Estimating the Impact of Interrupt Coalescing Delays on Steady State TCP Throughput

Marko Zec, Miljenko Mikuc, Mario Žagar
University of Zagreb, Faculty of Electrical Engineering and Computing
Unska 3, HR-10000 Zagreb, Croatia
E-mail: zec@tel.fer.hr, miljenko@tel.fer.hr, mario.zagar@fer.hr

Abstract: **Interrupt coalescing is a feature implemented in hardware on many of today's high-performance network interface cards (NIC). It allows a reception of a group of network frames to be notified to the operating system kernel via a single hardware interrupt, thus reducing the interrupt processing overhead, particularly at high packet rates. However, the delays introduced by interrupt coalescing can result in significant TCP throughput degradation. In this article we analyze the advantages and drawbacks of a generic interrupt coalescing implementation. We propose an approximate model defining relation between interrupt coalescing delays and steady state TCP throughput, which we validate by laboratory measurements. We conclude our report by showing how the proposed model can be used for determination of optimal delay duration, depending on specific environment and application.**

1. INTRODUCTION

The kernels in general-purpose operating systems (OS), such as various UNIX variants and clones, Microsoft platforms etc., traditionally use hardware interrupts for event notification purposes in communication with input/output (I/O) hardware components. Although most of the data transfers to or from the I/O devices are usually accomplished using the direct memory access (DMA) techniques, which require very little CPU attention, handling of hardware interrupts can impose a significant CPU overhead. This overhead can consume particularly significant amount of CPU resources in case of network interface cards (NIC-s) operating under high traffic loads. NICs normally use interrupt requests to notify the OS on arrival of network frames, and in some NIC driver implementations even on each flush of transmit DMA chain. To lessen the burden of interrupt processing CPU overhead, in recent years the vendors have introduced new types of network interface cards, which implement an option of delaying the generation of hardware interrupt for the predefined amount of time. During the delay window, a group of network frames can be received, whose arrival will be signaled to the OS by generation of only one interrupt request.

However, it is well known that the TCP throughput can be significantly influenced by the overall end-to-end delays between the communicating nodes. Therefore, a too long interrupt coalescing delay window can in many cases lead to unwanted TCP throughput degradation.

Following the analysis of interrupt processing overhead, we present a simple model which identifies the relation between the interrupt coalescing delay window length and a steady state TCP throughput. We validate it by comparing to results of TCP throughput measurements on a referent PC platform, in various scenarios.

2. INTERRUPT PROCESSING OVERHEAD COMPONENTS

In typical IA-32/PCI based systems, the vast majority of interrupts generated by NIC hardware are triggered by the reception of network frames. Most of the general-purpose OS utilize similar schemes for handling the hardware interrupts. However, in this article we focus on 4.4BSD implementation, as a referent example.

On 4.4BSD operating system architecture, handling of each hardware interrupt requires invocation of the following procedures [4]:

A. Hardware and software *context switching*, preservation of CPU registers, and change of active processor stack;
B. Accessing the registers of hardware interrupt controller (typically i8259 on IA-32 systems), and determination of the appropriate device driver interrupt service routine(s) (ISR);
C. Updating the interrupt counters;
D. Processing the interrupt requests inside all of designated ISRs

The processing duration for procedure D is usually variable, as it depends on the complexity of various operations performed on received network frames (IP routing, packet filtering, IPSEC encryption/decryption, termination of local TCP sessions, etc.). However, execution time of procedures A to C can last approximately 1 to 20 μs for each interrupt request [5] [2], depending on CPU, memory and system bus performance and throughput. The processing time for one invocation of specified procedures is relatively constant for specific system hardware, and mostly unrelated to the network traffic or current system load.

Under low to moderate traffic loads (approximately 10.000 packets per second on the current hardware), the overhead introduced by procedures A to C does not significantly influence the overall OS performance. However, at higher traffic loads, this overhead can become the dominant consumer of available CPU time, and in extreme situation lead to *receive-side livelock* [1], a situation where the system spends the whole CPU time solely in processing interrupt requests, and therefore cannot perform any other tasks.

There are several methods to address the interrupt processing overhead-related problems. The following two approaches are most common:

1. Complete abandonment of asynchronous event notification concept based on hardware interrupts and management of all communication with NIC adapters via periodical *device status polling* [5] [2].
2. Delayed generation of interrupts, in order to allow multiple frames to be processed in a single ISR routine invocation. This method is known as interrupt *mitigation* or *coalescing* [10].

The main advantage of both methods can be found in reducing the number of invocations of “overhead” procedures A to C for a given number of received packets per second, which enables significant savings of CPU cycles under high traffic loads.

However, both methods introduce additional variable delay in processing of received network frames. This delay can make significant impact on throughput of transport protocols based on sliding-window mechanisms, such as TCP. In order to avoid such possible throughput degradations, it is critical to choose the optimal parameters in implementation of interrupt coalescing methods.

3. MEASUREMENT OF INTERRUPT PROCESSING OVERHEAD

To determine the actual cost of interrupt processing, we have completed a simple experiment. The experiment involved modifying the ISR routine in the NIC device driver, in a way that it returns control to the OS interrupt dispatcher, without acknowledging the interrupt on the NIC hardware. This resulted in repeated invocation of the same ISR, as soon as it would exit. In order to prevent a complete deadlock, after a predefined number of invocations our modified driver would eventually acknowledge the hardware interrupt, and exit. We have thereby created a kind of interrupt-multiplier, where we could control the number of ISR invocations by modifying the actual frequency of external interrupt-triggering events (incoming network frames) and the deadlock counter (multiplier).

All measurements were performed on the same referent platform: IA-32/PCI system, AMD Athlon uniprocessor @ 1200 MHz, 100 MHz FSB, 256 MB SDRAM, VIA KM133 integrated chipset, two Intel EtherExpress 10/100 PCI network interface cards (i82558), FreeBSD 4.4 operating system.

The measurements were performed in four slightly different scenarios. In one case the device driver returned the control to the interrupt dispatcher immediately, while in the other version we included one programmed input/output (PIO) PCI read from i82558 control/status register (CSR) [6]. In one testbed-setup the NIC was configured to use IRQ 5 line, while in the other case it was using IRQ 9, which is somewhat slower because of the i8259 interrupt controller cascading.

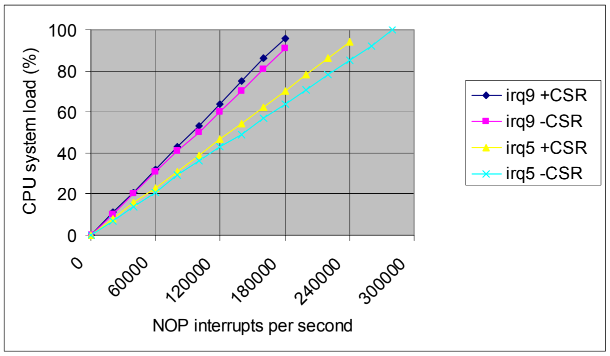


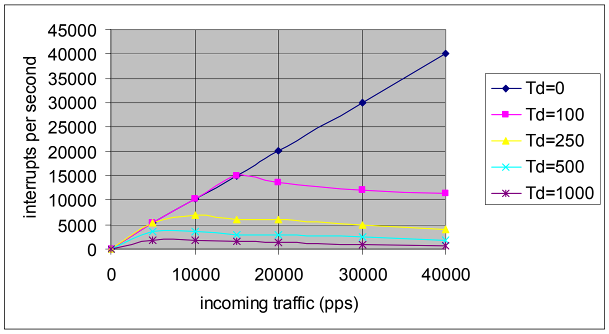
Figure 1 – Interrupt processing overhead – CPU load

From the measurement results shown in figure 1 it can be observed that the system starts experiencing livelock at approximately 180.000 interrupts per second, depending on the actual configuration. Note that in our testbed setup no actual work was done within the ISR, and there where no significant data transfers through the PCI bus. Therefore, in reality it can be expected that the system will experience livelock at even lower interrupt rates.

4. CPU OVERHEAD REDUCTION BY INTERRUPT COALESCING

Interrupt coalescing support is usually implemented in NICs either in the form specialized hardware timecounters, or as firmware delay loops. In our tests we used the modified FreeBSD “fxp” driver [7] for i82558 based NICs, which allowed us to control the delay-loop duration T_d between the arrival of the first network frame, and notification to the OS via hardware interrupt.

We measured the number of generated interrupts and the CPU load as a function of incoming traffic in packets per seconds, for various values of T_d parameter. The incoming traffic (ICMP echo requests) was silently discarded by IP access list, in order to prevent IP/ICMP tasks to significantly influence the measured CPU load. The results, shown in figure 2, outline almost proportional decrease in number of generated interrupts and CPU load, as the duration of interrupt coalescing delay T_d grows.



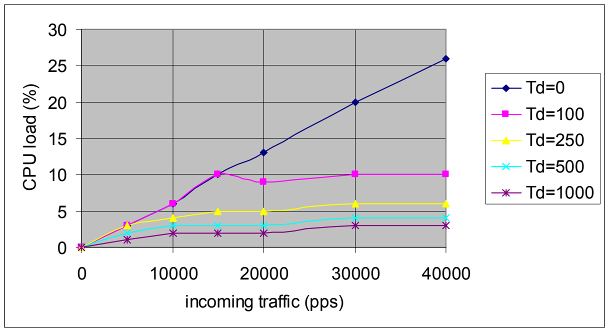


Figure 2 – Number of generated interrupts (top) and CPU load (bottom). T_d values are in microseconds.

5. RELATION BETWEEN INTERRUPT COALESCING DELAY AND STEADY STATE TCP THROUGHPUT

We will analyze the impact of interrupt coalescing delays on steady state TCP throughput using a simple system model with one input and one output FIFO-type queue, as shown in Fig. 3. Resembling the characteristics of typical environments where interrupt coalescing techniques are applied, such as in switched Ethernet LANs, we assume that there are no packet losses on the entire data path between communicating end nodes, and that the maximum unfragmented TCP segment size supported by the media will be used (1460 bytes).

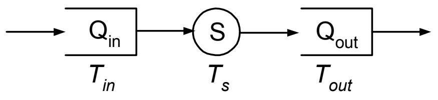


Figure 3 – A simple queuing system model

With T_{in} and T_{out} we will reference the average waiting times in input and output queues, and with T_s the average service time for a single network packet.

For stationary state TCP throughput analysis we can assume that the interarrival time between two incoming packets is approximately constant. Following that assumption, we can estimate T_{in} as:

$$T_{in} = \frac{T_d}{2} \tag{1}$$

Furthermore, if we assume the processing of received packets to be simple and fast, we can consider the duration of this process to be negelectable:

$$T_s \approx 0 \tag{2}$$

The number of network packets received from the input queue in a bundle (triggered by a single interrupt) can be calculated as:

$$N_{bundl} = B(T_d)T_d \tag{3}$$

where $B(T_d)$ denotes the average incoming traffic rate in packets per second.

The average waiting time in the output queue will be proportional to the number of packets bundled and initiated together to the same queue:

$$T_{out} = \frac{1}{2}\frac{N_{bundl}}{B_{\max}} \tag{4}$$

where B_{max} denotes the maximum packet rate supported by the medium, in packets per second (for 100-Mbit Ethernet and MTU=1500 it will be $B_{max} \approx 7500$). By substituting (4) in (5) we get:

$$T_{out} = \frac{B(T_d)T_d}{2B_{\max}} \tag{5}$$

Cumulative average delay of our system is:

$$T_{tot} = T_{in} + T_s + T_{out} \tag{6}$$

which after substitution of (1), (2) and (5) equals to:

$$T_{tot}(T_d) = \frac{T_d}{2}\left(1 + \frac{B(T_d)}{B_{\max}}\right) \tag{7}$$

For identification of stationary state TCP throughput as a function of additional delay, we will use the model [3] as a starting point:

$$B = \min\left(\frac{W_{\max}}{RTT}, \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_0 \min\left(1, 3\sqrt{\frac{3bp}{8}}\right) p(1+32p^2)}\right) \tag{8}$$

where B denotes the traffic rate in packets per seconds; W_{max} maximum size of TCP window (in number of packets); RTT total end-to-end *round trip time* for MSS sized packets;

b the number of packets being cumulatively acknowledged [9]; T_0 the initial retransmission timeout; *p* the packet loss probability.

In case of negelectable packet loss (*p*=0), which we can assume in case of non-saturated switched LAN environments, the equation (8) can be reduced to:

$$B = \frac{W_{\max}}{RTT} \tag{9}$$

The model [3] was originally conceived for analysis of TCP throughput behavior in long-haul WAN networks, in which the propagation delay is dominant, with *RTT* values in range of 100 ms and more. For applications in environments with minimal values of propagation delay, such as in LANs, the reduced form of this model (9) has to be adjusted for minimal values of *RTT*. We can accomplish this using the following (coarse) approximation:

$$B = \min\left(\frac{W_{\max}}{RTT}, B_{\max}\right) \tag{10}$$

Depending on total number *(N)* of interrupt coalescing NICs on the entire duplex end-to-end data path, the overal average value of *RTT* will be increased. Let's assume the contribution of each NIC in *RTT* increase to be linear:

$$RTT = RTT_{\min} + \sum_{i=1}^{N} T_{tot_i}(T_d) \tag{11}$$

If all NICs on the data path are configured with the same T_d parameter, the equation (11) becomes (12):

$$RTT = RTT_{\min} + N \cdot T_{tot}(T_d) \tag{12}$$

It is important to note that by equation (12) we ignore the asymmetric nature of typical TCP flow, as we calculate the same delay contribution to *RTT* in both forward and ACK directions, which differ significantly in packet sizes. It would be more accurate to use different equations for each packet flow direction, but at this point we will again sacrifice accuracy in order to preserve the simplicity of the final form of our model.

By substituting (12) in (9) we get:

$$B(T_d) = \frac{W_{\max}}{RTT_{\min} + N \cdot T_{tot}(T_d)} \tag{13}$$

and by further substituting (7) in (13):

$$B(T_d) = \frac{W_{\max}}{RTT_{\min} + \frac{NT_d}{2}\left(1 + \frac{B(T_d)}{B_{\max}}\right)} \tag{14}$$

The expansion of (14) results in square equation of *B(T_d)*:

$$\frac{1}{B_{\max}} B(T_d)^2 + \left(1 + \frac{2RTT_{\min}}{NT_d}\right) B(T_d) - \frac{2W_{\max}}{NT_d} = 0 \tag{15}$$

The dismissal of negative results of (15) leaves us with the remaining positive result, the equation (16). By applying the special case approximation (10) on (16), the final TCP throughput model *B(T_d)* appears (17).

As the equation (17) shows *B(T_d)* throughput in packets per second, we have to multiply (17) by the value of TCP maximum segment size (MSS), in order to calculate the throughput *R(T_d)* (18) in bytes per second. The later form is usualy more desireable and comprihensable in interpretation and comparation with real-world measurement results.

Let us consider how our model behaves in 100 Mbit Ethernet environment. The value of TCP *MSS* is 1460 bytes. The highest MSS-sized packet rate B_{max} is approximately 7500 packets per second. In our laboratory measurements we will use TCP implementations with the default maximum window size *WSS*=16384 bytes (FreeBSD 4.4, MS Windows 2000, AIX 4.3.3).

To determine the value of W_{max} we use the equation (19). Following the substitution of variables *MSS* and *WSS* with appropriate values for Ethernet and our observed TCP stack implementations, the equation results in W_{max} of 11 packets.

$$B(T_d) = B_{\max}\left(\frac{1}{2NT_d}\sqrt{(NT_d + 2RTT_{\min})^2 + \frac{8W_{\max}NT_d}{B_{\max}}} - \frac{RTT_{\min}}{NT_d} - \frac{1}{2}\right) \tag{16}$$

$$B(T_d) = B_{\max}\min\left(1, \frac{1}{2NT_d}\sqrt{(NT_d + 2RTT_{\min})^2 + \frac{8W_{\max}NT_d}{B_{\max}}} - \frac{RTT_{\min}}{NT_d} - \frac{1}{2}\right) \tag{17}$$

$$R(T_d) = MSS \cdot B_{\max}\min\left(1, \frac{1}{2NT_d}\sqrt{(NT_d + 2RTT_{\min})^2 + \frac{8W_{\max}NT_d}{B_{\max}}} - \frac{RTT_{\min}}{NT_d} - \frac{1}{2}\right) \tag{18}$$

$$W_{\max} = \text{int}\left(\frac{WSS}{MSS}\right) \tag{19}$$

Figure 4. shows the theoretical value of TCP throughput $R(T_d)$ as a function of interrupt coalescing delay, for different numbers N of interrupt coalescing NICs on the data path, and different minimum values of RTT.

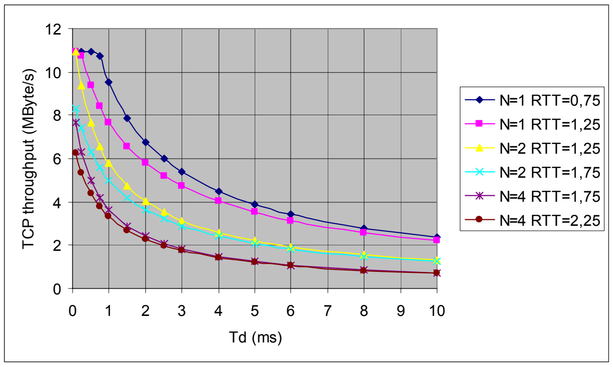


Figure 4 – $R(T_d)$ for different values of N and RTT_{min} (RTT in milliseconds)

6. VALIDATION OF THE PROPOSED MODEL

In order to verify the accuracy of our model, we have performed a series of TCP throughput measurements on various laboratory configurations.

We have divided our measurements in two categories. In our first setup, we have measured ftp transfer rate between our referent machine acting as a server (FreeBSD 4.4, with interrupt coalescing enabled NIC) and two different ftp clients (MS Windows 2000, AIX 4.3.3). Using NIC card setup with T_d=0, we measured the minimum value of RTT, which we substituted in our model equation (18).

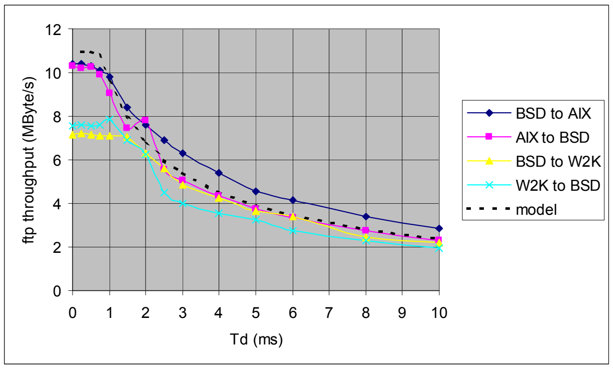


Figure 5 – $R(T_d)$ for N=2 , RTT_{min}=1,137 ms

We have measured the ftp transfer rate in accordance with the described configuration twice. Figure 5. shows the results obtained when all end stations were connected to the same LAN switch.

During the second series of measurements, the nodes were connected to different LAN switches, which were interconnected via a 155 Mbit ATM/LANE backbone. The second scenario introduced somewhat higher value of minimum RTT. The impact of the higher RTT on ftp transfer rate can be seen in the figure 6.

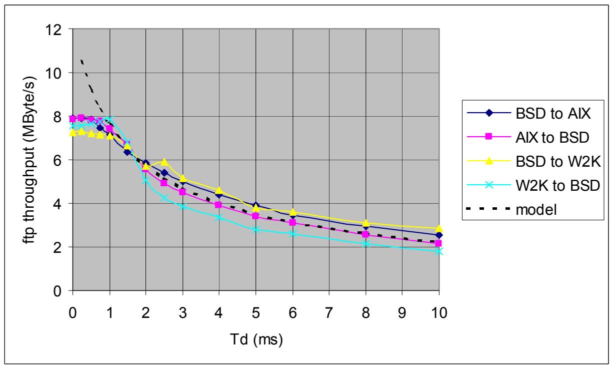


Figure 6 – $R(T_d)$ for N=1 , RTT_{min}=0,739 ms

In further experiments we configured our referent machine as a router, with two interrupt coalescing enabled NICs (N=2). The edge nodes did not use interrupt coalescing. Again, we performed the measurements in two scenarios. In one setup the BSD router and one end station were connected to the single LAN switch, while the other end-station was connected to the BSD router via a crossover cable. Figure 7. outlines the results of this test.

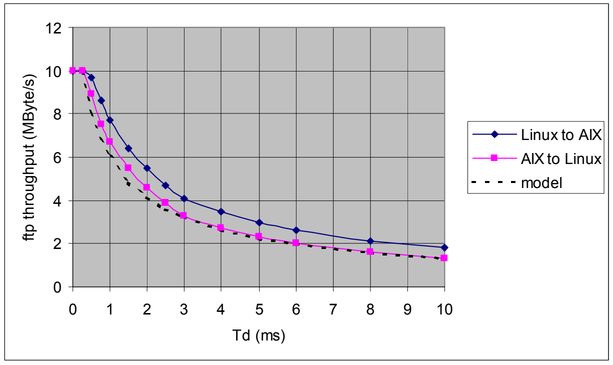


Figure 7 – $R(T_d)$ for N=1 , RTT_{min}=0,739 ms

The other testbed configuration included a second LAN switch and an ATM/LANE backbone, through which all the communications had to traverse. Again, this introduced additional delay, which made further impact on TCP throughput, as shown in figure 8.

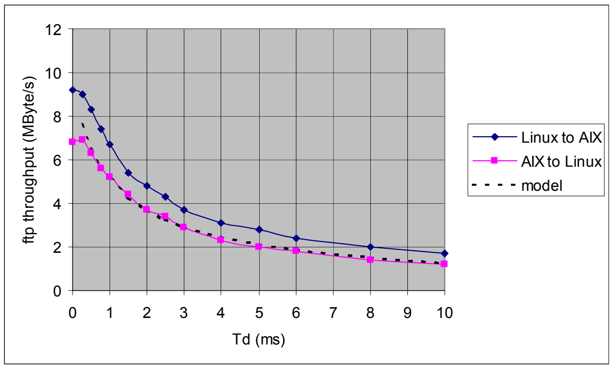


Figure 8 – $R(T_d)$ for N=2 , RTT_{min}=1,696 ms

7. CONCLUSION

By reviewing both measurement results and theoretical figures presented in this article, it is obvious that great care has to be taken in selecting the appropriate delay parameter in generic implementation of interrupt coalescing. The presented model can be used as a helpful tool for choosing the optimal delay values depending on media bandwidth, overall round-trip-times, and number of nodes in the data path that implement interrupt coalescing discipline.

Furthermore, from our measurements we have learned that interrupt coalescing techniques are somewhat better suited for implementation on end-nodes, rather than on packet-forwarding devices. We have seen that cascading of interrupt-coalescing nodes in the data path drastically decreases TCP throughput, because of high aggregate delays. Device polling [2][5], an alternative method for avoiding receive-side livelocks, does not introduce such excessive delays under normal operating conditions, and can therefore be better suited for routing applications. However, either of these two methods become almost mandatory for responding to today's high packet-rate traffic patterns, especially those associated with network-based denial-of-service (DoS) attacks.

In this article we have analyzed only the generic implementation of interrupt coalescing. The vendors have introduced different variations of non-linear extensions in processing of received network frames, such as coalescing queue depth limiting, selective processing of small (TCP ACK) frames, etc. These methods can help retaining the TCP throughput, however, our model is not suitable for analysis and estimation of their behavior. The future work could include the improvement of the proposed model to better reflect the asymmetrical nature of typical TCP flows, as well as the mentioned non-linear processing options.

The model should also be validated in environments other than 100-megabit Ethernet. It would be particularly interesting to investigate the accuracy of the proposed model in the emerging Gigabit-Ethernet LAN environments, as in such systems the interrupt coalescing methods can introduce both the most noticeable CPU overhead savings, as well as the potentially most drastic TCP throughput degradations.

REFERENCES

[1] Jeffrey C. Mogul, K. K. Ramakrishnan: "Eliminating receive livelock in an interrupt-driven kernel", ACM Transactions on Computer Systems, 1997.

[2] Eddie Kohler: "The Click Modular Router", PhD thesis, Massachusetts Institute of Technology, 2001.

[3] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, James F. Kurose: "Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation", IEEE/ACM Transactions on Networking, 2000.

[4] M. McKusick, K. Bostic, M. Karels, J. Quaterman: "The design and implementation of the 4.4BSD operating system", Addison-Wesley, 1996.

[5] L. Rizzo: "Device Polling support for FreeBSD", http://info.iet.unipi.it/~luigi/polling/, online document, Feb 2002.

[6] "82558 Fast Ethernet PCI Bus Controller with Integrated PHY – Datasheet", Intel Corporation, 1998.

[7] M. Zec: "FreeBSD fxp driver patch for bundling receive interrupts", http://www.tel.fer.hr/zec/BSD/fxp/, online document, Oct 2001.

[8] W. Stevens: "TCP/IP Illustrated, Vol 1: The Protocols", Addison-Wesley, 1994.

[9] V. Jacobson, R. Braden, D. Borman: "TCP Extensions for High Performance", RFC 1323, 1992.

[10] Peter Druschel, Larry L. Peterson, and Bruce S. Davie: "Experiences with a high-speed network adapter: A software perspective", Proceedings of the ACM SIGCOMM Conference, 1994.