.

Output: CFT_Pkt: The control plane packet with priority; FS: Forwarding signal to the Selection Engine unit in the PGM.

$index = 0$;
$pkt\_lookup\_key = cftpkt\_parser(CFT\_Pkt.ethpkt.header)$;
while $index < table\_size(ClassificationTable)$ **do**
  if $pkt\_lookup\_key == ClassificationTable[index].entry\_key$ **then**
    $CFT\_Pkt.MD.priority = ClassificationTable[index].entry\_pri$;
    $FS = 1$;
    $break$;
  else
    $index = index + 1$;
  end if
end while
if $index == table\_size(ClassificationTable)$ **then**
  $CFT\_Pkt.MD.priority = 0$;
  $FS = 0$;
end if
return CFT_Pkt, FS;

Table 1. The Classification of the Software Defined Networking (SDN) Control Plane Messages.

Message	Description	Priority	Remark
Modify_state	Controller manages the flow entries and port status of switch.	4	Reject Any Delay
Send_packet	Controller specifies the port on the switch that sends packets.		
Barrier	Confirm receipt of packets or completion of operations.		
Packet_in	Request controller to process packets.		
Flow_removed	Tell controller that flow entries have been deleted.		
Port_status	Tell controller that switch port has changed.		
Error	Tell controller that switch has failed.		
Hello	Used to initiate connection establishment between switch and controller.	3	Accept Appropriate Delay
Echo	Negotiate parameters and confirm alive.		
Features	Controller asks for functions supported by switch.	2	Insensitive to Delay
Configuration	Set or query configuration parameters of switch.		
Read_state	Query statistics on switch.		
Vendor	Functions defined by vendors.	1	Determined by Vendors

3.3. Packet Generation Module (PGM)

The PGM is used to generate the data plane traffic based on the pre-defined parameters and template packets provided by the software. As shown in Figure 5, we put forward an FPGA-based packet generation module, which contains:

(1) ***Input Parser.*** The Input Parser unit is used to distinguish whether it is a control plane packet or a software-defined configuration packet. If the packet is a control plane packet, the Input Parser forwards it to the Selection Engine unit. Otherwise, the Input Parser forwards it to the Traffic Generation unit.

(2) ***Traffic Generation.*** The Traffic Generation unit is used to continuously send template packets according to the configuration parameters provided by the CFT software, thereby generates the data plane traffic.
(3) ***TS Recorder.*** For each output packet, the TS Recorder unit is responsible for recording an output timestamp for the data plane packet, which will be used for further calculation and analysis.
(4) ***Selection Engine.*** The Selection Engine is used to determine the output of the control plane packets and the data plane packets based on the priorities and to receive the forwarding signal from the PSM.

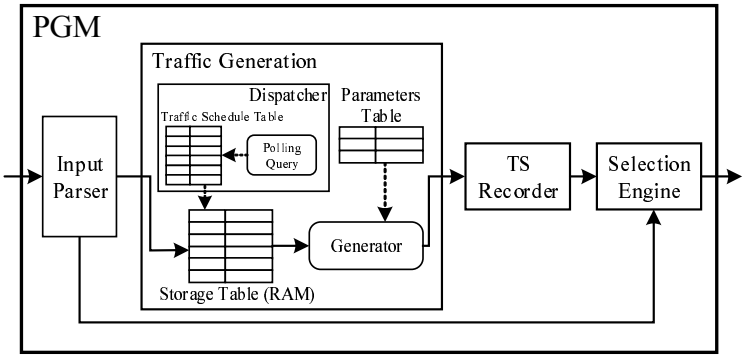

Figure 5. The design of Packet Generation Module.

To generate the network flow for the tested SDN switch, the Traffic Generation unit needs to produce a single flow or multi-flow based on the template packets provided by the software. For a multi-flow generation, we control the rate at which traffic is generated by controlling the sending interval between packets in the hardware FPGA. The output conflicts between concurrent multi-flow are avoided by adjusting the initial transmission time of different flow. Therefore, users need to configure the initial transmission time and transmission interval for each flow in the software. The initial transmission time, the transmission interval, and the storage address of the packet in block RAMs form a table entry of the Traffic Schedule Table, as shown in Table 2. The Traffic Generation unit internally maintains a timer and polls the Traffic Schedule Table to compare whether the current time and the initial transmission time of the entry are equal. If equal, the packet will be obtained from the block RAMs according to the address obtained in entry and sent to the TS Recorder unit. At the same time, the initial transmission time is updated to the original initial transmission time plus the transmission interval ($T = T + \Delta T$). If not, the Traffic Generation unit will poll other entries and compare the initial transmission time with the timer. The process of polling and comparison is terminated until the flow generation ends.

Table 2. The Definition of the Traffic Schedule Table.

Initial Transmission Time (T)	Transmission Interval (ΔT)	Address in RAM ($Addr$)
T_0	ΔT_0	$Addr_0$
T_1	ΔT_1	$Addr_1$
T_2	ΔT_2	$Addr_2$
...	...	...
T_n	ΔT_n	$Addr_n$

However, in the process of concurrent multi-flow generation, there must be a conflict, that is, at the current moment, more than one traffic meets the transmission requirements (the initial transmission time is equal to the present time). To solve this problem, we adopt a strategy, called compensation

transmission. The strategy is first to send the packet with a higher priority (stored in a high priority hardware queue). If the condition of sending the packet with a higher priority is not satisfied, the strategy to send a packet whose initial transmission time is less than the current time. We abstract the above processes into an algorithm called the Time-based Polling Scheduling (TPS), as shown in Algorithm 2.

Algorithm 2 Time-based Polling Scheduling (TPS)

Input: $GenModel$: Select the mode of the traffic generation (according to the total time or the number of packets); $TransTotalTime$: The total time to generate traffic; $TransTotalNum$: The total number of packets sent.

Output: $PktAddr$: The storage address of the packet in RAM;

1: $index = 0$;
2: **if** $GenModel == 0$ **then**
3: $timer = 0$;
4: **while** $timer < TransTotalTime$ **do**
5: **if** $timer >= TrafficScheduleTable[index].T$; **then**
6: $PktAddr = TrafficScheduleTable[index].Addr$;
7: $TrafficScheduleTable[index].T = TrafficScheduleTable[index].T$
8: $+TrafficScheduleTable[index].\Delta T$;
9: **return** $PktAddr$;
10: **end if**
11: $index = index + 1$;
12: $timer = timer + 1$;
13: **if** $index == tablesize(TrafficScheduleTable)$ **then**
14: $index = 0$;
15: **end if**
16: **end while**
17: **else**
18: $counter = 0$;
19: $timer = 0$;
20: **while** $counter < TransTotalNum$ **do**
21: **if** $timer >= TrafficScheduleTable[index].T$; **then**
22: $PktAddr = TrafficScheduleTable[index].Addr$;
23: $TrafficScheduleTable[index].T = TrafficScheduleTable[index].T$
24: $+TrafficScheduleTable[index].\Delta T$;
25: **return** $PktAddr$;
26: **end if**
27: $index = index + 1$;
28: $timer = timer + 1$;
29: $counter = counter + 1$;
30: **if** $index == tablesize(TrafficScheduleTable)$ **then**
31: $index = 0$;
32: **end if**
33: **end while**
34: **end if**

In Algorithm 2, if the current time is greater than or equal to the Initial Transmission Time (satisfying the condition of sending packets), obtain the storage address of the packet in RAM and update the Initial Transmission Time (line 5 to line 10 and line 21 to line 26 of the pseudo code). If not satisfying the condition of sending packets, poll the Traffic Schedule Table and compare the Initial

Transmission Time with the current time (line 11 to line 15 and line 27 to 32 of the pseudo-code). If the duration of the traffic generation or the total number of generated packets is exceeded, the process of the traffic generation is terminated (line 4 and line 20 of the pseudo-code).

When the control plane packets need to be forwarded, the Selection Engine unit determines whether to suspend the transmission of the data plane packets to send the control plane packets immediately or send the control plane packets after a while according to the priorities of the control packets, as shown in Table 1. Therefore, we design and implement an algorithm for precise control of packets output, called Cache-based Packet Output (CPO), in the Selection Engine unit, as shown in Algorithm 3.

Algorithm 3 Cache-based Packet Ouput (CPO)

Input: CFT_Pkt: Packets from the previous module PSM; TG_Pkt: Packets of the traffic generation; FS: Forwarding signal to the Selection Engine unit in the PGM.
Output: $Output\_Pkt$: Packets output to the next module Output Engine.

```
1:  if FS == 1 then
2:      if CFT_Pkt.MD.priority == 4 then
3:          Output_Pkt = CFT_Pkt;
4:          tmp_store(TG_Pkt);
5:          if controlSentFinish then
6:              Output_Pkt = TG_Pkt;
7:          end if
8:      end if
9:      if CFT_Pkt.MD.priority == 3 then
10:         tmp_store(CFT_Pkt);
11:         if timer == thresholdT then
12:             Output_Pkt = CFT_Pkt;
13:         else
14:             Output_Pkt = TG_Pkt;
15:         end if
16:     end if
17:     if CFT_Pkt.MD.priority == 2 then
18:         tmp_store(CFT_Pkt);
19:         if timer == userDefinedT then
20:             Output_Pkt = CFT_Pkt;
21:         else
22:             Output_Pkt = TG_Pkt;
23:         end if
24:     end if
25:     if CFT_Pkt.MD.priority == 1 then
26:         Output_Pkt = TG_Pkt;
27:         vendor_operation(CFT_Pkt);
28:     end if
29: else
30:     Output_Pkt = TG_Pkt;
31: end if
32: return Ouput_Pkt;
```

In Algorithm 3, for the control packets with highest-priorities, the Selection Engine unit forwards them immediately and stores the unsent data plane packets into a cache (line 2 to line 4 of the pseudo-code). After sending the control plane packets, the data plane packets in the cache are

forwarded (line 5 to line 7 of the pseudo-code). For other control packets, the Selection Engine unit first stores them into a cache and sends them after a while (line 9 to line 28 of the pseudo-code).

3.4. APIs for Test Cases

The test for the SDN switch in the CFT, either it is a performance test or a protocol correctness test, is based on the operations of sending and receiving packets and reading and writing registers. Therefore, the core of the CFT software is to abstract and provide four types of APIs for operating packets and registers. Our provided APIs together with the source code of user test cases which are both written in C language can be compiled to execution code by the GCC. Details of the APIs are as follows:

3.4.1. Operations on Packets

Whether it is traffic generation and monitoring of the data plane or control messages interaction and response of the control plane, these operations are implemented on the basis of sending and receiving packets. The details of these two APIs are as follows:

(1) ***pkt_gen(pkt).*** Through the API *pkt_gen(pkt)*, users can encapsulate their constructed packets (*pkt*) into a specific data format which is passed in the hardware pipeline. Packets will be sent after being processed in the pipeline.

(2) ***pkt_cap(cond).*** Users can set the conditions (*cond*) for receiving packets, such as a specific protocol, a specific IP address, a specific input port and so on. The packets captured by calling the library in the CFT Lib Layer are returned to users through the API *pkt_cap(cond)* after being filtered by *cond*.

We take the response to the *packet_in* message as an example to introduce the practical application of these two APIs. As shown in Figure 6a, we use the API *pkt_cap(cond)* to capture the control plane packets and the *cond* is the input port, which is directly connected to the SDN switch management port. Then, we determine the type of packet by parsing the content of the packet. As shown in Figure 6b, for *packet_in*, we construct a response packet *flow_mod* which contains the related rule for the data plane packets and send this packet through the API *pkt_gen(pkt)*.

```
void control_request_cap() {

  EthPkt  pkt;
  Cond  cond_cap;

  // set filter condition
  cond_cap.inport = connected_port;
  // use pkt_cap API to capture pkt
  pkt = pkt_cap(cond_cap);

  // distinguish of_req_pkt
  switch (pkt.ofp_header.type) {
    case Ehco: ...;
    case Hello: ...;
    case Port-Status: ...;
    ...
    case Packet_in:
      // lookup the table
      flowEntry = lookup_table(pkt);
      // construct the resp_pkt
      control_response_gen(flowEntry);
      break;
    default: ...;
  }
}
```

(**a**)

```
void control_response_gen(FlowTableEntry entry) {

  CFTPkt  resp_pkt;

  // set parameters for Metadata
  // set protocol
  resp_pkt.md.protocol = Flow_mod;
  // set MID
  resp_pkt.md.mid = 5;
  // set length
  resp_pkt.md.len = control_pkt_len;
  // set src, 0:netwrok, 1:cpu
  resp_pkt.md.src = 1;
  // set dst, 0:network, 1:cpu
  resp_pkt.md.dst = 0;
  // set outport
  resp_pkt.md.outport = connected_port;
  ...
  // set payload
  resp_pkt.payload = entry;

  // use pkt_gen API to send pkt
  pkt_gen(resp_pkt);

}
```

(**b**)

```
void traffice_generation(CFTPkt genPkt) {
  ...
  // set Meatdata
  // set MID, 4:PGM
  genPkt.md.mid = 4;
  // set length
  genPkt.md.len = config_pkt_len;
  // set dst, 0:network, 1:cpu
  genPkt.md.dst = 0;
  // set outport
  genPkt.md.outport = config_output_port;
  •••
  // config the template pkt
  pkt_gen(genPkt);
  // set generation parameters
  // 1: sending time (second)
  reg_wr(5, 1, gen_time);
  // 2: sending rate
  reg_wr(5, 2, gen_speed);
  ...
  if (start_flag) {
    //0: start signal of generation
    reg_wr(5, 0, 1);
  }
}
```

(**c**)

```
void statistic_obtain() {

  // obtain the statistics in SSM, 2:SSM
  ...
  // 3:bit_count_reg
  bit_count = reg_rd(2, 3);
  // 4:pkt_count_reg
  pkt_count = reg_rd(2, 4);
  ...

}
```

(**d**)

Figure 6. The application examples of the provided APIs.; (**a**) request control plane pakcet, (**b**) respond control plane packet, (**c**) traffic generation, (**d**) obtain statistic.

3.4.2. Operations on Registers

Configuring the parameters of traffic generation and obtaining statistic are both based on the operations of reading and writing registers. The details of these two APIs are as follows:

(1) ***reg_wr(MID, wrNum, wrVal).*** When modifying a register, users should provide the module number (*MID*), the register number (*wrNum*) and the value written into the register (*wrVal*).

By calling the API *reg_wr*, a constructed packet of writing register will be passed to the pipeline and the relevant register will be modified soon.

(2) ***reg_rd(MID, rdNum).*** When obtaining the value from the register, users need to provide the module number (*MID*) and the register number (*rdNum*). A constructed packet that carries the value will be obtained by calling the library in the CFT Lib Layer and the value will be returned to users through the API *reg_rd(MID, rdNum)*.

In Figure 6c, for users who need to configure the parameters of traffic generation (such as transmission interval), they need to provide the *MID* of the PGM (we define it as four), the number of the register and the value of transmission interval. Through the API *reg_wr*, they can modify the transmission interval between packets. As shown in Figure 6d, when users need to obtain the total number of received packets, users should provide the *MID* of the SCM (we define it as two) and the number of the register which stores the number of packets and call the API *reg_rd(MID, rdNum)*.

3.4.3. API Extension

The CFT APIs are developed based on the libraries in the CFT Lib Layer, including *libnet* and *libpcap*. Therefore, Users can develop their new APIs by calling these general libraries in the CFT Lib Layer, without learning new libraries. To develop new APIs, users only need to follow the steps below:

(1) Users need to develop the format of the packets used for interaction between the CPU and the FPGA;
(2) According to their specific requirements, users develop customized APIs based on the general libraries *libnet* and *libpcap* and complie them;
(3) Encapsulate these compiled APIs into the library *libcft* which contains the CFT APIs we provided. Therefore, different users can use these APIs to develop specific test cases by calling the library *libcft*.

4. Key Technologies of the Proposed CFT

In this section, we present several key technologies of our proposed CFT design.

4.1. CFT Reconfigurable Pipeline

The most important part of the CFT hardware is the reconfigurable pipeline which supports different testing functions in parallel and is implemented on FPGA with Verilog language. The pipeline is composed of several hardware functional modules. Each module is responsible for a specific testing function, such as PGM is used to generate traffic while SCM is used to monitor traffic. For each "packet" entering the module, the module first determines if the "packet" should be processed by itself. The "packet" mentioned here refers to a custom data structure (we call it CFT-Pkt) passing between hardware modules and we will discuss later. The CFT-Pkt carries the *MID* of the module that needs to process itself. Each module compares its *MID* with the *MID* carried in the CFT-Pkt. If they are equal, the CFT-Pkt is processed by the module. If not, the module forwards the CFT-pkt directly to the next module. We call this forwarding action *"bypass"*.

Based on this *"bypass"* mechanism, each module only processes the packets associated with itself and does not affect the unrelated packets. Although physically a hardware pipeline, it can be logically abstracted into multiple parallel pipelines to support different testing functions [21]. Therefore, in the same pipeline, we can not only perform different processing operations on packets (e.g., traffic generation and monitoring for packets) but also process different packets (e.g., the control plane packets and the data plane packets) at the same time. All modules in the same pipeline are based on the same time basis, so that the problem of clock synchronization between the control plane and the data plane can be well solved.

We define the specification of the interfaces between the modules in the hardware pipeline. All modules that follow this specification can be easily plugged into the hardware pipeline to provide new testing functions, which makes the pipeline scalable and expands the test coverage of the CFT.

In Figure 7, we compare the hardware pipeline design of the CFT with the hardware pipeline design of existing test tools [10,19] and further compare their extensions on the new test functions. We find that we can support the new test function by inserting specific function modules that conform to the inter-module interface specification and can call the common functions by multiplexing the function modules, in the CFT hardware pipeline. Adding new test functions to existing test tools requires a new functional pipeline to be developed and a large number of hardware modules of the same function are developed repeatedly. As shown in Figure 7, when we decide to add a new test function of sampling packets in a flow, we only need to develop a Sampler module which conforms to the interface specification and insert it into the CFT pipeline. However, existing tools need to build a new functional pipeline, supporting the function of sampling. The module of InputTS and Collector are developed repeatedly, which consumes valuable hardware resources and affect the deployment of the new function. In Section 5, we demonstrate that the hardware pipeline we designed and implemented consumes as little hardware resources as possible to support more testing functions.

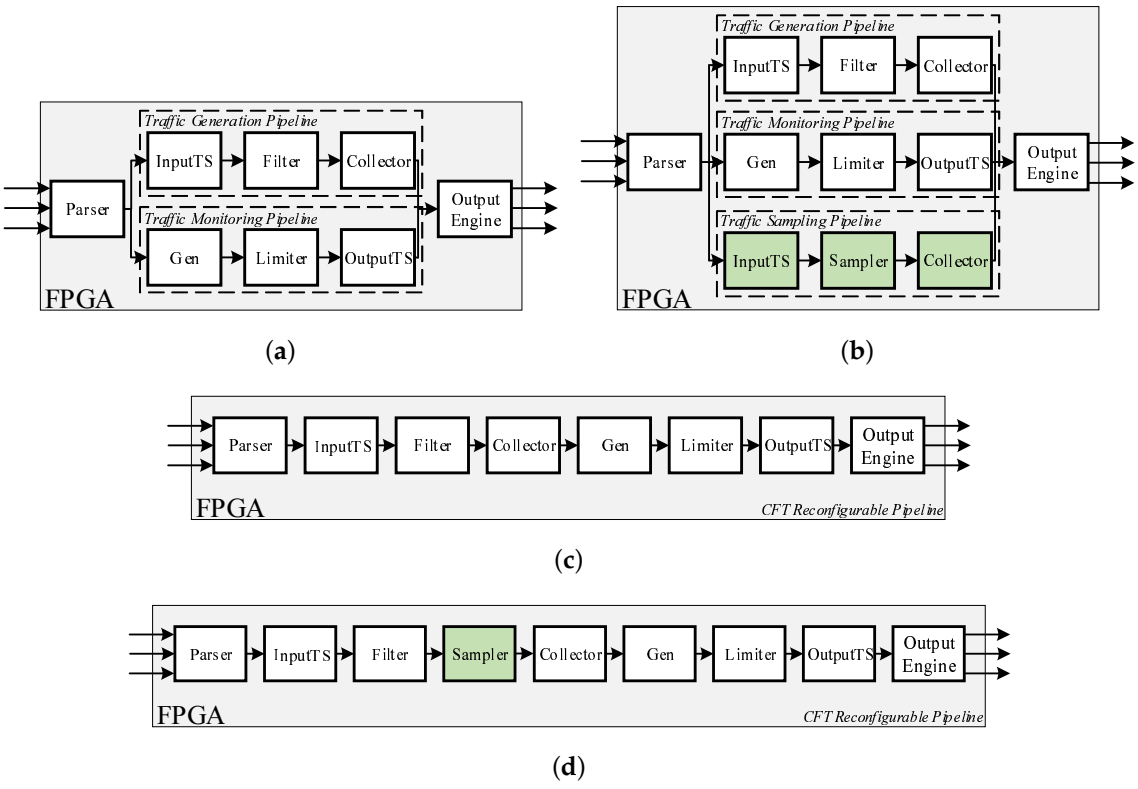

Figure 7. The differences in the pipeline extension; (**a**) the pipeline of existing tools, (**b**) the extension pipeline of existing tools, (**c**) the pipeline of CFT, (**d**) the extension pipeline of CFT.

4.2. Key Data Structure in the Pipeline

From above, we show that CFT-Pkt is a core data structure across all modules in the CFT hardware pipeline. The CFT-Pkt is composed of the MetaData (MD) and the standard Ethernet packet, as shown in Figure 8.

The standard ethernet packet is constructed by the users and carries the data and information required for different functions, such as the control plane response packet or the template packet for a multi-flow generation.

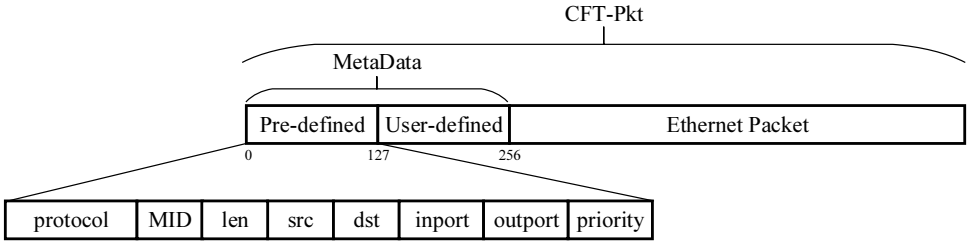

Figure 8. The format of the MetaData and fields in pre-defined segment.

Each standard Ethernet packet that enters the CFT hardware pipeline carries its MetaData. The MetaData is a 32-bytes data which includes a 16-bytes pre-defined segment and a 16-bytes user-defined segment. The pre-defined segment contains several useful information, such as protocol type (8 bits), MID (8 bits), input port (6 bits), packet length (12 bits), etc, which enables multi-processing for the CFT-Pkt. At the same time, most modules can process packets based on the contents of the MetaData, which avoids repeated operations like parsing packets and speeds up the processing of the modules. In case that some users may have additional requirements, we reserve the 16-bytes user-defined segment for user definition. By using MetaData in our proposed pipeline, we make different modules more flexible and efficient in processing the packets.

5. Evaluation

In this section, we evaluate our implementation of the CFT using Xilinx Zynq 7000 Board [22] and standard Linux host as a prototype development platform. Our evaluation focuses on three main aspects of the CFT: (1) Precision and Performance; (2) Protocol Correctness Detection; (3) Resource Allocation.

5.1. Precision and Performance

To evaluate the precision and performance of the CFT, we test a commercial OpenFlow-enabled SDN switch in terms of the throughput and processing latency with both CFT and off-the-shelf IXIA network tester [23]. The SDN switch is Pica8 P-3297 [24] which has 48 ports of 1GE and 4 ports of 10GE.

In the experimental setup for precision, we use the CFT to generate UDP packets of different size for the data plane. Before the test, we connect the two ports of the CFT directly to eliminate the originally existed latency (avoid errors when measuring the processing latency of Pica8) in the prototype development platform. We run the experiments to obtain the latencies for 100 groups and each group corresponds to a packet of a specific size (ranging from 64 Bytes to 1024 Bytes). During the test of the precision and performance of Pica8, we run the test for 50 groups and record both the throughput and the latency. The only difference is that we replace the CFT with the IXIA network tester [23].

The comparison of the throughput and processing latency of Pica8 between the CFT and the IXIA is demonstrated in Figure 9. According to the results in Figure 9a, we can find that the throughput of the CFT and the IXIA is almost same, with a difference of no more than 0.1%. Since the working frequency of the CFT is 100Hz, we can find that the CFT is able to provide nanosecond-level timestamps for packets, from Figure 9b. The difference between the latencies obtained by the CFT and the IXIA is less than 1%, according to Figure 9b. Based on the above results, we ensure that the CFT has no worse performance and precision than commercial network testers.

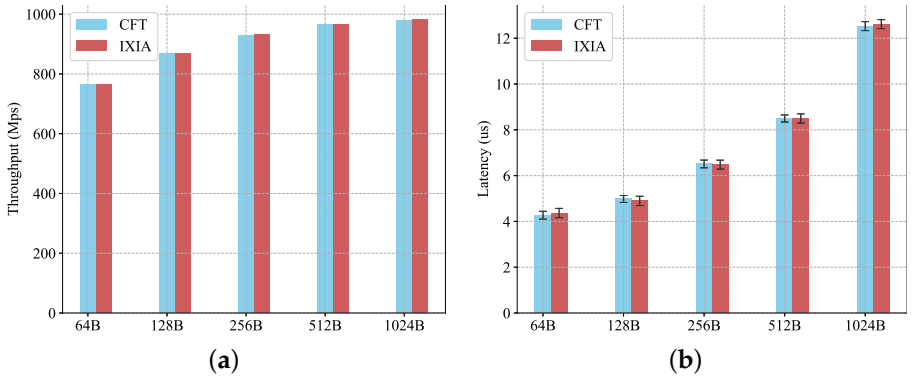

Figure 9. The comparison of the performance between the CFT and the IXIA network tester regarding the throughput and latency; (**a**) throughput, (**b**) latency.

Compared to software tools, since the CFT has an FPGA Board, it can generate traffic for the data plane with high performance. So, in the experimental setup for performance, we compare the CFT with a popular software tool for traffic generation, Iperf [25], which runs on a laptop with the operating system of Ubuntu 16.04, a 1GE port and an Intel i5 quad-core processor. During the whole test (60 s for traffic generation), there is no user process running on the operating system except Iperf. We use the CFT and the Iperf to generate UDP packets with different traffic rate we set, ranging from 600 Mbps to 900 Mbps respectively and record the actual generated rate by the IXIA network tester.

As shown in Figure 10, the traffic rate of the CFT is more stable than the Iperf. In each round, the CFT can keep a constant and precise rate of traffic generation as we set. However, the rate of the Iperf experiences a significant jitter during the whole test. Besides, when we set the traffic rate from 600 Mbps to 800 Mbps, the actual rate of the Iperf is a little higher. When we set 900 Mbps, Iperf can only reach 877 Mbps on average. Since the Iperf is developed based on the Socket API, it may be impacted by the process scheduling or interrupt mechanism inside the operating system.

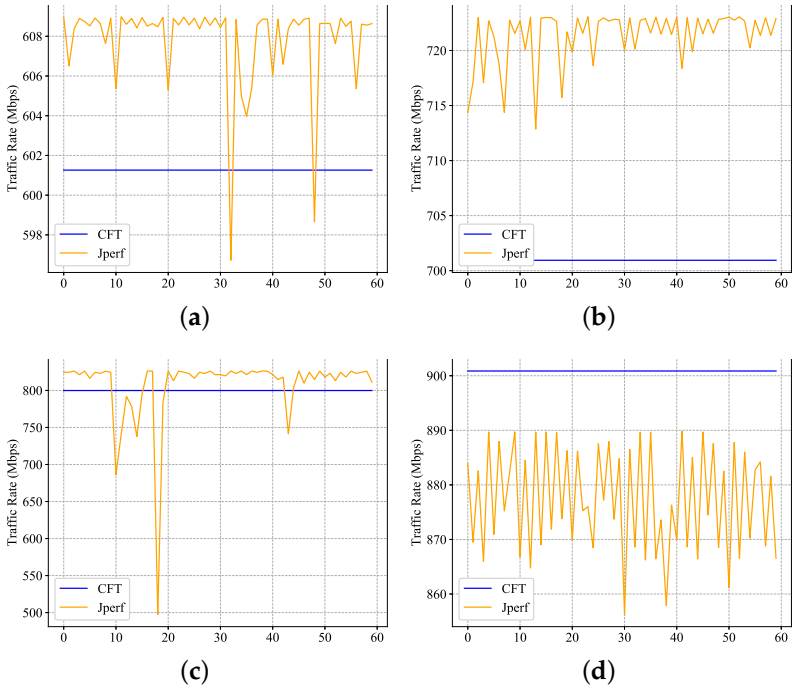

Figure 10. The comparison of the performance between the CFT and the Iperf software network tester at different rates; (**a**) traffic rate—600 Mbps, (**b**) traffic rate—700 Mbps, (**c**) traffic rate—800 Mbps, (**d**) traffic rate—900 Mbps.

5.2. Protocol Correctness Detection

Many solutions essential for the correctness and reliability of the OpenFlow deployment rely on knowing when the SDN switch applied a given command from the controller in the data plane [13]. The *Barrier* message is used to tell if the command from the controller has been applied in the SDN switch. According to the specification of the OpenFlow, once the SDN switch received the *Barrier_request* message, it needs to complete all the transactions and send the *Barrier_response* message to the controller after finishing all the transactions [1]. As shown in Figure 11a, there is a time difference between the time when the controller sends the *Barrier_request* message and the time when the controller receives the *Barrier_response* message. The size of the time difference depends on the number of transactions that need to be completed. However, some SDN switches response to the controller once they received the *Barrier_request* message without completing the transactions, as shown in Figure 11b [9]. Therefore, in this section, we verify whether the CFT can detect the protocol correctness of the SDN switches by testing whether the implementation of the *Barrier* message conforms to the OpenFlow specification.

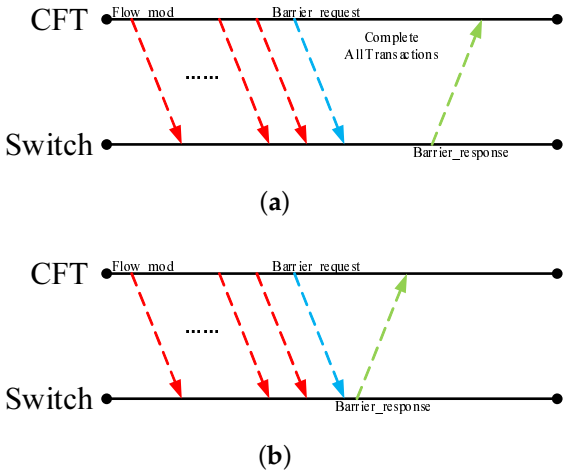

Figure 11. The different implementations of the *Barrier* message; (**a**) the correct implementation of the *Barrier* message, (**b**) the incorrect implementation of the *Barrier* message.

To verify that the *Barrier* message is implemented correctly, we observe the workflow of the *Barrier* message from the perspective of the control plane and the data plane. We expect to receive a *Barrier_response* message in the data plane once the flow table of the SDN switch has been updated, implying that the change can be seen from the data plane. We record the time when the *Barrier_request* message arrived at the controller and the arrival time of the first data plane message that was forwarded after the *Barrier_request* message from the controller. Because the control plane packets and the data plane packets received by the CFT are both processed by the CFT pipeline, the timestamps of all received packets are based on the same clock. Therefore, by comparing the two arrival time, it can be determined whether the *Barrier* is implemented according to the specification.

We construct 50 UDP flow with different source ports and destination ports and generate in a round-robin fashion on the data plane. For each flow, we set a forwarding rule that matches the source and destination ports of the flow. The forwarding rule is used to forward the traffic to the specified port of the tested SDN switch. We also set a low priority rule which discards all the packets. The low priority rule prevents the SDN switch sending the *Packet_in* message to the CFT when the CFT has not configured the forwarding rules. We perform 50 rounds of testing and insert the corresponding rule i to forward the matching flow i in the round i, where i is from 1 to 50. At the same time, we record the arrival time of the *Barrier_response* message and the arrival time of the first packet received from the specified port in each round i. As the number of rounds increases, the number of forwarding rules in

the SDN switch increases accordingly. Therefore, the arrival time of the *Barrier_response* message and the arrival time of the first packet received from the specific port both increase consequently, because of an increase in the number of transactions before the *Barrier_response* message.

From Figure 12a, we observe that the Pica8 P-3297 correctly implements the mechanism of the *Barrier* message, because as the number of round increases, the time at which the *Barrier_response* message is received and the arrival time of the first forwarded packet both increase, while the former arrives before the latter. This show that after receiving the *Barrier_request* message, the Pica8 P-3297 processes all the previous transactions before returning a *Barrier_response* message to the controller, which conforms to the OpenFlow specification. However, the OVS (Open VSwitch) and the Switch 1 do not conform to the specification because the arrival time of the *Barrier_response* does not increase continuously but remains constant, while the time to receive the first forwarder packet is increasing, according to Figure 12b,c. This shows that the *Barrier_response* is not sent after the previous transactions are processed and can not be used to confirm whether the changes in the data plane have been updated. Based on the above results, we prove that the CFT can effectively detect the protocol correctness of the SDN switch. Meanwhile, we also prove that the necessity of merging the control plane and the data plane into a single pipeline, ensuring that the time when the control plane receives the *Barrier_response* message and the time when the first packet is received by the data plane are both based on the same clock.

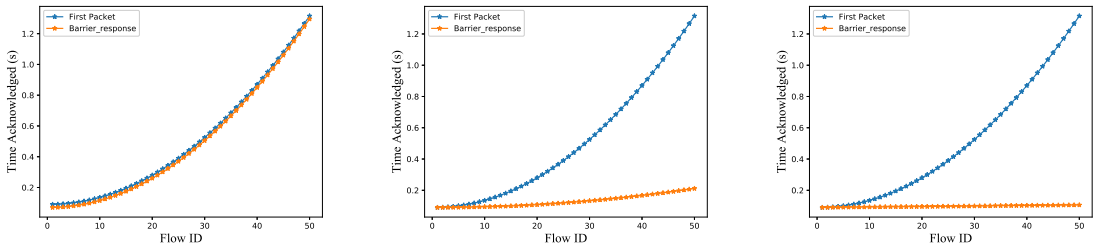

Figure 12. The correctness detection of *Barrier* on different SDN switches; (**a**) Pica8 P-3297, (**b**) Open VSwitch, (**c**) Switch 1.

5.3. Resources Allocation

As we described in Section 4, due to the characteristic of the CFT reconfigurable pipeline we proposed, we can implement more testing functions with as little hardware resources as possible. Therefore, different from existing work, we implement both the traffic generation and traffic monitoring in a single pipeline. So, we evaluate the actual resource occupancy percentage of the CFT in this section. In Table 3, the first percentage is hardware resources of the hardware module to total hardware resources of the CFT pipeline while the second percentage is hardware resources of the hardware module or the CFT pipeline to total hardware resources of Xilinx Zynq 7000 SoC Board.

Table 3. Resource Usage on Xilinx Zynq 7000 SoC Board.

	LUTs	Registers	Block RAM
SCM	616 (8.86%,1.16%)	601 (6.35%,0.56%)	1 (3.70%,0.71%)
PGM	2624 (37.76%,4.93%)	1553 (16.42%,1.46%)	6 (22.22%,4.29%)
CFT Pipeline	6949 (/,13.06%)	9459 (/,8.89%)	27 (/,19.29%)
Total Resources	53200 (/,/)	106400 (/,/)	140 (/,/)

The CFT hardware pipeline is implemented upon the five stages pipeline of FAST [26] and we insert three functional modules (SCM, PSM and PGM) into the pipeline. On the Xilinx Zynq 7000

Board, the comparisons of resource usage between the plugin modules and the CFT pipeline are listed in Table 3. From Table 3, we can find that the modules we insert only introduce about 47% LUTs, 23% Registers and 26% Block RAM usage compared to the whole pipeline. The plugin modules only occupy 6.09 % LUTs, 2.02% Registers and 5.00% Block RAM of the Xilinx Zynq 7000 Board.

In Table 4, we present the comparison between the five stages of FAST and the CFT reconfigurable pipeline. Compared to the FAST pipeline, we can find that our proposed pipeline has only increased the resource consumption of LUTs by 6.26%, Registers by 2.4% and Block RAM by 5.72%, which means that the CFT supports two new testing functions with a little resource consumption. However, if we use two separated pipelines for traffic generation and monitoring, there will be a significant improvement in resource usage. Therefore, for resource-intensive devices such as FPGA, the CFT reconfigurable pipeline is able to provide more flexibility in implementing additional functions.

Table 4. The Comparison of the CFT Reconfigurable Pipeline and the FAST Five-stage Pipeline in Resource Usage.

	LUTs	**Registers**	**Block RAM**
CFT Pipeline	13.06%	8.89%	19.29%
FAST Pipeline	6.80%	6.49%	13.57%
Increase	6.26 %	2.4%	5.72%

6. Related Work

SDN switches are accepted by vendors and researchers increasingly as they support the programming of the data plane which accelerates the innovation of network and the development of new protocol and functions [1,17]. To ensure that SDN switches conform to the specification of the southbound protocol and achieve the expected performance and accuracy, the industry and academia have conducted some researches on the test of SDN switches.

OFLOPS [9] is a software tester which provides detailed measurements for OpenFlow-enabled switches. OFLOPS is able to simulate both the control plane and the data plane in software based on NIC or to extend the data plane to NetFPGA [27] to improve the performance of the traffic generation and monitoring. OFSuite_Performance [28] is a Linux-based software test tool which is responsible for testing the performance of OpenFlow-enabled switches in the data plane. OFTest [18] is implemented in Python to focus on the protocol correctness of the OpenFlow-enabled SDN switches. OFCProbe [29] is aimed at verifying the performance of SDN controllers. Although these software test tools provide high flexibility, they still need to be improved in terms of the performance of the data plane and the accuracy of the timestamps. The tools mentioned above are all developed for the OpenFlow protocol and lack extended support for other southbound protocols.

OFLOPS-Turbo [10] is the integration of the OFLOPS with the OSNT [19] platform, which improves the performance and accuracy in the data plane. FlowScope [30] is mainly responsible for packets capture and storage in high performance with optimized data structure and hardware support. However, the above tools have to take extra work to eliminate the time asynchronization between different devices and introduce additional resource consumption and time delay.

ndb [14] is an SDN network debugger for programmers, which implements two primitives (*breakpoints* and *packet backtraces*), useful for debugging an SDN. NICE [11] is an efficient software detection tool for uncovering bugs in OpenFlow applications through a combination of model checking and symbolic execution while SOFT [12] is mainly responsible for figuring out the inconsistent between different OpenFlow agents. Compared with the CFT, the tools mentioned above are only the detection of OpenFlow application bugs in the field of software engineering, rather than focusing on the characteristics of SDN switches (performance and protocol correctness).

OSNT [19] and ATPG [31] are primarily used to construct and generate traffic for the data plane. Though we share similar goals on the data plane, our proposed CFT provides additional support for the test of the control plane.

Spirent [32], BROADCOM [33] and IXIA [23] are commercial testers for the tests of SDN switches regarding the performance and the protocol correctness. However, because these testers are commercially available, there are deficiencies in scalability, flexibility and follow-up to new protocols and features. At the same time, its high price makes most research institutions have difficulty in affording it, which is not conducive to the development and promotion of SDN switches [9,10,19].

References [34,35] are used to detect the consistency of updates in SDN networks. OFTEN [16], which extends NICE by enabling communication between the model checker and real switches, is a tool for systematic testing of the behaviors of integrated OpenFlow networks. perfbench [15] is a comprehensive software tool for testing SDN controllers, SDN switches and integrated SDN networks, which shares the same goal with OFTEN. We will extend our CFT to support the verification for SDN controllers and integrated SDN networks in the future.

7. Conclusions and Future Work

Our paper proposes the concept of CFT, which is an open CPU and FPGA co-design Tester for SDN switches. The CFT takes full advantage of high performance and accuracy of the hardware and flexibility and scalability of the software to enable the fast and accurate verification of SDN switches regarding the performance and protocol correctness. The CFT effectively solves the problem of clock asynchronization between the control plane and the data plane by merging them into the same hardware pipeline we implemented. The CFT reconfigurable pipeline we implemented is composed of hardware modules which support different testing functions such as traffic generation and traffic monitoring. The implementation of the *bypass* mechanism, based on the MetaData (included in CFT-Pkt), allows multiple functions to be executed in parallel in our proposed pipeline. Above the hardware, the CFT provides four types of APIs which are responsible for operations on packets and registers. By using APIs we provided, users are able to develop their test cases, such as the verification of the performance or the detection of the southbound protocol correctness. Prototype implementation and evaluation reveal that the CFT can not only test the performance and accuracy of the SDN switch on the data plane but also verify the protocol correctness on the control plane. We truly believe that the CFT is an excellent tester for SDN switches, which significantly promotes the development of SDN.

In future work, there are several research directions we intend to pursue to improve the CFT:

- Abstracting a domain-specific language (DSL) for testing based on the CFT software APIs;
- Extending the testing range of the CFT to the test and verification of SDN controllers and integrated SDN networks.

Author Contributions: Conceptualization, Y.J. and H.C.; Methodology, Y.J., H.C. and X.Y.; Software and Hardware, Y.J. and X.Y.; Validation, Y.J., X.Y. and W.Q.; Formal analysis, Y.J. and Z.S.; Investigation, H.C. and W.Q.; Writing—original draft preparation, Y.J.; Writing—review and editing, H.C.; Funding acquisition, W.Q. and Z.S.

Funding: This research was funded by the National Natural Science Foundation of China under grant NO. 61802417, NO. 61702538 and partially by the Scientific Research Program of National University of Defense Technology under grant NO. ZK18-03-40, NO. ZK17-03-53.

Acknowledgments: I would like to personally thank Junshuai Li and Jingtao Peng, engineers of the National University of Defense Technology, who support the deployment and implementation of the experiment. Thanks to Gianni Antichi, from Queen Mary University of London, for his guidance and help on the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Open Flow Switch Specification 1.5.1. 2019. Available online: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf (accessed on 21 July 2019).
2. Munjal, A.; Singh, Y.N. An improved autoconfiguration protocol variation by improvising MANETconf. In Proceedings of the 2014 IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS), New Delhi, India, 14–17 December 2004; IEEE: Piscataway, NJ, USA, 2014; pp. 1–3.
3. Kim, H.; Feamster, N. Improving network management with software defined networking. *IEEE Commun. Mag.* **2013**, *51*, 114–119. [CrossRef]
4. Caria, M.; Jukan, A.; Hoffmann, M. A performance study of network migration to SDN-enabled traffic engineering. In Proceedings of the 2013 IEEE Global Communications Conference (GLOBECOM), Atlanta, GA, USA, 9–13 December 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 1391–1396.
5. Davoli, L.; Veltri, L.; Ventre, P.L.; Siracusano, G.; Salsano, S. Traffic engineering with segment routing: SDN-based architectural design and open source implementation. In Proceedings of the 2015 Fourth European Workshop on Software Defined Networks, Bilbao, Spain, 2 November 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 111–112.
6. OpenDayLight Organization. 2019. Available online: https://www.opendaylight.org/ (accessed on 21 July 2019).
7. Project Floodlight. 2019. Available online: http://www.projectfloodlight.org/ (accessed on 21 July 2019).
8. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Com. Commun. Rev.* **2008**, *38*, 69–74. [CrossRef]
9. Rotsos, C.; Sarrar, N.; Uhlig, S.; Sherwood, R.; Moore, A.W. OFLOPS: An open framework for OpenFlow switch evaluation. In Proceedings of the International Conference on Passive and Active Network Measurement, Vienna, Austria, 12–14 March 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 85–95.
10. Rotsos, C.; Antichi, G.; Bruyere, M.; Owezarski, P.; Moore, A.W. OFLOPS-Turbo: Testing the next-generation OpenFlow switch. In Proceedings of the 2015 IEEE International Conference on Communications (ICC), London, UK, 8–12 June 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 5571–5576.
11. Canini, M.; Venzano, D.; Perešíni, P.; Kostić, D.; Rexford, J. A NICE way to test OpenFlow applications. In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), San Jose, CA, USA, 25–27 April 2012; pp. 127–140.
12. Kuzniar, M.; Peresini, P.; Canini, M.; Venzano, D.; Kostic, D. A SOFT way for openflow switch interoperability testing. In Proceedings of the ACM CoNEXT 2012 the 8th International Conference on Emerging Networking Experiments and Technologies, Nice, France, 10–13 December 2012; pp. 265–276.
13. Kuzniar, M.; Peresini, P.; Kostic, D. *What you Need to Know About SDN Control and Data Planes*; Technical Report; EPFL: Lausanne, Switzerland, 2014.
14. Handigol, N.; Heller, B.; Jeyakumar, V.; Maziéres, D.; McKeown, N. Where is the debugger for my software-defined network? In Proceedings of the First Workshop on Hot Topics in Software Defined Networks, Helsinki, Finland, 13 August 2012; ACM: New York, NY, USA, 2012; pp. 55–60.
15. Blenk, A.; Basta, A.; Henkel, L.; Zerwas, J.; Kellerer, W.; Schmid, S. perfbench: A tool for predictability analysis in multi-tenant software-defined networks. In Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos, Budapest, Hungary, 20–25 August 2018; ACM: New York, NY, USA, 2018; pp. 66–68.
16. Kuzniar, M.; Canini, M.; Kostic, D. OFTEN testing OpenFlow networks. In Proceedings of the 1st European Workshop on Software Defined Networks (EWSDN 2012), Darmstadt, Germany, 25–26 October 2012.
17. Monsanto, C.; Reich, J.; Foster, N.; Rexford, J.; Walker, D. Composing software defined networks. In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), Lombard, IL, USA, 2–5 April 2013; USENIX: Berkeley, CA, USA, 2013; pp. 1–13.
18. OFtest. 2019. Available online: http://www.projectfloodlight.org/oftest/ (accessed on 23 July 2019).
19. Antichi, G.; Shahbaz, M.; Geng, Y.; Zilberman, N.; Covington, A.; Bruyere, M.; McKeown, N.; Feamster, N.; Felderman, B.; Blott, M.; et al. OSNT: Open source network tester. *IEEE Netw.* **2014**, *28*, 6–12. [CrossRef]
20. Gu, X. Hardware-Based Network Monitoring with Programmable Switch. Senior Thesis, University of Illinois, Urbana, IL, USA, May 2019.

21. Li, J.; Yang, X.; Sun, Z. DrawerPipe: A reconfigurable packet processing pipeline for FPGA. *J. Comput. Res. Dev.* **2018**, *55*, 717–728.
22. Xilinx Zynq 7000 SoC. 2019. Available online: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (accessed on 18 June 2019).
23. Ixia Makes Networks Stronger. 2019. Available online: https://www.ixiacom.com/ (accessed on 23 June 2019).
24. Pica8. 2019. Available online: https://www.pica8.com/product/ (accessed on 23 June 2019).
25. iPerf—The ultimate speed test tool for TCP, UDP and SCTP. 2019. Available online: https://iperf.fr/ (accessed on 15 July 2019).
26. Yang, X.; Sun, Z.; Li, J.; Yan, J.; Li, T.; Quan, W.; Xu, D.; Antichi, G. FAST: Enabling fast software/hardware prototype for network experimentation. In Proceedings of the International Symposium on Quality of Service, Phoenix, AZ, USA, 24–25 June 2019; ACM: New York, NY, USA, 2019; Article No. 32.
27. NetFPGA. 2019. Available online: https://netfpga.org/ (accessed on 10 June 2019).
28. OFSuite_Performance. 2019. Available online: https://www.sdnctc.com/index.php/test_identify/test_identify/id/47 (accessed on 23 July 2019).
29. Jarschel, M.; Metter, C.; Zinner, T.; Gebert, S.; Tran-Gia, P. OFCProbe: A platform-independent tool for OpenFlow controller analysis. In Proceedings of the 2014 IEEE Fifth International Conference on Communications and Electronics (ICCE), Danang, Vietnam, 30 July–1 August 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 182–187.
30. Emmerich, P.; Pudelko, M.; Gallenmüller, S.; Carle, G. FlowScope: Efficient packet capture and storage in 100 Gbit/s networks. In Proceedings of the 2017 IFIP Networking Conference (IFIP Networking) and Workshops, Stockholm, Sweden, 12–16 June 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 1–9.
31. Zeng, H.; Kazemian, P.; Varghese, G.; McKeown, N. Automatic test packet generation. In Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, Nice, France, 10–13 December 2012; ACM: New York, NY, USA, 2012; pp. 241–252.
32. Spirent—Network, Devices & Services Testing. 2019. Available online: https://www.broadcom.com/ (accessed on 23 June 2019).
33. BROADCOM. 2019. Available online: https://www.broadcom.com/ (accessed on 23 June 2019).
34. Foerster, K.T.; Schmid, S.; Vissicchio, S. Survey of consistent software-defined network updates. *IEEE Commun. Surv. Tutor.* **2018**, *21*, 1435–1461. [CrossRef]
35. Zhou, W.; Jin, D.; Croft, J.; Caesar, M.; Godfrey, P.B. Enforcing customizable consistency properties in software-defined networks. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA, USA, 4–6 May 2015; pp. 73–85.