2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines

Scalable 10 Gbps TCP/IP Stack Architecture for Reconfigurable Hardware

David Sidler, Gustavo Alonso
Systems Group
Dept. of Computer Science
ETH Zürich
{dasidler, alonso}@inf.ethz.ch

Michaela Blott, Kimon Karras,Kees Vissers
Xilinx Research
Dublin, Ireland
& San Jose, CA
{mblott, kimonk, kees.vissers}@xilinx.com

Raymond Carley
Dept. of Electrical
& Computer Engineering
Carnegie Mellon University
rcarley@ece.cmu.edu

***Abstract*—TCP/IP is the predominant communication protocol in modern networks but also one of the most demanding. Consequently, TCP/IP offload is becoming increasingly popular with standard network interface cards. TCP/IP Offload Engines have also emerged for FPGAs, and are being offered by vendors such as Intilop, Fraunhofer HHI, PLDA and Dini Group. With the target application being high-frequency trading, these implementations focus on low latency and support a limited session count. However, many more applications beyond high-frequency trading can potentially be accelerated inside an FPGA once TCP with high session count is available inside the fabric. This way, a network-attached FPGA on ingress and egress to a CPU can accelerate functions such as encryption, compression, memcached and many others in addition to running the complete network stack.**

This paper introduces a novel architecture for a 10 Gbps line-rate TCP/IP stack for FPGAs that can scale with the number of sessions and thereby addresses these new applications. We prototyped the design on a VC709 development board, demonstrating compatibility with existing network infrastructure, operating at full 10 Gbps throughput full-duplex while supporting 10,000 sessions. Finally, the design has been described primarily using high-level synthesis, which accelerates development time and improves maintainability.

I. Introduction

TCP/IP is the cornerstone of modern network communications with its support for reliable data transfer including flow control, congestion avoidance, duplicate data suppression and in-order delivery. However, this is associated with substantial complexity. Foong [1] stipulates that 1 Hertz of CPU processing is required to send or receive 1 bps of TCP/IP which equates to 8 cores clocked at 1.25 GHz for 10 Gbps line-rate processing. The reasons for this are manyfold and well understood. First of all, as TCP is connection-oriented, the implemented network stack needs to keep state for every connection, something which naturally becomes more complex with the number of open sessions. Secondly, to ensure reliable data transfer, data has to be buffered until an acknowledgment has been received. Additionally, segmentation and reassembly are needed together with out-of-order processing to packetize incoming and outgoing data streams from the application layer. Finally, TCP is interrupt-intensive in nature as, for example, every time a packet is received or a transmission time-out occurs an interrupt is triggered. Together with its large footprint, which exceeds the capacity of standard instruction caches and therefore causes a high miss rate, this leads to poor branch-predictability on standard x86 execution and disrupts co-executing applications [2].

As a result, TCP/IP offload is increasingly integrated into network interface cards with many of the major vendors, such as Broadcom, Emulex, and Chelsio offering full offload since 2011 [3]. TCP/IP Offload Engines (TOEs) have also emerged for FPGAs offered by vendors such as Intilop, Fraunhofer HHI, PLDA and Dini Group [4], [5], [6], [7]. However, these implementations target high-frequency trading which is driven by latency requirements [8]. To minimize latency, these stacks constraint session support as we explain further in section II.

Our design aims to expand the applicability of FPGAs by creating a flexible architecture that, in addition to delivering full line-rate throughput, can also scale to high session counts, an essential prerequisite to deployment in networked servers in data centers. Beyond alleviating the TCP/IP bottleneck on the host CPU, the solution aims to accelerate applications such as encryption, compression, memcached [9] or higher-level protocol processing such as JMS [10] by pushing the TCP termination completely inside the FPGA. This enables acceleration through reconfigurable logic for potentially thousands of sessions.

To maximize the stack's applicability, it was essential to create a flexible solution that allows to efficiently and easily adapt the design to different congestion avoidance schemes, potentially out-of-order processing, etc., while using a minimal resource footprint. To achieve this, we adopted a C++-based design flow using high-level synthesis (HLS) that simplifies the design, makes it more flexible and easier to customize towards specific requirements. As an added bonus, the HLS-based design is automatically portable to any device supported by the tool and significantly increases productivity.

In more detail, the key contributions of this paper are as follows:

- Sustained 10 Gbps bandwidth TCP/IP stack through dataflow architecture
- Support for 10,000 concurrent connections with external data buffers, hash-based session lookup, and linear scalability
- Design flexibility through C/C++ design using Vivado HLS
- Control flow features and out-of-order segment processing
- Performance evaluation in a real world setup

This paper is structured as follows: We present related work in section II. The overall system architecture is introduced in section III, while the TOE is discussed in detail in section IV. We evaluate the system in section V. Section VI concludes the paper.

978-1-4799-9969-9/15 $31.00 © 2015 IEEE
DOI 10.1109/FCCM.2015.12

II. Related Work

There have been both commercial and academic TCP/IP implementations on configurable logic in the past. On the academic front, Dollas et al. [11] presented an open TCP/IP architecture in 2005 with limited throughput (estimated at 350 Mbps) and limited connection support (31 active and 16 passive). Buffers are shared between all protocols and connections which limits parallelism. [12] and [13] presented TOEs using a combined hardware-software solution. A PowerPC CPU handles most of the TCP processing, while checksum validation and computation, Address Resolution Protocol (ARP), Internet Control Message Protocol (ICMP) and IP packets are processed by the FPGA. A complete TOE for FPGAs without processor assist was presented in [14]. With the target application being embedded devices, resource usage was paramount and feature set limited. Most recently, [15] presented a TOE with a centralized scheduler, which is estimated to sustain 4 Gbps for receiving and 40 Gbps for transmitting data for maximum segment sizes of 1460 bytes, targeting asymmetric workloads with large size packets such as video on demand. In contrast, our data-flow based approach supports 10 Gbps full-duplex even with worst case minimum-size packets of 64 B, thereby enabling much higher packet rates. Furthermore, we support a more complex and deeper packet buffering system which offers 64 KB per session as opposed to the single buffer used for all connections in [15]. This provides more flexibility and less tighter coupling to the application. Out-of-order (OOO) processing is equally supported with minor differences on the data structures; We store only offset and length for OOO blocks instead of all SEQ numbers explicitly and write the OOO blocks directly to their position within the receive buffer instead of sorting through received segments when access is required. Some other work has focused on parts of a TCP/IP stack. [16] presented an ARP module. [17] showed a User Datagram Protocol (UDP) stack targeting 1 Gbps. Finally [18] uses a basic low performance TCP/IP stack to configure and test a circuit running on the FGPA.

In regards to commercial systems, multiple TCP/IP stacks for FPGAs are available from vendors such as Intilop, Fraunhofer HHI, Dini Group and PLDA [4], [5], [6], [7]. Up until 2014, all of the available TOEs were ultra-low latency and low session count, typically well below 256 concurrent sessions, with the key driving application being high-frequency trading (HFT). To minimize latency, these stacks are typically constrained in session support as high session counts directly impact latency two-fold. Firstly, for every session, packets need to be buffered for both packet reception and transmission. Assuming a typical TCP window size of 64 KB and given that standard FPGAs are typically limited to hundreds of Mb in on-chip memory (for example 132.9 Mb for the current 20 nm Xilinx generation [19]), packet buffering for anything more than a few hundred sessions needs to be moved off-chip into external DRAM. This has an adverse effect on latency (we have measured the overhead of external packet buffering to be over 600 ns and increasing with segment size, see section V-B). Secondly, to associate an incoming or outgoing packet with a session, one needs to perform a lookup using the four-tuple of source and destination IP address, source and destination TCP port. Lookup problems can be concluded within 1 clock cycle when ternary content addressable memory (TCAM) style architectures are deployed. In essence, one stores every entry of the session table inside the chip together with a comparator to the incoming four-tuple that enables a comparison of all entries within a single cycle. However, these architectures become prohibitively resource expensive for high session counts.

Since the beginning of the year 2014, Intilop has announced a high session count variant with support for up to 16,000 sessions which is probably the closest related available TCP/IP stack in terms of feature set. A detailed architectural comparison is at this stage not possible, as Intilop has not disclosed the design. However, our implementation is written in C++ which increases design flexibility, maintainability and portability.

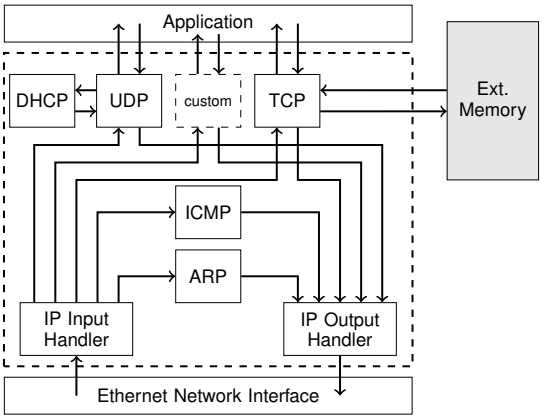

Fig. 1. Block diagram of the implemented TCP/IP stack

III. System Architecture

Implementing TCP/IP involves the implementation of a stack of protocols such as ARP, ICMP, IP, UDP, TCP and Dynamic Host Configuration Protocol (DHCP) whereby each protocol handles a different aspect of the communication. This is reflected in Fig. 1 which illustrates the system architecture of our stack within the dashed lines. At the lowest layer, the stack interfaces with the *Ethernet Network Interface* which includes both the Media Access Control (MAC) and the physical layer which handle layer 1 and 2 functionality of the network stack both of which are standard Xilinx IP cores. Incoming packets are parsed by the *IP Input Handler* which also validates IP checksum, determines if they match any of the supported protocols, and forwards the packets to the corresponding modules accordingly. Invalid packets are discarded and unsupported protocols are forwarded to a separate interface.

The *ARP* module handles address resolution and replies and generates ARP requests when needed. For this, it contains a lookup table that maps IP addresses to their corresponding MAC addresses. The ARP table is implemented in on-chip block memory (BRAM). ICMP enables the exchange of control related messages between two hosts and is processed in the *ICMP* module. Our implementation supports the most popular subset, namely "echo" or "ping" messages, "destination unreachable" messages, which are created upon packet reception for a closed port, and "TTL expired" messages, which are returned when datagrams with an expired time-to-live (TTL) field are observed. The *UDP* module handles the processing of the corresponding protocol, which is a stateless light-weight

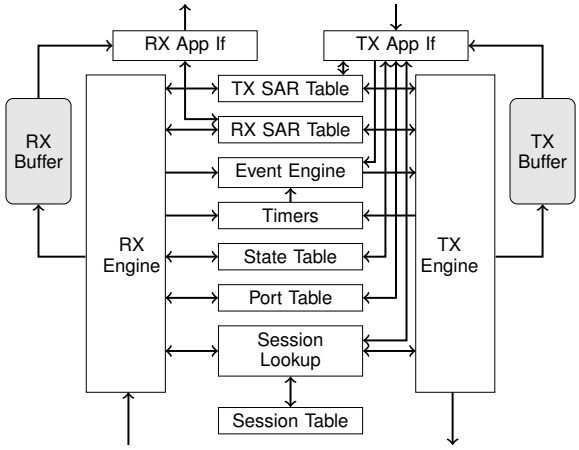

Fig. 2. Block diagram of the TCP module

protocol that allows for quick and easy data exchange between two end-points. The majority of the design complexity resides with the *TCP* module and forms the key part of our work. We describe its architecture in detail in section IV. Both the *UDP* and the *TCP* module interface to the *IP Output Handler* which merges all the data streams back into a single stream, computes the IP checksum, issues a lookup to the ARP table, formats headers and passes them onto the *Ethernet Network Interface* for transmission. Finally, our stack also includes a minimal DHCP client which allows for dynamic configuration of the device's IP address over the network.

IV. TCP Architecture

The TCP Offload Engine is the key component of this work. In this section we will discuss its architecture, processing algorithms, and its key characteristics.

A. Architecture

As shown in Fig. 2, a fundamental aspect of the design is that it is divided into two parallel paths, one for incoming packets (RX), and one for outgoing packets (TX). Both paths consist of a protocol engine (*RX* and *TX Engine*), a packet buffer (*RX* and *TX Buffer*) and an application interface (*RX* and *TX App If*). The two paths are implemented as data-flow architectures and store and share connection state information through the central data structures located at the heart of the architecture. Separating the two data-flow pipelines from the state-storing data structures, as well as the careful partitioning of data structures itself, is essential to achieving line-rate bidirectional throughput. The protocol engines handle all protocol processing, the packet buffers store the packets between the engines and the application interfaces which provide and receive data to and from the user application. The following paragraphs elaborate on the data structures, packet buffering mechanics, as well as the two protocol engines.

1) Session lookup, port and state table, timers and event engine: The *Session lookup* module maps the four-tuple (source and destination IP addresses and TCP ports) to a so-called session ID which represents a connection and is used as an index for all other data structures. At the heart of the lookup problem is a scalable TCAM implementation which is based on a hash table and presented in [20]. In comparison to a traditional TCAM, it uses less resources and scales linearly in its resource requirements with the number of entries. The *Port Table* keeps track of the state of each port which can be closed, listen or active. According to the standard, we support two port ranges, one for static ports (0 to 32,767), which we use for listening ports, and one for dynamically-assigned or ephemeral ports (32,768 to 65,535), which are used for active connections. For every incoming packet the *RX Engine* queries the *Port Table* to check if the destination port is either in the listen or active state. If this is not the case, then the packet is immediately discarded. Similarly, the application interfaces are accessing the *Port Table* to listen on a port or get a new ephemeral port. The *State Table* stores the current state of each connection. The state values represent the states as specified by RFC793 [21], i.e. CLOSED, SYN-SENT, SYN-RECEIVED, etc. State values can be updated from the *RX Engine* but also from the *TX App If* when a new connection is opened. To guarantee consistency of the state entries, we support atomic read-modify-write operations (RMW). The locking is fine-grained so that only the currently accessed entry is locked. The *Timers* module supports all time-based event triggering as required by the protocol with the following kinds: The "Retransmission Timer" keeps track of the retransmission intervals for packets which have been sent but not acknowledged by the remote host. The "Probe Timer" is set in case data is available but cannot be sent immediately and transmission has to be postponed. Finally, the "Time-Wait Timer" handles the long time-out in the TIME-WAIT state before the connection reaches the CLOSED state. We adopt a highly efficient architectural approach as introduced by [22] where each timer is represented by a single table with one entry per connection with one timer per connection for retransmission, as explained in RFC6298 [23], rather than for every segment. This provides linear scaling of embedded memory (BRAM) resources with number of open connections. In more detail, to start the timer, the entry of this connection is set to the value of the time-interval. Every n_{th} cycle, where n is the maximum number of connections, the entry of each connection with an active time-interval is decremented until zero is reached and the appropriate event is triggered. This serial probing of all connection timers is a viable approach, as TCP timers operate with a millisecond granularity. Given a clock period of 6.4 ns and one timer access per clock cycle, we can update 156,250 connections per millisecond. Generated events are routed through the *Event Engine*, together with events from the *RX Engine* and the *TX App If*, to the *TX Engine*. The aggregated event stream is then merged with outstanding acknowledgments [24].

2) TCP buffer and window management: TCP operation requires the buffering of payloads to facilitate retransmission and flow control for both receive and transmit. Our implementation allocates two fixed-sized buffers of 64 KB per connection. This means that for a design supporting 10,000 connections, a total of 1.3 GB of external memory is necessary. Since this amount of memory space is not available on-chip, external DDR3 memory is used. The buffers are implemented as circular buffers, each in a pre-allocated region of the memory. Managing each of the buffers requires a set of pointers to

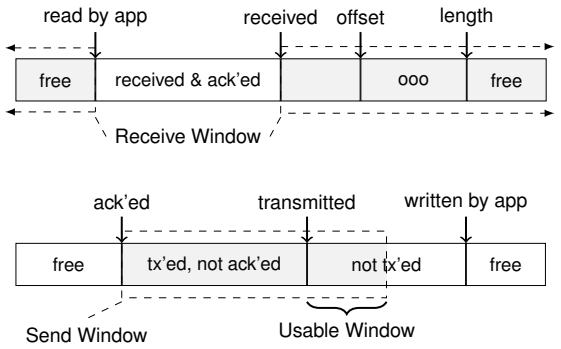

Fig. 3. TCP buffer and window management

keep track of various, relevant locations in the segment stream. These pointers are stored in the *TX* and *RX SAR Tables*, which are instrumental for handling all segmentation and reassembly (SAR) functionality as well as maintaining the TCP windows. Fig. 3 illustrates the two packet buffers. For RX, the buffer space is divided as follows: data that was received (and acknowledged) but not read by the application, available free space to receive new segments, and a non-contiguous number of blocks of OOO received data (as described in section IV-C3). On the transmission side, we need to keep track of three partitions: transmitted but not yet acknowledged data, data written by the application to the buffer but not yet sent, and finally free space. In addition to these three pointers, the *TX SAR* also stores the *Send Window* which is advertised by the other device and represents the size of its receive buffer. The *Usable Window*, as shown in the figure, can be computed on the fly and is not explicitly stored in the table. Both tables hold one entry per connection and scale therefore linearly with the maximum number of connections.

3) RX and TX Engine: The *RX Engine* is responsible for the protocol processing on incoming packets and performs a series of checks and data structure updates before writing the segment payload into the *RX buffer*. This includes the following functions[1]: The *RX Engine* computes the TCP checksum and verifies its correctness. It extracts meta-information (SEQ number, ACK number, length, IP addresses and TCP ports, window size, TCP flags) and checks if the destination port is accessible. The *RX Engine* also issues the session ID lookup of the connection and queries the current state of the arrived segment's TCP connection from the *State* and *SAR Tables*. The updated state is computed and written back while timers are set and events triggered as needed as (for example for the transmission of acknowledgment packets). If the packet contains a payload that is within the required receive window, then it is written to the *RX Buffer* and the application is notified. If the received data is valid but out of order, then the data is written to the *RX Buffer* and processed as described in section IV-C3.
The *TX Engine* is responsible for transmitting TCP segments which can stem either from the user application providing new data for transmission, or from the multitude of events such as a simple retransmission event or an open session request from the application, which triggers the sending of a SYN segment. These events can be set off by the *RX Engine*, any of the *Timers* or the *TX App If*. The main component in the *TX Engine* is a state machine triggered by the *Event Engine*. In contrast to the *RX Engine*, in the *TX Engine* the session ID is known when the event arrives and thus the data structures can be immediately queried for all the necessary meta-data required to generate the packet. A reverse lookup is however needed to determine the source and destination IP addresses and TCP ports from the session ID. Upon completing these actions, the *TX Engine* triggers other modules to construct the TCP header, fetch data from external memory for the payload if needed and compute the TCP checksum. Headers and payload are then concatenated and streamed out to the *IP Output Handler*.

[1]TCP options are currently not supported and are ignored.

B. Life of a Packet

This section describes the life of a packet for the typical three-way TCP handshake when a session is set-up. This highlights many different event types and shows the interaction between the *RX* and *TX Engine*. In this scenario, the FPGA acts as a server and listens on a specific port while another device connects as a client to that port.

1) A SYN packet arrives from the *Ethernet Network Interface*
2) The *IP Input Handler* determines that this is a valid IP packet and checks its IP checksum.
3) It then establishes that this is a TCP segment and forwards it to the *TCP* module.
4) Within the *TCP* module, the TCP pseudo-header is first constructed and the TCP checksum is verified.
5) Subsequently, the destination port is queried from the *Port Table* and verified that it is in the LISTEN state.
6) Then a new entry is created in the *Session Table* and the corresponding session ID is returned.
7) The *RX Engine* queries all meta information from *State* and *SAR Tables*. After ensuring that this is a new connection, the state is transitioned from LISTEN to SYN-RCVD.
8) The *RX Engine* initializes the *SAR tables* and triggers a SYN-ACK event.
9) The SYN-ACK event passes through the *Event Engine* and reaches the *TX Engine*.
10) The *TX Engine* extracts the session ID from the SYN-ACK event.
11) On the basis of this, the *TX Engine* then queries all necessary meta information to construct a SYN-ACK segment.
12) It then computes and inserts the TCP checksum as well as part of the IP header with IP addresses and TCP ports.
13) The packet is streamed to the *IP Output Handler* which computes and inserts the IP checksum. It also queries the MAC address corresponding to the destination IP address from the ARP Table and prepends the MAC header.
14) An ACK packet arrives to complete the handshake.
15) Again the *IP Input Handler* validates the packet as a TCP segment.
16) The segment is then forwarded to the *TCP* module, where the *RX Engine* parses and verifies the TCP checksum for correctness.
17) The *RX Engine* queries the *Port Table* to ensure that the port is still open for listening.

18) Then the session ID is retrieved from the *Session Lookup*. This time an entry already exists.
19) The *RX Engine* queries the state and meta information from the *State* and *SAR Tables* respectively. After verifying that the processed ACK is replying to the previously sent SYN-ACK, the state is transitioned from SYN-RCVD to ESTABLISHED. Pointers in the *SAR Tables* are updated and written back.
20) The connection is now successfully established and data can be exchanged.

C. Key Characteristics

This section expounds the key characteristics in our architecture to meet the driving requirements, which include 10 Gbps line rate, scalability to high session count, out-of-order processing, high-level synthesis, and support for flow-control.

1) 10 Gbps line-rate support: The *TCP* module was designed to process traffic at high data rates with 10 Gbps link-rate independently of the segment size which required a careful investigation of all data paths. To ensure this, we have designed the data paths between all modules in a data-flow fashion with a 64b data bus clocked at 156.25 MHz. Furthermore, we checked that external memory access requirements were met. Roughly speaking, we require twice the incoming and outgoing throughput in memory bandwidth, as every packet in RX and TX direction is read and written once. Therefore, the requirement amounted to 40 Gbps for the memory access which is less than 17% of the total of 239 Gbps of theoretical bandwidth that the two 64b 932 MHz SODIMM interfaces found on the VC709 board provide. Since TCP segments might be stored at any byte-offset and the length can vary to a great extent, the actual required bandwidth might be significantly higher depending on the workload. Finally, the most critical aspect of the design are the shared central data structures as for each segment, they have to be accessed and sometimes updated from various modules simultaneously. It is crucial to verify that even in the worst case, this process does not constrain the bandwidth.
We first consider the available time budget and then evaluate whether this requirement can be met for the most contentious data structures: Minimum Ethernet frames together with preamble and inter-frame gap amount to 84 B on the wire (64 B + 8 B + 12 B). With that the highest possible packet rate is around 15 million packets per second [Mpps]. Given a data bus width of 64 b and a 156.25 MHz clock frequency, the *TCP* module has to accept a new segment every 11 cycles which forms then the upper limit for all connection state processing. We, thus, have to ensure that all sub-modules have timely access to this state information and can complete their processing within the given time limit.
The modules in which the highest contention occurs are the *State Table*, *RX SAR* and *TX SAR Table*. As seen in Fig. 2 they are accessed from the two protocol engines and the application interfaces. As explained above, we need to service all accessing modules in the budgeted 11 cycle window. For this, we have detailed all the required accesses in TABLE I. The *State Table* is accessed from the *RX Engine* which performs a *RMW* operation and from the *TX App If* which reads the state of a connection before sending data out. Because these two accesses might happen concurrently, the entry which is undergoing a *RMW* operation has to be locked to ensure consistency. As listed, the locked access from the RX Engine takes 5 cycles. Assuming the worst case in which the *TX App If* accesses the same entry simultaneously, an additional 2 cycles have to be budgeted for a read access. This brings the total number of cycles per packet to 7, which is well within the budget.

TABLE I. CONCURRENT ACCESS TO THE MOST CONTENTIOUS DATA STRUCTURES

State Table			**RX SAR**			**TX SAR**		
source	type	cycles	source	type	cycles	source	type	cycles
RX Eng	rmw	5	RX Eng	r	2	RX Eng	r	2
TX App	r	2	RX App If	r	3	TX Eng	r	4
			TX Eng	r	4	RX Eng	w	1
			RX Eng	w	1	TX App If	w	1
			RX App If	w	1	TX Eng	w	1
Total		**7**			**11**			**9**

The *RX SAR Table* is more contentious as it is being accessed from three modules, namely the *RX Engine*, the *RX App If* and the *TX Engine*. The latter only does a read operation while the former two perform both *RMW* operations. However, as the modules all operate on separate aspects of the data structure, a locking during the *RMW* cycles is not required and operations can be interleaved. As shown in the table, the three read operations are followed by the two write operations which brings the sum to 11 cycles and exactly fits the budgeted window. Similarly, the *TX SAR Table* is accessed by the *RX Engine*, the *TX App If* and the *TX Engine*. Like the *RX SAR*, all modules can concurrently access without jeopardizing consistency as they are processing different fields of the data structure. The table shows the two read operations followed by the three write operations taking in total 9 cycles, which is well within the budget. With that, all data paths of the system are designed to sustain the line-rate requirement.

2) Scalability: As mentioned previously, a critical aspect of our architecture was that the resource requirements scale linearly with the number of supported sessions such that we can support deployments in data centers where thousands of servers communicate directly with each other. The only aspect in the design that changes with increasing session count are the data structures and buffers. The packet buffers are stored externally, and as previously determined require 64 KB$\times$2$\times$number of sessions. Even for 10 K sessions, this amounts to 1.3 GB which can easily be met with one 2 GB SODIMM. Given current DRAM support in today's devices (around 256 GB per 20nm device), it is feasible to support up to 1.97 million sessions from a packet buffering perspective. More critical are the resource requirements of the internal data structures. All tables (*State, Port, SAR*) and *Timers* hold one entry per connection as discussed in section IV, therefore they scale linearly. Further, the session lookup data structure also scales linearly with the number of sessions as has been shown in [20]. Thus, scaling is linear with number of supported sessions and limited by the amount of BRAM inside the FPGA while LUT and FlipFlop usage will remain mostly constant. Upper limits are discussed in section V. Further scaling, using external SRAM, is possible, however DRAM is not feasible as the data structures need to support a high access bandwidth with a fine granular access as discussed in the previous paragraphs.

3) Out-of-Order (OOO) Packet Processing: Packets can be dropped or re-ordered due to priorities or simply because they took different routes through the internet. Our design supports out-of-order reception of TCP segments as it offers a severe reduction in packet retransmissions, therefore improving the effective throughput of the system. The OOO processing occurs in the *RX Engine* and makes use of the *RX Buffer* and *RX SAR table*. The OOO segments are arranged into OOO blocks that consist of two pointers: the length of the block & the offset of the block from *received*, as seen in Fig. 3 which in essence equates to the sum of the sequence (SEQ) number and payload length of the previous in-order segment[2]. Each pointer is 13 bits by default, which is less than the 32 bits of SEQ number required otherwise. This saves 38 bits per block, which is significant across 10,000 sessions. The *RX Engine* expects the SEQ number of each packet it receives to be the same as *received*. If this is not the case, the packet is considered to be out-of-order and the following steps are executed:

1) If the SEQ number is between *read by app* and *received*, then the packet is dropped as it has already been received.
2) Otherwise, the OOO segment is accepted. The payload is written directly into the *RX Buffer* and the two pointers to the block are written to the *RX SAR table*.
3) If OOO segments overlap or are adjacent to previously received OOO data, then the pointers are updated accordingly.

Similarly to standard Linux TCP/IP stack, the OOO processing functionality in our design is highly customizable as the maximum number of OOO segments and the maximum distance to *received* are defined through global variables. This way it is possible to tweak the OOO processing to a particular application, or omit it entirely.

4) Flow Control & Performance Optimizations: Although flow control and congestion avoidance are essential features of TCP/IP, the focus of our work was on a scalable and high throughput architecture. We have adopted a simplified variation of the *Slow Start* algorithm as detailed in RFC5681[25] with an aim to limit the design's complexity. As an optimization, the so-called initial *Congestion Window* is set to 10×Maximum Segment Size as proposed by [26]. The *Congestion Avoidance* algorithm, which takes over after *Slow Start*, increases the window size in incremental steps. There are much more elaborate approaches used in real-world deployment such as TCP New Reno, TCP BIC, TCP CUBIC or Compound TCP. We defer the addition of such functionality to future work.
In regards to TCP performance optimizations, we implemented the following two additions: Firstly, to avoid the so-called small packet issue, where an application emits data in small chunks frequently only 1 byte in size, the TCP module makes use of *Nagle's algorithm* [27]. Secondly, we also support *Fast retransmit* as proposed in RFC2581 [28] which triggers a retransmit on duplicate acknowledgments, such that packet loss is detected before the retransmission timer times-out which decreases the retransmission latency significantly.

5) Using High-Level Synthesis (HLS): To speed up the development time and improve design flexibility, we have written the entire design in C++ using Vivado HLS [29]. This includes all modules in the stack, namely ARP, ICMP, UDP, TCP, DHCP and IP handlers, the only exception being the session lookup module for which we leveraged an existing implementation.
HLS provides significantly higher design abstractions such as data structures, built-in concepts for data streams with hidden flow control, simplified stitching of modules, and automated BRAM and FIFO instantiations. With that, the code becomes much more expressive, focused on the actual functionality and less cluttered with hardware design details. To illustrate this, TABLE II quantifies the number of lines of code for the current prototype implementation and its submodules. With a smaller more expressive code base, its legibility and maintainability are enhanced. Furthermore, it is easier to trim a design to specific requirements, thereby minimizing its resource and power footprint, and to make more complex, algorithmic changes, such as congestion avoidance and flow control schemes. As an added bonus the code becomes more portable and can be synthesized for any supported family.

[2]Wrap-arounds of SEQ numbers and pointers are being considered.

TABLE II. LINES OF C++ CODE

	IP Handlers	ARP Server	ICMP Server	TCP	**Total**
LoC	927	293	425	5,778	**7,423**

V. EVALUATION

In this secction our implementation is evaluated in regards to throughput, latency and resource consumption. Scalability and flexibility have been sufficiently discussed in section IV-C. For the performance evaluation, we utilized the following setup: The presented network stack was implemented and validated on a Xilinx VC709 evaluation board which features a Xilinx Virtex7 XC7VX690T FPGA. The design utilizes two 932 MHz DDR3 SODIMMs with 4 GB each and one 10 G network interface. In the experimental setup, the VC709 is connected via a Cisco Nexus 5596UP switch to ten servers, each equipped with an 8 Core Intel Xeon E5-2609 clocked at 2.4 GHz and 64 GB of main memory, running Linux (kernel 3.12) and an Intel 82599 10G Ethernet Controller.

A. Throughput

In our testbed, we operate the FPGA as a standalone device where the application is completely contained within the FPGA. The test application consumes all incoming packets and independently generates a stream of outgoing packets.
All measurements report application-level throughput which is the throughput that the application can effectively achieve. Obviously, its maximum value is lower than the actual link bandwidth of 10 Gbps and depends on the MSS which affects the ratio between actually transmitted data and its overhead. Our TCP module uses the default MSS value of 536 B. Taking the packet overhead consisting of Ethernet, IP and TCP headers into account, we can compute the maximum possible TCP throughput to be:

$$\frac{10Gbps * (536B * 8)}{(536B + 24B + 12B + 40B) * 8} = 8.76\,Gbps$$

8.76 Gbps should be considered an upper bound since some segments, e.g. SYN, ACK, are much smaller than the MSS and other non-TCP packets, like ARP, are also consuming part of the available bandwidth.

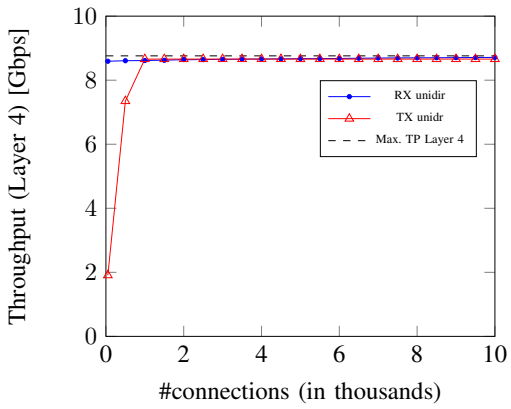

Fig. 4. Throughput depending on number of simultaneous connections

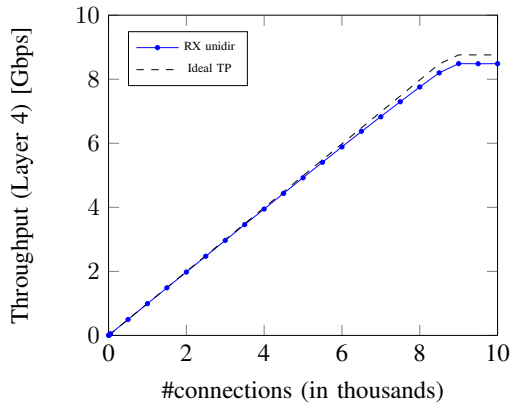

Fig. 5. Scaling number of simultaneous connections with fixed throughput per connection

1) High Load: To test the RX path, two servers generated traffic. For testing the TX path eight nodes were used to consume traffic coming from the FPGA whereby the number of concurrent connections was step-wise increased from 25 to 10,000. Each measurement took 120 s. The software clients measured the application throughput and the overall sum of all connections is plotted in Fig. 4 against the maximum theoretical throughput of 8.76 Gbps. On the RX side, 8.5 Gbps is reached with 25 connections while on the TX side 500 connections are required to reach 8.6 Gbps. The discrepancy stems from the servers not being able to consume the data at link-rate which leads to packet drops and retransmissions. With increasing number of connections, the load is better distributed between the different machines which reduces packet drop rates, and with that retransmissions and its negative impact on throughput.

2) Quality of Service: A further experiment aimed at illustrating how well the TCP module can service multiple connections when each connection uses limited but constant throughput. Two nodes are sending 96 KB of data roughly every 800 ms to achieve an average throughput of 1 Mbps per connection whereby the number of concurrent connections is increased from 2 to 10,000 in steps of 250. Fig. 5 shows the sum of the measured RX throughput in relation to the theoretical maximum throughput as a function of active connections. As can be seen from the graph, the throughput scales linearly with increasing number of connections whereby the discrepancy increases with the number of connections. The reasons for this are twofold: Firstly, the chance for packet drops increases with more connections and in comparison to the *High Load* experiment, connections cannot compensate for each other as they are all limited to a maximum of 1 Mbps throughput. Secondly, an increasing number of control and management packets are on the network which reduces the amount of available bandwidth.

TABLE III. STACK LATENCIES

Type	Cycle [6.4 ns]	Time[μs]
SYN-ACK	176	1.1
Payload [1 B]	170 (RX) + 131 (TX)	1.1 + 0.8
Payload [536 B]	375 (RX) + 402 (TX)	2.4 + 2.6

TABLE IV. RESOURCE USAGE ON VC709

	Network Interface	Memory Interface	TCP/IP Stack	**Total**	% of XC7VX690T Resources
FF	5,581	57,637	20,611	**83,829**	**9.6%**
LUT	5,321	43,591	19,026	**67,938**	**15.6%**
BRAM	8	36	279	**323**	**21.9%**

B. Latency

Latency was not the primary concern in our implementation, however it is an important characteristic of a network stack. With this in mind, we have measured the latency of our design in conjunction with a simple loopback application on the FPGA. Traffic was generated as described above and cycle-accurate latency was measured inside the FPGA with an embedded logic analyzer, namely Chipscope [30]. TABLE III lists the measured results which represent the time from the first word entering our network stack (as indicated within the dashed lines in Fig. 1) to the first word leaving it. The remaining latency through GTX, PHY, and MAC adds no more than 300 ns as we have observed in our projects. In regards to the table, the first row indicates the time it takes to send a SYN-ACK segment after arrival of a SYN as measured from ingress to egress of the stack. In this path, no external memory is involved. In the bottom 2 rows, we show the latency of a basic data packet with either 1 B or 536 B payload measured at the network interface and the application interface (RX) and vice versa (TX). The RX and TX path both have one read and write access to the external memory. As expected, segments with larger payloads incur higher latencies due to the additional DRAM access time and the TCP checksum verification (RX side) and calculation (TX side) which require buffering of a full segment before proceeding. Direct comparisons to processor-based platforms are difficult to conduct, however typical network to applications (RX) and application to network (TX) latencies as reported in [8] are in the range of 2μs - 20μs. With that we are well competitive.

C. Resources

TABLE IV lists the resource usage of our prototype in both absolute and relative terms. Network and memory interfaces are existing Xilinx IP cores while the TCP/IP stack is the work covered in this paper. Combined, the design utilizes around 83,000 Flip-Flop's and 68,000 LUT's respectively which equates to 9.6% and 15.6% of the overall available resources. All our data structures are using BRAMs with mostly linear dependency on the number of supported sessions with a small exception of the FIFO queues within the data-flow architecture. For 10,000 sessions, this amounts to 21.9%. Given this, we would argue that the design leaves sufficient space on the given device to implement large-scale applications even for a large amount of concurrently active sessions. Conversely, we can calculate an upper bound for number of supported sessions for a XC7VX690T device (as used on the VC709) with 1470 36 kb BRAM blocks and a XCVU190 [19] with 3780 36 kb BRAM blocks to be 44,148 and 115,665 sessions respectively.

VI. Conclusion

In this work we have presented a novel architecture implementing an entire TCP/IP stack on a programmable device. This architecture is designed for processing 10 Gbps data full-duplex, while handling thousands of concurrent sessions. The architecture's resource requirements scale linearly with the number of supported sessions to over 115,000 given today's 20 nm devices. The design was implemented almost entirely using C++, which shortened development significantly, simplified verification and provides greater design flexibility. The implementation encompasses flow control and out-of-order processing. Finally, we evaluated our design on a Xilinx VC709 development platform using a combination of software tests, proving performance, compatibility and robustness.
Future work includes adding support for key TCP options, more elaborate flow control interfaces and caching of selective session packet buffers in on-chip RAM.

References

[1] A. P. Foong, T. R. Huff, H. H. Hum, J. P. Patwardhan, and G. J. Regnier, "Tcp performance re-visited," in *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*. IEEE, 2003, pp. 70–79.

[2] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing soc accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 36–47.

[3] L. G. B. Wheeler and J. Bolaria, "A guide to 10g ethernet controllers and adapeters, fourth edition," 2010.

[4] https://www.plda.com/products/fpga-ip/xilinx/fpga-ip-tcpip/quicktcp-xilinx/.

[5] 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.

[6] http://www.intilop.com/tcpipengines.php/.

[7] http://www.dinigroup.com/new/TOE.php/.

[8] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers, "A low-latency library in fpga hardware for high-frequency trading (hft)," in *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*. IEEE, 2012, pp. 9–16.

[9] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10gbps line-rate key-value stores with fpgas," in *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[10] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java message service," *Sun Microsystems Inc., Santa Clara, CA*, 2002.

[11] A. Dollas, I. Ermis, I. Koidis, I. Zisis, and C. Kachris, "An open tcp/ip core for reconfigurable logic," in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, April 2005, pp. 297–298.

[12] Z.-Z. Wu and H.-C. Chen, "Design and implementation of tcp/ip offload engine system over gigabit ethernet," in *Computer Communications and Networks, 2006. ICCCN 2006. Proceedings.15th International Conference on*, Oct 2006, pp. 245–250.

[13] S.-M. Chung, C.-Y. Li, H.-H. Lee, J.-H. Li, Y.-C. Tsai, and C.-C. Chen, "Design and implementation of the high speed tcp/ip offload engine," in *Communications and Information Technologies, 2007. ISCIT '07. International Symposium on*, Oct 2007, pp. 574–579.

[14] T. Uchida, "Hardware-based tcp processor for gigabit ethernet," *Nuclear Science, IEEE Transactions on*, vol. 55, no. 3, pp. 1631–1637, June 2008.

[15] Y. Ji and Q.-S. Hu, "40gbps multi-connection tcp/ip offload engine," in *Wireless Communications and Signal Processing (WCSP), 2011 International Conference on*, Nov 2011, pp. 1–5.

[16] M. Perrett and I. Darwazeh, "A simple ethernet stack implementation in vhdl to enable fpga logic reconfigurability," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, Nov 2011, pp. 286–290.

[17] F. Herrmann, G. Perin, J. de Freitas, R. Bertagnolli, and J. dos Santos Martins, "A gigabit udp/ip network stack in fpga," in *Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on*, Dec 2009, pp. 836–839.

[18] C. de Schryver, P. Schlafer, N. Wehn, T. Fischer, and A. Poetzsch-Heffter, "Loopy — an open-source tcp/ip rapid prototyping and validation framework," in *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, Dec 2013, pp. 1–6.

[19] http://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale.html/.

[20] W. Jiang, "Scalable ternary content addressable memory implementation using fpgas," in *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, Oct 2013, pp. 71–82.

[21] J. Postel, "Transmission control protocol," Internet Engineering Task Force, RFC 793, June 1981. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[22] C. Neely, G. Brebner, and W. Shang, "Flexible and modular support for timing functions in high performance networking acceleration," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 513–518.

[23] V. Paxson, A. M., C. J., and S. M., "Computing tcp's retransmission timer," Internet Engineering Task Force, RFC 6298, September 2001. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6298.txt

[24] B. E., "Requirements for internet hosts - communication layers," Internet Engineering Task Force, RFC 1122, January 1989. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1122.txt

[25] A. M., V. Paxson, and B. E., "Tcp congestion control," Internet Engineering Task Force, RFC 5681, September 2009. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5681.txt

[26] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing tcp's initial congestion window," *ACM SIGCOMM Computer Communications Review*, vol. 40, pp. 27–33, 2010. [Online]. Available: http://ccr.sigcomm.org/drupal/?q=node/621

[27] J. Nagle, "Congestion control in ip/tcp internetworks," Internet Engineering Task Force, RFC 896, January 1984. [Online]. Available: http://www.rfc-editor.org/rfc/rfc896.txt

[28] A. M., V. Paxson, and S. W., "Computing tcp's retransmission timer," Internet Engineering Task Force, RFC 2581, April 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2581.txt

[29] Xilinx Inc., *Vivado Design Suite User Guide: High-Level Synthesis*.

[30] http://www.xilinx.com/tools/cspro.htm.