

CESNET Technical Report 22/2008

Packet Capture Benchmark on 1 GE

TOMÁŠ MRÁZEK, JAN VYKOPAL

Received 12.12.2008

Abstract

This report describes packet capture benchmark test and its results. We compared packet capture throughput of the standard Linux TCP/IP stack and *PF_RING* with common Intel NIC and *szedata2* with COMBO cards developed by the *Liberouter* project. The throughput is measured according RFC 2544 and by simpler but widespread “iperf-like” technique on 1 Gigabit Ethernet.

Keywords: packet capture, benchmark, PF_RING, szedata2, COMBO6X, Intel, Linux

1 Introduction

There are many application requiring reliable packet capture at gigabit speeds, for example network monitoring tools or intrusion detection and prevention systems. The most common way of packet capture is via the standard Linux TCP/IP stack. The next method, *PF_RING* [1], bypasses the TCP/IP stack and thus speeds up packet capture. The last mentioned method is *szedata2* [2]. This hardware-independent method is believed to provide the highest performance in connection with hardware-accelerated COMBO¹ cards. A common multiport NIC (Network Interface Card) is one of applications of COMBO cards.

Figure 1 depicts these three approaches in the host system architecture.

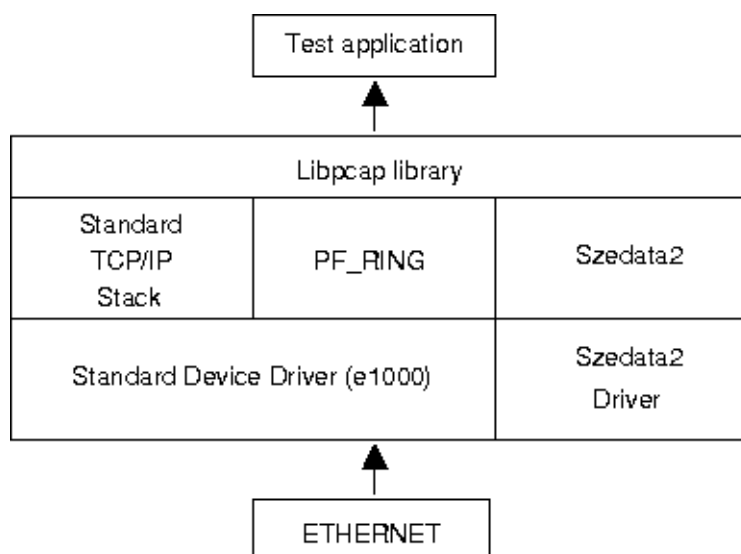


Figure 1. All three tested approaches and host system architecture.

¹ <http://www.liberouter.org/hardware.php>

If we use the first method, the incoming packets are forwarded to the operating system. The kernel handles the traffic and the frames go through the TCP/IP stack to the running application via `libpcap`² library. This library is widely used, e. g., in *tcpdump*, *Wireshark*, *Snort*, *fprobe* and many others applications are based on this library. Unfortunately, the performance is slowed down by large amount of CPU interrupts during handling the traffic.

PF_RING copies all incoming packets into the circle buffer and discards the original ones. If the buffer is full, the packets are discarded immediately without being copied. The copied packets are not forwarded to the higher layers of the TCP/IP stack but they stay in the buffer. If the application need to read the packets, it has to call function *mmap()* to get a pointer to the circle buffer. Then it can read them. There are two independent pointers for reading and writing data. Newly incoming packets cause rewriting the old values.

The last method is called *szedata2*. It derives benefit from special card of the *COMBO* family. Similarly to *PF_RING*, the main advantage of this approach is bypassing the TCP/IP stack. *Szedata2* should provide high throughput thanks to aggregation of short packets into bigger blocks. This approach lowers the overhead for DMA transmissions. The principle of transmitting packets is based on swapping of these packets between two circle buffers. The first one is in RAM and the second one in the network interface card. DMA controllers handles the proper usage of pointers for flawless communication of buffers. The size of buffer can vary (hundreds of MB by default). The buffer is divided into 4kB blocks that corresponds to the size of a page in the most of recent operating systems.

2 Test Description

We tested packet capture performed by the mentioned techniques to benchmark their performance, namely throughput. To ensure test objectivity we put a lot of effort on a configuration of a test application and a *device under test* (DUT in short). DUT stands for the network cards in a host computer and the operating system. Following sections of the report focus on every part of preparations and testing itself in detail.

There are two main methods how to measure throughput. The first one is defined in RFC 2544 [3] as “*the fastest rate at which the count of test frames transmitted by the DUT is equal to the number of test frames sent to it by the test equipment.*”

The test application sends traffic from the hardware test tool Spirent TestCentre 2000³ (TC2000 in short) at wire speed on 1 Gigabit Ethernet. The RFC 2544 defines test frame lengths contained in test traffic as follows: 64, 128, 256, 512, 1024, 1280 and 1518 B. In addition, we also randomly chose these frame lengths: 73, 207, 313, 726, 1174 and 1319 B.

According the RFC 2544, the time of sending can vary but the default value is 60 seconds. The test application uses *binary searching algorithm* that means halving intervals corresponding to the result of the last iteration. The test starts at wire

² <http://www.tcpdump.org/>

³ <http://www.spirent.com/portal/index.cfm?ws=356&media=8>

speed. If there is a packet loss, it lowers the speed by the defined step. If not, it raises the speed. The packet loss is the difference between sent packets from *TC2000* and received packets by the test application. The step depends on the difference between the two last speeds. The end of the test is set by the precision of this step. For example, the value of 1 % means that if the step is lower than 1 % of bandwidth the application ends and prints the results.

The second method of measuring the throughput is simpler than the previous one. We call it “iperf-like” method. The same traffic is continuously sent at wire speed and the test application captures this traffic and it prints instant throughput value every second. After defined time traffic and the application is stopped. The application counts the throughput by dividing the received packets by total time of sending. We chose this method because it corresponds to the networking best effort policy. It is similar to the *frame loss rate* defined in RFC 1242 [4] as “percentage of frames that should have been forwarded by a network device under steady state (constant) load that were not forwarded due to lack of resources.”

These two methods are different so the results can also be different. The measuring method defined in RFC is more strict to packet loss.

Another problem that could occur during testing is a *saturation*. It can happen after sending the traffic at the speed when the DUT is unable to receive all packets. In that case, following behaviour of DUT is unpredictable and the test results are distorted.

3 Test Bed

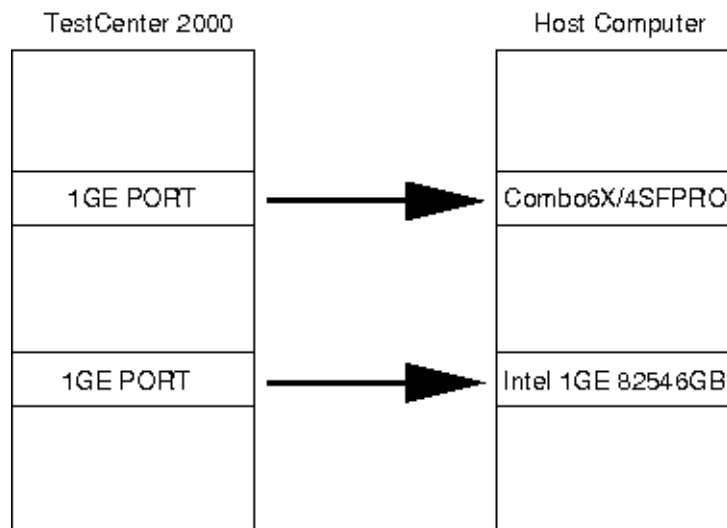


Figure 2. The test bed.

We chose *TC2000* as a lossless generator of synthetic traffic at wire speed (1 Gbps). It is capable of sending various types of traffic: frames with fixed, random or incremental length and *IMIX* which can correspond to common Internet traffic.

Technical parameters of the host computer are described in Appendix A.

We tested Intel 82546GB⁴ and COMBO6X/4SFPRO⁵ network interface cards. Both cards were plugged into PCI-X slot in the host machine and connected to *TC2000* by one port. The first two packet capture methods, the standard Linux TCP/IP stack and *PF_RING*, were tested with Intel NIC and the third method, *szedata2*, with COMBO6X card.

We believe the following settings should help to increase the packet capture throughput:

- Enabled kernel compile parameters *RX polling (NAPI)* and *mmapped IO*.
- Disabled CRC check in case of the packet capture via standard TCP/IP stack.
- Tuning e1000⁶ driver parameters such as *InterruptThrottleRate* and *RxIntDelay*.
- Disable useless services in operating system.
- Balance load among several CPU cores by *taskset*.

4 Test Application

We developed a test application *counter* for measuring the throughput. There are three “versions”. The only difference between them was a compilation process. We used standard *libpcap v0.9.7* for TCP/IP stack testing, patched *libpcap v0.9.7* with *PF_RING* and patched *libpcap v0.9.8* with *szedata2*. The differences between version 0.9.7 and version 0.9.8 are irrelevant to our testing and thus it should not influence the performance.

The test application works as follows. From the beginning it waits in poll. After the first burst of received packets, the test application starts to print instant throughput to the output. In case of measuring the throughput according to *RFC 2544*, the continuous output is suppressed. For capturing a packet we can use two functions: *pcap_loop()* or *pcap_next()*. We decided for *pcap_next()* because it should provide higher performance or at least the same as *pcap_loop()*.

Next, received packets are discarded at *libpcap* level to provide more CPU time for handling interrupts and thus to increase the throughput. We can stop the test application at any time by sending *SIGINT* signal.

For more details see the source code in Appendix B.

5 Settings and Test Results

First, we measured iperf-like throughput. We also monitored the CPU load caused by the test application by Linux function *time* and kernel from *top* output (the value was only approximate and differed slightly during the tests). Table 1 depicts the results for *szedata2*.

⁴ <http://developer.intel.com/design/network/products/lan/controllers/82546gb.htm>

⁵ <http://www.liberouter.org/hardware.php>

⁶ <http://www.intel.com/support/network/sb/cs-009209.htm>

Table 1. Iperf-like throughput of *szedata2* with COMBO6X NIC.

Frame Length [B]	Throughput [pps / Gbps]	App./kernel load [%]
1518	81275 / 1.0	11.5 / 0
1319	93353 / 1.0	12.3 / 0
1280	96154 / 1.0	12.4 / 0
1174	104690 / 1.0	12.4 / 0
1024	119732 / 1.0	12.8 / 0
726	167560 / 1.0	14.2 / 0
512	234963 / 1.0	17.4 / 0
313	375375 / 1.0	22.1 / 0
256	452899 / 1.0	25.8 / 0
207	550066 / 1.0	29.4 / 0
128	844595 / 1.0	41.7 / 0
73	1344086 / 1.0	57.1 / 0
64	1488095 / 1.0	63.6 / 0

It is obvious that *szedata2* has the throughput equal to 1 Gbps (theoretical maximum throughput) at all measured lengths. The CPU load caused by the test application was higher for the shortest packets than for the longest ones. It is not caused by *szedata2* itself but by the test application which reads continuously timestamps from each incoming packet to be able to print instant throughput. Without reading this information the application load is about 6 % for all frame lengths. We also measured the load caused by the kernel and for *szedata2* it was 0 %.

Packet capture via next two methods, *PF_RING* and standard stack, achieved different throughput depending on the CPU on which the test application was running. *Throughput #1* denotes result of a measurement when the test application was running on the same CPU as the CPU which handles the incoming traffic. But in case of *Throughput #2* we assigned a different CPU to the test application. The load #1 corresponds with Throughput #1 and load #2 with Throughput #2 of course.

We were also able to change the parameters of the network driver *e1000*. We tried many combinations of *InterruptThrottleRate* and *RxIntDelay* parameters but none of them had a significant influence (more than several percent) on throughput. The reported results were measured with default settings of *e1000* driver (*InterruptThrottleRate* = 8000, *RxIntDelay* = 0).

Tables 2 and 3 show the results for *PF_RING* and the standard Linux TCP/IP stack, respectively.

PF_RING provide slightly better results and depends on chosen CPUs. Both methods achieved throughput of 1 Gbps at the frame length equal to and greater than 512 B. At length 313 B *PF_RING* achieved only 0.9 Gbps in case we used different CPUs. However, it captured all the packets if only one CPU was utilized. The same behaviour for *PF_RING* was observed for the lengths of 256 and 207 B.

The standard stack was unable to capture all the packets with using the same CPU nor with different ones for handling incoming traffic. *PF_RING* was also unable to capture all the packets for lengths shorter than 207 B where the better performance was measured by using different CPUs.

Table 2. Iperf-like throughput of *PF_RING* with Intel NIC.

Frame Length [B]	Throughput [pps / Gbps]		App./kernel load [%]	
	#1	#2	#1	#2
1518	81274 / 1.0	81279 / 1.0	9 / 1	23 / 1
1319	93359 / 1.0	93359 / 1.0	10 / 1	25 / 1
1280	96160 / 1.0	96160 / 1.0	9 / 1	26 / 1
1174	104697 / 1.0	104697 / 1.0	13 / 1	27 / 1
1024	119739 / 1.0	119739 / 1.0	12 / 1	30 / 1
726	167571 / 1.0	167571 / 1.0	15 / 1	38 / 1
512	234978 / 1.0	234978 / 1.0	17 / 1	50 / 1
313	375375 / 1.0	338143 / 0.9	30 / 8	68 / 100
256	452928 / 1.0	343014 / 0.757	42 / 15	72 / 100
207	550691 / 1.0	360587 / 0.655	81 / 50	74 / 100
128	288621 / 0.342	379419 / 0.449	99.9 / 85	78 / 100
73	142171 / 0.106	401708 / 0.3	23 / 78	78 / 100
64	150915 / 0.101	404517 / 0.272	23 / 78	77 / 100

Table 3. Iperf-like throughput of the Linux TCP/IP stack with Intel NIC.

Frame Length [B]	Throughput [pps / Gbps]		App./kernel load [%]	
	#1	#2	#1	#2
1518	81276 / 1.0	81276 / 1.0	19 / 1	31 / 1
1319	93356 / 1.0	93357 / 1.0	21 / 1	34 / 1
1280	96158 / 1.0	96158 / 1.0	21 / 1	35 / 1
1174	104695 / 1.0	104695 / 1.0	23 / 1	37 / 1
1024	119736 / 1.0	119737 / 1.0	25 / 1	40 / 1
726	167571 / 1.0	167571 / 1.0	33 / 1	54 / 1
512	234964 / 1.0	234971 / 1.0	44 / 2	72 / 2
313	375389 / 1.0	375370 / 1.0	79 / 15	99.9 / 3
256	401845 / 0.887	370045 / 0.817	99.9 / 35	99.9 / 5
207	386925 / 0.703	386306 / 0.702	99.9 / 40	99.9 / 8
128	299121 / 0.354	363735 / 0.431	99.9 / 100	99.9 / 100
73	136142 / 0.101	383673 / 0.285	99.9 / 100	99.9 / 100
64	86989 / 0.058	354233 / 0.238	99.9 / 90	99.9 / 100

Generally, the CPU load corresponds with measured throughput. Note that there is an interesting anomaly in case of load generated by *PF_RING* at length 128 B. We inspected neighbouring frame lengths and found significant difference between lengths of 104 B and 103 B where the load generated by the test application dropped from 89 % to 23 %.

For a better comparison we plotted Figures 3, 4, 5 and 6. Figure 7 and 8 represent kernel and test application load as a function of frame lengths.

Next, we performed the throughput tests defined by *RFC 2544*. The time of each iteration was set to 60 seconds. The results are shown in Tables 4, 5 and 6.

Again, only *szedata2* proved to be absolutely lossless at all tested packet lengths. *PF_RING* also proved to be more efficient than standard TCP/IP stack. From

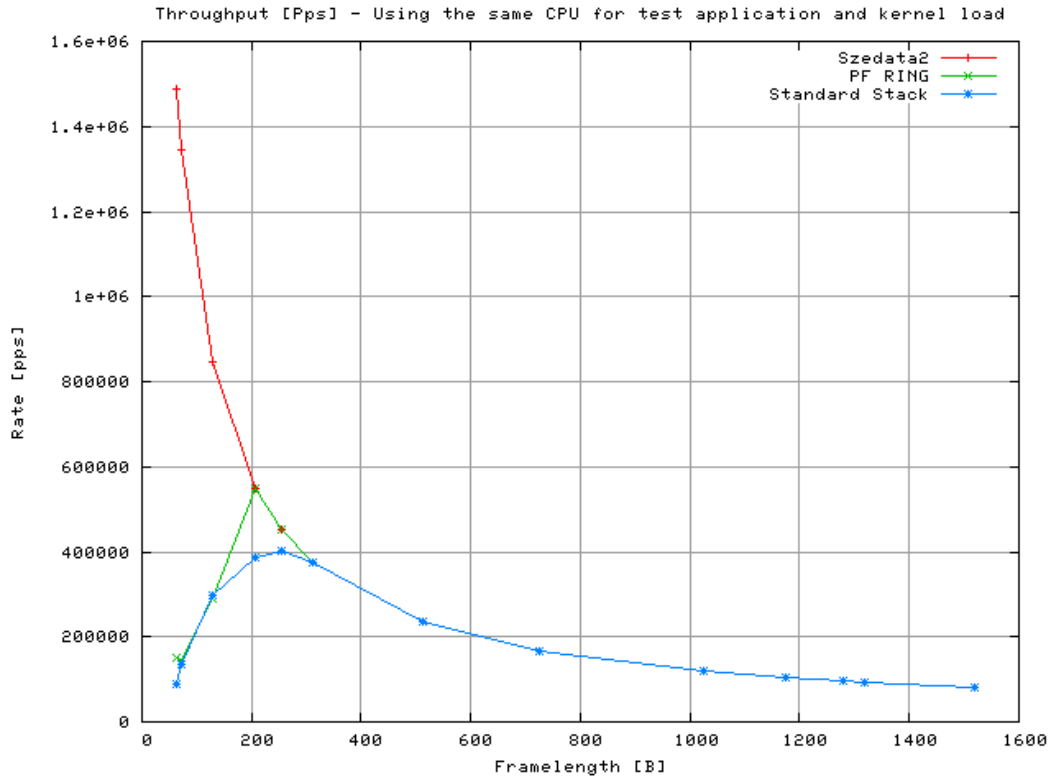


Figure 3. Iperf-like packet capture throughput #1 of *szedata2*, *PF_RING* and the standard Linux TCP/IP stack.

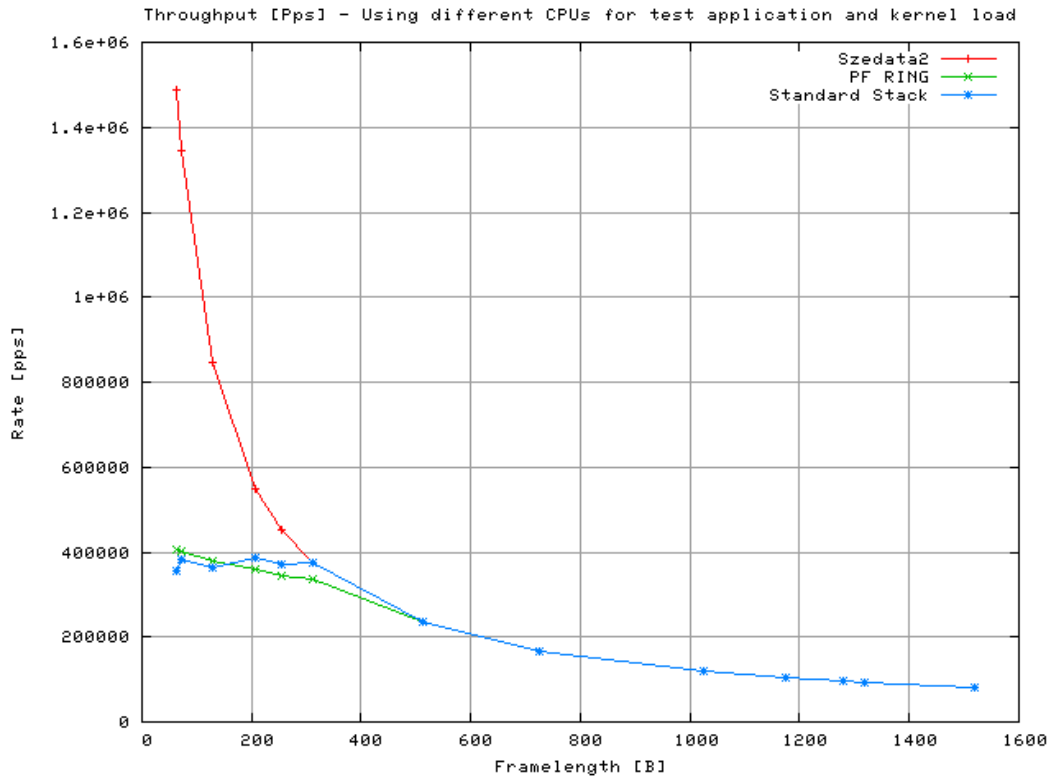


Figure 4. Iperf-like packet capture throughput #2 of *szedata2*, *PF_RING* and the standard Linux TCP/IP stack.

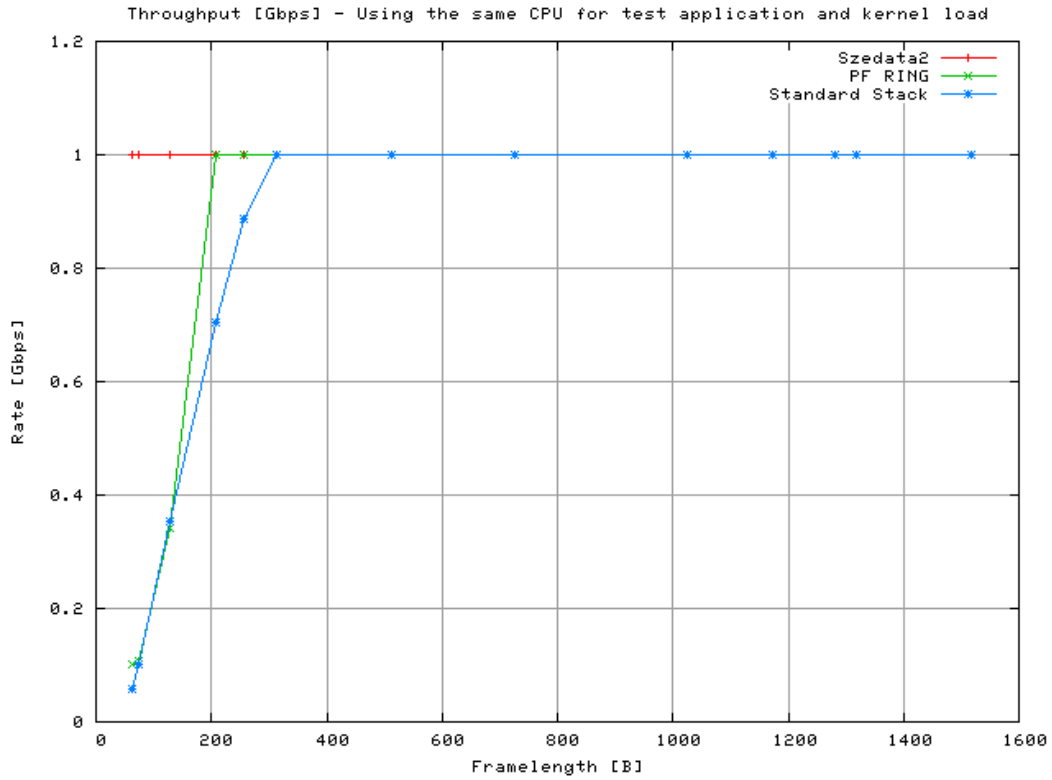


Figure 5. Iperf-like packet capture throughput #1 of *szedata2*, *PF_RING* and the standard Linux TCP/IP stack.

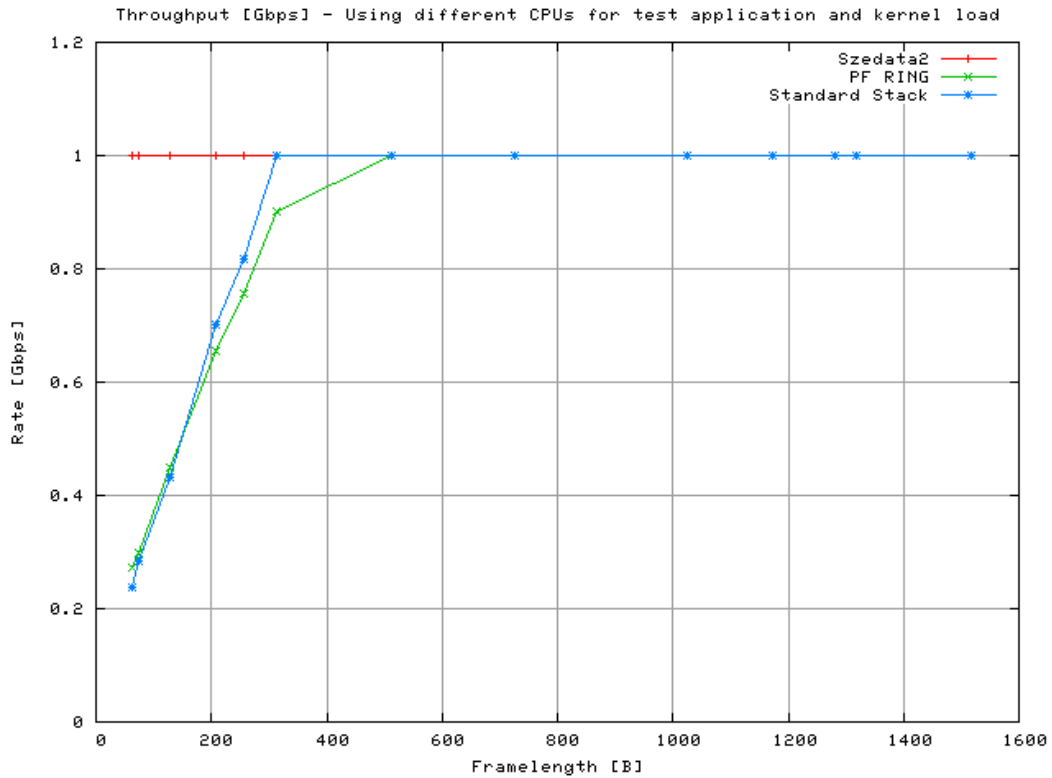


Figure 6. Iperf-like packet capture throughput #2 of *szedata2*, *PF_RING* and the standard Linux TCP/IP stack.

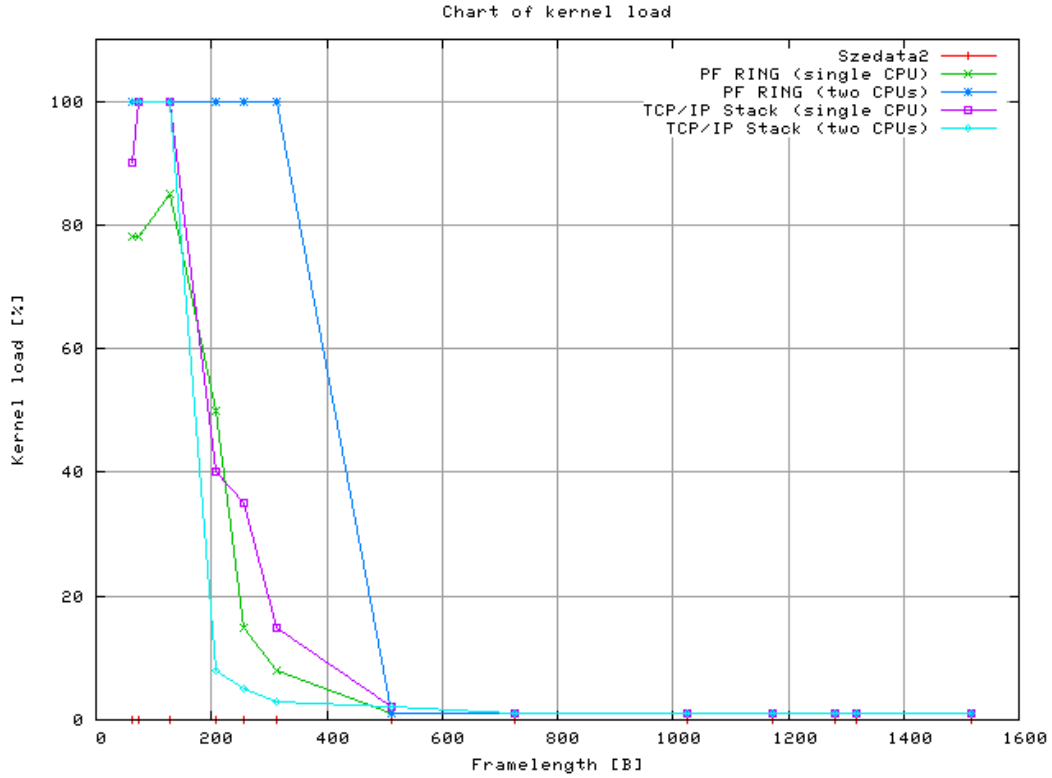


Figure 7. Dependency of kernel load on frame lengths for *szedata2*, *PF_RING* and standard Linux TCP/IP stack.

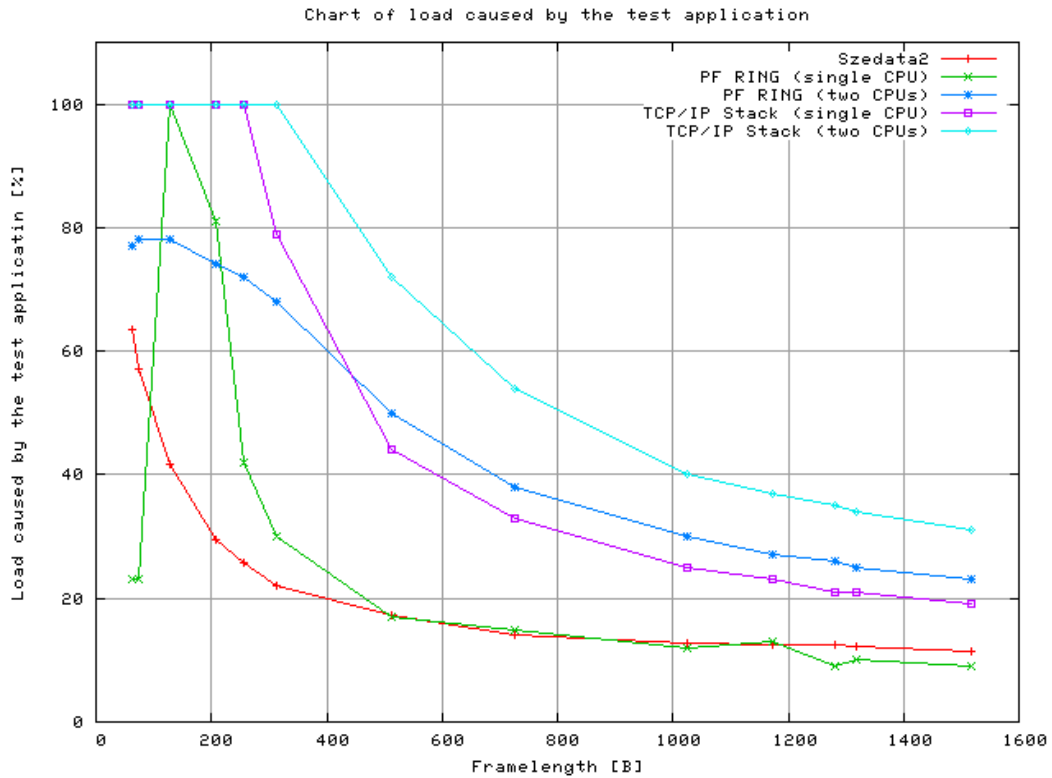


Figure 8. Dependency of load caused by the test application on frame lengths for *szedata2*, *PF_RING* and standard Linux TCP/IP stack.

Table 4. RFC 2544 throughput of *szedata2* with COMBO6X NIC.

Frame Length [B]	Throughput [pps / Gbps]	
	#1	#2
1518	81274 / 1.0	81274 / 1.0
1319	93353 / 1.0	93353 / 1.0
1280	96154 / 1.0	96154 / 1.0
1174	104690 / 1.0	104690 / 1.0
1024	119732 / 1.0	119732 / 1.0
726	167560 / 1.0	167560 / 1.0
512	234963 / 1.0	234962 / 1.0
313	375375 / 1.0	331753 / 0.884
256	452899 / 1.0	331713 / 0.732
207	550066 / 1.0	349541 / 0.635
128	844595 / 1.0	362912 / 0.430
73	1344086 / 1.0	387212 / 0.288
64	1488095 / 1.0	377837 / 0.254

Table 5. RFC 2544 throughput of *PF_RING* with Intel NIC.

Frame Length [B]	Throughput [pps / Gbps]	
	#1	#2
1518	81274 / 1.0	81274 / 1.0
1319	93353 / 1.0	93353 / 1.0
1280	96154 / 1.0	96154 / 1.0
1174	104690 / 1.0	104690 / 1.0
1024	119732 / 1.0	119732 / 1.0
726	167560 / 1.0	167560 / 1.0
512	234962 / 1.0	234962 / 1.0
313	355947 / 0.948	331753 / 0.884
256	387441 / 0.855	331713 / 0.732
207	426439 / 0.774	349541 / 0.635
128	593031 / 0.702	362912 / 0.430
73	624790 / 0.465	387212 / 0.288
64	632150 / 0.425	377837 / 0.254

lengths equal to or greater than 512 B it reached maximum throughput of 1 Gbps. Consequently, the RFC 2544 throughput goes down till approximately 400 Mbps in case of using the same CPU for test application and kernel operations or 250 Mbps in case of using different CPUs. RFC 2544 throughput of standard TCP/IP stack is very low, particularly if we used the same CPU for test application and kernel operations. Figures 9, 10, 11 and 12 depict these results.

The reason of poor throughput of TCP/IP stack is a losing of few packets (for example about 350 packets from several millions) in almost every iteration of binary search. Standard stack was almost unable to capture all the packets except the longest ones.

On the contrary, *PF_RING* profited from this behaviour. It was unable to capture all the packets shorter than 512 B at speed of 1 Gbps. But when we slowed

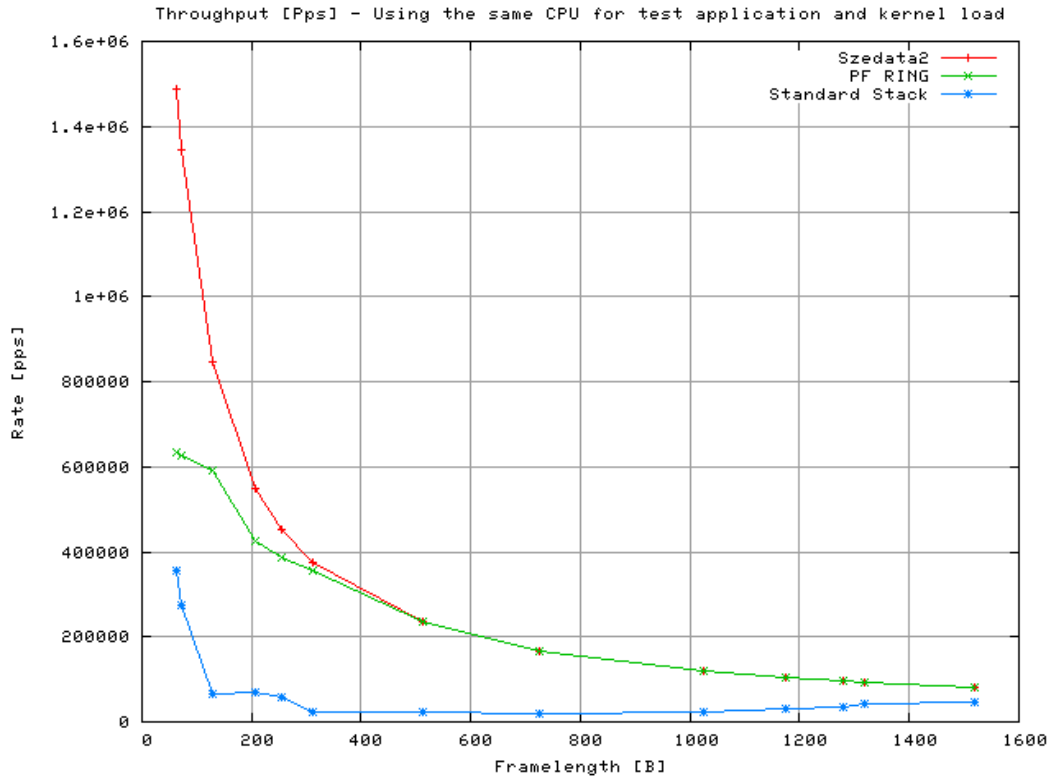


Figure 9. RFC 2544 packet capture throughput #1 of *szedata2*, *PF_RING* and the standard Linux TCP/IP stack.

Table 6. RFC 2544 throughput of the Linux TCP/IP stack with Intel NIC.

Frame Length [B]	Throughput [pps / Gbps]	
	#1	#2
1518	45717 / 0.562	81274 / 1.0
1319	40842 / 0.438	93353 / 1.0
1280	36245 / 0.377	96154 / 1.0
1174	31898 / 0.305	104690 / 1.0
1024	24087 / 0.201	102778 / 0.858
726	20618 / 0.123	78380 / 0.468
512	24093 / 0.103	80768 / 0.344
313	23461 / 0.062	57919 / 0.154
256	59266 / 0.131	119417 / 0.264
207	68833 / 0.125	119382 / 0.217
128	65159 / 0.077	150113 / 0.178
73	273017 / 0.203	165386 / 0.123
64	354585 / 0.238	232515 / 0.156

down the speed of incoming traffic it did not lose any packet. So the speed could be increased and the final throughput is higher than in case of the iperf-like testing.

Finally, we measured packet capture throughput of Internet mix (IMIX) traffic. We chose the following distribution of four packet lengths: 57 % of 64B packets, 7 % of 570B packets, 16 % of 594B packets and 20 % of 1518B packets. The measured throughput was 1 Gbps for all three methods. Only the load caused by the test

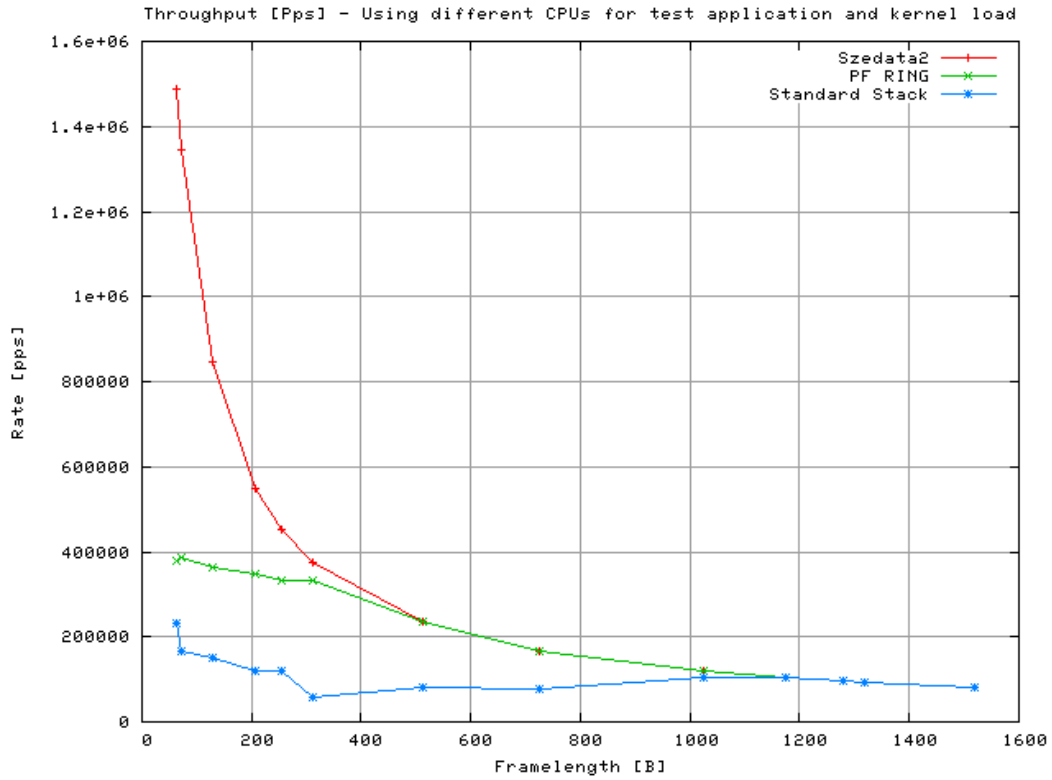


Figure 10. RFC 2544 packet capture throughput #2 of *szedata2*, *PF_RING* and the standard Linux TCP/IP stack.

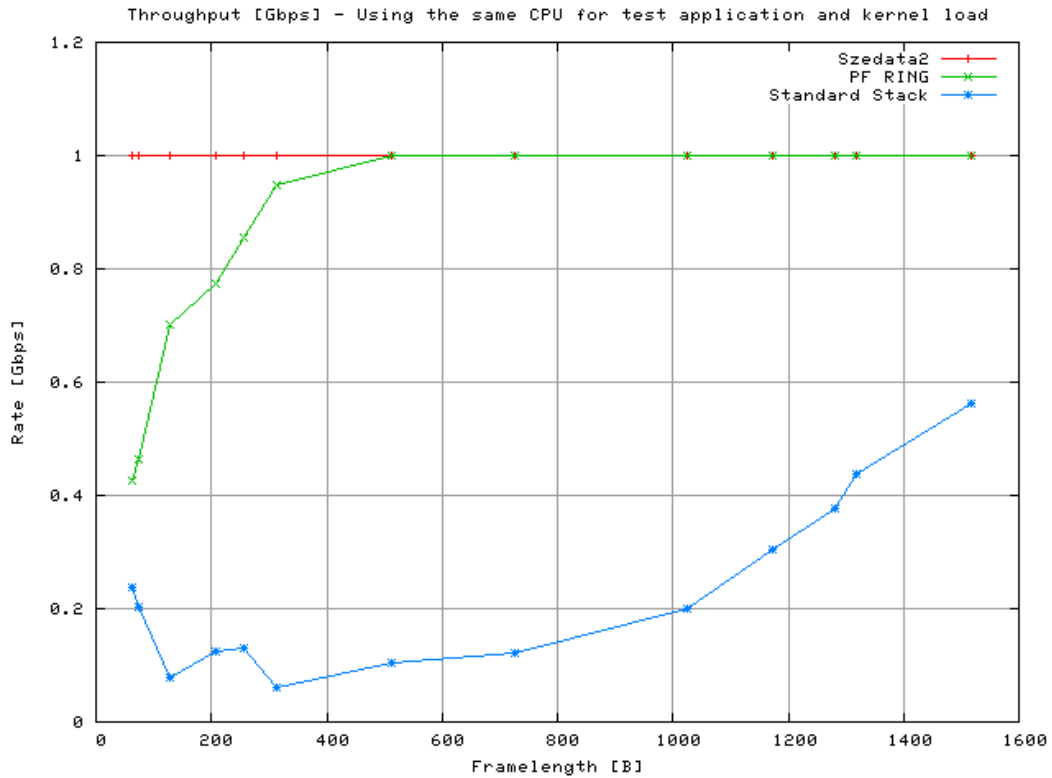


Figure 11. RFC 2544 packet capture throughput #1 of *szedata2*, *PF_RING* and the standard Linux TCP/IP stack.

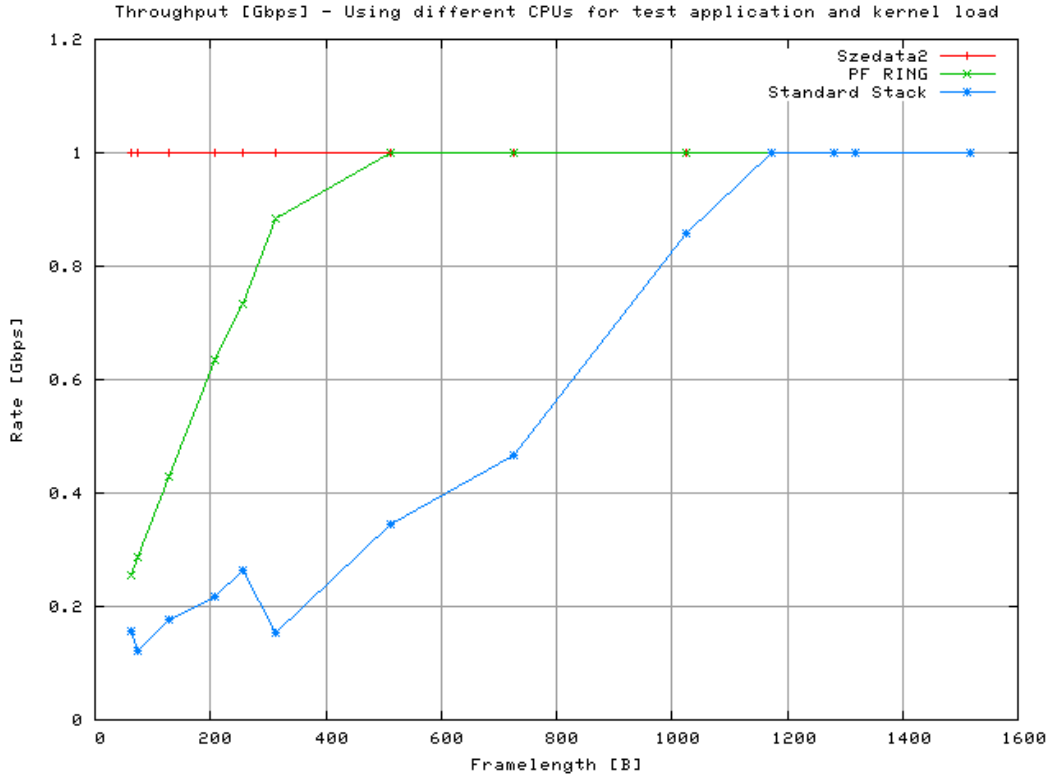


Figure 12. RFC 2544 packet capture throughput #2 of *szedata2*, *PF_RING* and the standard Linux TCP/IP stack.

Table 7. Load caused by all tested methods during throughput test of IMIX packet distribution.

Method	App./kernel load #1 [%]	App./kernel load #2 [%]
Standard Stack	54 / 10	75 / 4
PF_RING	22 / 5	53 / 1
szedata2	19 / 0	20 / 0

application and the kernel differed. Table 7 shows actual values.

6 Conclusion and Future Work

We conclude *szedata2* is the fastest method in the benchmark. Its throughput does not depend on packet length and is always 1 Gbps (theoretical maximum throughput). On the contrary, the other two methods suffers by significant loss at the shortest packet lengths. *PF_RING* achieve better performance for packet capture than the standard Linux TCP/IP stack, as we suppose. The described saturation appeared so we excluded anomalous results caused by this effect.

Also the CPU load was monitored during the testing. As opposed to *PF_RING* and standard stack, *szedata2* did not cause any kernel load at all tested frame lengths. We found out the performance depends whether all relevant processes are running on one CPU or are distributed among two processors.

Next, it was expected that the specific settings of parameters of network device

driver *e1000* should increase the performance. But we did not notice any significant influence on the throughput during the testing.

Finally, we can confirm that the results may vary due to chosen testing procedure. RFC 2544 testing methodology is rigorous whereas iperf-like testing corresponds to the best effort policy. For example, the former should be used when packet loss is *mission critical*, such as in security applications.

Our future work will be aimed at packet capture benchmark on 10 Gigabit Ethernet. We would like to benchmark a new generation of COMBO cards, namely COMBOv2, with *szedata2* and Myricom network interface card with PF_RING and the standard Linux TCP/IP stack.

Appendix A. Used SW/HW equipment

Testing device: Spirent TestCenter 2000 *Testing software:* Tcl library version 2.20

Host computer:

- Number of CPU: 2 (quad core)
- CPU: Xeon(R) CPU E5335 @ 2.00GHz 16 KB L1 + 8 MB L2 cache
- RAM: 4 GB
- MB: Supermicro X7DB8
- OS: Redhat Enterprise Linux ES release 4 (Nahant update 1)
- Kernel: Linux 2.6.26.3_32bit
- PCI bridge: c610.05.0c (2008/06/09 19:36) NetCOPE Bridge
- MAC: 00:30:48:35:d1:90
- COMBO cards: COMBO6X / COMBO-4SFPRO/1G
- COMBO6X S/N: CAM7000047
- COMBO-4SFPRO S/N: CAM70000468
- Intel card: 82546GB Gigabit Ethernet Controller
- Driver: e1000
- Driver version: 7.3.20-k2-NAPI
- PCI: PCI-X
- libpcap: version 0.9.7 and 0.9.8
- PF_RING: version 3.8.6
- ethtool: version 1.8

Appendix B. Source Code of the Test Application

```
#include <pcap.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/poll.h>
#include <time.h>
#include <stdint.h>
#include <sys/time.h>
```

```

#define INTERVAL 1000      /* 1 second by default */
int ctrl_c = 0;
/* to catch an interrupt signal (ctrl+c) */
static void sig_usr(int signo)
{
    if (signo == SIGINT) {
        ctrl_c = 1;
    }
}
int main(int argc, char *argv[])
{
    uint64_t p_count = 0; /* Total packet count */
    uint64_t p_last = 0; /* Packets in last interval */
    uint64_t next;      /* Next time check */
    uint64_t now;       /* Current time */
    uint64_t start;    /* Start packet arriving time */
    uint64_t last;     /* Last packet incoming time */
    struct timeval t_now;
    pcap_t *handle;    /* Pcap handle */
    int pcap_socket;
    /* Pcap error string buffer */
    char errbuf[PCAP_ERRBUF_SIZE];
    //catch SIGINT
    signal(SIGINT, sig_usr);
    if (argc < 2) {
        fprintf(stderr, "Need argument where to listen
            (%s eth0)\n", argv[0]);
        exit(-1);
    }
    char *dev = argv[1]; /* The device to sniff on */
    printf("Pcap library version:%s \n",
        pcap_lib_version());
    /* Open the session in promiscuous mode */
    handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device %s: %s\n",
            dev, errbuf);
        return (-2);
    }
    pcap_socket = pcap_fileno(handle);
    /* set first next interval */
    gettimeofday(&t_now, NULL);
    /* accuracy in ms */
    start = next = (t_now.tv_sec * INTERVAL) +
        (t_now.tv_usec / INTERVAL);
    next += INTERVAL;

```

```

/* using polling */
while (!ctrl_c) {
    /* The header that pcap gives us */
    struct pcap_pkthdr header;
    /* do not use poll() function with szedata2 */
#ifdef POLL
    struct pollfd pfd;
    int ret;
    pfd.fd = pcap_socket;
    pfd.events = POLLIN;
    ret = poll(&pfd, 1, INTERVAL);
    /* packet may not come */
    if (ret > 0 && pfd.revents & POLLIN
        && pcap_next(handle, &header)) {
    #else
        if (pcap_next(handle, &header)) {
    #endif
        last = now = header.ts.tv_sec * INTERVAL
            + header.ts.tv_usec / INTERVAL;
        if(!p_count) {
            start = now;
        }
        p_count++;
    } else {
        gettimeofday(&t_now, NULL);
        /* accuracy in ms */
        now = (t_now.tv_sec * INTERVAL) +
            (t_now.tv_usec / INTERVAL);
    }
    if (now >= next) {
        uint64_t delta = now - next + INTERVAL;
#ifdef OUTPUT
        /* print instant throughput, not in rfc2544 */
        if (p_count) {
            printf ("iteration: packets %llu,
                pps %llu\n", p_count - p_last,
                    ((p_count - p_last) * INTERVAL) / delta);
        }
#endif
        p_last = p_count;
        next = now + INTERVAL;
    }
}
if (p_count) {
    /* different output */
#ifdef OUTPUT

```

```

    /*print real packterate from first to end packet*/
    printf ("\ntotal: packets %llu, time %llu.%llu,
    pps %llu\n", p_count, (last-start) / INTERVAL,
    (last-start) % INTERVAL,
    (p_count * INTERVAL) / (last - start));
    #else
    printf("%llu\n", p_count);
    #endif
}
/* close handle */
pcap_close(handle);
return (0);
}

```

References

- [1] DERI, L. *PF_RING User Guide*, 2008. Available online⁷.
- [2] SLABY, J. *Rapid Data Transfers on COMBO Platform*, Diploma Thesis, Brno: Masarykova univerzita, 2008. Available online⁸.
- [3] BRADNER, S.; McQUAIDRFC, J. *Benchmarking Methodology for Network Interconnect Devices*, RFC 2544⁹, IETF, 1999.
- [4] BRADNER, S. *Benchmarking Methodology for Network Interconnect Devices*, RFC 1242¹⁰, IETF, 1991.

⁷ https://svn.ntop.org/trac/browser/trunk/PF_RING/doc/

⁸ http://is.muni.cz/th/98734/fi_m/

⁹ <http://tools.ietf.org/html/rfc2544>

¹⁰ <http://tools.ietf.org/html/rfc1242>