FAST: Enabling Fast Software/Hardware Prototype for Network Experimentation

Xiangrui Yang
yangxiangrui11@nudt.edu.cn
National University of Defense Technology

Zhigang Sun
sunzhigang@nudt.edu.cn
National University of Defense Technology

Junnan Li
lijunnan@nudt.edu.cn
National University of Defense Technology

Jinli Yan
yanjinli@nudt.edu.cn
National University of Defense Technology

Tao Li
taoli.nudt@gmail.com
National University of Defense Technology

Wei Quan
w.quan@nudt.edu.cn
National University of Defense Technology

Donglai Xu
XuDongLai0923@163.com
Hunan Xperis Network

Gianni Antichi
g.antichi@qmul.ac.uk
Queen Mary University of London

ABSTRACT

The evolution of new technologies in network community is getting ever faster. Yet it remains the case that prototyping those novel mechanisms on a real-world system (i.e. CPU-FPGA platforms) is both time and labor consuming, which has a serious impact on the research timeliness. In order to bring researchers out of trivial process in prototype development, this paper proposed FAST, a software hardware co-design framework for fast network prototyping. With the programming abstraction of FAST, researchers are able to prototype (using C, verilog or both) a wide spectrum of network boxes rapidly based on all kinds of CPU-FPGA platforms. FAST framework takes care of managing DMA, PCIe and Linux Kernel while providing a unified API for researchers so they can focus only on the packet processing functions. We demonstrate FAST framework's easy to use features with a number of prototypes and show we can get over 10x gains in performance or 1000x better accuracy in clock synchronization compared with their software versions.

CCS CONCEPTS

• **Networks** → **Network design principles**; **Network experimentation**; *Programmable networks.*

KEYWORDS

FPGA, Software Hardware Co-design, Network Prototype

ACM Reference Format:
Xiangrui Yang, Zhigang Sun, Junnan Li, Jinli Yan, Tao Li, Wei Quan, Donglai Xu, and Gianni Antichi. 2019. FAST: Enabling Fast Software/Hardware Prototype for Network Experimentation. In *IEEE/ACM International Symposium on Quality of Service (IWQoS '19), June 24–25, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3326285.3329067

1 INTRODUCTION

The emerging trend of software hardware co-design to satisfy both flexibility and performance requirements has been extensively explored in network community, especially in datacenter networks [6, 7, 9, 13, 26]. For instance, host software defined network (host SDN) [6] and SmartNICs [7] have greatly boosted better solutions for flexible, scalable and high performance network services in Azure [6] and other cloud datacenters. Before deployed in large-scale networks, each network function, as well as integrated system, requires several rounds of solid and comprehensive real-world validation in order to find the perfect partition between software and hardware.

However, building prototypes using CPU-FPGA platforms can be extremely complex and time consuming, considering myriad details both in hardware (gmii-rgmii conversion, DMA engine, PCIe, etc.) and software (Linux Kernel, I/O driver, system calls, etc.), as has been stressed by many literatures [6, 13, 18, 28]. Well-known frameworks or platforms, such as NetFPGA [18], DPDK [27] and P4 runtime [2], provide useful features for rapid software or hardware network prototyping, but lack overall consideration of software and hardware co-design, which we argue is the key point for fast prototyping in the era of SmartNIC. But as far as we concern, there is little state-of-art frameworks that provide such convenience in network community.

Thus, we present the design and implementation of FAST[1], a programming framework to enable fast network prototyping on various CPU-FPGA platforms. The FAST framework is based on three principles. First, while prototyping a network box, the software and hardware module should be treated equally. In this way, users

[1]FAST is short for Fpga-Accelerated Switching platTform.

could rapidly abstract the prototype into a module-graph [3], and map the modules onto software and hardware separately according to specific requirements. Second, the framework should hide all the platform details for users as long as they need not be modified. Finally, the framework should provide users with flexible and user-friendly APIs for network processing. In this way, researchers can focus on packet processing functions without distractions.

FAST can satisfy a wide spectrum of network prototyping. First, FAST provides well-defined programming spaces both on CPU and FPGA, which are User Application (UA) space User Module (UM) space, as well as programming specifications, for users to describe network functions with both C and verilog code. Outside user's scope, FAST tackles platform details, so users can develop their network prototypes on different CPU-FPGA platforms using the same code without worrying about compatibility. Second, FAST exposes a set of unified and efficient APIs for UAs and UMs to exchange packets, register values and valuable metadata with each other. We hide DMA, Linux Kernel and other details on the path between CPU and FPGA, so that users could define the data and control flow across the platform based on their own needs. Third, FAST also enables highly reusable software and hardware modules for agile network prototyping. Specifically, a five-stage pipeline is provided as default settings in FPGA, which include a parser, a key extractor, a table matcher, an action engine and an output engine. They can either serve the users as samples or be reused as building blocks for agile prototyping.

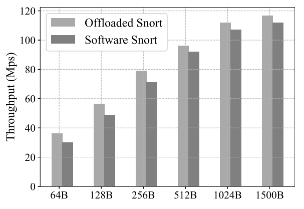
This paper makes the following contributions:

- We introduce an unified programming framework for fast network prototyping on CPU-FPGA platforms and its programming abstractions.
- We describe the comprehensive details behind FAST on different types of platforms and the trade-off between hiding platform details and supporting comprehensive innovations.
- We show how the proposed framework simplifies a wide spectrum of novel network prototypes, by demonstrating a various of real-world cases we developed using FAST during these years.
- We open-source FAST project [8] on github to encourage both network researchers and platform providers to take the advantages of and optimize FAST during their experimental and productive work.

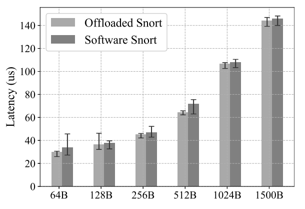
The rest of the paper is organized as follows. We explain our motivation and approach in Sec. 2. Sec. 3 describes the key abstraction of FAST framework. In Sec. 4, we demonstrate the design details behind FAST and some optimization technologies. In Sec. 5, we evaluate FAST by introducing a set of prototypes we developed with FAST. We discuss related works in Sec. 6 and conclude in Sec. 7.

2 MOTIVATION AND APPROACH

In this section, we demonstrate our motivation for a unified software hardware network co-design framework, and present our approaches to its development.



(a) Throughput comparison



(b) Latency comparison

Figure 1: The performance comparison between 5-tuple matching function offloaded Snort and software-only Snort. 90th% percentile and min is represented in latency using error bars.

2.1 Towards a Rapid and Unified Co-design Framework

FAST targets rapid network prototyping with CPU-FPGA platforms of any kind. The main reason for using CPU-FPGA co-design to implement network services is flexibility and performance needs, as demonstrated by existing works [6, 7, 13, 28]. In the era of high speed network (10/40/100GE), offloading network functions to FPGA also saves CPU cycles and memories for user applications. However, resources on a FPGA board (e.g., SmartNIC) are limited [16, 17]. A fast prototyping framework would give the chance for a quick assessment of different offloading options, so that we can make the best tradeoff between performance and cost, and reduce time-to-market simultaneously.

To stress it clear, taking the offloading of Snort [24] for example. The main function in Snort's Detection Engine consists of packet header process and payload process. In order to improve the performance, we offloaded 5-tuple matching function (packet header process for UDP) onto FPGAs and remain other functions on the software. To evaluate it, we ran offloaded Snort 2.02 and its software baseline both on a Xilinx Zynq7000 SoC board[2] [33] (Cortex A9 CPU of 866MHz and Artix-7 FPGA of 100MHz). We install a rule which consists of 5-tuple matching on header field and exact payload matching on L4 full payload. The same UDP packets, all of which will hit the installed rule, are transmitted to the Snort-listened port by a IXIA network tester [11] and we obtained 2000 groups of statistics. The average throughput (1GE line rate) and latency (drop rate = 0%) are demonstrated in Fig. 1. However, only less than 8% throughput improvement and about 3.5% latency reduction in average are obtained.

This demonstrates that offloading 5-tuple matching function is not an acceptable option considering the tradeoff between performance gains and additional cost on FPGA. To understand the result, we compare the increase in memory usage between both settings. The result is shown here:

	Idle Mode	**Busy Mode**
Offloaded version	10.402%	12.097%
Software version	10.386%	12.192%

According to the result, we find the poor performance gain is because 5-tuple matching function only costs very limited resources

[2]Because Snort drops packets in high-speed, to measure the exact performance, we modify Snort to forward packets to a specified port after PacketCallback() function.

Table 1: Comparison between existing solutions and FAST framework for fast network prototyping.

Solutions	Definition	Software or Hardware?	Programing Language	Bounded Platform	OpenSource?
FAST	Programming framework	Software & Hardware	C/C++ & Verilog	None	Yes
ClickNP [13]	Programming framework & language	Hardware (FPGA)	ClickNP	Intel FPGA	No
NetFPGA [18]	Hardware platform	Software & Hardware	C/C++ & Verilog	NetFPGA	Yes
netmap [23]	Developing APIs & libraries	Software	C/C++	None	Yes
P4 runtime [2]	Programming language	Hardware (ASIC & FPGA)	P4	Tofino ASIC	No
DPDK [27]	Developing APIs & libraries	Software	C/C++	None	Yes
SDNet [32]	Programming framework	Hardware (FPGA)	PX or P4	Xilinx SoC & FPGA	No
Emu [28]	Developing libraries	Hardware (FPGA)	.NET	NetFPGA	Yes

Table 2: Key notions in FAST

Notion	Description
UA (User Application)	A piece of software code processing packet or control FPGA logic.
UM (User Module)	A hardware module that user writes to process packet.
MID (Module ID)	A specific ID to identify a UA or UM.
VAS (Virtual Address Space)	A range of virtual address for UA to access registers in UM.
MD (Metadata)	A piece of data carried before every fast packet to record info during processing.
FAST APIs	A set of APIs for UA to send/recv/ modify packets or read/write registers.

on the software, and packets still need to be sent to CPU for processing (e.g., payload matching). This proved that software hardware co-design does not always bring great performance gains. Rounds of prototyping and deep analysis of different design options matters a lot. It emphasizes that a quick assess of different co-design options is extremely critical.

Previous works tried to address the challenge of fast network function prototyping, as demonstrated in Table 1. Those solutions either focus on providing libraries and APIs for software prototyping (e.g., netmap [23], DPDK [27]), or considering platforms and frameworks for hardware prototyping (e.g., ClickNP [13], P4 runtime [2], SDNet [32]). NetFPGA [18] provides a good direction but is hardware-bounded and does not dedicated to support co-design of different models. FAST manages both aspects and provides a unified network prototyping framework. Moreover, FAST is not bounded to any specific platforms thus provides a general solution for prototyping on any off-the-shelf CPU-FPGA combinations, especially on SmartNICs.

2.2 Requirements and Approach

By analyzing former works and the motivation of providing a programming framework for various CPU-FPGA platforms, we identify four key requirements for a software hardware co-design framework.

R1: The framework must hide platform-related engineering details (gmii-rgmii conversion, DMA engine, PCIe spec, I/O Drivers, etc.) behind standard interfaces and data structures.

R2: When program with FAST, users are able to have a unified view of all the functional modules in a specific network box no matter the module is on the software (written in C) or on the hardware (written in verilog).

R3: The standard data structures in the framework should deliver rich packet processing infos as well as user defined flexibility.

R4: Code modules written by developers must be highly-reusable to enable agile prototyping of others.

Our main goal is to provide a set of well-defined software and hardware specifications, and to design a unified programming backend on various CPU-FPGA platforms as the support. FAST consists of several components to meet these requirements.

Modular Programming Space: FAST provides modular programming features by abstracting packet processing functions as modules. Each module, no matter on the software or hardware, has an unique ID. Users can define how packets travel along all the software and hardware modules in FAST based on the specific ID.

Metadata & Packet-Header-Vector are two key data structures in FAST to provide useful information during the packet lifetime in CPU-FPGA platforms. They are also the only necessary data flows between each hardware module on the FPGA part.

Highly Flexible FAST Stack: FAST stack allows packets from FPGA to bypass TCP/IP stack and to process them using user-defined actions with minimal modifications in the kernel space, or in the user space, to enable protocol innovations.

FAST APIs are designed to provide flexible packet processing on CPU. On the one hand, it allows packet exchanging between hardware and software while bypassing TCP/IP stack, and exposes rich information in the metadata along with the packet. On the other hand, it supports direct control of FPGA logic by reading and writing hardware registers from CPU.

3 FRAMEWORK AND PROGRAMMING ABSTRACTION

In this section, we provide the design and usage of FAST framework and explain how it simplifies network prototyping based on CPU-FPGA platforms. First, we give the notions we use in FAST so as to make the description clearer.

Table 2 gives a brief description of key notions in FAST framework. In general, a **UA** is a piece of software program that users write for packet processing on CPU side, or as a control plane for FPGA logics. On the other hand, a **UM** (written in verilog) lies on a specific space of FPGA, which we call UM Space, as a hardware processing module. UA and UM are the key abstractions FAST provided on the CPU and FPGA to simplify both software and hardware programming. Each UA or UM has a global unique ID, which we call **Module ID** (MID). Besides, we are also inspired by memory management mechanism on CPU, and propose the Virtual Address

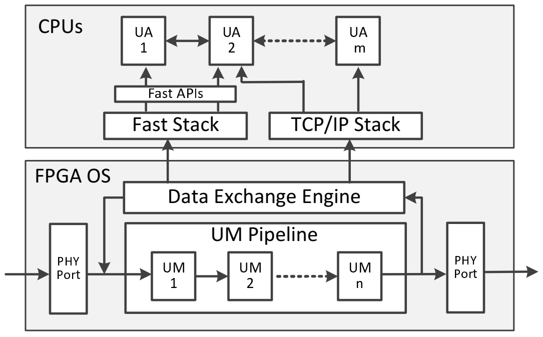


Figure 2: FAST Framework.

Space (VAS) for flexible FPGA register read and write operations on software.

3.1 FAST Framework

Given the key notions, now we introduce the proposed network processing framework. As shown in Fig. 2, FAST defines two spaces for users to program a network box, UA and UM space. On the software, UAs can play either the role of data plane or control plane by processing packets or read and write registers via FAST APIs. On the hardware, UM pipeline is designed for users to write hardware functions for high-performance packet processing on FPGA. In the middle, FAST provides a number of mechanisms to simplify packet processing and data exchanging.

When traffic is received from a physical port on the FPGA board, it enters the **FPGA OS**. We term FPGA OS as hardware logics that deals with standard and platform-related PHY and MAC layer logics, which includes but not limited to gmii-rgmii conversion, checksum calculation, frame interval recognition and metadata generation. After the pre-processing, a MAC-layer frame[3] will enter the UM pipeline, with a metadata attached to it. The metadata keeps recording timestamps, protocol type, packet length, and other useful infos. UMs in the pipeline can modify, drop, schedule or send the packet up to the CPU. All these actions are written to a specific field of metadata according to the FAST specification we defined so that they will be executed correctly.

Above the UM pipeline is the **Data Exchange Engine** (DEE). DEE in real-world systems could be a DMA engine or other control unit that is used to exchange data between software and hardware. In FAST, DEE is a key module for packets to travel between UMs and UAs, and enable register read and write operations. If a packet is sent to UAs, After processed by DEE, the I/O driver will determine whether the packet should be delivered to FAST Stack or TCP/IP Stack based on metadata. If the packet is delivered to the FAST Stack, the metadata keeps attached to the packet to provide processing info for UAs. Otherwise, metadata will be eliminated from the packet, which will latter be processed by standard TCP/IP stack.

Over the stack, FAST provides a set of APIs for UAs to send or receive packets. Moreover, as a software hardware co-design framework, FAST APIs also give UAs the ability to read & write hardware registers. In this case, UA can serve the network box as the control plane. The register read & write operations can further

[3]For consistency, we use the term packet in the following sections.

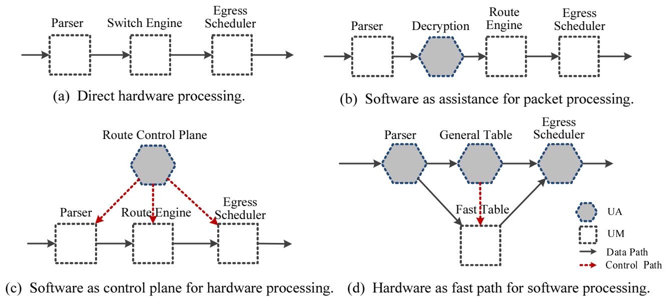


Figure 3: Software hardware co-design models FAST support.

be encapsulated as rule configuration APIs for UA. Examples of FAST APIs are shown:

```
//ua register to the FAST stack
static int fast_ua_register(int mid);
//for sending packets to UMs
int fast_ua_send(struct fast_packet *pkt,int pkt_len);
//for read hardware registers from UA
u32 fast_ua_hw_rd(u8 dmid,u32 addr,u32 mask);
//for write hardware registers from UA
void fast_ua_hw_wr(u8 dmid,u32 addr,u32 value);
//provide similar functions like recv() in socket.
void fast_ua_recv();
```

3.2 Packet Processing Model

Packet processing is often abstracted as a processing graph, as has been widely discussed [2, 3]. FAST also leverages the graph design as the abstraction basis. However, in order to provide comprehensive supports for network prototyping, we summarize co-design models of network processing to four specific types. The simplest model is hardware solely processing (Fig. 3(a)). This is common in prototypes with high performance requirements and simple control logic. Sometimes the software needs to be inserted into data plane (Fig. 3(b)) providing functions that are hard to be implemented on hardware (e.g., encryption and decryption). Another typical model is setting software as the control plane of hardware pipelines, which is not only true in prototypes but also off-the-shelf devices such as routers (shown in Fig. 3(c)). In the era of NFV, hardwares, especially FPGAs, sometimes are required as fast paths or caches in order to deliver high performance for NFs that are implemented on software (Fig. 3(d)).

In this part, we demonstrate key mechanisms in packet processing path and data structure along the path.

3.2.1 *Packet Processing Path.*

Among the four models above, the most challenging problem lies in allowing packet traveling cross the software and hardware regardless of orders. As this is a key step to enable modules mapping flexibly between software and hardware. In order to solve it, we proposed a novel MID Index Mechanism. First of all, as both UA and UM have an unique mid, we need a way to distinguish the two cases. In FAST, we allocate mid of UM in a bottom-up way, while allocating mid of UA in a top-down way. Specifically, assume that mid is n bits long, the mid of UM can be allocated from 0 to $2^{n-1} - 1$. And the mid of UA is allocated from $2^{n-1} + 1$ to $2^n - 1$. We reserve 2^{n-1} for special use that will be discussed in Sec. 4.

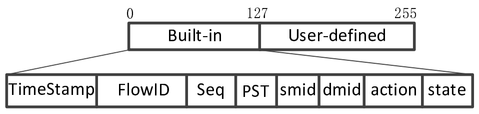


Figure 4: Metadata format and fields in built-in segment.

Once the packet enters UM pipeline, the first built-in UM would parse the packet and tag the tuple {**current mid, next mid**} into the metadata. When the packet is received by each module, the module will check if the **next mid** contained in the metadata matches with the mid of itself. If so, the module updates the {**current mid, next mid**} field, processes the packet and sends it to next module. Or the module will be by bypassed alternatively. If the packet destination module is a UA (on the software), the last built-in UM on the hardware will modify the action field in metadata to notify the DEE that the upcoming packet should be sent to a specific UA. Then the packet will travel across DEE and FAST Stack, and finally be received by the specific UA. After the UA processing is finished, the UA can choose to send the packet back to UM pipeline, or pass it to another UA on the software.

A specific packet processing path along the graph can be based on a particular protocol, five-tuples or other values as long as they are defined in metadata. By this simple way, FAST is able to map the modules inside a processing graph across software and hardware flexibly.

3.2.2 *Key Data Structures.*

From above, we show that metadata is a core data structure assisting packet processing across all FAST modules. Besides metadata, we also defined **Packet Header Vector** (PHV) and **Fast Control Packet** (FCP) for user's convenience in FAST. In this part, we give more details about these key data structures.

Metadata is carried by each packet that enters FAST pipeline. Basically, metadata is a 32-bytes data (Fig. 4) including 16 bytes built-in segment and 16 bytes user-defined segment. The built-in segment contains many useful infos, such as timestamps, protocol type (the PST field), flow id, next mid, etc., that enable multi-functional packet processing. Moreover, the actions that FAST supports (header modification, packet drop, multicast, etc.) are also tagged in metadata for egress engine to execute accordingly. In case that users may have additional requirements, we reserve the 16 bytes user-defined segment for user definition. We demonstrate how this benefits lots of network function developments in Sec. V.

PHV is another useful data structure we used in FAST. Usually processing the whole packet in each module inside a pipeline is unnecessary and of high-latency. The better method is to store the packet payload in a data cache and only process the packet header in those modules. Then the assembler (located in the last stage of pipeline) assembles the packets and execute the action according to metadata. FAST strongly encourages users to apply this idea in their UM pipeline, as this also greatly improves the module re-usability.

As a framework, FAST is not bound to any specific protocol groups. Thus, we define the first 1024 bit of each packet as the PHV to cover as many layers as possible. Users may generate PHV in their first UM or simply use the built-in parser (we wrote as a sample UM) to generate it. PHV should travels in the UM pipeline simultaneously with metadata and can be modified by any UM that

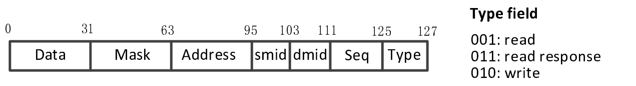


Figure 5: Format of FAST Control Packet (FCP).

user writes. When the packet is assembled, PHV will update the first 1024 bits of the packet accordingly.

FCP is a piece of special metadata sent from a UA to the UM pipeline for generic register read and write operations. The packet format is shown in Fig. 5. Each FCP is 128 bits and can carry a 32-bit data. The highest bit indicates whether the packet is a FCP or a data packet. Then the 126 to 124 bit indicates the specified type (read, read response or write). And be noted that the 95 to 64 bit is intended for a 32-bit virtual address (We will discuss in next part) for locating the specific 32-bit register. The data is carried in the lowest 32 bits of the packet. With FCP, we are able to access hardware register values using a unified method on different CPU-FPGA platforms.

3.2.3 *Virtual Address Space.*

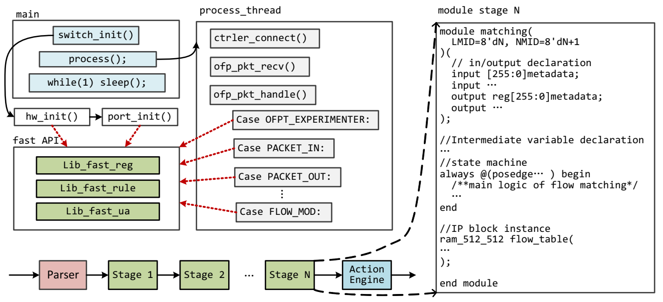
Sometimes UA needs to serve the network box as the control plane. This means UA need to keep track of and modify hardware states, which is usually achieved by reading or writing hardware registers. Virtual Address Space (VAS) is proposed to support unified register access from software on various CPUs-FPGA platforms.

The main trade-off here is how to make a specific register easy to locate, while assuring enough addressing space for all the UMs. We solve this by introducing a two-layer addressing mechanism. First, As each UM has a unique mid attached to it, we use the field **DMID** in FCP to locate the specific UM. Then, we provide the users with a 32-bit (4GB) space for addressing a 32-bit register in a UM. So that users can assign the 4GB within their UM flexibly. What the users need to do is writing a self-defined decoder (using case statement in verilog) in the UM that decodes the address of each register. When a UA sends a read or write request, the FCP will travel along the UM pipeline. And only the correct response UM will decode the address in FCP, and attach the correct register value to the FCP or write the value carried by FCP to the register.

3.3 Modular Programming

FAST is designed to prototype a large spectrum of network services and functions, ranging from generic packet processing box (white-box switch, IDS, load balancer, etc.) to specialized, customized acceleration card (SmartNIC, encryption card, etc.). We show how to build a white-box switch (OpenFlow version) using FAST framework. According to [20], the pipeline of the switch can be abstracted as a set of **flow table** followed by an **execution engine** to execute the actions recorded in the **action set**. Every packet header (and attached metadata) is generated once enters the pipeline and travels through a flow table accordingly. Then the matched actions is tagged in the action set field and finally executed before egressed from the pipeline. In addition, the switch also contains an software agent (switch OS) for interaction with the control plane and the management of hardware state [4].

Using FAST to program a white-box switch is quite straightforward as shown in Fig. 6, Each component we discussed above can



```
module matching(
  LMID=8'dN, NMID=8'dN+1
)(
  // in/output declaration
  input [255:0]metadata;
  input ⋯
  output reg[255:0]metadata;
  output ⋯
);

//Intermediate variable declaration
…
//state machine
always @(posedge⋯ ) begin
  /**main logic of flow matching*/
  …
end

//IP block instance
ram_512_512 flow_table(
…
);

end module
```

Figure 6: The programming of a white-box switch using FAST framework.

be perfectly mapped on to FAST programming model. On the hardware, the pipeline can be directly mapped into UM pipeline. First, we use a built-in UM to parse and generate metadata, PHV and matching key from an incoming packet. From the second module, extracted matching keys are used to matching a flow entry. All the matched actions are stored in metadata, which will be executed in the last module. As FAST defines the input and output data structures of each module, writing a hardware module becomes quite simple. Fig. 6 also gives a brief template for writing a hardware module in FAST.

On the software, the switch OS is implemented as a UA. It can leverage socket or any communication tool kit to interact with the controller. Once received message from the control plane, it calls FAST APIs to read/write/configure hardware accordingly. Besides, the switch OS also listens hardware event (using FAST API). E.g., packet received from pipeline will trigger `packet_in` event, then the switch OS will forward the packet to the controller via southbound interface (i.e. OpenFlow).

The source code of a white-box switch (based on OPF Spec 1.5) based on FAST framework can be found at [8]. And we have added the necessary functions of switch OS in the FAST libraries to support iterative development for other developers. As FAST deals with platform relate details, we only wrote the functions which is simple and straightforward. Additionally, a prototype written in FAST can be migrated to other CPU-FPGA platforms seamlessly, which greatly reduces the workload of reproducting a network box.

4 FAST DESIGN AND IMPLEMENTATION

We have implemented FAST framework on a several off-the-shelf platforms and there are also a wide range of prototypes has been written based on FAST [14, 34, 36]. In this section, we discuss the FAST internals on different CPU-FPGA platforms and some improvements to optimize typical prototyping.

4.1 Multi-platform Implementation

The interconnection between CPU and FPGA is multiple and can influence the design and implementation of FAST. Here we stress two most typical interconnections and how FAST is designed and implemented accordingly.

4.1.1 PCIe as Interconnection. PCIe is a common choice for the interconnection between CPU and FPGA. The two major FPGA

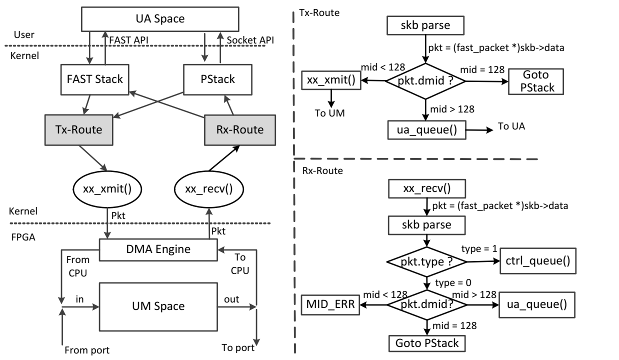


Figure 7: FAST implementation on PCIe interconnection platforms.

vendors both provide a wide spectrum of SoCs that combine multi-core CPU with FPGA to deliver easy-to-program features and high performance. Examples are Xilinx ZYNQ series [33] and Intel ARRIA series [10]. Besides, most pure FPGA boards have reserved PCIe interfaces that can be plugged into a multi-core server.

On such platforms, we use DMA as the default I/O engine between CPU and FPGA. And FAST Stack is implemented bypassing the standard TCP/IP stack in the kernel space. The overall implementation is shown in Fig. 7(a). First, the DMA engine is implemented on FPGA to support efficient packet I/O. Then, in the kernel space, we modify two core functions of packet I/O driver, which are `xx_xmit()` and `xx_recv()`, to process the packet received from or sent to FPGA board. Above them, we add two key modules allowing packets that based on FAST format (including data packet and FCP) bypassing TCP/IP stack, named Tx-Route and Rx-Route.

Tx-Route: As demonstrated in Fig. 7(b), when data is received from UA space, the Tx-Route will assign a socket buffer (skb) to store it. Then the skb payload is casted using the `fast_packet` data structure. In this step, the dmid of the packet is parsed as a key to find the destination. If dmid is less than 128, which indicates the packet should be sent to a UM whose mid equals to the parsed dmid, Tx-Route will transfer the packet to FPGA using `xx_xmit()`. If dmid is larger than 128, which indicates the packet should be sent to another UA, Tx-Route will push the packet to a specific UA using NetLink. Otherwise (dmid=128), the packet will be sent to TCP/IP Stack and processed accordingly.

Rx-Route: The processing flow (Fig. 7(c)) in Rx-Route is slightly different from Tx-Route. Once the `xx_recv()` obtained the packet sent from DMA engine, it will be processed by Rx-Route. After the skb data is casted using `fast_packet`, firstly Rx-Route determines whether the packet is a data packet or a FCP. If it is a FCP, then it will be pushed in a queue with higher priority (using `ctrl_queue()`). This is because in most network boxes, read and write hardware registers often closely relates to the correctness of the system. If it is a data packet, it is processed with the same way as in Tx-Route. Be noted that the DMA engine should not send a packet with dmid less than 128 to CPU. So if a parsed dmid is less than 128, a `MID_ERR` will be thrown by Rx-Route.

4.1.2 Ethernet as Interconnection. Other than PCIe, FAST also supports ethernet as the interconnection between CPU and FPGA. This is useful when CPU and FPGA are located at different places.

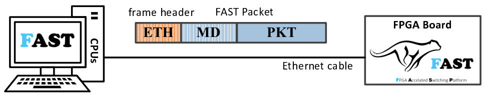


Figure 8: Ethernet interconnection for FAST implementation.

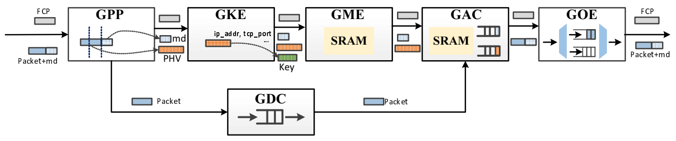


Figure 9: FAST built-in five-stage pipeline in UM space. Users can insert self-written UMs or rewrite some of them to prototype their own network box with less workload.

In such scenarios, the FPGA board is mostly connected to a remote server using an ethernet cable. So the data exchanging between CPU and FPGA must be based on ethernet. As demonstrated in Fig. 8, in order to exchange data between CPU and FPGA in FAST format, the `fast_packet` which is used to communicate between UA and UM is encapsulated in a link layer frame to travel on the ethernet cable. And we use a specific MAC address[4] to tag the frame that encapsulates a fast packet. Thus, the NIC should be set as promiscuous mode in order to receive packets of different MAC address. In the kernel space, we use the same packet processing logic as in PCIe interconnection mode, except that the `xx_xmit()` and `xx_recv()` need to encapsulate the packet additionally if it is in FAST format. Be noted that while using ethernet as interconnection, the TCP Segment Offloading (TSO) of NIC should be disabled as the packet may bypass the TCP/IP Stack.

However, we find that compared with PCIe as interconnection, prototypes runs on such platforms may suffer from higher latency. Thus, when building network prototypes on platforms using ethernet as interconnection, it is more appropriate to program UA as the control plane (modify or obtain registers), while leaving the high-performance data plane packet processing on the UM pipeline.

4.2 Scalable Built-in UM Pipeline

FAST has been used by many researchers for quite a time. During their developments, we find that some particular modules are often used by most researchers. In order to further simplify the development on FAST, we provide a five-stage built-in pipeline in the UM space, which can serve as not only building blocks to develop novel prototypes but also samples for new comers to learn how to process packet on FPGA.

As shown in Fig. 9, the five-stage pipeline include six UMs, which are a parser (GPP), a key generator (GKE), a matching engine (GME), an action composer (GAC), an output engine (GOE) and a data cache (GDC). Once the packet enters the pipeline, it is firstly processed by GPP, GPP parses the 256 bit metadata and the 1024 bits PHV from the head of the packet and deliver them to GKE. Besides, the whole packet is stored in FIFOs (GDC). Then the GKE leverages the PHV field to generate a key which will latter be used to match the rules in the matching table. The key in GKE is 512 bits long, and includes protocol values (mac addresses, ip addresses, tcp ports, etc.) from MAC to transport layer. Users have the access to modify it in case user-defined keys are needed.

Followed by is GME, GME is an matching engine with a 512 bit wide lookup table. GME takes the key from GKE and matches it in the lookup table. The matching result is a 16 bits long action index, which is tagged into metadata and is latter used to obtain the corresponding action accordingly in GAC. In GAC, firstly the matched action will be parsed from metadata. Then GAC reads the whole packet from GDC and use the modified metadata and PHV to overwrite the original packet. Finally GAC transfers the updated packet, metadata and PHV to GOE, during which time FIFOs are used to keep the three types of data synchronized. GOE is the last UM in the pipeline. This module is used to modify or schedule packets according to the action written in metadata. For instance, if a packet is indicated to be dropped, GOE would not transfer the packet out of UM space.

Besides, the FCP travels though all the modules in the pipeline except GDC. The input and output of FCP in each module are connected together. If users want to add specific registers inside these built-in module and support read/write operations from UA, they need to allocate the VAS and decode the FCP accordingly.

We find providing the five-stage pipeline to users, one can easily write his own prototypes. [36] and [35] are two typical open-source prototypes based on the FAST built-in pipeline.

Table 3: Platforms that support FAST framework.

Type	FPGA	CPU	Interconnection	Rate
NetMagic 08	ArriaII GX	Null	Ethernet	1 GE
OpenBox S4	Artix-7	Cortex A9	PCIe	1 GE
OpenBox S28	Virtex-7	Cortex A9	PCIe	10 GE
OpenBox S56	ArriaV GT	Intel Core i7	PCIe	40 GE

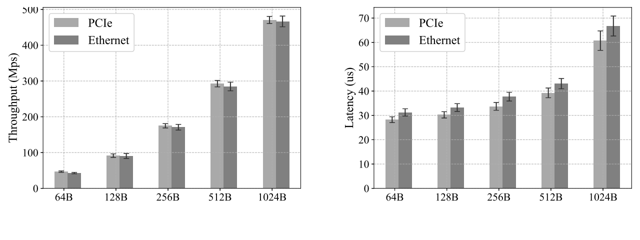
5 EVALUATION

In this section, we evaluate the performance of FAST framework on several platforms we built (as shown in Table 3) and how FAST benefits a wide spectrum of network prototyping using software and hardware co-design. First we explore the performance of FAST based on different interconnections between CPU and FPGA. Then, we demonstrate several network prototypes we developed based on FAST regarding developing simplicity, performance and functionality.

5.1 Throughput and Latency

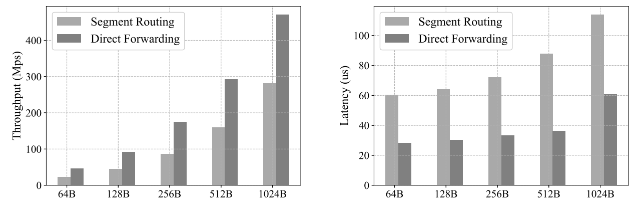
Our first experiment compares the throughput and latency between PCIe and ethernet interconnections. We use OpenBox S4 and NetMagic 08 [15] to conduct the experiment. Too keep irrelevant variables consistent, we leverage ARM developing board (ARM Cortex-A9, 866MHz) as the remote CPU for NetMagic 08. On both platforms, we use the built-in UM pipeline on FPGA, and write a UA to allow packet received from FPGA being sent to port 1 immediately. Then, we wrote a default rule on GME to enable all packets received from port 0 be pushed to the specific UA we mentioned above. Finally, The IXIA PerfectStorm network tester [11] is utilized to send packets to port 0 of both OpenBox S4 and NetMagic 08 and

[4]We use the MAC address of {0xee,0xee,0xee,0xee,0xee,0xee} in the current version. However, it can be modified as the address is a configurable parameter in FAST.



(a) Throughput (b) Latency

Figure 10: The performance comparison between PCIe interconnected platforms and Ethernet interconnected platforms. Error bars denote standard deviations.



(a) Throughput (b) Latency

Figure 11: The performance comparison between segment routing and direct software forwarding.

receive packets from port 1 for performance evaluation. During the experiment, we change the packet rate and size of the network tester gradually to obtain the bottleneck throughput between CPU and FPGA[5].

Both platforms achieved similar throughput according to packet size (shown in Fig. 10(a)). Moreover, the throughput almost doubles when we double the packet size, this indicates a roughly constant packet rate (pps). This is an expecting result because the skbuff space that linux kernel allocated for each packet is around 1800 bytes, which is larger than the size of default MTU. As the software only deals with metadata and packet header in this setting (no extra modifications of payload), the time cost for each packet remains similar.

On the other hand, ethernet interconnection suffers from slightly longer latency (shown in Fig. 10(b)) because of the additional transmission delay on NIC. Additionally, the processing delay increases slightly with the packet size, which demonstrates that processing delay can be influenced by copy operations between kernel and user space. Fortunately, both throughput and latency can be drastically optimized by kernel bypassing techniques like DPDK [27] and netmap [23].

5.2 Prototyping Cases

We briefly verify whether FAST can support all four models of software hardware co-design we proposed using prototypes we developed and evaluate their performance. All these prototypes are open-sourced on github [8].

5.2.1 Segment Routing. Segment Routing (SR) [5] supports routing info inserted into packet header so as to instruct packet forwarding along the path. It reduces complicity of the control plane design in MPLS and speed up routing convergence drastically compared with RSVP. We developed the network router that supports SR (in IPv6) based on OpenBox S4 using FAST framework. In the prototype, core logics of data plane are implemented in hardware as UMs and software as UAs simultaneously.

We test the correctness of our SR router in a topology containing 2 PC (A and B), 5 routers and 3 different paths. We use the floodlight [30] as the controller for SR. Then we send ICMP packets fromA to B. We randomly change the SR routing path on the controller, and confirmed that the ping routing successfully changed according to the configuration on the controller.

We also test the performance of the SR router we developed using IXIA PerfectStorm network tester [11]. The network tester are connected in the topology as the replacement of PC A and B. Fig. 11 compares the throughput and latency between segment routing and direct software forwarding. Both methods suffer from the performance bottleneck of packet I/O in linux kernel. Additionally, for segment routing, operations including packet header parsing, modification and matching act as another main performance barrier, costing 20-200 Mbps throughput and 30-60 us latency. Considering the low-end CPU on OpenBox S4 (dual core with only 866 MHz frequency), the performance is reasonable.

5.2.2 VNF Accelerating. In the scenario of NFV, FAST needs to support VNF acceleration by offloading some logics onto FPGA from general processors. We evaluate the feasibility of this model by building a FPGA accelerated packet filter firewall on OpenBox S4.

In the experimental setting, we rewrite iptables [22] by FAST framework to offload 16 entries of its matching table onto FPGA (according to time correlation), so as to enable fast packet processing on the hardware. In the control setting, we migrate iptables to the CPU of OpenBox S4 without hardware assistance. We use network tester to generate traffic of different missing rate on the hardware[6] and observe the performance.

The performance under different missing rate on hardware are demonstrated in Fig. 12. It is apparent that using FAST accelerated

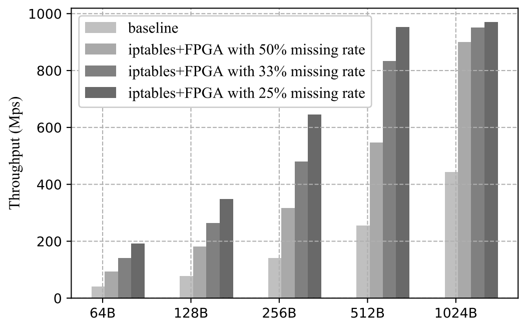


Figure 12: Performance under different missing rate of hardware matching.

[5]Standard MAC core (IP block) provided by Xilinx and Altera satisfy line rate processing on FPGA, thus we can visualize the bottleneck between CPU and FPGA.

[6]We insert a random packet between every N same packets, thus the missing rate can be calculated as $\frac{1}{1+N}$.

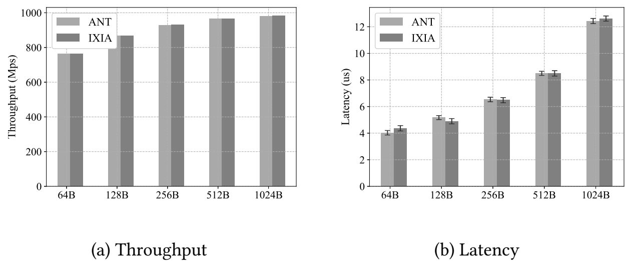


Figure 13: The performance of the Pica8 P3297 switch tested by ANT and IXIA network tester.

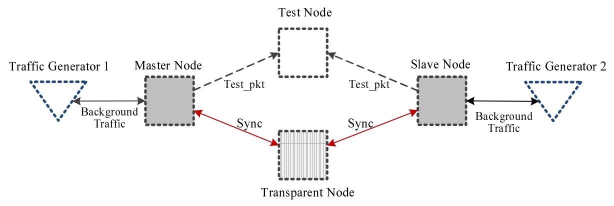


Figure 14: Topology of PTP evaluation.

packet filter firewall showed much higher throughput (2x-4x) compared with original version. This verified the feasibility of using FAST to build a hardware accelerator for VNF.

5.2.3 Network Testing. We developed the agile network tester (ANT) based on OpenBox S4 using FAST framework to evaluate the feasibility of using UA as control plane of UM pipeline. Briefly, the UM pipeline of ANT is able to generate user-defied traffic using a template packet and statistics valuable results in registers (packet rate, latency, drop rate, etc.). the UA in ANT is responsible for generating template packet, configuring parameters (test time, traffic rate, etc.) and collecting results (throughput, latency, drop-rate, etc.).

We evaluate the precision and performance of ANT by comparing the testing results of a Pica8 P3297 switch [19] with IXIA network tester. In both settings, we connect port 2 and 3 of the switch to the tester. Both ANT and IXIA generated traffic of different packet size, ranging from 64B to 1024B, to port 2 and collected results from port 3. We repeated the experiment for 100 times and the results are shown in Fig. 13. ANT showed similar accuracy as IXIA network tester with less than 1.5% deviation both in throughput and latency test. This verified the feasibility of using FAST to develop network devices when software serves hardware as the control plane.

5.2.4 Precise Time Synchronizing. High level time synchronization is often a key step for protecting real-time traffic in a hybrid environment. In order to support time sensitive services, we implemented Precision Time Protocol (PTP)[25] in OpenBox S4 based on FAST framework. PTP in FAST is fully implemented on FPGA and the timestamp are tagged based on peripheral clock to eliminate deviations introduces by FIFOs on the port.

We evaluate the precision of PTP based the a topology (Fig. 14) consisting of 4 OpenBox S4 and 2 network testers (used to generate traffic). The slave node synchronized with the master node through a transparent node. The traffic generators keeps inject background

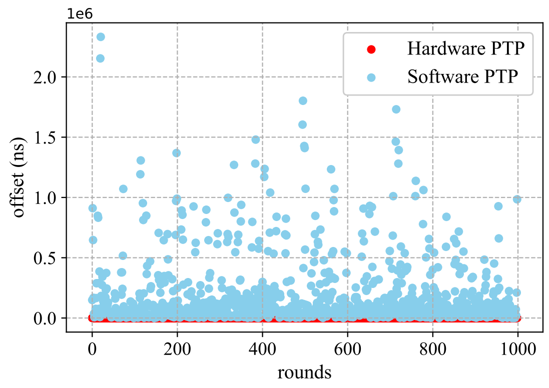


Figure 15: PTP performance evaluation of both hardware and software versions.

traffic to the network. Every 1ms, the master and the slave node will send a pair of signal packets to the test node, which timestamps the incoming two packets and calculate the drift of time synchronization. We ran the experiment for 1000ms and record all the deviations. Then we ran the software version of PTP using [1]. The result is shown in Fig. 15. As we expected, the time drift of each round does not exceed 24ns and is about 11ns in average for PTP implemented using FAST, which is far more precise than software version and is enough to satisfy time sensitive services.

6 RELATED WORKS

We discuss and compare the state-of-art works that are directly related to FAST.

NetFPGA [18] is a FPGA board with hardware components such as RAMs, PHY and MAC chips, etc. to allow college students and researchers to build network prototypes. By offering the abstraction of modular programming on Xilinx FPGA, NetFPGA greatly reduced the FPGA developing workload by code reusing. FAST was largely inspired by NetFPGA and used a similar hardware abstraction. However, FAST provides more comprehensive APIs on software and a novel index mechanism to support a wider range of co-design models. In addition, FAST is accessible on any off-the-shelf CPU-FPGA combinations which also increased versatility.

ClickNP [13] is another framework providing rapid network prototyping on FPGA. It simplifies network prototyping by allowing users to use high-level language to program FPGA. However, ClickNP only focuses on FPGA and does not provides users with a holistic view of network functions across software and hardware, which has been proved to be crucial in network community [7, 29]. The similar shortcoming exists in several other works [2, 28, 31]. In contrast, FAST takes the whole packet processing as a chain of modules (UA and UM), and users can choose how to map them across software and hardware flexibly.

There are also numerous works that can be integrated with FAST to deliver flexibility or easy-to-use features. OpenBox [3] is a software-defined framework for network-wide development, deployment and management of NFs' control plane. FAST can be integrated with OpenBox as Service Instances to enable rapid and network-wide NF deployment on servers with FPGA accelerating cards. mOS [12] and Bro [21] are two representative middlebox developing frameworks with support of L4 flow management. Though FAST does not provide abstractions for stateful processing, we have integrated a subset of both mOS and Bro with FAST for high performance stateful processing on CPU-FPGA platforms.

7 CONCLUSION AND FUTURE WORKS

Although using software and hardware (CPU-FPGA platforms) co-design for network prototyping has become a consensus, existing works failed to provide researchers with a framework that supports rapid and efficient network prototyping on these platforms. This work addresses the challenge with a general and easy-to-use programming framework for network boxes on any CPU-FPGA platforms. FAST provides a holistic view of the network function and uses a modular design for logics both on the CPU and FPGA. In this way, users can map their modules across software and hardware according to their specific requirements and restrictions.

We demonstrated the effectiveness and performance of FAST by a large spectrum of prototypes we designed using the framework. Through out evaluation, FAST satisfies all four software hardware co-design models and provide better performance compared with the software-based prototypes. However, as FAST aims at packet-level processing, building flow-level network boxes using FAST remains a future work and is out of the scope of this paper. We open-sourced FAST and all the prototypes we build based on FAST on github to encourage researchers to contribute to the FAST community and simplify their prototype development using FAST.

ACKNOWLEDGMENTS

This work is supported in part 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. ZK17-03-53.

REFERENCES

[1] Vaibhav Aggarwal. 2019. IEEE1588-PTP. Retrieved Jan 17, 2019 from https://github.com/bestvibes/IEEE1588-PTP
[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[3] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2015. Openbox: Enabling innovation in middlebox applications. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, 67–72.
[4] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. 2018. FBOSS: building switch software at scale. In *Proceedings of the 2018 ACM SIGCOMM Conference*. ACM, 342–356.
[5] Clarence Filsfils, Nagendra Kumar Nainar, Carlos Pignataro, Juan Camilo Cardona, and Pierre Francois. 2015. The segment routing architecture. In *Global Communications Conference (GLOBECOM), 2015 IEEE*. IEEE, 1–6.
[6] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud.. In *NSDI*, Vol. 17. 315–328.
[7] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Renton, WA*.
[8] FAST Group. 2019. FAST Project. Retrieved Mar 17, 2019 from http://www.fastswitch.org
[9] PK Gupta. 2019. Xeon+FPGA Platform for the Data Center. Retrieved Jan 17, 2019 from https://www.archive.ece.cmu.edu/calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf
[10] Intel. 2019. Intel ARRIA-V FPGAs. Retrieved Mar 1, 2019 from https://www.intel.com/content/www/us/en/product-s/programmable/soc/arria-v.html
[11] IXIACOM. 2019. ixia perfectstorm 10/1GE. Retrieved Jan 17, 2019 from https://www.ixiacom.com/zh/products/perfectstorm-101ge
[12] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. 2017. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes.. In *NSDI*. 113–129.
[13] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 1–14.
[14] Dagang Li, Rong Du, Ziheng Liu, Tong Yang, and Bin Cui. [n.d.]. Multi-copy Cuckoo Hashing. ([n. d.]).
[15] Tao Li, Zhigang Sun, Chunbo Jia, Qi Su, and Myungjin Lee. 2011. Using NetMagic to observe fine-grained per-flow latency measurements. In *ACM SIGCOMM Computer Communication Review*, Vol. 41. ACM, 466–467.
[16] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. 2018. Multi-path transport for {RDMA} in datacenters. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 357–371.
[17] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting network support for RDMA. In *Proceedings of the 2018 ACM SIGCOMM Conference*. ACM, 313–326.
[18] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. 2008. NetFPGA: reusable router architecture for experimental research. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*. ACM, 1–7.
[19] Pica8 Open Networking. 2019. Pica8 P3297 datasheet. Retrieved Mar 1, 2019 from https://www.pica8.com/wp-content/uploads/pica8-datasheet-48x1gbe-p3297.pdf
[20] Open Networking Foundation (ONF). 2019. OpenFlow Switch Specification 1.5.1. Retrieved Jan 17, 2019 from https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf
[21] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. *Computer networks* 31, 23-24 (1999), 2435–2463.
[22] Michael Rash. 2007. *Linux Firewalls: Attack Detection and Response with iptables, psad, and fwsnort*. No Starch Press.
[23] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*. 101–112.
[24] Martin Roesch et al. 1999. Snort: Lightweight intrusion detection for networks.. In *Lisa*, Vol. 99. 229–238.
[25] Ruxandra Lupas Scheiterer, Chongning Na, Dragan Obradovic, and Günter Steindl. 2009. Synchronization performance of the precision time protocol in industrial automation networks. *IEEE Transactions on Instrumentation and Measurement* 58, 6 (2009), 1849–1857.
[26] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. 2019. Direct Universal Access: Making Data Center Resources Available to {FPGA}. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*.
[27] Zidong Su, Bruno Baynat, and Thomas Begin. 2017. A new model for DPDK-based virtual switches. *2017 IEEE Conference on Network Softwarization (NetSoft)* (2017), 1–5.
[28] Nik Sultana, Salvator Galea, David Greaves, Marcin Wójcik, Jonny Shipton, Richard Clegg, Luo Mai, Pietro Bressana, Robert Soulé, Richard Mortier, et al. 2017. Emu: Rapid prototyping of networking services. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 459–471.
[29] Jürgen Teich. 2012. Hardware/software codesign: The past, the present, and predicting the future. *Proc. IEEE* 100, Special Centennial Issue (2012), 1411–1430.
[30] Ryan Wallner and Robert Cannistra. 2013. An SDN approach: quality of service using big switch's floodlight open-source controller. *Proceedings of the Asia-Pacific Advanced Network* 35 (2013), 14–19.
[31] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*. ACM, 122–135.
[32] Loring Wirbel. 2014. Xilinx SDNet: A New Way to Specify Network Hardware. (2014).
[33] Xilinx. 2019. Xilinx ZYNQ-7000 FPGAs. Retrieved Mar 1, 2019 from https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html
[34] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 ACM SIGCOMM Conference*. ACM, 561–575.
[35] Xiangrui Yang. 2019. Agile Network Tester. Retrieved Mar 1, 2019 from https://github.com/Winters123/antDev
[36] Xiangrui Yang, Biao Han, Zhigang Sun, and Jinfeng Huang. 2017. Sdn-based ddos attack detection with cross-plane collaboration and lightweight flow monitoring. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 1–6.