Analyzing and Optimizing the Linux Networking Stack

Raffaele Bolla*, Roberto Bruschi*, Andrea Ranieri*, Gioele Traverso§

** Department of Communications, Computer and Systems Science (DIST)*

University of Genoa

Via Opera Pia 13, I-16145 Genova, Italy

Email: raffaele.bolla, roberto.bruschi, andrea.ranieri@dist.unige.it

§ *InfoCom Genova*

Piazza Alessi 2, I-18128 Genova, Italy

Email: gioele.traverso@infocom.genova.it

It is well known the Linux Operative System provides a support so complete and effective to networking functionalities, to be widely used in many advanced fields, like, among the others, software routers and GRID architectures. Notwithstanding the arising popularity and the continuous development of the Linux Operative System, there is no clear indication about the maximum networking performance level that it can achieve, and no information about the computational weight required by packet processing operations. Our main aim in this work is to analyze and to optimize the Linux networking stack performance, trying to evaluate the impact of networking functionalities on the whole system. At this purpose, to better characterize the performance bottlenecks, we have performed both external and internal measurements, benchmarking both a standard Linux kernel and an optimized version.

Keywords. Linux networking, performance evaluation.

I. Introduction

Today, many Open Source Operating Systems (OSs), like Linux and FreeBSD, include networking capabilities so much sophisticated and complete, that they are often used in all such cases where an intensive and critical network usage is expected. In fact, thanks to the flexibility of the Open Source approach, these OSs are becoming more and more widespread in many usage fields, as, for example, in network firewall, web server, software router and GRID architecture. Notwithstanding their arising popularity and their continuous development, there is no clear indication about the maximum performance level that these architectures can achieve, and about the computational weight that they introduce.

In this work, we have decided to use the Linux OS, since it is the most widespread among the most efficient Open Source OSs. Moreover, Linux can boast one of the most complete networking stack (i.e., networking source code is about the 40% of the whole kernel), and it offers support to a wide range of hardware components. Besides we have to underline that Linux is still deeply inclined to the networking usage, but it is a general purpose architecture, and so it presents a large number of aspects that can be tuned or optimized to enhance its networking performance.

Our main aim in this work is to analyze and to optimize the Linux networking stack performance, trying to evaluate the impact of networking functionalities on the system. Moreover, to simplify the working environment, we have decided to take only layer 2 and 3 functionalities into account, since in every respect they are the backbone of the overall software architecture. Thus all the benchmarking tests here reported have been carried out by forwarding IPv4 datagrams (i.e., reception and transmission) through a single Gigabit Ethernet interface. At this purpose, in addition of the "classical" external performance measurements (i.e., latency and throughput), we have decided to perform also internal measurements, with the objective of better understanding the software structures and dynamics included in the Linux network code. These internal measurements have been carried out with both an advanced tool called Oprofile, and with some kernel patches, proposed by us, to dump the values of some critical Linux internal variables.

The paper is organized as in the following. The state of art is in the next Section, while Section III describes the OR software architecture and some parameter tuning and kernel patches that we have used to obtain the maximum performance. Section IV describes the external and internal performance evaluation tools used in the tests, while Sections V reports the numerical results of all the most relevant experiments. The conclusions and future activities are in Section VI.

II. Related Works

Some interesting works regarding the Linux networking stack can be found in the scientific literature. [1] reports some performance results (in packet transmission and reception) obtained with a PC Linux-based testbed. Some evaluations have been realized also on the network boards, see for example [2]. Other interesting works regarding Linux based routers can be found in [3] and in [4], where Bianco et al. report some interesting performance results. This work tries to give a contribution to the investigation by reporting the results of a large activity of optimization and testing realized on a router architecture based on Linux software.
In our previous works [5, 6, 7], carried out inside the BORABORA project [8], we have focused our attention on testing, optimizing and evaluating the performance of the basic packet forwarding functionalities. To this purpose, besides classical external (throughput and latency) measurements, we have adopted also profiling tools to analyze in depth and to optimize the internal behaviour of Linux. In [7], we have proposed a survey on how the presence of control plane functionalities in Software Router can impact on the forwarding performance.

II. The Linux Networking Architecture

As outlined in [6], while all the packet processing functions are realized inside the Linux kernel, the large part of daemons/applications requiring network functionalities run in user mode. Thus, we have to outline that, unlike most of the high-end commercial network equipments (e.g. IP routers), the packet processing functionalities and the control ones have to share the CPUs in the system. [7] reports a detailed description of how the resource sharing between the applications' plane and the packet process can have effect on the overall performance in different OR configurations (e.g., SMP kernel, single processor kernel, etc.).
The critical element for the network functionalities is the kernel where all the link, network and transport layer operations are realized. During the last years, the networking support integrated in the Linux kernel has experienced many structural and refining developments. For these reasons, we have chosen to use a last generation Linux kernel, more in particular a 2.6 version.
Since the older kernel versions, the Linux networking architecture is fundamentally based on an interrupt mechanism: network boards signal the kernel upon packet reception or transmission, through HW interrupts. Each HW interrupt is served as soon as possible by a handling routine, which suspends the operations currently processed by the CPU. Until completed, the runtime cannot be interrupted by anything, even by other interrupt handlers. Thus, with the clear purpose to make reactive the system, the interrupt handlers are designed to be very short, while all the time consuming task are performed by the so called "Software Interrupts" (SoftIRQ) in a second time. This is the well known "top half – bottom half" IRQ routine division implemented in the Linux kernel [9].

SoftIRQs are actually a form of kernel activity that can be scheduled for later execution rather than real interrupts. They differ from HW IRQs mainly in that a SoftIRQ is scheduled for execution by an activity of the kernel, like for example a HW IRQ routine, and has to wait until it is called by the scheduler. SoftIRQs can be interrupted only by HW IRQ routines.
The "NET_TX_SOFTIRQ" and the "NET_RX_SOFTIRQ" are two of the most important SortIRQs in the Linux kernel and the backbone of the whole networking architecture, since they are designed to manage the packet transmission and reception operations, respectively. In details, the forwarding process is triggered by a HW IRQ generated from a network device, which signals the reception or the transmission of packets. Then the corresponding routine makes some fast checks, and schedules the correct SoftIRQ, which is activated by the kernel scheduler as soon as possible. When the SoftIRQ is finally executed, it performs all the packet layer 2 and 3 forwarding operations.
Thus, the packet forwarding process is fundamentally composed by a chain of three different modules: a "reception API" that handles the packet reception (NAPI), a module that carries out the IP layer elaboration (both in reception and in transmission) and, finally, a "transmission API" that manages the forwarding operations to egress network interfaces. In particular, the reception and the transmission APIs are the lowest level modules, and are composed by both HW IRQ routines and SoftIRQs. They work by managing the network interfaces and performing some layer 2 functionalities.
More in details, the NAPI [10] was introduced in the 2.4.27 kernel version, and it has been explicitly created to increase the packet reception process scalability. It handles network interface requests with a interrupt moderation mechanism, which allows to adaptively switch from a classical interrupt management of the network interfaces to a polling one. This is done by inserting, during the HW IRQ routine, the identifier of the board generating the IRQ to a special list, called "poll list" , by scheduling a reception SoftIRQ, and by disabling the HW IRQs for that device. When the SoftIRQ is activated, the kernel polls all the devices, whose identifier is included in the poll list, and a maximum of quota packets are served per device. If the buffer (RxRing) of a device is emptied, then its identifier is removed from the poll list and its HW IRQs re-enabled, otherwise its HW IRQ are left disabled, the identifier kept in the poll list and a further SoftIRQ scheduled.
While this mechanism behaves like a pure interrupt mechanism in presence of low ingress rate (i.e., we have more or less a HW IRQ per packet), when traffic raises, the probability to empty RxRings, and so to re-enable HW IRQs, decreases more and more, and the NAPI starts working like a polling mechanism.
For each packet received during the NAPI processing, a descriptor, called *skbuff*, is immediately allocated and used for all the layer 2 and 3 operations. A packet is elaborated in the same NET_RX SoftIRQ, till it is enqueued in an egress device buffer, called Qdisc. Each time a NET_TX SoftIRQ is activated or a new packet is enqueued, the Qdisc buffer is served. When a packet is dequeued from the Qdisc buffer, it is placed on the Tx Ring of the egress device. After the board transmits one or more packets successfully, it generates a HW IRQ. After, during a SoftIRQ, all the descriptors of transmitted packets are de-allocated, and the Tx Ring is refilled by new packets coming from the Qdisc buffer.

Performance tuning

The whole networking kernel architecture is quite complex and it has many aspects and parameters that should be refined or tuned for a system optimization. For what concerns the parameter tuning, we have used the same configuration adopted in [6].
The 2.6.16.13 kernel images include four structural patches that we have produced with the aim of testing and/or optimizing the kernel functionalities. In particular, the used patches are described in the following.

Skbuff Recycling patch

In [20], we studied and realized a new version of the skbuff Recycling patch, originally proposed by R. Olsson [11] for the "e1000" driver. In particular, the new version is stabilized for 2.6.16.13 kernel version, and extended to the "sundance" driver.
This patch allows to intercept the skbuff descriptors of sent packets before their deallocation, and to re-use them for new incoming packets. As shown in [5], this architectural change heavily reduces the computation weight of the memory management operations, and it allows to achieve a very high performance level (i.e., about the 150-175% of the maximum throughput of standard kernels).

Performance Counter patch

To study in depth the OR internal behaviour, we have decided to introduce a set of counters in the kernel source code, with the aim of understanding how many times a certain procedure is called, or how many packets are kept per time. In detail, we have introduced the following counters:

- IRQ: number of interrupt handlers generated by a network card;
- tx/rx IRQ: number of tx/rx IRQ routine per device;
- tx/rx SoftIRQ: number of tx/rx software IRQ routines;
- Qdiscrun and Qdiscpkt: number of times when the output buffer (Qdisc) is served, and number of served packets per time.
- Pollrun and Pollpkt: number of times when the rx ring of a device is served, and the number of served packets per time.
- tx/rx clean: number of times when the tx/rx procedures of the driver are activated.

The values of all these parameters have been mapped in the Linux "proc" file system.

IV. Testbed and measurement tools

For what concerns the traffic generation and measurement, we have used a professional equipment, namely Agilent N2X Router Tester [12], which allows to obtain throughput and latency measurements with very high availability and accuracy levels (i.e., the minimum guaranteed timestamp resolution is 10 ns).

The internal measurements have been carried out by using, besides the counter patch described in the previous Section, a specific SW tool (called profiler) placed inside the OR. This software tool can trace the percentage of CPU utilization for each SW modules running on the node. The problem is that many of these profilers require a relevant computational effort that perturbs the system performance. We have experimentally verified that one of the best is Oprofile [13], an open source tool that realizes a continuous monitoring of system dynamics with a frequent and quite regular sampling of CPU HW registers. Oprofile allows the effective evaluation of the CPU utilization of both each SW application and each single kernel function with a very low computational overhead.

V. Numerical Results

To benchmark the performance of the Linux networking stack, we have selected a new generation Linux kernel, a 2.6.16.13 version, and used it with two hardware architectures. The presence of more hardware architectures gives us the possibility to better understand which and how performance bottlenecks can depend from the selected hardware, and to evaluate how networking performance scales according hardware capabilities.

With this idea we have selected two hardware architectures, namely Gelso and Magnolia respectively, which include fast I/O busses (i.e. PCI-X and PCI Express), and fast CPUs:

- Gelso: based on SuperMicro X5DL8-GG mainboard, equipped with a PCI-X bus and with a 32 bit 2.4 Ghz Intel Xeon;
- Magnolia: based on a SuperMicro X7DBE mainboard, it is equipped with both the PCI Express and PCI-X busses, and with a 5050 Intel Xeon, which is a dual core 64bit processor.

In particular, Magnolia can be considered like the "evolution" of Gelso, which does not include 64bit CPUs and PCI Express busses.

About the network interface cards (NIC), we have used the Intel PRO 1000 XT server adapters for the PCI-X bus and the Intel PRO 1000 PT dual port adapters for the PCI Express.

With such hardware architectures, we have benchmarked the maximum performance of the Linux networking stack in a very simple environment: a single data flow that is forwarded at IPv4 layer. The data flow is received and transmitted by a single Gigabit Ethernet interface, similarly to a server with very high network usage.

Observing Fig. 1, which shows the maximum forwarding rate according different packet sizes, we can note how all the software/hardware architectures can achieve almost the maximum theoretical rate only for packet sizes equal or larger the 256 Bytes. This kind of bottleneck can be better appreciated in Fig. 2, where the throughputs versus packet ingress rates are reported in most critical case of 64 Bytes packet size, while Fig. 3 shows the corresponding latency values.

From all these Figs., we can note how the Magnolia hardware architecture provide higher performance with respect to the Gelso one. Under similar conditions, if the NIC is placed on the PCI-X bus and the same standard kernel version is used, Magnolia achieves only nearly 20kpkt/s more than Gelso. But, when the skb recycling patch is applied, the performance difference becomes much greater: while Magnolia can forward up to 830 kpkt/s, Gelso reaches a bit less than 650 kpkt/s.
The use of PCI Express bus enhances considerably the performance of both the standard and the optimized kernels to 840 kpkt/s and 1390 kpkt/s, respectively.
This rise is due to the efficiency of PCI Express bus. In fact, it allows DMA transfers with very low latencies and with a very low control overhead, which probably lead to less heavy accesses to the RAM with a consequent benefit for the CPU memory accesses. Thus, in other words, this performance enhancement is substantially due to a more effective memory access of the CPU caused by the features of PCI Express DMA.

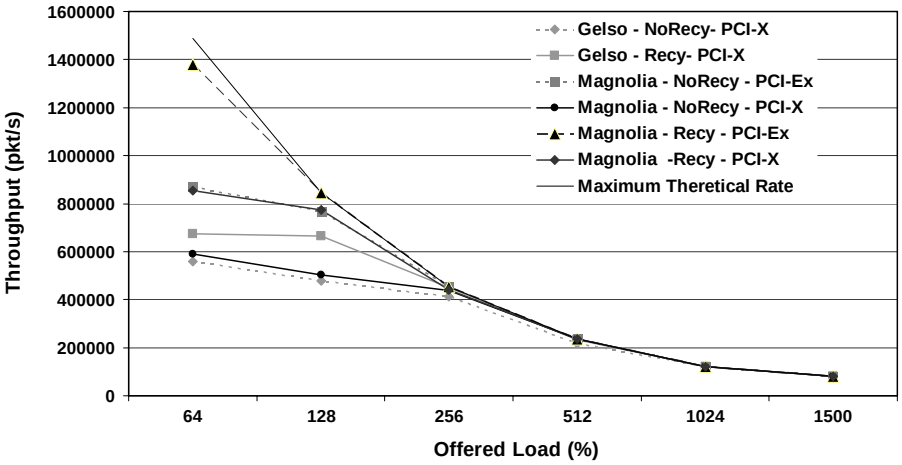

Fig. 1. Maximum forwarding rates according different packet sizes (layer 2 sizes).

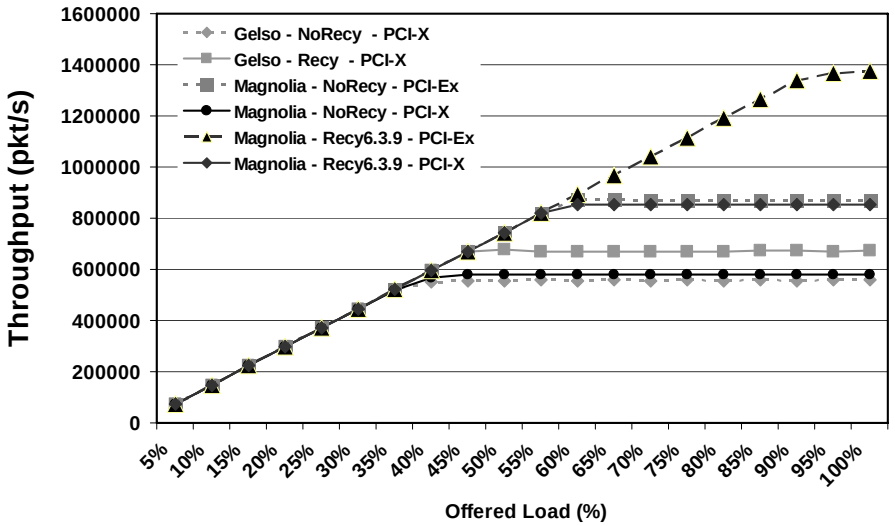

Fig. 2. Forwarding rate according different traffic load values with 64 Bytes sized packets. The traffic load percentage is calculated on the Ethernet Gigabit speed.

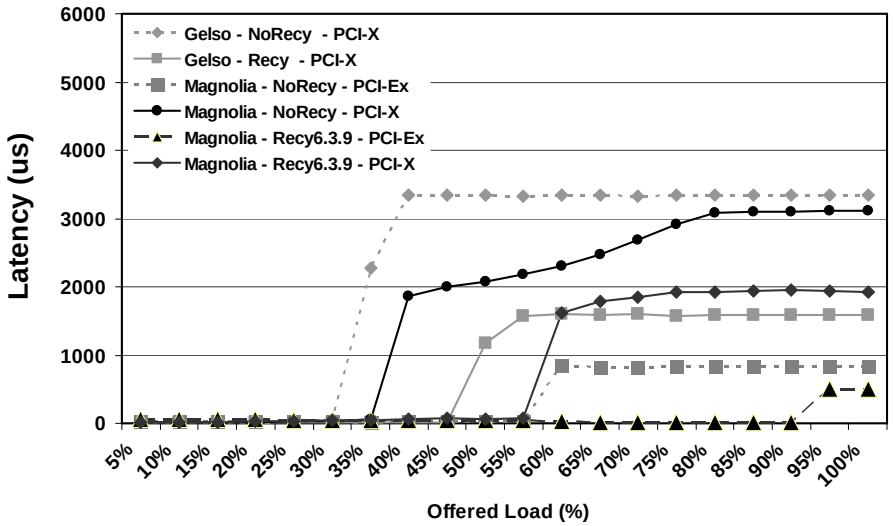

Fig. 3. End-to-end latency values according different traffic loads with 64 Bytes sized packets. The traffic load percentage is calculated on the Ethernet Gigabit speed.

This is also evident by observing the latency values shown in Fig. 3, where the curves exhibit the classical "step" shape (i.e. the step position coincides with the saturation point, after which the buffers fill up): forwarding through a PCI Express bus allows to halve the latency at saturation with respect to the PCI-X tests.

Moreover, since packet latency at saturation mostly depends on the average packet processing time, a lower latency is somehow an index of a lower computational weight of forwarding process. In fact, the skb recycling patch permits to further lower the packet latency with respect to the standard kernel, too.

Figs. 4 and 5 show the internal measurements obtained with Oprofile with Gelso and with both standard and optimized kernels. All the kernel profiled kernel functions have been grouped in different sets that represent the most important activities. The sets that we have selected are "idle", (OS) "scheduler", "memory", "IP processing", "NAPI", "TxAPI", "IRQ", "Ethernet processing" and "oprofile", respectively.

Comparing Figs. 4 and 5, we can note how memory management functions rise more slowly in the optimized kernel case till they achieve the saturation (the 35% of the full Gigabit load for the standard kernel, and about the 50% for the patched one).

For what concerns the other functionalities, their behaviour is clearly bound with the number of forwarded packets: the weight of almost all the classes raises linearly up to the saturation point, and after it remains more or less constant.

On the contrary, the IRQ handlers show a particular behaviour, caused by the NAPI paradigm: when the traffic load is low their computational weight is high, since the Rx API works like an interrupt mechanism; while, for higher loads, it starts to lower more and more. When IRQ weight becomes zero, the OR reaches the saturation point, and works like polling mechanism.

Fig. 6 reports the Oprofile results in the 1500 Bytes packet case. With respect to the 64 Bytes packets, the CPU utilization of IRQ handlers remains quite high for a wide range of traffic offered load (up to the 80-90%). This because in presence of such packet size, the kernel has to process a lower packet number per second, and NAPI begins to behave as a polling mechanism only for higher rates. Moreover, when NAPI works as an interrupt mechanism (e.g., one packet at each IRQ), the CPU idle is maintained over the 10%; while, when the packet rate rises and the forwarding process requires more computational resources, CPU idle lower to the 0%. All the operations performed per packet (IP and Eth. Processing) are obviously lower than in the 64 Bytes packet case

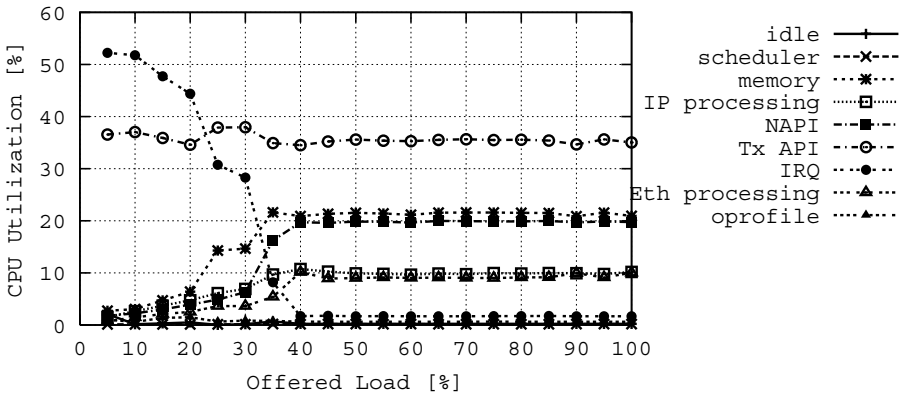

Fig. 4. CPU utilization of the kernel networking functionalities according different traffic offered load (64 Bytes sized packets) for Gelso with the standard kernel.

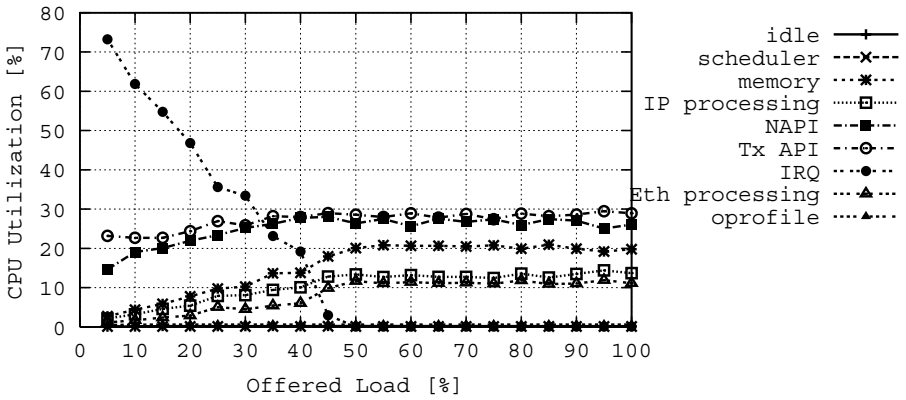

Fig. 5. CPU utilization of the kernel networking functionalities according different traffic offered load (64 Bytes sized packets) for Gelso with the skbuff recycling patch.

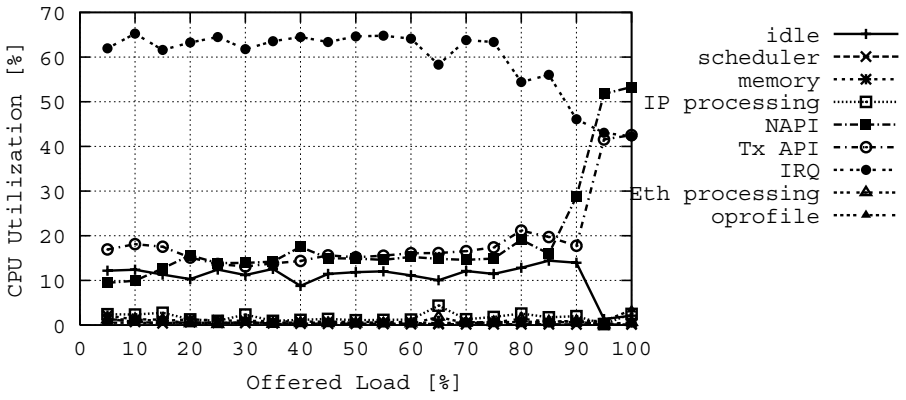

Fig. 6. CPU utilization of the kernel networking functionalities according different traffic offered load (1500 Bytes sized packets) for Gelso with the skbuff recycling patch.

This is confirmed also by the performance counters reported in Figs. 6 and 7, where we can view both the tx and rx boards reducing their IRQ generation rates, while the kernel passes from polling the rx ring twice per received packet, to about 0.22 times. Also the number of Rx SoftIRQ per received packet decreases as offered traffic load raises. Moreover, as shown in [7], a so high IRQ rate at low traffic offered loads can negatively impact on user plane processes. Since IRQ handlers

are originated by PCI boards, and they can interrupt any other kernel/user activities without being interrupted by anything, a high number of interrupts can result in a computational overhead not controlled by the OS, with a consequent performance deterioration of any other software activities. For what concerns the transmission dynamics, Fig. 7 shows very low function occurrences: the Tx IRQ routines low their occurrences up to the saturation, while the “wake” function, which represents the number of times that the Tx Ring is cleaned and the Qdisc buffer is served during a Rx SoftIRQ, shows a mirror behaviour: this because when the OR reaches the saturation, all the tx functionalities are activated when the Rx SoftIRQ starts.

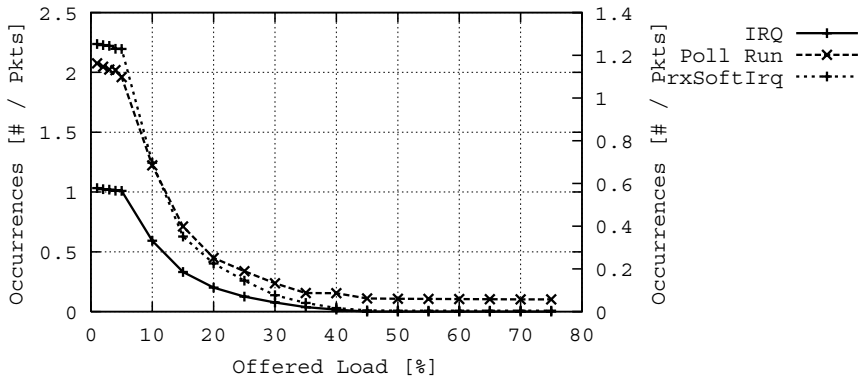

Fig. 7. Number of IRQ routines, of polls and of Rx SoftIRQ (second y-axe) for the RX board for the skbuff recycling patched kernel, in presence of an incoming traffic flow with 64 Bites sized packets.

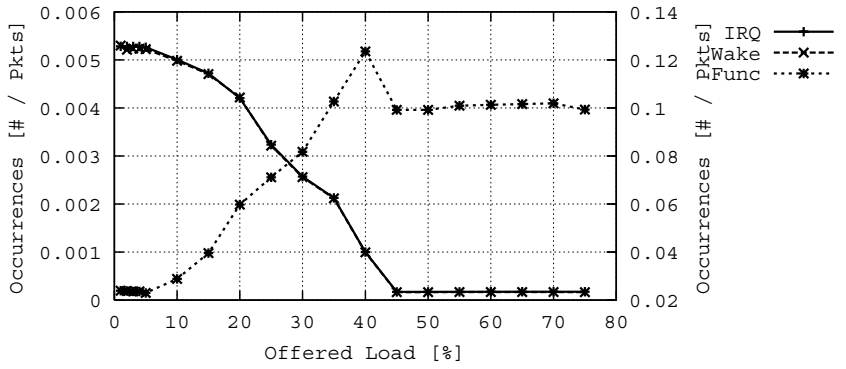

Fig. 8. Number of IRQ routines for the TX board, of Tx Ring clean by TxSoftIRQ (“func”) and by RxSoftIRQ (“wake”) for the skbuff recycling patched kernel, in presence of an incoming traffic flow with 64 Bites sized packets. The second y axe refers to “wake”.

VI. Conclusion

The main contribution of this work has been reporting the results of a deep activity of optimization and testing realized on the Linux networking stack, and, more in particular, on the layers 2 and 3. The main objective has been the performance evaluation (with respect to IPv4 packet forwarding) of both a standard kernel and its optimized version on two different hardware architectures. The benchmarking has been carried out with both external (i.e., throughput and latency) and internal (i.e., kernel profiling and internal counters) measurements. The benchmark results show that, while for large packets sizes almost all the hardware architectures (with high bandwidth I/O busses) can achieve the full Gigabit speed, for small sized packets the maximum performance level generally decays, since there is a bottleneck on the packet processing rate. However, using some kernel enhancements (i.e., skb recycling patch) and PCI Express bus together, the maximum

throughput with minimum sized packets can achieve about 1390 Kpkt/s (nearly the 95% of full Gigabit speed). Besides this considerable performance rise, the profiling results show that packet receiving and transmitting processes generally consumes a not negligible share of CPU time.

Reference

1. A. Barczyk, A. Carbone, J.P. Dufey, D. Galli, B. Jost, U. Marconi, N. Neufeld, G. Peco, V. Vagnoni, "*Reliability of datagram transmission on Gigabit Ethernet at full link load*", LHCb technical note, LHCB 2004-030 DAQ, March 2004.
2. P. Gray, A. Betz, "*Performance Evaluation of Copper-Based Gigabit Ethernet Interfaces*", 27th Annual IEEE Conference on Local Computer Networks (LCN'02), Tampa, Florida, November 2002, pp.679-690.
3. A. Bianco, R. Birke, D. Bolognesi, J. M. Finochietto, G. Galante, M. Mellia, M.L.N.P.P. Prashant, Fabio Neri, "*Click vs. Linux: Two Efficient Open-Source IP Network Stacks for Software Routers*", HPSR 2005 (IEEE Workshop on High Performance Switching and Routing), Hong Kong, May 12-14, 2005
4. A. Bianco, J. M. Finochietto, G. Galante, M. Mellia, F. Neri, "*Open-Source PC-Based Software Routers: a Viable Approach to High-Performance Packet Switching*", Third International Workshop on QoS in Multiservice IP Networks, Catania, Italy, Feb 2005
5. R. Bolla, R. Bruschi, "RFC 2544 Performance Evaluation and Internal Measurements for a Linux Based Open Router", Proc. of IEEE 2006 Workshop on High Performance Switching and Routing (HPSR 2006), Poznan, Poland, June 2006, pp..9-14.
6. R. Bolla, R. Bruschi, "A high-end Linux based Open Router for IP QoS networks: tuning and performance analysis with internal (profiling) and external measurement tools of the packet forwarding capabilities", Proc. of the 3rd International Workshop on Internet Performance, Simulation, Monitoring and Measurements (IPS MoMe 2005), Warsaw, Poland, Mar. 2005, pp. 203-214.
7. R. Bolla, R. Bruschi, "IP forwarding Performance Analysis in presence of Control Plane Functionalities in a PC-based Open Router", Proc. of the 2005 Tyrrhenian International Workshop on Digital Communications (TIWDC 2005), Sorrento, Italy, Jun. 2005, and in F. Davoli, S. Palazzo, S. Zappatore, Eds., "Distributed Cooperative Laboratories: Networking, Instrumentation, and Measuments", Springer, Norwell, MA, 2006, pp. 143-158.
8. Building Open Router Architectures based on Router Aggregation (BORA BORA) project, website at http://www.telematica.polito.it/projects/borabora/.
9. K. Wehrle, F. Pählke, H. Ritter, D. Müller, M. Bechler, "*The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel*", Pearson Prentice Hall, Upper Saddle River, NJ , USA, 2004.
10. J. H. Salim, R. Olsson, A. Kuznetsov, "*Beyond Softnet*", Proc. of the 5th annual Linux Showcase & Conference, Nov. 2001, Oakland California, USA.
11. Skb recycling patch, ftp://robur.slu.se/pub/Linux/net-development/skb_recycling/.
12. The Agilent N2X Router Tester, http://advanced.comms.agilent.com/n2x/products/ index.htm
13. Oprofile, website at http://oprofile.sourceforge.net/news/.