Self-described buffer: a novel mechanism to improve packet I/O efficiency in Linux

Jinli Yan, Lu Tang, Zhigang Sun, Tao Li and Donglai Xu
College of Computer, National University of Defense Technology
Changsha, P.R. China
Email: yan_jinli@126.com

***Abstract*—Socket buffer (SKB) is the standard data structure for exchanging packets and their control information between NIC driver and protocol stack. The overhead of dynamic SKB management has been considered as the significant bottleneck in packet I/O. Some novel non-SKB mechanisms, such as DPDK, were thus proposed to solve the problem. However, these mechanisms usually cannot be widely adopted in the data path of most packet forwarding applications, due to their incompatibility with SKB. In this paper, a new SKB-compatible mechanism, namely Self-described buffer (SDB), is proposed to improve the efficiency of packet I/O. SDB eliminates SKB allocation/deallocation overhead by offloading SKB management into NIC hardware. It also reduces the overhead of dynamic binding/unbinding operations existed in SKB management by statically binding related information in advance using the free space of Databuf. To evaluate the proposed approach, a SDB-enabled NIC and its driver has been designed and implemented based on FPGA. Experimental results show that the proposed SDB achieves 2x throughput compared with a traditional SKB mechanism in raw packet forwarding, and 34.75% improvement for typical network forwarding applications (e.g. IP forwarding, Bridge forwarding and SDN forwarding) on average.**

***Index Terms*—packet I/O, SKB, packet buffer management, packet forwarding.**

I. Introduction

Nowadays, commodity CPU/OS platform is widely used in packet processing for its high flexibility and scalability. Although the commodity CPU/OS platforms provide efficient supports for computing-intensive applications. They can hardly perform well for fast network processing because of some limitations, such as poor locality and high latency, existed in the traditional packet I/O architecture[1]. On commodity CPU/OS platforms, packet I/O (including the procedure of receiving and transmitting packets) is the bottleneck of network processing especially packet forwarding applications[2]. Therefore, optimizing packet I/O is important for improving the performance of network processing.

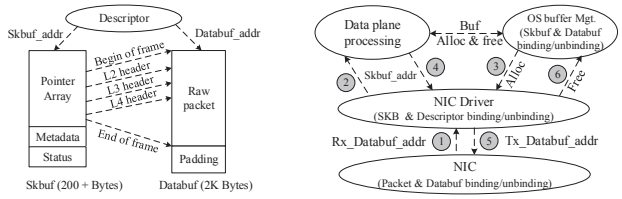
Packet buffer management, consisting of buffer allocation and deallocation operations, is an important part of packet I/O. In Linux, SKB is the key packet data structure for information exchange between the standard Network Interface Controller (NIC) driver and protocol stack. It is composed of a control block (*Skbuf*) and a data block (*Databuf*). *Skbuf* stores the control information of packet while *Databuf* holds the raw packet itself. Packet buffer management indicates the management of socket buffer (SKB) in Linux. In receive side, a SKB is dynamically allocated for every packet received in NIC. Then it is delivered to protocol stack or network applications for further processing. After that, the packet will be transmitted to NIC and SKB will be deallocated by OS. SKB management has a significant overhead due to cache misses, virtual-to-physical translation and etc[3]. As measured in [4], SKB management consumes 63% of the CPU utilization for receiving a single 64B sized packet.

Some techniques have been proposed to reduce the overhead of packet I/O in commodity CPU/OS platform. The core idea is to optimize the path of packet I/O by replacing SKB with the customized lightweight data structure[5], such as PF_RING DNA[6], netmap[7], Intel DPDK[8]. However, these mechanisms are not compatible with SKB data structure, which means the network applications developed based on SKB need a dramatic change before porting to the platforms based on these techniques. Although SKB recycling mechanism has been designed to reuse SKBs for the improvement of packet I/O efficiency, the shared resource competition and queue management will cause non-neglect overhead to packet I/O[9].

In order to optimize packet I/O, we investigate the conventional packet I/O architecture and the overhead of SKB management in Linux. In this process, several observations are found: (1) allocation/deallocation and binding/unbinding of SKB-related data structures are the main operations of SKB management. (2) most operations of SKB allocation/deallocation in packet forwarding applications are accomplished only in NIC driver, which means it is possible to offload SKB allocation/deallocation into NIC. (3) the Ethernet MTU (Maximum Transmission Unit) of received packet is usually 1.5KB, while the size of *Databuf* allocated by NIC driver is 2KB. Therefore, extra space can be utilized to store key information in initialization.

Motivated by these observations, a new SKB-compatible mechanism, namely Self-described buffer (SDB), is proposed to optimize packet I/O of commodity CPU/OS platforms. We design a new data structure SDB, which is composed of a customized descriptor and a raw packet. In initiation, we pre-allocate SDB buffer and *Skbuf* firstly. Then the addresses of SDB and *Skbuf* are filled in the descriptor of SDB, which means the overhead of binding/unbinding operation is

This work is supported by National High-tech R&D Program of China (863 Program) (Grant No. 2015AA0156-03), National Natural Science Foundation of China (Grant No. 61202483).



(a) Key data structure of Packet I/O (b) Architecture of packet I/O

Fig. 1. Architecture and key data structures of packet I/O in Linux.

eliminated by pre-binding the related data structures. After that, the addresses of SDB and *Skbuf* are delivered into the buffer management module of NIC. During packet processing, NIC is responsible for the SKB buffer allocation/deallocation operations without software. When the NIC receives a packet, it transmits the constructed SDB packet and *Skbuf* to the allocated buffers. After the packet is transferred into a link, the NIC reclaims the corresponding address of SDB buffer and *Skbuf* into buffer management module. Overall, the new mechanism ameliorates the major bottlenecks of packet I/O, which makes commodity CPU/OS platforms more suited for high speed network processing.

The remainder of this paper is organized as follows. Section II explains the motivation of our work. Section III introduces the mechanism of SDB in detail. The experiment results are introduced in Section IV. The related work and the conclusion of the paper are given in Section V and VI, respectively.

II. Prerequisites and motivation

A. Packet I/O architecture in Linux

As shown in Fig. 1a, the key data structures of packet I/O in Linux can be divided into three parts. (1) *Databuf*: it is a SKB data block (2KB DMA buffer), which holds the raw packet (1.5KB at most) and padding (reserved field). (2) *Skbuf*: this buffer is used for storing the SKB control block (264B in Linux kernel v3.13.11) including pointer array, metadata and the status of packet. Pointer array holds the head pointer, tail pointer and L2/L3/L4 header of *Databuf*. All information stored in *Skbuf* will be used by protocol stack or packet forwarding applications. (3) *Descriptor*: it contains the DMA address of *Databuf* (*Databuf_addr*) and the virtual address of *Skbuf* (*Skbuf_addr*), status of SKB buffer and etc. Multiple descriptors make up a ring to manage packets received/sent in NIC driver.

The architecture of packet I/O in Linux is composed of four modules and three critical binding/unbinding operations, as shown in Fig. 1b. The detailed processing is illustrated as follows. Before being transferred into NIC driver, NIC allocates a DMA address of *Databuf* (*Rx_Databuf_addr*) for the received packet (*Packet & Databuf binding*) (step 1). The corresponding SKB buffer is then delivered into *Data plane processing module* with the virtual address of *Skbuf* (*Skbuf_addr*) (step 2). *OS buffer Mgt. module* allocates *Skbuf* and *Databuf* respectively and fills the pointer array of *Skbuf* with the information of *Databuf* (*Databuf & Skbuf binding*), then NIC driver fills the descriptor with the DMA address of

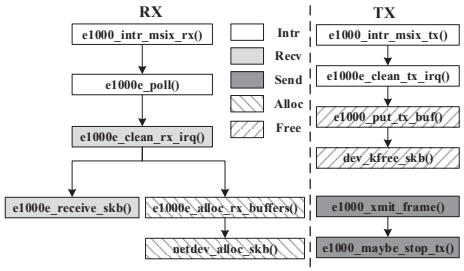


Fig. 2. RX/TX implementation in function level.

Databuf (*Databuf_addr*) and *Skbuf_addr* (*SKB & Descriptor binding*) (step 3). After completing the process, *Data plane processing module* invokes the transmit function provided by NIC driver (step 4). NIC transfers the packet to the link with DMA address of buffer *Tx_Databuf_addr* (including *Packet & Databuf unbinding*) (step 5). SKB buffer is deallocated by the corresponding function of *OS buffer Mgt. module* (including *SKB & Descriptor unbinding* and *Databuf & Skbuf unbinding*) (step 6).

Overall, SKB management is the main part of conventional packet I/O, which contains plentiful dynamic allocation/deallocation and the binding/unbinding operations of SKB-related data structures.

B. Overhead of SKB management

As shown in Fig. 2, the implementation of RX/TX in NIC driver can be divided into five parts: *Intr* (hardware interrupt processing), *Recv* (receiving process), *Send* (transmitting process), *Alloc* (allocation and binding) and *Free* (deallocation and unbinding). To analyze the overhead of each part, we use a profiling tool Oprofile to measure the CPU utilization of Intel 82574L with e1000e driver (Linux v3.13.11). The total CPU utilization of these five parts is regarded as the total overhead. The overhead breakdown is illustrated in Fig. 3. From this figure, we can see that the *Alloc* and *Free* operations are the heaviest CPU workload among the above mentioned five parts. They take up 53.68% of the total CPU utilization.

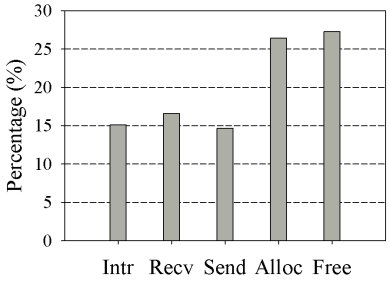


Fig. 3. Overhead breakdown.

The analysis and experimental results verify that SKB management introduces the most overhead. And motivated by the observations described in Section I, we proposed a SKB-compatible mechanism named SDB where the SKB management is offloaded into NIC, and the binding/unbinding operations are completed by pre-filling the available space of *Databuf* with related metadata.

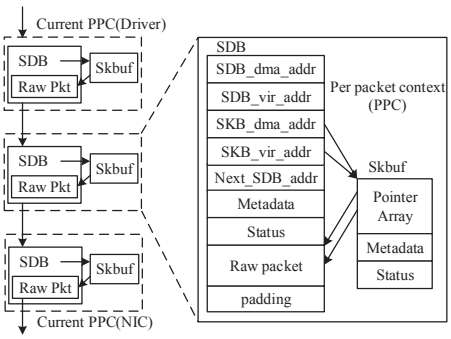


Fig. 4. Data structure.

III. Self-described buffer

A. Key Data structures of SDB

In order to reduce the overhead of packet I/O, we remove the RX/TX ring in the traditional packet I/O architecture and organize the received packets via linked list, as shown in Fig. 4. The per packet context (PPC) is divided into *Skbuf* and SDB (2KB) that is composed of descriptor and raw packet. And some fields in descriptor are illustrated as follows.

- *SDB_dma_addr*: DMA address of SDB, NIC uses this address to perform a DMA transfer.
- *SDB_vir_addr*: virtual address of SDB, it is used to acquire the corresponding SDB buffer by the NIC driver.
- *SKB_dma_addr*: DMA address of SKB buffer, with which the constructed *Skbuf* in hardware is moved to corresponding space in memory.
- *SKB_vir_addr*: virtual address of SKB buffer.
- *Next_SDB_addr*: virtual address of next SDB buffer, which is used to access the linked list.
- *Metadata*: some control information of packet.
- *Status*. it is used to determine whether the SDB buffer is valid.

As shown in Fig. 4, the descriptor of SDB and raw packet is stored in a continuous memory space. *Skbuf* and raw packet can be found by the *SKB_vir_addr* and the pointers stored in *Skbuf* respectively. The tight coupling between SDB buffer and *Skbuf* is thus implemented.

B. Implementation model

The implementation model of SDB is illustrated in Fig. 5, which is composed of seven modules.

- *OS buffer Mgt. module*: it is used to pre-allocate *Skbuf* and SDB buffers.
- *Skbuf & SDB binding module*: this module executes the binding operations for *Skbuf* and SDB by filling in the *SKB_dma_addr*, *SKB_vir_addr* and the pointer array of *Skbuf* in initiation.
- *NIC SDB Mgt. module*: the SKB allocation/deallocation operations in software are offloaded into *NIC SDB Mgt. module*. Therefore, this module is used to allocate and deallocate SDB buffers during I/O processing.
- *DMA engine module*: it is in charge of *Packet & SDB binding* and the construction of SDB packet and *Skbuf*. When NIC receives a packet, it constructs the SDB packet and *Skbuf* and allocates the *SDB_dma_addr* and *SKB_dma_addr* from *NIC SDB Mgt. module*. In the transmit direction, the allocated SDB buffer and *Skbuf* are reclaimed into *NIC SDB Mgt module*.
- *Free SDB cache*: it is used to store SDB buffers. All addresses of pre-allocated SDB buffers are delivered into *NIC SDB Mgt. module* if there are enough resources. Otherwise, some of them are stored in *Free SDB Cache* and will be delivered into *NIC SDB Mgt. module* when the module is short of SDB buffers.
- *SDB NIC driver*: we organize the received packet with a linked list. *SDB NIC driver* acquires *Skbuf* with the *SKB_vir_addr* stored in SDB packet. Thus, the *SKB & Descriptor binding* can be removed.
- *Data plane processing module*: it processes the SKB buffer with *Skbuf_addr* provided by *SDB NIC driver*.

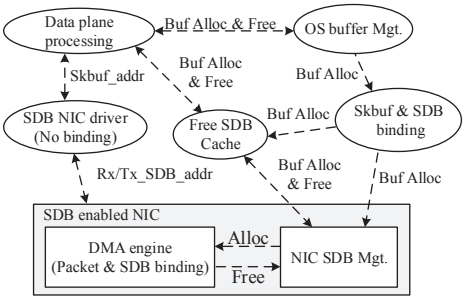


Fig. 5. Implementation model.

C. Processing flow

As aforementioned, we pre-allocate a given number of *Skbuf* and SDB buffers firstly. The DMA addresses of these buffers are delivered into *NIC SDB Mgt. module*. Besides, we create an array to store the virtual addresses of the whole SDB buffers, which is used to deallocate buffers when unloading SDB NIC driver.

The workflow of our proposed approach is illustrated in Fig. 6. *Rxdma* allocates a SDB buffer from *NIC SDB Mgt. module* after receiving a packet (step 1). Afterwards, it transmits the constructed SDB packet and *Skbuf* to the corresponding buffers with *SDB_dma_addr* and *SKB_dma_addr*, the *Next_SDB_addr* field of SDB is filled with the address of next SDB buffer (step 2). *Driver RX* checks the status field of SDB packet when polling (access packets directly without any interrupts) the linked list. If the status value is zero, the corresponding packet has not been transferred into this buffer yet. In this case, *schedule()* function will be called to schedule other tasks. Otherwise, it means the SKB buffer needs to be processed (step 3). The SKB buffer is delivered into protocol stack or network processing applications(step 4). When the packet processing is done, *Driver TX* delivers the *SDB_dma_addr* to *Txdma* by writing register of NIC directly (step 5). *Txdma* transmits the packet into NIC hardware with *SDB_dma_addr* (step 6). The addresses of *Skbuf* and SDB are reclaimed into *NIC SDB Mgt. module* after transmitting (step

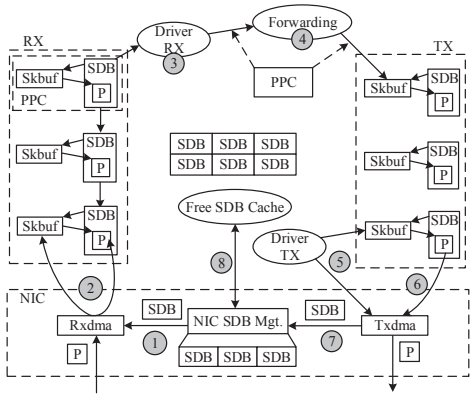


Fig. 6. Processing flow.

7). When the number of SDB buffer in *NIC SDB Mgt. module* is beyond the maximum, the redundant buffers is reclaimed into *Free SDB cache*. On the contrary, when it is insufficient, we allocate SDB buffers from *Free SDB cache* into *NIC SDB Mgt. module* (step 8).

IV. Evaluation

A. Experiment setup

To evaluate the performance of our approach, we have designed a SDB-enabled NIC based on FPGA (Altera EP4SGX180) and its driver. The SDB-enabled NIC is connected with a general computer platform which integrates an Intel i7 Quad-Core processor using the standard PCIE interface. This general computer platform also supports different types of commodity NICs. In the experiment, to compare our approach with traditional SKB mechanisms, a commodity NIC (Intel 82574L with e1000e driver) is deployed on the general computer platform. Ixia tester is used for packet generation and performance measurement.

The software environment of both platforms are the same, which is Ubuntu 14.04 OS with Linux v3.13.11. To analyze the system overhead, such as CPU cycles, cache misses, we use Oprofile to monitor and profile the performance of Linux OS with the performance counters in CPU. Oprofile allows to evaluate the CPU utilization of applications, functions and codes with a very low computational overhead. The detailed system configurations are listed in TABLE I.

TABLE I
System configurations

	Commodity Computer Platform
Processor	Intel i7-4700EQ 4core, 8thread, 2.4Ghz
ICache/DCache	32KB 8-way, 64B cache line
L2 Cache	256KB 8-way, 64B cache line
Main memory	2GB 1333Hz DDR3
NIC	Intel 82574L/SDB-enabled NIC
PCIE	PCIE 2.0 $\times$ 8
OS	Ubuntu 14.04
Kernel	Linux 3.13.11
Driver	e1000e driver/ SDB-enabled NIC driver

B. Benchmarks

We compared traditional SKB mechanism and SDB mechanism in different scenarios. Since SDB-enabled NIC driver is running in polling mode, the mode of e1000e is changed from interrupt to polling for a fair comparison. Moreover, to obtain accurate bandwidth, latency and overhead, all of these experiments are conducted without packet loss by regulating the sending bandwidth.

1) SKB raw forwarding: As the processing of protocol stack degrades the forwarding performance, we consider the SKB raw forwarding scenario firstly. After a received packet is converted into SKB, the transmit function are called by NIC driver directly without other processing. In order to obtain the maximum forwarding performance, we use a single port to receive and send. Moreover, affinity technology can eliminate the switch overhead between different cores efficiently, so we bind the RX and TX descriptor rings to Core 1 in Intel 82574L while bind Port 1 to Core 1 in SDB-enabled NIC.

2) Application forwarding: As aforementioned, SKB is the key data structure between NIC driver and protocol stack. In order to evaluate the performance of network forwarding applications, we designed three scenarios: IPv4 forwarding, Bridge forwarding and SDN forwarding.

a) IPv4 forwarding and Bridge forwarding: IPv4 forwarding and Bridge forwarding are the same in topology and affinity except for some specific configurations. We focus on the optimization of packet I/O, so a simple topology is constructed to reduce the processing overhead of protocol stack. Two ports of Ixia tester connect with two ports of system under test (SUT) respectively with bi-directional flow. In Intel 82574L, we bind the RX and TX rings of Port 1 and Port 2 to Core1 and Core 2 respectively. Similarly, we bind Port 1 and Port 2 of SDB-enabled NIC to Core1 and Core 2 respectively because the receiving and sending of a single port is running in the same thread.

b) SDN forwarding: Software Defined Network (SDN) is an emerging architecture that simplifies network management and improves programmability of network[10]. In order to evaluate the performance in SDN with SDB, we have designed a SDN switch, which is composed of Open vSwitch[11] (version 2.3.1) in software level and general computer platform as hardware platform. In Linux, virtual Ethernet port (*net_device*) is used to exchange packets and states between Open vSwitch and commodity NIC. Therefore, SDB-enabled NIC driver supports the basic functions of *net_device*.

To compare the forwarding performance of SDN switches in different number of rules, we install 1, 256, 8192, 65536 unique rules in Open vSwitch respectively. Packet flows are created in accordance with these rules so that every packet can match a corresponding rule successfully. Moreover, the related parameters of packet flow grows orderly to make sure there are no matched rules remaining in cache.

C. Experimental results

1) SKB raw forwarding: The bandwidth of SKB raw forwarding is shown in Fig. 7. When the packet size is 64B, SDB-enabled NIC can reach line rate while the bandwidth of Intel 82574L is only 380Mb/s. Thus, SDB improves the performance by 100%. The reasons can be explained as

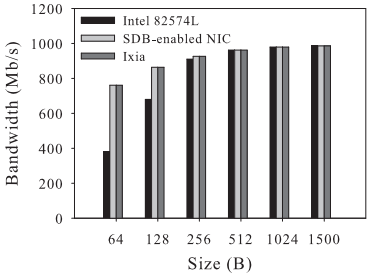
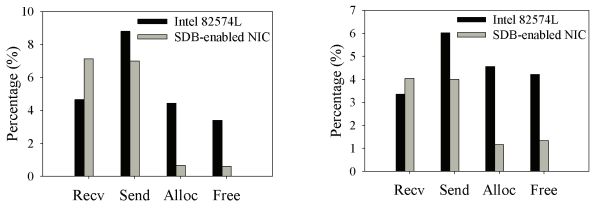


Fig. 7. Bandwidth of SKB raw forwarding.

follows. Firstly, the overhead of SKB management in Intel 82574L is higher than that of SDB-enabled NIC. SDB reduces this overhead via NIC hardware. Secondly, TX rings in e1000e driver are restricted in capacity. In detail, NIC driver needs to determine whether the TX ring is full. If not, the SKB buffers will be enqueued into the TX ring immediately. Otherwise, the received packets will be dropped directly. As the sending bandwidth is much lower than that of receiving bandwidth, the TX ring will be full at the beginning of the transmitting. On the contrary, SDB sends and recycles buffers by writing register directly without TX ring, which reduces the *Send* and *Free* overhead greatly.

2) Application forwarding:

a) IPv4 forwarding and Bridge forwarding: To compare traditional SKB mechanism with SDB mechanism in network forwarding applications, the packet processing flow is divided into four parts: *Recv*, *Send*, *Alloc* and *Free*.

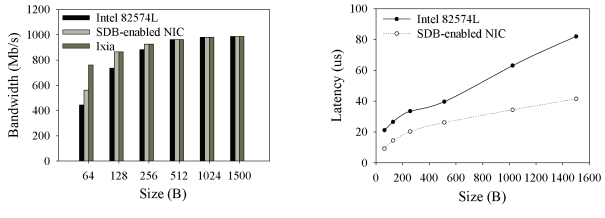


(a) IPv4 forwarding (b) Bridge forwarding

Fig. 8. Overhead breakdown.

Again, Oprofile is used to measure the CPU utilization (overhead) of each part at function level. The result of IPv4 forwarding is shown in Fig. 8a. The total CPU utilization of these four parts is 21.29% in Intel 82574L and 15.4% in SDB-enabled NIC. It means that SDB reduces the CPU utilization by 5.89%. Moreover, it is obvious that the *Alloc* and *Free* overhead in SDB is close to zero. These overheads are caused by other processes in kernel. The *Send* overhead (8.8%) is the highest among these four parts in IPv4 forwarding. The main reason is that *e1000_clean_tx_irq()* consumes a lot of CPU utilization (up to 5.1%). While the overhead of *Send* in SDB-enabled NIC is lower because the packets are sent by writing the register of NIC directly without TX ring.

The results of Bridge forwarding is shown in Fig. 8b. The total CPU utilization of these four parts in Intel 82574L is 18.14% while that in SDB-enabled NIC is 10.54%. So SDB reduces the CPU utilization by 7.6%. The reason why the *Alloc* and *Free* overhead in Bridge forwarding is higher than that in

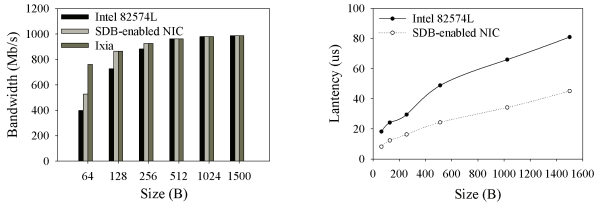


(a) Bandwidth (b) Latency

Fig. 9. Performance of IPv4 forwarding.

IPv4 forwarding is that there are lots of copy operations during packet processing.

The bandwidth and latency of these two NICs in IPv4 forwarding without packet loss are illustrated in Fig. 9. When the packet size is 64B, both of them do not reach line rate in Fig. 9a. The bandwidth of Intel 82574L is 444Mb/s while that of SDB-enabled NIC is 561Mb/s. Therefore, compared with traditional SKB mechanism, SDB increases the performance by 26.35%. SDB-enabled NIC and Intel 82574L reaches line rate bandwidth in 128B and 512B respectively. In Fig. 9b, the latency of Intel 82574L is higher than that of SDB-enabled NIC, and the difference between them enlarges as the packet size increases. This proves that the SDB effectively optimizes the SKB management.



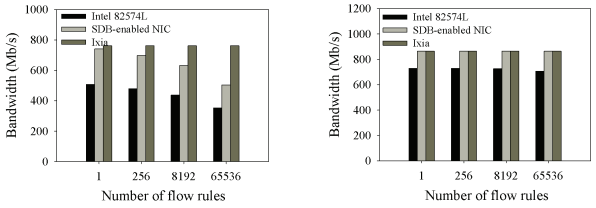
(a) Bandwidth (b) Latency

Fig. 10. Performance of Bridge forwarding.

Similarly, the bandwidth and latency in Bridge forwarding are shown in Fig. 10. When packet size is 64B, the bandwidth of Intel 82574L is 400Mb/s while that of SDB-enabled NIC is 527Mb/s in Fig. 10a. Therefore, compared with traditional mechanism, SDB increases the performance by 31.75%. Since there are lots of copy operations in Bridge forwarding, the bandwidth of IPv4 forwarding is higher than that of Bridge forwarding.

b) SDN forwarding: The forwarding performance of SDN forwarding in different number of rules is shown in Fig. 12. When packet size is 64B, both the bandwidth of SDB-enabled NIC and Intel 82574L decrease as the number of rules increases (see Fig. 11a). When the number of rules is 1 and 65536, the bandwidth of SDB-enabled NIC is 46.15% and 43.59% higher than that of Intel 82574L respectively. When the packet size is 128B, the bandwidth varies little among different rules, SDB-enabled NIC reaches line rate bandwidth, which is 18.5% higher than that of Intel 82574L (Fig. 11b). It is obvious that SDB improves the performance of SDN forwarding greatly.

In conclusion, SDB achieves 2x throughput compared with the traditional SKB mechanism in raw packet forwarding, and



(a) 64B (b) 128B

Fig. 11. Bandwidth of SDN forwarding.

26.35%, 31.75% and 46.15% improvement for IP forwarding, Bridge forwarding and SDN forwarding respectively. Compared with the network processing applications, SDB performs better for raw forwarding applications as we focus on the optimization of packet I/O. In addition, SDB reduces the CPU utilization of IP forwarding and Bridge forwarding by 5.89% and 7.6% respectively.

V. Related work

A wide spectrum of research has been done to optimize the efficiency of packet I/O. The software optimizations can be divided into two categories. One is revolutionary, such as PF_RING DNA[6], netmap[7], Intel DPDK[8], which improves the packet I/O efficiency by optimizing the path of packet I/O based on new lightweight data structure of packet instead of SKB[5]. However, these mechanisms are not compatible with SKB data structure. This means a plenty of open-source network applications developed based on SKB, such as Open vSwitch[11], Click router[12], are hardly ported to the platforms based on these techniques.

The other is evolutionary. To keep the compatibility with SKB, SKB recycling mechanism has been designed to reuse SKBs for the improvement of packet I/O efficiency[9]. After a packet is sent, the corresponding SKB is enqueued into the recycling queue. When allocating new SKBs, it acquires free SKBs from recycling queue to fill the RX ring firstly. Although this mechanism decreased the frequency of slab allocator[13] in Linux OS, the shared resource competition and queue management will cause non-neglect overhead to packet I/O.

Besides, optimization via hardware is a feasible way to accelerate packet I/O[14]. Guangdeng Liao et al. proposed a new server network I/O architecture where descriptor management is transferred from NICs to an on-chip network engine and packets are directly moved into cache[15]. Nevertheless, this technology needs modification of CPU micro-architecture. Sangjin Han et al. designed a GPU-accelerated software router PacketShader[4]. The packet protocol processing functions can be offloaded to GPU for acceleration. However, the programming on this heterogeneous platform is difficult.

VI. Conclusion

Improving the packet I/O efficiency on commodity CPU/OS platform plays a crucial role for network processing to catch up with the ever increasing network speed. In this paper, we identified that the overhead of SKB management is the dominating bottleneck in packet I/O. To reduce the overhead, we proposed a new SKB-compatible mechanism named SDB. It offloads the SKB management into NIC, which eliminates the overhead of SKB allocation/deallocation. In addition, the overhead of binding/unbinding is eliminated by pre-binding the related data structures utilizing the available space of *Databuf*. Experimental results show that SDB improves the packet I/O efficiency in commodity CPU/OS platform and reduces CPU utilization. Moreover, SDB maintains the transparency to various network processing applications due to its compatibility with SKB.

Acknowledgment

This work is supported by National High-tech R&D Program of China (863 Program) (Grant No. 2015AA0156-03), National Natural Science Foundation of China (Grant No. 61202483).

References

[1] D. Tang, Y. Bao, W. Hu, and M. Chen, "DMA cache: Using on-chip storage to architecturally separate I/O data from cpu data for improving I/O performance." in *HPCA*, vol. 10, 2010, pp. 1–12.

[2] A. Bianco, J. M. Finochietto, G. Galante, M. Mellia, and F. Neri, "Open-source PC-based software routers: A viable approach to high-performance packet switching," in *Quality of Service in Multiservice IP Networks*. Springer, 2004, pp. 353–366.

[3] L. Tang, J. Yan, Z. Sun, T. Li, and M. Zhang, "Towards high-performance packet processing on commodity multi-cores: current issues and future directions," *Science China Information Sciences*, vol. 58, no. 12, pp. 1–16, 2015.

[4] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a GPU-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 195–206, 2011.

[5] J. L. Garcıa-Dorado, F. Mata, J. Ramos, P. M. S. del Rıo, V. Moreno, and J. Aracil, "High-performance network traffic processing systems using commodity hardware," *Data Traffic Monitoring and Analysis*, pp. 3–27, 2013.

[6] L. Rizzo, L. Deri, and A. Cardigliano, "10 Gbit/s line rate packet processing using commodity hardware: Survey and new proposals," 2012.

[7] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[8] D. Intel, "Data plane development kit," *URL http://dpdk. org*.

[9] Q. YAO, J. LIU, Z. HAN, and C. SHEN, "Research on linux network packet buffer recycling toward multi-core processor," *Journal on Communications*, vol. 9, p. 018, 2009.

[10] O. N. Fundation, "Software-defined networking: The new norm for networks," *ONF White Paper*, 2012.

[11] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 117–130.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[13] J. Bonwick *et al.*, "The slab allocator: An object-caching kernel memory allocator." in *USENIX summer*, vol. 16. Boston, MA, USA, 1994.

[14] A. Bianco, R. Birke, G. Botto, M. Chiaberge, J. M. Finochietto, G. Galante, M. Mellia, F. Neri, and M. Petracca, "Boosting the performance of PC-based software routers with FPGA-enhanced network interface cards," in *High Performance Switching and Routing, 2006 Workshop on*. IEEE, 2006, pp. 6–pp.

[15] G. Liao, X. Zhu, and L. Bnuyan, "A new server I/O architecture for high speed networks," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 255–265.