Abstract
Currently there is a lot of work going on in the area of linux networking throughput enhancements (recent kernel developments, dpdk, netmap, etc.[1,2,5,6]). In this paper, we focus on linux deployments as routers/firewalls and ideas to improve throughput of linux forwarding path by combining well-known fast packet processing techniques without leaving the kernel space. First we integrated fast packet processing concepts from netmap/dpdk to create a software fastpath and then developed a co-existence model between network stack and fastpath where network stack is responsible for deciding whether to send a particular traffic via fastpath or not. This helped in moving required modules on fastpath incrementally while still using kernel stack for the rest.

Keywords
forwarding stack enhancements, fastpath

Introduction
Frameworks like Netmap and DPDK allows applications to send and receive packets at line rate using a number of fast packet processing techniques mainly I/O batching, Pre-allocated rx/tx buffers, efficient use of L2/L3 cache and memory mapped buffers. However, these frameworks are bypassing kernel stack and implement packet processing in user-space. This might look good for server application(s) but its not a practical choice for routers/firewalls as it requires whole network stack to be written/ported in user space. Besides that, linux already provides a well trusted network stack. Thus the idea about this paper is to integrate and enhance networking stack with the fast packet processing techniques mentioned above. In order to evaluate these techniques, we started comparing network stack with a similar application running on Netmap/DPDK. A linux device was configured as a simple router by keeping only require modules and unloading other modules. When this was compared with a Netmap/DPDK application, there was a good amount of difference between them.

Fastpath Framework
In our experiment, perf showed majority time was consumed in per packet allocation/deallocation and locks in transmit path. So, as a first step, instead of using standard Rx/Tx path, we used Netmap Rings for the same (having pre-allocated tx/rx buffers and batch I/O capabilities). To use the netmap rings, network interface card is required to be put in netmap mode. When NIC is in netmap mode, kernel will see the interface using normal netdevice structure but rx/tx functions are disconnected from network stack.

On Receive side, Netmap framework adds a hook (netmap_rx_irq) in driver’s napi callback that is used to wake up the userspace process to handle incoming packets. The userspace process then (using an ioctl) calls the netmap_rxsync function to receive the packets in the netmap ring. In order to transmit the packet, the application fills netmap’s tx ring and calls netmap_txsync that in turn calls driver specific Tx function.

In the modified approach, instead of waking up the userspace process from napi callback, netmap_rxsync is called to get the packets in netmap ring and packets are processed in kernel space.
Now, for each received packet in the batch, a top level fastpath function \texttt{(do\_fastpath)} is called, which does the routing lookup using packet data for which respective functions of routing code are modified to use packet data instead of skb structure. On Transmit side, only the packet pointers are moved from rx-rings to tx-rings, transmit signals are issued to NIC for batch mode transmit through \texttt{netmap\_txsync} function. Figure 2 shows high level details of the \texttt{do\_fastpath} function.

Inheriting from DPDK
While going through some sample applications of dpdk, we saw the use of forward prefetching. Although standard drivers use prefetch, its use is limited to the current packet. As we have batch of packets in netmap rings, forward prefetching can be used here. That is, we can issue prefetch for packets we are going to process in sometime. For example, when processing the first packet, it issued prefetch for the fourth packet and so on. It has helped significantly on x86 and x86-64 platforms where DDIO\cite{4} is not supported.

Firewall On Fastpath Framework
Like we converted routing code to use packet data instead of skb, connection tracking and NAT code was also modified in the same way.

From \texttt{do\_fastpath} function, conntrack lookup has been done and if it was the first packet of a connection, it was sent to network stack to complete the full journey. CONNMARK target was used in iptables rules to set FAST\_PATH mark to indicate the connection in fastpath. For all subsequent packets of the same connection, FAST\_PATH mark was checked during conntrack lookup and if set, it was transmitted through fastpath.

Eliminating routing lookup cost
Florian Westphal proposed a patch\cite{3} \texttt{[RFC] netfilter: conntrack: cache route for forwarded connections} that caches dst entries in conntrack structure. Using it, we avoided route lookups once we do the connection lookups.

Future Work / Discussion
\begin{itemize}
  \item Packet/Buffer holding support in Netmap rings
  \item Avoid data copies when sending packets to kernel network stack\cite{6}
  \item Fastpath Porting:Bridge,iptables/ipset/nftables,xfrm,QoS
  \item XDP(eXpress Data Path) Possibility
\end{itemize}

Performance numbers
The table 1 shows some performance numbers taken on linux kernel version 3.14. Results were taken on a single core of Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz with two 10G ports connected to ixia Breaking point systems.

Acknowledgments
We would like to thank Jimit Mahadevia for the joint work we did on the ideas and design. We would also like to thank Team at Sophos for initial proof of concept and development.
Figure 3: Firewall Fastpath

Table 1: Performance results

<table>
<thead>
<tr>
<th>Packet size</th>
<th>UDP</th>
<th>Stateful Firewall</th>
<th>Stateful firewall with fastpath</th>
<th>Improvement ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bytes</td>
<td>645 Mbps</td>
<td>1320960 pps</td>
<td>3114 Mbps</td>
<td>6377472 pps</td>
</tr>
<tr>
<td>128 bytes</td>
<td>1155 Mbps</td>
<td>1182720 pps</td>
<td>6240 Mbps</td>
<td>6389760 pps</td>
</tr>
<tr>
<td>256 bytes</td>
<td>2165 Mbps</td>
<td>1108480 pps</td>
<td>11690 Mbps</td>
<td>5985280 pps</td>
</tr>
<tr>
<td>512 bytes</td>
<td>4170 Mbps</td>
<td>1067520 pps</td>
<td>20000 Mbps</td>
<td>5120000 pps</td>
</tr>
</tbody>
</table>

*CPU was idle but 20Gbps test setup limit was reached.

Author(s) Biography(ies)

Nishit Shah & Jagdish Motwani are working as architects in network security group at Sophos mainly focusing on performance and scalability of network stack.

References

[2] Data Plane Development Kit (DPDK), (http://dpdk.org)