Demonstration of Path-based Packet Batcher for Accelerating Vectorized Packet Processing

Jinli Yan, Tao Li, Sheng Wang, Gaofeng Lv and Zhigang Sun
College of Computer, National University of Defense Technology
Changsha, Hunan, P.R. China
Email: {yanjinli10, taoli, lvever, sunzhigang}@nudt.edu.cn
xpuwilson@gmail.com

***Abstract*—Recently, a major challenge on generic multi-core network processing platforms is how to improve packet processing performance. Vector packet processor (VPP) is a modularized and high-performance software framework for building network dataplane applications. The key idea of VPP is to reduce instruction cache (i-cache) misses with vectorized packet processing. However, the packets in a vector may traverse different processing paths in some scenarios. In such case, the vector is split into several smaller vectors, and the per-packet overhead would increase. In this paper, we propose a Path-based Packet Batcher (PPB) to accelerate VPP. PPB is transparent to VPP, and it requires no modification to VPP. Before VPP processes packets, PPB batches the packets based on the processing paths they will traverse. We build a prototype based on FPGA to evaluate the performance optimizations to VPP with PPB. Experiment results show that the reduction of i-cache misses can be up to 57.6% when the batch size is 128.**

***Index Terms*—packet processing, i-cache misses, VPP, PPB**

I. Introduction and Motivation

Fd.io (fast data-input/output) is an open source project focusing on supporting flexible, programmable and composable I/O services on generic hardware platforms. Vector packet processor (VPP), as the key component of FD.io, is a modularized and high-performance software packet processing framework[1]. VPP facilitates network operators to build various network data plane applications.

The core idea of VPP is to reduce the number of i-cache misses in the traditional Run-to-Completion (RTC) model with vectorized packet processing model. To introduce these two models clearly, a simple example is given in Fig. 1a. The processing code is divided into three parts: $eth\_parse\ code$, $ip4\_unicast\ code$ and $ip6\_unicast\ code$. In RTC model, the next packet would not be processed until the current packet traverses the entire processing path in Fig. 1b. There are 11 code segment switches for six packets in total. If the processing logic is complex, the i-cache will overflow, and there are a significant amount of i-cache misses per packet. VPP abstracts the processing logic into a directed graph with multiple nodes. These graph nodes are optimized to fit inside the i-cache[6]. Here we suppose that each code fragment in Fig. 1a is a node. Each node processes a packet batch continuously before transmitting them to the next node in vectorized processing model, as shown in Fig. 1c. After the first packet heats up the i-cache, the rest of packets in this

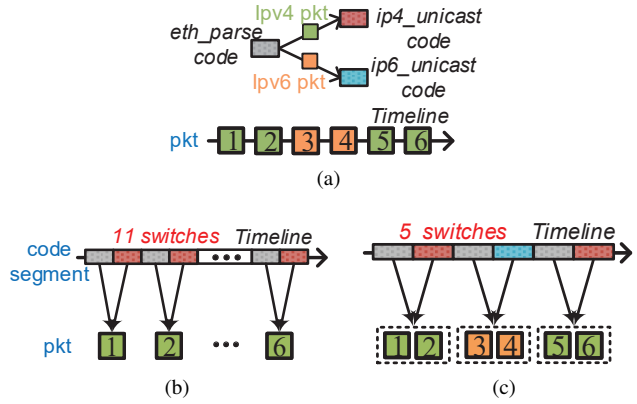


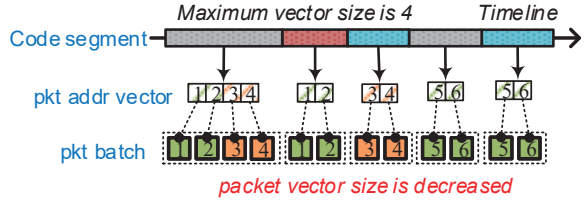
Fig. 1. (a) Processing graph and packet sequences. (b) Run-to-Completion model. (c) Vectorized packet processing model.

packet batch are processed "for free". The number of code segment switches is decreased to 5. When the vector size is set bigger, the number of required code switches become less. Besides, the packet vectors for each VPP node only store the addresses of packets to avoid excessive copy overhead. And these packet vectors are allocated from the pre-allocated memory without buffer allocation from operating system.

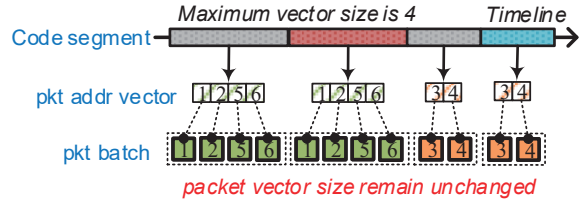
However, the RSS (Receive Side Scaling) in most existing NICs only classify packet flows according to static fields. Since the granularity of flow is coarse, the packets in one vector may traverse different processing paths in some scenarios[4]. We use the example in Fig. 1a to analyze the overhead in VPP. Each code segment is a VPP node. There are two processing paths: *ipv4_unicast* path and *ipv6_unicast* path. Here the maximum vector size in VPP is set as 4. As shown in Fig. 2a, eth_parse node is processing a vector containing two $ipv4$ packets and two $ipv6$ packets firstly. After the protocol parse in L2, two $ipv4$ packets are inserted into the packet vector of $ip4\_unicast$ node while two $ipv6$ packets are inserted into the packet vector of $ip6\_unicast$ node. The vector split makes the packet vector size in the following two nodes decreased. Hence, the frequency of i-cache switches would relatively increase. If the number of processing paths in one vector is large, the performance improvement from VPP will be reduced.

We observe that the number of processing paths in one vector is close related to the packet order. In ideal cases, all the packets in a packet vector traverse the same path without

splitting into small vectors. In this demo, we present a Path-based Packet Batcher (PPB) to accelerate VPP by adjusting packet order. PPB requires no modication to VPP framework and any VPP node, which means PPB is transparent to VPP. Before VPP processes packets, PPB organizes batches in which all the packets traverse the same path. As shown in Fig. 2b, the last two $ipv4$ packets are put ahead of the two $ipv6$ packets. Therefore, the eth_parse node processes a vector comprising of four $ipv4$ packets without split. Since the packet vector size remain unchanged, the frequency of i-cache switches would not increase. For more detailed design about PPB, please refer to Section II.



(a)



(b)

Fig. 2. (a) VPP overhead analysis. (b) Optimization with PPB

II. PPB Overview

We proposed a novel method PPB for accelerating vectorized packet processing. As shown in Fig. 3, PPB is composed of three main parts: *Parser*, *Matcher* and *Batcher*.

- **Parser**. A parser is used to identify and extract appropriate fields in a packet header. The extracted fields mainly determine which path a packet would traverse. In order to select path with fine granularity, a programmable parser is needed to support the parse for L1-L7 and new protocol types. The design of our programmable parser refers to [3]. It mainly comprises of a *lookup* unit and a *extract* unit. The *lookup* unit identifies type field according to the offset of header type. The *extract* unit extracts the corresponding fields and calculates the offset of the next type. At the end of parse, the extracted fields would be combined into a packet feature vector.
- **Matcher**. In this stage, a packet is mapped to a processing path by matching path table with packet feature vector. *Path Info* is a group of packet header fileds, including protocol types, address, port and etc. *QID* indicates which queue a packet should be stored. The path table supports wildcard matching. In ideal situation, each processing path has an exclusive *QID*. Since the storage resource is limited, several processing paths may be mapped to the same *QID*. When the bandwidth from a processing path is high, a packet batch could be organized in a short time. The performance improvement with PPB is more obvious if such processing path has an exclusive batch queue. A default entry is set for the processing paths in which the bandwidth is relatively low. We provide configuration interface for users to manage the path table (such as add, delete and modify operations). During system initialization, users add path entries according to the predicted traffic pattern. In this demo, the $ip4\_unicast$ and $ip6\_unicast$ packets are inserted into queue 1 and 2 respectively. The packets from the other paths are inserted into queue 3 by default, as shown in Fig. 3. During the process, the path table can be configured dynamically by monitoring the changes of traffic pattern.
- **Batcher**. The batcher mainly comprises of several queues. Each queue is mapped to a *QID*. The number of queues is related to the storage resources. The batcher organizes a packet batch from a queue each time. We use round-robin policy to choose a queue currently. In the future, the flow priority will be taken into account. In order to guarantee the requirement of latency in some scenarios, a timer is set for each queue. When the timeout occurs, the packet batch will be transmitted even though this batch does not reach the specified size.

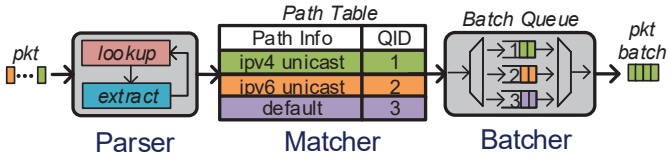


Fig. 3. PPB Architecture.

III. Evaluation

To evaluate the performance optimizations to VPP with PPB, we built a PPB prototype based on FPGA in an experimental platform named iRouter. The reason why we choose FPGA is that the high programmability and parallelism could satisfy the requirement for performance and dynamical configuration[5]. The demo topology that our demonstration uses is as shown in Fig. 4. It includes one iRouter controlled by PC B. There is an IXIA network emulator for generating traffic. The TX port of IXIA network emulator connects to iRouter, which is connected to a Server(Xeon CPU E5-2680 @2.8GHz) running with VPP(v18.04) and DPDK(v18.02). 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, IXIA network emulator sends 64B $ipv4$ and $ipv6$ packet by turns at the line speed in Gigabit link. The volume of packets is set to 100 million in total. Since VPP could reach wire-speed forwarding in Gigabit link, we measure the i-cache misses of VPP under different batch size with Perf[2]. Fig.5 demonstrates that the i-cache misses in the worst cases without PPB is almost 460 million times. As the batch size in PPB grows, the i-cache load misses decrease. The i-cache misses reduced approximately by 57.6%

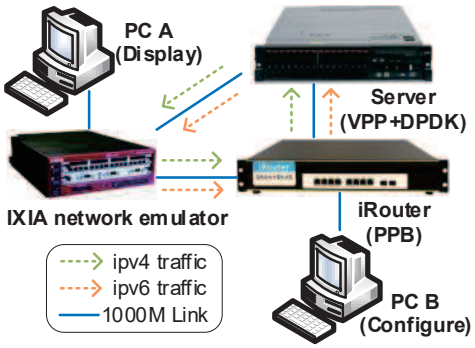


Fig. 4. Demo Topology.

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 (Altera Arria V 5AGTMC3D3F31I5).

In conclusion, the experimental results demonstrate that PPB improves the performance efficiently and consumes limited resources. We admit that some factors influence the performance improvement brought from PPB, such as storage resource, traffic patterns, latency requirement and etc. The demo is the first step for PPS. The factors mentioned above will be considered to adapt more scenarios better in the next step. We hope the PPB will be embedded into SmartNIC to accelerate network processing in the future.

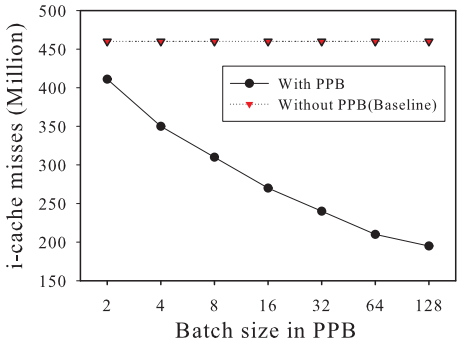


Fig. 5. I-cache misses under different batch sizes.

ACKNOWLEDGMENT

We thank Wei Quan, Donglai Xu and Yanlong Zhang for their help and comments. This research is supported by Chinese National Programs for Key Tech. of SDN, Supporting Resource Elasticity Scheduling, and Equipment Development (863 Programs) (Grant No.2015AA016103), National Natural Science Foundation of China (Grant No.61702538) and Research Project of National University of Defense Technology (Grant No.ZK17-03-53).

REFERENCES

[1] Fd.io. https://fd.io/.
[2] Arnaldo Carvalho De Melo. The new linux perf tools. 2010.
[3] 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.
[4] 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.
[5] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14. ACM, 2016.
[6] Leonardo Linguaglossa, Dario Rossi, Salvatore Pontarelli, Dave Barach, Damjan Marjon, and Pierre Pfister. High-speed software data plane via vectorized packet processing.