Design a Scalable Ethernet Network Interface Supporting the Large Receive Offload

For high communication rate up to 100 Gbps

Mohamed Elbeshti[1]
School of IT, Murdoch Uni
Perth, Australia
m.elbeshti@murdoch.edu.au

Mike Dixon
School of IT, Murdoch Uni
Perth, Australia
Mike.dixon@murdoch.edu.au

Terry Koziniec
School of IT, Murdoch Uni
Perth, Australia
terry.koziniec@murdoch.edu.au

Abstract— **The Ethernet speed has increased to 40-100 Gbps since the release of IEEE P802.3ba. In this paper, we have enhanced the Intel's Large Receive Offload Linux software driver function to manage the out-of-order packets and designed a scalable Network Interface based RISC core to support this function in the Network Interface. The RISC's performance and data movements for high communication rates up to 100 Gbps have been measured, and the results presented herein show that a cost-effective embedded RISC core can provide the required efficiency of the network interface to support a wide range of transmission line speeds, up to 100 Gbps. Furthermore, we have found several techniques that can contribute to packet processing and work with fewer headers and less data copying in a host memory.**

Keywords; *LRO; RISC core; TCP/IP; VHDL simulator; Cycle-accurate performance evaluations;Network Interface.*

I. INTRODUCTION

The early stage of development of high-speed network cards used off-the-shelf components and integrated these components on a Printed Circuit Board for the implementation of the network card. It is possible to integrate all the discrete components needed for an Network Interface NI in a single chip [5, 7] in ASIC. Moreover, specialized engines from the sequential machines and discreet logic to address the need for a protocol, reducing the cost of design and improving reliability and performance [7]. Using ASIC to design NI could provide a greater energy efficiency and better integration than the programmable-based. However, ASIC also limits flexibility, limits upgradability, and makes NI design tailoring difficult in changing the algorithm of the protocol or supporting a new version of protocols.

Network interface designers avoid the use of general-purpose processor (GPP) cores as an integral part of their ASICs. Different reasons lie behind such a decision, such as specialized cores being able to run at higher clock rates than the embedded processors. Furthermore, a GPP occupies a large space on the ASIC Chip. In addition, a commercial license is required to be used for GPPs as processing cores in the ASICs.

However, the Hardware Description Languages (HDL) have made possible the design of embedded processors for the performance of certain functions. Furthermore, it is possible to adapt these processors in order to comply with the functions for which they are designed. System-on-chip technology has enhanced the possibility of integrating the hardware blocks required in the NI and the GPP to be carried on one chip [6].

Many cost effective embedded cores have become available and can be ported to an ENI chip. These advances have directed this research in investigating the use of specialized RISC embedded cores in the high-speed scalable NI design.

In designing the NI, we have avoided using multiprocessing cores as processing cores at the NIC to serve a single function, such as the ones developed at Rice University and Purdue University which have proposed strengthening the network card with six processors to enable it to perform the 10 Gbps [4, 12]. The idea behind the multiprocessor is to divide the processing required for each incoming or outgoing packet. A 166 MHz controller with six processors can achieve 99 percent of theoretical throughput of 10 Gbps. The designers Hyong, Vijay and Scott developed the Tigo programmable NI, which was released in 1997 [11]. This depends on two 88 MHz MIPS R4000-based processors for the completion of data processing. Both processors perform either inbound or outbound functions. Despite these approaches achieving the goal of these networks, there are several concerns. The complexity of these proposed models is in the sharing of the main resources in the NI between the embedded processors. In this case of parallelization, Amdahl's law [16, 18] explains the in-depth processing inside a design. The implementation program is usually divided into two portions: the first, part "P", is the amount of the protocol processing program that can be made parallel, and the "1- P" is the other portion of the processing that cannot be parallelized; it remains serial. The maximum achievement on the NIC by using the N processors is:

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

Assuming the P is 90%, Then 1- P = 10 %. With this high assumption of parallelized code (90%), the problem is sped up by a maximum of a factor of 10, no matter how large the value of *N* used. Accordingly, using a number of processors to support the LRO can be accomplished only if this design has extraordinarily high values of *P*: This is known as the embarrassingly parallel problem. The complication could increase when LRO Software migrations would most likely

start from serial code bases. Therefore, the target software design needs to identify the solution to meet the migration requirements. Furthermore, the programming model should be Symmetric Multiprocessing (SMP) or Asymmetric Multiprocessing (AMP). However, CPU intensive code for parallel processing using SMP is difficult to redesign.

In this paper aims to investigate the possibility of improving the structure of the NI by placing a specialized single core unit for supporting the processing requirements for LRO at a high speed communication up to 100 Gbps. Offloading the LRO to NI could reduce the amount of data transfer over the system bus, since collecting messages from the same stream in to form large packet in the NI may reduce the number of overhead data (the network headers). In addition, less DMA initiation to move the data from the NI's buffer to host memory. We have designed a single-issue RISC core supported with three pipeline stages and a forward engine to process the LRO functions. Since the receiving-side and the sending-side operations are completely independent, the NI is designed to handle both operations in parallel. In this work, we have focused on the receiving side only for TCP/IP. The wide use of TCP applications over the Ethernet amounts to roughly 82% of the protocol usage [17] directing this research to evaluate to evaluate these protocols. Other types of protocols can also be studied and evaluated within this model. The target scalable NI, therefore, can be used as an open guide for protocol processing and future use.

The rest of this paper is organised as follows: Section 2 discusses the receive side processing. In section 3, the structure of the receive ENI model is discussed. The processing anaylsis is discussed in section 4. The VHDL-based simulation (behavior model) results are examined in section 5. The core design will be highlighted in section 6, followed by the conclusion.

II. LARGE RECIVE OFFLOAD PROCESSING

During the last decade, there have been improvements to network processing. There are a number of successful achievements in reducing the protocol overhead to support the server for the Gig bit communication networks and improve the system's I/O.

Offloading the protocol processing to NI has been used as a method for reducing the protocol overhead, such as the TCP Offload Engine (TOE) [8]. The TOE style then gives a wider field to a host processor to perform other services than performing the network protocol. However, offloading all the TCP functions processing from a server's CPU to an NI helps the server to focus on processing application requests rather than the network protocols. Nevertheless, shifting all the TCP/IP functions to the NIC is a more complicated situation and involves a significant change in the Operating System. In order to support these changes in the high-speed rates a number of issues are required to be addressed, such as security, Moore's law, performance, flexibility and costing. Mogul [3] describes these problems with analysis and evaluations. In addition, Mogul considers the TOE to be an impractical design for the high-speed network.

The proposal for accelerating the stack processing is obtained through the improvement of the system's I/O performance by increasing the size of the Ethernet frame size to 9600 bytes (the jumbo frame). This certainly improves the performance of the end nodes and reduces the overhead. However, this approach has not been universally deployed. The reason is that the increasing payload size can have a few negative impacts on the Gigabit networks. Firstly, when the large packet size is greater than the rate at which the router can deliver to its destination, the router divides into a smaller number of packets compatible with the port line that will deliver. In this case, there is additional information added to each package in order to facilitate the assembly process. However, most of the NICs are still supporting the MTU of 1500 bytes packets. Secondly, the re-sending of the entire packet could happen when the loss of one of the small fragments of the jumbo frame occurs. Thirdly, this approach imposes an additional burden on the processing of routers to deliver the packets, thereby consuming more overhead in the network. Furthermore, passed fragments may filter them due to firewall configuration, since they did not carry the TCP header.

The receiving side, however, is not exposed directly to operate within the NIC. This is not only because of the potential for out-of-order packets, where some of the NICs drop the out-of-order packets [2], but also due to the receive complexity, such as supporting different links.

Another strategy is the integration of the idea of the jumbo frame (9600 bytes) with the offload approach for the improvement of protocol processing. One of these research studies has suggested the use of the network stack processing technique called Large Receive Offload (LRO) on the host side [13, 14]. LRO is a software driver in the Linux platform. Intel has provided it to reduce the number of the arriving TCP/IP packets. LRO combines the same stream packets to be formed as large-sized packets inside a host memory (Socket buffer (SKB)). The LRO functions by generating SKBs only for the first packet of an LRO session. The following fragments will be added in the fragment list of that SKB. The LRO stores the packet as is into a separate SKB if they do not match the LRO requirements, and then passes it to the network stack for further processing.

void lro_receive_frags*(struct net_lro_mgr *lro_mgr,struct skb_frag_struct *frags,int len, int true_size, void *priv);*
void lro_vlan_hwaccel_receive_frags*(struct net_lro_mgr *lro_mgr,struct skb_frag_struct *frags,int len, int true_size, struct vlan_group *vgrp,u16 vlan_tag,void*priv);*

As a result, of these functions, the assembled group of the TCP packets that are related to a single packet inside the host memory is performed. However, while this approach benefits the receiving-side, but the host CPU spends a number of cycles to run the virtual LRO. In addition, the host CPU requires processing of the small packets that do not match the LRO's criteria, such as the out-of-order packets. This is an extra burden to the host CPU [15]. The CPU instructions are required to serve the LRO for a guest domain, which is around 1600 instructions, when the packet size is MTU 1500 bytes. The previous measurements for the Large Receive Offload Linux software found [15] the LRO costs 2927 CPU cycles, which is caused by the use of the LRO software optimization. In 10

Gbps, for instance, the end node could get around 812743.82 packets per second when a MTU (1500 bytes) [1]. Therefore, a host CPU requires 2927 cycles to be completed.

Determine the actual MIPS rate, and execution time for LRO program when;

- The communication line speed is 10 Gbps and, packet is 1500 bytes.
- The CPU clock speed is 4 Gbps

Then ; CPI = CPU Clock Cycles/Instruction count
= 2972/1600 = 1.85

Calculate execution time

Execution Time = Instruction Count * CPI / Clock Rate
= $2972 * 1.85 / 4 \times 10^{9}$
= 1.3×10^{-6} sec (1)

MIPS = clock frequency/(CPI*1000000)
= $(4 \times 10^{9})/(1.85*1000000)$
= 2153 MIPS (2)

This estimated calculation (2) shows that over 50% of the CPU power (4 GHz) is required to operate the LRO code while performing the LRO. The CPU power eventually increases when the number of incoming packets increases, especially when the packets become smaller than the MTU [19]. The host CPU devotes more cycles to complete the LRO programmers than other services. This increase of CPU usage has a side effect on the node performance. The budget time for supporting each packet is 1.3 ms (1). The MTU message needs an extremely short time, about 123.04 ns, when the line speed rate is 100 Gbps.

III. NI MODEL FOR RECEIVING SIDE

We have structured the proposed NI into three parts: communication Line Interface (LI), kernel processing, and Host Interface (HI) ("Figure 1"). The HI and LI are implemented in hardware. The processing unit in the NI, which commonly processes functions that are related to header processing, is embedded specialized RISC.

When packets arrive from the MAC layer into the RBI, the FSM in the RBI will enable one of the two buffer locations to accommodate the serial bits which have arrived from the transmission line. An FSM will enable one buffer and can switch to another buffer after interrupting the RISC-core on the receiving-side.

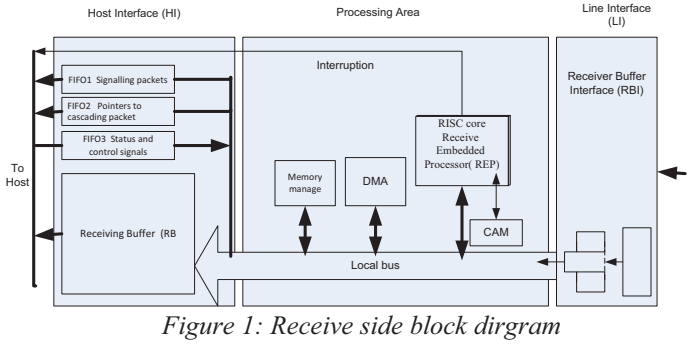

Figure 1: Receive side block dirgram

The RISC core will begin processing the packet headers which are located at the top of its body. Such implementations will support the NI 's processor at about 123.04 ns when the line rate is 100 Gbps, and the packet size is MTU (1500 bytes)[1]. The receiving side has another resources to support the RISC core, such as the use of the Content Addressable Memory (CAM) to accommodate the active connections for TCP and DMA. The NI communicates with the host through three FIFO buffers. These FIFOs were implemented as memory-based. The pointer of each FIFO is stored in the RISC's register. The RISC core reaches any FIFO by reading its address. Since the receiving-side is entirely independent of the sending-side, the REP, DMA and other devices share a single bus to operate the LRO.

The premise states that TCP network is connection-oriented. In TCP, two ends start opining the connection oriented communication. A connection is identified by Connection Identifiers (CIDs), the IP address and the virtual Port IDs (PID). The packets that transmitted over a signed connection should have the same identifiers. The RISC at the receiving-side can identify all the arrival packets that have the same identifier, which is related to the same TCP message. The RISC coalesces to form one packet. This is done by combining the incoming packets that have same link information, such as the IP address and port ID, into large-sized packets at the end node before they are presented to the TCP stack. Combining the arrived packets is also possible because TCP is a byte stream protocol. When the application passes a message down to lower layers, it cannot make any assumptions about the boundaries of the message. This means that within the same "receive call" an application can get the tail (end) of a message and the front (start) part of the next message. Therefore, formation of large data packets would be acceptable to most operating systems, because they do not require any modification in the composition of the operating system or change in the work of the protocol inside the host area.

In this case, part of the RISC core processor is multiplexing the arrival packets to its related message according to the packet's identifier. The CAM is used to hold 64-entry of the CIDs (IP address 32-bit and PIDs 32-bit). Such a size is adequate for the NI to establish a 64 active connection simultaneously during each IM window (after every 20 packets [9]). These CAM entries support the work of the linked-list in tracking the sequence of the arrived packets in order to link them to a related list. The CAM will flash after every 20 packets (the IM windows). The RB memory is divided into 64 pages. Each page can accommodate up to 30 KB (20 packets each with MTU). In the case that the entire stream is related to one stream, then it will coalesce into one page.

B. Linked-list design format.

After inserting a new entry in the CAM, all link has a start-address (pointer to the location of the packet) and end-address (the last packet arrived of this stream). These pointers are reduced to zeros. Changing these pointers depends on the processing that the entry packet needs. Each arrival packet may require different processing within the linked list, depending on its CIDs.

As soon as the TCP packet arrives at the NI, the IP header and TCP header will be processed. The IP address and the PIDs have been masked from the IP header and TCP header, respectively, in order to match these identifiers with the CAM.

After the match has been found, the payload needs to join the same data packet into RB. In order to manage the RB accurately, a Memory Management has added, which form as a Circulation Buffer (CB) to hold all the free pointers inside the RB. REP reads the head of the CB in order to get the free address location inside the RB. Sequence Number (SN) in the TCP header [10] contributes to the REP to identify the packets type to manage the linked-list. For example, the packet is Beginning of Message (BOM), which is the first packet arrived in a stream. This means there is no linked-list assigned before to this stream. The REP needs to create a new linked-list for this packet by inserting the Start-address and End-address in the CAM beside CID "Figure 2". The Start-address refers to the head of the linked-list (the address that is loaded from CB for this packet). The End-address refers to the tail of the linked-list which is the Null's address (Node's pointer), located at the end of the packet body. Thus, the linked-list with one node has been created for the arrived TCP/IP. Continuation-of-Messages (COM) are the packets that have the same connection identifier arriving after the BOM. After the packet's IDs matches the one in the CAM, the REP starts adding a new node to the existing linked-list (packet body and its pointer). The linked-list updates after adding a new node by making the current node pointer point to NULL (end-address). Next, the REP stores the NULL address of the current node at the CAM referring to the new end in the list. End-of-Message (EOM) is the last packet, when the PSH flag (inside the TCP header) is assigned and then appends the EOM packet to related streams inside the RB and deletes the linked-list of this stream. Users can refer to reference [10] or details about the TCP. The REP then initiates the DMA to transfer the packet body only from the RBI to the RB buffer to be linked with the previous amalgamated data of this stream inside the RB. The End-address in the CAM refers to Null value of the previous packet. Then it stores the address of the current packet in the same place where the NULL value (the End-address) of the previous location is. The REP is responsible for updating the SN and AKN inside the TCP header of the original TCP/IP of the large packets inside the RB. Furthermore, it needs to modify the length of the datagram by the total of the amalgamated bytes of the stream. When the PSH flag is "1", there is no liked-list assigned to this stream (start and end address == 0), which then moves the packet from RBI to the RB buffer. REP completes this packet as SSM. REP has nothing else storing the NULL value at the end of the packet body. Furthermore, there is no need to update the End-address or Start-address at CAM because no more packets will be amalgamated with this packet.

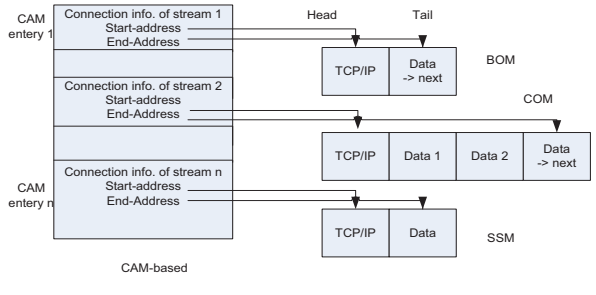

Figure 2: Linked list data structure.

C. Out of order processing

Out-of-Order (OOP) strategy is more complicated processing than BOM, COM or SSM packets. To prove this point, one can cite the example of the LSO that prefers to stop the coalescing packets when an out-of-order is distinguished. Additionally, it opens a new queue to hold the OOP packets. The processor is required to perform a number of steps (known as call netif_rx(), or netif_recive_skb()) which, if successful, appends the packet to related streams. These steps lead to the additional burden on the host CPU cycles and waste of memory size. The order-dependent is considered to having in-order-processing (IOP) architectures. A novel method have introduced for OOP processing architecture. The OOP starts after REP masks the SN of the arrived packet, and then it is compared with the SN expected to be reached at this linked-list. In the next step, the REP joins the arrived packet into the linked-list. In the case that the SN is not located between the rates of the target stream, then the REP creates a sub linked-list of this stream "Figure 3". A duplicate data segment can also be discovered after checking the SN of the TCP stream that has already been amalgamated before in the RB. Furthermore, the REP discards any of the lost packets.

The host CPU recognizes a sequence gap, or hole, which occurs in the TCP stream with the arrival packets. Such a gap can be treated by sending an acknowledgement to a sender to re-send that packet loss [10, 20]. The "window scale" is 16 bit only on the TCP header, which is providing 65k bytes (2^16). With the LRO, the combined effect is to recover from one packet loss per TCP widow, without changing the concept of the window stream. The host can discover the holes immediately after processing the stream of the larger packets. The host CPU requests a new packet for the missing sequence space to be re-sent. The REP deals with such messages as the SSM and sends them as is to the RB.

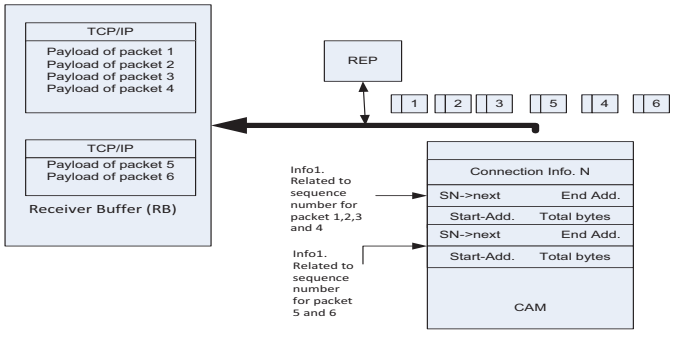

Figure 3: CAM structure

IV. PROCECESSING ANALYSIS

We started the model simulation by delivering different packets to the receiving-side that can keep the RISC busy while the DMA transfer cycle is in operation. The NI's cycle-accuracy has shown that RISC needs 15 instructions to complete identifying the BOM of the TCP. Avoiding the conflict of using the local bus, we have assigned instructions to the RISC that do not require the use of the local bus, such as checking the status of the current packet of the BOM "Figure 4". If the PSH flag sets to "1" [10] and there is no linked-list assigned to this stream, then the RISC treats this packet as the SSM; elsewhere, the RISC processes this packet as the EOM.

If the packet is EOM, the RISC core then executes two instructions to update the TCP and IP header inside the RB and one cycle to send the start-add to FIFO 2. The RISC core has to wait until the local bus is released by the DMA controller in order to complete the EOM processing.

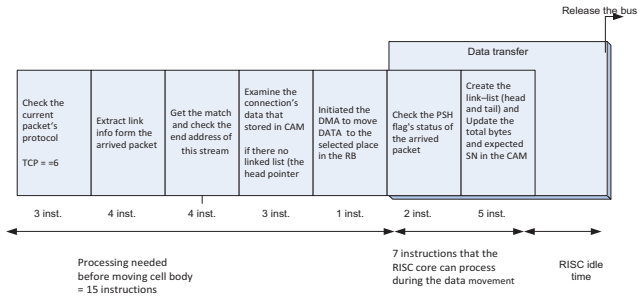

Figure 4: Total instruction for BOM of TCP is 22 instructions

D. Data moments

The TCP payloads vary from 6 to 1640 bytes [1]. The DMA required moving the Maximum Segment Size (MSS) from the SBI to RB is 366 cycles (183 cycles to read payload data over the 64-bit bus to the DMA's data register and 183 cycles to store it to RB). Clearly, the RISC core will be in idle mode until the DMA completes moving the data. The RISC can execute 7 instructions during the data moments and becomes idle with MSS at about 359 instructions. The idle cycle's time affects the performance of the network card and its capabilities to deal with high speed networks.

Reducing the idle instruction of the RISC core have studied. One of these solutions is the use of a multi-bus based on the receiving-side. The RISC can access the multiport memories while the DMA controller moves data.

The second scenario is to place data into the RB first, instead of the RBI, and then combine the message with the previous one.

The other approach is to use a DMA that runs at a higher clock rate than the RISC. We have adapted the way of using it that we presented in "Figure 1", since it is a straightforward scenario and easily implemented without any changes in the NI's architecture. We have started adjusting the DMA's clock to reduce the idle cycles. During the implementation, the DMA's clock speed is changed to reduce the idle cycles and to improve the performance of NI.

Small size packets, such as 64 until 256 bytes, may require less DMA cycles than other packets that have more payload bytes. However, while using these small size packets can improve the NI's performance, it affects the end node's throughput [19]. We have focused on the 512 bytes packet to MTU packets (1500 bytes). The use of small packets can be studied on this model, but they bear little payload data and may not be able to achieve 100Gbps.

To eliminate the RISC's idle cycles, we forced the DMA to complete its processing cycle in a shorter time than the first approach, where the DMA has the same as the RISC's clock cycle. Therefore, the DMA clock speed is increased to run faster than the RISC to enable the local bus to be available for both the DMA and RISC core.

The simulation results demonstrated that a RISC-based NI is scalable for a transmission line with a speed up to 100 Gbps. To reduce the design complexity, we presented a simple data path of the NI.

time we increased the DMA's clock to reduce the idle cycles, we noticed that the RISC core and DMA controller were working quickly to complete each message and transfer it to the RB.

During the behavior model analysis found when the DMA's clock becomes five times faster the embedded processor core, the NI performance increased significantly, where most, if not all, the idle cycles were reduced Table I. Table I presents the total RISC instructions need to complete TCP packets. It is clear that the processing rate increased when the packet gets smaller and the DMA have to work at high speed to finish data movements Figure 5. This is natural because the number of messages that the NI receives is less than in the case of 512, which is only 81274382 packets per second when the packet size is MTU [1]. The 1500 bytes, 1024 bytes and 512 bytes are applied to That is the reson of chosing

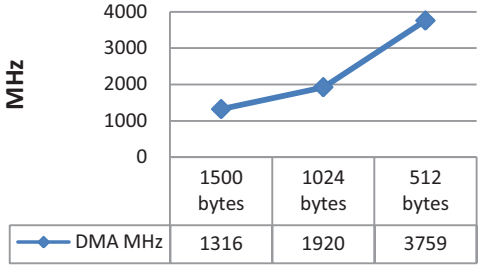

Figure 5: The DMA clock while transferring the TCP packets

TABLE I : TOTAL RISC INSTRUCTION AND IDLE CYCLES (WHEN THE MA HAS FIVE THAN THE RISC'S CLOCK RAT)

Packet Type	Packet Size					
	Total number of instruction					
	1500 bytes		1024 bytes		**512 bytes**	
	Total RISC Inst.	Idle Inst.	Total RISC Inst.	Idle Inst.	Total RISC Inst.	Idle Inst.
Single Segment Message	52	35	40	23	27	10
Beginning Of Message	52	30	40	18	27	5
Continuation Of Message	56	27	44	15	31	2
End Of Message	56	28	44	16	31	3
Out-Of-Order	59	25	47	13	32	0

We fixed the speed of the DMA to 3759 MHz on all type of packet sizes. The idle cycles associated with the RISC core processing have been eliminated (Table II). During the simulation, we have monitored the clock rate of the RISC. The highest RISC clock rate is recorded high, when it is performing the out-of-order of the TCP packets. The results have shown that a 751 MHz RISC processor can support the LRO function at a 100 Gbps lines, when the DMA speed is 3759 MHz, and the packet size is 512 bytes: "Figure 6". A RISC core with 263 MHz can be used to process the LRO at 100 Gbps when the packet size reached 1500 bytes.

TABLE II: TOTAL RISC INSTRUCTION WHEN DMA CLOCK IS 3759 MHZ

Packet Type	Packet Size					
	1500 bytes		**1024 bytes**		**512 bytes**	
	Total RISC Inst.	Idle Inst.	Total RISC Inst.	Idle Inst.	Total RISC Inst.	Idle Inst .
Beginning Of Message	25	3	22	0	27	5
Continuation Of Message	29	0	29	0	31	2
End Of Message	31	3	28	0	31	3
Single Segment Message	25	8	22	5	27	10
Out-Of-Order	32	0	32	0	32	0

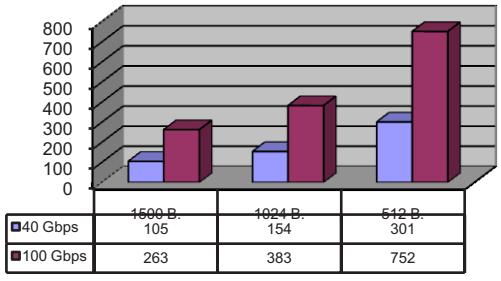

Figure 6: LRO for TCP/IP when DMA has five than the RISC's clock rate

VI. RISC CORE

This simplicity helps the RISC cores to manage and process the LRO at a low clock rate. Further, it has made it possible to reduce the cost of development of RISC-based NIs. Such NIs can be flexible enough to support protocol changes or can even adapt new protocols, whereas the customized logic-based NIs can only support certain functions for a single design.

Designing a RISC core for specialized application, namely NI control and data path, is simpler than using the off-the-shelf GPP processors. These general-purpose embedded processors are not optimized for an LRO function. Hence, some portions of GPP instructions that support general-purpose applications may not be required for the ENI design. For example, the Floating-Point Unit is not necessary for network interfaces. Furthermore, we found that using a data cache to store data is not required since it will not enhance the NI's performance or reduce the RISC clock's speed for this application. The elimination of these units in the design of the core simplifies the process of the development of NI and reduces the size and cost.

We have noticed that the RISC performs a few of the instructions to complete processing the LRO. These instructions are load, store, arithmetic and logic operations and conditional branches. Furthermore, we measured the total percentage of each of these instructions that the RISC core is required to perform the LRO: "Table II". It is hence concluded that a minimum instructions set can be used with the core, which would make the control unit design very simple and fast. In addition, the limited number of instructions that are required to support the Ethernet interface processing can reduce the size and complexity of the control unit leading to an increased speed.

TABLE II. THE INSTRUCTIONS PRESENTAGE USED FOR THE LRO FUNCTIONS

Operation type	TCP/IP Processing
Load	39.28 %
Store	17.8 %
Arithmetic and logic operation	28.27 %
Conditional branch	14.28 %
Reading/writing from/to RB	26.9 %
The LRO data structure	40.81 %

VII. CONCLUSION

We have presented computer simulation results to measure the amount of processing required for LRO functions for TCP/IP. The simulation results have shown that a cost-effective embedded RISC core can provide the required efficiency of the network interface to support a wide range of transmission line speeds, up to 100 Gbps. A 263 MHz RISC core can support the receiver side processing for up to 100 Gbps transmission speed for TCP/IP when the packet size is MTU bytes, while a core running at 752 MHz is found to support the 512 bytes. A fast DMA (3759 MHz) is required to eliminate the RISC idle cycles. This research could also provide some improvements over other methods that have been applied in the on-load methods. For example, the use of the zero-copy, RDMA or Direct Cache Access (DCA) with this LRO approach could contribute to the packet processing. These applications will work with fewer headers and data copy. For DCA, a few headers are will be copied into the host CPU's cache instead of a large number of small headers. Besides that, there will be less data calls to copy data in a host memory. This could increase the processing need by the Zero-copy function.

REFERENCES

[1]. G. Held "Ethernet Networks (4th ed.)," Design, Implantation, Operation and Management. John Wiley publisher LTD, 2003.

[2]. D. Schuehler and John Lockwood. TCP-splitter: A TCP/IP flow monitor in reconfigurable hardware. In Hot Interconnects-10, August 2002.

[3]. J. Mogul, "TCP Offload Is a Dumb Idea Whose Time Has Come," *Proc. 9th Workshop on Hot Topics in Operating Systems* (HotOS IX), Usenix Assoc., 2003.

[4]. P. Willmann, H. Kim, S. Rixner and V. Pai," An Efficient Programmable 10 Gigabit Ethernet Network Interface Card," Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture November 2005.

[5]. O. Elkeelany, On chip novel video streaming system for bi-network multicasting protocols, Integration, the VLSI Journal, v.42 n.3, p.356-366, June, 2009.

[6]. C. Cranor *et al* .Architecture considerations for CPU and network interface integration IEEE Micro, January–February (2000), pp. 18–26.

[7]. Y. Hoskote et al. A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 38(11):1866–1875, Nov. 2003.

[8]. A. Earls, "*TCP Offload Engines Finally Arrive,*" Storage Magazine, March 2002.

[9]. Interrupt Moderation Using Intel® GbE Controllers 2007. *download.intel.com/design/network/applnots/ap450.pdf.*

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

[11]. H. Kim, V. S. Pai and S.Rixner, **"** Exploiting task-level concurrency in a programmable network interface," Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp 61-72, 2003.

[12]. D. Schuff and S. Pai," Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface," Parallel and Distributed Processing Symposium, IEEE International, pp 1 – 10, 2007.

[13]. Leonid Grossman. Large Receive Offload implementation in Neterion 10GbE Ethernet driver. In *Ottawa Linux Symposium (OLS)*, 2005.

[14]. A. Menon and W. Zwaenepoel. Optimizing TCP receive performance. In USENIX Annual Technical Conference, June 2008.

[15]. K. Kumar Ram , J. Renato Santos , Y. Turner , Alan L. Cox , S. Rixner. "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 11-13, 2009, Washington, DC, USA.

[16]. Amdahl, G.M., "Validity of the single-processor approach to achieving large scale.
computing capabilities," Proceedings of AFIPS Conference, 1967, pp. 483-485.

[17]. M. Allman and A. Falk," on the effective evaluation of TCP," ACM SIGCOMM Comput. Rev., vol 29, no 5, pp 59-70, Oct 1999.

[18]. M. Hill and M. Marty, "Amdahl's law in the multicore era," IEEE Comput., vol. 41, no. 7, pp. 33–38, Jul. 2008.

[19] 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, December 2003.

[20] Sarang Dharmapurikar and Vern Paxson. Robust TCP stream reassembly in presence of adversaries. In USENIX Security Symposium, August 2005.