Implementation of TCP Large Receive Offload on Open Hardware Platform

Gianni Antichi
Dept. of Information Engineering
University of Pisa, ITALY
g.antichi@iet.unipi.it

Christian Callegari
Dept. of Information Engineering
University of Pisa, ITALY
c.callegari@iet.unipi.it

Stefano Giordano
Dept. of Information Engineering
University of Pisa, ITALY
s.giordano@iet.unipi.it

ABSTRACT

Nowadays, the bottleneck in network communications is not represented by the link capacity anymore, but by the receiver processing power. To face this problem, more and more offloading techniques have been developed and implemented in modern NICs, allowing the CPUs to offload some of the required processing onto the underlying hardware. In this work, we present an implementation on an open hardware platform of a stateless Large Receive Offload method (LRO). The presented results experimental demonstrate the effectiveness of the proposed implementation.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Network Protocols

; C.2.m [**Computer Communication Networks**]: Miscellaneous

Keywords

NetFPGA, TCP, LRO, High Performance

1. INTRODUCTION

In the last few years Internet has experienced an explosive growth, mainly due to the introduction of novel network paradigms and services. This growth has been made possible by a strong increase of the network performance, especially in terms of available bandwidth. As a result, the bottleneck in network communications is not represented by the link capacity anymore, but by the receiver processing power. Indeed, receiving huge quantities of network packets requires big amount of CPU processing power, that is not always available at the receiver host.

To face this problem, more and more techniques have been proposed to improve the network performance, proposing efficient modifications of the TCP protocol, so as to reduce the TCP overhead [2][3][6][9]. In the meanwhile, several offloading techniques have been developed and implemented in modern NICs, allowing the CPUs to offload some of the required processing onto the underlying hardware. This is justified by the fact that high-speed, simple tasks are better performed in hardware, where memory access is quick and resources are promptly available. As a result, offloading techniques have proved to significantly reduce CPU utilisation, leading to improved throughputs and better system performances.

But which features are eligible for an useful and efficient offloading? A lot of different tasks can be run by hardware only. Today, the whole TCP/IP stack can be completely managed by the NICs: this is performed by the so-called TOEs (Tcp Offload Engines) [11]. These stateful method, though, are known to cause security and support issues due to the need of breaking the kernel stack.

In this work, we present an implementation on an open hardware platform of a stateless Large Receive Offload method (LRO). In a nutshell, LRO consists in aggregating consecutive chunks of data on the receiving side, reducing the number of packets reaching the TCP/IP stack and, thus, the amount of work required to the CPU. This basic behavior is shown in Fig. 1, where the payloads of several consecutive packets are aggregated (in this work, on the NetFPGA level) to create a "new" single packet, to be forwarded to the OS TCP/IP stack. Although, its benefits in terms of sheer performance gain are minor when compared to a TOE, its implementation is completely transparent to the TCP/IP stack, requiring no changes to the OS kernel.

This work discusses the implementation of a basic LRO module on the NetFPGA 1G platform and describes the experimental tests carried out to demonstrate the effectiveness of the proposal, in terms of performance improvements.

The remainder of this paper is organised as follows: next section discusses some related works, while section 3 provides some background information on the LRO mechanism. Then section 4 briefly describes on the used hardware platform, while section 5 details the proposed module and section 6 discusses the experimental results. Finally, section 7 concludes the paper with some final remarks.

2. RELATED WORK

Over the years several solutions have been proposed to improve the performance of TCP/IP receiver processing (*i.e*, interrupt mitigation, jumbo frames, receive side scaling, remote direct memory access, direct cache access, TCP offload engine, large receive offload). Nowadays NIC drivers support such features in order to offload as much as possible

HPPN'13, June 18, 2013, New York, NY, USA.

the CPU, hence enabling host PC to deal with the constant growth of the traffic to process.

S. Makineni et al. [8] introduced the Receive side scaling idea for the first time. As of April 2007, the Linux kernel supports LRO for TCP in software only. Following such a lesson learned, L.Grossman [4] implemented this feature on a 10G ethernet driver, while T.Hatori et al. [5] studied the performances of a LRO engine implemented in a Xen virtualized environment.

Unfortunately, all of these solutions are studied and implemented from a software point of view only.

We believe that LRO mechanisms should be also approached from an hardware perspective. Indeed, being stateless and transparent offload mechanisms, they result to be well suitable for hardware implementation.

To the best of our knowledge there is no open solution that performs LRO directly on the hardware. For this reason, in this paper, we present an implementation of a simple LRO module on an open hardware platform (namely a NetFPGA platform).

3. LARGE RECEIVE OFFLOAD

In most of network packet processing applications, the number of packets to be processed roughly determines the performance. Given the streaming nature of the TCP protocol, in principle, it is possible to aggregate consecutive packets into bigger ones, directly when the TCP packets are received at the lowest level, without intervening on the applications.

LRO is a technique for increasing inbound throughput of high-bandwidth network connections by reducing CPU overhead. Taking advantage of such TCP property, LRO works by aggregating multiple incoming packets from a single stream into a larger buffer before they are passed higher up the networking stack, thus reducing the number of packets that have to be processed.

LRO technique can be adopted at the lowest possible level (*i.e.*, hardware) without disrupting the OS TCP state machine behaviour. Such a compatibility is crucial in order to reduce as much as possible compatibility and security issues and making at the same time this feature integrable with all existing OSs. LRO usually comes with the TCP checksum offload feature which saves the host CPU from having to compute the checksum. The savings strongly depend on the packet size. While small packets have little or no savings with this option, large packets have larger savings (*i.e.*, from 5 to 15 percent [7]). Hence, the LRO feature should be used with Hardware Checksum offloading option enabled in order to save as much as possible CPU cycles.

When a new eligible packet (*i.e.*, we will define what packets are eligible for aggregation afterwards) arrives, the NIC must calculate IP/TCP checksums in order to ensure the integrity of the data that will be sent to the CPU-Host. While corrupted packets are discarded, eligible packets are aggregated and then sent to the OS stack. As stated above, the LRO-enabled NIC must ensure full compatibility with the OS kernel TCP/IP stack (*i.e.*, congestion control mechanism). For this reason, packets matching any of the following conditions (i.e., non-eligible packets) are passed untouched to the OS stack:

- Non-TCP packet
- Out-of-order packet

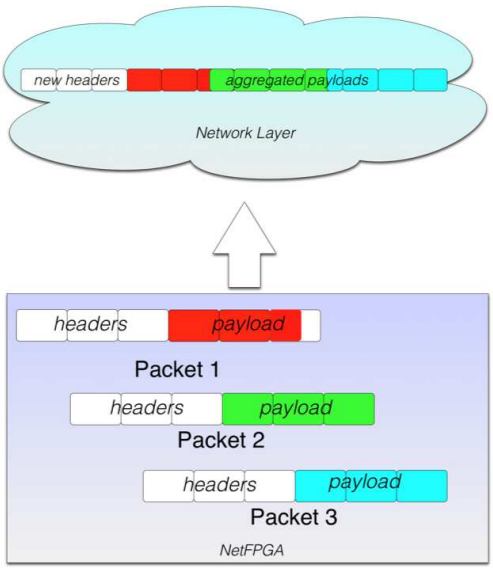

Figure 1: LRO

- IP options are present
- TCP packet is ack-only (payload is zero)
- SYN or FIN flags are set
- URG or PUSH flags are set
- TCP options other than timestamp are set

Once enough eligible packets have been collected, the resulting packet should be sent to the OS stack as a standard TCP one with the payload, containing the aggregation of the "original" packets payloads. To make it possible, the LRO-enabled NIC must update the IP/TCP length fields, use the TCP sequence number and timestamp of the first aggregated packet and use the ACK value of the last aggregated one. We point out that the checksum fields (*i.e.*, IP and TCP) should not be updated because the OS stack is informed that it has not to perform the integrity tests (*i.e.*, checksum offload feature enabled). Using the TCP timestamp of the first packet may affect the precise estimation of RTT values. However, in practice, since only packets which arrive very close in time to each other are aggregated, the timestamp values on all the TCP packets are expected to be almost the same, without any significant loss of precision.

4. NETFPGA

NetFPGA [10] is a low-cost platform, developed by the High Performance Networking Group at Stanford University, primarily designed as a tool for teaching networking hardware and router design. It is a standard PCI card that plugs into a standard PC. The card contains a Field Programmable Gate Array (FPGA) by Xilinx (Virtex-II pro) which is programmed with user-defined logic and has a clock of 125 MHz. The PCI interface connecting the host PC to the NetFPGA is managed by a small Xilinx Spartan II FPGA. Four 1GigE ports, 4.5MB of Static Ram (2 banks) and 64MB of DDR2 Dynamic RAM are also on board in the

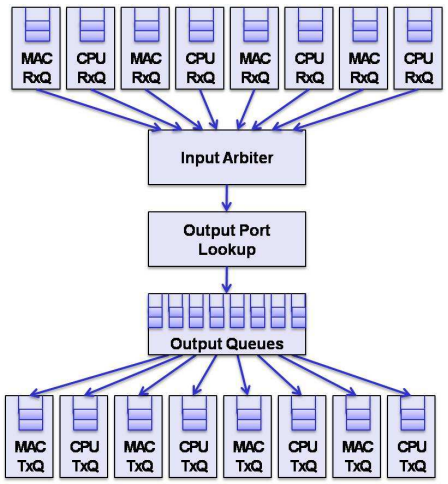

Figure 2: NetFPGA reference pipeline

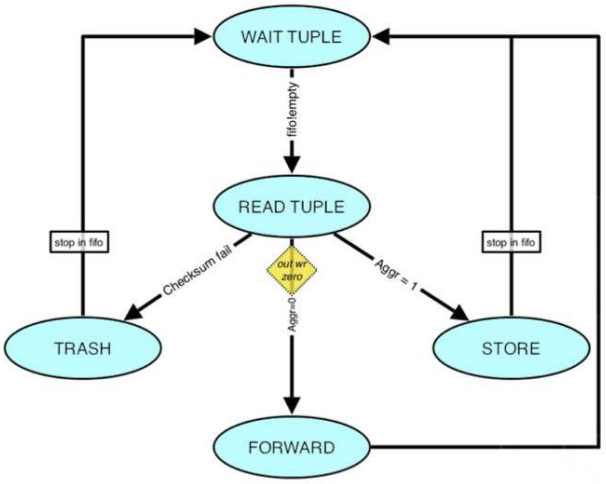

Figure 3: LRO Module Functioning

card. A reference package containing verilog source code for the FPGA, C code for the host PC and java code for the graphical interface can be downloaded from the NetFPGA website in order to run NetFPGA with basic networking functions such as Network Interface Card (NIC), PW-OSPF IPv4 Router and Layer 2 switch. The basic target for this board is the adoption of an FPGA as a networking accelerator in order to take advantage of the host PC flexibility to implement the control plane of the project. Thanks to its modularity, NetFPGA is a very useful system to test new ideas for next generation networks.

Fig. 2 represents the reference pipeline of the NetFPGA board. It has eight receive queues, eight transmit queues and a user data path in which custom modules are inserted. Each "MAC" (Media Access Controller) is a physical network port with an associate queue, and each MAC queue has an associate CPU queue used for communication between the NetFPGA and the host PC. The input arbiter services the set of eight queues in a round robin fashion.

5. LRO MODULE ARCHITECTURE

In this section we discuss the architecture of the module we have implemented, detailing the operations performed by the different blocks of the module.

Note that, for sake of simplicity, in the following we refer to the formatted units of data as packets, even if the term frame would be more correct, since they refer to the physical layer and not to the network layer.

The basic idea behind the LRO module is very simple: it analyses incoming packets, decides if they have to be aggregated or not, and forwards them to the right data port. This simple idea is sketched in Fig. 3, which shows a very simplified flowchart of the functioning of the module. In a nutshell the module is, at first, in a state in which it waits for a packet. When a new packet arrives, the module reads all the necessary information and on the basis of such information it decides if the packet is eligible for aggregation or not. Then it checks the packet checksums and finally performs one of the following operations: discards the packet (if the checksums have failed), forwards the packet (if the checksums are correct and the packet can not be aggregated), or sends the packet to the store module, where the aggregation is performed (if the checksums are correct and the packet can be aggregated).

Let us analyse, in more detail, how the single blocks of the module work.

First of all, when a new packet is received it is parsed to retrieve all the information needed for deciding if it can be aggregated or not (the packet fields that are parsed at this stage are shown in Fig. 4). Hence, on the basis of the fields that are marked in green in the figure (i.e., eth type of the ethernet header, version, header length, TTL, and protocol of the IP header, and flags of the TCP header), the module, if it is the case, selects the packet for aggregation and store the LRO_STRUCT structure (depicted in Fig. 5) in a buffer. Such a buffer that contains all the data needed to identify the different packets that can be aggregated together (e.g., the flow identifiers) also contains a bit (namely the AGGR bit) that indicates if the packet is eligible for aggregation or not. It is important to highlight that such a structure is also created in the case the packet cannot be aggregated; this is necessary because, before sending the packet to the following block, the module has to verify the checksums (after the checksums verification the module will use the AGGR bit to send the packet to the correct block).

It is important to highlight that the checksums verification is usually performed at the NIC driver level. But, in case LRO is performed at the hardware level (as we have done in this work), the packet that is sent to the card driver is an aggregated packet (and not the original one) and this would result in a checksum verification fail. For such a reason we have modified the NetFPGA driver (that natively would not support such a feature) so as to support checksum offloading. For the sake of completeness, we also have to say that another possible approach would have been to correctly set the checksums fields after the aggregated packet is formed, but this would have caused unnecessary delays and waste of CPU resources on the receiving side.

If the received checksum values are correct, the packet is assumed to have been received uncorrupted, and the CSUM bit in the LRO_STRUCT is set.

At this point, if the checksum verification has failed the packet is dropped, otherwise the packet is sent either to the

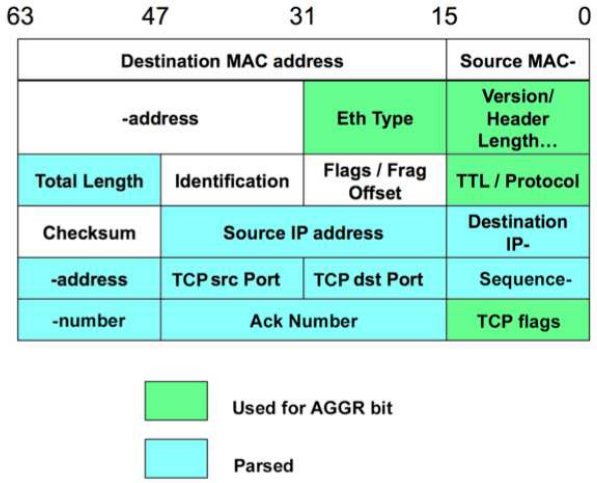

Figure 4: Information used for aggregation decision

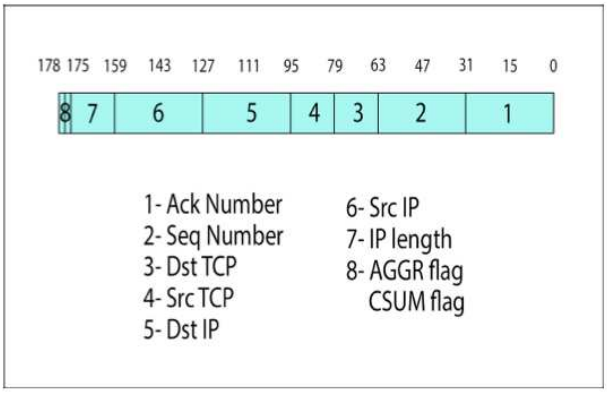

Figure 5: LRO_STRUCT

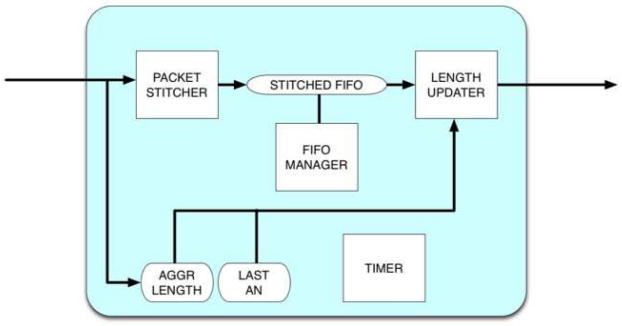

Figure 6: Store Block

forward block (if AGGR bit is zero) or to the store block (if AGGR is set). In the first case, the forward block acts very similarly to the reference NIC block: it simply changes the packet metadata so that it is forwarded to the right port.

Instead, in the second case the packet is sent to the store block that represents the core of the LRO module, where the "real" offloading operations are performed. Indeed, it is where integrable, consecutive packets belonging to the same flow are stored and put together to form a new packet, which payload consists in the aggregation of the payloads of the stored packets (the architecture of this block is shown in Fig. 6).

As it can be seen from the figure this block is composed by several sub-blocks, each one responsible for a different operation, with the blocks at the top of the figure that are responsible for the "real" aggregation of the payloads, while the others are responsible for "side" operations such as update of the new length of the packet and timing of the aggregation. Let us analyse in more detail how the store block works, by referring to the flowchart presented in Fig. 7.

At the beginning, the module is waiting for the first packet (sent by the previous block); when the packet arrives the module invokes the stitcher that, roughly, allocates a buffer in which it starts storing the received payload. When the end of packet (EOP) signal arrives, the module updates the length of the aggregated packet (being the first packet it will assume the same value of the actual packet length). After that it moves to a state in which it waits for a new packet. While being in this state, two different events can occur: either the timer expires or a new packet arrives. In the first case, the module closes the packet (aggregated until that moment) and sends it to the upper layers of the stack. Then, it resets all the variables and buffers and moves to the "wait first packet" state. In the second case, if a new packet arrives, the module invokes the stitcher that stores the packet in a buffer together with the tuple information. After the EOP signal is received the module compares the tuple information of the just received packet with the LRO_STRUCT: if the information match, the new packet belongs to the same flow of the previous one/ones, hence the payloads are aggregated and the length of the aggregated packet is updated. At this point, the length of the aggregated packet is compared with the maximum admissible length: if this limit has been reached the aggregated packet is closed and sent, the variables are reset and the module moves to the "wait first packet" state; otherwise the module goes to the "wait new packet" state. Instead, in case the match between

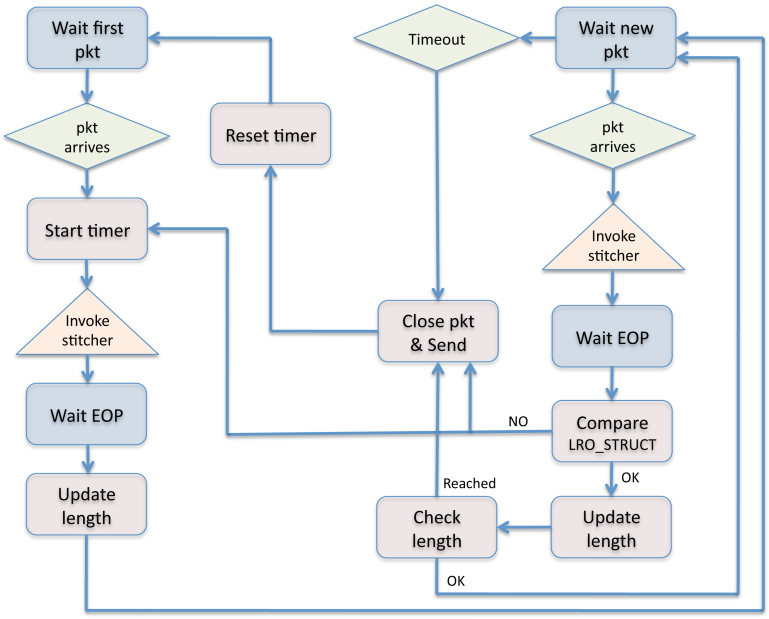

Figure 7: Store Block: Flowchart

the LRO_STRUCT and the tuple information fails, meaning that the new packet belongs to a different flow, the aggregated packet is closed and sent, while the module considers the new packet as the first one of a new flow and starts the aggregation process again (moving to the "start timer" state of the flowchart).

One last consideration has to be done on the need, when aggregating the packets, of aligning the different payloads. This need derives from the fact that the NetFPGA data bus is 64-bit wide. In more detail, when receiving a packet, the NetFPGA ethernet controller may insert some padding to align the payload to 64 bits (see the red packet in Fig. 8). Hence, when a new eligible packet (green packet in Fig. 8) is received, the LRO module has to cope with such a padding, by substituting it with valid data taken from the new packet and, if it is the case, by adding a new padding at the end of the new aggregated payload.

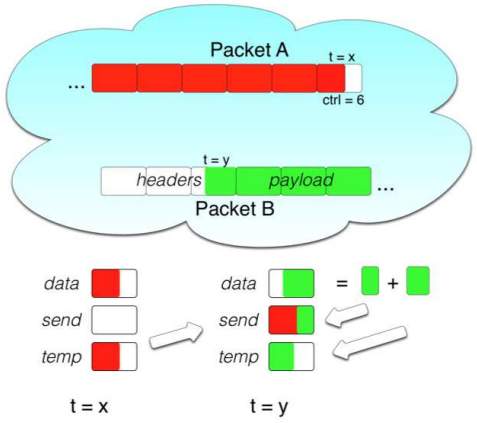

Figure 8: Padding Operations

6. EXPERIMENTAL RESULTS

We evaluated the actual performance of the proposed implementation of the LRO through a variety of experiments. Regarding the traffic generator, we have used the Spirent AX4000 traffic analyzer [1], which is an ASIC-based tool able to reach the full line rate also in the worst case scenario (packet length of 64 bytes). Instead, regarding the PC that hosts the NetFPGA, it is equipped with a AMD phenom X4 9650 quad-core 2.3GHz and 4GB of RAM.

We have considered two distinct scenarios: the first one is simply composed by the aforementioned packet generator directly connected to one of the ethernet ports of the NetFPGA programmed as a common NIC (in this case the

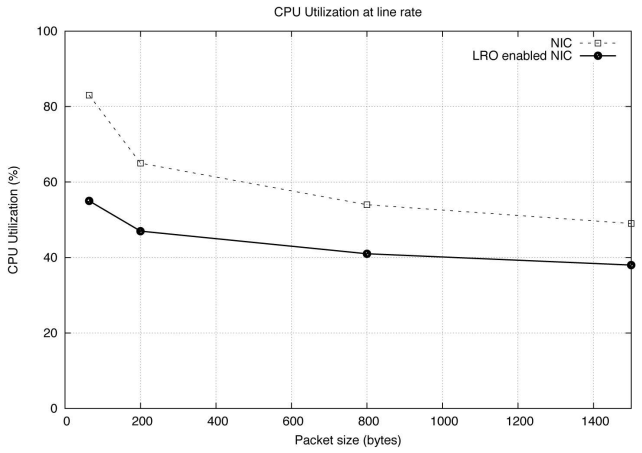

Figure 9: CPU Utilization

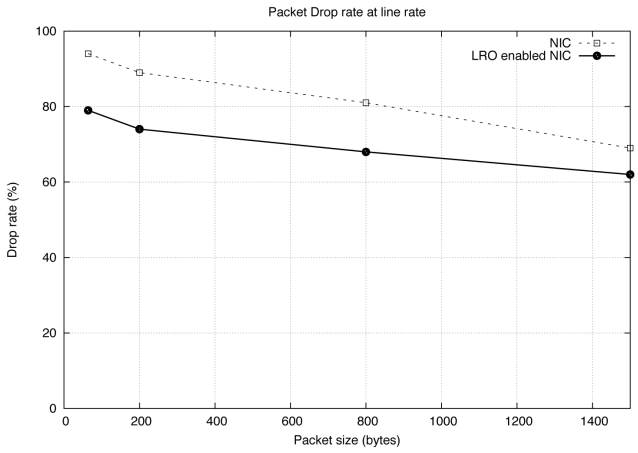

Figure 10: Packet drop rate

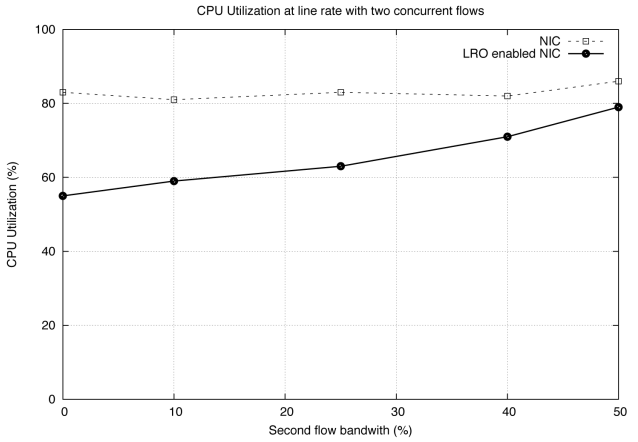

Figure 11: CPU Utilization with 2 flows

LRO support offered by the Linux kernel is used), and is referred to, in the following, as "NIC"; the second scenario, instead, is composed of the same traffic generator connected to the NetFPGA programmed as a LRO-enabled NIC, and is referred to as "LRO-enabled NIC".

As far as the performance metrics are concerned, we have used two distinct parameters: the CPU utilisation and the percentage of dropped packets. It is worth noting that the latter implicitly gives information about the achieved throughput, indeed the throughput obviously increases when the losses decrease. Nonetheless, providing the drop ratio is considered more fair, because the improvements in the achieved throughput also depend on the TCP version and not only on the considered aspects.

The first tests have been carried out to inspect the impact of the packet length on the system performance. Fig. 9 shows the CPU utilisation when receiving line rate traffic with growing packet length from 64 bytes to 1514 bytes. We can easily conclude that, as expected, the LRO benefits are more noticeable when we have small size packets (*i.e.*, saving 28%). While, when the packet length is equivalent to the MTU (Maximum Transfer Unit) we only save the 12 % of the CPU resources. These results are strongly justified by the fact that the number of packets that can be aggregated (that roughly gives an idea of the amount of CPU offload), before reaching the maximum aggregate length and thus being sent to the TCP/IP stack, is inversely related to the length of the packet payload.

Still referring to the same test session, Fig. 10 shows the number of dropped packets at line rate considering the two scenarios and varying the incoming packet length. Again, we can easily notice that improvements offered by our module are more noticeable when the system has to process small packets. Indeed, with a packet length of 64 bytes the LRO enabled NIC allows a reduction in the drop rate of the 15 %, while, with a packet length of 1514 bytes, such a gain decreases to about the 7 %.

These first tests have been performed by generating one TCP flow only and, in general, we can expect different performance when more than one single flow reach the destination host. For such a reason, figure 11 shows the CPU utilisation when the NIC receives two concurrent TCP flows. By analysing the plot we can see that, as expected, when the bandwidth of the two concurrent TCP flows is exactly the same, the LRO benefits are less noticeable since the NIC will aggregate less packets before sending them to the CPU, while when the bandwidth of the second flow decreases towards zero, the system behaviour is the same as in the case of one single flow.

7. CONCLUSIONS AND FUTURE WORKS

In this paper, we have proposed an implementation of a stateless Large Receive Offload (LRO) method on an open hardware platform (namely a NetFPGA). This method consists in aggregating consecutive chunks of data on the receiving side, reducing the number of packets reaching the TCP/IP stack and, thus, the amount of work required to the CPU.

In the experimental tests, we have compared our implementation with the one available in the Linux kernel, showing the effectiveness of the proposed solution. Indeed, the NetFPGA implementation behaves better than the other one in all the considered scenarios. These results are strongly justified by the fact that the Linux implementation (not being a hardware implementation) is not able to completely offload the CPU.

In the near future, we plan to extend the presented work

by performing some more experimental tests (e.g., increasing the number of concurrent flows) and comparing our solution with commercial ones.

Acknowledgemnt

This work was partially supported by GreenNet (FIRB 2008), a research project supported by the Italian Ministry for University and Research (MIUR).

8. REFERENCES

[1] Spirent AX-4000 Homepage. http://www.spirent.com/Solutions-Directory/AX_4000.

[2] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. *Communications Magazine, IEEE*, 27(6):23–29, 1989.

[3] P. Druschel and G. Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In K. Petersen and W. Zwaenepoel, editors, *OSDI*, pages 261–275. ACM, 1996.

[4] L. Grossman. Large receive offload implementation in neterion 10gbe ethernet driver. In *Linux Symposium*, 2005.

[5] T. Hatori and H. Oi. Implementation and analysis of large receive offload in a virtualized system. In *Virtualization Performance: Analysis, Characterization, and Tools , 2008.*, 2008.

[6] J. Kay and J. Pasquale. Profiling and reducing processing overheads in tcp/ip. *IEEE/ACM Transaction on Networking*, 4(6):817–828, 1996.

[7] B. Leitao. Tuning 10gb network cards on linux. In *Linux Symposium*, 2009.

[8] S. Makineni, R. Iyer, D. Sarangam, N. D., L. Zhao, R. Illikkal, and J. Moses. Receive side coalescing for accelerating tcp/ip processing. In *International Conference on High Performance Computing, 2006.*, 2006.

[9] A. Menon and W. Zwaenepoel. Optimizing tcp receive performance. In *USENIX, 2008.*, 2008.

[10] NetFPGA. *http://www.netfpga.org.*

[11] S. Sandhya and R. Hernandez. Introduction to tcp offload engines. *Dell Power Solutions*, 2004.