Optimizing a Hardware Network Stack to Realize an In-Network ML Inference Application

Marco Hartmann, Lukas Weber, Johannes Wirth, Lukas Sommer, Andreas Koch
Embedded Systems and Applications Group, TU Darmstadt, Germany
{hartmann, weber, wirth, sommer, koch}@esa.tu-darmstadt.de

***Abstract*—FPGAs are an interesting platform for the implementation of network-attached accelerators, either in the form of smart network interface cards or as In-Network Processing accelerators.**

Both application scenarios require a high-throughput hardware network stack. In this work, we integrate such a stack into the open-source TaPaSCo framework and implement a library of easy-to-use design primitives for network functionality in modern HDLs. To further facilitate the development of network-attached FPGA accelerators, the library is complemented by a handy simulation framework.

In our evaluation, we demonstrate that the integrated and extended stack can operate at or close to the theoretical maximum, both for the stack itself as well as an network-attached machine learning inference appliance.

***Index Terms*—In-Network Processing, 100G, Network, TCP/IP**

I. Introduction

Numerous previous works have demonstrated the huge potential for acceleration that can result from attaching FPGAs directly to the network. In such scenarios, FPGAs can not only be used to implement smart network interface cards (SmartNIC) [1] and accelerate the network protocol stack, but they can also be employed to additionally offload parts of the application itself, e.g., machine learning inference in the case of Microsoft's Brainwave project [2], or network security [3].

While these approaches already demonstrate significant speedup, even more potential can be unlocked by moving the computation *into* the network, as so-called *In-Network Processing* (INP) [4]–[6]. This does not only allow moving computation closer to the origin of the data, but also facilitates distributed processing across the network.

However, with the network hardware available today, In-Network Processing is still severely limited. On the one hand, platforms such as Barefoot's Tofino provide high performance, but are limited with regard to programmability and the available memory on the device [7]. On the other hand, platforms providing full programmability, e.g., through Micro-C, provide only limited performance. FPGAs with high-speed network interfaces can provide both, high flexibility for the design *and* high performance.

Independent of whether FPGAs should be used as INP accelerator or SmartNIC, the availability of a high-throughput hardware network stack is crucial for successful deployment, and such a stack was presented by Ruiz et al. in [8].

In order to make this particular stack more accessible to researchers and designers, and to facilitate and automate the design of network-attached, FPGA-based accelerators, we integrate this stack with the open-source [9] TaPaSCo framework. This integration not only makes the network stack available in highly complex heterogeneous System-on-Chip (SoC) designs, but also allows using TaPaSCo's automatic design-space exploration for automatic and efficient traversal of large design spaces for network-attached accelerators.

After providing details on some modifications to the original stack (Section IV-A), making the stack more flexible, and describing the integration with TaPaSCo (Section IV-B), we present a library of easy-to-use design primitives (Section IV-C) that allow accelerators to communicate with the network stack on multiple protocol levels. As testing and verification through simulation are important steps of any design process, we also present a fast hardware/software-co-simulation for network-attached accelerators (Section IV-E).

To demonstrate how the design primitives and hardware/software-co-simulation facilitate the design of accelerators, we present a case study, transforming a host-attached FPGA-based accelerator for Sum-Product Network inference [10] into a network-attached accelerator (Section VI). Before that, we evaluate the raw performance of the network stack and various configurations in Section V.

We also provide related work in Section II, necessary background information in Section III, and a conclusion and outlook to future development in Section VII.

II. Related Work

FPGA-based network stacks are well established in the academic and commercial domain with several implementations that provide differing feature sets, connection speeds, and latencies. Compared to traditional software stacks, most of them lack more complex features like IP segmentation and only support particular default configurations without optional protocol extensions.

The availability of commercial TCP Offload Engines (TOE) that are capable of 100 Gigabit (100G) is still limited. One of them is developed by the Fraunhofer Heinrich-Hertz-Institute and Missing Link Electronics [11], which supports a single TCP session and can typically implement send and receive buffers in BRAM due to their low memory space requirements.

Most commercial TOEs support sub-100G connection speeds and are optimized for low latency [12]–[15]. A typical application field is High-Frequency Trading (HFT), where any

reduction in latency may increase the profitability of a financial trading algorithm.

Notably, ultra-low latency stacks generally tend to support fewer concurrent sessions, because FPGAs only offer a limited amount of low-latency on-chip memory, which is required for the per-connection TCP buffers.

In addition, there are various FPGA-based network stacks pursuing different design goals in the academic space: For example, the authors in [16] describe a hybrid approach where only the most data-intensive parts of the TOE are implemented in hardware, while the more control-intensive parts are handled by firmware running on a CPU.

The work in [17] presents a TOE that can achieve $4\,\mathrm{Gbit/s}$ of throughput from up to 2048 receive sessions, and $40\,\mathrm{Gbit/s}$ of throughput to up to 20480 transmit sessions, targeting asymmetric workloads such as video on demand. It does not support jumbo frames, and the maximum segment size (MSS) is fixed to $1460\,\mathrm{B}$. The memory architecture uses external SRAM for storing per-session state information and DRAM for the send and receive buffers.

The authors of a 100G-capable intrusion-prevention system in [18] observed that a per-session receive buffer with a fixed size is often unnecessary, since only $0.3\,\%$ of network packets arrive out of order and require buffering for reordering. A more memory-dense buffer architecture based on linked lists and dynamic allocation that supports as many as 100k concurrent sessions while still fitting into BRAM is developed. Packets that arrive in order are handled on a constant-time *fast path* without being buffered, whereas out-of-order packets are handled on a *slow path* that requires non-deterministic amounts of time.

A 10G-capable TCP/UDP network stack is presented in [19] which was designed to scale well in the number of sessions, verifiably achieving 10k open connections, however, at the cost of increased latencies. Unlike most published TOEs, this project handles the complexities of the TCP protocol through the use of high-level synthesis (HLS). By designing the stack in C++, the entire implementation comprises less than 8k lines of code and is thus significantly more compact than comparable implementations in HDL.

Several other research projects are based on this stack: For example IBM's *cloudFPGA* [20], which seeks to deploy large-scale datacenter applications on network-attached FPGAs, or the HLS-based HFT application in [21], which builds on top of the UDP stack and achieves a round-trip latency of $869\,\mathrm{ns}$.

The *Limago* network stack published in [8] used the 10G stack from [19] as a basis, which was then upgraded for 100G support. The authors added several new features in order to achieve this higher link speed, such as TCP window scaling and a hash table based on cuckoo-hashing, which replaces the previous slower session lookup mechanism. During this upgrade process, methods to improve checksum calculation in hardware were investigated in [22]. The authors concluded that an HLS-based approach does not perform satisfactorily and consequently developed a solution in VHDL.

Most of *Limago's* upgrades are also part of the 100G TOE published by ETH Zurich in [23], which is the successor to the 10G stack in [19]. It is also the hardware network stack implementation used in this work.

III. Background

This section presents necessary background information on the employed TCP/IP stack and the TaPaSCo framework.

A. Hardware TCP/IP Stack

The design of a 100G hardware TCP/IP stack is a significant undertaking that is off the scope of this paper. Instead, an existing TCP/IP stack published by the Systems Group at ETH Zurich is leveraged. An early 10G-capable variant of the stack is presented in [19], which has been upgraded for 100G support in [8]. The TCP/IP stack sets itself apart from other hardware TCP/IP stacks reviewed in Section II in several ways.

Except for few SystemVerilog-based wrapper modules, the entire project is implemented in C++ using Vivado HLS [24]. Designing the TCP/IP stack in a high-level language sacrifices some control over the resulting circuity but also increases productivity, in particular with regards to the highly control-intensive TCP protocol implementation. In addition, an HLS-based design methodology comes at the advantage of an easily maintainable and extensible codebase.

The TCP/IP stack is designed to support a large number of TCP sessions, which is demonstrated in experiments in [19] with 10,000 concurrent connections. Each session requires unique send and receive buffers, where the size of a single buffer ranges from $64\,\mathrm{KiB}$ to $256\,\mathrm{KiB}$, depending on the TCP *window scaling* configuration. For 10,000 sessions, this yields a total required buffer size between $1.3\,\mathrm{GB}$ and $5.2\,\mathrm{GB}$, which can only be implemented in off-chip memory like DRAM. This is a deliberate tradeoff that chooses a higher number of concurrent sessions at the cost of increased latency [19, p.42].

Per-session information, such as the connection status or timer values of the retransmission mechanism, is kept in BRAM-based tables indexed by a session ID. The session IDs are stored in a hash table, which handles collisions and realizes single-cycle lookups and deletions [8].

Hardware designs that use the TCP/IP stack are split into three AXI-Stream-interconnected Xilinx Vitis kernels: the *user kernel* containing the user logic of a network application, the *network kernel* containing the TCP/IP stack itself, and the *CMAC kernel* containing the Ethernet subsystem and physical layer implementation. The *network kernel* internally consists of multiple HLS-based IP cores that are wrapped by a SystemVerilog top module. The host software uses an OpenCL API provided by the Xilinx Runtime "XRT" to interact with the FPGA design.

B. The TaPaSCo Framework

The *Task Parallel System Composer* (TaPaSCo) [25] is an open-source framework providing a toolflow for the automated generation of System-on-Chip FPGA designs with a particular

focus on task-parallel computation. TaPaSCo aims to increase the *portability* and *scalability* of FPGA designs.

TaPaSCo includes base FPGA designs, referred to as *platforms*, for multiple Xilinx FPGA families. The *platform* typically contains platform-specific implementations of the memory subsystem, interrupt subsystem, interface to the host CPU, distribution networks for clock and reset signals, and a *status core* containing descriptive information about the design.

The *platform* acts as a hardware abstraction layer and provides a standardized interface to the *architecture* component of TaPaSCo, which itself is decoupled from the underlying FPGA technology. The *architecture* contains TaPaSCo *Processing Elements* (PE), which are application-specific compute kernels that can be implemented by the user either in an HDL or HLS based design flow. Due to the separation between *platform* and *architecture*, a PE needs to be designed only once and can be used across all FPGAs supported by TaPaSCo without any changes, which increases a design's portability.

Different types of PEs can be instantiated in different multiplicities, yielding what is called a *composition*. TaPaSCo supports automated *Design Space Exploration* (DSE), which can assist in finding a throughput-optimal *composition*. These mechanisms allow to easily scale a design without changing the actual user logic of the PEs.

Further to the hardware toolflow, TaPaSCo provides a runtime with a C/C++ API that allows host software to interact with the FPGA design. In particular, the API provides functionality to schedule jobs onto PEs, handle data transfers between host and FPGA, and monitor the execution state of individual PEs.

TaPaSCo contains several plugins, referred to as *features*, which add optional functionality that is not available across all supported FPGAs but specific to a particular *platform*. The *Network* feature adds an Ethernet subsystem to the FPGA design and selectively connects PEs to it, effectively providing them with link-level access to a network. The feature is available on multiple TaPaSCo platforms in a 10G variant. On the Xilinx Alveo U280 and the BittWare XUP-VVH platform, the Network feature additionally supports the instantiation of a 100G Ethernet backend. Three different operating modes are supported by the Network feature: In *singular* mode, a single PE is attached to the Ethernet subsystem, whereas in both *broadcast* and *round-robin* mode, the subsystem is shared by multiple PEs. We will use this feature in *singular* mode to implement the CMAC portion of the network stack architecture that was outlined in Section III-A.

IV. Implementation

This section describes the contributions with the goal of providing assisting tools and libraries for developing networked applications on FPGAs with the TaPaSCo framework.

A. Modification and Extension of the Network Stack

In preparation of using the TCP/IP stack withing the TaPaSCo ecosystem, we replaced the host software of [23], which uses OpenCL and the Xilinx Runtime XRT, with an implementation that makes use of the TaPaSCo runtime.

1) Parameterizable Data Width: While the TCP/IP stack's individual HLS cores are designed with parameterizable bit width, several of the HDL modules instantiate fixed-width IP cores.

The TaPaSCo Network feature supports both 10G and 100G Ethernet subsystems that have data interfaces of different bit widths. With the objective to attach to both subsystems natively, we reworked the TCP/IP stack to be more flexible and allow build-time configuration of the data bit width.

2) Issues with Buffer Memory Addresses: The TCP/IP stack can bypass the TCP receive buffer and directly deliver received data to the application layer. This is an optional configuration aimed at reducing latency. By default, the bypass optimization is enabled, such that the receive buffer is not used. In the original release [23], both logical and arithmetical errors existed in the calculation of buffer memory addresses, leading to data corruption when buffer bypassing is disabled. Using the simulation infrastructure presented in Section IV-E, we were able to track down the issues and fix them.

B. Integration with the TaPaSCo Framework

With its Network feature, the TaPaSCo framework already supports the automated instantiation of an Ethernet subsystem within an FPGA design. We use this subsystem to complement those parts implemented by the TCP/IP stack into a full implementation of the internet protocol suite.

In preparation for protocol comparisons during the experimental evaluation, we extended the Network feature with support for the Xilinx *Aurora 64B/66B* [26] point-to-point link-layer protocol, which can now optionally be used instead of the Ethernet link-layer protocol. Both Ethernet and Aurora can use the same physical layer implementation.

Since the AXI-Stream data interfaces of both Xilinx CMAC [27], and Xilinx Aurora [28] IP are clocked at higher frequencies (approx. $322\,\mathrm{MHz}$ and $403\,\mathrm{MHz}$, respectively) than the TCP/IP stack within the PE ($250\,\mathrm{MHz}$), a clock domain crossing is required on the data path between them. This is implemented by an AXI-Stream interconnect that is optionally instantiated by the TaPaSCo Network feature. To prevent a continuous transaction coming from a slow clock domain from being broken into multiple partial transfers within a faster clock domain, the interconnect is configured to *packet mode*, which buffers the entirety of a frame and forwards it in one piece. As the Xilinx CMAC contains only minimal internal buffering and implements *cut through* semantics on both RX and TX data paths [27, p.11], disabling the *packet mode* may result in a buffer-underrun in the CMAC and a corrupted packet on the wire.

The 100G network bandwidth places high demands on the bandwidth of the RX and TX buffer memories. In our case, even when using the RX buffer bypass, a DRAM-based memory subsystem would limit the achievable network throughput. To avoid being bound by memory performance, we instead make use of the *High-Bandwidth Memory* (HBM) feature of

TaPaSCo, which allows a PE to attach to high-bandwidth on-chip memory modules. In the resulting FPGA design, the DRAM-based memory subsystem for TCP buffers is replaced by an HBM-based subsystem. With this optimization, we achieve a full saturation of the link bandwidth and are not limited by memory bandwidth, as long as RX buffer bypassing is enabled (a detailed evaluation follows in Section V-D).

C. Design Primitives for Network Access

The interface between the TCP/IP stack and the user kernel comprises 16 AXI-Stream interfaces. Of those, 4 are used for UDP-related functions and 12 for TCP-related functions. The source code release of the TCP/IP stack in [23] contains a basic C++ library that simplifies the design of HLS-based user kernels by appropriately interacting with the 16 AXI-Stream interfaces. This HLS toolflow is also available for the TaPaSCo integration, but finer-grained control over the generated circuity may be required for more sophisticated user kernels. This is generally achieved by designing the user kernel in a dedicated HDL. Because of its good integration with TaPaSCo, the Bluespec [29] language is chosen as the target HDL.

To facilitate the development of Bluespec-based user kernels, we implemented a novel library that exposes all functionality of the TCP/IP stack via an idiomatic Bluespec API. Like the C++ library, the Bluespec library on the backend attaches to the 16 AXI-Stream interfaces of the TCP/IP stack and appropriately interacts with them.

The bit width of any data-carrying AXI-Stream interface changes depending on the configured data width of the TCP/IP stack. However, the user-visible bit width of the Bluespec library API is not determined by the TCP/IP stack configuration but can be freely chosen by the user. The library contains custom AXI-Stream width-converters that adapt the bit width of the TCP/IP stack to the user-configured API bit width. As a result, user kernels are portable across 10G and 100G subsystems when using our Bluespec library.

The library's TCP support is fully featured and supports data transferring, the opening and closing of connections, and putting a TCP port into *listen* state. Whenever a new packet shall be transmitted by the user kernel, it must first be announced to the TCP/IP stack. Data transmission can only start once the TCP/IP stack confirms the announcement with a status message. According to the authors, the duration of this handshaking sequence varies between 10 and 30 clock cycles [23], during which the stack must e.g. verify that the TCP send buffer has enough free capacity for the new packet. To maximize link utilization, the announcement of new packets and the transfer of data words belonging to an already announced packet are automatically pipelined by the library. Furthermore, the interface exposed to the user kernel is transfer-based rather than packet-based, meaning that the user can simply supply a stream of payload data which is then split into MSS-sized packets by the library automatically.

The user-facing UDP Bluespec interface is less complex than the one for TCP. It consists of methods for getting and putting data words and metadata of a new packet, where metadata includes a packet's length, the source and destination port, and the destination IP address.

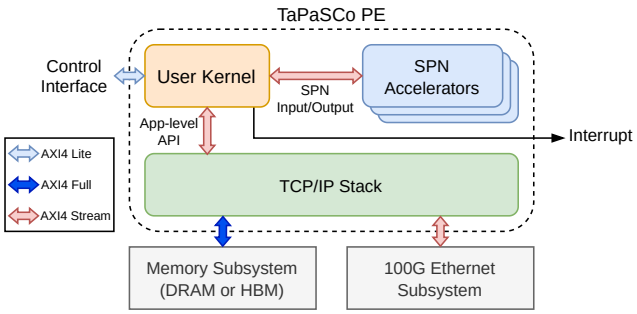

Fig. 1. Architecture of a TCP/IP-capable PE with connections to TaPaSCo subsystems.

The Bluespec library is available as free and open-source software [30].

D. Creating a TCP/IP-capable Design

This section describes how the individual parts, described in previous sections, are combined into a complete design that is capable of network communication via TCP/IP. A mapping between the seven layers of the *OSI Reference Model* [31] and the individual parts that implement those layers is provided below.

- Layers 1, 2 (partially: MAC sublayer) are implemented by the Ethernet subsystem instantiated by the TaPaSCo Network feature that includes modifications as described in Section IV-B. Depending on the TaPaSCo configuration, a 10G or a 100G backend is used.
- Layers 2 (partially), 3, 4 are implemented by the TCP/IP stack that includes modifications as described in Section IV-A. The data width of the stack matches that of the Ethernet backend.
- Layers 5, 6, 7 are implemented by the user kernel using the Bluespec library introduced in Section IV-C. Alternatively, HLS-based user kernels are also supported.

Fig. 1 shows a visualization of how a TCP/IP-capable TaPaSCo PE, consisting of a user kernel using the Bluespec library and a TCP/IP stack, integrates with the subsystems of a TaPaSCo design. The figure also shows a set of SPN accelerators attaching to the user kernel which are used in the case study in Section VI. In addition to the AXI-Lite control port, the PE module has two AXI-Full interfaces to the memory subsystem, and an RX and TX AXI-Stream interfaces to the TaPaSCo Ethernet subsystem. The dedicated memory subsystem is needed for hosting TCP TX and RX buffers, which generally are too large to be implemented in internal memory such as BRAM (cf. Section III-A).

When creating a TCP/IP-capable design for either the XUP-VVH or the AU280 board, which both use an FPGA of the Xilinx UltraScale+ family, special care has to be taken when placing the FPGA design. Internally, these particular FPGAs

are not one monolithic chip, but are divided into three separate dies that are referred to as *super logic regions* (SLR). The SLRs are mounted on a silicon interposer and interconnected using a *Stacked Silicon Interconnect*. For designs where components are spread over different SLRs, timing closure may be hard to achieve because the connections between SLRs are limited and induce a higher-than-normal delay. For instance, the HBM ports are located in the bottommost SLR, whereas the GTY transceivers that attach to the boards's QSFP28 connector may be located in the uppermost SLR.

To remedy timing failures due to suboptimal placement onto SLRs, manual placement hints or SLR crossing register slices, which trade improvements in frequency for additional latency and area, can be used.

E. Fast Hardware/Software-Co-Simulation of TCP/IP-capable Applications

This section describes the approach of simulating the behavior of a user kernel and the TCP/IP stack in a way that is not purely static and testbench-driven but dynamic in the sense that the simulated model can interact with its environment by exchanging Ethernet packets with the host operating system running the simulator.

A TCP/IP-capable TaPaSCo PE consists of several components implemented in different languages, with each offering its own simulation infrastructure: HLS modules can be natively compiled and tested, Bluespec modules can be simulated using *Bluesim*, and SystemVerilog modules can be simulated using an HDL-Simulator.

Seeing that both C++ and Bluespec can be compiled to Verilog, the fully integrated design can be simulated at the HDL level. While this HDL simulation is slower than a simulation of individual Bluespec or C++ components, it is the only way of assessing the behavior of the whole system.

The *Vivado Simulator* is a mixed-language (Verilog, SystemVerilog, VHDL), event-driven HDL simulator that is part of the Vivado software suite. It is chosen as the underlying HDL simulator since it integrates well with the remainder of the Vivado-based development flow of both TaPaSCo and the TCP/IP stack and since it supports the simulation of encrypted Xilinx IP cores. It contains the proprietary *Xilinx Simulator Interface*, a C API that allows a C/C++-based testbench to interact with a device under test (DUT) by reading and writing its top-level signals. A testbench implemented in C/C++ can use external libraries or operating system APIs, thus realizing complex interactions with a DUT that would be challenging to implement in an HDL-based testbench.

1) Architecture: To effectively simulate the behavior of a TCP/IP-capable TaPaSCo PE, the testbench must model the Ethernet layer to which the PE connects via its AXI-Stream ports. While hard-coding several test Ethernet frames into the testbench may be reasonable for stateless upper-layer protocols, this quickly becomes infeasible for the TCP protocol, where state information is attached to each TCP session e.g. in the form of sequence and ACK numbers. This implies that an effective testbench for TCP-based applications needs to be capable of processing network packets by, at least partially, implementing all involved network protocols.

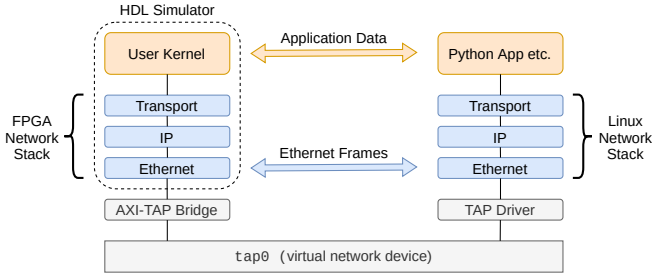

Fig. 2. Simulator architecture for TCP/IP-capable TaPaSCo PEs

Implementing any packet processing logic in Verilog seems unproductive, considering that even basic software implementations of a TCP/IP stack in high-level languages span several thousand lines of code (e.g., [32] and [33], both Linux userspace stacks with roughly 4k resp. 6k lines of C, or [34], a standalone embedded stack with roughly 24k lines of Rust).

Therefore, the complexity of the simulator architecture is reduced by (1) implementing the testbench in a higher-level language (C++) and (2) offloading packet processing itself to the Linux TCP/IP stack.

2) High-level language for testbench: The Vivado Simulator can be instructed to compile an HDL design into a C library that implements a behavioral simulation model of the design. A testbench written in C/C++ can be linked with this library and use the Xilinx Simulator Interface (XSI) for communication with the top module. By appropriately calling XSI functions, the testbench can write values to top-level HDL ports, read current values from those ports, and advance the simulation time.

3) Linux Stack Offloading: Since the testbench is C++-based, it may leverage arbitrary APIs of the host operating system. In this particular case, it hands over any packet parsing and processing tasks to the fully-featured Linux TCP/IP stack, such that the simulator itself does not need to include dedicated logic for this.

The simulator interacts with the Linux TCP/IP stack via a *TAP* device, which is a type of virtual network interface. A TAP device behaves like a standard Linux network interface, but rather than attaching to a physical network interface controller that interacts with a transmission medium, the data stream on the data link layer is exposed via a file descriptor. This file descriptor is read and written by a simulator component called *AXI-TAP Bridge* that translates between HDL signals and Ethernet frames.

Like the TAP device, the FPGA TCP/IP stack implements all upper-layer protocols down to the link layer, thus it is possible to bridge the Linux TCP/IP stack and the FPGA TCP/IP stack on this layer, effectively emulating the physical layer. An overview of the simulator architecture in relation to protocol layers and the positioning of the *AXI-TAP Bridge* is shown in Fig. 2.

In more detail, after reassembling an Ethernet frame from AXI-Stream beats, which are transmitted by the FPGA stack via its TX AXI-Stream port, this frame is written to the TAP device. The frame is then parsed and processed according to its content by the Linux TCP/IP stack. Conversely, any application data sent from a userspace process via the TAP network interface is encoded by the Linux TCP/IP stack into Ethernet frames that are translated by the AXI-TAP Bridge into AXI-Stream beats which are written to the FPGA stack via its RX AXI-Stream port.

F. FSM-based Simulator Architecture

A primary concern of the C++-based testbench is to interact with the AXI-Stream ports of the TaPaSCo PE simulation model for the exchange of Ethernet frames. In addition to these two AXI-Stream ports, the TaPaSCo PE contains an AXI-Lite control interface, which also needs to be driven by the testbench to set the PE arguments or control its execution state.

Seeing that AXI-Stream is unidirectional and AXI lite supports full-duplex, there are a total of four independent data streams to and from the TaPaSCo PE. While the AXI-Lite control interface is not strictly performance-critical, the AXI-Stream channels are. Waiting for a `TVALID` or `TREADY` signal in one channel must not block the other from sending or receiving data, as this would inaccurately model the underlying full-duplex connection of a real-world application.

As a result, the testbench must be able to handle four independent data streams to and from the PE simultaneously. This is achieved by an architecture of four FSMs that execute in parallel, where each FSM handles one data stream by interacting with its associated HDL signals in each clock cycle.

Read and write operations on the AXI-Lite interface are each implemented by an FSM that has a command queue for the addresses and data to be read or written. They are used e.g. to set an argument of the PE or to determine the return value of the PE.

The TX and RX AXI-Stream interfaces are each implemented by an FSM that is able to process one data word per clock cycle. Ethernet frames coming from the TAP device are split into individual AXI-Stream beats and transferred onto the RX AXI-Stream interface. AXI-Stream beats coming from the TX interface are buffered and reassembled into an Ethernet frame, which is then forwarded to the TAP device.

G. "In Circuit" Emulation

The proposed architecture enables arbitrary userspace software like *wireshark*, *netcat* or custom Python applications to interact with the live simulation model. Using these high-level tools significantly simplifies the development and debugging of TCP/IP-capable TaPaSCo PEs. The simulator supports two methods of interacting with the AXI-Lite control interface of the PE. First, a simple UNIX signal handler can trigger a predetermined AXI transaction, and second, arbitrary AXI transactions can be triggered via a UDP control socket.

The simulator is available as free and open-source software [35].

V. Evaluation

In order to evaluate the achievable network performance of TaPaSCo designs, throughput and latency measurements using the network protocols TCP, UDP, plain Ethernet, and Aurora are performed and compared with each other.

A. Experimental Setup

Multiple experiments are conducted using a setup consisting of two FPGA boards, one BittWare XUP-VVH and one Xilinx Alveo U280, connected via a 100 Gigabit network link.

Early experiments have shown that in setups consisting of one FPGA and one commodity server, the server-side often severely limits network performance. Achieving throughput close to the line rate requires non-trivial optimizations on the server-side, as demonstrated in [36]. Therefore, we do not further consider this type of setup in the evaluation of this work. Instead, we use one of the FPGAs as traffic generator to test the true capabilities of the setup in a throughput test.

The FPGA designs for benchmarking different network protocols are generated using TaPaSCo and contain exactly one PE that operates at $250\,\mathrm{MHz}$. At this frequency, the $512\,\mathrm{bit}$ AXI-Stream interface between PE and Ethernet or Aurora subsystem can provide a theoretical throughput of $128\,\mathrm{Gbit/s}$. A simulation of the TCP/IP stack using an MSS of $4\,\mathrm{KiB}$ shows that a link utilization of over $97\,\%$ is reached on this AXI-Stream interface. The resulting net data rate is sufficient to saturate the $100\,\mathrm{Gbit/s}$ line rate of the Ethernet backend.

Since a TCP buffer implementation in DRAM was found to bottleneck the system even when using RX buffer bypassing, if not mentioned otherwise, the buffers are implemented in HBM, which was found to not limit performance.

The benchmark PE implements both a throughput and a latency test, in both cases using a server-client-architecture. For the protocols TCP and UDP, it contains the TCP/IP stack and uses the Bluespec networking library introduced in Section IV-C as foundation for the test implementation.

The throughput achievable on the link between the two FPGAs using a particular setup is measured by timing the duration it takes to transfer $100\,\mathrm{GB}$ of data. Network packets are sent from the client to the server PE as fast as possible. The amount of payload per packet is configurable and is varied between $1\,\mathrm{KiB}$ and $8\,\mathrm{KiB}$, depending on the specific experiment.

For the case of TCP, splitting the $100\,\mathrm{GB}$ transfer into MSS-sized packets is handled by the Bluespec network library. For UDP and Ethernet, this is implemented within the user kernel itself. Depending on the specific network protocol, different amounts and different kinds of packet headers are prepended to the payload. While the payload size is kept constant across protocols, this results in overall packets of varying sizes. As Aurora is a packet-less protocol, splitting the data is not required in the Aurora-based test. Instead, all data is transmitted in a single headerless frame.

The latency of the link between the two FPGAs is measured as the round-trip time (RTT) of a $32\,\mathrm{B}$-sized ping packet. UDP,

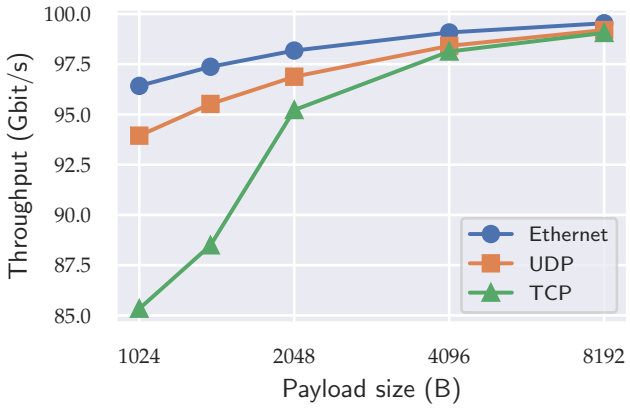

Fig. 3. Throughput of TCP, UDP, and Ethernet, for different payload sizes per packet. Note that the y-axis does not start at zero.

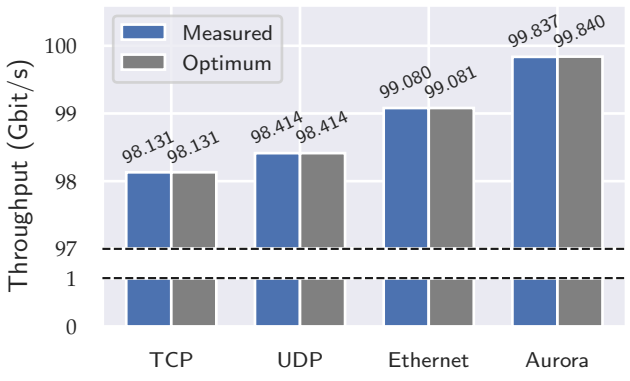

Fig. 4. Measured throughput and theoretical maximum for different protocols. Except for Aurora, all protocols use a payload size of 4 KiB

Ethernet, and Aurora are connectionless protocols where the ping packet can be sent to a server without any prerequisites. The same is not true for a TCP-based latency test, where a TCP connection has to be established prior to transmitting the ping packet.

B. Throughput

In this experiment, the relationship between the choice of network protocol, the payload size, and the measured throughput is evaluated. The payload size affects the fraction of protocol overhead, and thus places an upper bound on the achievable throughput. The experiment is conducted using receive buffer bypassing (see Section V-D for more details). Furthermore, a TCP window size of 256 KiB is used.

The usable throughput at the application layer ("goodput") is obviously lower than the bandwidth of the 100G network link, and depends on the ratio between payload size and total size of an Ethernet frame. For plain Ethernet using an MTU of 4 KiB, the theoretical goodput is equal to

$$\frac{100\,\mathrm{Gbit/s}\cdot 4\,\mathrm{KiB}}{4\,\mathrm{KiB}+8\,\mathrm{B}+14\,\mathrm{B}+4\,\mathrm{B}+12\,\mathrm{B}} = 99.081\,\mathrm{Gbit/s}. \quad (1)$$

Using an MSS of 4 KiB and assuming an IPv4 header with no options (20 B), a TCP header with no options (20 B), and taking into consideration the overhead from Eq. (1), the theoretical goodput of a TCP connection is given by

$$\frac{100\,\mathrm{Gbit/s}\cdot 4\,\mathrm{KiB}}{4\,\mathrm{KiB}+38\,\mathrm{B}+20\,\mathrm{B}+20\,\mathrm{B}} = 98.131\,\mathrm{Gbit/s}. \quad (2)$$

Assuming equivalent constraints as with TCP, the UDP protocol achieves a theoretical goodput of

$$\frac{100\,\mathrm{Gbit/s}\cdot 4\,\mathrm{KiB}}{4\,\mathrm{KiB}+38\,\mathrm{B}+20\,\mathrm{B}+8\,\mathrm{B}} = 98.414\,\mathrm{Gbit/s}. \quad (3)$$

Since Aurora is not a packet-based protocol, it does not carry any packet header overhead. In its reference implementation, however, the *clock compensation* mechanism inhibits data transmission for a maximum of 8 clock cycles every 4992 clock cycles [28, p.20], resulting in a different kind of protocol overhead. Assuming an infinite data frame, the worst-case goodput of Aurora 64B/66B amounts to

$$\frac{100\,\mathrm{Gbit/s}\cdot 4992\,\mathrm{B}}{5000\,\mathrm{B}} = 99.840\,\mathrm{Gbit/s}. \quad (4)$$

Fig. 3 shows throughput measurements for payload sizes between 1 KiB and 8 KiB. The payload size 1460 B is significant, as it is the largest MSS that fits into an Ethernet frame with default MTU of 1500 B. As both UDP and TCP operate on top of Ethernet, both carry a higher protocol overhead and achieve strictly lower throughput. For the same reason, UDP performs better than TCP, particularly for small payloads.

Fixing the packet based protocols to a payload size of 4 KiB, the measured throughput and calculated optimum for TCP, UDP, Ethernet, and Aurora are shown in Fig. 4. With this setup, the measured throughput of all protocols is virtually equivalent to the theoretical optimum derived in Eqs. (1) to (4).

The achievable throughput of a protocol expectedly is inversely proportional to the amount of overhead carried by it. From this perspective, Aurora is of particular relevance since it can be considered the limit case where packet header overhead is reduced to zero. However, due to the *clock compensation* overhead, Aurora cannot achieve perfect bandwidth utilization.

C. Latency

A comparison of RTT latency measurements for different protocols is shown in Fig. 5. Naturally, both UDP and TCP have a higher latency than Ethernet since these protocols are constructed on top of Ethernet. Also expectedly, TCP has the highest latency of all since its implementation is by far the most control-intensive. All latency measurements for TCP are executed within an established TCP session.

The average duration of a TCP handshake, which is necessary for establishing a TCP connection, is 2579 ns, i.e. slightly faster than the RTT of a ping packet. This is plausible because the data-less handshake packets do not traverse the full TCP/IP stack, such that the handshaking sequence is processed faster than a data-carrying ping packet.

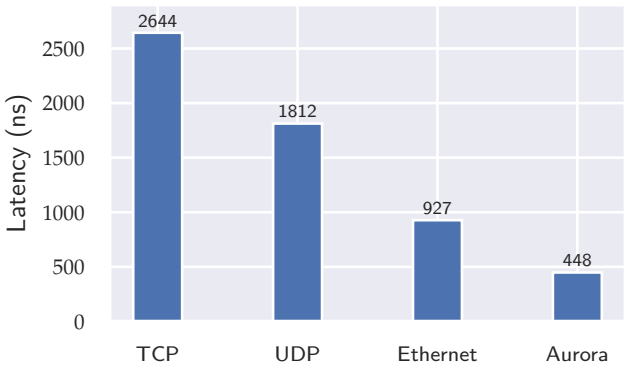

Fig. 5. RTT latencies using different protocols. For TCP, the duration of the handshake is not included.

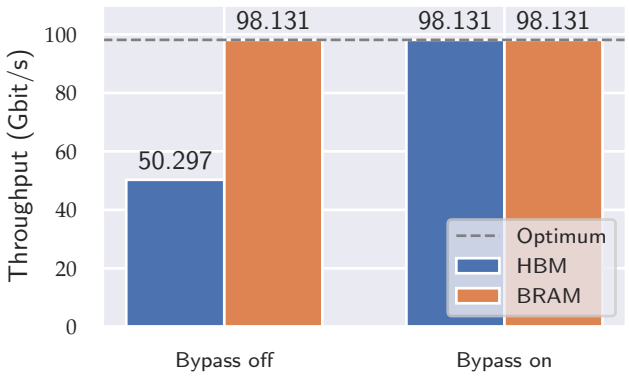

Fig. 6. TCP throughput with and without receive buffer bypass, for HBM and BRAM based buffer implementations. The MSS is 4 KiB, and the corresponding theoretical maximum throughput is marked.

D. RX Bypass

This section presents the examination of the configurable receive buffer bypass of the TCP/IP stack for TCP connections. For this, two variables are considered: (1) whether or not the bypass is enabled in the FPGA design, (2) which memory technology is used to implement the buffers. Memory subsystems based on HBM and BRAM are evaluated, resulting in four possible configurations that are compared in terms of throughput and latency.

1) Throughput: Fig. 6 shows the results of a throughput test for all four configurations. The theoretical maximum is calculated according to Eq. (2) and is marked in the figure. The two configurations using receive buffer bypassing perform extremely close to the theoretical maximum, regardless of the memory technology used. The throughput of the HBM-based system approximately halves if the bypass is disabled, whereas the BRAM-based system achieves the same performance with or without bypassing.

Regardless of the configuration, all transmitted data is written to memory once. However, assuming there are no retransmissions, this data is never read back. Disabling the bypass further increases memory pressure, since all received data is written to memory and read back shortly after when the application requests new data from the receive buffer. For HBM specifically, this access pattern yields suboptimal performance due to bus turnaround times between write and read operations [37, p.23].

The throughput result of the HBM-based configuration without bypass is noticeably low, yet plausible, considering that data must be written to and read from memory at approximately 50 Gbit/s (6.25 GB/s). In [38], a sequential combined read/write throughput of 12.9 GB/s was measured using a single HBM channel on an Alveo U280, which aligns with the throughput result of the experiment in this work. It is thus concluded that memory performance can impede the achievable throughput performance, and that memory pressure can be relieved by employing RX buffer bypassing.

2) Latency: For each of the four possible configurations, Fig. 7 shows both the duration of a TCP handshake (HS) as

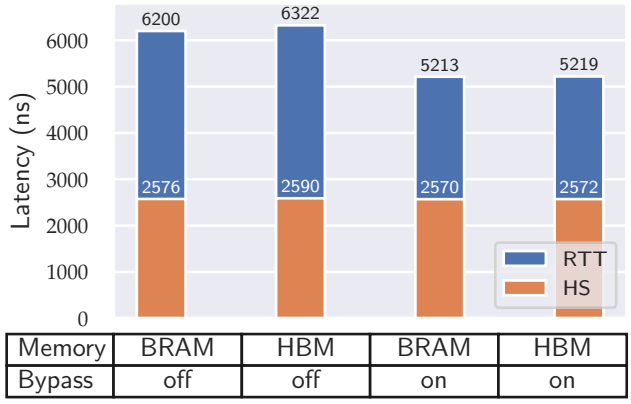

Fig. 7. Latency results of a TCP ping test, with and without buffer bypass, for HBM and BRAM based buffer implementations. The figure shows the duration of the TCP handshake (HS) and the subsequent RTT of a ping packet.

well as the RTT of a ping packet that is sent immediately after the handshake completes. It is noticeable that the duration of the TCP handshake is largely unaffected across all different configurations. This is expected because data-less TCP control packets like SYN, SYN+ACK, and ACK are never buffered and thus independent of the buffer architecture.

When enabling the buffer bypass, the RTT decreases by approx. 27 % for BRAM and by approx. 29 % for HBM. This decrease is caused by the fact that a received ping packet is now directly delivered to the application, instead of being first written to memory and then read back before being delivered to the application. This bypassing takes place at the server, when receiving the initial ping packet, and at the client, when receiving the ping reply. Generally, the TCP receive buffer is essential if the application layer cannot process bursts of incoming packets at line rate or if packets regularly arrive out-of-order and require buffering for reordering.

TABLE I
OVERVIEW OF RESOURCE UTILIZATION OF DIFFERENT TCP/IP STACK CONFIGURATIONS.

Component	CLB LUTS		Registers		Block RAMs	
	abs.	%	abs.	%	abs.	%
TCP & UDP	121491	9.3	212599	8.2	463.0	23.0
TCP only	114966	8.8	185244	7.1	439.5	21.8
UDP only	32395	2.5	93886	3.6	120.0	6.0

E. Resource Utilization

In Table I, the stack's resource utilization is summarized for a configuration that implements both the TCP and the UDP protocol, one that implements only UDP, and one that implements only TCP. As expected, the implementation of the TCP protocol proves to be the most resource-intensive. The TCP-only configuration requires almost four times the number of LUTs and BRAMs and almost twice the number of registers compared to the UDP-only version. As a result, the TCP-only configuration is similar in resource utilization to the combined TCP and UDP configuration.

VI. CASE STUDY: IN-NETWORK ACCELERATION OF SUM-PRODUCT NETWORK INFERENCE

In order to demonstrate how the design primitives described Section IV-C can be used to realize a network-attached accelerator for an actual application and which handy role simulation (Section IV-E) can play in the design process, we will use an existing FPGA-based accelerator [10], [39] for the inference in so-called *Sum-Product Networks* (SPN) as an example.

A. Sum-Product Network Background

Similar to other probabilistic (graphical) models, Sum-Product Networks [40] are recently receiving increasing attention from industry and academia alike. Due to their true probabilistic semantics, Sum-Product Networks can much better cope with the uncertainties found in real-world applications and are also able to quantify their uncertainty over their own output by means of probabilities, which makes them an appealing complement and alternative to currently more widely used machine learning techniques such as neural networks.

Sum-Product Networks capture the joint probability over a set of variables in the form of a directed acyclic graph (DAG), with three different types of nodes. Leaf nodes represent univariate distributions (e.g., Gaussian) over a single variable. Product nodes, on the other hand, represent a factorization of independent subsets of variables, while the weighted sum nodes indicate a mixture of multiple distributions. An example is shown in Fig. 8.

The graph structure of an SPN can either be handcrafted, complemented by weight learning, or automatically be learned from training data. An overview of the various available learning algorithms can be found in [41]. The survey also provides a nice overview of a wide range of usage examples for Sum-Product Networks, ranging from medical imaging [42] to approximate query processing for databases [43].

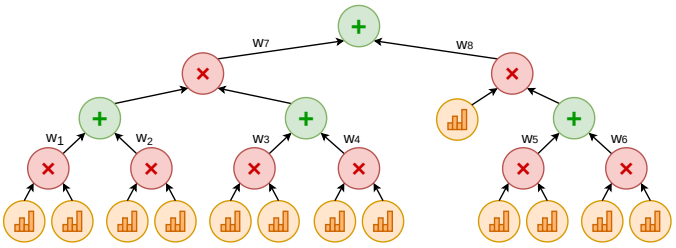

Fig. 8. Example for the graph structure of a Sum-Product Network, capturing the joint probability distribution over a set of variables.

After the graph structure has been obtained, inference can be used to answer probabilistic queries. To this end, the SPN DAG is traversed in a bottom-up fashion starting at the leaf nodes, where the (partial) input evidence is used to obtain probabilities. After propagating these probabilities through the graph, performing the respective operations, the final probability value is obtained at the single root node of the SPN.

The existing SPN accelerator [39], which we use as an example here, is designed to accelerate this inference step. Training of the SPN is assumed to have taken place beforehand.

B. Streaming-based Accelerator

In our prior work, we have employed SPN-accelerators to accelerate batch-wise processing of SPN inference queries [10], [39]. For the work presented here, due to the streaming-based interfaces of the networking stack, we had to adapt the corresponding accelerators to make them compatible. In prior work, the accelerator uses four distinct sub-modules for the SPN inference. 1) A control register allows configuration of the accelerator. 2) A Load Unit is responsible for loading input data from on-device DRAM. 3) The SPN-Datapath performs the actual inference and is fed by the Load Unit. 4) Results from the SPN-Datapath are passed to a Store Unit, which will write back the data.

In this work, we have adapted the SPN-Datapath from prior work to run as a free-running kernel. This means that no configuration is necessary. Additionally, the Load- and Store Unit have been stripped from the accelerator. Instead of using AXI4 for loading and storing the input- and output-data, we now rely on AXI-Stream to feed data directly into the SPN-Datapath. The results are then passed on via a second AXI-Stream interface. Overall, this makes the accelerators a lot more light-weight and reduces the control- and configuration overhead to zero. Data is pushed into the accelerator via AXI-Stream and results can be received via a second AXI-Stream interface. The bitwidths of both of those interfaces may be varied depending on the underlying SPN. The AXI-Stream slave interface (used for receiving input-vectors) is sized according to the size of a single input-vector (in this work, we use up to 640 bit wide input-vectors). The AXI-Stream master interface (used for sending the inference results) is 64 bit wide, due to the fact that the output is a single IEEE

TABLE II
DEGREE OF REPLICATION OF DIFFERENT NIPS-SPNS USED IN THE EXPERIMENTAL EVALUATION.

NIPS Variant	10	20	30	40	50	60	70	80
Number of Instances	5	3	2	2	1	1	1	1

754 double precision float. The accelerator is able to accept a complete input-vector every cycle. Due to its deeply pipelined nature, the accelerator is able to processes a single input-vector every cycle, assuming the the pipeline is kept full. The latency of the accelerator depends on the underlying SPN, since the SPN datapath varies in depth with the corresponding SPN.

C. Network Integration

The data path of the Xilinx CMAC is implemented by a 64-byte-wide AXI-Stream interface. Since the AXI-Stream slave interface of an SPN can have an arbitrary byte-width, a width-conversion is necessary in the general case when connecting the RX-path of the CMAC and slave-side of the SPN. The same is required for connecting the master-side of the SPN to the TX-path of the CMAC. In both cases, a Multiple-In Multiple-Out (MIMO) module is employed for this purpose.

To leverage the full bandwidth of the network connection while keeping the clock frequency of the SPN in an acceptable range, it may be necessary to replicate the SPN accelerator module several times in order to multiply the inference rate. The architecture of the TaPaSCo PE we used during the evaluation is equal to the one previously shown in Fig. 1.

D. Experimental Evaluation

In this section, we will discuss the results of the experimental evaluation using the SPN-accelerators presented in the prior sections. Each of the eight NIPS accelerator represents a different benchmark from the NeurIPS corpus [44], and the number indicates the number of inputs in the input-vector of the corresponding SPN. For example, NIPS10 will use 10 input-values, with each input-value being 8 bits wide. Thus the overall size of the input vector is 10 bytes or 80 bits.

Since all of the used NIPS-SPNs are able to run at $250\,\mathrm{MHz}$, we have to use replication, to ensure that the full available network bandwidth can be exploited. At $250\,\mathrm{MHz}$, we have to make sure that the SPNs can accept at least $50\,\mathrm{B}$ of data per cycle. Thus, for NIPS10, we need to replicate it five times to achieve this. Larger SPNs (like NIPS80) do not need to be replicated. The specific degree of replication that we used during evaluation is listed in Table II.

The resulting throughputs are depicted in Fig. 9. It is important to note, that due to the point-to-point nature of the Aurora protocol, it did not make sense to include it in the evaluation. This is due to the fact that it would be incompatible with the concept of replicating accelerators to exploit the available bandwidth. Instead, the results are limited to the comparison of TCP, UDP and plain Ethernet.

Apart from that, Fig. 9 shows that the throughput is very close to the theoretical limit of $100\,\mathrm{Gbit/s}$ for all different protocols. This shows that the presented networking stack does not only work with synthetic benchmarks, but also with more real-world applications, like SPN inference. Additionally, the peak throughput was achieved for many different accelerators using different input-widths, which also highlights the flexibility of the stack.

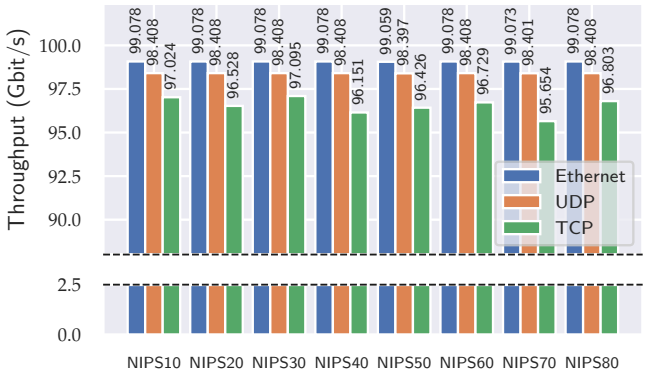

Fig. 9. Throughput of different SPN variants using different network protocols for input and output data transfer.

VII. CONCLUSION & OUTLOOK

In this work, we have integrated a high-throughput hardware network stack into the open-source TaPaSCo framework. During the integration process, several limitation of the existing stack have been removed and a combination with faster HBM memory was developed to allow for flexible configuration of the stack and better performance.

Next to that, we also developed a library of easy-to-use design primitives for network-attached accelerators in a modern HDL. The library is complemented by a simulation framework which leverages the Linux TCP stack to allow implementing testbenches for accelerators in high-level languages such as C/C++ or Python. The combination of the design primitives, the new simulation framework and TaPaSCo's automatic design-exploration framework make network-attached SoC-designs more accessible for researches and significantly facilitate the design process.

Our evaluation demonstrates that the integrated stack is able to achieve the maximum theoretical possible throughput. This finding has been confirmed in the case study, using a machine learning inference accelerator for Sum-Product Networks as an example, which also demonstrates how the design primitives can be used to attach existing accelerators to the network.

ACKNOWLEDGEMENTS

This work has been co-funded by the German Research Foundation (DFG) as part of project D2 within the Collaborative Research Center (CRC) 1053 MAKI and by the German Federal Ministry for Education and Research (BMBF) with the funding ID ZN 01|S17050. The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software.

References

[1] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: Smartnics in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[2] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[3] S. Mühlbach, M. Brunner, C. Roblee, and A. Koch, "Malcobox: Designing a 10 gb/s malware collection honeypot using reconfigurable technology," in *IEEE Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*. IEEE, 2010.

[4] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XVI. New York, NY, USA: Association for Computing Machinery, 2017, p. 150–156. [Online]. Available: https://doi.org/10.1145/3152434.3152461

[5] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, "The case for in-network computing on demand," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303979

[6] J. Hofmann, L. Thostrup, T. Ziegler, C. Binnig, and A. Koch, "High-performance in-network data processing," in *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2019, Los Angeles, United States.*, 2019.

[7] J. Wirth, J. A. Hofmann, L. Thostrup, A. Koch, and C. Binnig, "Exploiting 3d memory for accelerated in-network processing of hash joins in distributed databases," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, S. Derrien, F. Hannig, P. C. Diniz, and D. Chillet, Eds. Cham: Springer International Publishing, 2021, pp. 18–32.

[8] M. Ruiz, D. Sidler, G. Sutter, G. Alonso, and S. López-Buedo, "Limago: an FPGA-based Open-source 100 GbE TCP/IP Stack," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sep 2019, pp. 286–292.

[9] "Tapasco," https://github.com/esa-tu-darmstadt/tapasco.

[10] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch, "Automatic mapping of the sum-product network inference problem to fpga-based accelerators," in *IEEE International Conference on Computer Design (ICCD)*. IEEE, 2018.

[11] "Tcp/ip & udp network protocol acceleration platform (npap)," https://www.missinglinkelectronics.com/index.php/menu-products/menu-network-protocol-accelerator, [Online, accessed June 2021].

[12] "25g bit tcp offload engine (toe)," https://intilop.com/resources/product_briefs/25G_1K-Sess_TCP+UDP_Offload+MAC+Host_IFUltra-LowLatency(INT-25011).pdf, [Online, accessed June 2021].

[13] "Enyx ip cores," https://www.enyx.com/ip-cores, [Online, accessed June 2021].

[14] "Algo-logic: Ultra-low-latency 10g tcp endpoint," https://www.algo-logic.com/10g-tcp-endpoint, [Online, accessed June 2021].

[15] "Tcp offload engine: Lightspeed tcp," https://ldatech.com/Solutions/LightSpeedTCP, [Online, accessed June 2021].

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

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

[18] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. USENIX Association, 2020, pp. 1083–1100. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng

[19] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 36–43.

[20] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached fpgas for data center applications," in *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2016, pp. 36–43.

[21] A. Boutros, B. Grady, M. Abbas, and P. Chow, "Build fast, trade fast: Fpga-based high-frequency trading using high-level synthesis," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2017, pp. 1–6.

[22] G. Sutter, M. Ruiz, S. Lopez-Buedo, and G. Alonso, "Fpga-based tcp/ip checksum offloading engine for 100 gbps networks," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2018, pp. 1–6.

[23] "Vitis with 100 gbps tcp/ip network stack," https://github.com/fpgasystems/Vitis_with_100Gbps_TCP-IP, [Online, accessed June 2021].

[24] "Vivado design suite user guide: High-level synthesis (ug902 (v2020.1) may 4, 2021)," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf.

[25] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, "The tapasco open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2019, pp. 214–229.

[26] "Aurora 64b/66b protocol specification (sp011 (v1.3) october 1, 2014)," https://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b_protocol_spec_sp011.pdf.

[27] "Ultrascale+ devices integrated 100g ethernet subsystem v2.4 (pg203 april 4, 2018)," https://www.xilinx.com/support/documentation/ip_documentation/cmac_usplus/v2_4/pg203-cmac-usplus.pdf.

[28] "Aurora 64b/66b v12.0 (pg074 december 4, 2020)," https://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b/v12_0/pg074-aurora-64b66b.pdf.

[29] "Bsv documentation," http://wiki.bluespec.com/Home/BSV-Documentation, [Online, accessed June 2021].

[30] "Bluenet," https://git.esa.informatik.tu-darmstadt.de/net/bluenet.

[31] H. Zimmermann, "Osi reference model - the iso model of architecture for open systems interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, 1980.

[32] "nstack," https://github.com/jserv/nstack, [Online, accessed May 2021].

[33] "tapip," https://github.com/chobits/tapip, [Online, accessed May 2021].

[34] "smoltcp," https://github.com/smoltcp-rs/smoltcp, [Online, accessed May 2021].

[35] "net-sim," https://git.esa.informatik.tu-darmstadt.de/net/net-sim.

[36] M. Hock, M. Veit, F. Neumeister, R. Bless, and M. Zitterbart, "Tcp at 100 gbit/s–tuning, limitations, congestion control," in *2019 IEEE 44th Conference on Local Computer Networks (LCN)*. IEEE, 2019, pp. 1–9.

[37] "Axi high bandwidthmemory controller v1.0 (pg276 (v1.0) january 21, 2021)," https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf.

[38] Y.-k. Choi, Y. Chi, J. Wang, L. Guo, and J. Cong, "When hls meets fpga hbm: Benchmarking and bandwidth optimization," 2020.

[39] L. Sommer, L. Weber, M. Kumm, and A. Koch, "Comparison of arithmetic number formats for inference in sum-product networks on fpgas," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 1–10.

[40] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, 2011.

[41] R. Sánchez-Cauce, I. París, and F. J. Díez, "Sum-product networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[42] F. Rathke, M. Desana, and C. Schnörr, "Locally adaptive probabilistic models for global segmentation of pathological oct scans," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2017, pp. 177–184.

[43] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "Deepdb: Learn from data, not from queries!" *CoRR*, vol. abs/1909.00607, 2019. [Online]. Available: http://arxiv.org/abs/1909.00607

[44] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml