# Implementation of TCP Large Receive Offload on Multi-core NPU Platform

Li Jie
School of Computer
National University of
Defense Technology
Changsha, China
Email: lijie13d@nudt.edu.cn

Chen Shuhui
School of Computer
National University of
Defense Technology
Changsha, China
Email: shchen@nudt.edu.cn

Su Jinshu
School of Computer
National University of
Defense Technology
Changsha, China
Email: sjs@nudt.edu.cn

***Abstract*—Nowadays, the ethernet is developing much faster than memory and CPU technologies, protocol processing has become the bottleneck of TCP performance on end systems. Modern NICs usually support offload techniques such as checksum offload and TCP Segmentation Offload(TSO), allowing the end system to offload some processing work onto the NIC hardware. In this paper, we propose an implementation of Large Receive Offload(LRO) on a multi-core NPU platform to improve TCP performance, particularly, we employ a so called active ACK mechanism to make very large packets(64KB) aggregation possible. We present experiment results to demonstrate the effectiveness of our proposal.**

***Keywords*—Multi-core NPU, TCP, LRO**

## I. Introduction

In the last few years, ethernet bandwidth has increased from 1Gbps to 100Gbps, while memory bandwidth from DDR2's 8533.33MBps [1] to DDR4's 19200MBps [2], the top speed of processors settled around 4GHz and has not increased much since the year of 2005 [3]. This performance gap makes memory access and protocol processing become the bottleneck of TCP, instead of link capacity. The constantly increasing network bandwidth has caused a severe burden for CPU, optimizing TCP processing mechanism can mitigate this situation and improve TCP performance on end systems.

Traditional TCP acceleration techniques such as checksum optimization [4] [5] [6] [7], zero-copy [8] and interrupt coalescing [9] focused on the host side, protocol processing is still done by host CPU. TOE [10] can offload the entire TCP protocol processing workload and improve TCP performance dramatically, but its implementation is very complex and often causes security and compatibility issues [11]. TSO [12] optimizes TCP's data sending path by offloading the data segmentation and checksum calculation functions, the technique has become rather mature because of its simplicity. LRO [13] improves TCP performance by reducing the number of packet headers processed by CPU, but it works in the NIC driver layer and the packet aggregation job is still done by host CPU.

A multi-core Network Processing Unit(NPU) is an integrated circuit which has a feature set specifically targeted at the networking application domain, it usually has excellent packet processing capability for the following reasons:

1) More than a dozen hardware based, low-switching-overhead threads. The large number of hardware contexts enables software to more effectively leverage the inherent parallelism exhibited by packet processing applications.
2) Favorable I/O features. A multi-core NPU can import packets from network interface to memory with high throughput, moreover, its flexible dispatching component can distribute packets to different threads or cores according to application configurations and pipeline with the corresponding processing threads.
3) Well designed message passing mechanism among different threads. A multi-core NPU often employs crossbar structure or SRAM as its message transfer medium, which makes thread synchronization efficient and elegant.

Different TCP flows are weakly correlated and can be processed concurrently, this fact naturally leads to the idea of employing a multi-core NPU's excellent packet processing capability to accelerate TCP processing on an end system. In this paper, we propose to use multi-core NPU as NIC and implement LRO on it, our implementation reduces the number of packets processed by host network stack and the number of interrupts generated by NIC, eventually improves TCP performance on an end system. The experiment results demonstrate the effectiveness of our proposal. Further more, our implementation only involves the NIC hardware and driver layer, there is no difference between the multi-core NPU and a normal NIC from the kernel network stack's and user applications' angle, our implementation does not suffer TOE's compatibility and security problems.

## II. Related Work

Large Receive Offload is a NIC driver layer technique for increasing TCP data receiving throughput, it works by aggregating multiple small data packets of the same flow into large but much fewer ones, the aggregated packets are then delivered to the kernel network stack for further processing, as shown in Figure 1.

LRO was first proposed by Grossman [13] and implemented in the NIC driver program for Neterion Xframe-II. When

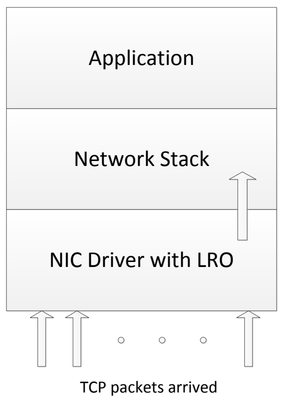

Fig. 1. NIC driver with LRO

a packet arrives, driver program first decides whether to buffer the packet for aggregation, or discard the packet, or pass the packet directly to kernel network stack. Once the total data length of accumulated packets reaches a predefined threshold, driver program performs packets aggregation by setting appropriate values for the new packet's IP and TCP headers.

Themann [14] patched LRO into Linux operating system as a generic implementation, which enabled other developers to equip their NICs with LRO functionality through only a few code modifications.

Hatori [15] et al. implemented LRO in a Xen virtualized system on both physical and virtual interfaces, they achieved less CPU utilization and higher data receiving throughput. While Antichi [16] et al. implemented LRO on NETFPGA open hardware platform, their experiment results showed lower CPU utilization and packets drop rates, but the performance tests are conducted only on two concurrent TCP flows.

## III. System Architecture

In our proposed scheme, we use multi-core NPU as NIC and implement LRO on it to accelerate TCP processing on the data receiving path. Packets reordering, packets aggregation and checksum verification(calculation) are done by the multi-core NPU, NIC driver program is responsible for forwarding the reconstructed big packet to the kernel network stack. Our system resides in the NIC hardware and driver layers, as shown in Figure 2.

The multi-core NPU's numerous threads are divided into three categories: packet sending threads, packet receiving threads and a timeout checking thread(we will explain the reason of its existence in the next section).

The packet sending process is quite similar to a regular NIC, the receiving process of an ethernet packet, however, is much more complex. The multi-core NPU needs to check if a packet is suitable for LRO, handle out-of-order packets, check packet receiving timeout for a TCP flow, and reconstruct accumulated TCP data packets(we will describe these functionalities in more detail later). The packet receiving process sequence is shown in Figure 3: it starts by checking timeout messages,

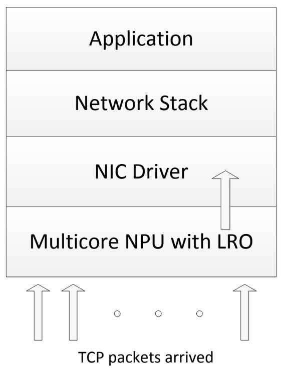

Fig. 2. NIC with LRO

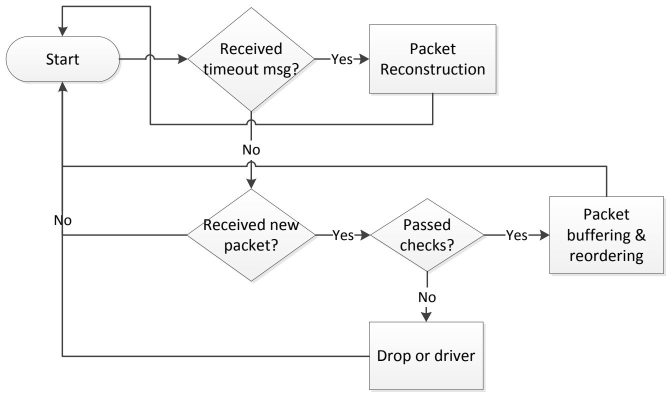

Fig. 3. Packet Receiving Process Sequence

these messages are sent by a particular timeout checking thread, a timeout message indicates which TCP flow has paused receiving new packets, and causes the multi-core NPU to aggregate buffered data packets of this flow. Then, the multi-core NPU checks for a newly arrived packet and runs a series of tests to see if the packet fits LRO requirements. If the packet passed those tests, it is buffered according to the TCP flow it belongs and reordered by its sequence number for further packets reconstruction; if it failed LRO tests, it will be forwarded directly to the NIC driver program(e.g. a UDP packet) or simply discarded(e.g. incorrect checksum). Since the multi-core NPU has done nearly all the dirty work, the NIC driver program's job is rather simple and straightforward, very like normal NIC driver programs except that: the driver program needs to pre-allocate consecutive memory pages for storing aggregated packet's data, and construct a correct skbuff data structure for it.

## IV. System Functionalities

### A. TCP Connection Management

Our system employed two data structures for TCP connection management: ConnectionDescriptor and ConnectionTable.

ConnectionDescriptor is used to represent a TCP connection, it contains information including: the four-tuple which can uniquely identify a TCP connection; the number and total data length of buffered packets; and the maximum consecutive sequence number after packets reordering.

Each packet receiving thread of the multi-core NPU needs to maintain multiple TCP connections, i.e., multiple ConnectionDescriptors, thus a fast search mechanism is necessary. ConnectionTable is introduced to fulfill this requirement, it is designed as a hash table, and used for searching the corresponding ConnectionDescriptor when a TCP packet arrives. Our system utilizes the multi-core NPU's packet dispatch mechanism to distribute packets of the same connection to a fixed packet receiving thread. When a packet receiving thread receives a TCP packet, it calculates a hash value based on the packet's four-tuple, and looks for the corresponding ConnectionDescriptor via ConnectionTable. Each packet receiving thread maintains an independent ConnectionTable, so the search and update operations do not need to interact with other threads, synchronization overheads are naturally avoided.

Our system monitors TCP's three-way handshake sequence, when the multi-core NPU receives a SYN packet, it considers a TCP connection is being established. The corresponding packet receiving thread gets a free ConnectionDescriptor, initializes it with SYN packet's four-tuple information and adds it to the ConnectionTable.

Our system also monitors FIN packets to see if a connection is being tared down. When a packet receiving thread receives a FIN packet, it searches for the related ConnectionDescriptor, aggregates its buffered packets and generates an interrupt to make the driver program process this reconstructed packet. Then the ConnectionDescriptor is removed from ConnectionTable and marked as free for future use.

As with ConnectionTable, each packet receiving thread maintains its ConnectionDescriptors in an independant memory space to avoid thread synchronization operations.

## B. Packets Filtering

Each packet receiving thread of the multi-core NPU runs a series of tests to see if a packet is appropriate for reordering and aggregation operations, packets which failed these tests are discarded or forwarded to the driver program, packets which passed these tests will be buffered on their corresponding ConnectionDescriptors for further processing. The detailed filter sequence is shown in Figure 4, note that we do not require packets' TCP sequence numbers to be consecutive.

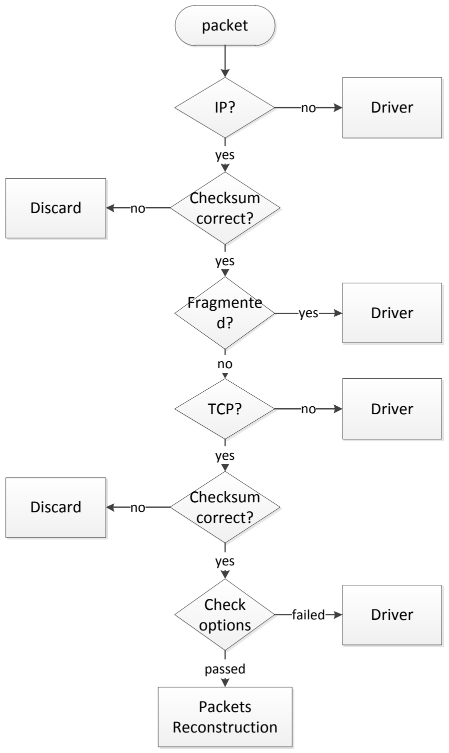

Fig. 4. Packets Filtering

**Algorithm 1** ReorderPacket(pkt)

```
Input: newly arrived TCP data packet
  $desc \leftarrow SearchDescriptor(pkt)$
  if $desc.nextseq == pkt.seq$ then
    $Insert(pkt, desc)$
    $UpdateNextSeq(desc)$
  else
    if $desc.nextseq > pkt.seq$ then
      $Drop(pkt)$
    else
      $Insert(pkt, desc)$
    end if
  end if
```

## C. Packets Reordering

Each ConnectionDescriptor contains a list for buffering TCP data packets, multi-core NPU sorts packets by their TCP sequence numbers and stores them in an increasing order. Pseudo code of our packets reordering algorithm is shown in Algorithm 1. The $Insert$ function adds a packet to its corresponding ConnectionDescriptor's packets list according to the TCP sequence number, or discards the packet if it has been received before, as shown in Algorithm 2. Variable $nextseq$ records the maximum consecutive sequence number of buffered packets and infers the consecutive data length of those packets.

## D. Packets Reconstruction

A packet receiving thread reconstructs buffered TCP data packets if one of the following three conditions is satisfied:

1) consecutive payload length of the buffered packets has reached a predefined threshold
2) a FIN packet is received
3) a certain connection timed out for receiving packets

The detailed processing sequence is shown in Figure 5. Packets reconstruction is done in two steps: (1) multi-core NPU modifies IP and TCP headers of the first packet, DMA the first packet's entire content(including headers) and the rest

**Algorithm 2** Insert(pkt, desc)

```
Input: TCP data packet and a connection descriptor
  for all $p$ in $desc.pktlist$ do
    if $p.seq == pkt.seq$ then
      $return$
    else
      if $p.seq > pkt.seq$ then
        $insert\ pkt\ in\ front\ of\ p$
        $return$
      end if
    end if
  end for
```

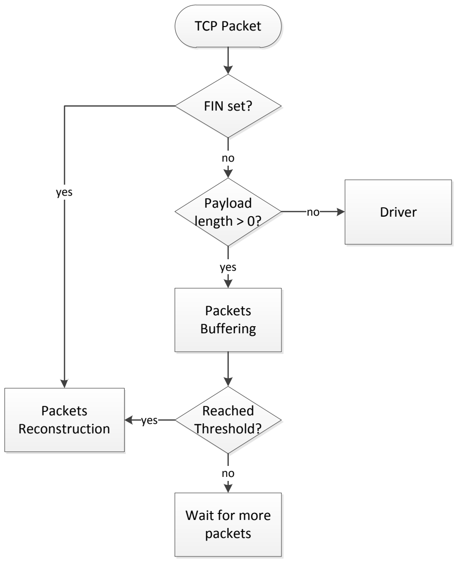

Fig. 5. TCP Packets Processing

packets' payload data to host memory; (2) driver program constructs correct skbuff data structure for the aggregated packet and forward it to kernel network stack.

## *E. Timeout Check for Packet Receiving*

A TCP connection can remain valid without any packet interactions for hours, the following situation explains the necessity of timeout check for packet receiving: multi-core NPU sets the threshold of buffered packets' data length to be 64KB, while the sending side only transmitted 8KB data and paused. Since the threshold is not reached and no FIN packet is received, the received data packets will be buffed in the multi-core NPU's memory permanently and will never reach to the data receiving host.

To solve this problem, each ConnectionDescriptor maintains two more variables for packet receiving timeout check: (1) $average$, which represents the average time interval of two adjacent packets; (2) $last$, which records the last time ConnectionDescriptor received a packet. When a packet arrives, the packet receiving thread reads system time $current$ and updates $average$ by the following formula:

$$average = (average + current - last)/2 \quad (1)$$

$last$ is updated by $current$.

We used a specific thread of the multi-core NPU to execute the timeout checking task for every ConnectionDescriptor in our system, and named this thread as timeout checking thread. The timeout checking thread works in an infinite loop, it reads system's $current$, a ConnectionDescriptor's $last$ and calculates their difference, if the difference is larger than a hundred times of the ConnectionDescriptor's $average$, the corresponding TCP connection is considered timed out. The timeout checking thread then sends a timeout message to the corresponding packet receiving thread, which performs packets reconstruction when received this message.

## V. Critical Improvement Techniques

In this section, we will describe the critical techniques we employed to improve our system's performance.

### *A. DMA Load Balance*

The multi-core NPU has an independent DMA engine which cooperates with hardware threads through message transmission. If a thread wants to transfer data between multi-core NPU and the host main memory, it sends a message which contains information including data source address, destination address and data length to the DMA engine; when the message is received, DMA engine executes the requested operation and sends a message back to the corresponding thread indicating whether the operation failed or succeeded.

A problem occurred when testing our system: we make the client test program to initiate multiple TCP connections to the server test program simultaneously and start data transmission after the connections are established, but we found that only a few connections can conduct data transmission normally while the rest connections' establishment failed due to timeout. The reason behind this phenomenon is that multi-core NPU's DMA engine buffers DMA request messages in a stack, when a few connections are established, they start data transmission immediately and overwhelm the DMA engine with request messages, other connections' SYN packets can not reach to the host kernel network stack and thus timed out.

To solve this problem, we utilized the timeout checking thread to add a DMA load balance functionality for the packet receiving threads: it maintains an individual DMA request message queue for each packet receiving thread, packet receiving threads now send DMA request messages to the timeout checking thread instead of the DMA engine, the timeout checking thread buffers these messages in their corresponding queues and chooses one message at a time in a Round-Robin fashion between the queues, the chosen message is then sent to the DMA engine. When the requested operation is done, DMA engine informs the corresponding packet receiving thread of the operation result by sending a message to it. Our DMA load balance mechanism only adds one more message transmission and two additional queue operations for each DMA request, it is highly efficient and simple for implementation.

### B. Active ACK Mechanism

TCP sets a retransmission timer after sending data, if no corresponding ACK packet is received after the timer timed out, it retransmits data and performs congestion control by setting the data sending window size to 1 MSS(Maximum Segmentation Size). Consider the following situation: multi-core NPU sets data length threshold for packet aggregation larger than the amount of data the sending side can transmit before retransmission timer times out, it buffers received packets and waits for more to come, kernel network stack will not receive these packets thus no ACK is sent out, which eventually leads to data retransmission. What is worse, the retransmitted packets will be considered as redundant and discarded by the packet receiving threads, this makes data transmission unpracticable and violates our original goal for TCP acceleration.

An intuitive strategy for solving the afore-mentioned problem is to decrease multi-core NPU's data length threshold for packet aggregation, but the underlying network and timeout value of the data sending side are constantly changing, which makes it difficult for the multi-core NPU to choose appropriate threshold values for each TCP connection. Decreasing threshold value also means more interrupts and more packet headers for the host kernel network stack to process, consequently the system performance suffers.

TCP specifications tell us that duplicate ACKs are harmless, this fact leads to an alternative solution of our problem, we let the multi-core NPU mimic TCP's consecutive ACK mechanism: when a packet receiving thread receives a new packet, it performs packets reordering by inserting the packet to its corresponding ConnectionDescriptor's packets list, then it generates an ACK packet according to the Connection-Descriptor's $nextseq$ value and sends the ACK to the data sending side. Particularly, we set the data receive window size to be 0xFFFF, which makes the sending side to transmit data faster.

## VI. Implementation and Performance Evaluation

We implemented our system using XLS416 produced by RMI, it has 16MB PCIe shared memory and 4GB DDR2 DRAM. The PCIe shared memory can be read/writen directly by host computer and XLS416, thus we used it for information sharing and store data structures such as packet descriptors, device state variables and miscellaneous counters. The 4GB DRAM is used by network interfaces for packet buffering and by hardware threads to maintain data structures such as ConnectionDescriptors and ConnectionTables.

XLS416's 16 hardware based threads are allocated as follows: 1 packet sending thread, 1 timeout checking thread, 13 packet receiving threads and the last one is reserved for system shell program. Our system optimizes TCP's data receiving path, thus only one thread is used for sending packets and as many as possible threads are used for receiving packets.

Most multi-core NPUs produced by RMI and other manufacturers are also equipped with XLS416's key hardware features, and only differs in specific configurations such as processing core frequency, DRAM and PCIe bandwidth. Data structures and algorithms we employed in our system can be easily implemented on other multi-core NPU platforms, resource allocation can also be done in a similar way, thus we conclude that our system design is highly compatible.

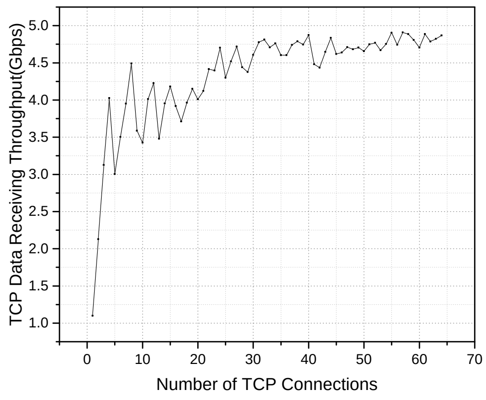

Fig. 6. TCP Data Receiving Throughput

Our system focuses on optimizing TCP processing on the data receiving path, thus we chose TCP data receiving throughput as the performance evaluation criterion. We built two simple test programs including server side and client side: the client side first requests multiple TCP connections with the server, and starts data transmission after all the connections are established; server side records the time interval between connections' establishment and data transmission's completion, then calculates the data receiving throughput. Since our system buffers TCP data packets on the multi-core NPU, it is not suitable for delay-sensitive applications such as online games, instead, it is particularly suitable for applications with large blocks of data transmission such as ftp and samba servers. The client host is equipped with Intel Core i3 530 2.93GHz CPU, 4GB main memory and Intel 82599 NIC, the server host has the same hardware equipments except that it uses XLS416 as NIC, both hosts run Red Hat Enterprise Linux 6.4, a 10Gbps optical fiber is used to connect them back-to-back.

During the test, we set the number of TCP connections from 1 to 64 and calculated data receiving throughput respectively, the results are shown in Figure 6.

From Figure 6 we can see that TCP data receiving throughput increases as the number of TCP connections increases when the latter is small, however, the growth slows down as the number of TCP connections becomes large and TCP data receiving throughput finally stabilizes around 4.75Gbps. The reason caused this phenomenon is the amount of packet receiving threads on XLS416 is fixed, when the number of TCP connections is small, XLS416 will dispatch packets to a subset of packet receiving threads and other threads are left idle, XLS416's computing power is not fully utilized; the situation changes as the number of TCP connections increases, packets will be evenly distributed to all threads and XLS416's computing power is exhausted, thus the maximum TCP data receiving throughput is achieved.

As a comparison, we also implemented XLS416 as a normal NIC which does not aggregate TCP data packets and tested TCP data receiving throughput under the same hardware and software configurations, surprisingly, we found that TCP data receiving throughput is approximately 913.3Mbps and irrelevant to the number of TCP connections. The following factors lead to this huge performance gap:

1) when implemented as a regular NIC without LRO functionaliry, XLS416's computing power and storage space are nearly not used at all, multiple connections' packets are distributed to different threads, the threads then compete for the sole DMA engine to transfer packets' data to the host memory, this leads to a serial style access of the DMA engine and the overall TCP data receiving throughput is no different from a single connection situation. What is even worse, DMA engine restarts each time a request arrives and only transfers a small block of data(1500 bytes maximum), which results in a very poor performance. However, when XLS416 is implemented with LRO, a packet receiving thread requests DMA engine access for TCP data packets only when one of the three conditions which we stated in section IV is satisfied, most of the time, it buffers the packet, performs packets reordering job and waits for more packets, meanwhile the DMA engine can be used by another thread for data transmission. This leads to a pipelined style access of the DMA engine, thread synchronization overhead and DMA latency are hidden in the multiple packet receiving threads, DMA engine now transfers a large block of data(approximately 64KB) each time it is activated and a much better performance is achieved;
2) the number of packets processed by host CPU and interrupts generated by XLS416 become much more;
3) notice that the client's data sending speed affects the server's data receiving speed, when implemented with L-RO functionality, XLS416 automatically generates ACK packets and causes the client side to send data faster, which in turn increases the server's data receiving throughput.

To minimize the number of packets and interrupts processed by the host CPU, we chose Linux kernel data structure skbuff's storage upper bound(64KB) as the data length threshold for packets aggregation. We utilized XLS416's network accelerator for hardware checksum verification of incoming packets, the packets reordering and reconstruction algorithms used in our system are designed to be simple and efficient, which do not involve any packet data copy operations, XLS416's memory bandwidth should not be our system's performance bottleneck. On the other hand, XLS416, is actually a low-end product, which connects to the host computer through a PCI-e1.1$\times$4 bus, we tested its performance and achieved only 6Gbps data transmission throughput, thus we draw the conclusion that XLS416's PCI-e bandwidth is the performance bottleneck of our system.

## VII. Conclusion

In this paper we proposed the implementation of LRO on a multi-core NPU platform, by aggregating data packets, we reduce the number of packets and interrupts processed by the host CPU and optimize TCP performance on an end system. Particularly, our system automatically generates ACK packets, which enables us to reconstruct very large packets(64KB) and makes the data sending side to transmit data faster. The experiment results demonstrate that our approach can significantly improve TCP data receiving performance.

In the near future, we plan to implement our system on multi-core NPU platforms with different hardware configurations, choose more evaluation criterions such as host CPU utilization ratio, and analyse the factors which can affect our system's performance in more detail.

## Acknowledgment

This work has been supported by the National Natural Science Foundation of China under Grant No. 61379148.

## References

[1] Wikipedia, "Ddr2 sdram — wikipedia, the free encyclopedia," 2016, [Online; accessed 17-May-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=DDR2_SDRAM&oldid=713378746
[2] ——, "Ddr4 sdram — wikipedia, the free encyclopedia," 2016, [Online; accessed 17-May-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=DDR4_SDRAM&oldid=713478122
[3] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "Cpu db: recording microprocessor history," *Communications of the ACM*, vol. 55, no. 4, pp. 55–63, 2012.
[4] R. Braden, D. Borman, and C. Partridge, "Computing the internet checksum," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 2, pp. 86–94, 1989.
[5] T. Mallory and A. Kullberg, "Incremental updating of the internet checksum," 1990.
[6] A. Rijsinghani, "Computation of the internet checksum via incremental update," 1994.
[7] K. Kleinpaste, P. Steenkiste, and B. Zill, "Software support for outboard buffering and checksumming," in *ACM SIGCOMM Computer Communication Review*, vol. 25, no. 4. ACM, 1995, pp. 87–98.
[8] H.-k. J. Chu, "Zero-copy tcp in solaris," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. Usenix Association, 1996, pp. 21–21.
[9] Y. Dong, D. Xu, Y. Zhang, and G. Liao, "Optimizing network i/o virtualization with efficient interrupt coalescing and virtual receive side scaling," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 2011, pp. 26–34.
[10] E. Yeh, H. Chao, V. Mannem, J. Gervais, and B. Booth, "Introduction to tcp/ip offload engine (toe)," *10 Gigabit Ethernet Alliance (10GEA)*, 2002.
[11] J. C. Mogul, "Tcp offload is a dumb idea whose time has come." in *HotOS*, 2003, pp. 25–30.
[12] G. W. Connery, W. P. Sherer, G. Jaszewski, and J. S. Binder, "Offload of tcp segmentation to a smart adapter," Aug. 10 1999, uS Patent 5,937,169.
[13] L. Grossman, "Large receive offload implementation in neterion 10gbe ethernet driver," in *Linux Symposium*, 2005, p. 195.
[14] J. Theman, "lro: Generic large receive offload for tcp traffic," 2007.
[15] T. Hatori and H. Oi, "Implementation and analysis of large receive offload in a virtualized system," *Proceedings of the Virtualization Performance: Analysis, Characterization, and Tools (VPACT08)*, 2008.
[16] G. Antichi, C. Callegari, and S. Giordano, "Implementation of tcp large receive offload on open hardware platform," in *Proceedings of the first edition workshop on High performance and programmable networking*. ACM, 2013, pp. 15–22.