PPB: a Path-based Packet Batcher to Accelerate Vector Packet Processor

Jinli Yan
College of Computer
National University of Defense Technology
Changsha, China
Email: yanjinli10@nudt.edu.cn

Lu Tang*
College of Computer
National University of Defense Technology
Changsha, China
Email: lutang@nudt.edu.cn

Tao Li
College of Computer
National University of Defense Technology
Changsha, China
Email: taoli@nudt.edu.cn

Gaofeng Lv
College of Computer
National University of Defense Technology
Changsha, China
Email: lvever@nudt.edu.cn

Wei Quan
College of Computer
National University of Defense Technology
Changsha, China
Email: w.quan@nudt.edu.cn

Hui Yang
College of Computer
National University of Defense Technology
Changsha, China
Email: yanghui@nudt.edu.cn

***Abstract*—Fast Data I/O (FD.io) is an open project to promote network processing based on generic hardware platforms, providing a feasible path to accelerate Network Function Virtualization (NFV). As the core of FD.io, vector packet processor (VPP) is a modularized and high-performance software framework for building network data plane applications. The core idea of VPP has two parts. Firstly, VPP processes a group of packets (packet vector) every time to reduce the instruction cache (i-cache) misses with vectorized processing model. Secondly, VPP decreases the data cache (d-cache) misses by prefetching the following packets. We observe that the received packet order has a significant impact on the performance of VPP. When the packets within a vector traverse different paths, the vector would be split into multiple small vectors. The frequency of i-cache and d-cache misses would increase.**

In this paper, we design a model to quantify the overhead in VPP. The key point to reduce the overhead is to decrease the number of processing paths in a vector. We propose a Path-based Packet Batcher (PPB) to accelerate VPP by adjusting the packet order. Before VPP processing packets, PPB batches the packets based on the processing paths they will traverse. We build a PPB prototype based on FPGA to evaluate the performance optimizations. The experimental results show that the reduction of i-cache and d-cache misses can be up to 57.6% and 17% respectively.

***Index Terms*—High Performance, Packet Processing, Cache Misses, VPP, PPB.**

I. Introduction

Network Function Virtualization (NFV) [15] addresses the problems on traditional proprietary devices by decoupling the hardware from software. It supports fast delivery and deployment of network functions (NF) in the generic hardware platform. With the requirement of high-bandwidth and low-latency, accelerating NFV becomes a hot spot.

*Lu Tang is the corresponding author.

In order to improve the performance of NFV, Intel, Cisco, et al. build an open project named Fast Data I/O (FD.io) [3], focusing on providing a high-throughput, low-latency, and resource-efficient I/O services. Vector packet processor (VPP), as the core technology of FD.io, is a high-performance packet processing framework with vectorized processing model, packet data prefetch and other optimizations [18]. The IP forwarding performance of VPP could reach up to 15+ Mpps without any drops in a single core. The NFV acceleration with VPP is represented in two parts. One is using VPP as a vSwitch in NFV infrastructure (NFVi). The other is developing Virtual Network Functions (VNF) with VPP.

The main goal of VPP is to reduce the instruction cache (i-cache) and data cache (d-cache) misses. In the run-to-completion (RTC) model, the next packet would not be processed until the current packet traverses the entire processing path. If the processing logic is complex, the i-cache will overflow and there are a significant amount of i-cache misses. In order to decrease the i-cache misses, VPP divides network processing code into numerous nodes. The code segment in each node is optimized to fit inside the i-cache. Each node processes a group of packets (packet vector) every time with a vectorized processing model. After the first packet heats up the i-cache, the rest of the packets in this packet batch are processed "for free". With regard to the d-cache misses, each node processes 2/4 packets in every loop. Before the current packets being processed, the following packets in the next loop would be prefetched into d-cache. In this way, packet prefetch and processing are parallel and the probability that packet data hit in d-cache would be improved greatly. In most cases, each node only accesses the packet data from memory in the first loop.

The packet order has a significant impact on the perfor-

mance of VPP. In an ideal case, all the packets in a packet vector traverse the same path without splitting into small vectors. However, the RSS (receive side scaling) [11] in most existing NICs only classifies packet flows with static fields. Since the granularity of flow is coarse, the packets in one vector may traverse different processing paths in some scenarios. When the vector is split into several small vectors, the overhead will increase in two aspects. On one hand, the number of i-cache switches goes up as the packet vector size decreases. On the other hand, the number of memory access increases as the sum of nodes that packet vector traverses grows. It means that the d-cache misses rises.

The reason why a packet vector is split into multi-vectors is the tight coupling between packet parse and processing. The packet parse mainly determines the processing path that a packet would traverse. Thus, we could not obtain the complete processing path only by parsing the packet partially.

In this paper, we present a Path-based Packet Batcher (PPB) to accelerate VPP. Before VPP processes packets, PPB organizes batches in which all the packets traverse the same path by adjusting the packet orders. Therefore, the number of processing paths in one packet vector would decrease. Our main contributions are:

- We design an overhead evaluation model for VPP. To the best of our knowledge, it is the first time to quantify the overhead of VPP. And we find that the key point to accelerate VPP is reducing the number of processing paths in each packet vector.
- We propose the architecture of PPB. PPB is responsible for adjusting the packet orders at the pre-processing stage, which is transparent to VPP. The network functions based on VPP could run normally without any modification. Moreover, PPB provides configuration interfaces to implement user policies.
- We build a PPB prototype based on FPGA for its high flexibility and parallelism. The experimental results show that the reduction of i-cache and d-cache misses can be up to 57.6% and 17% respectively. The core logic of PPB consumes only 5.7% logic resources and 6.3% register resources of the FPGA (Altera Arria V 5AGTMC3D3F31I5).

The rest of the paper is organized as follows. We introduce the motivation for accelerating VPP in Section II. We present the design of PPB in Section III. Section IV presents evaluation results, followed by a discussion in Section V. We discuss the related work in Section VI and conclude the paper in Section VII.

II. Motivation

In this section, we first introduce how VPP accelerates NFV and the principles of VPP. Then, We analyze the drawbacks of VPP and quantify the overhead of VPP.

A. NFV acceleration with VPP

The core idea of NFV is to decouple hardware from software, making software-based network functions developed and deployed on the generic hardware platform. As shown in Fig. 1, the architecture of NFV mainly consists of NFV infrastructure (NFVi) and virtual network functions (VNF). NFVi is responsible for providing the computing, storage and network resources. Virtual Switch(vSwitch), as the core component of NFVi, is a packet forwarding engine based on software. It supports the communication service between VNF and external networks and among VNFs. VNF is deployed in VMs or containers on the top of NFVi. Although NFV supports the fast deployment of VNFs, there is a performance gap between the NFV platform and traditional dedicated devices. Therefore, how to accelerate NFV becomes a very valuable research topic.

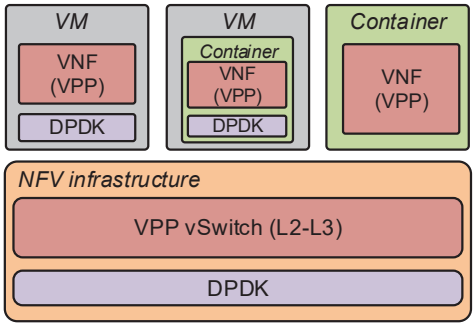

Fig. 1. NFV acceleration with VPP.

Intel Data Plane Development Kit (DPDK) is a high performance userspace I/O framework with zero-copy, kernel bypass, polling mode, and other optimizations. Although DPDK solves the bottleneck of receiving and sending packets, it lacks a framework for users to develop high-performance NFs. The vector packet processor (VPP) is orthogonal to DPDK for providing a high-performance packet processing framework.

Before introducing the principles of VPP, we describe the cache hierarchy briefly [9]. In general, there are 3 levels of cache in current CPU architecture. The L1 cache (per-core) is divided into an instruction cache (i-cache) and an data cache (d-cache). In L2 cache and L3 cache, data and instructions share the storage. The access time of L1 cache is only almost 4 CPU cycles while that of L2 and L3 cache are around 12 and 30 CPU cycles. When the data or instructions are not present at the cache hierarchy, the main DDR memory would be accessed for more than several hundred CPU cycles. Since the network processing is I/O intensive, there are plenty of memory accesses produced by DMA transmission of the network interface card. The memory access latency from CPU would be amplified because of the conflicts of DMA transmission. In order to achieve higher performance, it is very significant to avoid accessing data or instruction from memory.

The core ideas of VPP is reducing the i-cache and d-cache misses by vectorized processing model and packet data prefetch. VPP abstracts the packet processing logic into a directed graph. It divides the network processing code into multiple nodes, the code segment of each node is optimized to fit inside the i-cache. Each node processes a group of packets (packet vector) every time. After the first packet heats up the i-cache, the rest of the packets in this packet batch are processed

"for free". With respect to the data-cache, the processing logic within each node is a loop structure. Each node processes 2 or 4 packets in each loop. Before the packets in the current loop being processed, the following packets in the next loop would be prefetched into d-cache. In this way, packet prefetch and processing are parallel and the probability of packet data hit in d-cache would be improved greatly. In most cases, each node only accesses the packet data from memory in the first loop.

VPP is widely applied to accelerate NFV [4]. For instance, Virtualized Broadband Remote Access Server (vBras) is a typical application of NFV in carrier networks [5]. Bras is the access gateway for the broadband applications in the edge of a backbone network. The traditional Bras only provides the basic network access functions, such as PPPoE (Point to Point Protocol over Ethernet), NAT (Network address translation) and etc. In order to support multiple functions in a dynamic network, vBras deploys virtual network functions (VNF) in containers [19] and virtual machines(VM). In Fig.1, the applications of VPP consist of two aspects. Firstly, VPP is used as a vSwitch in NFV infrastructure (NFVi), providing L2 and L3 forwarding functions. Secondly, many VNFs in Containers or VMs are developed with VPP, such as network protocols, traffic scheduling and security defense.

B. Overhead evaluation model

The packet order has a significant impact on the performance of VPP. In an ideal state, all the packets in a packet vector traverse the same path and the per-packet overhead is the lowest. However, in most cases, the packet vector is split into multiple small vectors. The i-cache and d-cache misses would increase.

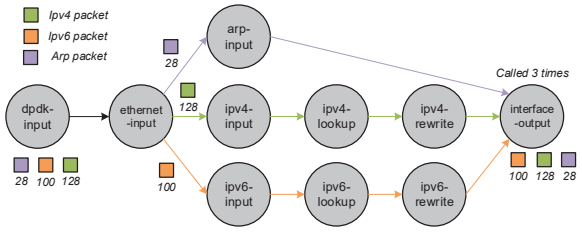

Fig. 2. Packet vector split in VPP.

For example, there are three processing paths for ARP packet, IPv4 packet, and IPv6 packet respectively in Fig. 2. VPP traverses every node layer by layer. Since the output node for these three paths is the *interface-output*, the node function would be called 3 times. The first node *dpdk-input* organized a packet vector with 256 packets (28 ARP packets, 128 IPv4 packets, and 100 IPv6 packets). After being processed by the *ethernet-input* node, this vector would be split into three vectors. In the *ethernet-input* node, the i-cache switches happen every 256 packets while that every 28 packets in the *arp-input*. We assume every node processes 4 packets and prefetches 4 packet headers in each loop. In the ideal situation, the first 4 packets within 256 packets would be accessed from memory in the *ethernet-input* nodes. As for the *arp-input* node, the first 4 packets within 28 packets would be acquired from memory. That is to say, the frequency of i-cache and d-cache switches improves.

In order to quantify the overhead of VPP, we set up an overhead evaluation model. The symbols used in this model are listed in Table I. Here we use latency to represent overhead. We divide the overhead in each node into business processing overhead and non-business processing overhead. The former is the overhead of network function, such as lookup and modify the packet header. The latter indicates the overhead from the VPP framework, composed of i-cache and d-cache misses from fetching instructions and packet data.

TABLE I
SYMBOL DESCRIPTION

Symbol	Description
N	The total number of packets
M	The maximum size of one vector
n	The number of packet vectors
I	The overhead of i-cache misses in each node
D	The overhead of d-cache misses in each node
L	The size of cache-line for i-cache and d-cache
P	The size of i-cache
W	The amplification brought from memory access conflict
A_M	The overhead of one memory access
B_i	The business processing overhead in node i
s_i	The packet vector size in node i
k_i	The number of prefetched packets each time in node i
$O_{node}(i)$	The total overhead in node i
$O_{vec}(j)$	The total overhead of packet vector j
d_j	The number of nodes that vector j traverses
O_{total}	The total overhead of all the packet vectors

Firstly, we compute the overhead of one vector in node i ($O_{node}(i)$). We assume the packet vector size in node i is s_i. The value of $O_{node}(i)$ contains the overhead of business processing (B_i), i-cache misses(I) and d-cache misses(D), as shown in Eq. (1).

$$O_{node}(i) = s_i \times B_i + I + D \tag{1}$$

As for I, the i-cache misses appear when a node process the first packet. Since the size of each node is equal to that of i-cache (P), the I for each node is almost the same. The size of every memory access is a cache-line (L). Here we assume the overhead for one memory access is A_M and the amplification brought from the conflict with DMA is W. Because the instructions are hit in i-cache for the left packets, the overhead from the i-cache access is omitted here. The I is shown in Eq. (2):

$$I = \left\lceil \frac{P}{L} \right\rceil \times A_M \times W \tag{2}$$

With regard to D, we assume node i processes k_i packets in each loop. And it prefetches k_i packet headers (64B) before processing the current packets. Since the packet prefetch and processing are parallel, the following packets would be hit in d-cache without accessing memory. Therefore, the d-cache misses are mainly from acquiring the first k_i packets. Here

we omit the latency from accessing d-cache for simplicity. We derive that

$$D = k_i \times \left\lceil \frac{64}{L} \right\rceil \times A_M \times W \tag{3}$$

Secondly, we analyze the total overhead of packet vector j ($O_{vec}(j)$). The number of nodes that vector j traverses is d_j. The $O_{vec}(j)$ is as follows.

$$O_{vec}(j) = \sum_{i=1}^{d_j} (s_i \times B_i + I + D) \tag{4}$$

Finally, we acquire the total overhead of all the packet vectors(O_{total}). We assume the total number of packets is N and the maximum size of one vector is M. The number of packet vectors n is $\lceil \frac{N}{M} \rceil$. The O_{total} is expressed in Eq. (5).

$$O_{total} = \sum_{j=1}^{n} \sum_{i=1}^{d_j} (s_i \times B_i) + \sum_{j=1}^{n} \sum_{i=1}^{d_j} (I + D) \tag{5}$$

Actually, the processing path for each packet remains constant. The total overhead brought from business processing $\sum_{j=1}^{n} \sum_{i=1}^{d_j} (s_i \times B_i)$ is a fixed value. In order to accelerate VPP, we need to optimize the $\sum_{j=1}^{n} \sum_{i=1}^{d_j} (I + D)$. From Eq. (2) and (3), the value of $I + D$ is constant. The d_j could be changed by reducing the number of processing paths in a packet vector. In an ideal situation, there is only one processing path for a vector and d_j is minimum.

Form Eq. (5), the percentage that business processing overhead occupies impacts the performance improvement brought from reducing i-cache and d-cache misses. With regard to the computation-intensive applications, such as IPsec, the overhead from the encryption and hashing is dominant. In this case, the acceleration from PPB is relatively less. On the contrary, for the functions with low computing load (such as IP forwarding), the optimization is more obvious because the percentage that non-business processing overhead takes up is relatively higher.

In conclusion, our goal is to reduce the number of processing paths in a packet vector, which is closely related to the packet order. In this paper, our core idea is to adjust the packet orders according to the packet processing paths. First, the processing path for each packet is obtained by complete packet parse. Then we organize batches in which all the packets traverse the same path. A more detailed design is introduced in Section III.

III. PPB Design

PPB is a novel method that accelerates VPP by reducing the packet vector split. PPB is implemented based on FPGA for its high flexibility and high performance. In this section, we set three main design goals for PPB. Then we provide details on the architecture and core components.

A. Design goals

Programmability and Extensibility. PPB must provide configuration interfaces for users to implement their policies. In particular, the tables could be configured dynamically to adapt to different traffic patterns better. Besides, PPB should support fine-grained processing path classification and make it easy to add support for the parse of new protocols.

Resource utilization. The PPB is only responsible for adjusting packet orders without implementing any network function. It should be realized easily and occupy a few resources. The best way to implement PPB is by reusing and modifying the existing modules. The FPGA-based SmartNIC provides "Parse→Match→Action" processing model that PPB could be easily mapped to.

Backward Compatibility. Finally, using PPB should require no modification to the VPP framework and any VPP node. In other words, PPB is transparent to VPP. It should be dynamically loaded and unloaded without affecting VPP.

B. PPB architecture

The architecture of PPB is depicted in Fig. 3. PPB is composed of three main parts: *Parser*, *Matcher* and *Batcher*.

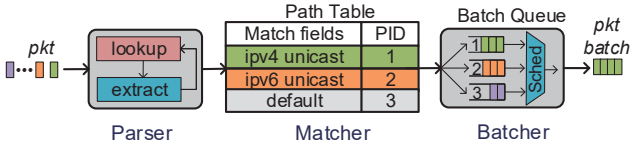

Fig. 3. PPB Architecture.

Parser is used to identify and extract appropriate fields in the packet header. The extracted fields mainly determine which flow a packet belongs to. Here we classify the packet flows according to the processing paths. Since many network functions (such as firewall, load balancer, and intrusion detection system) process packets in L3+, a programmable parser is needed to support the parse for L1-L7 and new protocol type. The design of our programmable parser refers to [13]. It mainly comprises of a *lookup* unit and an *extract* unit. The *lookup* unit identifies the type field according to the offset of the header type. The *extract* unit extracts the corresponding fields and calculates the offset of the next type.

Matcher has a path table storing the mapping relationships between match fields and path id (PID). The match fields determine which path a packet belongs to. The PID indicates which queue a packet would be sent to. During system initialization, users configure the path table according to the predicted traffic pattern. During the process, the path table can be configured dynamically by monitoring the changes of the traffic pattern. In Fig. 3, the $ip4\_unicast$ and $ip6\_unicast$ packets are inserted into queue 1 and 2 respectively. The other flows (such as ARP) are matched to the default entry and inserted into queue 3.

Batcher mainly comprises of several queues. Different queues store the packets traversing different paths. The scheduler organizes a packet batch from a queue each time. The size of a packet batch is represented with *batch_size*. Since

the storage resource is limited, the number of processing paths exceeds that of batch queues in some scenarios. In this case, the packets from multiple paths are mapped to one queue. The bandwidth of elephant flows is high so that the number of packets is up to *batch_size* in a short time. The performance improvement with PPB is more obvious when an elephant flow has an exclusive batch queue. On the contrary, the mice flows have low bandwidth. We aggregate several mice flows into one batch queue. We use a round-robin policy to schedule a queue currently. In the future, the flow priority will be taken into account.

IV. Evaluation

To evaluate the performance optimizations to VPP with PPB, we built a PPB prototype based on FPGA (Altera Arria V5AGTMC3D3F31I5) in an experimental platform named iRouter [2]. The topology in our experiment is as shown in Fig. 4. It includes one iRouter controlled by PC B. There is an IXIA network emulator for generating traffic. One Dell Server (Xeon CPU E5-2680 @2.8GHz, 32KB i-cache, and 32KB d-cache) runs with DPDK (v18.02) and VPP (v18.04). The Operating system is Ubuntu 16.04 LTS. PC A displays the received bandwidth, packet loss rate, and average latency. All the links among iRouter, server, PC, and IXIA are 1GE fibers.

To simulate the worst traffic pattern, the IXIA network emulator sends 64B $ipv4_unicast$ and $ipv6_unicast$ packet by turns at the line speed in the Gigabit link. The volume of packets is set to 100 million in total. The manager thread and worker thread is bound to CPU core 0 and 1 respectively. Since VPP could reach wire-speed forwarding in the Gigabit link all the time, we use Perf [10] to measure the i-cache load misses and d-cache load misses of VPP with and without PPB. Perf is a profiler tool that can measure many events with a very low computational overhead. We use Perf to measure the cache-misses on CPU core 1 and then remove the percentage that DPDK accounts for.

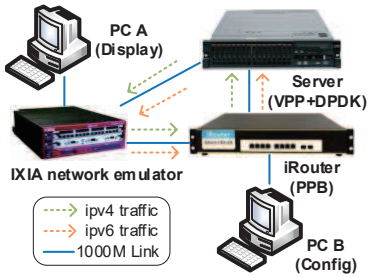

Fig. 4. Experimental Topology.

Fig. 5(a) demonstrates that the i-cache misses in the worst cases without PPB are almost 460 million times. As the *batch_size* in PPB grows, the i-cache load misses decrease approximately by 57.6% when the *batch_size* is 128. Similarly, in Fig.5(b), the i-cache misses in the worst cases without PPB reach up to 129 million times. The d-cache load misses goes down when the *batch_size* rises. There is a 17% reduction for d-cache misses when the *batch_size* is 128. The core logic of PPB consumes only 5.7% logic resources and 6.3% register resources of the FPGA.

The experimental results demonstrate that PPB improves performance efficiently and consumes only a few resources. We admit that some factors influence the performance improvement brought from PPB, such as storage resources, traffic patterns, latency requirements, etc. The demo is the first step for PPB. The factors mentioned above will be considered to adapt to more scenarios better in the future.

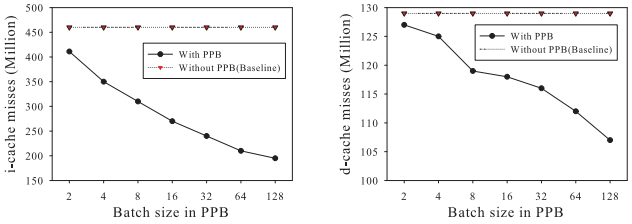

Fig. 5. (a) I-cache misses under different batch sizes. (b) D-cache misses under different batch sizes.

V. Discussion

Latency. We admit PPB produces some latency when adjusting packet order. But the overhead caused by the i-cache and d-cache misses is much more serious. We assume the i-cache size is 32KB, the cache-line is 64B, the memory access latency is 200ns. The overhead of i-cache misses for a node is almost 100μs. As to the 10GE port, the wire speed is 14.88Mpps (64B). The average latency for one packet is 0.067μs. It means 1.5K packets will come when the i-cache is replaced. Besides, we consider that some mice flows (such as ARP, web search) are latency-sensitive. If these packets are mixed with other packets, the performance would decrease. We provide two ideas to solve this problem. One is offloading the processing logic into SmartNIC [12], providing a fast data-path for these flows. The other is dispatching these flows into an exclusive CPU core without affecting other flows.

Multi-core scalability. In this paper, we mainly focus on optimizing the performance of VPP in a single core. In the next step, the multi-core resources would be fully exploited. VPP uses duplication scheme to scale on multiple cores. It means that the worker thread in each core has a complete processing graph. Our idea is to replace the RSS mechanism with PPB. PPB maps the batch queues to multiple cores. In an ideal situation, each batch queue has an exclusive CPU core. When the number of batch queues is higher than that of CPU cores, the elephant flow and latency-sensitive flow have a high priority to occupy a CPU core.

VI. Related Work

Batch processing. The batch processing is used in packet I/O and packet processing [16]. The I/O batching means to receive and send a group of packets each time, which amortizes the overhead associated with NIC (such as lock acquisition, system call cost, and etc.) [6]. The I/O batching is widely

applied in high-performance I/O frameworks (Intel DPDK [1], netmap [20]). Batch computing emphasizes to process a group of packets every time for reducing the i-cache misses. In this paper, we mainly research how to accelerate batch computing. The main challenge that batch computing faces is the packets within a packet vector traverse different processing paths. There are two existing solutions. One is to split the packet vector into multiple vectors [18] [6]. This way, it needs to allocate and organize new vectors. The other is to flag some packets for the subsequent nodes to ignore them. Although it avoids reorganizing the vectors, the processing overhead would improve with unnecessary waiting. It is suitable to flag the outport and error packets. Different from both methods, PPB classifies the packets according to the processing path in the pre-processing stage, followed by organizing a packet batch that traverses the same path. Therefore, the packet vector split overhead is reduced greatly.

Hardware acceleration. The existing hardware platforms accelerating network processing consist of ASIC, GPU, and FPGA. RMT [7] achieves programmable packet switching in ASIC by mapping the processing logic into 32 stages. However, being uniformly assigned to each stage, the memory resource would be waste in most cases. dRMT [8] solves this problem by sharing the memory resource. But the latency becomes larger in order to avoid the access conflicts in crossbar. Packetshader [14] and NBA [16] proposed to accelerate network functions with GPU. GPU is deployed with the look-aside model and the NIC could not DMA the packets into GPU directly. ClickNP [17] and SmartNIC [12] focus on offloading the software functions into FPGA. FPGA is deployed as a fast data-path while CPU acts as a slow data-path. There are three reasons why PPB is implemented in FPGA. Firstly, PPB adjusts packet orders in the pre-processing stage, followed by the process in CPU. Secondly, FPGA is suitable to support new protocol parse and fine-grained classification. And many existing modules could be reused. Thirdly, the processing latency is low in FPGA for its high performance.

VII. Conclusion

We design an overhead evaluation model to quantify the overhead produced by VPP. We find that the number of processing paths in a packet vector has an important impact on the performance of VPP. When the packet vector is split into multiple vectors, the i-cache misses and d-cache misses would increase. In order to accelerate VPP, we propose a Path-based Packet Batcher (PPB) to organize packet batches that traverse the same path each time. We build the prototype based on FPGA. The experimental results show that the reduction of i-cache and d-cache misses can be up to 57.6% and 17% respectively. PPB could be implemented easily and consumes only a few resources. We hope that PPB would be embedded into SmartNIC to accelerate network processing in the future.

Acknowledgment

This work is supported by National Key Research and Development Program of China (Grant No.2018YFB1800505, 2018YFB1800402), National Natural Science Foundation of China (Grant No.61702538, 61802417, 61601483), Training Program for Excellent Young Innovators of Changsha (Grant No.kq1905006) and Research Project of National University of Defense Technology (Grant No.ZK17-03-53, ZK18-03-40).

References

[1] DPDK. http://dpdk.org/.
[2] FAST Project. http://www.fastswitch.org/.
[3] Fd.io. https://fd.io/.
[4] Scaling the Container Dataplane. https://www.slideshare.net/MichelleHolley1/scaling-the-container-dataplane.
[5] vBras White Paper. http://res.www.zte.com.cn/mediares/zte/Files/PDF/white_book/ZTE_Carrier-Class_Virtualized_BRAS_White_Paper.pdf.
[6] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems*, pages 5–16. IEEE Computer Society, 2015.
[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110. ACM, 2013.
[8] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. dRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 1–14. ACM, 2017.
[9] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE micro*, 30(2), 2010.
[10] Arnaldo Carvalho De Melo. The New Linux Perf Tools. 2010.
[11] Yaozu Dong, Dongxiao Xu, Yang Zhang, and Guangdeng Liao. Optimizing Network I/O Virtualization with Efficient Interrupt Coalescing and Virtual Receive Side Scaling. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 26–34. IEEE, 2011.
[12] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.
[13] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design Principles for Packet Parsers. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 13–24. IEEE Press, 2013.
[14] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-Accelerated Software Router. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 195–206. ACM, 2010.
[15] Hassan Hawilo, Abdallah Shami, Maysam Mirahmadi, and Rasool Asal. NFV: State of the Art, Challenges, and Implementation in Next Generation Mobile Networks (vEPC). *IEEE Network*, 28(6):18–26, 2014.
[16] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (Network Balancing Act): a High-Performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the Tenth European Conference on Computer Systems*, page 22. ACM, 2015.
[17] Bojie Li, Kun Tan, et al. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14. ACM, 2016.
[18] Leonardo Linguaglossa, Dario Rossi, Salvatore Pontarelli, Dave Barach, Damjan Marjon, and Pierre Pfister. High-Speed Software Data Plane via Vectorized Packet Processing.
[19] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, 2014.
[20] Luigi Rizzo. Netmap: a Novel Framework for Fast Packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.