Reducing the Per-packet Processing Overhead at a Receive-side Using Large Receive Offload

Dr. Mohamed Elbeshti
Dept. of Computer Technology, Faculty of Information Technology
Zawia University

Abstract:

Reducing the per-packet processing overhead is necessary for increasing inbound throughput at high-bandwidth network connections. Decreasing a host CPU overhead at the receiving-side by offloading part of the network protocols processing to Network Interface (NI) can be one of procedures for supporting high-speed communications lines beyond 10 Gbps. This paper presents an appropriate processing mechanism approach of the Large Receive Offload (LRO). The LRO methodology achieved by combining the received TCP/IP packets that have the same packet header information (from same stream) at a NI's buffer before

sending them to a host memory. Using a programmable-based core unit at a NI for processing the LRO functions increases flexibility and scalability for supporting different network protocols, like TCP/IP and UDP/IP protocols. This paper also used a single RSIC core processor in the NI for supporting the TCP/IP protocol processing at high speed networks. A RISC R2000/R3000 processor has been chosen as a target core engine for verified the proposed the LRO. The type and number of the instructions-set that is required for processing TCP/IP packets has been monitored. The results shown that A 917 MHz RISC core can support the TCP/IP packets at the NI's receiving-side when a communication line speed beyond the 10 Gbps. The RISC clock rate measured without data movements within the NI and when the Maximum Transport Unit (MTU) is set to 512 bytes or larger. This result can lead network designers to use a single processor at the network devices instead of using completed multicourse for serving the TCP or UDP packets at high-speed networks.

***Keyword**: TCP/IP protocol, RISC processor, LRO, NI*

1.Introduction.

The per-packet processing overhead is required in order to reduce the bottleneck especially after the high-speed communications lines become faster than 10 Gbps [1]. The data path of the Large Receive Offload (LRO) processing starts after receiving a valid TCP/IP packet from the Media Access Control MAC Unit . A host CPU interruption is required when there is a valid packet stored in memory. A DMA initiation then is required to pass each packet from the NI buffer through the system bus to its user space in the host memory [2]. Updating the

memory descriptors is required after placing a packet into the host memory. The CPU then starts to process the packet(s). One possible solution to reduce the pr-packet processing is the use of jumbo frames (9000 bytes with one header) that can carry up to 9000 bytes of payload in size, reducing header processing at the end node [3]. However, jumbo frames are not universally used within the Ethernet due to legacy compatibility reasons (RFC 791 - Internet Protocol). The original IEEE 802.3 specifications [4] defined a valid Ethernet frame size to be 1518 bytes as MTU. This paper provides a novel approach for Large Receive Offload (LRO) by combining the received TCP or UDP packets that have the same packet header information in the Network Interface (NI) buffer. This reduces the number of headers that a host CPU is required to process, reducing the number of packet headers passes from the NI through the system bus to the host memory. The proposed methodology for the LRO processing is to reduce the per-packet processing at the end node. The out-of-order packets (packets arrived out of sequence at the end node) is also implemented. The Packet Processing Unit (PPU) at the NI is designed to support the required processing for TCP/IP at high communication speed. The core engine is careful chosen to avoid the complexity of using multi-cores such as EZChip's NP-1-4, Intel's IXP1200, 2400, 2800, 2850 NPs, IBM's Power NP, when designing the NI. Using a programmable-based NI to implement proposed LRO functions increases flexibility and scalability by supporting the TCP/IP and/or UDP/IP . In addition, using a single processor for packet processing is also possible for simplifying the data path and sharing NI resources.

In this paper, the LRO processing has been verified using a SPIM Simulator based-RISC (R2000/R3000). The type of the instructions set

that is required for processing the LRO function has been recorded. The number of cycles required for processing TCP packets has been taken in order to measure the RISC's clock rate.

2. Related Implementations of Large Receive Offload Processing :

Xen [5] evaluated the LRO performance within the host area. The number of instructions that executed per packet in the guest domain for virtualized Linux environment is measured . The result shows that processing required for the LRO causes the CPU to spend 2927 cycles for a guest domain for 10 Gbps. The CPU instructions are required to process the virtual LRO data stream, which is around 1600 instructions, when the packet size is 1500 bytes. When 1500 bytes set as a default size, the end node receives around 812,743.82 packets per second (TCP standard flow). this is of course it raises the CPU usage [6]. According to Kumar and his team when measuring Xen of the Linux performance, the calculation of CPU usage when a virtual LRO is applied is as follows:

- Lets assuming the communication line speed is 10 Gbps, the packet size is 1500 bytes and the CPU clock speed is 4 GHz.

Then the determination of the actual MIPS rate and execution time for virtual LRO is;

CPI = (CPU Clock Cycles/Instruction count) = 2972 / 1600 => 1.85

; CPU spends 2927 cycles for a guest domain

; 1600 is total instruction count

Execution time calculation = Instruction Count * CPI * Cycle time

OR

= Instruction Count * CPI / Clock Rate

$= 2972 \times 1.85 \; / \; 4 * 10^{9} = 1.3 * 10^{-6}$ sec .... (1)

MIPS = clock frequency / (CPI * 1000000)

$= (4 * 10^9) / (1.85 * 1000000) \Rightarrow 2153$ MIPS .... (2)

From the estimated calculation (Equation 2) shows that over 50% of the CPU usage (4 GHz) is required to operate the LRO code whilst performing the LRO. The host CPU devotes more cycles to the completion of the LRO functions than other services when the number of incoming packets increases, especially when the packet size is smaller than 1500 bytes [7]. However, 1500 bytes packet needs an extremely shorter time at 100 Gbps (about 123.04 ns), but the budgeted time for supporting each packet in this calculation is only 1.3 ms (Equation 1), which is not enough time for processing a packet at 40 Gbps or100 Gbps.

The other methodology for reducing the pre-packet over head is the Receive Side Coalescing (RSC) [8]. The RSC is a stateless offload technology that has been used to reduce the use of the CPU and network processing on the receiving-side by offloading tasks from the CPU to the RSC enabled network adapter. This technique coalesces the packets into large packets inside the host memory and reduces TCP/IP processing by around 20 percent. However, this approach still carries a number of drawbacks, including the transferring of data from the NI to host memory. Data transfer involves initiating the DMA to transfer a payload area over the system bus, to be allocated to the host memory. Each TCP/IP stream leads to several DMA initiations and these initiations may cost the host CPU around 300 ns when the message size is 32KB [9]. The NI's engine is responsible for moving packets from NI's buffer to the kernel space at a host CPU using the DMA. Another concern with the RSC approach is that the out-of-order packets force the RSC to stop functioning. This is due to a number of small packets that are not coalesced. As a result, a host CPU could have more TCP/IP headers in

need of processing. There are also a number of processes associated with out-of-order packets processing, which require more CPU processing (such as looking up the Transmission Control Blocks (TCB) for packets that have not passed the RSC criteria test). This processing is estimated to take 25 CPU instructions [10]. Furthermore, the RSC does not provide any solutions for sending TCP data or a framework of relations between the host and the NI. The RSC does not indicate the use of other protocols (such as UDP/IP) nor are there any suggestions for an algorithm for any other protocol than TCP.

3- Designing a Scalable Interface for LRO:

Designing a scalable NI that supports LRO requires several factors should be considered. For insatnce, The NI has a core engine that can adequately identify the specifications of each arrived packet before linking them to form a large packet. The NI is capable to support different network protocols. The LRO processing is able to support the out-of-order packet processing.

In particle there are two approaches that can be used for the treatment of large packets to enhance TCP processing performance,. A selective acknowledgement (SACK) [11] mechanism can be applied as a suitable processing approach for LRO processing. With SACK, a receiver can inform the sender about all segments that have been successfully received, allowing the sender to resend the missing packets that have already been lost. The SACK approach also requires the approval of both parties and has to be explicitly enabled by the system administrator. Using SACK, an extra burden of processing and extra data is added to the stack processing inside the option area of the IP header.

The other approach is to send the final acknowledgment number of the amalgamated packets [12]. This will not change the protocol behavior of sending acknowledgements to the sender. Avoiding the retransmitting of data during the amalgamation of the arrived packets reduces the time of data amalgamation to be less than 0.5 ms [13]. This work has measured the maximum number of amalgamate packets can be stored in the NI's before sending them to host memory. For instance, if 200 TCP packet is arrived to the NI and the size of these packets is 500 bytes, the total time is 24600 ns (200 packets * 123 ns, the time required for a packet at 100 Gbps, which is less than 0.5 ms)

This paper does not attempt to specify in detail the congestion control algorithms for implementing TCP with acknowledgments but will illustrate the proper behavior of TCP processing within the proposed LRO functions within the NI. Amalgamating UDP packets is less complex compared to TCP since there are no traffic control requirements. Yet UDP packets have larger amounts of data (the payload) than TCP, which requires more time to be moved from one location to another inside the NI.

Designing a programmable NI that includes LRO functions to support high-speed communication rates of up to 100 Gbps is challenging. The processing includes managing the different virtual TCP connections that a host CPU makes with other remote hosts, and rearranging the out-of-order packets so as to combine more packets in the NI's buffer. The other considerations is required to be studied while designing the NI is the packet size. Supporting different data streams is also important, where an end node could have several data streams. Scalability, performance and complexity are also should considered when designing a NI.

The core engine is the heart of NI. Using of a single core unit for the NI to support the LRO functions for high-speed communication lines is one of the contribution of this work. Using a dedicated RISC as a core engine, one for the sending-side and the other for the receiving-side, will allow more features to be added to the NI, such as reducing the complexity of designing the NI with multiple core engines [14]. The NI designed by splitting the packet processing into three parts: the communication Line Interface (LI), Kernel processing, and the Host Interface (HI) (Figure 1). The HI and LI parts will be implemented within the hardware.

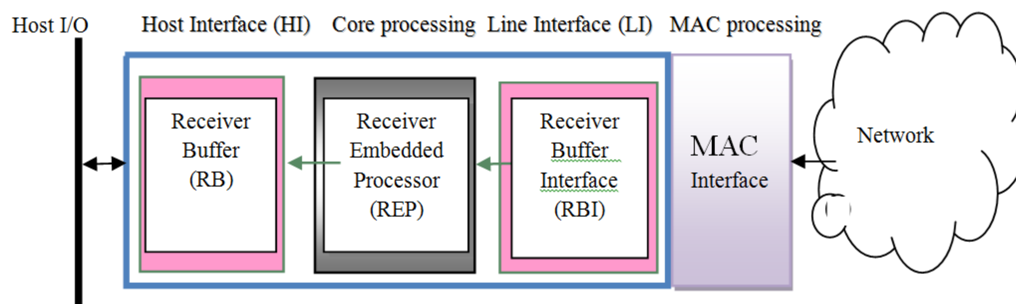


Figure 1: The Ethernet Network Interface structure

a. Receiving-Side Processing RSP

LRO processing determines whether the packet is eligible for amalgamation. The packet is not eligible if any of the following conditions applies for the TCP and the IP standard[15]: Non-Padded frame (IP total packet length must be received packet length), Non-TCP or UDP packet, IP options are present, IP ECN CE is set, TCP segment has no data, CWR (Congestion Window Reduced) flag is set, ECE (ECN Echo) flag is set, SYN flag is set, FIN flag is set, URG flag is set, ACK flag is not set, TCP Timestamp option is present, and the IP header does not have the MF bit set (More Fragments) and offset is zero.

If these types of packets are discovered by the embedded processor, it buffers the packets without any changes. The procedure of amalgamating packets runs until the Interrupt Moderation (IM) [16] timer expires or the network interface buffer reaches its quota. When amalgamating the packet of a stream, the embedded core is required to update the ACK number to be the same as the last packet's ACK. The total length of the packet (inside the IP header) also has the total size of the datagram. The UDP length inside the UDP header changes to the total size of the datagram.

A server deals with a number of users and each user is considered to be a separate connection. Each connection may have a number of virtual connections, each with a different identification. A copy of these identifications is required to be stored in the NI which helps the core within the receiving- side processor to manage incoming packets.

The main function of the receiver section of the NI is to amalgamate the arriving packets into a complete large packet by linking the packets that have the same identifier in the Receiver Buffer (RB) using a linked-list mechanism (explained more in next section). The linked-list mechanism by linking the packets from a single steam into one linked-list. The packet information also helps the Receive Embedded Processor (REP) core to recognize the packet types. For example, from the packet information, the REP determines if it is the Beginning of Message (BOM). The BOM is the first packet to arrive at the NI of a stream where the REP needs to create a new linked-list. The Continuation of Message (COM) is a packet that already has a linked-list created. The End of Message (EOM) is the last packet of a connection. This stops the linked-list if the Push flag (PUSH) inside the TCP header. The PUSH bit is not a record marker and is independent of segment boundaries. This packet is considered as the EOM packet of the stream. A Single Segment Message (SSM) is a single TCP message packet that carries the

application data that is equal to or less than 1500 bytes. Signaling packets such as FIN or SYN always flow among the TCP packets. The signaling packets are processed as SSM but have less processing requirements than SSM. For example, the RISC is not required to link the SSM to a stream. The RISC sends such packets 'as is' to the HI.

The TCP and UDP protocols required different processing scenarios to identify the sequence of the arriving packets. For instance, TCP has a Sequence Number (SN) field, which is used to detect the sequence of the packet. To maintain this stream, clients on either side of the TCP session maintain 32 bits SN that can have a value between 0 and 4,294,967,295 and an Acknowledged Number (ACK) which is used to keep track of how much data it has sent or received. The SN is included in each transmitted packet and acknowledged by the opposite host as an ACK to inform the sending host that the transmitted data was received successfully. UDP packets have *Identification*, fragment flag and packet offset fields which are used to identify the fragmented packets.

b. TCP Processing Methodology:

As soon as a TCP packet arrives at the NI, the IP and TCP headers will be processed. The IP address and the Port IDs are masked from the IP header and TCP header. The Lookup Memory is used to store the active connection information. The core engine sends the arrived packet's connection information to Lookup Memory to find a match (the connection information is sent by the CPU to the NI). If a match of the connection information between the arrived packet and the one in the Lookup Memory is found, then linking-list processing starts. The TCP processing methodology uses the SN bits for determining the arrived packets. Figure 2 demonstrates the BOM, COM, EOM and SSM within a TCP stream. The BOM is the first packet received from a TCP stream which carries the first SN (SN = n) and the ACK (m+1) of the stream. COM is a packet that arrived at the NI after the arrival of the BOM of the

same stream. EOM is the last packet of the stream when the PUSH is signaled [17]. The Single Segment Message (SSM) packet is the first packet of the stream that has the PUSH flag set or it carries small data (e.g., the packet size is equal or less than the MTU). Figure 2 illustrates the inter-packet processing methodology for receiving TCP packets. After getting a match between the packets from Lookup data, the REP then examines the packet's type. If the arrived packet is a BOM, the packet needs to be transferred from the Receiver Buffer Interface (RBI) to the Receiver Buffer (RB). If the packet is a COM or an EOM, the body of the packet needs to be linked with the previous amalgamated data of this stream inside the RB.

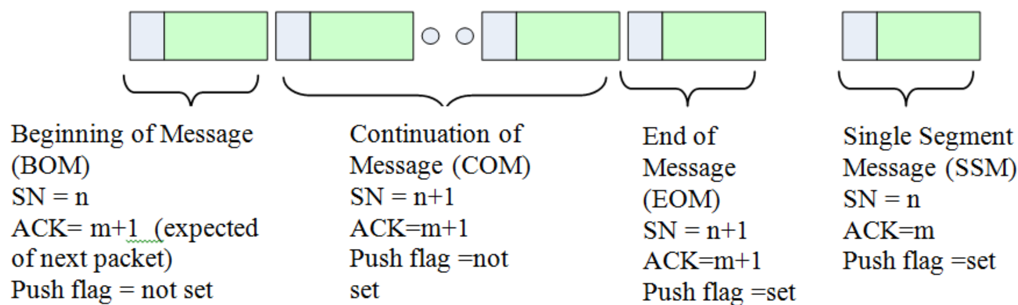
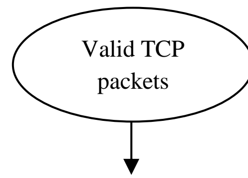


Figure 2: Beginning of Message, Continuation of Messages, End of Message and Single Segment Message of a TCP stream

The processing methodology for received TCP packets is illustrated in Figure 3. If the arrived packets pass the LRO criteria (e.g., packets are not SN or FIN) and a match between the packet connection information (the IP address and port ID) with the Lookup Memory entries was found, the REP then examines the packet type. If the arrived packet is a BOM, the packet needs to be transferred from the Receiver Buffer Interface (RBI) to the RB. If the packet is COM or EOM, the body of the packet needs to be linked with the previous amalgamated data of this stream inside the Receiver Buffer.


Valid TCP
packets

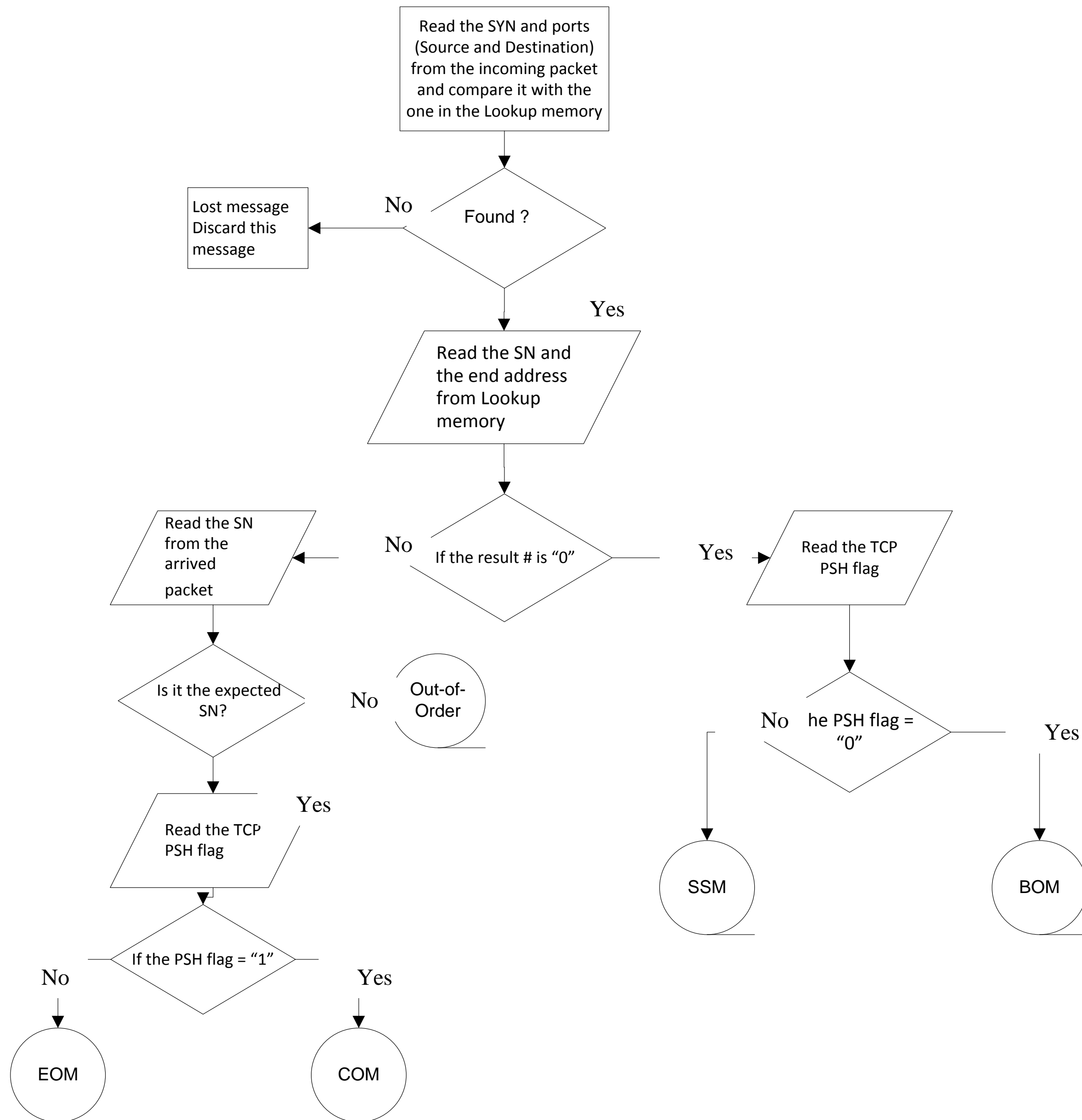


Figure 3: Processing flow of TCP of LRO

e. Linked-list Structure Format:

After inserting a new entry in the Lookup memory, all connection pointers, such as the Start-Address and End-Address of the packets inside the Host Interface (HI), are reset to zero. Change of these pointers depends on the processing that the entry packet requires. Each arrived packet may require different processing within the linked-list, and depends on its connection information, including the sequence number and Port ID. As soon as the TCP packet arrives at the NI, the IP and TCP headers will be processed. The IP address and the Port IDs have been masked from the IP header and the TCP header in order to match these identifiers with the Lookup memory. After the match has been found, the payload needs to join the same data packet in the RB. Instead of the REP searching for a free location inside the HI, a Circulation Buffer (CB) mechanism has been added to hold all free pointers inside the HI. The REP reads the head of the CB in order to get the address of a free location inside the RB for the arrived packet. The free pointers that refer to the available location inside the HI occurrences are collected after the host reads the amalgamated data.

The 32-bits SN in the TCP header assists the REP in identifying the received packets. For example, if the first arrived packet of a stream is the BOM, this means there is no linked-list previously assigned to a stream. The REP then needs to create a new linked-list for the arrived packet by inserting the Start-Address and End-Address in the Lookup memory beside the connection information in the packet headers (Figure 4). The Start-Address refers to the head of the linked-list (the address that is loaded from the CB for this packet). The End-Address refers to the tail of the linked-list, which is the Null address (Node's pointer), located at the end of the packet body. The SSM requires one pointer, pointed to the head of the list.

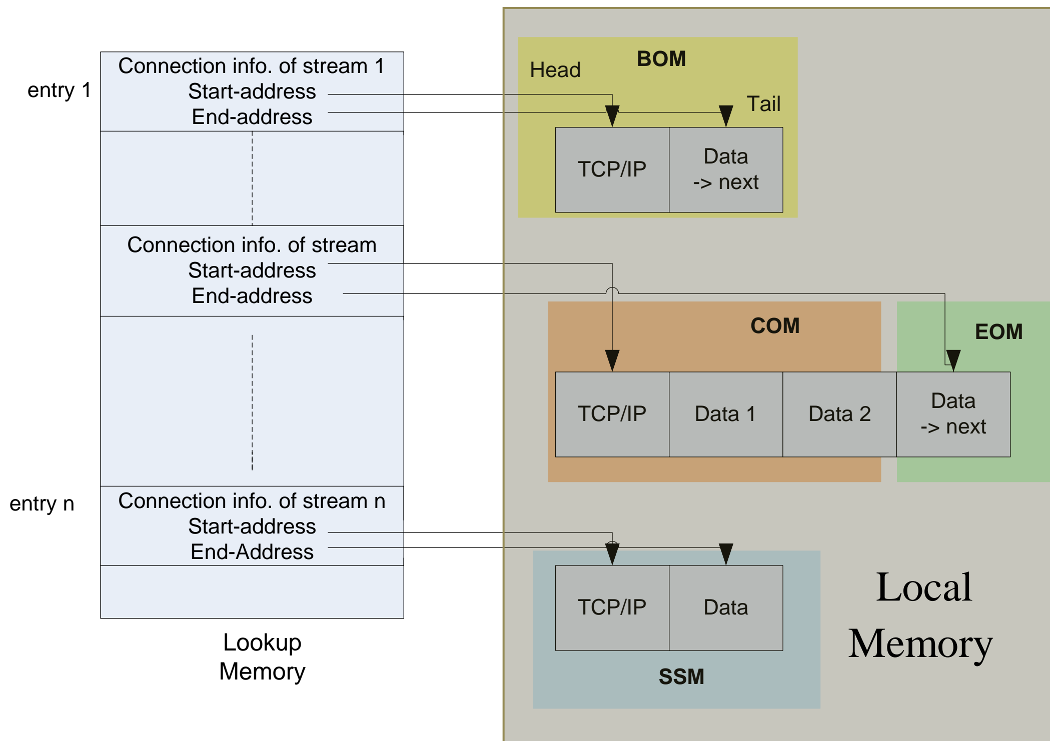


Figure 4: Linked-list data structure

COM refers to the packets arriving after the BOM that have the same connection identifier. After a match of the IP and Port IDs is found in the Lookup memory, the REP starts adding a new node to the existing linked-list (the packet body and its pointer). The linked-list is updated after adding a new node by setting the current node pointer of the node to NULL (End-Address). Next, the REP stores the NULL address of the current node at the Lookup memory which refers to the new end in the list.

If the EOM is discovered the REP needs to stop amalgamating the packets for this stream. The REP then appends the EOM packet to the related stream inside the RB. Initiating the DMA to transfer the packet body from the RBI to the RB buffer then deletes the linked-list of the

stream. The End-Address in the Lookup memory, which refers to the NULL value of the previous packet, is read by the REP. The REP stores the address of the current packet in the same place as the NULL value of the previous location (End-Address). The REP is responsible for updating the TCP header of the original TCP/IP packet of the amalgamated large packets which are inside the RB. Furthermore, it needs to update the length of the datagram to the total amalgamated number of bytes of the stream inside the IP header and acknowledgement number inside the TCP header. When the PUSH flag is equal to “1”, the REP examines the Lookup memory’s Start-Address and End-Address of the connection. If the Start-Address and End-Address are equal to “0”, then there is no linked-list assigned to this stream; thus the packet is moved from the Line Interface to the RB buffer. The REP then completes this packet as a SSM. The REP then stores the NULL value at the end of the packet body. Furthermore, there is no need to update the Start-Address and End-Address in the Lookup Memory because no more packets will be amalgamated within this stream. In this case, the current packet is amalgamated to the previous packet for the same stream, which then stores the NULL value at the end of the current pointer node. Finally, the NULL address is stored in the Lookup Memory with the same link information, which refers to the End-Address of this TCP stream. When the PUSH flag (inside the TCP header) is equal to “1”, the REP needs to add this node to the end of the linked-list, following the same procedure as that used for EOM. With EOM, there is no need to extend the linked-list further because it is the last packet of the TCP/IP stream, and there is no need to store the End-Address in the Lookup memory. However, when the Push flag is equal to “1” and the Start-Address and End-Address of this connection is equal to “0”, the current packet is moved from the Line

Interface to the RB buffer. The REP then processes this packet as SSM. With SSM, the REP does not need to update the Lookup memory entries that are related to this message, such as the Start-Address and End-Address of the linked-list, since no more packets will be amalgamated after this packet.

f. Out-of-order Processing at the RSP:

The Out-of-order procedure is a more complicated form of processing than BOM, COM or SSM. The RSC prototype [8] prefers to stop the coalescing of packets when an out-of-order packet is identified, whereas the LRO [9] opens a new queue if out-of-order packets are discovered. These steps lead to an increase of processing cycles for the host CPU. In this research, a new approach has been implemented to manage the out-of-order processing inside the NI. The out-of-order processing starts after the REP reads the sequence number (SN) of the arrived TCP packet. It is then compared to the SN expected to be reached at this linked-list. In the next step, the REP joins the arrived packet to the linked-list. In the case where the SN is not located between the boundaries of the amalgamated stream, the REP creates a sub-linked-list of this stream (Figure 5). A duplicate data segment will be discovered after checking the SN of the TCP stream that has been amalgamated beforehand in the RB. Furthermore, the REP discards any lost packets that do not have a match with the Lookup entry. However, the re-ordering can be performed within the Interrupt Moderation size only. If the lost packet of the same stream arrives after the Interrupt Moderation has expired, it will be sent to the host as a Single Segment Message (the mechanism of the retransmission of lost packets and calculating the time are out of the scope of this paper).

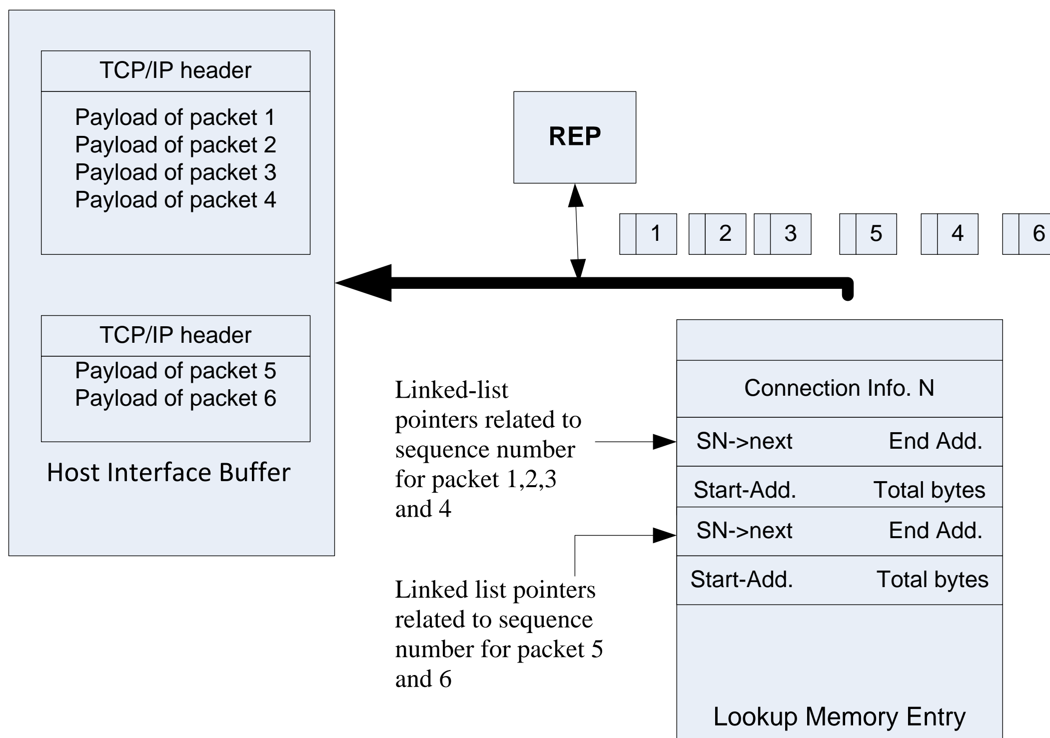


Figure 5: Lookup Memory structure

The REP is responsible for updating the TCP header (e.g., the sequence and acknowledged number) for stored large packets. The total length of the packet is also updated according to the total amalgamated data. The receiver host CPU sends a confirmed acknowledgement sequence number to the sender to inform the sending host that the transmitted data was received successfully. When data segment is sent, a timer then is started. The sender expected to get an ACK before the timer expires. Lost packet, or hole, which could occurs during transmission of the TCP stream, the destination host did receive the ACK, then it rearranges the re-send the holes according to sequence number that received from the receiver host. This sequence number is included on

each transmitted packet, and acknowledged by the opposite host [18]. When the packet arrives at the NI, the REP processes the packet as SSM and sends it to host "as is".

4. Modelling SPIM Simulator Architecture:

The SPIM simulator has been used to process the LRO functions, including communication with the host, data movements and packet header processing (Figure 6). Since it is implemented within the NI, sending and receiving are processed in two different processors and both are run in parallel. The simulation for the sending function can be performed independently of the receiving function.

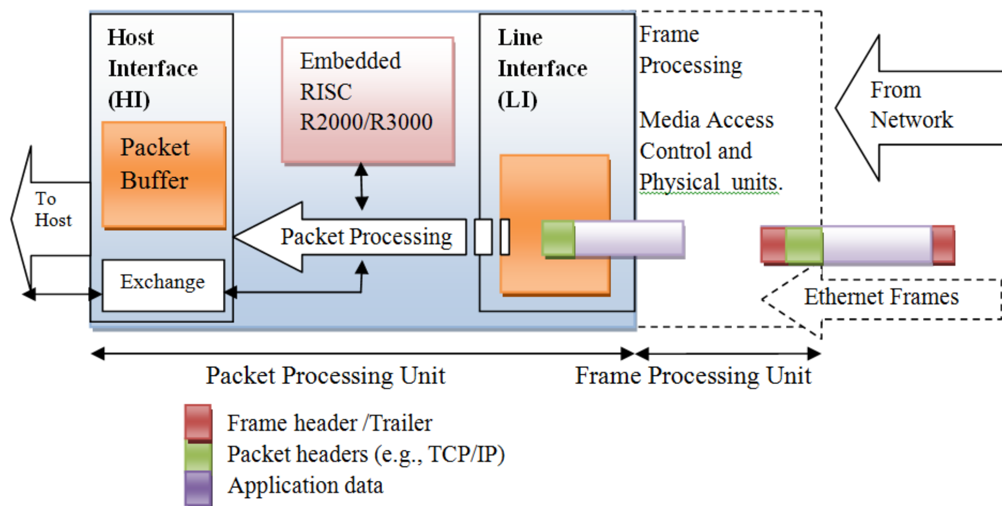


Figure 6: Receiving block diagram

The components of the Packet Processing Unit have been configured within the SPIM simulator. The embedded RISC is the core unit. The packet buffers required for LRO are implemented inside the simulator's memory. These buffers are:

- The Line Interface (LI) buffer to store the packets as they are moved from the MAC.
- The Host Interface (HI) buffer which is used to store packets before delivering them to the host. While transferring packets from the HI, the NI's core is able to store new packets in the lower space of the HI.
- The Host-NI Communication (HNIC) buffer is used to exchange control and status messages between the host and the NI. This includes sending the addresses of the packets that have been sent by the CPU inside the Packet Buffer.
- Another buffer is used to store a list of pointers in the free space inside the HI buffer. After the host reads the packets from the HI, the free pointers are sent back to the NI. The embedded processor uses these pointers to store the arrived packets.
- The Look-up table stores the active TCP connections (e.g., the destination IP address and the source and destination port IDs) that are required to hold the active identifiers to support the link-list mechanism.

In a physical NI, the LI, HI and HNIC buffers are hardware components. The Look-up table is implemented within the simulator’s memory.

As the SPIM simulator does not have a DMA unit, instead a programmed I/O is applied for data movement. The embedded processor core is permitted to simulate the initialization of the control information for the DMA controller but not the data movement itself. This makes the simulation processing extremely close to reality, where the processor at the Packet Processing Unit needs to initialize the DMA controller to move data. With the Programmed I/O method, the embedded processor handles all the procedures for moving the packet payload from the LI

buffer to the HI buffer . Each clock cycle, the processor loads 32 bits to its register (step1) and then during the next cycle, stores these 32 bits inside the HI buffer (step 2).

The embedded core continues to load and store operations until all bytes have been moved to the HI buffer. The copying of data packets from one location to another can be done by using the Copy Memory Address function, because the source and destination address are stored in the same memory. However, using load and store is very close to reality, as the LI and the HI are implemented as separate memory.

4.1 Simulation Processing Analysis *:*

The purpose of this section is to validate the LRO processing algorithm through the SPIM simulator. To get accurate results from the LRO processing at the SPIM simulator, captured data from a real network environment is used. Two other applications are used to capture the real data. The first application is the Microsoft TechNet NTTTCP [19], which was developed by Microsoft, to send and receive the TCP and UDP streams. The NTTTCP is used to send data applications between computers. The other application is Wireshark Libpcap, which is used to capture the packet flow of the streams that are running at the end node [20]. The NTTTCP application starts sending the TCP/IP and UDP/IP streams to the targeted machine. The application followed by captures the arrived packets using Wireshark Libpcap and then exports the data of the TCP/IP and the UDP packets in hexadecimal which then form a Hexadecimal format file (Figure 7). Then, the streams of the real packets are stored inside the LI buffer. The specialized instruction set was developed to create efficient TCP or UDP processing based on the virtual LRO [2] algorithm and following the RFCs for TCP/IP or UDP/IP

[11,12,13,15,17]. As the captured files are in sequence and there is no lost or duplicated data, manual changes of the Hexadecimal file have been made to simulate lost and out-of-order packets. These changes include the change to the position of the ordered packets or the deletion of a packet from a stream.

```
+---------+--------------+----------+
01:59:09,206,211   ETHER
|00|22|75|29|ec|95|20|68|9d|9a|34|6c|08|00|45|00|01|27|13|ef|40|00|80|06|a9|0f|c0|a8|02|08|4d|ea|2c|38|d1|42|00|50|8c|ef|0e|
c0|62|07|23|00|50|18|00|40|ac|8c|00|00|47|45|54|20|2f|52|2f|41|31|51|4b|49|47|4a|68|4e|6a|64|6a|4e|6a|49|33|59|7a|63|33|5a
|54|52|6a|4d|32|45|34|59|32|45|31|4e|7a|45|33|5a|54|52|6c|5a|44|49|35|4e|7a|56|6a|45|67|51|41|42|77|59|54|47|4f|51|42|49|6
7|45|44|4b|67|51|49|41|78|41|41|4b|67|51|49|42|52|41|42|4b|67|63|49|42|.
.......
+---------+--------------+----------+
01:59:09,527,074   ETHER

|20|68|9d|9a|34|6c|00|22|75|29|ec|95|08|00|45|00|00|28|ad|86|40|00|32|06|5e|77|4d|ea|2c|38|c0|a8|02|08|00|50|d1|42|62|07|2
3|00|8c|ef|0f|bf|50|10|00|01|7f|b8|00|00|
+---------+--------------+----------+
01:59:09,815,821   ETHER
20|68|9d|9a|34|6c|00|22|75|29|ec|95|08|00|45|00|00|c2|ad|87|40|00|32|06|5d|dc|4d|ea|2c|38|c0|a8|02|08|00|50|d1|42|62|07|2
3|00|8c|ef|0f|bf|50|18|00|01|a8|e2|00|00|48|54|54|50|2f|31|2e|31|20|32|30|30|20|4f|4b|0d|0a|43|6f|6e|74|65|6e|74|2d|54|79|7
0|65|3a|20|61|70|70|6c|69|63|61|74|69|6f|6e|2f|6f|63|74|65|74|2d|73|74|72|65|61|6d|0d|
1:59:09,817,228   ETHER
|20|68|9d|9a|34|6c|00|22|75|29|ec|95|08|00|45|00|05|ae|ad|88|40|00|32|06|58|ef|4d|ea|2c|38|c0|a8|02|08|00|50|d1|42|62|07|2
3|9a|8c|ef|0f|bf|50|10|00|01|c8|46|00|00|31|34|0d|0a|03|12|08|e4|01|12|01|fe|32|0a|08|04|10|ed|c9|cb|15|18|80|0a|0d|0a|31|6
4|66|34|0d|0a|0a|f1|3b|41|53|55|21|56|50|53|7a|02|07|06|13|27|00|00|00|a9|1d|00|00|f4|26|00|00|78|da|25|9a|77|3c|d6|6b|14|
c0|5f|e3|b5|67|46|52|89|54|b6|b2|b2|ae|ac|ac|24|91|2d|59|65|64|54|46|11|4
                                                                                                                   .......
```

Figure 7: A snapshot of TCP/IP Hexadecimal format

Performing the processing of out-of-order packets in the NI enhances the packet processing, where the target core engine has to manage the received packets and puts them in order before sending them to host memory (Figure 8). Assuming that there are 8300 bytes of application data sent from one end to another and that the MTU is 1500 bytes, six packets of application data will travel from the sender to the destination. Each packet carries 1460 bytes of application data, except the

last packet, which will carry 1000 bytes. In addition, there will be 40 bytes of TCP/IP headers encapsulated within 1460 bytes of application data. With the Linux LRO implementation, all packets will pass through the system bus (application data + TCP/IP headers), assuming that these packets have arrived out-of-order at the destination. If the order of these packets is the following packets: 1, 2, 4, 3, 6 and 5. With virtual LRO processing, these packets need four queues inside the host memory. Packet 1 and 2 will be stored in the first queue, packets 4 and 3 will each be stored in a new queue because they are out of sequence. Packets 5 and 6 will be stored in the last queue. Table 1 illustrates the processing scenario where the virtual LRO is implemented to support the out-of-order packets. The table presents the comparison between the proposed LRO inside the NI and the virtual LRO.

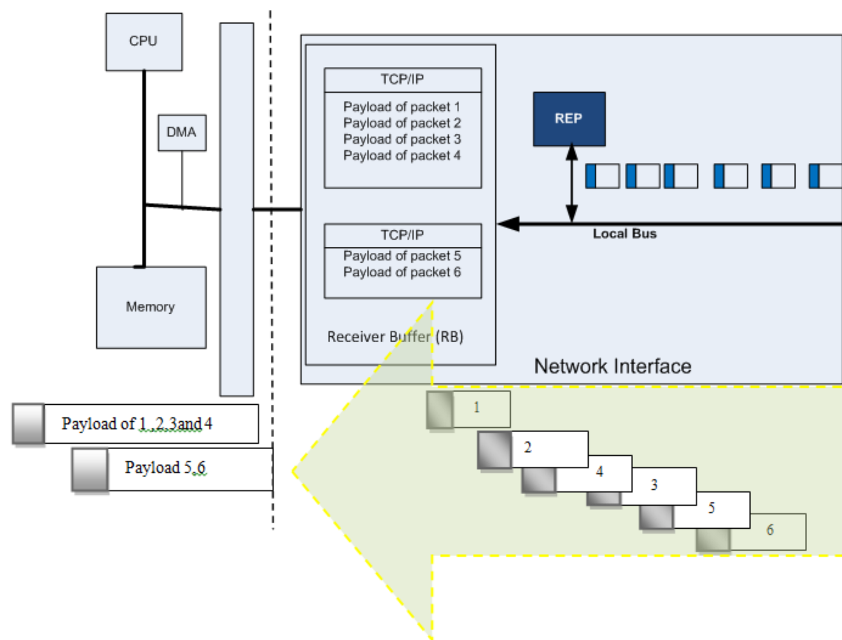


Figure 8: Offloading the LRO approach to the Network Interface

Table 1: A comparison between the virtual LRO processing (RSC) and the proposed offloaded LRO

	Virtual LRO (RSC)	Proposed LRO in network interface
Data size	8300 byte	8300 byte
Total headers passed over the system bus	Six TCP/IP or UDP/IP header	Two TCP/IP or UDP/IP header
Estimation of DMA initiation	Six DMA initiations	Two DMA initiations
Segment header overhead	The overhead is 240 bytes (40 X 6 TCP/IP header)	The overhead is 80 bytes (40 X 2 TCP/IP header)
Processing the out-of-order	Not ordered packet. According to the LRO implementation, it needs 4 queues Queue1 packet 1 and 2 Queue2 packet 4 Queue3 packet 3 Queue4 packet 5 and 6	Only two queues
Processed by	A host CPU processor	Network interface's engine

4.2 Instruction Cycles :

During the simulation, the amount of processing required for network interface protocols and data movement is measured. The number of instructions required for the LRO functions processing is also determined for TCP and UDP. After the embedded processor finishes processing one packet, it fetches the new connection identifier or a pointer that has been sent by the host through the HNIC. After finishing amalgamating the large packets in the HI, the core processor is required to send the Start-Address and the total bytes to the HNIC. The amount of the execution that the processor takes for different types of operations for TCP and UDP has been analyzed during this simulation. The analysis of the type and the total of implementation instructions needed by the embedded processor to complete the address of each BOM, COM and EOM message individually for TCP/IP and UDP/IP were studied. Table 2

presents the number of cycles required to complete the out-of-order packets for a TCP or a UDP packet within the SPIM simulator.

The Programmed I/O is applied for the data movement. The RISC core spends over 400 cycles on moving 1500 bytes packets from the HI to the LI. As the packet size is reduced, the number of cycles becomes less. The RISC required 45 cycles to complete the processing of the 64 bytes TCP packet and 56 cycles for the processing of UDP packets. UDP has higher cycles since it needs more Programmed I/O cycles.

Table 2: Number of cycles needed to complete out-of-order TCP packets

Protocol	**Type of processing**	**64 bytes**	**128 byte**	**256 bytes**	**512 bytes**	**1024 bytes**	**1500 bytes**
TCP	Header processing	29	29	29	29	29	29
	Lookup data	10	10	10	10	10	10
	Moving data	6	22	54	118	246	365
	Total	45	61	93	157	285	404

The overall processing percentage when the processor processes the TCP packets is determined (Figure 9). The percentage of moving data is increased when the data packet gets larger. The highest processing percentage of packet processing is reached when the packet size is 1500 bytes. This is because the core requires more processing to move the payload of the packet than other types of smaller packets. The shaded area in the Figure 9 shows that the RISC requiring nearly 90 percent of the total cycles for moving of the payload data of 1024 bytes or larger.

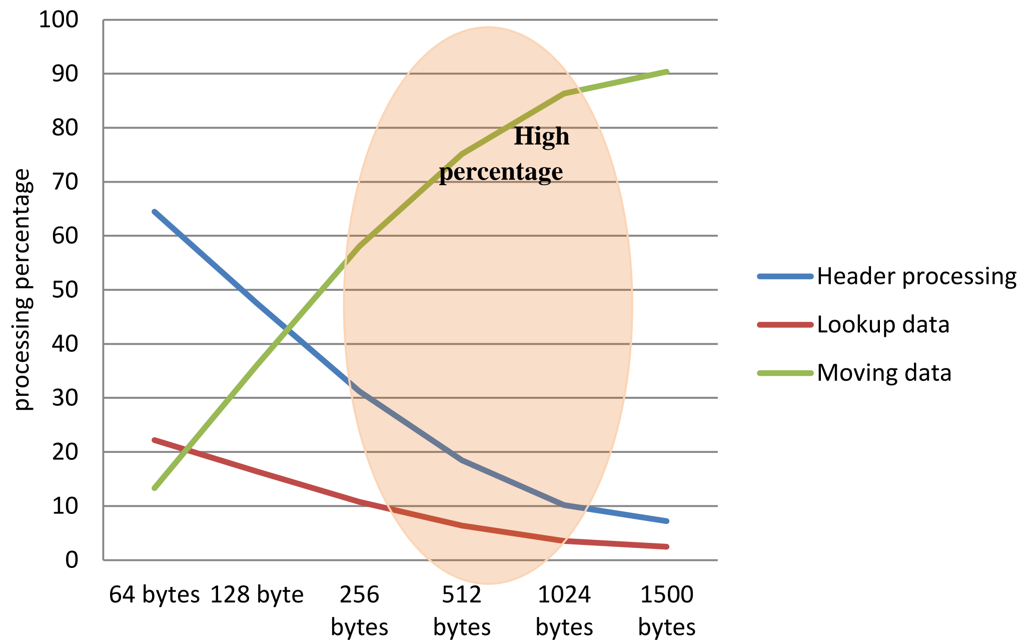


Figure 9: Total percentage of data movements of LRO

4.3 Instruction Type:

The instruction-type is required when designing a processor for LRO. Using selected instructions that can run the LRO processing functions reduces the design complexity and makes the RISC controller simpler and faster. The instructions used are memory-register, register-to-memory, arithmetic and logic instructions. In this work the instruction-set that are required for the LRO is recorded. Table 3 illustrates the instruction types and their format while processing the LRO.

4.4 Total Registers:

SPIM processing for the proposed LRO required 32 general registers (r0 to r31). Floating Point registers are confirmed as not required during the processing of the proposed LRO functions (Figure 10). All the Floating Point resisters remain zero.

Table 3: Instruction types that are used with LRO processing

Category	Instruction	Format	Examples	Comments
Arithmetic	Add	add $s10, $s2, $s3	Accumulating the total bytes	Three operands; data in registers
	Add immediate	addi $s8,$s4,40	Adding the size of headers to the total bytes	Used to add constants
Data transfer	Load	lw $s1,50($s2)	Loading the TCP, UDP or IP from LI. $s1 = Memory[$s2 + 50]	Data from memory to register this include LH and LB
	Store	sw $s10,100($s2)	Store total bytes in HI Memory[$s2 + 100] = $s10	Data from register to memory, including SB and SH
Logic	Shift Right	sr $11,20	Shift data register to the right	Used for shifting data
	And	and $s10,$s1,x00ffffff	Extract the total length form the first 32 bits of the IP header	Used to mask the register with constant
Condition branches	Branch equal	beq $s1,17	To check the protocol type (UDP = 17)	Condition jump

Figure 10: Floating Point registers during the processing of the proposed LRO

5. RISC Clock Rate:

The simulator has measured the amount of processing that is required for TCP/IP and UDP/IP protocol processing and for data movement. Different TCP/IP and UDP/IP packets have been delivered to the simulator. The number of processed instructions required for the protocols, with and without data movement processing, were measured in MIPS, where every instruction was processed in one cycle. Therefore, the results shown below represent the required speed of the RISC core in terms of the MIPS, while processing the requirements for header processing alone without data movement or lookup data for both TCP/IP and UDP/IP. The RISC required 29 cycles for the header processing when out-of-order packets are processed. The Lookup processing

required 10 cycles. Figure 11 represents the maximum Receive Embedded Processor (REP) clock rate required to perform header processing for the out-of-order packet sizes of 1500, 1024 and 512 bytes. It is clear that the clock rate of the RISC core gets higher whilst performing the processing of the smaller sized packets. This is obvious because the RISC is required to process about 234962406 packets in a second when the MTU is 512 bytes and around 81274382 packets when the MTU is 1500 bytes [23].

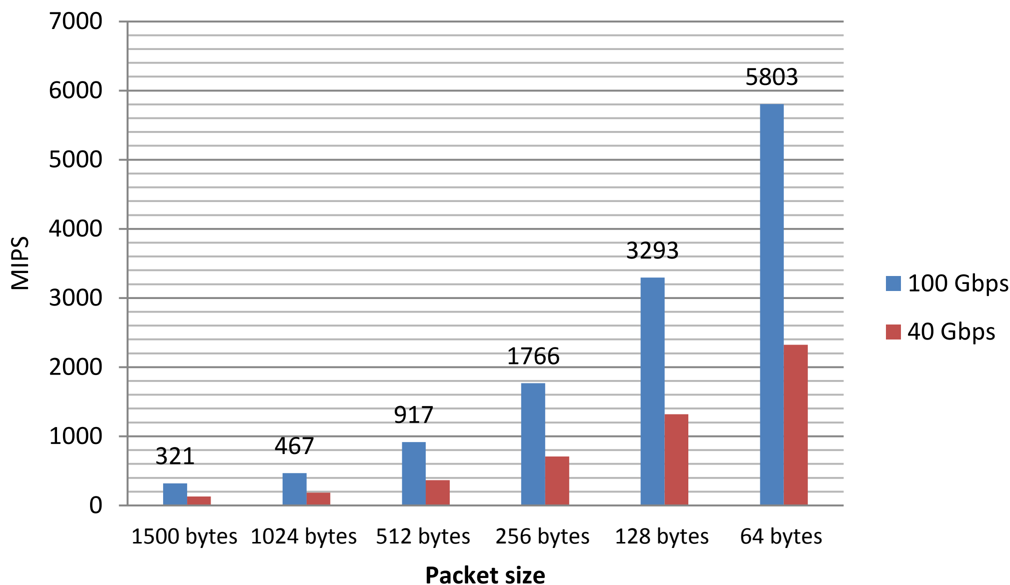


Figure 11: RISC clock rate for packet header processing

The data movement for the packet payload is simulated using the Programmed I/O for data movements. The amount of MIPS required for packet processing varies depending on the packet size. Even though the small packets carry less data, the core has to run faster to manage the processing of the packets at 100 Gbps. A RISC core with 3322 MIPS is required for the LRO when 1500 bytes packet is processed and the line

speed is 100 Gbps. A core with 9492 MIPS is required when the packet size is 512 bytes (Figure 12).

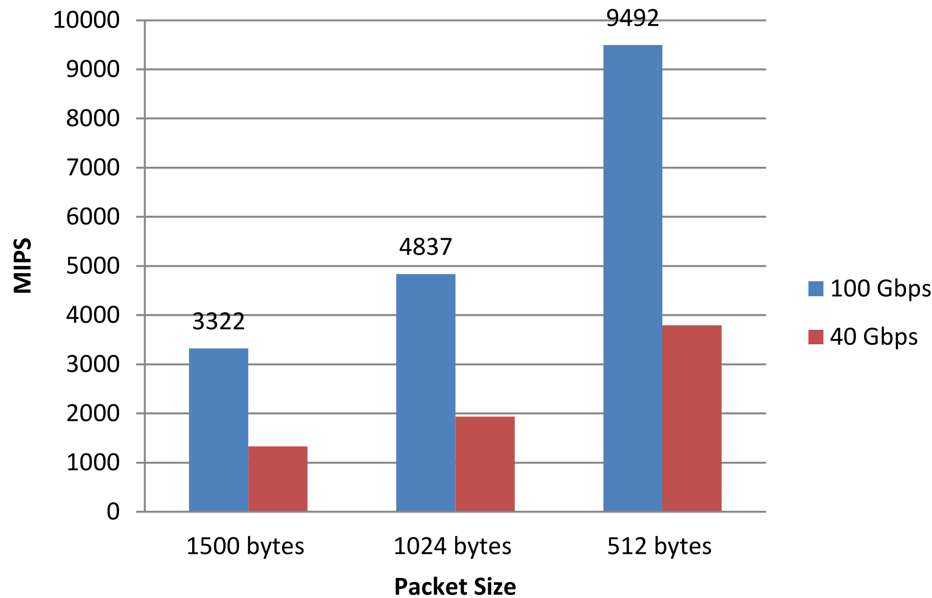


Figure 12: MIPS required for the Receiving-side using Programmed I/O

5.2 Network Interface Design Considerations for 100 Gbps:

According to the SPIM simulator, 404 cycles are required when the out-of-order 512-byte packets are processed. Therefore, the 9492 MIPS of the RISC cycles was caused by several factors. The first factor was that the core uses the Programmed I/O (P I/O) for the data movements approach. With the P I/O, the core needs to move each 32 bits into local register and then move it form the local register to its destinations place. The second factor is the local bus width, which is 32 bits. Other factors include the Lookup processing and RISC pipeline stages. monitoring the source and destination memories addresses is also factor for increasing the RISC/s cycles

5.3 Future work for designing the NI :

Using the Programmed I/O for data movement is slow because there are too many unnecessary overheads for small transfers [21] and the RISC core is tied up with moving data from one location to another, especially when the core requires moving large amounts of data. This affects the performance of the RISC core, making it unavailable for other activities. The use of DMA in the NI is more efficient for network applications than the programmed I/O if the DMA clock rate is adequate to be designed with the NI . Furthermore, when the DMA is used, the transfer of data from one place to another, the local bus is busy and the processor cannot perform any instructions related to using the local bus. Owing to this factor, the treatment of processes within the NI should be carefully selected and takes advantage of the processor's ability to complete other tasks that are not related to the use of the local bus.

From the mentioned aspects and more, the DMA becomes an appropriate approach in the future design as a method for the data movement function of the NI within the proposed model. In the proposed model, the DMA is responsible for moving data between the LI and the HI (Figure 13). Step 1, the RISC core initiates the DMA controller. Since the local bus of the receiving-side is shared between the DMA and the RISC core, the RISC core will have to release the local bus to the DMA to perform the data block transfer. Each transfer of 64 bits consumes two cycles. In Step 2, the second DMA cycle reads the 64 bits from the source buffer to the DMA's register. Step 3, the 64 bits move from the DMA's register to the destination buffer. The DMA state machine will then provide the read and write signals to the source and destination buffers. The state machine in the DMA is also required to increment the address counter and store 64 bits in the destination buffer. The use of the DMA reduces the RISC processor instruction cycles.

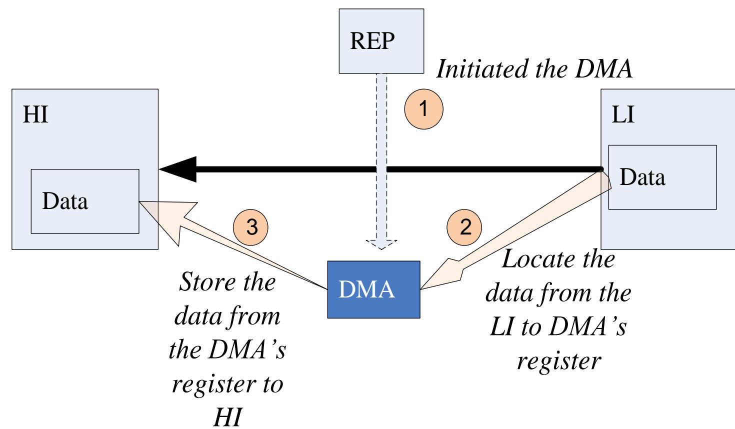


Figure 13: DMA approach for data movement inside the Network Interface

Local Bus Width Each load or store from the LI to the HI can hold up to 32 bits. This is because the local bus width of the SPIM simulator is 32 bits. Using a wider bus than 32 bits for the proposed NI increases the amount of data moved within each cycle (e.g., 64 bits). Future enhancement. the SPIM simulator has 4 pipeline stages. These stages consist of fetching the instructions, decoding, executing and writing back. Reducing the pipeline stages is developed to improve the execution throughput of the RISC [21]. The proposed RISC's pipeline will be discussed in chapter 6.In addition to managing the active connection information can be accessed by the NI's engine, either by using a fast look-up-table, such as Content Addressable Memory (CAM) [22], or by accessing the TCP/IP Control Block (TCB) from the host

When the DMA use, During the transfer of data from one place to another, the local bus is busy and the processor cannot perform any instructions related to using the local bus. Owing to this factor, the treatment of processes within the NI has been carefully selected and takes advantage of the processor's ability to complete other tasks that are not related to the use of the local bus. For example, in the case of receiving packets, the processor is made to deal with the linked-list in the CAM,

carrying out such tasks as updating it or reading from it. Using scheduled jobs is an appropriate approach [21], but this is only useful with multi-core processors where more job numbers are required. Using the overlapped technique reduces the packet processing time available to complete a packet. In addition, these enhancements of the NI's structure could increase the performance of packet processing in high-speed networks.

Conclusion:

Large Receive Offload (LRO) is a technique used to support reducing the pre-packet overhead at the end node. The methodologies of amalgamating the TCP and UDP functions have been discussed. The proposed LRO function works similar to the Jumbo Frame (9000 bytes) mechanism. This paper extends the existing LRO by supporting the out-of-order packets processing. Using a single processor at the NI in this research shall contribute to improving the structure of the network card in terms of scalability, simplicity and the ability to support a high-speed rate such as 100 Gbps.

The SPIM simulator results show that the A 917 MHz RISC core can support the Receiver unit processing for a transmission speed of rate up to 100 Gbps for TCP/IP without data movements when the MTU is 512 bytes or larger (without data movement). However, using Programmed I/O for data movements increases the RISC's cycles to be over 9000 MHz when the packet size is 512 bytes.

The future Enhancing in the receiving-side processing can be accomplished by adding additional units such as the Content Addressable Memory CAM to store the active connection information and its location inside the Receiving Buffer. In addition, the NI uses the DMA's unit to complete the transfer of data from the Line Interface to the Host Interface, or vice-versa.

References :

[1] *IEE Standard P802.3bm " 40 Gb/s and 100 Gb/s Fiber Optic Task Force" IEEE-SA Standards Board on 16 February 2015 http://www.ieee802.org/3/bm/ [June 2017].*

[2] *L. Grossman. Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In Linux Symposium, 2005.*

[3] *A. Das and S. Debbarma. Performance of Jumbo Sized Data on Jumbo Frame and Ethernet Frame Using UDP over IPv4/IPv6. Advanced Computing, Networking and Security (ADCONS), 2013 2nd International Conference on. 15-17 Dec. 2013.*

[4] *IEEE Standard "Local and metropolitan area networks 802.1D," http://www.dcs.gla.ac.uk/~lewis/teaching/802.1D-2004.pdf, 2004, [Jan 2017] .*

[5] *K. Kumar, J. Renato, Y. Turner and L. Alan "Achieving 10 Gb/s using safe and transparent network interface virtualization," Proceedings of the 2009 ACM SIGPLAN/SIGOPS, international Conference on Virtual Execution Environments, March 2009.*

[6] *X. Pu et al., "Who is your neighbour: net I/O performance interference in virtualized Clouds," IEEE Transactions on Services Computing, VOL. 6, No. 3, pp 314-328, 2013.*

[7] *S. Makineni and R. Iyer, "Measurement-based analysis of TCP/IP processing requirements," In 10th International Conference on High Performance Computing (HiPC 2003), Hyderabad, India, Dec. 2003.*

[8] *S. Makineni et al. "Receive side coalescing for accelerating TCP/IP processing," HiPC 2006. LNCS 2006, pp. 289-300.*

[9] *H. Jin and C. Yoo. "Impact of protocol overheads on network throughput over high-speed interconnects: Measurement, Analysis, and Improvement," Journal of Supercomputing, Vol 41, Number 1, July, 2007.*

[10] *Doug Freimuth " Server network scalability and TCP offload," USENIX '05 Paper. USENIX Annual Technical Conference, 2005.*

[11] *E. Yilmaz "Throughput analysis of Non-Renegable Selective Acknowledgments (NR-SACKs) for SCTP". Computer*

Communications. Volume 33, Pages 1982-1991. Issue 16, 15 October 2010,

[12] *F. Gont and A. Yourtchenko, "On the implementation of the TCP urgent mechanism," Internet RFC 6093, 2011.*

[13] *V. Paxson et al "Computing TCP's retransmission timer" Internet RFC 6298, 2011.*

[14] *NP-4, "100-Gigabit network processors for carrier Ethernet applications, product brief," EZchip Technologies, 2010.*

[15] *J. Heffner, M. Mathis, B. Chandler, and J. Heffner. "IPv4 Reassembly at high data rate," RFC 4963, 2007.*

[16] *Intel. "Interrupt Moderation Using Intel® GbE Controllers," download.intel.com/design/network/applnots/ap450.pdf, 2007 [Jan. 2017].*

[17] *J. B. Postel, "Transmission Control Protocol," NIC- RFC 793, Information Sciences Institute, Sept. 1981.*

[18] *S. Dharmapurikar and V. Paxson. "Robust TCP stream reassembly in presence of adversaries," In USENIX Security Symposium, Aug. 2005.*

[19] *Microsoft TchNet. NTTTCP, last update. Ver 5.28,"* *http://gallery.technet.microsoft.com/NTttcp-Version-528-Now-f8b12769, 2008 [Jan. 2017].*

[20] *Wireshark, "network protocol analyzer,"*
http://www.wireshark.org [Feb. 2017].

[21] *D. Patterson and J. Hennessy. "Computer organization and design, 4th ed. The Hardware/Software Interface. Publisher: Morgan Kaufmann, 2009.*

[22] *K. Pagiamtzis and A. Sheikholeslami, "Content Addressable Memory (CAM) circuits and architectures: a tutorial and survey," IEEE Journal of SolidState Circuits, vol. 41, 2006, pp. 712–727.*

[23] *J. Mogul and S. Deering, "Path MTU discovery", RFC 1191, Nov. 1990.*