Accelerate In-Line Packet Processing Using Fast Queue*

Chun-Ying Huang[1], Chi-Ming Chen[1], Shu-Ping Yu[1], Sheng-Yao Hsu[1], and Chih-Hung Lin[2]
[1] Department of Computer Science and Engineering, National Taiwan Ocean University
[2] Networks and Multimedia Institute, Institute for Information Industry
Email: chuang@ntou.edu.tw, {cmchen,spyu,syhsu}@snsl.cs.ntou.edu.tw, chlin@iii.org.tw

***Abstract*—It is common for network researchers and system developers to run packet processing algorithms on UNIX-like operating systems. For the ease of development, complex packet processing algorithms are often implemented at the user-space level. As a result, performance benchmarks for packet processing algorithms often show a great gap when packets are input from different sources. An algorithm that performs well by reading packets from a raw packet trace file may get a worse result when it reads packets directly from a network interface. Such a phenomenon gets much worse when the algorithm is going to process packets in-line.**

In this paper, we identify the performance bottleneck of existing in-line packet processing implementations in the Linux operating system. Based on the observation, a new software architecture, named *Fast Queue*, is proposed and implemented to show that the identified bottleneck can be effectively eliminated. Experiments show that the proposed software architecture reduces 30% of CPU utilization. In addition, the overall system throughput can be improved by a factor of 1.6 when it is applied to the well-known snort-inline open source intrusion detection system.

***Index Terms*—Fast Queue, in-line packet processing, zero copy interface**

I. Introduction

Emerging network applications and threats make networks harder to be managed well. Undesirable network traffic, including multimedia streams, game traffic, peer-to-peer shares, or network intrusions, often consumes valuable network resources and causes unexpected network damages in commercial networks. Therefore, it is important to properly manage and filter network flows transmitted in a network. In the past, flows are usually managed by enforcing policies based on the so called five tuples, i.e., the source IP address, the source port, the destination IP address, the destination port, and the transport layer protocol.However, modern network applications often bypass these policies by using randomized port numbers. As a result, it is a must to develop advanced packet processing algorithms to handle these mutated network applications.

Packet processing algorithms vary on complexities. A traffic classification algorithm can be easily done by examining only the first n payload bytes in a packet [11], [4]. However, it can be much more complex because classifying a flow often requires a number of preprocessing steps before a decision can be made. For example, intrusion detection systems [10], [9] use algorithms to handle fragmented packets, data compressions, obfuscated content encodings, and pattern matchings. A WAN optimization system [2] use algorithms to handle cache management, packet coalescing, compression, and decompression. When the complexity of a packet processing algorithm increases, it would be much easier for developers to implement the algorithm at the user-space level instead of the kernel-space level.

Implementing complex packet processing algorithms at the user-space level has two major benefits. One benefit is the ease of implementation. Developers are not required to understand the underlying kernel details to store and process packets. Another benefit is the confinement of coding errors. If an implemented algorithm is crashed, the fault can be confined in the user-space without affecting other parts of the OS kernel. In addition, testing and debugging at the user-space level is also much easier. However, performance penalties make it impractical to implement packet processing algorithms at the user-space level. Figure 1 shows the extra costs brought by a user-space implementation of a packet processing algorithm. The benchmark is done with a Linux operating system running on a Pentium-III 1000Mhz CPU. The implemented packet processing algorithm does nothing. It simply intercepts packets forwarded by the operating system and put them back to the system, both at the user-space level. The overall throughput is 120Mbps and the CPU utilization is almost 100% during the packet forwarding process. From the figure, we can see that most of the CPU resources are consumed by accessing packets from the OS kernel and issuing software interrupts.

There are already zero copy solutions that reduce the extra cost brought by moving data between the user-space and the kernel-space [1], [3], [5], [7], [6]. However, they do not completely solve the problem. As we can see in Figure 1, in addition to data movement, a great portion of CPU resources is consumed by software interrupts, which are issued twice for each packet. Thus, we need a new architecture to reduce *both extra overheads incurred by data movements and software interrupts*. In this paper, we propose Fast Queue to improve the performance of in-line packet processing. By using memory mapped ring buffers and high-resolution timers, a user-space packet processing algorithm is able to access packets from

*This work was supported in part by National Science Council under the grant number NSC 97-2218-E-019-004-MY2 and by Taiwan Information Security Center at NTUST(TWISC@NTUST) under the grant number NSC 99-2219-E-011-004.

Function name	CPU usage
skb_copy_bits	33%
kfree	3.27%
copy_to_user	2.99%
__alloc_skb	2.57%
handle_IRQ_event	2.35%

Fig. 1. Sampled additional costs incurred by kernel-user space interactions.

the operating system with a low overhead interface and hence improves the overall performance.

The rest of the paper is organized as follows. In Section II, previous researches related to the proposed solution are reviewed and discussed. The proposed solution and a reference implementation are introduced in Section III. With the implementation, we evaluate the performance improvement in Section IV by using two different in-line packet processing algorithms. Finally, a concluding remark is given in Section V.

II. Related Work

Many researches and implementations have focused on eliminating the extra overheads brought by user- and kernel-space interaction. A number of works targeted on the acceleration of socket programming interface. Chu [1] proposed a zero-copy TCP socket and implemented it on the Solaris operating system. In addition, this work classifies existing solutions into four different models, as follows:

1) User accessible interface memory
2) Kernel-network shared memory
3) User-kernel shared memory, and
4) User-kernel page remapping with copy-on-write (COW)

Each model has its own advantages and disadvantages. Chu's and a latter implementation on FreeBSD [3] both use the fourth model, i.e., user-kernel page remapping with COW. With this model, data sent by a user is directly transferred from the user's buffer to the network interface via DMA and vice versa. No CPU interaction is required. However, all involved buffers must align on page boundaries and occupy an integral number of MMU pages. This is not a problem when sending data since fragmented user buffer can be transmitted using CPU copy. However, a programmer has to avoid overwriting buffers that have been written to the socket but not yet freed by the kernel. To receive data correctly, the data must be at least a page in size and page aligned in order to be mapped into the kernel. Therefore, network interface drivers must arrange receive buffers in such a way that, after DMA, user payload shows up on a page boundary in the buffer. These limitations increase difficulties for programmers to manage buffers properly, restrict the size of MTU, and require supports provided by network interface hardware.

Maltz et al. [8] proposed a solution to improve the performance of application proxy servers implemented at the user-space level. The authors add several new `ioctl` commands that are able to "splice" two established TCP connections at the kernel-space level. When two TCP connections are spliced, data received by one connection is immediately forwarded to the other connection and vice versa. The motivation behind the solution is straightforward. The authors observe that the major work of an application proxy server is to forward data between the two network connections associated by the proxy. Even if a proxy server has to examine the content transmitted in a forwarded network connection, it is often done only for the very first bytes of the connection data stream. Hence, the data forwarding job can be moved to the kernel instead of staying at the user-space level. The evaluation shows that the data forwarding throughput for spliced TCP connections is almost equivalent to IP packet forwarding. However, when two TCP connections are spliced, the user-space proxy server is no longer able to know what is being forwarded by the kernel.

There are some other solutions to improve the performance of network services. The `sendfile` system call [7], [5] is able to reduce the cost of sending file content through the network. Traditionally, a network server has to iteratively read each file data block into a user space buffer and then write the buffered block to a opened network socket. With `sendfile`, the network server is able to do the same job by binding the descriptors of the file and the network socket, and the kernel does the rest for the server. Consequently, the two extra data movements between the user- and the kernel-space for each buffered data block can be eliminated. The Linux kernel also provides a performance improved implementation to reduce the cost of moving data from the kernel- to the user-space when sniffing packets directly from the network interface. In the past, programmers use the `PACKET_SOCKET` interface [6] to create a descriptor and then read packets by the `read` system call. Now, the `PACKET_SOCKET` interface supports memory mapped operations. To use the memory mapped technique, a proper size of a framed ring buffer must be initialized first to receive packets copied from network interfaces. The ring buffer is then shared by the user- and the kernel-space codes. On receipt of a packet by the kernel, the packet is placed in the current available frame in the ring buffer, mark the frame as occupied, and signals the user space program to process packet frames. Packets are dropped if no frame is available. Once the user space program has processed a packet, the corresponding frame is then marked as available and thus the kernel is able to continue receiving more packets. The famous `pcap` packet capture library now also leverages the new memory mapped interface to improve its performance.

Although a number of solutions are able to improve the performance of socket operations and packet capturing, they are not enough for in-line packet processing at the user-space level. There is one fundamental difference for in-line packet processing. Compare with the above techniques, the user-space program is not an end point of a packet. When an intercepted packet has been processed, it must be re-injected into the kernel as soon as possible. In addition, the number of user-kernel interactions for each packet must be reduced because the number of packets can be huge in a busy network. Therefore, care must be taken when designing the solution.

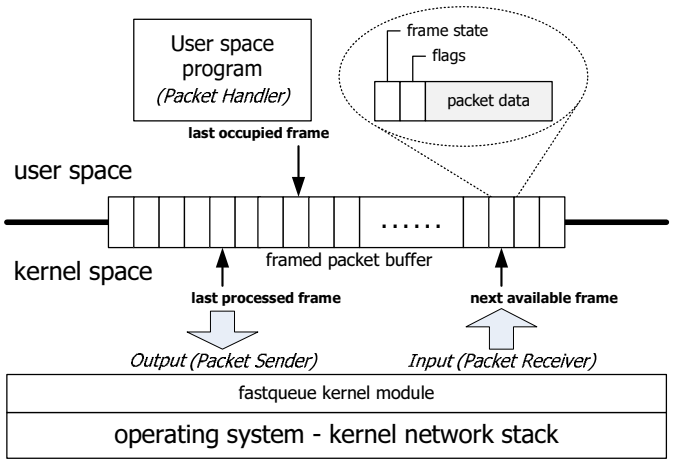

Fig. 2. The architecture of the proposed solution.

III. The Proposed Solution

A. Architecture

The goal of the proposed solution is simple, i.e., improve the in-line packet processing performance. Modern UNIX-like operating systems have their own interfaces to intercept network packets. For example, the Linux operating system has the netfilter-queue and the FreeBSD operating system has the divert socket. However, they are not efficient enough especially when they are running on low computation power devices such as embedded systems. The proposed solution improves the overall system performance by eliminating frequently used user-kernel interactions. It follows the "user-kernel shared memory" model to reduce the cost of moving packets between the user- and the kernel-space level. The proposed system architecture is depicted in Figure 2. There are three roles in the architecture. The packet receiver and the packet sender are both implemented as parts of the kernel. On the contrast, the packet handler, which implements the packet processing algorithm, is implemented as a user-space program. All the three roles share the same memory area in the kernel, which is configurable by the user space program. The shared memory is actually a framed ring buffer. Each frame has a frame state, which can be one of the below three states:

- *Available*: A frame of this state means that it is vacancy and is ready to receive packets. When a shared memory space is just created, all frames' states are reset to *available*.
- *Occupied*: If a packet has been placed in a vacancy frame, the frame state is changed to the *occupied* state. The received packet then waits for the user-space program to process it. A notification signal is also sent to a user-space program if the program is waiting for incoming packets.
- *Processed*: If a frame state is set to this state, it means that the packet stored in the frame should be re-injected into the network. A frame is usually set to the *processed* state right after it has been processed by the user space program. When a frame is set to this state, an additional flag must be set to tell the kernel how to handle the re-injected packet, i.e., *accept* or *drop* the packet.

```
/* An infinite loop                                    */
1 while true do
2     while the FO pointer points to an occupied frame do
3         Process the frame;
4         Set the frame flag to either accept or drop;
5         Set the frame state to processed;
6         Move the FO pointer to the next frame;
7     Call poll or select to wait for incoming packets –
      Enter sleeping state;
```

Fig. 3. The pseudo-code for the packet handler. This piece of codes is actually packet-driven since it is waken up only on receipt of packets.

To manage the ring buffer properly, three pointers are used to indicate the correct positions to access, as shown in Figure 2. The three pointers indicate the position of 1) *the next available frame (FA)*, 2) *the last occupied frame (FO)*, and 3) *the last processed frame (FP)*. A pointer is used only by one of the three roles. For example, the packet receiver uses the FA pointer to find the first available frame in the ring buffer; the packet handler uses the FO pointer to find the next to-be-processed frame; and the packet sender uses the FP pointer to find the next to-be-sent frame. At the system initialization phase, all the pointers point to the first frame in the ring buffer. A pointer is moved one frame forward if the corresponding role has finished processing a packet. With these pointers, a proper frame can be accessed in a constant time.

Readers should notice that one Fast Queue can be used by only one user space program. That is, all queued packets are processed by the same packet processing algorithm. However, this limitation can be eliminated easily by creating multiple Fast Queues. Please refer to Section III-C for the details.

B. Algorithms

The three roles mentioned in Section III-A are driven by different manners. It is naïve that the packet receiver can be driven by incoming packets, which are triggered on receipt of network packets by a network interface. Similar to the packet receiver, the packet handler can be also driven by incoming packets. The packet handler enters a sleeping state by using system calls such as `poll` or `select`. Then, it can be waken up as well when an incoming packet is queued. The algorithms for the packet handler and the packet receiver are depicted in Figure 3 and Figure 4, respectively.

Compare with the packet receiver and the packet handler, the design of the packet sender is a little bit different. This is because when a packet has been processed by the packet handler, only the state and the additional flag of the processed frame is affected. Neither interrupts nor events are generated for memory access operations. Therefore, instead of a packet-driven design, the packet sender is executed periodically. During the execution, it checks the ring buffer to see whether

Input: *pkt* - the received packet.

```
/* On receipt of a packet                    */
if the FA pointer points to an available frame then
    Place pkt in the frame;
    Set the frame state to occupied;
    Move the FA pointer to the next frame;
    Wake up the sleeping packet handler;
else
    /* Take the default action               */
    Re-inject te packet into the kernel network stack: Ask
    the kernel to drop or accept the packet based on the
    default action;
```

Fig. 4. The pseudo-code for the packet receiver. This is a interrupt handler registered to handle incoming packets received by the network interface card.

```
/* An infinite loop                          */
while true do
    while the FP pointer points to a processed frame do
        Re-inject the packet into the kernel network
        stack: Ask the kernel to drop or accept the packet
        based on the frame flag;
        Set the frame state to available;
        Move the FP pointer to the next frame;
    Sleep for a fixed period of time;
```

Fig. 5. The pseudo-code for the packet sender.

the FP pointer points to a processed packet. Once a processed packet is found, the packet is re-inject to the kernel. The algorithm of the packet sender is depicted in Figure 5.

C. Implementation

The proposed solution has been implemented on a Linux operating system. It is implemented as a kernel module hooking on the built-in netfilter firewall. There are several benefits to hook the Fast Queue kernel module on the built-in netfilter firewall. First, it is able to leverage netfilter packet filtering rules to filter out packets that are not required to be processed by the packet processing algorithm. Second, it is also easier to create multiple Fast Queues and then feed packets to the queues based on packet tags or packet filtering rules. Queues with different priorities therefore provide different level of QoS capabilities. Third, since the Linux netfilter firewall is able to intercept packets at different places in the operating system kernel, a packet processing algorithm is hence able to choose a proper place, e.g., incoming, outgoing, or forwarding, to process packets. These benefits make life easier for programmers and researchers since they can focus only on the design of the packet filtering algorithm instead of worrying about where, when, and how to intercept network packets.

Our implementation can be discussed in two parts. For the user-space part, a character device node placed in `/dev/fastqueue` is registered as the interface between the user- and the kernel-space. Before reading packets from the shared memory, a user-space application must run the following three initialization steps:

1) Open the `/dev/fastqueue` device.
2) Use the `ioctl` system call to allocate a fixed size memory as the shared ring buffer.
3) Use the `mmap` system call to map the kernel-space ring buffer into user-space address.

Then, the user-space program is able to read packet data from the ring buffer, process it, and then pass the processed result via the same buffer.

For the kernel-space part, the implemented module use the `nf_register_queue_handler` to register a call-back function for intercepting packets from netfilter firewall packet queue at the initialization phase. On receipt of a packet, it is placed in the previously allocated ring buffer. If the ring buffer has not been allocated or is full, a received packet is dropped or accepted by the kernel according to default policies. At the initialization phase, the module also register a timer handler, which is used to trigger the packet sender periodically so that processed packets can be finally re-injected into the kernel network stack. To improve the processing efficiency, we use a high-resolution timer of 1000HZ to check the availability of processed packets. Modern hardware already supports high resolution timers, which is capable of providing an extreme high clock tick frequency up to 1GHZ. However, it is harmful to system performance if a extremely high clock tick rate is used to trigger kernel functions.

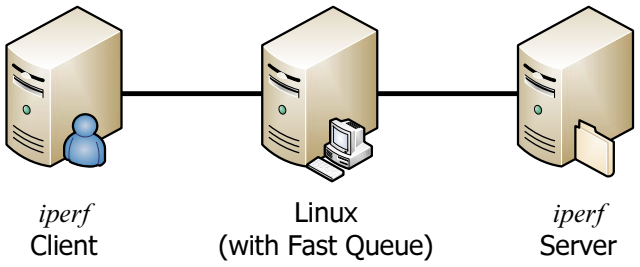

Fig. 6. The benchmark environment.

IV. Evaluation

To evaluate the proposed solution, we use two different packet processing algorithm to benchmark the performance. One is the `NULL` packet processing algorithm and another is the well-known open source snort intrusion detection system. Readers should notice that the `NULL` packet processing algorithm actually does nothing. It simply intercepts packets from the operating system and then ask the kernel to accept the intercepted packets immediately. The use of the `NULL` algorithm is to show the effectiveness on reduction of the overall CPU utilization. The benchmark environment is shown in Figure 6, the proposed solution and the packet processing algorithms are run on the middle device. The left side and the right side device runs the *iperf* performance benchmark software client and server, respectively. The TCP throughput and CPU utilization is measured to see the effectiveness of the proposed architecture. For hardware configurations, all network interfaces installed on the devices are gigabit

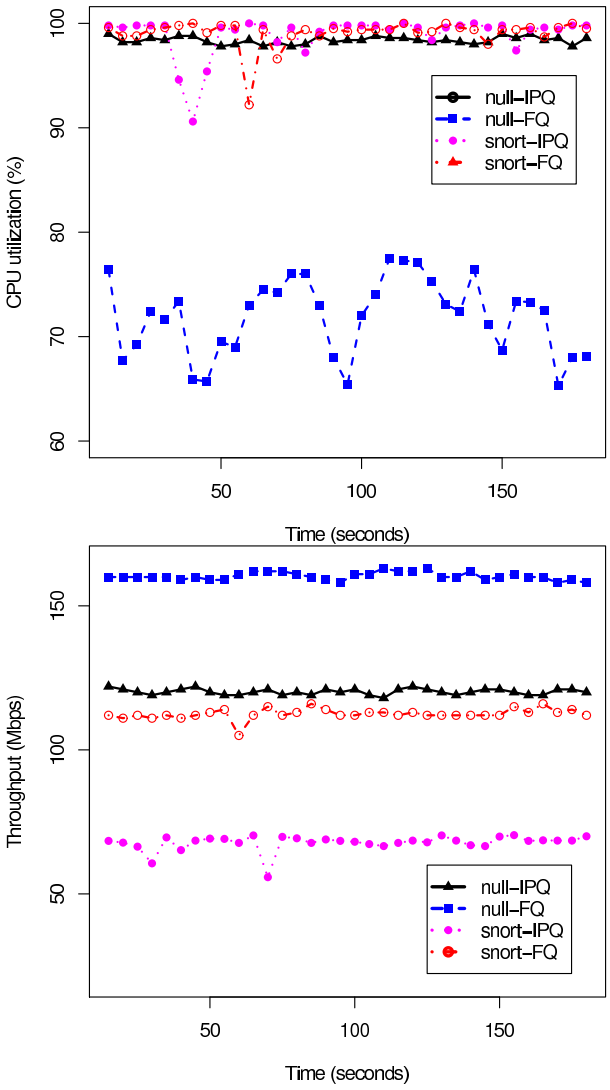

Fig. 7. CPU utilization (upper) and system throughputs (lower) benchmarked on a P-III 1G system.

network interfaces. The middle device is an Intel Pentium-III 1GHZ machine and the left and the right device are virtual machines running on an Intel Core 2 Duo 2.4GHZ machine. For software configurations, the ring buffer size and the timer tick frequency set on the middle device is 2048 packets and 1000HZ, respectively. The client side *iperf* commands used to benchmark for the TCP performance are shown below.

```
iperf --client $SERVER_IP \
      --time 180 --interval 5
```

Figure 7 shows the performance benchmark result for the proposed Fast Queue (FQ) and the system default implementation Netfilter IP queue (IPQ). We find that all the CPU resources are depleted by the packet forwarding process except the NULL algorithm that intercepting network packets using Fast Queue. Compare with the built-in queueing mechanism, we can see that Fast Queue reduces more than 30% of CPU resources on average. From the benchmarked utilization results, the performance gap can be easily identified. From the figure, the overall throughput of the snort intrusion detection system has been improved by a factor of 1.6, i.e., improved from an average of 70Mbps to 112Mbps.s

V. Conclusion and Future Work

In this paper, a software-based high performance packet queueing mechanism is proposed for packet processing algorithms implemented at the user-space level. It is also implemented on the Linux operating system to show its effectiveness. Benchmark results show that the proposed solution effectively improves the system performance in terms of both CPU utilization and system throughput. We believe that the proposed solution is beneficial for many existing embedded platforms. It brings performance boosts without changing the hardware. Although the preliminary implementation already shows a great improvement on the performance, there are still some future works can be done. We would like to further analyze and model the performance of the proposed solution to find out the proper configuration, i.e., ring buffer size and clock tick frequency, to match the network performance requirements. In addition, it is also worth to discuss how multiple queues affect the system performance and what level of QoS capabilities can be provided with multiple queues.

VI. Acknowledgement

This work was conducted under the “Next Generation Security Technology Deployment and Enablement Project” of Institute for Information Industry which is subsidized by the Ministry of Economy Affairs of the Republic of China. We also thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] H.-K. J. Chu. Zero-copy TCP in Solaris. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, Jan. 1996. USENIX Association.

[2] N. Conner. *WAN Optimization for Dummies*. Wiley Publishing, Inc., May 2009.

[3] A. Gallatin and K. Merry. zero_copy, zero_copy_sockets — zero copy sockets code. [online] http://www.freebsd.org/cgi/man.cgi?query=zero_copy&sektion=9.

[4] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: multilevel traffic classification in the dark. In *Proceedings of the SIGCOMM 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 229–240, New York, NY, USA, 2005. ACM.

[5] D. G. Lawrence. sendfile - send a file to a socket. [online] http://www.freebsd.org/cgi/man.cgi?query=sendfile&sektion=2.

[6] Linux man-pages project. packet, AF_PACKET - packet interface on device level. [online] http://www.kernel.org/doc/man-pages/online/pages/man7/packet.7.html.

[7] Linux man-pages project. sendfile - transfer data between file descriptors. [online] http://www.kernel.org/doc/man-pages/online/pages/man2/sendfile.2.html.

[8] D. A. Maltz and P. Bhagwat. TCP splice for application layer proxy performance. *Journal of High Speed Networks*, 8(3):225–240, 1999.

[9] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998.

[10] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX LISA Conference*, pages 229–238, Nov. 1999.

[11] M. Strait and E. Sommer. Application layer packet classifier for linux. [online] http://l7-filter.sourceforge.net/.