2019 29th International Conference on Field Programmable Logic and Applications (FPL)

Limago: an FPGA-based Open-source 100 GbE TCP/IP Stack

Mario Ruiz*, David Sidler† Gustavo Sutter*, Gustavo Alonso† and Sergio López-Buedo*‡
* High Performance Computing and Networking Research Group,
Escuela Politécnica Superior, Universidad Autónoma de Madrid, Spain
{mario.ruiz, gustavo.sutter, sergio.lopez-buedo}@uam.es
† Systems Group, Department of Computer Science, ETH Zürich, Switzerland
{dasidler, alonso}@inf.ethz.ch
‡ NAUDIT HPCN, Spain; sergio@naudit.es

***Abstract*—The realization that the network is becoming an important bottleneck in computing clusters and in the cloud has led in the past years to an increase scrutiny of how networking functionality is deployed. From TCP Offload Engines (TOEs) to Software Defined Networking (SDN), including Smart NICs and In-Network Data Processing, a wide range of approaches are currently being explored to increase the efficiency of networks and tailor its functionality to the actual needs of the application at hand. To address the need for an open and customizable networking stack, in this paper we introduce Limago, an FPGA-based open-source implementation of a TCP/IP stack operating at 100 Gbit/s. To our knowledge, Limago provides the first complete description of an FPGA-based TCP/IP stack at these speeds, thereby illustrating the bottlenecks that must be addressed, proposing several innovative designs to reach the necessary throughput, and showing how to incorporate advanced protocol features into the design. As an example, Limago supports the TCP *Window Scale* option, addressing the *Long Fat Pipe* issue. Limago not only enables 100 Gbit/s Ethernet links in an open source package, but also paves the way to programmable and fully customizable NICs based on FPGAs.**

I. Introduction

The growing amount of data and the complexity of the workloads that characterize modern distributed computing have turned the network into a potential bottleneck [1]. Besides, in cloud environments, the network also limits the number of virtualized/containerized applications that can be deployed on a single server: The more CPU cycles needed to deal with an increasingly complex networking stack, which needs to provide not only TCP/IP packet processing but additional functionality such as Network Function Virtualization (NFV) or Remote Direct Memory Access (RDMA), the less CPU cycles that are available to applications. In addition, the trend towards specialization seen in cloud computing opens up the possibility of tailored network designs through Smart Network Interface Cards (NICs), which push application-level processing to the network [2]. As a result, we are witnessing a flurry of activity around programmable networks based on a variety of designs and architectures.

A concrete example of these developments is provided by Microsoft Catapult [3], a deployment of FPGAs in the cloud that has evolved through several generations [4], [5]. The current version inserts an FPGA on the data path between the top of rack (ToR) switch and the server machine. Hence, all network traffic in and out of the host goes through the FPGA. The FPGA is then used to augment the network functionality with system and application-level features. For instance, it can be used as a customizable smartNIC to offload network virtualization functionality [5], application-level functionality such as RDMA packet processing to support key-value stores [6], or for distributed machine learning algorithms [7]. Catapult is, by far, not the only possible design. In IBM's cloudFPGA [8], the FPGA is deployed as a network-attached accelerator. Similarly, Caribou [9] deploys FPGAs as storage nodes that extend the TCP/IP stack with distributed consensus functionality (a network function) [10] as well as scans and string processing (application-level functionality) [9].

Promising as they are, for FPGA-based designs a challenge remains: Scalability with increasing network bandwidth. To address this challenge, in this paper we introduce Limago, an open-source 100 Gbit/s TCP/IP network stack on an FPGA. Limago explores the changes needed to upgrade an existing open-source TCP/IP stack from 10 Gbit/s [11] to 100 Gbit/s, but maintaining the same high-productivity design methodology, based on Vivado-HLS, that was utilized in the previous design. In doing so, Limago illustrates how to tackle the problem of FPGA-based packet processing at such rates. From the existing design, Limago inherits the scalability in terms of the number of connections as well as the control flow and congestion avoidance functionality. Limago not only transforms and adapts these existing features to increase the supported bandwidth from 10 Gbit/s to 100 Gbit/s, but also contributes novel features widening its applicability. The changes are non-trivial extensions of the existing stack. For instance, the data path had to be widened by 8x and the operating frequency doubled to reach the target bandwidth, several low-level architectural changes and balanced pipeline stages were necessary to meet timing, and accessory modules were redesigned. Limago also incorporates functionality such as the TCP *Window Scale* option, an extension to the basic TCP/IP protocol, addressing the *Long Fat Pipe* issue.

Limago serves as a platform for further research in programmable networking and as a design guideline on how to tackle high network bandwidths with FPGA-based systems.

1946-1488/19/$31.00 ©2019 IEEE
DOI 10.1109/FPL.2019.00053

II. Challenges at 100 Gbit/s

Limago uses the 322 MHz clock provided by the integrated 100G CMAC, and a 512-bit AXI4-Stream interface. With respect to the 10 Gbit/s version, that is an 8x increase in the width of the datapath and more than a 2x increase in the operating frequency. Moreover, the smallest packet (64-Byte) just fits into a single transaction and, for such short packets, the processing rate must be 148.8 million packets per second. The greater data rate implies novel designs for several components often taken for granted. For instance, existing SmartCAM designs, used for flow identification, do not operate at such frequency and a new solution is thus needed. Similarly, certain optimizations are optional at lower rates, but a must at such bandwidth. For instance, the *Long Fat Pipe* issue might not be observable at 10 Gbit/s but must be addressed to reach 100 Gbit/s. This requires additional circuitry to support and negotiate the TCP Window Scale option.

A. TCP/IP Checksum

Checksum computations are widely used when processing TCP/IP packets. Limago uses an efficient implementation, leveraging 7 to 3 Carry Save Adder (CSA) circuits [12] to calculate the checksum within one clock cycle. The module was written in HDL to achieve a low latency in this recurrent circuit. Actually, this is one of the few modules of Limago written in HDL; the vast majority of blocks are written in Vivado-HLS. But in this case, an efficient and low latency implementation was needed, impossible to achieve with the Vivado-HLS version being used (2018.2). The circuit is described in detail in [13].

B. CuckooCAM

The 10 Gbit/s version of the stack is based on the smartCAM [14] module provided by Xilinx. It used a four-tuple consisting of IP source and destination addresses plus TCP source and destination ports as a key. We replaced this module with our own implementation, CuckooCAM, based on cuckoo hashing and requiring one clock cycle for lookup and deletion. In CuckooCAM, insertion time depends on the load factor and occupancy can exceed 90% due to a secondary memory structure known as a stash. It is clocked at 322 MHz, providing more than 300 million lookups per second. The width of the key and value are configurable; therefore, we have reduced the size of the key to a three-tuple by removing Limago's own IP address, which does not change during operation. The reduction of the key from 96-bit to 64-bit results in a significant reduction in BRAM usage for this module (22%).

C. DRAM Memory Access

To support a large number of connections, the TOE uses external memory for its receive and send buffers. In particular, this is necessary for the send buffer, where the payload has to be stored until it is acknowledged. For 100 Gbit/s, the resulting requirements in terms of memory bandwidth are close to the peak bandwidth provided by DDR4. Additionally, the offsets into the receive and send buffer are determined by the TCP sequence number. This can result in unaligned memory accesses affecting the memory bandwidth further. Therefore, we verified the viability of storing the buffers in external DDR4-2400 through several microbenchmarks, varying the memory-alignment as well as the access size. We observed a peak bandwidth of 125 Gbit/s with 64-Byte aligned words and approximately a 6% performance loss when transfers were not aligned, thereby ensuring the design achieves enough memory bandwidth for all cases. Since the buffers in external memory are organized as a circular buffer, additional logic is required to handle the wrap-around when the "end" of the buffer is reached. Particularly, a single data transfer is split into two transfers (one before the wrap-around and one after), requiring data re-alignment. The HLS code for this module was redesigned carefully to guide the synthesis tool to the most efficient implementation involving a 64 to 1 multiplexer.

D. TCP Window Scale Option

Links with a large $bandwidth \times delay$ product suffer from the *Long Fat Pipe* issue: those links where the bandwidth $\times$ delay product is larger than the buffer size [15]. The Window Scale option is used to allocate any fix-size buffer in the range of 2^{16} to 2^{30} bytes, thereby leading to a better usage of links.

Currently, Window Scale is the only supported TCP option in Limago. Due to the lack of a standard TCP option layout, the parsing of options is done sequentially, one clock cycle each. Fortunately, the Window Scale option is only negotiated during the initial three-way handshake, *i.e.*, options are only parsed once in the lifetime of a connection. The Window Scale is set to the minimum value advertised by both endpoints. Support for the Window Scale option has to be enabled at synthesis.

Finally, the maximum number of connections depends on the external DRAM capacity and the Window Scale factor, as shown by Equation 1. $DRAM_b$ is the $log_2(DRAMSize)$ and WS_b is the $log_2(WindowScale)$. As an example, with 4 GB of DRAM and a Window Scale of 128, $2^{32-7-16} = 2^9 = 512$ concurrent connections can be supported.

$$\#conn = 2^{DRAM_b - WS_b - 16} \tag{1}$$

III. Related Work

The benefits of TCP/IP offloading are well-known [16]–[18]: reduced CPU utilization and bypassing of the Operating System. In a TOE, packet processing is moved to the NIC, whereas the control decision remains in the host. Nowadays, most NICs offer some degree of offloading. In this section, we focus on FPGA implementations of TCP/IP.

LDA technologies [19] offers an FPGA-based TOE. Their solution includes independent transmitter and receiver modules. For sixteen connections, 44 BRAMs and 2,704 LUTs are necessary. Published results for this TOE using Solarflare NICs are based on 10 Gbit/s connections. Chevin Technology [20] offers a 10/25 Gbit/s TCP/IP core, which can work both as client or server. It supports 1 to 256 simultaneous connections. The Tx and Rx buffers can be configured from

1 KiB to 1 GiB, implying Window Scale support. For sixteen connections, 5 BRAMs and 12,000 LUTs (plus the external buffer) are necessary. Enyx [21] offers an RTL TOE solution with up to 4,000 connections, but not further details about resource utilization are provided. They also have announced a 25 Gbit/s implementation [22]. Dini [23] offers a 10 Gbit/s solution where the FPGA is used as a NIC. The buffer size is configurable from 4 KiB to 64 KiB and it supports up to 128 connections per instantiated IP-Core and out-of-order packet delivery. Algo-Logic [24] supports full duplex rates up to 20 Gbit/s per instance, claiming more than 200 Gbit/s can be achieved with multiple instances. The design targets low latency applications such as high-frequency trading.

The authors in [25] presented a comparison of three 10 Gbit/s alternatives: a pure software TCP/IP stack, a software TOE with kernel-bypassing and a hardware TOE (Fraunhofer HHI 10 GbE TCP/IP) with kernel-bypassing, concluding that the hardware solution has less latency and a more deterministic behaviour. The work in [26] presents a complete TOE implementation supporting jumbo frames and configurable Maximum Segment Size (MSS) and timestamp. Only one connection is supported with a 90 ns latency for a 100- Byte packet. Their solution is compared against a commercial, one achieving better latency. The paper in [27] introduces a TCP implementation using XFSMs, which is claimed to be "code-once-port-everywhere". The implementation is tested over three different architectures, software, FPGA, and NS3 emulator, reaching similar results. Probably, the closest work to Limago is [28], an asymmetrical standalone TCP/IP implementation oriented to video-on-demand, which supports 20,480 connections working as a client and 2,048 connections working as a server. It also can send up to 40 Gbit/s but only receive up to 4 Gbit/s. The starting point for Limago is a 10 Gbit/s TOE written by Sidler *et al.* in C++ using Vivado-HLS [11], [29].

IV. Limago Architecture

Figure 1 shows Limago's main components. We use AXI4-Stream to interface with the application logic as well as with the network modules. Since CMAC exposes an LBUS interface, we added an adapter module that converts between AXI4-Stream and LBUS. **Rx and Tx checksum** and **CuckooCAM** are respectively presented in sections II-A and II-B.

Inbound Packet Handler: parses Ethernet and IPv4 headers of every incoming packet. If the packet matches the filter, the signal TDEST will carry a different identifier for each kind of packet. Then an AXI4-Stream Switch forwards the packet to the appropriated module. If the packet does not belong to one of the previous categories, it is dropped.

ARP module: when an ARP request arrives and the IP address matches, it generates an ARP reply packet. Its main function is to associate IP addresses with MAC (physical) addresses, which is done using a 256-element table. The ARP module also receives MAC address requests from the *Outbound Packet Handler*. If the entry is not present in the table, an ARP request packet will be generated, and a miss

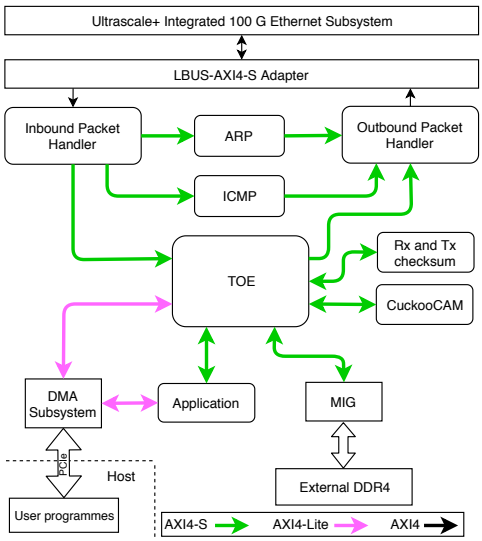

Fig. 1: General Architecture Overview.

will be reported. Additionally, an ARP request is generated at startup to notify other network devices in the same LAN.

ICMP module: provides responses to echo request packets, *a.k.a.*, ping. The module is useful to verify connectivity and gives a fair estimation of the Round-Trip delay Time (RTT).

Memory Interface: is composed of a Data Mover and Memory Interface Generator (MIG), both Xilinx IP-Cores. The MIG exposes a 512-bit AXI4 memory mapped interface and communicates with the off-chip DDR4 memory. The Data Mover is in charge of merging data and commands, which are produced in a streaming fashion, to an AXI4 interface.

Outbound Packet Handler: gathers packets coming from ARP, ICMP and TOE modules. If needed, a MAC address lookup, consisting of the IP destination address, is issued to the *ARP module*. If the lookup is a hit, the Ethernet header is constructed using the returned MAC address, prepended to the packet, and transmitted. Otherwise, the packet is dropped and an ARP request is generated instead. Moreover, the packet size is evaluated and padded to 60- Byte if needed.

DMA subsystem: we use the DMA for PCI Express (PCIe) Subsystem Xilinx IP-Core for providing users access to memory mapped registers within the logic. Limago uses the Xilinx's drivers both for debugging and communication. The necessary customization is built on top of them.

V. TOE Architecture

This section describes the overall architecture of the TOE (Figure 2). It is divided into three parts, the incoming data path (*Rx Engine*), the outgoing data path (*Tx Engine*), and the state-keeping data structures [11], [29]. The dash boxes are optional modules that can be enabled at synthesis.

A. Rx Engine

Incoming packets are processed by the *Rx Engine*. To verify the checksum, first the TCP pseudo header is constructed

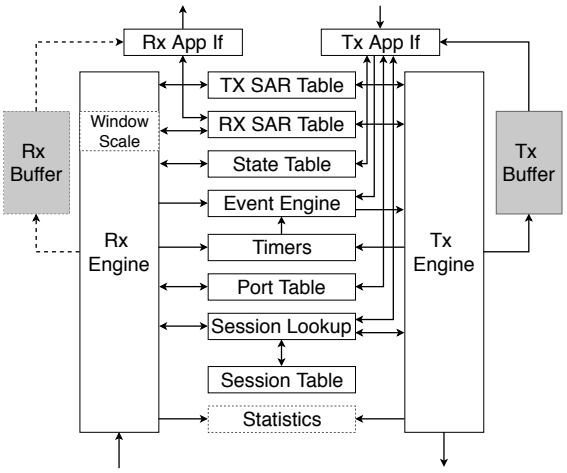

Fig. 2: TOE Architecture Overview.

and prepended to the packet payload. The pseudo header and packet payload are then forwarded to the *Rx checksum* module. If the result equals to zero, the checksum is valid, and the packet is passed on to the next module, otherwise it is dropped. Valid packets are parsed to extract the necessary fields from the IPv4 and TCP headers, which is done in one clock cycle. The *Rx Engine* contains a Finite State Machine (FSM) that makes decisions based on the extracted fields. First, it looks up the destination port in the *Port Table*, if the port is not in the `LISTEN` mode the packet is discarded. Next, using the three-tuple — IP source address, TCP source and destination port — a look-up to the CuckooCAM is issued. The CuckooCAM returns a tuple with a boolean indicating if the lookup is a hit, and a 16- bit sessionID. The sessionID is used as an index to look-up the state of the connection in all the other tables. If the lookup was a miss but the packet has the SYN flag set, the three-tuple is inserted with a new sessionID and a SYN-ACK event is generated. The FSM uses the sessionID to retrieve the sequence and acknowledgment number from the two *SAR Tables* and, if necessary, updates them. Finally, if the packet contains a payload, a notification is sent to the application while the payload is written to the *Rx Buffer* — in the case that is enabled. The FSM in the *Rx Engine* enforces a strict order of the packets and does not support out-of-order processing.

B. Data Structures

Session Lookup: provides the means to interface with the CukooCAM, using the three-tuple to obtain the sessionID. The sessionID is used to index every data structure to access the state of the corresponding connection. In case of a SYN or a SYN-ACK packet, if the three-tuple has not been inserted yet, it will be inserted using a new sessionID identifying the new connection. Additionally, the *Session Lookup* module contains a table that maps the sessionID to the three-tuple. This mapping is used by the *Tx Engine* to generate the IPv4 and TCP headers of outgoing packets.

Port Table: keeps track of the state of each port, which can be `CLOSE`, `LISTEN` or `ACTIVE`. The standard port range for static and ephemeral ports are used. If an incoming packet targets a port in `CLOSE` state, it is discarded and a RST packet is generated as a response.

State Table: stores the current state of each connection as specified by RFC793 [30]. The *State Table* can be updated by the *Rx Engine* when incoming packets are processed and by the *Tx App If* when the application opens a connection. Consistency is guaranteed by using atomic operations.

Timers: this module supports all time-based event triggering as required by the protocol, three timer modules are implemented: *Re-transmission*, *Probe* and *Time-Wait Timer*. It follows the same approach of the original version, which provides linear scaling of on-chip memory.

Event Engine: gathers events from the *Rx Engine*, the *Timers*, and the *Tx App If*. Consequently, events are merged and forwarded to the *Tx Engine* that processes them to generate the corresponding outgoing packets. Each event will trigger the generation of a new TCP packet.

Buffering and Window Management: Since TCP is a stream-based protocol, it requires buffering on the receiving and transmitting side. On the receiving side, data is buffered in case the application is not able to immediately consume it. On the sending side, buffering is required for re-transmission in case of packet loss. Thus, when supporting multiple connections, the amount of memory that is needed increases linearly with the number of connections. For more than ten concurrent connections, the routing of on-chip memory becomes very complex and using DRAM to store the payloads becomes therefore mandatory. For every connection the memory buffer is logically implemented as a circular buffer which is stored in a fixed and pre-allocated segment within the off-chip memory. Stored in the **Tx and Rx SAR Tables** there are pointers, *e.g.*, *ack'ed*, *transmitted*. which represent the state of the TCP window of each connection at a given time. The information stored in these tables is mandatory to handle the segmentation and reassembly (SAR) of packets. Moreover, to support the *Window Scale* TCP option, the *Tx SAR Table* stores the *Window Scale* negotiated when the connection is established. This value defines the size and boundaries of the buffer.

Statistics: this module gathers events for inbound and outbound packets. The values can be read through an AXI4-Lite interface using the DMA subsystem. This element is optional and can be removed at synthesis.

C. Tx Engine

Each event triggers the generation of a new packet; the packet generation is done by the *Tx Engine*. The source of new packets can be the user application by either initiating a data transmission or by opening a new connection, which triggers a SYN packet. The *Rx Engine* generates events that generate ACK packets, including SYN-ACK. The *Timers* module triggers timeout-related events, such as re-transmission, probe packets and FIN packets for teardown. Like the *Rx Engine*, the *Tx Engine* has a FSM to handle each possible

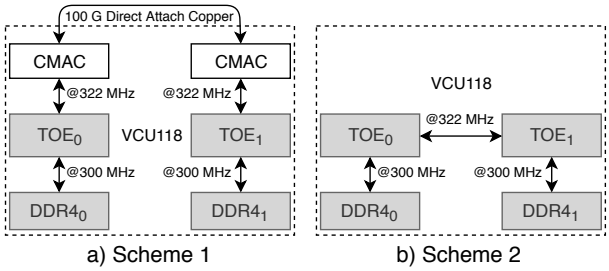

Fig. 3: TOE Interconnection Schemes

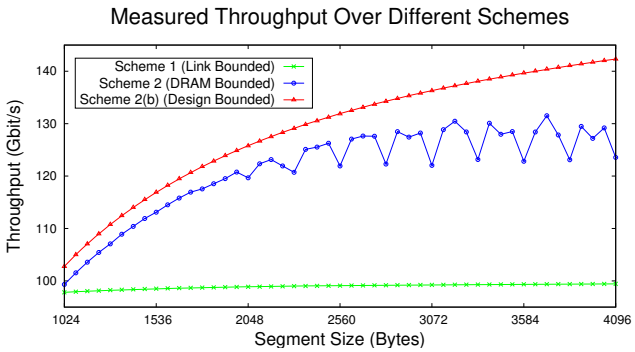

Fig. 4: Limago Performance over different schemes.

event. Contrary to *Rx Engine*, since each event carries the sessionID and event type, the sessionID is known when the event arrives. Consequently, the data-structures are queried immediately getting the necessary metadata to generate the packet. The Destination IP address and TCP ports are queried from the *Session Lookup*. Once the metadata is retrieved, the TCP pseudo header can be built. If the packet has payload, it is fetched from the external memory or directly from the application. Prepending the TCP pseudo header with the payload, the *Tx Checksum* computes the TCP checksum. Later, the IP header is prepended to the TCP packet. Finally, the packet is forwarded to the *Outbound Packet Handler*.

VI. Evaluation

A. Setup

The evaluation of Limago covers both functionality and performance. In terms of functionality, first the ARP and ICMP modules were tested using Linux-GNU's `arping` and `ping` programs. Then, to test the TOE, we implemented an echo server transmitting the received payload back to the sender. Such a design allows to test both the Rx and Tx Engines. The same echo server is used to verify the correct functionality of the internal elements as well as verifying connectivity.

For the performance evaluation, we use iPerf [31] version 2. We implemented iPerf in hardware, using Vivado-HLS, supporting both client and server modes. As a client, the application actively opens a connection and sends data at the highest possible rate to the server. As a server, the application waits for a SYN packet to establish a new connection. Once the connection is established, the client starts transmitting data and the application on the FPGA consumes the incoming payload while the TOE acknowledges the received packets. We have also built a user program on top of the Xilinx DMA driver to interact with the iPerf application deployed on the FPGA.

Limago was tested using two different configurations (Fig. 3). Scheme 1 corresponds to a standard implementation, each TOE communicates with the corresponding CMAC, and a 100G cable connects both CMACs. In this case the maximum throughput is limited by the Ethernet connection. Scheme 2 removes the Ethernet CMAC and connects the TOE using a 512-bit AXI4-Stream interface clocked at 322 MHz. The idea behind this configuration is to verify the maximum throughput. In the second configuration, we also have tested replacing DDR4 with a 512 KiB URAM — Scheme 2(b). This allows us to verify the physical bounds for each part of the design.

B. Bandwidth

To test the bandwidth of Limago we measured the throughput under two configurations (Fig. 4) one to test the throughput over a network and another to test the maximum processing rate of Limago when not limited by the network. In this experiment, TOE_0 transmits data to TOE_1, *i.e*, only the memory attached to TOE_0 is involved. The throughput reported measures the complete Ethernet frame, *i.e.*, including the Ethernet, IP and TCP headers as well as the payload. In this experiment, the application transmitted segments ranging between 1024- Byte to 4096- Byte in steps of 64- Byte, using only one connection, each experiment lasted five minutes. For scheme 1, using external DRAM and transmitting packets over the 100 Gbit/s Ethernet link, the throughput is bound by the network. Scheme 2, using DRAM, Limago transmits more than 100 Gbit/s for all cases. However, beyond 2048- Byte segment size, the DRAM bandwidth limits the throughput. Scheme 2(b), using on-chip URAM, looks like a logarithmic function where the throughput increases with an increasing segment size. These experiments show that Limago is able to surpass 100 Gbit/s when it is not bound by network.

We also have carried out experiments with multiple connections at the same time. For those experiments we have used two servers and a Huawei cloudEngine 8800 switch. The specifications of the severs are as follows: both servers run on a 4.14.7-gentooHPC OS and use a Mellanox MT27800 ConnectX-5 100 Gbit/s NIC; server A has an Intel Xeon CPU E5-2630 v4 at 2.20 GHz and 128 GB of RAM memory, whereas, server B has an Intel Xeon Gold 6126 CPU at 2.60 GHz and 192 GB of RAM memory. All offloading capabilities have been enabled in both machines, using `ethtool`. We use iPerf to test the performance, this time the servers work as a client, which means they send the data. Three different scenarios have been evaluated, each server individually and both servers combined. For both servers combined, each one contributes with half of the connections. The number of concurrent connections range from two to thirty in steps of two, each test lasted five minutes and was repeated five times, we

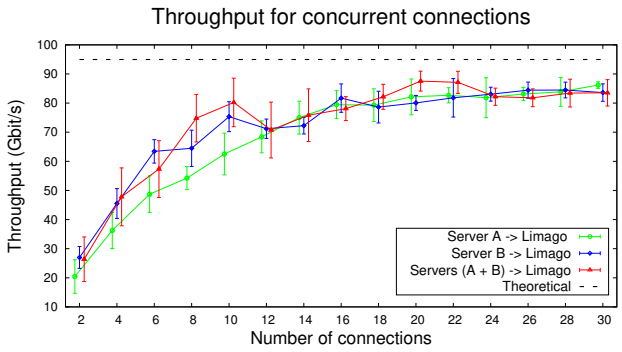

Fig. 5: Concurrent Connections, mean and standard deviation.

used the maximum packet size which is 1460-Byte. Figure 5 shows the results — which are measured at the application level — the mean and standard deviation are plotted, as well as the theoretical maximum. In general, the performance increases with a higher number of concurrent connections, until it is stable. With regard to both servers sending data simultaneously, a better performance is not observed, from this we notice that the switch could be the bottleneck. Further experiments are necessary to confirm this.

C. Resource Usage and Code Complexity

Limago has been implemented using Vivado and Vivado HLS 2018.2. The prototype uses a VCU118 board with a Virtex Ultrascale+ FPGA. Table I list the BRAM usage of the TOE for a wide variety of number of connections at a specific Window Scale. The LUTs and Flip-Flop cost is omitted to due to small difference between the different scenarios — 36 K to 41 K. As explained earlier and confirmed by the actual BRAM usage, the data structures in our implementation scale linearly with the number of supported connections.

The resource usage of Limago for 10,000 connections and no Window Scale is listed in Table II. The overall LUT usage is at 10% whereby 3.1% is used by the TOE. The TOE is also 1.5% of the available Flip-Flops which is around 20% of the total usage. But at the same time 90% of the logic resources are still available and can be used to deploy an application on the FPGA. BRAM capacity is a scarcer resource, the TOE uses almost 12% of them, overall around 80% of BRAM and 100% URAM capacity is still available for further use. The table also shows the resource summary for the 10 G starting point implementation, the resources of the TOE increased by a factor of 1.2 to 2.1. The overall logic resources increased by a factor of two and the BRAM usage by 20%. Particularly noteworthy, the tenfold bandwidth increase, at worse, only requires twice as much resources.

Limago has ten core modules, seven of them are written in HLS. Apart from the checksum, the other two HDL modules are straightforward, however determinism is needed. We used cloc [32] to count the lines of code (no headers), the HLS part is 7,456 lines; whereas the HDL is 1,482 lines.

TABLE I: No. of BRAM18 for different TOE configurations.

#conn	Window Scale 0	1	2	3	4	5	6	7
	Number of BRAM18							
1	198	221	221	222	222	222	222	222
128	202	225	225	226	226	226	226	226
512	202	225	225	226	226	226	226	226
1,024	204	227	227	233	233	233	233	233
2,048	228	251	251	257	257	257	257	257
4,096	276	299	299	305	305	305	305	311
8,192	371	397	397	403	403	409	409	414
10,000	495	514	514	520	526	532	538	544
16,384	566	602	613	619	625	631	637	643
32,768	974	1,023	1,035	1,047	1,059	1,071	1,083	1,095
65,536	1,774	1,843	1,867	1,891	1,915	1,947	1,963	1,995

TABLE II: Full design resource usage on VCU118

Element	**LUT**		**FF**		**BRAM**	
100 G						
Memory	17,423	1.5%	25,995	1.1%	41.5	1.9%
CMAC	14,614	1.2%	39,550	1.7%	26.5	1.2%
ARP	1,260	0.1%	3,193	0.1%	1.5	0.1%
ICMP	2,056	0.2%	5,561	0.2%	0	0.0%
Inbound	1,816	0.2%	6,293	0.3%	8.5	0.4%
Outbound	2,680	0.2%	9,324	0.4%	34	1.6%
CuckooCAM	2,095	0.2%	1,392	0.1%	36	1.7%
TOE	36,469	3.1%	36,229	1.5%	247.5	11.5%
Total	**119,844**	**10.1%**	**178,339**	**7.5%**	**441.5**	**20.4%**
10 G						
TOE	15,415	1.3%	16,616	0.7%	186.5	8.7%
SmartCAM	2,201	0.2%	1,772	0.1%	57.5	2.7%
Total	**77,393**	**6.6%**	**85,306**	**3.6%**	**369**	**17.1%**

VII. Conclusions

Limago is an open-source [33] 100 Gbit/s TCP/IP stack that can be implemented on FPGA to enable research and development in programmable NICs and in-network computing. Starting from a pre-existing stack operating at 10 Gbit/s, Limago provides a tenfold increase in bandwidth at the cost of a mere 20% increase in BRAM usage, without jeopardizing the ability to support multiple connections of the original design, and maintaining the same design methodology based on Vivado-HLS. The current prototype has been implemented and successfully tested on Xilinx VCU118 and Alveo U200 boards. Future work includes further optimizations of the stack to, *e.g.*, enable reordering of out-of-order packets and additional TCP features taking advantage of the increasing availability of High Bandwidth Memory in FPGAs. This feature will improve the throughput when packet loss occurs as well as support application level processing [34], [35].

Acknowledgment

This work was supported in part by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under project TRÁFICA (MINECO/FEDER TEC2015-69417-C2-1-R) and by the European Commission under project METRO-HAUL (grant agreement No. 761727). Part of the work on Limago of M. Ruiz and G. Sutter was done at ETH Zürich as part of a visit funded by Univ. Autónoma de Madrid and José Castillejo program, respectively.

REFERENCES

[1] A. Lord, A. Soppera, and A. Jacquet, "The Impact of Capacity Growth in National Telecommunications Networks," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2062, p. 20140431, 2016.

[2] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, "The Case For In-Network Computing On Demand," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: ACM, 2019, pp. 21:1–21:16. [Online]. Available: http://doi.acm.org/10.1145/3302424.3303979

[3] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A Cloud-Scale Acceleration Architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, October 2016.

[4] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *ISCA'14*, 2014.

[5] D. Firestone, A. Putnam, S. Mundkur, D. Chiou *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *NSDI'18)*, 2018.

[6] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC," in *SOSP '17*, 2017.

[7] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, M. Ghandi, D. Lo, S. Reinhardt, S. Alkalay, H. Angepat, D. Chiou, A. Forin, D. Burger, L. Woods, G. Weisz, M. Haselman, and D. Zhang, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *IEEE Micro*, vol. 38, pp. 8–20, March 2018.

[8] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in Hyperscale Data Centers," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, 2015.

[9] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent Distributed Storage," *PVLDB*, vol. 10, no. 11, Aug. 2017.

[10] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a Box: Inexpensive Coordination in Hardware," in *NSDI'16*, 2016.

[11] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 36–43.

[12] J.-P. Deschamps, G. J. Bioul, and G. D. Sutter, *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*. John Wiley & Sons, 2006.

[13] G. Sutter, M. Ruiz, S. Lopez-Buedo, and G. Alonso, "FPGA-based TCP/IP Checksum Offloading Engine for 100 Gbps Networks," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2018, pp. 1–6.

[14] Xilinx Inc., "Exact Match Binary CAM Search IP for SDNet," Xilinx Inc., Tech. Rep., 11 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/cam/pg189-cam.pdf

[15] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger, "RFC7323: TCP Extensions for High Performance," Tech. Rep., 2014.

[16] A. Currid, "TCP Offload to the Rescue," *Queue*, vol. 2, no. 3, pp. 58–65, 2004.

[17] S. Senapathi and R. Hernandez, "TCP Offload Engines," *Network AND Communications magazine pp103-107*, 2004.

[18] W.-C. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda, "Performance Characterization of a 10-Gigabit Ethernet TOE," in *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*. IEEE, 2005, pp. 58–63.

[19] LDA Technologies, "LDA Lightspeed TCP," http://www.ldatech.com/lda-lightspeed-tcp/, accessed: 2018-07-17.

[20] Chevin Technologies, "Chevin Technology's TCP/IP," https://chevintechnology.com/ethernet-ip-2/ct1008-xgtcp/, accessed: 2019-01-09.

[21] Enyx, "10G TCP/IP Full-Hardware Stack IP Core Offload Engine for Xilinx FPGA," http://www.enyx.com/nxtcp-xilinx/, accessed: 2018-07-17.

[22] Enyx, "Enyx Premieres 25G TCP and UDP Offload Engines with Xilinx Virtex UltraScale+ 16nm FPGA on BittWares XUPP3R PCIe Board," http://www.enyx.com/2016/11/enyx-premieres-25g-tcp-udp-offload-engines-wxilinx-virtex-utlrascale-16nm-fpga-bittwares-xupp3r-pcie-board/, accessed: 2018-07-17.

[23] Dini Group, "TCP Offload Engine IP - 128 Sessions (TOE128)," https://www.dinigroup.com/web/TOE128.php, accessed: 2018-07-17.

[24] Algo-Logic, "10G TCP Endpoint," http://algo-logic.com/tcp, accessed: 2018-07-17.

[25] U. Langenbach, A. Berthe, B. Traskov, S. Weide, K. Hofmann, and P. Gregorius, "A 10 GbE TCP/IP Hardware Stack as Part of a Protocol Acceleration Platform," in *Consumer Electronics¿ Berlin (ICCE-Berlin), 2013. ICCEBerlin 2013. IEEE Third International Conference on*. IEEE, 2013, pp. 381–384.

[26] L. Ding, P. Kang, W. Yin, and L. Wang, "Hardware TCP Offload Engine based on 10-Gbps Ethernet for low-latency Network Communication," in *Field-Programmable Technology (FPT), 2016 International Conference on*. IEEE, 2016, pp. 269–272.

[27] G. Bianchi, M. Welzl, A. Tulumello, G. Belocchi, M. Faltelli, and S. Pontarelli, "A Fully Portable TCP Implementation Using XFSMs," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. ACM, 2018, pp. 99–101.

[28] Y. Ji and Q.-S. Hu, "40Gbps Multi-Connection TCP/IP Offload Engine," in *Wireless Communications and Signal Processing (WCSP), 2011 International Conference on*. IEEE, 2011, pp. 1–5.

[29] D. Sidler, Z. István, and G. Alonso, "Low-latency TCP/IP Stack for Data Center Applications," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 2016, pp. 1–4.

[30] J. Postel *et al.*, "Transmission Control Protocol RFC 793," 1981.

[31] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "iPerf: the TCP/UDP Bandwidth Measurement Tool (2005)," *URL: http://iperf. sourceforge. net*, 2005.

[32] "Count Lines of Code (cloc)," https://github.com/AlDanial/cloc/.

[33] "100G-fpga-network-stack-core," https://github.com/hpcn-uam/100G-fpga-network-stack-core/.

[34] L. Woods, Z. István, and G. Alonso, "Ibex - An Intelligent Storage Engine with Support for Advanced SQL Offloading," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 963–974, 2014.

[35] L. Woods, J. Teubner, and G. Alonso, "Complex Event Detection at Wire Speed with FPGAs," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 660–669, 2010.