Low-Latency TCP/IP Stack for Data Center Applications

David Sidler   Zsolt István   Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zürich
{firstname.lastname}@inf.ethz.ch

Abstract—**TCP/IP is widely used both in the Internet as well as in data centers. The protocol makes very few assumptions about the underlying network and provides useful guarantees such as reliable transmission, in-order delivery, or control flow. The price for this functionality is complexity, latency, and computational overhead, which is especially pronounced in software implementations. While for Internet communication this is acceptable, the overhead is too high in data centers. In this paper, we explore how to optimize a TCP/IP stack running on an FPGA for data center applications with an emphasis on data processing (e.g., key value stores). Using a key-value store and a low-latency consensus protocol implemented on an FPGA as an example of the requirements that arise in data centers, we provide an extensive analysis of the overheads of TCP/IP and the solutions that can be adopted to minimize such an overhead. The proposed optimized TCP/IP stack minimizes tail latencies (a key metric in distributed data processing) and is efficiently implemented so as to be able to share the FPGA with application logic.**

I. Introduction

TCP/IP provides reliable transmission, in-order delivery, and control flow over unreliable networks. This functionality makes the protocol specification and implementation very complex. Software based network stack implementations introduce a significant latency overhead and also consume a non negligible part of the available compute resources for which they might compete with the application. One way to overcome those issues is the use of network cards which provide partial or full TCP offload to reduce the load on the host machine. For applications on reconfigurable hardware there are several commercial [1], [2], [3], [4] TCP/IP implementations available. Most of them targeting ultra low-latency applications such as high-frequency trading. As a result those stacks only support a limited number of concurrent connections, typically between 64-128. In the academic field, [5] presented a TCP/IP stack with limited throughput (350 Mbps) and connection (32) support. [6], [7] presented a hybrid hardware-software implementation of a TCP/IP stack using an FPGA with an embedded PowerPC CPU. A network stack targeted for embedded devices by [8] focuses on low resource usage by limiting the feature set. The TCP/IP stack in [9], is able to consume data at 4 Gbps and generate it at 40 Gbps, targeting asymmetric workloads like video on demand.

In earlier work [10] we presented a TCP/IP stack supporting thousands of concurrent connections at 10 Gbps line-rate. The goal was to enable data center applications serving thousands of clients concurrently. This original implementation followed the TCP specification as close as possible. In this paper we present an optimized version of this TCP/IP stack for data center applications. The optimized network stack targets two types of applications. The first one is a client facing application, such as a key-value store [11], [12]. The second one is an FPGA-based distributed system, for instance a consensus protocol [13], which uses peer-to-peer communication between the FPGA boards.

When integrating these types of applications with the network stack, two challenges have to be addressed. First, the memory bandwidth available on an FPGA board, in our case a Xilinx VC709, has to be shared among the application and the TCP/IP stack. The original version of the stack uses DDR memory to buffer the payload of incoming and outgoing packets. When adding an application which also requires access to the DDR memory, the memory bandwidth can easily become a bottleneck. Second, although the existing hardware implementation has a path latency of 1-5μs depending on packet size, data center applications are very sensitive to latencies. Generally in data center applications, the service provider and the client have a service level agreement which clearly defines median and, especially, tail latencies.

An additional issue we observed when using the TCP/IP stack for inter FPGA communication was that, despite having a low latency implementation in hardware, latency is introduced by the TCP protocol in form of Delayed Acknowledgments, conservative timers, and Nagle's algorithm.

Data center networks are very controlled environments and provide much more guarantees than the Internet. Recent work [14], [15] has shown that data center networks exhibit infrequent reordering of messages, have a fixed length network topology, and provide high reliability. In such a network, some functionality of the TCP protocol which is based on conservative assumptions about the underlying network can easily be relaxed without giving up any guarantees of TCP. By relaxing or disabling certain features latency and throughput of data center applications can be significantly improved.

In this paper we present the design of an optimized TCP/IP stack that:

- Reduces the latency of the given stack further to facilitate latency sensitive data center applications.
- Uses application and network knowledge to optimize TCP protocol mechanisms without losing any functionality.
- Reduces the required memory bandwidth to DDR, thereby freeing up resources and memory for the application using the network stack.

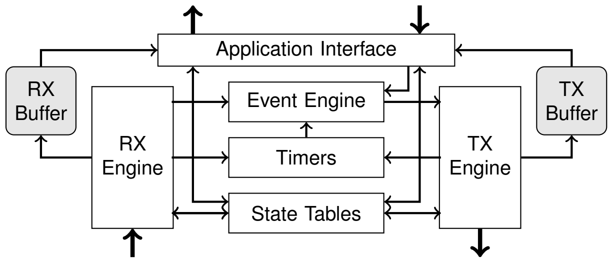

Fig. 1. Block diagram of the original TCP module

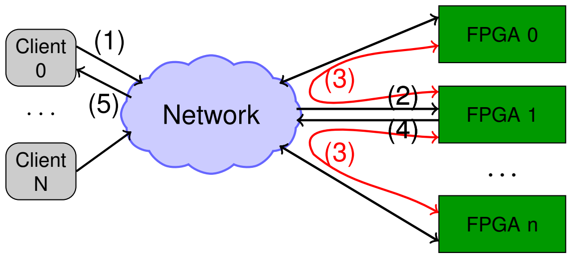

Fig. 2. Application scenario: (1) the client sends a request to the application, (2) an FPGA in the back-end receives the client request, (3) request processing might lead to inter FPGA communication, (4) the FPGA sends a response back to the client, (5) the client receives the response.

II. Background

A. Network Stack

The network stack presented in this work is based on earlier work[10] focused on maintaining 10 Gbps line-rate throughput while supporting thousands of concurrent connections. The implementation adhered to the TCP protocol specification and did not make any assumptions about the network or application. In contrast, the network stack presented in this paper is optimized for data center applications. The focus of this implementation is on reducing latency and minimizing access to DDR memory by tailoring the stack to data processing applications in a data center environment. In this section a short overview of the original architecture is given, followed by the optimizations for data center applications in the next section.

Fig. 1 shows how the TCP/IP stack is divided into three parts, the two data paths, RX and TX path, and the state-keeping data structures in the center. The two data paths are heavily pipelined and can process data at line-rate. Each path has its own payload buffer. To simplify memory management, a fixed size 64 KB buffer is allocated for each connection. In the case of supporting 10,000 concurrent connections this adds up to 1.3 GB of required memory. Since this amount does not fit into the on-chip BRAM, the RX and TX buffers are allocated in DDR memory. In the center of the architecture are multiple data structures. The State Tables keep track of the TCP connection state and windows. The timers are necessary for retransmission, probing, and connection time-outs. The Event Engine aggregates events from different sources, RX Engine, Timers, and Application, and forwards them to the TX Engine. The TX Engine produces packets based on those events and sends them to the network.

A key characteristic of this architecture is the clear separation between the two data paths and the data structures which keep the state of each TCP connection. This allows the data paths to operate and access the data structures independently without interfering with each other. As a result, full-duplex line-rate throughput is achievable. A drawback of this architecture is the amount of DDR memory and memory bandwidth required, since these two resources are scarce on most FPGA boards. Because on each path the packet is read and written to/from DDR, the required memory bandwidth with full duplex at 10 Gbps is roughly 40 Gbps. As we show in section IV, a single DDR module on the Xilinx VC709 board cannot provide the required memory bandwidth.

B. Application Setup

Fig.2 shows an example deployment of multiple FPGAs in the data center. The FPGAs are connected over the network to each other and to the clients. Apart from standalone applications like the key-value store, such a deployment also enables FPGA-based distributed systems as presented by [16], [13]. As illustrated in Fig.2, multiple clients can send requests to the back-end where each request, depending on the application, can lead to data exchange between the FPGAs. To keep the total latency in the back-end low, the latency between the FPGAs is crucial.

III. Tailoring the Network Stack

Using knowledge about the application and the network infrastructure allows us to make certain assumptions. For the application we can make three assumptions. First, client requests fit inside a *maximum transfer unit* (MTU). An MTU equals to the 1500 B size limit of Ethernet frames, respectively 9000 B for jumbo frames. Second, clients are synchronous which means they do not have more than one outstanding request. Third, the application also implemented on the FPGA is designed to process data at line-rate. These assumptions might not hold for inter FPGA communication where multiple outstanding requests and large chunks of data spreading over multiple Ethernet frames can be exchanged. For the data center network, we assume high reliability and infrequent reordering of messages, as shown by [14], [15].

A. Nagle's algorithm

The goal of Nagle's algorithm [17] is to optimize the network bandwidth utilization. The TCP/IP headers are at least 40 B in size and introduce some overhead in terms of bandwidth, since they are transmitted with each TCP segment. To reduce the overhead, Nagle's algorithm tries to maximize the amount of payload transmitted with each segment. The algorithm works as follows: if the payload to be transmitted is smaller than an MTU, it holds the data for a short amount of time in the hope that more payload will be transmitted on the same connection immediately. Except for the case that no unacknowledged data is in flight. Then data is transmitted immediately independent of its size. It is commonly known [18] that, depending on the application, Nagle's algorithm can interact badly with the mechanism of Delayed Acknowledgments. Therefore Nagle's algorithm can be disabled in most software stacks through the TCP_NODELAY flag [19], [20]. For client communication, based on our assumptions, Nagle's algorithm is not triggered. For inter FPGA communication, it is beneficial to disable the algorithm to reduce latency and avoid negative interaction with Delayed Acknowledgments. As a result, in our optimized stack we removed Nagle's algorithm completely, thereby also saving hardware resources.

B. Delayed Acknowledgment

Similar to Nagle's algorithm, the Delayed Acknowledgment described in RFC1122 [21] is a mechanism to increase the network bandwidth utilization. In TCP, data packets get acknowledged with an ACK packet, a simple control packet which contains no payload. However, ACKs share the available bandwidth with packets containing data. Since any other packet belonging to the same connection also carries the ACK number, sending of an ACK can be avoided if the ACK can be piggybacked with another packet. The mechanism of Delayed Acknowledgments delays ACKs up to 0.5 seconds in the hope the ACK number can be either sent with a data packet or merged with a later generated ACK. In the context of inter FPGA communication, we considered removing the Delayed Acknowledgments logic completely. However, experiments have shown a significant reduction in goodput, especially for small packets which are common in a distributed application where many control messages are exchanged. Instead of removing the delay we reduced it to 64 μs, expecting a reply from the application to piggyback the ACK within $5\mu s$.

C. Retransmission- & Probe-Timers

The original TCP specification has defined very conservative time-out values for retransmission, probing, and connection time-out. These values were defined for unreliable and slow wide-area networks (e.g. the Internet). In contrast, data centers deploy reliable high-speed links which lead to more predictable RTTs. To reduce the response time to failures, we adapted the time-out values according to our infrastructure.

D. On-chip RX buffer

Based on our assumptions about the client requests and that the application can process data at line-rate, the amount of buffer space on the RX path can be reduced significantly. As mentioned, inter FPGA communication can contain data chunks larger than an MTU. The TCP/IP stack enforces the ordering of the segments, but reassembly of the segments has to be done by the application itself. The additional buffer space in the application to accumulate these larger data chunks incurs a minimal resource overhead which scales linearly with the number of inter FPGA connections.

E. Reducing memory access to the TX buffer

Since for the optimized stack Nagle's algorithm was removed, there is no requirement to buffer on the TX path to accumulate enough payload to fill a segment up to the size of an MTU. The TX Buffers are solely required for retransmission of data in case of a failure. As a consequence, on transmission, payload can be directly forwarded to the TX Engine after it is written into DDR memory. On retransmission, the payload only has to be read from the DDR memory, no write is needed. As a result, to generate a data packet in the TX Engine, DDR memory is either written or read which bounds the required memory bandwidth to 10 Gbps, half the bandwidth of the original version. In addition to reducing the memory bandwidth, directly forwarding the payload to the TX Engine also reduces the latency on the TX path.

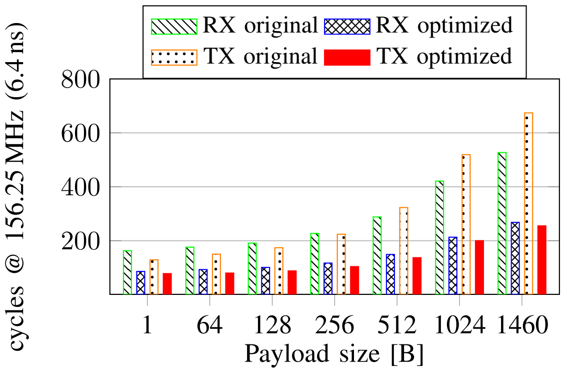

Fig. 3. Latency improvement on data paths

IV. Evaluation

The original and optimized network stack were deployed on a Xilinx VC709 evaluation board. The board is equipped with a Virtex 7 X690T, two DDR3 SODIMMs with 4 GB each, and four 10 G network interfaces of which one was used. In the experimental setup, the VC709 is connected to a 10 G Cisco Nexus 5596UP switch. Connected to the same switch are 8 server machines, each equipped with dual Xeon E5-2630 v3, 256 GB of main memory, and an Intel 82599ES 10 G network card. In our experiments, a basic echo server is implemented and connected to the network stack on the FPGA. Each server machine is running 100 clients to generate load.

A. Reduced latency on data paths

First we compare the latency of the original and optimized network stack without load. The measurement was done on the hardware itself. Fig. 3 shows the latency in clock cycles at 156.25 MHz (6.4 ns) with increasing packet size. The checksum computation on each path requires a store-and-forward of the complete segment, therefore the latency increases linearly with the segment size. The original implementation required two store-and-forwards on the TX path, as can be seen in the figure by the quicker growth of the TX latency in comparison to the RX latency. In the optimized version we were able to remove one of these store-and-forwards. Subtracting the cycles for the store-and-forward from the RX and TX path measurements of the optimized stack leads to a constant processing overhead of 85 cycles on the RX path and 70 cycles on the TX path. Overall, the gain is between 2x and 3x in terms of latency.

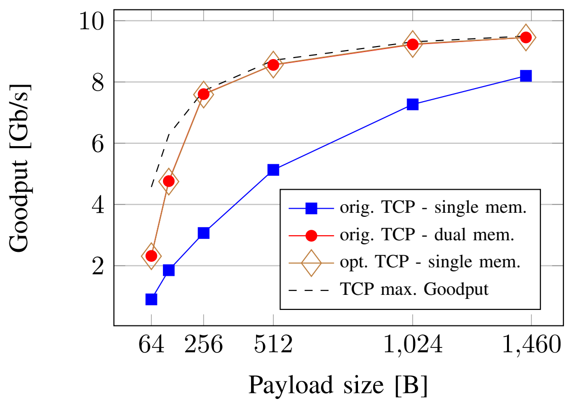

Fig. 4. Comparing goodput with varying payload size.

TABLE I
RESOURCE REQUIREMENTS ON VC709 FOR 10,000 CONNECTIONS

	FF	LUT	BRAM	Mem. alloc.	Mem. bw.
TCP/IP org.	27,189	25,364	342	1,300 MB	40 Gbps
TCP/IP opt.	24,744	22,468	353	650 MB	10 Gbps
Diff to org.	**-9.0%**	**-1.4%**	**+3.2%**	**-50%**	**-75%**

B. Improved throughput by avoiding DDR memory access

The experiment compares the original memory bandwidth used and the required memory bandwidth in the optimized stack. We compare three different setups, the original version where the RX and TX buffer share a single memory controller, the original version where the RX and TX buffer each have their own memory controller, and the optimized version where only the TX buffer is in DDR memory. The optimized version further reduces memory bandwidth by either only writing the data to the buffer in case of transmission or only reading it from the buffer in case of retransmission. Fig. 4 shows how the original TCP stack is clearly memory bound when attached to a single memory controller. The original stack using two memory controllers, one for each data path, and the optimized version, which only stores the TX buffers in DDR, achieve the same throughput. Both implementations are network bound for payload sizes of 256 B and larger, as can be seen by the dashed line which represents the theoretical maximum goodput. For smaller payloads the throughput is bound by the load generating server machines.

C. Resources

TABLE I compares the resource usage between the original version of the network stack and the optimized version. In the table we do not list the resource consumption of the network and memory interfaces, since they both remain in the design. We expect that the freed up memory controller will be used by the application. Logic consumption in the optimized version is slightly lower due to removal of the RX buffer interface. On the other hand more BRAMs are required, since the RX buffer was moved from DDR into BRAM. This also cuts the fix allocated memory in the DDR in half. Since in the original stack every packet was read and written to the DDR memory on the RX and TX path, this potentially required up to 40 Gbps of memory bandwidth. Due to the removal of DDR on the RX path and optimizations on TX path, data is either written or read from DDR, the required memory bandwidth cannot exceed 10 Gbps.

V. CONCLUSION

In this work we have presented a low-latency TCP/IP stack for data center applications. We were able to reduce the latency by making some assumptions about the application and the underlying network. The main assumption is that client requests always fit into a single Ethernet frame. Based on this the buffering of payload on the RX and TX path was heavily optimized to reduce latencies on the paths, but also reducing memory access. Due to limited memory bandwidth on most FPGA boards the latter is a requirement to deploy complex applications on the same chip as the TCP/IP stack. The improvements introduced in this paper showed a reduced latency of 2-3x on each data path.

ACKNOWLEDGMENT

This research is funded in part through an equipment grant from Xilinx, Inc.

REFERENCES

[1] https://www.plda.com/products/fpga-ip/xilinx/fpga-ip-tcpip/quicktcp-xilinx/.
[2] U. Langenbach, A. Berthe *et al.*, "A 10 GbE TCP/IP hardware stack as part of a protocol acceleration platform," in *ICCE-Berlin'13*.
[3] http://www.intilop.com/tcpipengines.php/.
[4] http://www.dinigroup.com/new/TOE.php/.
[5] A. Dollas, I. Ermis, I. Koidis, I. Zisis, and C. Kachris, "An open TCP/IP core for reconfigurable logic," in *FCCM'05*.
[6] Z.-Z. Wu and H.-C. Chen, "Design and implementation of TCP/IP offload engine system over Gigabit Ethernet," in *ICCCN'06*.
[7] S.-M. Chung, C.-Y. Li, and other, "Design and implementation of the high speed TCP/IP offload engine," in *ISCIT '07*.
[8] T. Uchida, "Hardware-based TCP processor for Gigabit Ethernet," *Nuclear Science, IEEE Transactions on*, vol. 55, no. 3, June 2008.
[9] Y. Ji and Q.-S. Hu, "40Gbps multi-connection TCP/IP offload engine," in *WCSP'11*.
[10] D. Sidler, G. Alonso, M. Blott, K. Karras *et al.*, "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware," in *FCCM'15*.
[11] Z. Istvan, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10Gbps key-value stores on FPGAS," in *FPL'13*.
[12] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA memcached appliance," in *FPGA'13*.
[13] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *NSDI'16*.
[14] D. R. K. Ports, J. Li, V. Liu, and other, "Designing distributed systems using approximate synchrony in data center networks," in *NSDI'15*.
[15] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "NetPaxos: Consensus at network speed," in *SOSR'15*.
[16] A. Putnam, A. Caulfield, E. Chung *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA'14*.
[17] N. J., "Congestion control in IP/TCP internetworks," Internet Engineering Task Force, RFC 896, January 1984. [Online]. Available: http://www.rfc-editor.org/rfc/rfc896.txt
[18] S. Cheshire, "TCP performance problems caused by interaction between Nagles algorithm and Delayed ACK," http://www.stuartcheshire.org/papers/NagleDelayedAck, 2005.
[19] "tcp(7): TCP protocol – linux man page," http://linux.die.net/man/7/tcp.
[20] "setsockopt function (Windows)," https://msdn.microsoft.com/en-us/library/windows/desktop/ms740476%28v=vs.85%29.aspx.
[21] 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