How modern NICs speed up Packet-Processing Performance of PC-Systems

Rainer Schönberger
Betreuer: Florian Wohlfart, Daniel G. Raumer
Seminar Future Internet SS2013
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: rainer.schoenberger@mytum.de

ABSTRACT
To fulfill the increasing demand of higher network speeds, ethernet standards and corresponding network adapters are commercially available nowadays, which enable transfer rates of up to 10Gb/s. However the technology above the ethernet layer, especially the IP and TCP protocols, were not changed and optimized that rapidly, hence processing packets in these upper layers represent a bottleneck. Also popular additional high level features like IPSec require even more work to be done for packet processing. In this paper, a set of advanced methods is described, which are currently already implemented in the intel 82599 NIC controller, to speed up packet processing by optimizing and maxing out the potential of network interface cards. This includes taking over work of higher protocol levels and thus reducing the mentioned bottleneck of network speed development.

Keywords
NIC, packet processing, intel 82599, niantic, IPsec, offloading, RSS, multiple queues, interrupt moderation, MSI-X, TCP Segmentation, RSC, PTP

1. INTRODUCTION
1.1 Bottlenecks of high speed networks
The current ethernet standard introduced in IEEE 802.3 specifies a low level communication system, describing an implementation for the ISO OSI layers 1 and 2. In this standard, technology to enable data transfer rates as fast as 10Gb/s (for example 10GBASE-SR) are introduced [1]. In most current applications of this standard however, the maximum transfer rate can not be reached.
To understand why, one has to consider the main delaying factors for packet transmission [4]. The first one is the currently high *per-packet-cost*, which is dominated by large data and protocol processing overheads resulting from upper layer protocols. With this overhead, computer systems currently are almost unable to handle and process incoming packets efficiently at rates as fast as one packet per 67ns [10, 4]. One of the most commonly used upper layer protocol stacks, operating on top of ethernet, is TCP/IP. Processing packets in the TCP/IP stack at these high rates is an enormous challenging task and already requires a considerable amount of cpu utilization, even at lower speeds [11].
Another cost factor is *per-byte-cost*. This describes the delay and cpu work caused by copying and transferring data, calculating checksums or encryption. To be able to send or receive packets in the first place, they have to be transferred from and to the CPU. Current transfer methods like *Direct Memory Access*, which hand data over to the CPU via shared main memory regions, are already pushed to their limits.
Never the less numerous methods exist, to solve the problems, stated above, to a certain extent. In the following paper some techniques are presented, which are implemented in modern network interface cards (NICs), to speed up packet processing and thus enable higher transfer rates while reducing CPU load.

The remainder of this document is organized as follows. As an additional introduction the role of NICs in the ISO OSI model is analyzed. In the beginning of section 2 the operation principle of modern NICs is presented at the example of the intel 82599 controller. After that, special features implemented in this NIC to speed up packet processing are described in detail. Section 3 gives a summary of the described features and concludes by analyzing the future development of NICs.

1.2 Role of NICs in the ISO OSI model
As shown in figure 1, the ethernet standard specifies layer 1 (physical layer) and 2 (data link layer). A network interface card operates exactly on these two layers, providing a connection to a transmission medium over physical layer dependant hardware (PHY) containing a method to do line coding (PCS) and send signals to the medium (PMA). Also access control to the medium (MAC) as well as a method to send and receive ethernet packets to and from other participants in a network, using an addressing system (LLC) is provided [1, 2].
Traditionally the upper layers are implemented as part of the operating system, also called host system or host in this paper. With modern NICs however, more and more work from layer 3 and 4 (TCP/IP) moves from the operating system to the NIC hardware as described in this paper. Thus the narrow and exact defined scope of NIC duties becomes blurry.

2. THE INTEL 82599 NIC CONTROLLER
2.1 Overview
The Intel 82599, also nicknamed *ninantic*, is a 10Gb/s ethernet controller IC. It supports two separate interfaces, for driving *Media Access Units* (MAU) to transfer data over optical fiber or copper wires. With this, it enables support for various ethernet standards, including 100BASE-TX or

doi: 10.2313/NET-2013-08-1_01

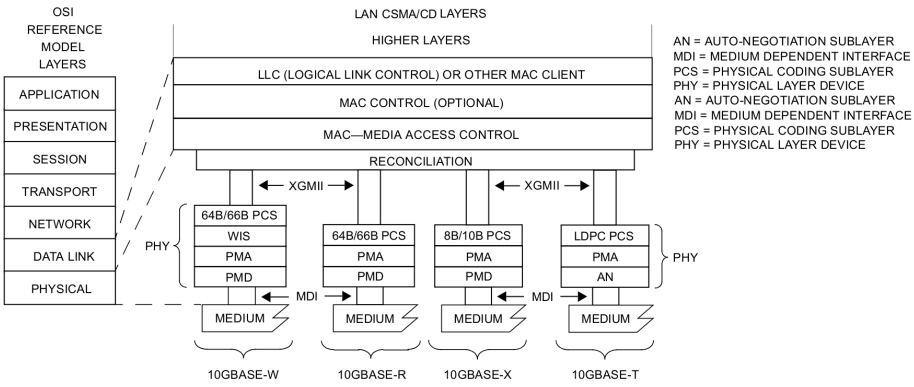

Figure 1: Ethernet - ISO OSI Layers. [1]

10GBASE-SR.
The controller is designed for high end server NICs such as the Intel *Converged Network Adapter X520* [3] and provides a number of features to speed up packet processing. This is why in the following, this controller is considered as a reference for giving an overview of the current state of the art NIC optimization techniques.

2.2 Path of a packet through NIC to OS

2.2.1 General concepts

As shown in figure 2, the intel 82599 ethernet controller is connected to the CPU via a PCI-Express bus. To transfer packets, Direct Memory Access (DMA) is used to read/write data from the individual Rx-/Tx-Queues (see chapter 2.3.5) directly to main memory with minimum influence on CPU work.
In the operating system, packets are stored in memory buffers represented by *descriptors* often called *mbufs*. These descriptors contain additional information for packets, used as a method by the protocol stack to pass meta data to other layers and pointers to the memory buffers containing the packets.
The NIC uses much simpler descriptors that the OS. These are organized in *NIC rings* which are circular arrays of NIC descriptors to represent the memory buffers and meta data. Figure 3 shows these different descriptors and how packets between NIC and the operating ststem are exchanged via shared buffers in main memory [4]. Each Rx-/Tx-Queue can be configured to point to a specific ring of memory buffers. So data to be sent/received can be written or read from a ring, served by a single queue.
While the packet is being processed in the NIC, various filtering and CPU offloading features are applied on the fly (see chapter 2.3).

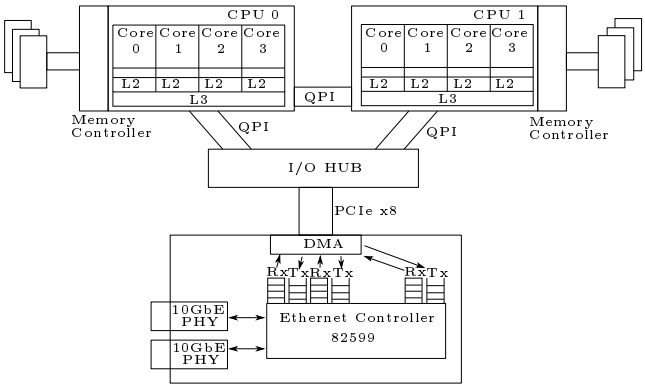

Figure 2: Intel 82599 connectivity overview. [5]

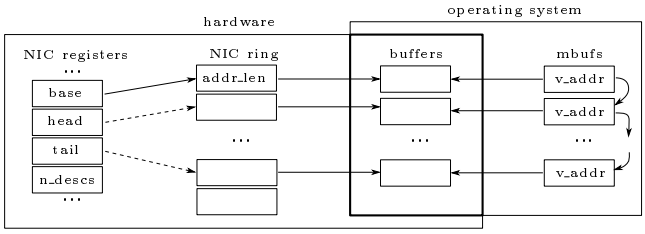

Figure 3: Packet exchange between NIC and OS via buffers in main memory. [4]

2.2.2 Sending

The sending process of a packet [2] can be split into 4 basic steps, whereas step 2 and 3 intermix to a certain extent, because data is already processed in on the fly while still being received by DMA.

1. Host side preparation:
   First of all the host creates a ring of NIC descriptors and assigns one of the NIC's Tx-Queues to the ring. When the TCP/IP stack requests the transmission of a packet, the descriptors are initialized to point to the corresponding data buffers of the packet. After that, the host updates the Queue Tail Pointer (TDT) of the assigned Tx-Queue.

2. Transfer to NIC:
   The DMA controller of the NIC is configured to sense a change of the TDT. Once this happens, the DMA controller sends a PCIe request to fetch the according descriptors. As soon as the first descriptor is received, it is placed in the appropriate location of the descriptor queue and the NIC sends PCIe requests to get the packet data from system memory.

3. Processing in the NIC:
   When the packet data is being received by the DMA

doi: 10.2313/NET-2013-08-1_01

all programmed data manipulations (see chapter 2.3) are applied on the fly and the packet is stored in the transmit FIFO.
Packets in this FIFO are eventually forwarded to the MAC Part of the NIC, where the L2 checksum is applied and the packet is finally sent over the wire.

4. Cleaning up:
After all packet data has been transfered to the NIC, the appropriate descriptors are updated and written back to host memory. An interrupt is generated to tell the host, that the packet has been transmitted to the NIC.

2.2.3 Receiving

Like in the previous chapter, one can also spilt up the receiving process of an ethernet packet [2] into 4 simple steps:

1. Host side preparation:
To be able to receive packets from the NIC the host first has to configure and associate a new ring of descriptors with one of the NIC's Rx-Queues. These descriptors are initialized to point to empty data buffers. After that, the Queue Tail Pointer (RDT) is updated by the host, to let the NIC know, that it is able to receive data.

2. Processing in the NIC:
When a packet enters the Rx MAC part of the NIC, it is forwarded to the Rx filter. According to this filter, the packet is placed into one of the 128 Rx FIFOs or dropped (see chap. 2.3.5).

3. Transfer to the host:
The DMA controller fetches the next appropriate descriptor from the according host memory ring (specified by the Rx FIFO). As soon as the whole packet is placed in the Rx FIFO, it is written to the memory buffers, referenced by the descriptors, via the DMA controller.

4. Handing over control to the host:
When the packet is transmitted to host memory, the NIC updates all used descriptors and writes them back to the host. After that, an interrupt is generated to tell the host that a received packet is ready in host memory. The host then hands over the packet to the TCP/IP stack and releases associated buffers and descriptors.

2.3 Special Features

As previously mentioned, the intel 82599 controller has a variety of advanced functions to speed up packet processing by either unburden the cpu from tasks, which easily can be achieved in hardware (*offloading*) or providing advanced interfaces and filtering options to enable better scaling in multicore and multiprocessor systems. In the following some of these techniques are described in detail.

2.3.1 Checksum offloading

For faster packet processing, the intel 82599 controller supports calculation of layer 3 (IPv4 only) and layer 4 (TCP or UDP) checksums in hardware for both receive and transmit functionality.
In the transmission process checksum offloading can be enabled for specific packets by configuring the corresponding Tx descriptor (see chap. 2.2.2). Thereby the packet type, size information of headers and payload must be provided. The hardware then adds the calculated checksums to the correct locations.
On the receive side, checksum calculation is also possible in hardware. But before this is done the packet is passed through a variety of filters, to check if checksum calculation makes sense. These include MAC destination address verification, IP header validation and TCP/UDP header validation. After a packet passes all the filters, the checksum is calculated and compared to the included checksum in the packet. A checksum error is then communicated to the host by setting the according *Checksum Error* bit in the *ERROR* field of the receive descriptor [2].
Checksum offloading tremendously reduces cost for preparing packets to be sent in software.

2.3.2 IPsec offloading

Ipsec describes a set of protocols, which enable authentication and encryption on the IP layer. There are two main protocols specified:

- *Authentication Header* (AH) is a method which provides authentication of IP packets. This is done by appending a special header to the packet, containing a cryptographic hash from the whole IP packet (including parts of the header).
- *Encapsulating Security Payload* (ESP) enables encryption of the IP payload and optionally authentication via an appended authentication hash block after the encrypted payload.

A specific communication secured with IPSec (called *IPSec flow*) shares encryption parameters, like an algorithm type and secret keys (security association). These parameters have to be exchanged before the secured communication takes place.
The intel 82599 chip supports offloading encryption and authentication work for up to 1024 established[1] IPSec flows to hardware. Therefore the controller implements the *AES-128-GMAC* algorithm for authentication (AH or ESP without encryption) and the *AES-128-GCM* algorithm for encryption plus authentication (ESP) [2]. Offloading IPSec work for flows with other encryption types is not possible with the intel 82599.

Figure 4 shows a simplified example of three established IPSec flows, whereas two of them are offloaded to the NIC hardware and one still beeing handled in software.
On the transmission path the hardware only encrypts the packet, creates a authentication hash and places the calculated data in the appropriate location within the IPSec packet. Hence a complete IP packet with a valid IPSec header must be provided by software with the unencrypted data as a payload. The NIC is then ordered via the packet

[1] Security association exchange has to be done by software and the corresponding parameters and keys have to be provided to the NIC. Also the software has to specify which flows should be offloaded.

doi: 10.2313/NET-2013-08-1_01

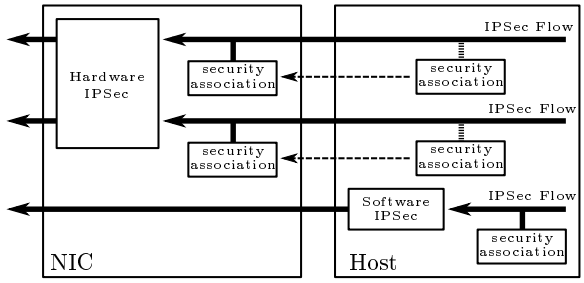

Figure 4: IPSec offloading principle of operation.

descriptor to do the IPSec offloading. On the receiving side the 82599 decrypts the payload and checks the packet for authenticity, again maintaining the IPSec packet structure. The host is then notified about a authentication success or failure via the packet descriptors [2].

2.3.3 TCP segmentation offloading (TSO)

Big TCP packets exceeding the size of the maximum transmission unit (MTU) have to be split into multiple packets. Normally this is done by the TCP/IP Stack on the host side, but with modern NICs, the host is able to offload this task to the networking hardware. With that, TCP packets, bigger than the maximum MTU, can be handed over to the NIC. The host only has to calculate the number of packets, the data has to be split in and provide header information to the device driver.
After the NIC has split the TCP payload of the packet into multiple parts, the packet header is parsed and adjusted for the individual packets. Thereby only the MAC header can be copied without changing. The IP and TCP headers have to be adjusted for each packet. Now the corresponding Ethernet CRC, IP and TCP checksums have to be calculated in hardware (see chap. 2.3.1). The new packets are then formed with updated headers and checksums and sent over the wire [2, 7].
Table 1 shows a typical TCP packet sent to the NIC, as well as multiple TCP segments leaving the NIC. Thereby an individual header and field checksum (FCS) was calculated in hardware for each packet using information from the so called *Pseudo Header* provided from the TCP/IP Stack.
This feature has a number of positive effects on performance:

- CPU workload is reduced, because the host does not need to do segmentation and only needs to calculate one header instead of several headers for all the segmented packets
- Overhead due to data transmission between TCP/IP Stack and device driver or device driver and NIC is reduced, because larger blocks of data are transmitted instead of many small pieces.
- Only one interrupt per transmitted TCP packet is generated instead of one per segment.

2.3.4 Receive side coalescing (RSC)

RSC is motivated by the same reasons as *TCP segmentation*, described in the previous chapter, but operates on the receiving side of the NIC. The goal is to identify TCP packets

Table 1: TCP segmentation [2]

Packet sent from host:

Pseudo Header			Data
Ethernet	IP	TCP	DATA (TCP payload)

Packets sent from NIC:

Head	Data first MSS	FCS	...	Head	Data next MSS	FCS	...

from the same connection (flow) and coalesce them into one large packet with a single header, which then is transmitted to the host [11].
The intel 82599 has buffers to coalesce packets from several TCP connections at the same time. Each of these buffers is associated with a *RSC context*, which stores exact information and a hash value for identifying the connection and header information, as well as offset values for new data to be placed into the buffer. To identify packets from a

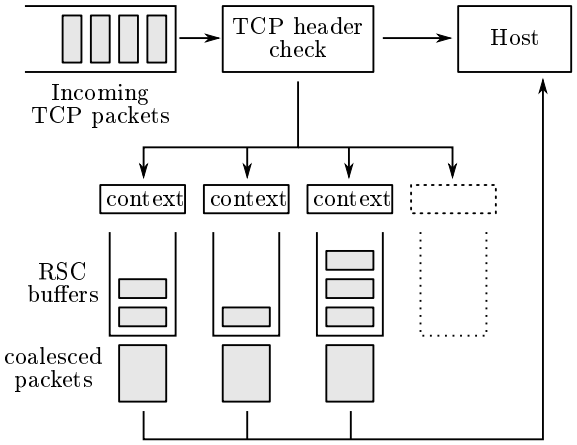

Figure 5: RSC principle of operation.

specific connection, the source plus destination IP addresses and source plus destination TCP port numbers are extracted from the TCP header and a hash value of this data is calculated. This hash is then compared with the hash in each RSC context. If no match is found, a new buffer and context is created. If a match was found, the connection identifying data is compared again with the data in the context, this time for an exact match. After passing this test, the packet data is coalesced to the buffer and the corresponding context is updated with new offsets [2, 11].
To finish a coalescing process and forward the packet to the host, besides other reasons, one of the following conditions have to be met [2]:

- The corresponding RSC buffer is full.
- A packet from an existing connection (RSC context match) with the wrong sequence number arrives, meaning packet loss has occurred, or the order of packets was not maintained during transmission.
- A interrupt assertion timeout occurs. This is, when collected events should generate an interrupt using the

doi: 10.2313/NET-2013-08-1_01

ITR interrupt moderation technique as described in section 2.3.6

- An LLI-flagged packet arrives (also see chap. 2.3.6)

Figure 5 shows a simplified functional diagram of the RSC process. In this example three RSC contexts already exist and a forth context might be opened when a packet arrives which does not match any of the existing contexts.

2.3.5 Multiple Rx-/Tx-Queues

The support of multiple receive and transmit queues is a key feature of 82599 based NICs. A variety of secondary features is achieved by the use of these queues. Two of them are mentioned here:

Programmable L3/4 filtering. For faster processing of packets on the host, the intel 82599 controller supports 128 programmable filters, operating on Layer 3 and 4, which are applied on incoming packets to determine the destination Rx queue.
Each filter is described by a so called 5-tuple consisting of

- IP-Protocol type (TCP/UDP/SCTP/other)
- IP Source address, IP destination address
- Source port, destination port.

To specify which parts of the 5-tuple are required to match, a bitmask has to be supplied for each filter. After a packet matches one or more filters the matching filter with the highest priority is selected and the packet is forwarded to the Rx queue specified by the filter [2].
This feature is especially useful for software routers or firewalls, as it offloads work for basic packet filtering into the NIC hardware, reducing both CPU utilization and routing latency.

Receive Side Scaling (RSS). This is a technique to enable scaling with increasing number of CPUs. Thereby the received packets are distributed to several queues, which then are bound to specific CPUs or cores [2, 6]. But with this some problems occur. For example when ethernet packets from the same TCP connection are distributed to several CPUs they have to be merged and ordered again by the TCP stack. So the distribution of these packets causes overhead.
To solve problems of this kind, a hash function is utilized to determine the destination Rx queue (see figure 6). As seen in table 2, the hash function therefore uses different parts of the packet as input data depending on the packet type, to ensure a meaningful distribution to CPUs. This allows packets from one TCP or UDP connection depending on their parameters in the header to be handled in the same queue to enable independent processing of the queues by multiple processors or cores. As a hashing algorithm, the intel 82599 uses the *NdisHashFunctionToeplitz* function (for details see [2] p. 326).

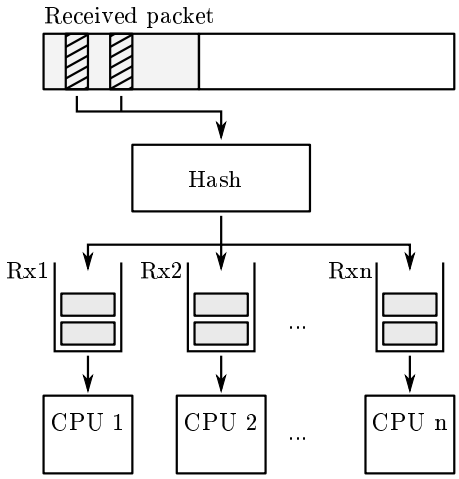

Figure 6: RSS principle of operation.

Table 2: RSS hashing input data [2, 6]

Packet type	Components used for hash
IPv4/v6 + TCP	SourceAddress + DestinationAddress + SourcePort + DestinationPort
IPv4/v6 + UDP	SourceAddress + DestinationAddress + SourcePort + DestinationPort
IPv4/v6 + unknown	SourceAddress + DestinationAddress

2.3.6 Interrupt management

Interrupts from PCI and PCIe devices are not generated by separate interrupt wires, but by so called *Message-Signaled Interrupts* (MSI or the extended specification MSI-X) [8, 9]. Thereby an interrupt is triggered by writing to a special memory region. With the MSI-X standard up to 2048 different interrupt messages are supported [8], whereas the intel 82599 only utilizes 64 different MSI-X interrupts [2].
When a relevant event occurs within the NIC (Rx/Tx descriptor writeback for a specific Queue, Rx Queue full, ...) the *Extended Interrupt Cause Register* (EICR) is updated according to the event, whereas the queues can be assigned to bits in the EICR register according to figure 7. A interrupt is then generated and the cause dependant MSI-X message is sent. In legacy MSI mode, only one type of interrupt message is sent and the host has to determine the interrupt cause in software by reading the EICR register [2]. Hence MSI-X interrupt mode improves interrupt latency.

With a high rate of packet transmission, the interrupt rate also increases proportionally. In case of a network adapter operating at speeds up to 10Gbit/s, this might use a big amount of CPU time to handle the interrupts. The intel 82599 NIC controller features two *Interrupt Moderation* techniques, to handle high interrupt rates:

Time-based Interrupt Throttling (ITR). Usually it is not important for the operating system to be notified as soon as every single packet arrives. Hence this feature allows the

doi: 10.2313/NET-2013-08-1_01

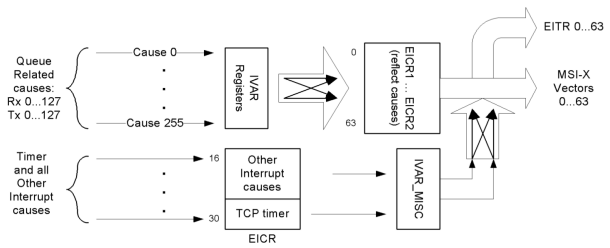

Figure 7: Interrupt causes and their mapping through EICR to MSI-X messages. [2]

user to limit the maximum rate of interrupts. Therefore an *ITR Counter* is introduced, which decrements the 9 Bit ITR value every $2\mu s$ in 1Gb/s or 10Gb/s operating mode or every $20\mu s$ in 100Mb/s mode. Once the timer reaches zero, accumulated events trigger an interrupt and the ITR value is reloaded with a user specified value. If events happen while the timer is not zero, the EICR register is updated and no interrupt is sent [2].

Low Latency Interrupts (LLI). On the one side *Time-based Interrupt Throttling* reduces CPU load, but on the other side interrupt latency is increased according to the configured maximum interrupt rate. So when configuring ITR, a compromise between these two parameters has to be made. But this states a problem, as some important events have to generate interrupts with as low latency as possible (for example, if the receive descriptor ring is running dry, or timing critical packets are handled). To solve this, the intel NIC controller supports bypassing of the ITR throttling feature for specific events, allowing immediate interrupt initiation, called *Low Latency Interrupt*. Additionally to queue assignment, the 5 tuple filters described in chapter 2.3.5 can also be configured to generate an LLI interrupt on a match.

2.3.7 Direct Cache Access (DCA)

DCA makes it possible for PCIe devices to write data directly to the cache of a specific CPU. Thus reducing memory bandwidth requirement, as the CPU does not often need to read data from memory, because less cache misses occur [10]. A DCA flagged PCIe Packet is therefore sent to the I/O Controller and the data is directly written to the CPU Cache via the *Quick Path Interconnect* (QPI) or a similar system (see fig. 2). This feature is especially useful, as memory latency alone can significantly slow down packet transmission [10].

2.3.8 Precision Time Protocol (PTP)

PTP is specified in the IEEE1588 standard. It presents a technique, which enables to accurately synchronize clocks over a local area network. It is similar to NTP, which is used in larger networks. Thus PTP provides a much better precision up to the sub microseconds range. To enable time synchronization with a large pool of clocks connected in one network, the PTP standard specifies an algorithm to autonomously build up a *master-slave hierarchy*, in which the clients find the best candidates for a clock source [12, 2].

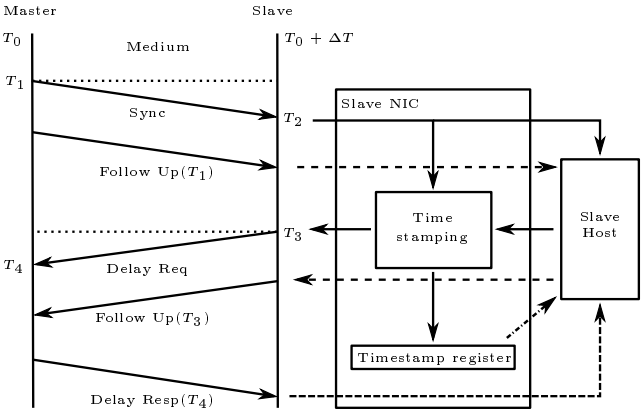

Figure 8: PTP Sync. Principle of operation. [2]

The clock synchronization itself is shown on the left side of figure 8. The master periodically sends a Sync packet and captures the time (T_1) when the Sync packet leaves the NIC. If the client receives a Sync packet, it also captures the current time (T_2) according to its own shifted clock and memorizes the value. After the Sync packet has been sent by the master, a Follow Up packet including the captured time stamp T_1 of the Sync packet is sent[2]. The client then sends a Delay Req packet to the host, which is again timestamped by the client (T_3) and the server (T_4), as mentioned before. In a Delay Resp message the server finally sends the time stamp T_4 to the client [12, 2]. Now enough data has been collected to calculate the clock difference:

$$(T_0 + \Delta T + T_2) - (T_0 + T_1) = (T_0 + T_4) - (T_0 + \Delta T + T_3)$$

$$\Rightarrow \Delta T = \frac{(T_2 - T_1) - (T_4 - T_3)}{2}$$

In version 1 of the PTP standard the packets sent for synchronization are UDP Layer 4 packets. With version 2 of the standard however, the protocol is also specified for pure ethernet frames identified by a special EtherType value in the MAC header.
The intel 82599 supports hardware time stamping of incoming and outgoing Sync and Delay_Req packets. This is supported for ethernet PTP messages as well as for UDP packed PTP messages. Thereby the time stamp is captured by the NIC exactly, before the last bit of the ethernet *Start of Frame Delimiter* is sent over the wire. The chip only takes the time stamp of the transmitted or received packet and provides the value in NIC registers. The rest of the PTP implementation (including building and sending the Follow_Up messages) has to be done in software by reading the time stamp registers.
Figure 8 shows the role of the NIC during a PTP clock synchronization and which PTP packets trigger the hardware time stamping logic.
Because the time stamping logic is located as near as possible to the physical medium interface, very high accuracy is achieved with hardware PTP support in the intel 82599 [2].

[2]The time stamp can not be included in the Sync packet, because at the time, the Sync packet is built the time stamp does not yet exist, as it is captured when the packet is already being sent.

doi: 10.2313/NET-2013-08-1_01

3. CONCLUSION

In the previous sections an overview of current techniques were presented, to speed up packet processing in high speed networks. Such as moving work from the TCP/IP Stack to the hardware (offloading features), enabling better utilization of multi processor systems and optimizing data transfer and communication between NIC and the host system. To enable networking on even higher speeds than 10Gb/s, these efforts might not be enough and hence the NICs are still a subject for active research. Ideas for integrating parts of NICs into CPUs to further improve communication and data exchange with the CPU exist [13] and it is a visible trend that more and more work will be done by NIC hardware in the future to conquer latency and high CPU load.

4. REFERENCES

[1] *IEEE Std 802.3 - 2008 Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications - Section Four*, In IEEE Std 802.3 - 2008 (Revision of IEEE Std 802.3 - 2005), IEEE New York, NY, USA, 2008

[2] *Intel 82599 10 GbE Controller Datasheet Rev. 2.76*, Intel, Santa Clara, USA, 2012

[3] *Product brief - Intel Ethernet Converged Network Adapter X520*, Intel, Santa Clara, USA 2012

[4] L. Rizzo: *Revisiting Network I/O APIs: The netmap Framework*, In Queue - Networks Volume 10 Issue 1, ACM New York, NY, USA 2012

[5] *Evaluating the Suitability of Server Network Cards for Software Routers*, Maziar Manesh, ACM PRESTO Philadelphia, USA 2010

[6] *Receive Side Scaling*, Microsoft msdn online documentation, `http://msdn.microsoft.com/en-us/library`, 2013

[7] *Offloading the Segmentation of Large TCP Packets*, Microsoft msdn online documentation, `http://msdn.microsoft.com/en-us/library`, 2013

[8] *Introduction to Message-Signaled Interrupts*, Microsoft msdn online documentation, `http://msdn.microsoft.com/en-us/library`, 2013

[9] G. Tatti: *MSI and MSI-X Implementation*, Sun Microsystems, In PCI-SIG Developers Conference, 2006

[10] R. Huggahalli, R. Iyer, S. Tetrick: *Direct Cache Access for High Bandwidth Network I/O*, In Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on, IEEE, 2005

[11] S. Makineni, R. Iyer: *Receive Side Coalescing for Accelerating TCP/IP Processing*, In HiPC'06 Proceedings of the 13th international conference on High Performance Computing, Springer-Verlag, Berlin, Heidelberg, 2006

[12] J. Eidson: *IEEE-1588 Standard Version 2 - A Tutorial*, Agilent Technologies, Inc, 2006

[13] N. Binkert, A. Saidi, S. Reinhardt: *Integrated Network Interfaces for High-Bandwidth TCP/IP*, ACM, San Jose, California, USA, 2006

doi: 10.2313/NET-2013-08-1_01