EFFICIENT PC-FPGA COMMUNICATION OVER GIGABIT ETHERNET

*Nikolaos Alachiotis, Simon A. Berger, Alexandros Stamatakis**

The Exelixis Lab
Department of Computer Science
Technische Universität München
email: {alachiot,bergers,stamatak}@in.tum.de

ABSTRACT

As FPGAs become larger and more powerful, they are increasingly used as accelerator devices for compute-intensive functions. Input/Output (I/O) speeds can become a bottleneck and directly affect the performance of a reconfigurable accelerator since the chip will idle when there are no data available. While PCI Express represents the currently fastest and most expensive solution to connect a FPGA to a general purpose CPU, there exist several applications with I/O requirements for which Gigabit Ethernet is sufficient.

To this end, we present the design of an efficient UDP/IP core for PC-FPGA communication that has been designed to occupy a minimum amount of hardware resources on the FPGA. An observation regarding the internet checksum algorithm, allows us to reduce the hardware requirements for computing the checksum. Furthermore, this property also allows for initiating packet transmission immediately, i.e., the UDP/IP core can start a transmission without the requirement of receiving, storing, and processing user data beforehand.

The UDP/IP core is available as open-source code. A comparison with related work on UDP/IP core implementations shows that our implementation is significantly more efficient in terms of resource utilization *and* performance. The experimental results were obtained on a real-world system and we also make available the PC software test application that is used for performance assessment to allow for reproduction of our results.

1. INTRODUCTION

A general problem of FPGAs is, that they often suffer from Input/Output (I/O) bottlenecks, that is, their full processing potential can not be exploited because of relatively slow communication with the outer world. The outer world will typically be a general purpose CPU that invokes compute intensive functions on the FPGA.

*This work is funded under the auspices of the Emmy-Noether program by the German Science Foundation (DFG)

As FPGAs become more powerful in terms of available reconfigurable hardware resources, they are increasingly used as accelerator devices, for instance because of their inherent ability to process more data in parallel. A common trend is to connect a FPGA-based board to a general purpose processor, usually a personal computer (PC), for offloading computationally intensive parts or functions of an application to the FPGA. Function offloading to the FPGA can thus significantly speed up the execution of scientific codes for instance. To exploit the full acceleration potential of a reconfigurable architecture, the time the FPGA idles because it is waiting for an I/O operation should be minimized.

The fastest *and* most expensive interconnect technology currently available for FPGAs is PCI Express. While PCI Express appears to represent the ideal solution for communication between a host PC and a board, several drawbacks exist. A dedicated PCI Express driver is required on the PC side as well as a PCI Express enabled FPGA board. Only recently, Xilinx made available a reference design for Endpoint PCI Express solutions [1] which also includes PC drivers for Linux and Windows operating systems. Similar PCI Express-based solutions are available for purchase as commercial IP cores. In terms of programming overhead, an Ethernet-based solution may be preferable due to only a few lines of code are required to access the Ethernet port of a PC. To this end, we present the implementation of a hardware unit that allows for transmission of UDP (User Datagram Protocol) [2] datagrams that are encapsulated into Ethernet packets which comply with the 4th revision of the Internet Protocol [3].

The Internet Protocol version 4 (IPv4) is the most widely used Internet Layer protocol. We find that, the combination of IPv4 with UDP, represents the optimal solution in terms of hardware resource requirements for data transmission between a host PC and a FPGA board. Because of the comparatively small protocol overhead, UDP allows for high transmission rates while—at the same time—requiring low programming overhead on the PC side, i.e., programming interfaces for UDP are well documented and readily available for all common operating systems.

Most modern FPGAs contain EMAC (Ethernet Media Access Controller) blocks that allow for direct access to external physical layer (PHY) devices on the board. A PHY is required for the EMAC to connect to an external device, for example, a network, a host PC, or another FPGA. One can for instance deploy the Xilinx Core generator to configure and generate EMAC wrapper files that contain a user-configurable Ethernet MAC physical interface, e.g., MII, GMII, SGMII, and to instantiate RocketIO serial transceivers, clock buffers, DCMs etc. Xilinx also provides an optimized clocking scheme for the physical interface as well as a simple FIFO-loopback example design which is connected to the EMAC client interface.

Although the EMAC wrapper files greatly simplify the usage of the EMAC, extra logic is required to create packets that comply with Transport Layer protocols and that will be accepted by the Linux kernel. The Packet Transmitter Unit (PTU) we present here can be connected directly to the EMAC client interface, i.e., it can be used to replace the FIFO-loopback example design, and also allows for encapsulating UDP packets within IPv4 packets (UDP/IP) by using a very small fraction of hardware resources. On an average-sized FPGA like the Virtex 5 SX95T, the UDP/IP core occupies less than 1% of the available slices and can operate at a frequency exceeding 125MHz, which is a prerequisite for achieving Gigabit speed.

The hardware unit is freely available for download as open source code at: `http://wwwkramer.in.tum.de/exelixis/countIPv4.php` and `OpenCores.org` (project name: *IPv4 packet creator and transmitter*). The number of downloads since the first release (January 07, 2010) amounts to 189. The JAVA test-applications we used for performance assessment are also included in these archives.

The remainder of this paper is organized as follows: Section 2 describes the observation and underlying concept that allowed us to minimize the required hardware resources and to thereby singinificantly increase transmission speed. In Section 3 we review related work on UDP/IP and TCP/IP core implementations. The hardware architecture is described in Section 4. In the following Section 5, we present details regarding the verification process (5.1), the setup of the Ethernet-based communication platform (5.2), the computational experiments (5.2.1 and 5.2.2), and provide information about resource utilization (5.3). In Section 5.3 we also conduct a performance comparison with a commercially available UDP/IP core [4]. We conclude in Section 6.

2. MINIMIZING TRANSMISSION COST

The unit for transmitting UDP IPv4 Ethernet packets has been designed and implemented within the framework of a larger project that focuses on the implementation of a reconfigurable accelerator [5, 6] that shall serve as a co-processor for likelihood-based phylogenetic inference programs (programs for reconstruction of evolutionary trees from molecular data) such as RAxML [7]. To minimize the FPGA resources used for Ethernet communication, such that more reconfigurable logic is available for the implementation of the likelihood function accelerator modules, we assume that errors in the data field occur so rarely, that a hardware implementation of an error-recovery unit does not justify the allocation of additional FPGA resources. Since the solution proposed here is geared towards direct communication between a PC and a FPGA through an Ethernet cable, our assumption is supported by the high experimental transmission success ratios that are provided in Section 5.2.

The underlying idea of our implementation is to deploy a LUT (Look Up Table) that contains all static fields required by the 802.3 MAC frame, as well as the IPv4, and UDP protocols. Although a LUT for the static fields has been used before [4], we extend this approach in a way that minimizes the required hardware resources. Furthermore, our approach allows for immediate start of transmission, i.e., once a user-send-request arrives at the input port of the packet transmitter, the transmission of a packet can already commence during the next clock cycle.

In the following, we describe how the LUT is initialized (2.1), how the IPv4 header checksum is calculated (2.2), and how a packet is transmitted (2.3).

2.1. LUT initialization

As already pointed out, the LUT is used to store the static fields of the packet to be transmitted. These fields include the destination MAC address, the source MAC address, and the Ethertype; these are the main fields that form part of the 802.3 MAC frame. We also store the header fields of the IPv4 and UDP header sections in the LUT.

The IPv4 header section contains the following fields: Version, Header Length, Differentiated Services, Total Length, Identification, Flags, Fragment Offset, Time to Live, Protocol, Header Checksum, Source Address, and Destination Address [3]. If the *Total Length* and *Header Checksum* fields are excluded, all remaining fields are considered to be static, since our focus is on a direct fast connection between a PC and a FPGA. The UDP header section consists of the Source Port, the Destination Port, the Length, and the Checksum [2]. As before, if the *Length* and *Checksum* fields are excluded, the *Port* fields can be considered as being static.

All of the above static fields, are used to initialize the LUT during the memory configuration process. We retrieve the values of these fields once, by sending a *basic* packet from the PC to the FPGA. By *basic*, we refer to a packet that does not contain any data, that is, the *Total Length* field of the IPv4 packet only refers to the length of the header fields. The rationale for choosing this approach is provided in Section 2.2. We used the Chipscope Pro Analyzer [8] to

monitor the data of the *basic* packet. As already mentioned, it is sent only once, and since it is sent from the PC to the FPGA, we need to switch the order of the address and port fields prior to the initialization of the LUT. By interchanging the values of those two fields, the data in the LUT now correspond to a transmission from the FPGA to the PC.

2.2. Checksum calculation

As mentioned in Subsection 2.1, initially a *basic* packet is generated and sent to the FPGA in order to retrieve the static fields. Extra logic is required to calculate the values of the non-static fields, that is, the *Length* and *Checksum* fields. An adder is deployed to add the user data length to the constant header length; a subtracter is required for calculating the IPv4 checksum (see below).

The IPv4 protocol only provides checksum-based header integrity, i.e., the computation of the checksum is conducted on the header fields only. Since all fields in the header section, except for *Total Length*, are constant, we investigated how the IPv4 checksum field changes as a function of the *Total Length* field.

The standard method to compute the checksum is defined within RFC 791 [3]; it is defined as the 16-bit one's complement of the one's complement sum of all 16-bit words in the header. During the computation of the header checksum, the actual value of the checksum field is set to zero.

The term *one's complement sum* is often confused with *one's complement addition*. While a one's complement addition simply requires the sum of the one's complement values of the numbers to be added, the one's complement sum is calculated as follows: Each time a carry-out, i.e, a 17th bit, is produced, this bit should be added to the LSB (Least Significant Bit). Figure 1 depicts an example for calculating the one's complement sum of three 16-bit values. During the first addition, a carry-out is produced and added to the LSB. During the second addition, no carry-out is produced, thus the one's complement sum has already been calculated.

The above example shows that associativity does not hold calculating the one's complement sum. In other words, the result depends on the order of the operands. The calculation of the header checksum can be optimized by considering the specific position of the *Total Length* field in the IPv4 header section. As described in the IPv4 protocol [3], the *Total Length* field is placed after the following fields: *Version*, *Header Length*, and *Differentiated Services*. This position corresponds to the second 16-bit operand, since the *Version* and *Header Length* fields occupy 4 bits and the *Differentiated Services* field 8 bits. Thus, the *Total Length* field, is the second operand of the addition that needs to be initially carried out, to then obtain the one's complement sum of the 16-bit values in the header section. We take advantage of the fact, that no carry-out can be produced by the first addition and calculate the value of the header checksum by subtracting the *Total Length* from the checksum value of the aforementioned *basic* packet. The first addition never produces a carry-out, because the maximum length of every transmitted packet does never require more than 12 bits. Thus, the initial 16-bit addition can not overflow.

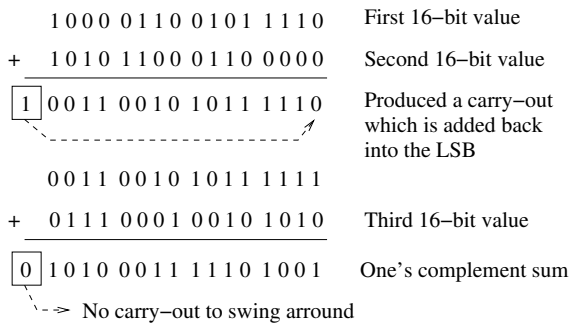

Fig. 1. Calculation of one's complement sum of three 16-bit values. (*http://mathforum.org/library/drmath/view/54379.html*)

The rationale for only using a single subtraction is, that the *base* checksum value of the *basic* packet corresponds to a checksum that was calculated using a *reduced Total Length* value for the first addition that is required by the calculation of the one's complement sum. This *reduced* value does not include the length of the user-data that is attached to the packet. Since the first addition does not produce a carry-out, we can correct the calculated value of the header checksum, which is already a one's complement of the one's complement sum, by simply subtracting the user-data length.

The UDP checksum can simply be set to zero without the packet being rejected by the receiver [2], as long as the header integrity of the IPv4 packet is maintained via a correct IPv4 checksum. UDP assumes that error checking and correction is either not necessary or performed at the application layer, thus avoiding error handling overhead at the network interface level.

2.3. Temporal Work-Flow of a Transmission

The resource-efficient (in terms of hardware resources) method we propose for establishing point-to-point communication between a PC and a FPGA is illustrated in Figure 2, while Figure 3 depicts the temporal work-flow of a transmission.

Constant header fields that are stored in the LUT are transmitted first during the time interval $[t - t_{L1})$. This first LUT-content transmission phase requires 16 clock cycles. In parallel to this transmission, the *Total Length* (IPv4) and *Length* (UDP) fields are calculated by the respective adders, and the *Header Checksum* (IPv4) is computed by the subtracter. At the 17th clock cycle, the transmission of LUT contents stops, and the output of the adder that calculated the *Total Length* is transmitted (interval: $[t_{L1} - t_{A1})$). This

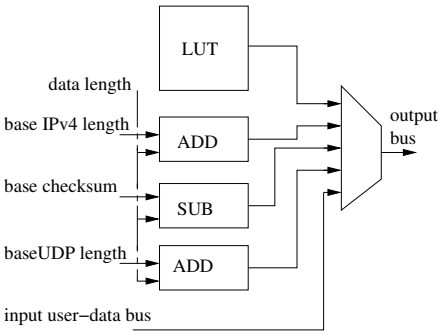

Fig. 2. Basic concept of the PTU method.

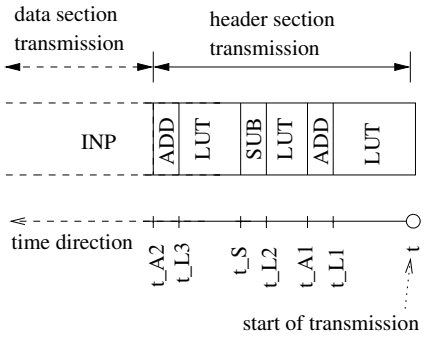

Fig. 3. Temporal Work-Flow of a Transmission.

transmission requires 2 clock cycles. Then, a LUT-content transmission phase starts again (interval: $[t_{A1} - t_{L2})$) and stops only 6 clock cycles later when the *Protocol* field has been transmitted. Thereafter, the *Header Checksum* field that was calculated by the subtracter is transmitted (interval: $[t_{L2} - t_S)$). Then, the final LUT-content transmission phase begins and several static fields are transmitted for 12 clock cycles (interval: $[t_S - t_{L3})$). When the second and final byte of the *Destination Port* has been sent, the output of the adder that contains the UDP length is transmitted (interval: $[t_{L3} - t_{A2})$). When all static and calculated fields have been transmitted, we can start sending user-data (t_{A2}). The unit assumes that whatever data arrives at the input bus is to be sent, that is, it is the user's responsibility to partition the data into bytes and send them to the transmitter unit at the correct clock cycles, i.e., any incoming data before or after the user-data transmission clock cycles will be ignored. Note that, the UDP checksum is considered as being a static field set to zero, thus zeros are transmitted during the 2 clock cycles that correspond to this field.

3. RELATED WORK

Traditional methods for PC-FPGA communication included the use of the serial (UART) and parallel ports. Because of their low transmission rates, those interfaces are frequently not available any more on modern personal computers. In principle, the same is true for current high-end FPGA boards, which are typically not equipped with serial or parallel ports any more. The old interfaces have been replaced by high-speed communication ports, such as USB 2.0, 1000 BASE-T (Gigabit Ethernet), or PCI Express.

Ethernet-based solutions appear to be the most attractive ones for PC-FPGA communication, when PCI Express-based solutions are not necessarily required to accommodate the I/O traffic of a FPGA design. Gigabit Ethernet allows for using a standard network cable, while PCI Express requires a PCI Express motherboard. In addition, Ethernet offers higher transmission rates than USB 2.0. A thorough bibliographical search revealed that several alternative designs that implement TCP/IP or UDP/IP protocols exist [4, 9, 10].

Kühn *et al.* used UDP/IP over Gigabit Ethernet [9], to establish high-speed PC-FPGA communication. The authors use an operating system (Linux) that is running on the embedded PPC (PowerPC). This represents the most straightforward solution as long as the available FPGA board is equipped with PPC blocks. However, not all FPGAs contain PPC blocks and the deployment of soft-coded processors like Microblaze solely for transmission of Ethernet packets, may lead to a large and inefficient allocation of FPGA area. Our approach allows for direct transmission of Ethernet packets using the UDP/IP core in stand-alone mode, i.e., without the need for a processor and an operating system that runs on it.

Three different stand-alone implementations of UDP/IP stack cores have previously been presented in [4]; these designs represent commercial IP cores. Löfgren *et al.* point out that when it comes to the design of FPGA-based Ethernet connected embedded systems, the priority and necessity of requirements such as cost, area, flexibility etc. varies. Because of this variation, a UDP/IP stack "template" design will thus not be suitable to accommodate distinct embedded network system requirements [4]. Therefore, they present three different implementations denoted as *minimum*, *medium*, and *advanced* UDP/IP cores. Our minimal solution occupies only 22% of the total hardware resources for the *minimum* UDP/IP core, can be used to transmit and receive packets at Gigabit speed (the *minimum* commercial IP core only operates at 10/100 Mbps), and is available for download as open-source code. More details and a resource usage comparison between our minimal approach and the *minimum* UDP/IP core are provided in Subsection 5.3. We did not conduct comparisons with the *medium* or *advanced* IP cores, since they have been designed to provide increased flexibility and also comprise implementations of additional protocols which are not necessary for establishing a direct PC-FPGA communication.

Finally, Dollas *et al.* [10] presented an architecture for an open TCP/IP core, comprising implementations of all nec-

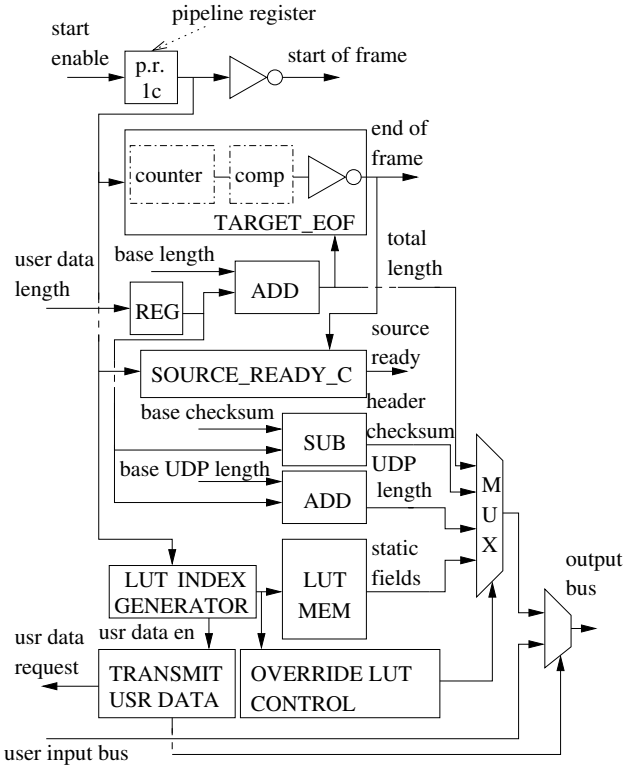

Fig. 4. Block diagram of the Packet Transmitter Unit.

essary protocols (ARP, ICMP, UDP) such that the TCP/IP stack is fully compatible with real-world applications. Due to the complexity of their architecture that is due to supporting a large number of protocols, a direct comparison to our light-weight design would be unfair, in particular with respect to resource utilization and because the design objectives of the architectures are significantly different. Furthermore, a full TCP/IP core is not necessary for direct PC-FPGA communication, since one can establish a connection using fewer protocols and therefore less hardware resources.

4. THE PACKET TRANSMITTER ARCHITECTURE

In the following we describe the design of the reconfigurable architecture that performs the transmission of UDP/IP Ethernet packets. In Figure 4 we provide the block diagram of the top-level unit.

The packet transmitter requires an active-high enable signal (*start_enable* signal in Figure 4) that needs to be sent by the application. This signal is delayed for one clock cycle and inverted in order to be compliant with the active-low *start_of_frame* signal of the EMAC. The *start_of_frame* output port of the packet transmitter should be connected to the *start_of_frame* input port of the EMAC. The delay by one clock cycle is required to provide enough time for addressing the LUT and to make available the first datum for transmission.

The TARGET_EOF module is used for generating the *end_of_frame* signal that should be connected to the respective input port of the EMAC. This module contains a 11-bit counter and a comparator. The delayed *start_enable* signal is used to initialize the counter that counts the total number of bytes that are transmitted at each clock cycle. The comparator is used to compare the number of transmitted bytes to the *Total Length* value, which represents the point in time at which the entire packet has been sent, i.e., the clock cycle during which the last byte of user data will be sent to the EMAC. Note that, the counter starts from position -X, where X is the number of bytes that are being transmitted *before* the IPv4 header section, i.e., the MAC addresses and the Ethernet Type, as required by the 802.3 MAC frame. The output signal of the comparator is inverted since EMAC expects an active-low *end_of_frame* input.

The adder (ADD) that is located right below the TARGET_EOF module is used to add a constant value (*base_length*) to the number of user data bytes (*user_data_length*). The *base_length* value corresponds to the number of bytes in the fields that precede the user data, excluding the bytes required for the 802.3 MAC frame.

The SOURCE_READY_C component sets the value of the *source_ready* signal depending on the *start_enable* and *end_of_frame* signals respectively.

A subtracter (SUB) is used to compute the header checksum, as outlined in Section 2.2. The computation of the *base checksum* value has already been described in detail in Section 2.2. This value is hard-coded in the design.

The adder (ADD) below the checksum subtracter (SUB) in Figure 4 calculates the length field of the UDP packet. The *base_UDP_length* is the constant number of bytes for the header section of the UDP frame, i.e., the Source and Destination Ports, the Length, and the Checksum.

The LUT_INDEX_GENERATOR module is used to generate the correct LUT addresses at the corresponding clock cycles. It is triggered by *start_enable* signal, which, as previously mentioned, is delayed by one clock cycle. Once the addressing process has been initiated, the counter is incremented at each clock cycle to index the next memory position in the LUT. As described in Section 2.3, some of the fields that precede the user data are not static: the IPv4 header checksum, the *Total Length* field, and the UDP length. The OVERRIDE_LUT_CONTROL component creates the appropriate selection signals for the 4_to_1 multiplexer, such that the LUT output is ignored and the non-static fields are selected and sent at the respective clock cycles.

Finally, the TRANSMIT_USR_DATA module monitors the addresses generated by the LUT_INDEX_GENERATOR and activates a *usr_data_request*. In this way, the user application is informed that the user data section of the UDP packet will start after exactly c cycles, i.e., at the cth cycle

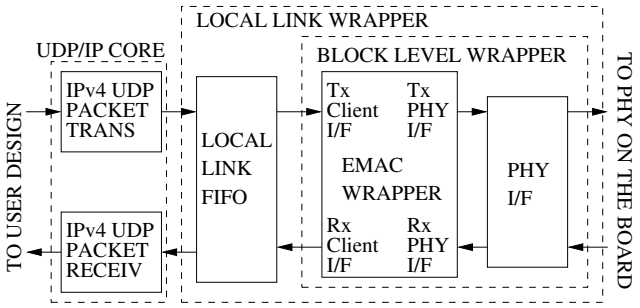

Fig. 5. Example connection of the Packet Transmitter and Receiver Units to the Xilinx Local-Link Wrapper.

after *usr_data_request* is set. When this point is reached, the packet transmitter will forward to the EMAC whatever data is received at the *user_input_bus* input port. In our experimental setup we set $c := 2$ since the user data for transmission are stored in block rams and therefore one clock cycle is required for addressing, and a second clock cycle is required for retrieving user data from the block RAM component.

Figure 5 shows at which level we connected the UDP/IP core to the EMAC, based on the wrapper files of the example design provided by Xilinx. Note that, the design of the Packet Receiver Unit (IPv4_UDP_PACKET_RECEIVER) is trivial; we therefore do not provide a block diagram for this component. The packet receiver unit consists of a counter, a comparator, and logic. When a packet arrives, the incoming bytes are counted and the comparator detects the last byte of the header section, i.e., the UDP checksum field. The header-section identification mechanism simply counts the bytes that have been received and compares the value of the byte counter to the fixed header-section length which is known a priori. When the last byte of the UDP checksum field has been received, which is the last byte of the header section, the receiver module sets the *usr_data_valid_out* signal and simply forwards the incoming bytes to the output data bus which is connected to the user design, for instance to a block RAM component.

5. EXPERIMENTAL RESULTS

Initially, we verified the functionality of the UDP/IP core (Section 5.1). Thereafter, we provide a detailed description of the experimental design and evaluate the core in terms of transmission and success rates (Section 5.2). Finally, Section 5.3 provides a performance and resource utilization comparison with the *minimum* commercial UDP/IP core [4].

5.1. Verification

In order to verify the correctness of the proposed architecture, we conducted extensive post place and route simulations as well as tests on an actual FPGA device. As simulation tool, we used Modelsim 6.3f by Mentor Graphics. For hardware verification we used the HTG-V5-PCIE development platform equipped with a Xilinx Virtex 5 SX95T-1 FPGA. The Chipscope Pro Analyzer verification tool was used to monitor the input and output ports of the EMAC Local Link Wrapper, which were directly connected to the respective ports of the UDP/IP core to track the expected signals.

5.2. Experimental Evaluation

In order to measure the transmission rate and the success rate of our Ethernet-based communication platform, we performed two experiments (Subsections 5.2.1 and 5.2.2). The FPGA board was connected to a DELL Latitude e4300 notebook, which is equipped with an Intel Core 2 Duo P9400 CPU at 2.4GHz running under Linux. A standard CAT5 twisted-pair cable was used to connect the integrated Intel 82567LM Gigabit Ethernet port of the laptop to the Gigabit Ethernet connector on the board. Since the board used for our experiments is equipped with two PHY devices, we used the connector that supports the SGMII interface and which is connected to the RocketIO GTP/GTX of the Virtex-5 FPGA. In order for the RocketIO transceiver to operate properly and allow for Gigabit speed, we generated a 125MHz external clock using the on-board oscillator that is directly connected to the RocketIO transceiver. The UDP/IP core receives a clock signal with the same frequency to smoothly operate in conjunction with the EMAC. On the PC side we used a JAVA application for transmitting and receiving UDP datagrams. The current design does not include an implementation of the ARP protocol [11]. To send UDP packets to the FPGA, a respective ARP cache entry must therefore be added manually, every time the connection is reset.

5.2.1. FPGA → PC Transmission

The aim of this first experiment was to determine what the upper bound for one-way transmission is, i.e., only the FPGA sends packets to the PC. For this experiment, the FPGA was programmed to transmit UDP datagrams with 1,472 bytes at a near-maximum rate. The size of the transmitted packets is near the maximum length of 1,518 bytes for a non-VLAN tagged frame, as specified in the IEEE STD 802.3 2002 specification [12]. The transmission rate is limited by the speed at which the packet transmitter can send packets. The implementation requires a gap of one clock cycle between two packets. However, transmission of packets at this speed can not be handled by the EMAC; as a result, most packets could not be sent properly. For this reason, we introduced a spacing of 40 clock cycles between packet transmissions.

The packets contained fixed, constant data in the data field and we also attached a serial number to each of them to

facilitate detection of packet loss and/or corruption. On the PC side, the packets were received and analyzed by a JAVA program which is based on the NIO (New I/O) API [13] that uses direct buffers to maximize performance.

The JAVA program counts the number of correctly received packets (*N_RX*) during a fixed time period of 10 seconds. The number of packets actually sent by the FPGA ($N\_TX = N\_RX + N\_LOST$) is determined by comparing the serial number of each incoming packet to the serial number of the last packet received, i.e., the packet that arrived exactly before the incoming packet. The real data rate can be calculated, based on these two values, in MB/s as $RX = (N\_RX * 1472Bytes)/(10Seconds * 10^6)$. Finally, the transmission success ratio can be derived as follows: $LR = N\_RX/N\_TX$.

Our tests show a transmission rate higher than 115.6 MB/s and a success rate ranging between 98.83% and 99.25%. The small percentage of packet loss is expected and it is due to the design of the UDP/IP standard. When the operating system or the receiving program on the PC can not process the incoming data fast enough, arriving packets are discarded. The UDP/IP standard provides no mechanism for the sender to detect this type of packet loss, so reliable data transmission has to be implemented at the application level (e.g., the receiving application can send a re-transmission request in case of a lost packet). Because of the very high success rate that exceeds 98%, the impact on application performance due to transmission reliability and infrequent re-transmission requests will be insignificant.

We also investigated the impact on transmission quality induced by increasing the time gap (number of cycles) between packet transmissions. For time gaps of 80 and 160 clock cycles, we observed a reduction of the transmission rate while the success ratio remained in the 98.83% - 99.25% range.

5.2.2. FPGA ↔ PC Transmission

In the second experiment, we focused on two-way communication between the PC and the FPGA board. The Gigabit Ethernet ports of the PC and the board can operate at full-duplex, which means that the maximum transfer rate can be simultaneously achieved by both the transmitting, as well as in the receiving device. To measure the full-duplex transfer rate we extended the previous experiment as follows:

We used two threads, a sender- and a receiver-thread. The sender-thread continously transmits a fixed number of packets (N_DTX) at maximum speed. We set $N\_DTX := 10^6$ for our experiment. The maximum speed is limited by the PC hardware components and the operating system. Once an incoming packet is registered by the FPGA, a reply-packet filled with a predefined constant data-pattern is sent back to the PC immediately, that is, the transmission of packets from the FPGA side is not continuous anymore, but is triggered by the incoming packet. On the PC side, the packet that was sent by the FPGA is received by the receiver-thread.

FPGA Resources(Total)	UDP/IP core	% Total
# Slice Registers (58,880)	79	1%
# Slice LUTs (58,880)	155	1%
# Occupied Slices (14,720)	67	1%

Table 1. Resource utilization on a Virtex 5 SX95T-1 FPGA.

Both threads measure the data rate of the outgoing and incoming packets in the same way as in the first experiment. In addition, the receiver thread counts the number of correctly received packets (N_DRX), i.e., the packets with a valid data-pattern. In analogy to the first experiment, the success rate can be calculated as $SR = N\_DRX/N\_DTX$. In contrast to one-way communication (FPGA → PC), here, we measure the success rate for sending packets back and forth between the PC and the FPGA while the connection is utilized at maximum full-duplex speed.

The sender transmission rate was limited by the PC and amounted to 113.11 MB/s, while the receiver rate was 111.67 MB/s. Both rates are close to the theoretical maximum of a Gigabit Ethernet connection which is 125MB/s and our results demonstrate that full-duplex transfer rates can be used and achieved in practice. The success rate is comparable to the success rate of the one-way experiment and ranged between 98.7% and 99.2%.

5.3. Resource utilization and Comparison

As already mentioned, the board used for verification and performance assessment of the UDP/IP core was equipped with a Virtex 5 SX95T-1 FPGA. Table 1 shows the resources occupied by the core on this FPGA.

Furthermore, since the core presented by Löfgren *et al.* [4] was mapped to a Xilinx Spartan3 XC3S200-4, we reconfigured and mapped our UDP/IP core to the same FPGA in order to conduct a fair comparison to their *minimum* implementation. The reconfigured core for Spartan3 FPGAs is also made available for download. Table 2 provides a resource consumption comparison between the *minimum* core by Löfgren *et al.* [4] and our UDP/IP core. Note that, the results that refer to our core are as they appeared in the Xilinx reports after the implementation process (post place and route) while the results that refer to the commercial core have been taken from [4]

As the results in Table 2 indicate, our component allows for PC-FPGA communication at Gigabit speed, while only utilizing a minimum of hardware resources. Our implementation occupies *significantly* less Xilinx Slices and does not use any block RAMs. The core can also operate at a frequency higher than 125MHz, which is required for operating at Gigabit speed.

	Löfgren core	our UDP/IP core
Xilinx Slices	517	111
Xilinx BRAMS	3	0
FMax(MHz)	90.7	131.8
Duplex Mode	FULL	FULL
Length(Bytes)	256	1472*
Speed(Mbps)	10/100	10/100/1000
ARP	NO	NO
RARP	NO	NO
ICMP	NO	NO
TCP"channel"	NO	NO

Table 2. Performance comparison on a Spartan3 XC3S200-4 FPGA: UDP/IP core of [4] VS our implementation. *Note that the Length(Bytes) in our implementation is limited by the EMAC configuration (Xilinx EMAC wrapper files generated by Core generator).

6. CONCLUSION & FUTURE WORK

We presented a new architecture that allows for efficient communication between a PC and a FPGA, both in terms of hardware resource utilization as well as with respect to achieving transmission rates and success ratios near the respective maxima. The underlying idea that allows for immediate transmission of packets, as soon as a *usr_send_request* arrives, consists of storing all static fields of the header section in a LUT and only using a subtracter to calculate the header checksum field based on the *Total Length* value.

The UDP/IP core is available as open source code for download at: `http://wwwkramer.in.tum.de/exelixis/countIPv4.php` or at `OpenCores.org` (project name: IPv4 packet creator and transmitter). Since the release in early January 2010 it is downloaded 6.5 times per day on average. Thus, the main contribution of our paper consists in making available an open-source, fast, and resource-efficient Gigabit Ethernet communication unit for FPGAs to the community.

Future work will focus on designing a memory hierarchy-based interface between a PC and a FPGA, that is, we intend to improve the remote memory accesses of the reconfigurable co-processor by utilizing on-board DRAM memory and providing a mechanism to transparently use *user address space* on the PC from within the FPGA.

7. REFERENCES

[1] Xilinx, "Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions," (last visited: 03-02-2010), http://www.xilinx.com/support/documentation/application_notes/xapp1052.pdf.

[2] J. Postel, "User Datagram Protocol," RFC 768 (Standard), Internet Engineering Task Force, August 1980.

[3] J. Postel, "Internet Protocol," RFC 791 (Standard), Internet Engineering Task Force, September 1981. (Updated by RFC 1349).

[4] A. Löfgren, L. Lodesten, S. Sjöholm, H. Hansson, "An analysis of FPGA-based UDP/IP stack parallelism for embedded Ethernet connectivity," in *Proceedings of the 23rd IEEE NORCHIP Conference*, pp.94-97, November 2008.

[5] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis, "Exploring FPGAs for Accelerating the Phylogenetic Likelihood Function," in *Proceedings of HICOMB2009*, held in conjuction with IPDPS, Rome, Italy, May 2009.

[6] N. Alachiotis, A. Stamatakis, E. Sotiriades, and A. Dollas, "A Reconfigurable Architecture for the Phylogenetic Likelihood Function," in *Proceedings of FPL 2009*, Prague, September 2009.

[7] A. Stamatakis, "RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," *Bioinf.*, vol. 22, no. 21, pp. 2688–2690, 2006.

[8] Xilinx, "ChipScope Pro 10.1 Software and Cores User Guide," (last visited: 03-02-2010), www.xilinx.com/ise/verification/chipscope_pro_sw_cores_10_1_ug029.pdf.

[9] W. Kühn et al.,"FPGA based compute nodes for high level triggering in PANDA," *Journal of Physics: Conference Series*, vol. 119, pp. 022-027, 2008.

[10] A. Dollas, I. Ermis, I. Koidis, I. Zisis, C. Kachris, "An Open TCP/IP Core for Reconfigurable Logic," *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pp. 297-298, 2005.

[11] D. Plummer, "Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware," RFC 826 (Standard), Internet Engineering Task Force, November 1982. (Updated by RFCs 5227,5494).

[12] The ETHERNET Working Group, "IEEE STD 802.3 2002 specification," (last visited: 03-02-2010), http://www.ieee802.org/3/.

[13] Java, "New I/O APIs," (last visited: 03-02-2010), http://java.sun.com/j2se/1.4.2/docs/guide/nio/.