Improving Passive Packet Capture: Beyond Device Polling

Luca Deri
NETikos S.p.A.
Via del Brennero Km 4, Loc. La Figuretta
56123 Pisa, Italy
Email: luca.deri@netikos.com
http://luca.ntop.org/

Abstract
Passive packet capture is necessary for many activities including network debugging and monitoring. With the advent of fast gigabit networks, packet capture is becoming a problem even on PCs due to the poor performance of popular operating systems. The introduction of device polling has improved the capture process quite a bit but not really solved the problem.
This paper proposes a new approach to passive packet capture that combined with device polling allows packets to be captured and analyzed using the NetFlow protocol at (almost) wire speed on Gbit networks using a commodity PC.

Keywords
Passive packet capture, device polling, Linux, NetFlow.

1. Introduction

Many network monitoring tools are based on passive packet capture. The principle is the following: the tool passively captures packets flowing on the network and analyzes them in order to compute traffic statistics and reports including network protocols being used, communication problems, network security and bandwidth usage. Many network tools that need to perform packet capture ([tcpdump], [ethereal], [snort]) are based on a popular programming library called *libpcap* [libpcap] that provides a high level interface to packet capture. The main library features are:

- Ability to capture from various network media such as ethernet, serial lines, virtual interfaces.
- Same programming interface on every platform.
- Advanced packet filtering capabilities based on BPF (Berkeley Packet Filtering), implemented into the OS kernel for better performance.

Depending on the operating system, libpcap implements a virtual device from which captured packets are read from userspace applications. Despite different platforms provide the very same API, the libpcap performance varies significantly according to the platform being used. On low traffic conditions there is no big difference among the various platforms as all the packets are captured, whereas at high speed[1] the situation changes significantly. The following table shows the outcome of some tests performed using a traffic generator [tcpreplay] on a fast host (Dual 1.8 GHz Athlon, 3Com 3c59x

[1] Note that high speed is relative to the speed/specs of the used to capture traffic.

ethernet card) that sends packets to a mid-range PC (VIA C3 533 MHz[2], Intel 100Mbit ethernet card) connected over a 100 Mbit Ethernet switch (Cisco Catalyst 3548 XL) that is used to count the real number of packets sent/received by the hosts[3]. The traffic generator reproduces at full speed (~80 Kpps) some traffic that has been captured previously, whereas the capture application is a simple application named *pcount* based on libpcap that counts and discards, with no further analysis, the captured packets.

Traffic Capture Application	**Linux 2.4.x**	**FreeBSD 4.8**	**Windows 2K**
Standard Libpcap	0.2 %	34 %	68 %
Mmap Libpcap	1 %		
Kernel module	4 %		

Table 1. – Percentage of captured packets (generated by tcpreplay)

The experience learnt from this experiment is:

- At 100 Mbit using a low-end PC, the simplest packet capture application is not able to capture everything (i.e. there is packet loss).
- During the above experiment, the host used for capturing packets was not responsive when the sender injected packets on the network.
- Out of the box, Windows and *winpcap* [winpcap], the port of libpcap to Win32, perform much better than other popular Unix-like OS.
- Linux, a very popular OS used for running network appliances, performs very poorly with respect to other OSs used in the same test.
- *Libpcap-mmap* [libpcap-mmap], a special version of libpcap that exploits the mmap() system call for passing packets to user space, does not improve the performance significantly.
- Implementing a Linux kernel module based on *netfilter* [netfilter] improves the performance significantly, but still under Linux most of the packets are lost. This means that Linux spends most of its time moving packets from the network card to the kernel and very little from kernel to userspace.

An explanation for the poor performance figures is something called *interrupt livelock* [mogul]. Device drivers instrument network cards to generate an interrupt whenever the card needs attention (e.g. for informing the operating system that there is an incoming packet to handle). In case of high traffic rate, the operating system spends most of its time handling interrupts leaving little time for other tasks. A solution to this problem is something called *device polling* [rizzo]. Polling is a technique for handling devices, including network cards, that works as follows:

- When a network device receives a packet it generates an interrupt to request kernel attention.
- The operating system serves the interrupt as follows:

[2] This PC has almost the same speed of a low-end Pentium-III based PC.

[3] Many people believe that the interface counters (e.g. the values provided by `ifconfig` or `netstat -ni`) can be trusted. This statement is true if the system is able to handle all the traffic. If not, the receiver will usually display values that are (much) less than the real traffic.

- It masks future interrupts generated by the card (i.e. the card cannot interrupt the kernel).
- It schedules a task for periodically polling the device to service its needs.
- As soon as the driver has served the card, the card interrupts are enabled.

Although this technique may seem not to be effective, in practice it gives the operating system control over devices and it prevents devices from taking over control over the kernel in case of high traffic rate. FreeBSD implements device polling since version 4.5 whereas Linux has introduced it with *NAPI* (New API)[4]. Note that the network device driver must be polling-aware in order to exploit device polling. The author has repeated the previous traffic experiment on Linux and FreeBSD (Windows does not seem to support device polling nor provide facilities for enabling it) using device polling. The figure below shows how systems behave when the input traffic increases. Basically as long as the system has enough CPU cycles to handle all the traffic, there is not much difference between the different setups. However for non-polling systems there is a maximum full-capture speed (see the vertical dashed line represented in the figure below) after which the system spends most of the available cycles to handle interrupts leaving little time to other tasks, hence the packet loss.

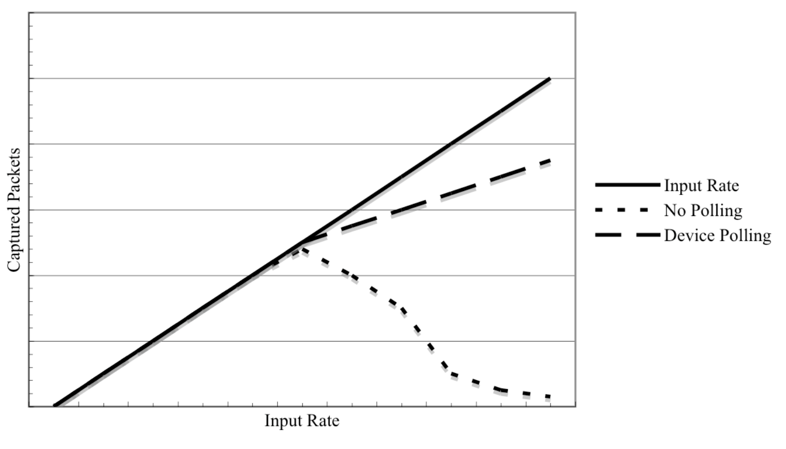

Figure 1. – Packet Capture Performance: Polling vs. non-polling

The results represented in table 1 show that, when plotting the curve of figure 1 for different operating systems in the case where polling is not used, there is always a dashed line. The performance figures of table 1 just show that for Linux the dashed line is on the left with respect to the one of FreeBSD, but that there is always a point after which the system will start losing packets.

The experience learnt from this experiment is:

- Device polling is a very good solution to improve both packet capture performance and system responsiveness under high traffic load.
- Device polling, especially on FreeBSD, has some disadvantages in terms of CPU utilisation. Moreover various experiments using different FreeBSD kernel

[4] NAPI support is available from kernel 2.4.23 onwards.

setups (e.g. with HZ value ranging from 1000 to 5000), libpcap tuning (e.g. increasing the receiver pcap buffer) and kernel parameters (`sysctl kern.polling.*`) do not significantly affect the overall performance nor the CPU utilization.
- Even with kernel polling, FreeBSD performs significantly better than Linux running the same userspace libpcap-based application.

The following chapter explains why device polling is just the starting point and not the ultimate solution.

2. Beyond Device Polling

As shown in the previous chapter, the very same simple libpcap-based application performs significantly different on Linux and FreeBSD. The gap between the two operating systems changes at Gbit speeds using a modified version of *stream.c*[5] as traffic generator (Dual 1.8 GHz Athlon, Intel Gbit ethernet card) that sends packets to a mid-range PC (Pentium III 550 Mhz, Intel Gbit ethernet card) connected over a cross cable. In all the tests libpcap has been configured to capture the first 128 bytes of the packet (pcap `snaplen` parameter).

Packet Size (bytes)	**Linux 2.6.1 with NAPI and standard libpcap**	**Linux 2.6.1 with NAPI and libpcap-mmap**[6]	**FreeBSD 4.8 with Polling**
64	2.5 %	14.9 %	97.3 %
512	1.1 %	11.7 %	47.3 %
1500	34.3 %	93.5 %	56.1 %

Table 2. – Percentage of captured packets (generated by stream.c) using kernel polling

After running all the experiments, the author realized that:
- Linux needs a new speed bump for efficiently capturing packets at high network speeds. The mmap version of libpcap is a big improvement but it is not very efficient with small packets.
- FreeBSD performs much better than Linux with small packets, but its performance decreases with large packets.
- Kernel device polling is not sufficient for capturing (almost) all packets in all the different setups.

Doing some further measurements, it seems that most of the time is spent moving the packet from the adapter to the userspace through the kernel data structures and queues. The mmap-version of libpcap reduced the time spent moving the packet from the kernel to userspace but has not improved at all the journey of the packet from the adapter to

[5] `stream.c`, a DoS (Denial of Service), can be downloaded from http://www.securiteam.com/unixfocus/5YP0I000DG.html.
[6] The test has been performed with the following setup: PCAP_FRAMES=max PCAP_TO_MS=0 PCAP_SNAPLEN=128

the kernel. Therefore the author has designed a new packet capture model based on the following assumptions and requirements:

- Design a solution for improving packet capture performance that is general and not locked to a specific driver or operating system architecture.
- Given that network adapters are rather cheap, it is not too costly to allocate a network adapter only for passive packet capture, as the goal is to maximize packet capture performance and not reduce the overall costs.
- Device polling proved to be very effective, hence (if available) it should be exploited to improve the overall performance.
- For performance reasons, it is necessary to avoid passing incoming packets to the kernel that will pass then to userspace. Instead a straight path from the adapter to the user space needs to be identified in order to avoid the kernel overhead.
- Facilities such as packet sampling should be implemented efficiently. In fact with the current libpcap, in case of sampling all the packets are moved to userspace and then the sampled packets are discarded with a large waste of CPU cycles.

The idea behind this work is the following:

- Create a new type of socket (`PF_RING`) optimized for packet capture that is based on a circular buffer where incoming packets are copied.
- The buffer is allocated when the socket is created, and deallocated when the socket is deactivated. Different sockets will have a private ring buffer.
- If a PF_RING socket is bound to an adapted (via the `bind()` syscall), such adapter will be used in read-only mode until the socket is destroyed.
- Whenever a packet is received from the adapter (usually via DMA, direct memory access), the driver passes the packet to upper layers (on Linux this is implemented by the `netif_receive_skb` and `netif_rx` functions depending whether polling is enabled or not). In case the PF_RING socket, every incoming packet is copied into the socket ring or discarded if necessary (e.g. in case of sampling when the specified sample rate has not been satisfied). If the buffer is full, the packet is discarded.
- Received packets for adapters with bounded PF_RING sockets, by default are not forwarded to upper layers but they are discarded after they have been copied into the rings. This practice increases the overall performance, as packets do not need to be handled by upper layers but only by the ring.
- The socket ring buffer is exported to userspace applications via mmap (i.e. the PF_RING socket supports mmap).
- Userspace applications that want to access the buffer need to open the file, then call `mmap()` on it in order to obtain a pointer to the circular buffer.
- The kernel copies packets into the ring and moves the write pointer forward. Userspace applications do the same with the read pointer.
- New incoming packets overwrite packets that have been read by userspace applications. Memory is not allocated/deallocated by packets read/written to the buffer, but it is simply overwritten.
- The buffer length and bucket size is fully user configurable and it is the same for

all sockets.

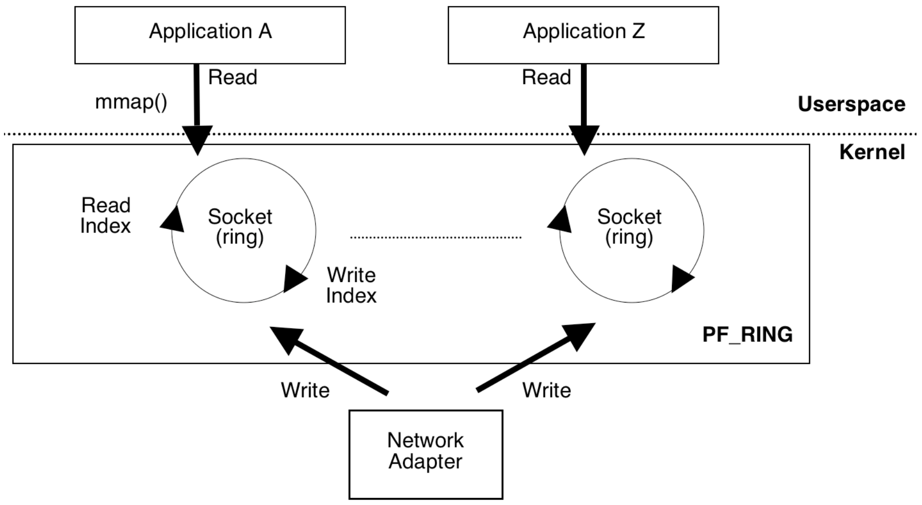

Figure 2. PF_RING Socket Architecture

The advantages of a ring buffer located into the socket are manifold, including:

- Packets are not queued into kernel network data structures.
- The mmap primitive allows userspace applications to access the circular buffer with no overhead due to system calls as in the case of socket calls.
- Even with kernel that does not support device polling, under strong traffic conditions the system is usable. This is because the time necessary to handle the interrupt is very limited compared to normal packet handling.
- Implementing packet sampling is very simple and effective, as sampled packets do not need to be passed to upper layers then discarded as it happens with conventional libpcap-based applications.
- Multiple applications can open several PF_RING socket simultaneously without cross interference (e.g. the slowest application does not slow the fastest application down).

The main difference with respect to normal packet capture is that applications that rely on libpcap need to be recompiled against a modified (ring/mmap-aware) version of the library as incoming packets are stored into the buffer and no longer in the kernel data structures. For these reason the author has extended libpcap with PF_RING support (on Linux the libpcap uses PF_PACKET sockets).

In order to evaluate the performance of the proposed architecture, the author has modified the Linux kernel code and implemented the PF_RING socket into a module. The modified kernel has been tested in the same environment used previously in order to evaluate the performance advantage with respect to the original kernel. The following table shows the test outcome.

Packet Size (bytes)	Linux 2.6.1 with NAPI and libpcap standard	Linux 2.6.1 with NAPI and libpcap-mmap[7]	FreeBSD 4.8 with Polling	Linux 2.6.1 with NAPI+PF_RING and extended libpcap
64	2.5 %	14.9 %	97.3 %	75.7 %
512	1.1 %	11.7 %	47.3 %	47.0 %
1500	34.3 %	93.5 %	56.1 %	92.9 %

Table 3. – Percentage of captured packets (generated by stream.c) using kernel polling

Basically the new solution:

- Has improved the packet capture speed with respect to the standard Linux.
- With large packets the PF_RING is as fast as libpcap-mmap(), whereas with medium/large packets is much faster.
- Still many packets are lost, especially with medium size packets.

A userspace application that has to access a mapped memory buffer can do it in two ways: with or without ring polling.

Packet Handling with Polling

```
sockFd = socket(PF_RING, SOCK_RAW,
                htons(ETH_P_ALL);
..
ringBufferPtr = mmap(NULL, ringSize,
                     PROT_READ|PROT_WRITE,
                     MAP_SHARED, sockFd, 0);

slotId = &ringBufferPtr->slotId;

while(TRUE) {
   /* Loop until a packet arrives */
   if(ringBufferPtr->slot[slotId].isFull) {
     readPacket(ringBufferPtr->slot [slotId]);
     ringBufferPtr->slot [slotId].isFull = 0;
     slotId = (slotId + 1) % ringSize;
   }
}
```

Packet Handling without Polling

```
sockFd = socket(PF_RING, SOCK_RAW,
                htons(ETH_P_ALL);
..
ringBufferPtr = mmap(NULL, ringSize,
                     PROT_READ|PROT_WRITE,
                     MAP_SHARED, sockFd, 0);

slotId = &ringBufferPtr->slotId;

while(TRUE) {
   if(ringBufferPtr->slot [slotId].isFull) {
     readPacket(ringBufferPtr->slot [slotId]);
     ringBufferPtr->slot [slotId].isFull = 0;
     slotId = (slotId + 1) % ringSize;
   } else {
     /* Sleep when nothing happens */
      pfd.fd = fd;
      pfd.events = POLLIN|POLLERR;
      pfd.revents = 0;
      poll(&pfd, 1, -1);
   }
}
```

Table 4. – Packet Retrieval in userspace Applications: poll() vs. polling

From the tests performed, it seems that the use of polling at userspace is not really effective in terms of performance gain. Actually when userspace polling is used, the CPU usage bumps from 12% (with 512 bytes packets) to over 95%, leaving fewer cycles to packet capture applications. After repeating the same test several times, the conclusion is that the use of userspace polling does not seem to improve the overall performance, or that at best the performance gain is not easy to measure. For this reason, all the ring-buffer tests have been performed using the poll() system call that has

[7] The test has been performed with the following setup: PCAP_FRAMES=max PCAP_TO_MS=0 PCAP_SNAPLEN=128

the advantage of keeping the system load low with respect to userspace polling that exhausts all the available CPU cycles.

What the author has noticed while performing the tests is that with the PF_RING solution, there is still some packet loss, although there are plenty of CPU cycles. For example, the CPU is loaded only at 8% when capturing 1500 bytes packets, with a packet loss of about 7%. This is somehow a contradiction, as the system should use the spare cycles to capture all the packets. After performing some measurements inside the Linux kernel using *rdtsc* (Real Time Stamp Counter)[8], the author noticed that at high incoming packet rate:

- A dummy (i.e. that return immediately) call to `poll()` is rather costly (well over 1000 cycles) and its cost increases significantly with the network load. This means that at high speeds while the user application is waiting the `poll()` to return, the kernel is discarding packets, as the ring is full.
- Kernel locks (spinlocks) used by PF_RING and kernel are rather costly in terms of waiting time.

As a consequence, there is a packet loss because the userspace application is blocked on the `poll()` although the system has plenty of CPU cycles available. Even allocating a large kernel buffer did not really help, as the system was still unpredictable regarding the time spent doing other activities. For this reason the author decided to look for a Linux kernel patch that provided predictability, namely the ability to determine task completion with some time constraints. Using a patch (*rtirq*) for prioritising interrupts, hence having a low and precise latency in the Linux 2.4.x kernel [kuhn], the results of the test changed dramatically as we can now capture all the traffic regardless of the packet size. There is still some packet loss that is due to errors on interface (e.g. packet overrun). Another advantage of using the rtirq patch is that the CPU load on the receiver is very limited (less than 30%) leaving plenty of CPU cycles for real traffic analysis. The following table shows how *nProbe* [nprobe] 3.0, an open source NetFlow v5/v9 probe written by the author, behaves with respect to simple packet count[9].

Packet Size (bytes)	**Linux 2.4.23/RTIRQ NAPI+PF_RING (Pkt Capture)**	**FreeBSD 4.8 with Polling (Pkt Capture)**	**Linux 2.4.23/RTIRQ NAPI+PF_RING (nProbe)**	**FreeBSD 4.8 with Polling (nProbe)**
64	207’603 pps	202’013 pps	192’515 pps	142’153 pps
512	172’576 pps	81’691 pps	165’966 pps	64’350 pps
1500	72’545 pps	40’690 pps	72’156 pps	34’986 pps

Table 5. – Captured and analyzed packets (no sampling)

[8] rdtsc is a Pentium instruction that returns the number of clock cycles since the CPU was powered up or reset.

[9] Unfortunately the author is using a PC for generating traffic, as he has no access to a hardware traffic generator for testing the architecture to its upper limit. Alternative solutions (e.g. a L2 switch with STP disabled, two ports in loop via a cross cable, with a broadcast packet injected) offer speed similar to the PC in terms of packet speed generation.

As the table shows:

- As Linux/RTIRQ/PF_RING has a limited load on the CPU, packet analysis using NetFlow performs almost as simple packet capture with very limited packet loss.
- As mentioned before, FreeBSD is rather efficient with packet capture, but its extreme load on the CPU is a penalty when more complex activities such as NetFlow analysis are required, as there are limited CPU cycles available for userspace applications.
- The implemented solution:
    - o Significantly both reduced the journey of the packet from the network card to the traffic capture application, and decreased the load on the kernel.
    - o Made the capture process independent from the packet size point of view, in terms of work done on the kernel (of course, the load on the PCI bus is the same as a vanilla kernel as the card moves the whole packet to the kernel even if only a portion of the packet might be necessary for traffic analyses).
- It is possible to analyze using NetFlow traffic flowing across Gbit network using a commodity PC and software network probes.

In order to understand the limit of the proposed architecture, a different test-bed has been used: using the same sender host, the receiver PC has been replaced with a Pentium 4 running at 1.7 GHz with a 32 bit Intel GE ethernet card. The table below shows the test outcome.

Packet Size (bytes)	**Linux 2.4.23/RTIRQ NAPI+PF_RING (Pkt Capture)**		**Linux 2.4.23/RTIRQ NAPI+PF_RING (nProbe)**	
64	550’789 pps	~202 Mbit	376’453 pps	~144 Mbit
512	213’548 pps	~850 Mbit	213’548 pps	~850 Mbit
1500	81’616 pps	~970 Mbit	81’616 pps	~970 Mbit

Table 6. – PF_RING Evaluation (Receiver Pentium 4 1.7 GHz, Intel GE 32-bit)

As the table shows, the NetFlow analysis is limited by the available CPU cycles and there’s a moderate packet loss only with tiny packets, whereas with medium and large packets there is basically no loss. The errors on the receiving interface have significantly decreased with respect to the previous setup; probably they can be reduced to zero using a 64-bit PCI card on the receiver side.
Considering that Gbit networks usually have jumbo (>= 9000 bytes) MTUs, the PF_RING solution can very well be used for analyzing traffic at wire speed. Furthermore, it is worth to remark that the figures shown in table 6 using a low-end PC, are far better than many high-end routers available on the market.

3. Work In Progress

PF_RING has significantly improved the performance of packet captures. However it has two limitations:

- It requires a kernel real-time patch in order to improve the system call performance and in particular of poll().
- Its performance is limited by both the way network drivers are written as well as the way NAPI fetches packets from the network adapter.

In the last month the real-time patch issue has been solved using an adaptive poll algorithm. In other words, whenever there are no packets to process, the library sleeps for a limited amount of time (usually a few nano-seconds, because otherwise the system will loose packets as the userspace application will not fetch packets from the kernel) before `poll()` is called. If, after the sleep, there are still no packets available, `poll()` is called, otherwise packets are processed immediately. The sleep time is adapted according to the incoming packet rate: the more incoming packets means shorter (or zero) sleep time. Thanks to this trick, `poll()` is called very seldom hence avoiding the need to improve system calls performance with third party patches.

As said before, PF_RING reduces significantly the journey of a packet from the network driver to userspace. In order to further improve the packet capture performance it is necessary to modify the implementation of network card drivers. This is because drivers have been written for general-purpose applications and are not optimized for packet capture, as a new socket buffer is allocated/freed for each incoming packet. This practice is necessary if the packet has to be queued in kernel structures, but it is a waste of time if incoming packets are handled by PF_RING as they are copied into the ring and then discarded. Some early experiments have demonstrated that hacking the driver significantly increases the overall performance. In fact in the same scenario of table 6, the capture speed has been increased to:

- Over 800 Kpps simply using a hacked Intel driver with an adaptive poll() algorithm without the RTIRQ patch.
- Over 1.2 Mpps using a Xeon PC with a 64 bit Ethernet card.

Future work items include:

- The positioning of this work with respect to commercial cards such as the DAG card [dag].
- Evaluation of the proposed architecture on a 10 Gbit network[10] using a fast PC, in order to evaluate its scalability.
- Study of the features that can be implemented with respect to packet transmission in order to have a complete send/receive architecture.

4. Final Remarks

This paper has described the design and implementation of a new approach to packet capture whose goal is to improve the performance of the capture process at Gbit speeds. The validation process has demonstrated that:

- A new type of socket in addition to changes into the Linux kernel, has significantly improved the overall capture process at both 100 Mbit and Gbit. It

[10] Intel is currently selling 10 Gbit Ethernet adapter cards for x86 PCs.

is now possible to capture packets at wire speed with almost no loss at Gbit speeds using a commodity PC.

- Userspace applications based on the circular buffer are faster than an equivalent kernel module not using the ring code running on the same PC using vanilla Linux.
- At Gbit speeds, packet loss is very minimal, if any, even using PCs with limited CPU power.

In conclusion, it is now possible to analyze traffic at Gbit speeds using commercial protocols such as NetFlow using commodity PCs without the need to purchase expensive hardware. When 10 Gbit cards will become cheaper, it will be worth to explore how the described solution performs in such situations.

5. Availability

This work is distributed under the GPL2 license and can be downloaded free of charge from the ntop home page (http://www.ntop.org/) and other mirrors on the Internet (e.g. http://sourceforge.net/projects/ntop/).

6. Acknowledgment

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union. The author would like to thank Yuri Francalacci <yuri.francalacci@netikos.com>, Loris Degioanni <loris.degioanni@polito.it>, Fulvio Risso <fulvio.risso@polito.it>, Alex Tudor <alex_tudor@agilent.com> and Luigi Rizzo <rizzo@iet.unipi.it> for the valuable discussions, challenging questions, suggestions and support throughout this research work.

7. References

[dag] *The DAG Project*, Univ. of Waikato, http://dag.cs.waikato.ac.nz/.

[ethereal] G. Combs, *Ethereal*, http://www.ethereal.com/.

[kuhn] B. Kuhn, *Interrupt Prioritization in the Linux Kernel*, http://home.t-online.de/home/BernHard_Kuhn/rtirq/.

[libpcap] Lawrence Berkeley National Labs, *libpcap*, Network Research Group, http://www.tcpdump.org/.

[mogul] J. Mogul and K. Ramakrisnan, *Eliminating Receive Livelock in an Interrupt-Driven Kernel*, Proceedings of 1996 Usenix Technical Conference, 1996.

[netfilter] H. Welte, *The Netfilter Framework in Linux 2.4*, Proceedings of Linux Kongress 2000.

[nprobe] L. Deri, *nProbe: an Open Source NetFlow Probe for Gigabit Networks*, Proceedings of Terena TNC 2003, Zagreb, May 2003.

[pcap-mmap] P. Wood, *libpcap-mmap*, Los Alamos National Labs, http://public.lanl.gov/cpw/.

[rizzo]	L. Rizzo, *Device Polling Support for FreeBSD*, http://info.iet.unipi.it/~luigi/polling/, BSDConEurope Conference, 2001.
[salim]	J. Salim and others, *Beyond Softnet*, http://www.cyberus.ca/~hadi/usenix-paper.tgz, Proceedings of 2001 Usenix Annual Technical Conference, Boston, 2001.
[snort]	M. Roesch*, Snort – Lightweight Intrusion Detection for Networks*, Proceedings of Usenix Lisa '99 Conference, http://www.snort.org/.
[tcpreplay]	A. Tuner and M. Bing, *tcpreplay*, http://tcpreplay.sourceforge.net/.
[tcpdump]	The Tcpdump Group, *tcpdump*, http://www.tcpdump.org/.
[winpcap]	L. Degioanni and others, *Profiling and Optimization of Software-Based Network-Analysis Applications*, Proceedings of the 15th IEEE SBAC-PAD 2003 Symposium, Sao Paulo, Brazil, November 2003, http://winpcap.polito.it/docs/.