Computer Communications 31 (2008) 3460–3468

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

The internals of advanced interrupt handling techniques: Performance optimization of an embedded Linux network interface

Stergios Spanos [a,*], Apostolos Meliones [b], George Stassinopoulos [a]

[a] School of Electrical & Computer Engineering, National Technical University of Athens, 9 Heroon Polytechneiou Str., 15780 Zografou, Athens, Greece
[b] Department of Digital Systems, University of Piraeus, 80 Karaoli & Dimitriou Str., 185 34 Piraeus, Greece

ARTICLE INFO

Article history:
Received 11 November 2007
Received in revised form 28 May 2008
Accepted 29 May 2008
Available online 7 June 2008

Keywords:
Linux
Embedded processor
Interrupt coalescing
NAPI
Performance optimization

ABSTRACT

Linux over the past few years has gained in popularity as the OS fit for embedded networking equipment. Its reliability, low cost and undisputed networking capabilities made it one of the most popular choices for the networking market. As access interfaces become faster and network applications more sophisticated, a lot of effort has been focused on the improvement of Linux networking performance. This paper analyzes the routing performance improvement of an embedded communications processor by endorsing advanced interrupt handling techniques. Even though these techniques are well known and are popular on a variety of networking hardware, their combination is not so often used. Moreover, their performance evaluation is poorly analyzed and documented. Even though the analysis is based on an embedded Linux system, these techniques are not hardware specific, and can be applied to a wide variety of networking systems.

1. Introduction

Embedded networking systems have gained a significant share of the networking market over the past few years. Among their main advantages are the reduced cost that they can offer and the ability to be tailored for specific types of applications [1]. However, due to their limited resources embedded networking systems require efficient software design. Linux [2] is particularly popular in network embedded systems because of its scalability [1], low cost and the superior networking capabilities [3]. The Linux IP stack has proven itself to be stable, efficient and well suited for networking applications and services that require high reliability and availability.

The need for efficiency in networking software has driven the developers' community to focus on making the Linux networking stack faster and able to handle the increasing amounts of data traffic. In this context, advanced interrupt handling techniques have been implemented and applied to series of Linux device drivers. Probably the best known technique is NAPI which has been applied to a large number of network device drivers. The internal procedures that are activated through NAPI are however poorly documented and experimental data supporting the theoretical NAPI background is quite rare. Apart from NAPI, there are as well different interrupt handling techniques that can be applied.

The paper focuses on the routing performance of the Freescale MPC82xx family of processors [4] running embedded Linux. The aim of this paper is to improve the overall routing capabilities of an embedded networking system by incorporating advanced interrupt handling techniques to the Linux device driver of its Fast Ethernet Interfaces. These modifications to the device driver can be applied to great variety of network device driver supported by Linux. We selected an embedded system since the need for improved performance is much more imperative in this kind of systems. Their limited hardware capabilities (compared to a conventional networking system) impose that the software should be as efficient as possible. We present a comprehensive analysis of each technique and the way it enhances the system's routing performance. We provide the experimental data and the theoretical model that describe the internal procedures which allow the performance improvement.

The paper is organized as follows: Section 2 provides the description of the existing Fast Ethernet device driver in the current Linux kernel distribution. Section 3 describes the alternative interrupt handling techniques. Section 4 presents the performance improvement that these changes incur to the forwarding performance of the reference system. Section 5 provides the internal procedures that each interrupt handling technique triggers and presents the theoretical model that sufficiently describe the internal procedures of these techniques.

2. The Linux Fast Ethernet device driver structure

The MPC82xx Fast Ethernet device driver follows a typical Linux network device driver structure: As a data packet arrives to the

* Corresponding author. Tel.: +30 2109817756.
E-mail address: sspan@telecom.ntua.gr (S. Spanos).

0140-3664/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2008.05.036

interface, an interrupt to the processor is generated. The Linux Interrupt Service Routine (ISR) of the driver assigned to the interface is called. It acknowledges the event, copies the packet to the system's backlog queue and informs the kernel that a network frame has been received. At some later point a kernel thread is awakened in order to process the frame. If the frame's destination is the system, it provides with the frame data to the appropriate system procedure. If the frame needs to be forwarded, it is transferred to the egress queue of the appropriate interface.

As far as the outgoing part of the device driver is concerned, the device driver receives the data from the OS and provides it to the appropriate structure of the network interface. The network controller takes over and transmits the data. After the frame has been transmitted, an interrupt is generated by the corresponding Ethernet interface. The driver ISR takes over once again. Its main task is to call the `dev_kfree_skb_irq` function in order to free the (no longer needed) socket buffers related to the transmitted frame. It also makes a series of control checks, updates the transmit buffer descriptor ring pointers and stores statistical data. Fig. 1 depicts this device driver model.

Under heavy load of traffic, this interrupt handling scheme may lead to system congestion. In an interrupt-driven system, hardware interrupts take priority over all other system activity. For small sized packages, the system may be forced to receive and forward up to 148 K of frames per second and per Fast Ethernet interface. Performance analysis for this type of device drivers has revealed that above a certain rate of incoming frames, the system's throughput falls dramatically due to congestion. The main reason for the congestion collapse effect is the interrupt receive livelock [5]: this is the state where no useful progress is being made, because the necessary resources are entirely consumed with interrupt handling. The kernel thread responsible for processing and forwarding the IP packets is not called and in effect, the backlog queue of the IP stack overflows and packets are dropped. It will therefore have no resources left to support delivery of the arriving packets to applications (or, in the case of a router, to forwarding and transmitting these packets). The useful throughput of the system will drop to zero. When the network load drops sufficiently, the system leaves this state, and is again able to make forward progress; this is why the state is characterized as a "livelock" instead of a "deadlock".

3. Advanced interrupt handling techniques

This section presents the description of the interrupt handling enhancements that can be applied to the device driver.

3.1. Reducing receive interrupts – the NAPI approach

The Linux NAPI (New API) driver interface, moderates the impact of heavily loaded network interfaces on overloaded systems [7]. It is based on early proposals to eliminate receive livelock in interrupt-driven system [5].

NAPI's objective is to minimize the number of incoming frames interrupts (Rx Interrupts). Network interfaces are set to interrupt upon the reception of the first incoming data packet. While serving this event, the ISR disables Rx Interrupts and notifies the kernel that there is an incoming packet waiting to be processed. At a later point, a software interrupt (softirq) is activated to poll all the devices that have registered to offer packets. Softirqs [6] are a Linux kernel mechanism for deferring non-time-critical tasks and taking them out of the ISR. They can execute with all interrupts enabled, helping in this way the system to keep the kernel response low.

Once all of the pending packets have been served the interface re-enables its interrupts for incoming traffic. However, only a certain amount of packets (quota) is granted to each interface allowing it to send as much packets. If the quota is exceeded and there are still packets to be processed, the interface backs off and returns the control to the OS. The ISR is scheduled to be called again in order to continue processing the remaining incoming traffic. The NAPI approach has the effect of gracefully moving to a polling regime when the system is fully utilized and to a low-latency interrupt-driven regime when the system is lightly loaded. NAPI has been already implemented as an option for a series of commercial Fast Ethernet and Gigabit Ethernet interface controller cards under Linux [8].

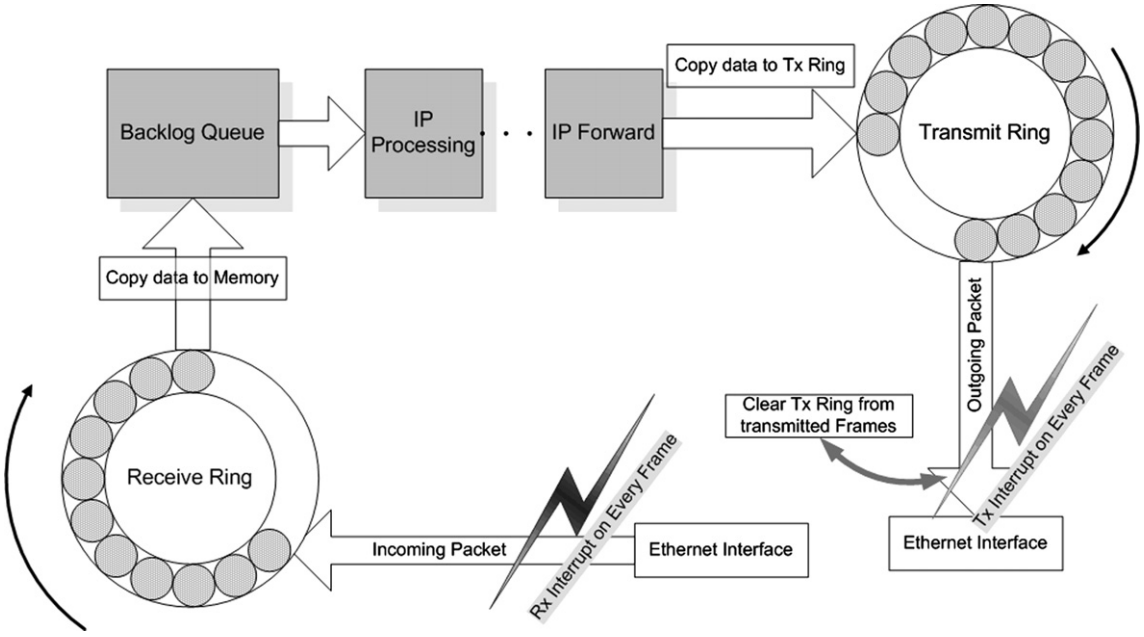

Fig. 1. Original Fast Ethernet device driver structure.

3.2. Moderating transmit interrupts

Even though the NAPI approach can be also applied to the outgoing traffic of a network interface, it is mainly implemented in incoming data direction for most of the device drivers. The NAPI approach assumes that the major "interrupt source" is the incoming traffic. There is, however, a major category of systems that (in their native Linux device driver implementation) generate interrupts upon the transmission of every outgoing packet (Tx Interrupts). For instance, the embedded PowerPC-based systems (e.g. MPC82xx, MPC8xx) use Fast Ethernet device drivers use this interrupt scheme. When these systems are acting as network routing elements, the majority of arriving packets will be forwarded. Therefore the number of Tx interrupts is expected to be equivalent to the number of Rx interrupts. For these systems the moderation of the Tx interrupts is of equal importance to the Rx interrupts reduction.

Two different methods can be used in order to moderate Tx interrupts. The first is to modify the way outgoing traffic is generated so that a single interrupt serves a certain amount of transmitted packets or a maximum rate of interrupts per second. The method is referred to as *Tx interrupts coalescing*. In this way the system is relieved of the continuous interrupt context switching. Once the Tx interrupt arrives however, the system needs to scan and free all of the socket buffers related to the already transmitted packets, by calling the `dev_kfree_skb_irq` function repeatedly. In our approach, the modified device driver with the moderated Tx interrupts, will generate one interrupt after 128 packets have been transmitted successfully.

A second, more radical approach is to completely disable Tx interrupts. In this approach, an alternative way has to be defined so that the system executes all the necessary tasks (which are interrupt triggered in the original version of the device driver) when the data packets are being transmitted. In our implementation, these tasks are performed just before each data packet transmission. The driver inspects the Tx ring and releases the socket buffers related to packets that have already been transmitted. It prepares the data of the Tx packet as in the original driver with the sole difference that the packet is not set to interrupt after its transmission. In this way, the system, as in the Tx interrupt coalescing scheme, is relieved of the Tx interrupt context switching load. Since socket buffers are freed before the preparation of each Tx packet, the driver balances its work load more rationally: the time-consuming task of managing the socket buffers is performed more frequently and for a smaller amount of frames than in the Tx interrupt coalescing scheme. Moreover, since the task is not performed inside an interrupt context, `dev_kfree_skb` is called instead of `dev_kfree_skb_irq`; a detail that can greatly modify the system's performance.

There are two main differences between the `dev_kfree_skb` and `dev_kfree_skb_irq` functions. The first is that during the call of `dev_kfree_skb_irq` the system's hard and soft interrupts have to be disabled. Therefore, while the system is busy freeing the skbs, it cannot serve any incoming or outgoing traffic interrupt events. The second difference is that `dev_kfree_skb_irq` does not immediately free the socket buffers (as `dev_kfree_skb` does) but it rather schedules the task to be performed later in time and to be triggered by a softirq. In this way one can observe that scheduling through sorftirqs can become quite popular for ISR functions. In this manner the system tends to "overload" its softirq subsystem. Using the `dev_kfree_skb` function to free the socket buffers, allows the system to continue to receive incoming frames for that period of time. The Tx interrupts deactivation method is already implemented on standard Linux device drivers (e.g. the SysKonnect 98xx Gigabit Ethernet device driver [9]).

3.3. Combining the Rx & Tx interrupt handling techniques

The advanced interrupt handling techniques, described in the previous sections, have already been applied to a variety of drivers, but they are mainly used on Gigabit Ethernet interfaces and definitely not on embedded systems. Moreover, there are very few drivers that implement combined Rx & Tx interrupt moderation schemes, while the performance implications of these modifications are not available.

In order to investigate these implications, we applied all of the above interrupt handling techniques to the MPC82xx Fast Ethernet Linux device driver and created device driver versions which incorporate these three modifications (the NAPI modification, the Tx Interrupt deactivation and the Tx Interrupt moderation). Going one step further, we created driver versions that implement combined Rx & Tx interrupt moderating techniques. Fig. 2 presents the modified driver structure with NAPI enabled and the Tx Interrupts disabled.

4. Performance evaluation

In this section we present the experiment results for the forwarding performance of the system and analyze the mechanisms that influence the performance variance.

4.1. Device driver variants and test bed setup

Six different variants of the 2.6.9 Linux device driver were used:

- Original device driver (referred to as *ORIGINAL*).
- Moderated Tx Interrupts device driver. The driver generates only 1 Tx interrupt after 128 transmitted packets (referred to as *TX-128*).
- Deactivated Tx Interrupts device driver. No Tx interrupts occur (referred to as *TX-OFF*).
- NAPI enabled device driver. Only the NAPI modifications have been applied on the driver (referred to as *NAPI*).
- NAPI & Moderated Tx Interrupts device driver. The driver incorporates the NAPI modifications and generates only 1 Tx interrupt every 128 transmitted packets (referred to as *NAPI-TX-128*).
- NAPI & Deactivated Tx Interrupts. The driver incorporates the NAPI modifications and generates no Tx interrupts (referred to as *NAPI-TX-OFF*).

The test bed used for the experiments consists of a reference system based on the MPC8250. It is running the standard Linux 2.6.9 kernel version which has was compiled in non-preemptive mode.

The MPC8250 communications processor consists of two major units: The main core (CPU) and the Communications Processor Module (CPM). The CPU is the processor's functional block which takes care of all the basic operating system operations, while the CPM is the individual part of the processor that interacts directly with the network interfaces. The clock speeds for the CPU/CPM/System-bus were set to 200/133/66 MHz, respectively. The Linux network stack, besides the Fast Ethernet device driver, was left intact. In all of the NAPI based drivers the NAPI-weight value (quota) was set to 32. The size of the Rx and Tx ring were set to 256 in all of the experiments.

The system incorporates two identical but independent Fast Ethernet Interfaces (eth0 and eth1), both using the same device driver. Each interface is served by a dedicated interrupt request line. On each interface, the system receives the frames, processes them and forwards them to the other Fast Ethernet interface.

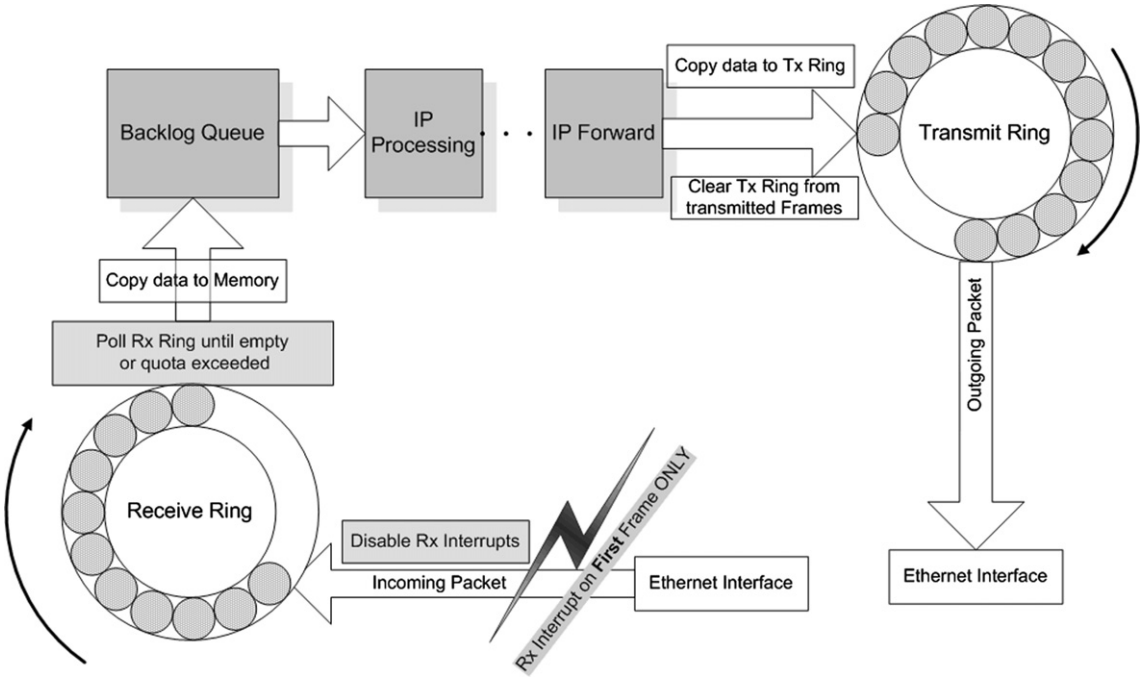

Fig. 2. Modified driver structure with NAPI enabled & Tx Interrupts fully disabled.

For the throughput and packet loss measurements the Spirent SmartBits 2000 Performance Analysis System [10] was used. Each one of the system's Fast Ethernet interfaces is connected to a ML-7710 SmartCard attached to the SmartBits 2000. The experiments were conducted using the SmartWindow and Smart-Applications software packages (Version 2.54). Smart-Applications conforms to RFC 1242 [11] and RFC 2544 [12]. Fig. 3 depicts the experimental testbed.

All of the performed tests were *bidirectional* i.e. both Ethernet interfaces of the system were simultaneously receiving and transmitting traffic. The bidirectional scenario was selected against the unidirectional scenario, in order to evaluate the system's performance under the heaviest possible network load. The reported results are the average of 5 trials, each one lasting 60 s. Constant load (i.e. frames with fixed length) was applied in all cases. Although it is rare to encounter a steady state load on a network device in the real networking world, measurement of steady state performance is useful when evaluating a networking system's performance. The running processes of the reference system are minimized and limited only to a few system and kernel processes and daemons (such as sh, inetd, syslogd) which are either in a sleeping state or relate only to the trials themselves. Since the system is not running any other tasks, we can safely assume that in the case of a device driver which is not able to achieve full speed performance

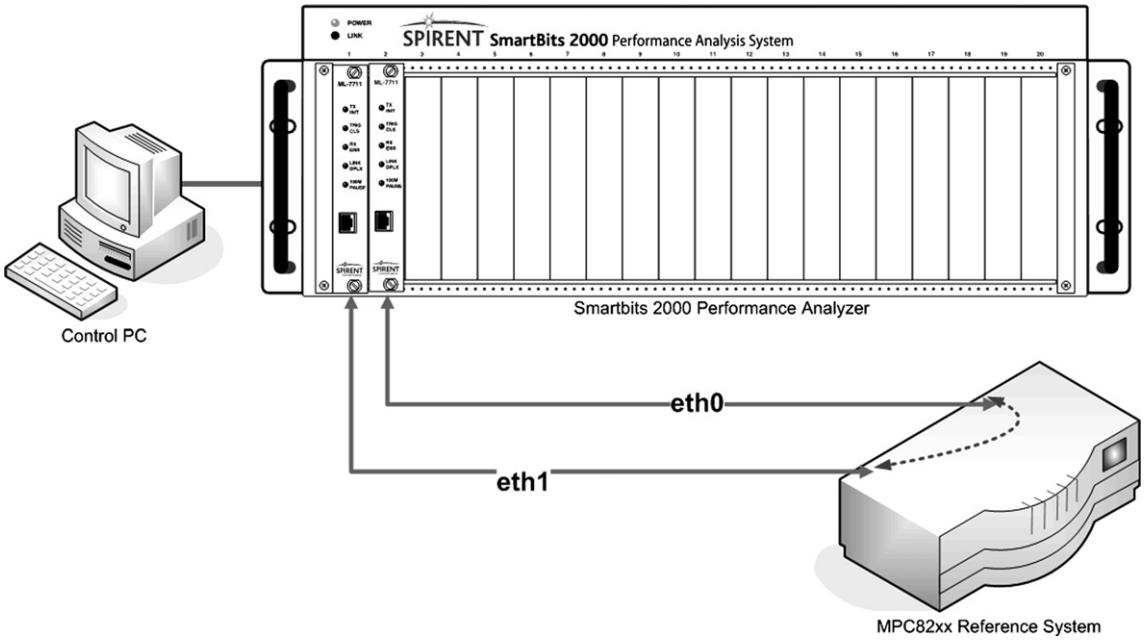

Fig. 3. Experimental testbed.

for a given packet size, the system dedicates the vast majority of its time for the processing of the network traffic.

The tests were repeated for a series of packet sizes from the minimum (64 bytes) to the maximum (1518 bytes) Fast Ethernet packet size. Their comparison however can be objective only for sizes up to 512 bytes. For frame sizes greater than 512 the NAPI-TX-OFF driver can forward 100% of the incoming traffic.

4.2. Experimental results and analysis

4.2.1. Maximum throughput experiment results

System throughput is the maximum non-blocking rate (on each interface) at which all of the offered frames (Rx and Tx) are received and transmitted without being dropped. Fig. 4 presents the throughput for the six different driver variants.

All of the modified driver versions present improved throughput. There is however significant variance of the improvement percentage for each of the device driver variants. Table 1 depicts the throughput performance of the six device driver variants, while Table 2 presents the percentage of improvement that each modified device driver offers, when compared to the original Linux driver performance.

Based on the results of Tables 1 and 2, we can make the following observations for each of the driver variants.

The TX-128 driver provides the smallest improvement. This is expected since the driver generates only one interrupt for every 128 transmitted frames. However, upon the interrupt occurrence, the driver has to perform the time-consuming task of releasing the socket buffers consecutively for all of the 128 transmitted frames. During this time period the system interrupts are disabled and therefore the system cannot receive any frames. The improvement occurs due to the reduced interrupt context switching.

The NAPI and TX-OFF versions of the driver, provide similar improvement to the system's performance. They both perform much better than the TX-128 driver, since in both cases the interrupts (Rx and Tx, respectively) are disabled. This improvement directly suggests that the moderation of Tx and Rx interrupts is of equal significance.

Table 1
Throughput performance of the 6 device driver variants

Packet size (bytes)	ORIGINAL (Mbps)	TX-128 (Mbps)	TX-OFF (Mbps)	NAPI (Mbps)	NAPI-TX-128 (Mbps)	NAPI-TX-OFF (Mbps)
64	12.38	13.14	15.00	15.95	17.83	25.85
128	20.96	21.89	25.00	25.78	28.14	42.65
256	35.52	37.27	43.46	41.50	46.86	65.71
512	58.95	61.01	68.74	69.54	73.84	96.64
896	83.89	86.50	95.22	92.95	96.17	100.00
1024	90.66	93.59	99.67	100.00	100.00	100.00
1280	100.00	100.00	100.00	100.00	100.00	100.00
1518	100.00	100.00	100.00	100.00	100.00	100.00

Table 2
Performance improvement (%) in comparison to the original driver

Packet size (bytes)	TX-128 (%)	NAPI (%)	TX-OFF (%)	NAPI-TX-128 (%)	NAPI-TX-OFF (%)
64	6.10	21.16	28.87	44.05	108.77
128	4.44	19.26	23.00	32.46	103.46
256	4.92	22.36	16.84	31.91	85.00
512	3.50	16.61	17.98	25.26	63.95
896	3.12	13.51	10.41	14.64	19.21

The combined NAPI-TX-128 driver is slightly faster than the plain NAPI version. This is expected since the driver takes advantage of the feature that both variants offer: the system is relieved of the Rx interrupts, and the Tx interrupts are also considerably moderated.

The combined NAPI-TX-OFF driver version, however, adds a real boost to the system's performance. No Tx interrupts occur and the Rx interrupts are nearly zeroed. The release of the socket buffers is balanced and performed without disabling the system's hard and soft interrupts.

4.2.2. System performance under heavy traffic load

A second significant improvement to the system's overall performance due to the interrupt handling modifications, is the system's overall enhanced stability. Since the number of interrupts

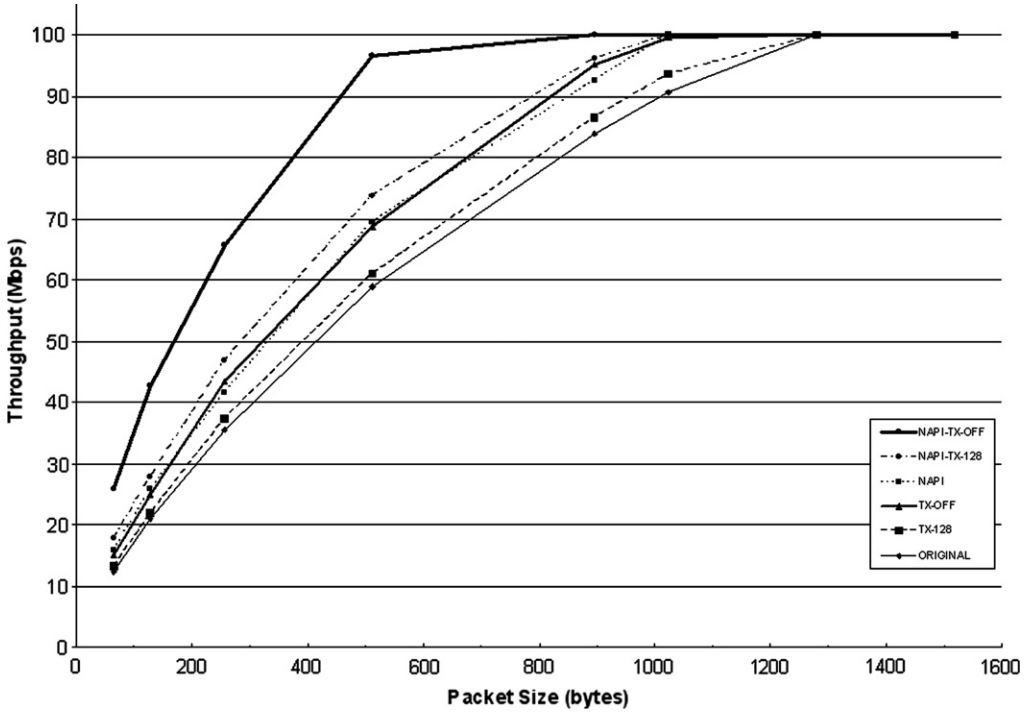

Fig. 4. System maximum bidirectional throughput for device driver variants.

Table 3
Number of generated interrupts while forwarding 2 M frames (512 byte @ 100 Mbps)

Driver version	Received frames	Transmitted frames	Rx Ints	Tx Ints
ORIGINAL	2,000,000	523,778	1,973,554	517,596
TX-OFF	2,000,000	540,419	1,966,224	0
NAPI ONLY	1,272,263	1,272,263	5	1,271,962
NAPI-TX-OFF	1,889,990	1,889,990	7	0

is reduced, the system is relieved of the continuous interrupt serving overhead and is able to utilize more efficiently its resources.

Table 3 depicts the number of generated interrupts for four variants of the device driver (ORIGINAL, NAPI, TX-OFF and NAPI-TX-OFF) while serving 1 M 512-byte packets that arrive into the system per interface (2 M in total) at full rate (100 Mbps). We recorded the number of received and transmitted frames. We characterize as "Received" all the packets that arrived to the system's network controller and were successfully copied to memory in order to be processed by the kernel. Drivers that do not use NAPI (ORIGINAL & TX-OFF) are capable of "receiving" all of the incoming frames. However, the high interrupt rate keeps the system so busy that the main executed task is actually the packet reception (congestion effect). The forwarded frames for the non-NAPI versions of the kernel are therefore limited.

Drivers that use NAPI are more balanced. Because of NAPI, the excessive traffic is not received but it is dropped at the Rx ring without any Linux intervention. All of the received packets are being forwarded. The Rx interrupts for both NAPI enabled drivers (NAPI & NAPI-TX-OFF) are extremely low. Moreover, the NAPI-TX-OFF not only needs just 7 Rx interrupts to receive the 2 M packets, it does not trigger any Tx interrupts. This is the reason why the

Table 4
Mathematical model values for device driver variants

Driver version	Constant time (μs)	K (μs/byte)
ORIGINAL	25.8	0.02
TX-128	24.5	0.02
TX-OFF	20.5	0.02
NAPI ONLY	20.5	0.02
NAPI-TX-128	18.3	0.02
NAPI-TX-OFF	11.5	0.02

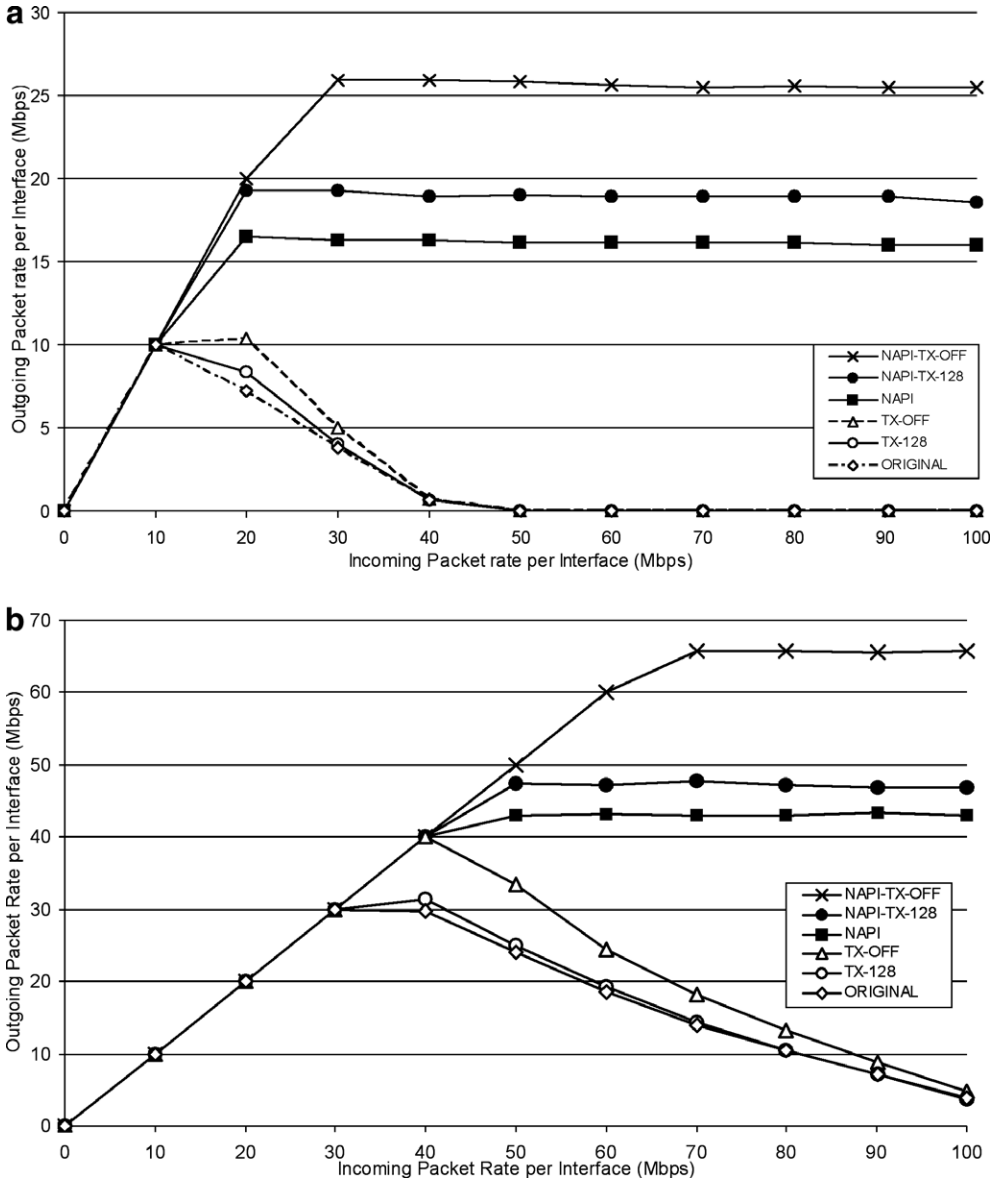

Fig. 5. Congestion collapse effect elimination, using NAPI based device drivers for packet size: (a) 64 bytes and (b) 256 bytes.

NAPI-TX-OFF driver presents improved performance (more than 48% compared to the plain NAPI driver) while sustaining the same stability levels.

Another aspect of the system's performance improvement is the elimination of the receive livelock. Fig. 5 depicts the congestion collapse effect for 64 and 256 byte frames and the way this is elim-

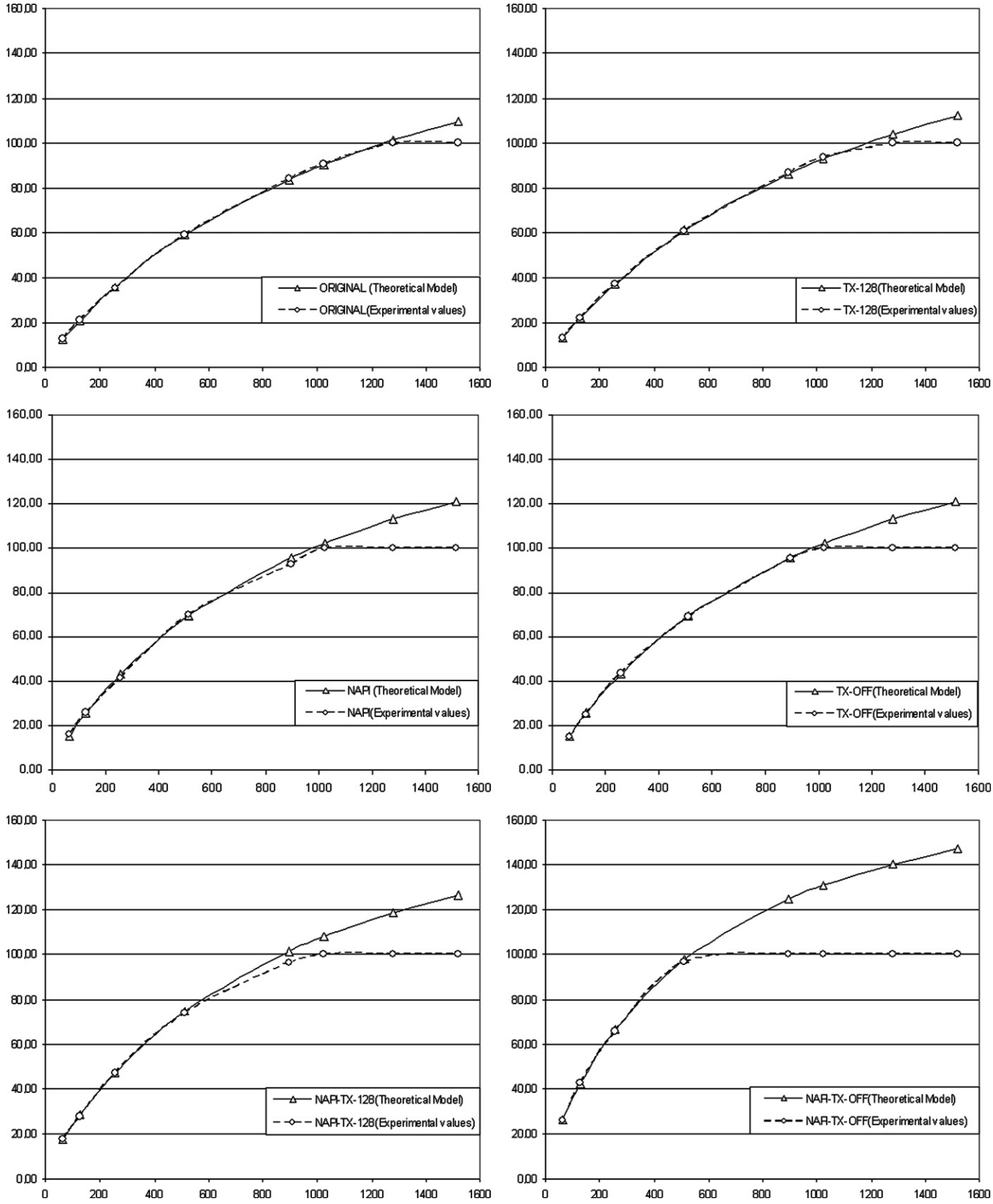

Fig. 6. Maximum system throughput based on the experimental values and the theoretical model.

inated using advanced interrupt handling techniques. The Original, TX-OFF and TX-128 drivers cannot keep up with the rate of incoming frames. When this rate exceeds the maximum non-blocking throughput the driver can serve, the system's performance starts to degrade. On the other hand NAPI, NAPI-TX-128 and NAPI-TX-OFF drivers retain a maximum stable forwarding rate, regardless of the increase in the rate of incoming frames. The Original Linux device driver reaches the point of congestion collapse when the overall incoming traffic reaches 120 K packets per second. The Maximum Loss Free Forwarding Rate for the Original driver is 37 K packets per second and 77 K packets for the NAPI-TX-OFF driver (for 64 byte frames). This is the rate that the NAPI-TX-OFF driver can maintain regardless of the incoming traffic rate without reaching a congestion collapse point This suggests that system is more stable and that resources are better used.

Moreover, the combined NAPI and Tx Interrupt moderation techniques drivers (NAPI-TX-OFF and NAPI-TX-128), not only retain the stability of the NAPI driver, but also significantly improve the overall system performance. The NAPI-TX-OFF driver increases the overall system throughput over 55% (53%) for 64 byte (256 byte) frames compared to the NAPI driver.

5. The performance improvement internals

As it is explained in [5], the processes executed by the device driver in order to perform the necessary processing tasks can be separated into two different types: the constant time and the variable time processes. The constant time processes are the ones whose duration is independent of the incoming traffic packet size. In this category, we include the interrupt context switching, the trivial procedures that are performed after an interrupt occurs and the process which takes care of the IP stack manipulation of the packet as it only checks with the headers of the frames. The variable time processes on the other side depend on the size of the incoming traffic. The process that copies the data from (to) the Rx (Tx) Ring to (from) the memory can be included in this category. Based on the above proposal, we have extracted a simple linear mathematical model which accurately complies with the actual system performance. The mathematic formula is:

$$T_{TOTAL} = T_{CONSTANT} + k \cdot S$$

S is the size of the packet processed in bytes and k is a constant which characterizes the system. It specifies the time needed (in μs) for the processing per byte of network traffic. Table 4 depicts the $T_{CONSTANT}$ & k values for all of the 6 driver versions.

In Fig. 6 we present the maximum system throughput based on the actual experimental values and the mathematical model, for all of the six device driver variants. The theoretical model values as it is expected are in total agreement with the real experimental values only for the packet sizes that the system throughput is less than the 100% of the maximum interface speed.

Having validated the accuracy of the mathematical model, we can use it in order to estimate the actual system performance for the various packet sizes. Fig. 7 depicts the maximum bidirectional throughput performance for each of the device drivers based on the theoretical model. It is obvious that the combined TX and RX interrupt handling techniques significantly improve the system's performance even in the case of large sized packets.

Going to detail as far as the model is concerned, we observe that the overall improvement of the performance for the combined NAPI-TX-OFF driver is far greater than the added improvement the two individual drivers (NAPI & TX-OFF). This can be explained if we take into account a series of factors:

1. The NAPI and TX-OFF device drivers present a reduced interrupt occurrence rate compared to the Original driver. However this rate is still high and causes the processing of each arriving packet to be inevitably disrupted. The interrupt rate for the combined NAPI-TX-OFF driver is nearly zeroed under high traffic. In this way the packet processing procedures run uninterrupted and therefore the system's scheduler is the one which controls the way each task is executed. The systems resources are utilized in amore efficient way since they are decoupled of non-deterministic events such as the occurrence of an interrupt.
2. The total time that the system is able to receive interrupts is greatly increased in the combined NAPI-TX-OFF driver. In the highly demanding environment of the embedded networking systems, this parameter can become quite important especially under heavy traffic load conditions. While the system is serving high network traffic, it is important to minimize the time the system cannot receive interrupts even in the case of the drivers that the interrupt occurrence rate is not intensive. The modified drivers not only reduce the interrupts but also increase the time window that these interrupts can be acknowledged as quickly as possible by the μP.

Table 5 presents the total time needed to process a frame that arrives to the system through one of the Fast Ethernet Interface and will be routed to the other. While the system is serving high

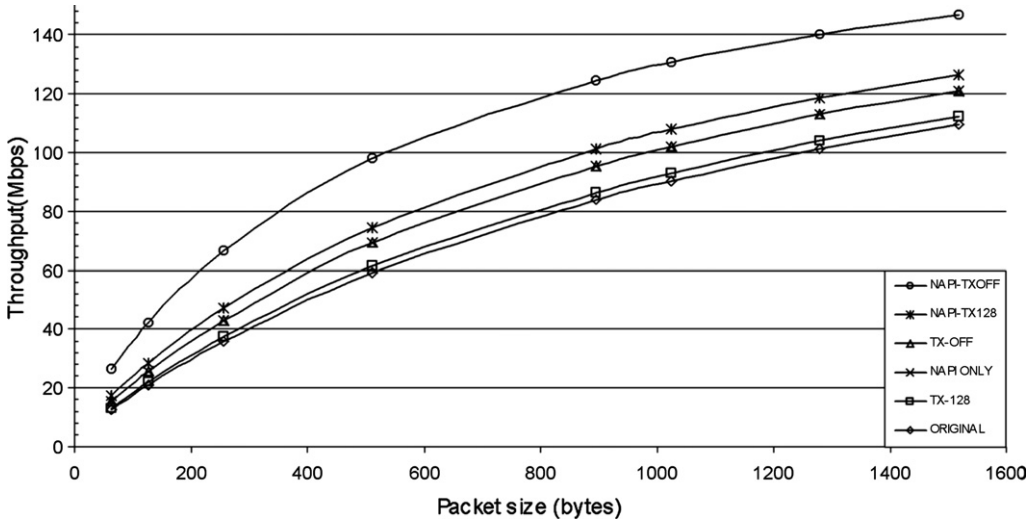

Fig. 7. Device driver variant maximum throughput based on the theoretical model.

Table 5
Total packet process time, interrupts enabled time and interrupt rate for the device driver variants

Driver	Total packet process time	Interrupts enabled time	Interrupt rate
ORIGINAL	$2 \cdot (T_{CS(IN)} + T_{CS(OUT)}) + T_{RX} + T_{COPY} + T_{IP} + T_{TX} + T_{SKB}$	T_{IP}	$2R$
TX-128	$\frac{129}{128} \cdot (T_{CS(IN)} + T_{CS(OUT)}) + T_{RX} + T_{COPY} + T_{IP} + T_{TX} + T_{SKB}$	T_{IP}	$(\frac{129}{128})R$
TX-OFF	$T_{CS(IN)} + T_{CS(OUT)} + T_{RX} + T_{COPY} + T_{IP} + T_{TX} + T_{SKB}$	$T_{IP} + T_{TX} + T_{SKB}$	R
NAPI	$T_{CS(IN)} + T_{CS(OUT)} + T_{RX} + T_{COPY} + T_{IP} + T_{TX} + T_{SKB}$	$T_{RX} + T_{COPY} + T_{IP}$	R
NAPI-TX-128	$\frac{1}{128} \cdot [T_{CS(IN)} + T_{CS(OUT)}] + T_{RX} + T_{COPY} + T_{IP} + T_{TX} + T_{SKB}$	$T_{RX} + T_{COPY} + T_{IP}$	$(\frac{1}{128})R$
NAPI-TX-OFF	$T_{RX} + T_{COPY} + T_{IP} + T_{TX} + T_{SKB}$	$T_{RX} + T_{COPY} + T_{IP} + T_{TX} + T_{SKB}$	$\simeq 0$

traffic and cannot route 100% of the incoming traffic, it is safe assume that all of the μP time is devoted to the incoming and outgoing packets processing. In Table 5 we identify the following time periods while processing a data packet:

- $T_{CS(IN)}$ and $T_{CS(OUT)}$ refer to the time needed for the OS context switching into and out of an ISR.
- T_{RX} refers to the time needed for the trivial tasks (logical tests, statistical counters update etc.) that take place during the reception of a data packet.
- T_{COPY} is the time needed to copy the data to the main memory. This time depends on the data size of the packet.
- T_{IP} is the time for the IP manipulation of the packet.
- T_{SKB} refers to the time that is needed to free the socket buffer structures connected to the frame processed.
- T_{TX} refers to the time needed for the trivial tasks (logical tests, statistical counters update etc.) that take place during the transmission of a packet.

From the time values of Table 5, we can observe that the difference between the constant time needed to process a frame is quite similar in all of the cases of the device driver variants. They mainly differ as far as the T_{CS} is concerned. There is however a great difference as far as the interrupts rate is concerned in each case. Moreover the time portion during which the system is able to receive interrupts is much greater in the combined NAPI-TX-128 & NAPI-TX-OFF drivers. The system is able to receive interrupts even during the time-consuming tasks of data copy or (in the case of NAPI-TX-OFF) during the skb manipulation procedure. This is an additional factor which in combination with the low interrupt rate allows the system to perform more efficiently.

6. Conclusions

We created six different Fast Ethernet device driver versions of an MPC82xx embedded Linux system incorporating advanced interrupt handling techniques in order to improve its overall routing performance. These techniques involved both Rx & Tx interrupts moderation. We ran a series of experiments and observed an overall performance improvement that can reach 107% while adding substantial stability to the overall system performance. We proposed a simple mathematical model that accurately describes the system overall networking performance and we have explained the internals of each device drivers and the way it improves the overall performance.

References

[1] K. Yaghmour, Building Embedded Linux Systems, O'Reilly, 2003.
[2] L. Torvalds, Linux OS, online. Available from: <http://www.linux.org>.
[3] Benvenuti Christian, Understanding Linux Network Internals, O'Reilly, 2005.
[4] Freescale Communications Processors. Available from: www.freescale.com.
[5] J.C. Mogul, K.K. Ramakrishnan, Eliminating receive livelock in an interrupt-driven kernel, ACM Transactions on Computer Systems 15 (3) (1997) 217–252.
[6] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel, 3rd ed., O'Reilly, 2005.
[7] J.H. Salim, R. Olsson, A. Kuznetsov, Beyond softnet, in: Proceedings of the 5th Annual Linux Showcase & Conference (ALS 2001), Oakland, CA, USA, November 5–10, 2001.
[8] Drivers include Intel, 3com, SMC, RealTek devices. Available in the Linux kernel tree, under:/drivers/net/. <http://www.kernel.org>.
[9] The Linux kernel tree, under/drivers/net/sk98lin. http://www.kernel.org.
[10] Spirent Communications. Available from: <http://www.spirentcom.com>.
[11] S. Bradner, Benchmarking Terminology for Network Interconnection Devices, RFC-1242, Harvard University, Bay Networks, July 1991.
[12] S. Bradner, J. McQuaid, Benchmarking Methodology for Network Interconnect Devices, RFC-2544, Harvard University, Bay Networks, May 1996.