Architecture of a Synchronized Low-Latency Network Node Targeted to Research and Education

Christian Liß, Marian Ulbricht, Umar Farooq Zia
InnoRoute GmbH
Munich, Germany
{liss,ulbricht,umar}@innoroute.com

Hartmut Müller
Verilogic
Munich, Germany

***Abstract*— As line-speeds are sufficient for most applications, reduction of latency and jitter are gaining in importance. We introduce and discuss the architecture of a novel networking device that provides low-latency switching and routing. It integrates an up-to-date FPGA with a standard x86-64 processor and targets Time-Sensitive Networking (TSN) and machine-to-machine communication (M2M). First results show a cut-through latency of 2 – 2.5 µs for its 12 Gigabit Ethernet ports and full line rate packet processing. It features frequency synchronization across networks and is easily extendable, enabling researchers to build experiments in areas like industrial, automotive, and 5G mobile access networks, with highest precision, repeatability, and ease.**

Keywords*— *Low-latency networking; Synchronous Ethernet; IPv6; FPGA-based packet processing; Time-Sensitive Networking

I. INTRODUCTION

Improvements in throughput have long been a major goal in networking. Industrial network research was looking for ways to limit latency and jitter to a low upper boundary, often employing cut-through hardware.[13][14] Data centers and special cost-insensitive applications like high-frequency trading were pursuing low-latency networking since years, but only recently the focus of more widely used, cost-sensitive applications changed to lower latency and lower jitter.[19]

In this paper, we present TrustNode, a network device targeted to research and education that provides a very low latency, compared to other devices with Gigabit Ethernet interfaces. We show how this latency was achieved and where we still see potential for improvement.

The architecture of the TrustNode FPGA design evolved from our original store-and-forward packet processing IP-Core FlowEngine [1], which is used in many countries around the world in VDSL2 DSL Access Multiplexers (*DSLAM*) to provide QoS benefits like uninterrupted phone calls and shaped playout of bursty traffic to millions of households, with significant volumes, e.g., in Brazil, Germany, and Russia. Both designs are FPGA-based packet processing pipelines, inspired by Stanford's NetFPGA [2] project. Pipelines are a natural selection for packet processing with FPGAs, because they can be mapped easily and efficiently, providing good resource utilization and even heat distribution across the FPGA's die.

The contribution of this paper is a device and an easily extendable FPGA design that provide a combination of lowest-latency cut-through forwarding, a reasonable number of Gigabit Ethernet ports, and an easily extendable and portable software interface allowing to harness the breadth and productivity of standard x86 software. In addition, the system provides frequency synchronization across connected networks over standard Ethernet, allowing for experimentation with novel network-wide scheduling approaches, e.g. for 5G access networks and industrial networks, and for precise and repeatable experimentation in general. It enables low-jitter timestamp synchronization and prevents frequency deviation-based packet loss.

II. STATE OF THE ART

NetFPGA [11] is the de-facto state of the art in network nodes for research and education. It consists of a set of boards that span a wide variety of network interface speeds, and data flow-based FPGA designs with PCIe-based control interfaces.[12] NetFPGA is targeted to general-purpose network processing function development and provides a wide variety of design projects, but it is not optimized for low-latency forwarding or a small design footprint, and each of the boards provides only few network interfaces.[15][16][17][18] The datapath of the design is relatively wide, which uses the FPGA resources very inefficiently. Experiments with the first version of our FlowEngine have shown that wide datapaths lead to routing congestion, an excessive use of on-chip memories, which have a fixed maximum interface width per primitive, and underutilized logic slices due to high register densities.[20] Not using explicit signaling for all packet properties, but implicit packet structures based on already existing signals, as well as using inline signaling of metadata, i.e., using a Network-on-Chip (*NoC*), would save additional resources.[21][22]

Various other vendors provide FPGA boards for network systems design, but are either targeted to higher interface speeds with fewer Ethernet ports [3], or are lacking FPGA designs targeted to low latency with access to the FPGA design's source code [4], which is the only option for full controllability and visibility. Porting the NetFPGA design is rarely an option, as the effort of porting and testing the design is high and would not offer low latency forwarding.

978-1-5090-2839-9/17/$31.00 ©2017 IEEE

Using general-purpose server processors for network processing became a viable option within the last few years, as they were optimized more and more for network processing, especially with the advent of cloud computing and optimized software development kits like Intel's DPDK [23]. Latencies of a few microseconds [6] can be reached in the best case with a combination of high-end server processors, high-end memory, and high-end network interface cards, but this lacks visibility, and controllability, as software processes do not allow for easy extension without risking performance drops.

Designs based on FPGA SoCs, like Xilinx' Zynq [5][24] provide both FPGA-based processing as well as software-based processing, but have the disadvantage of interlocked tool-flows between the software part and the hardware part, are bound to a limited set of vendor tools, and provide good economics only for small FPGA designs.

Commercial low-latency networking devices targeted towards high-frequency trading or data centers [10] do not offer full access to their FPGA designs, nor do they provide an economic option for academia, public or commercial research laboratories.

III. System Overview

TrustNode a robust, self-contained commercial-grade device that is equipped with an FPGA design for low-latency cut-through switching and routing of packets, providing high visibility and controllability. TrustNode's interfaces are ESD protected and current limited, which makes the device more robust and makes it suitable even for use by unexperienced students. The TrustNode FPGA design can easily be customized, parameterized, or extended, and is constantly extended by new functions, like advanced flow processing or encryption cores.

The TrustNode FPGA is a Xilinx Artix-200T FPGA, which is currently the biggest FPGA eligible for Xilinx free Webpack toolchain license, enabling users to easily deploy development environments. The FPGA is connected to 12 10/100/1000 Mbps Ethernet PHYs, as shown in Fig. 1. The PHY-MAC connection is realized as RGMII, which provides the smallest FPGA footprint, the shortest latency, and the lowest jitter, compared to the popular SGMII interface, which provides the smallest footprint on the board.

Processors and System-on-Chips usually map worse to reconfigurable logic than (packet) processing pipelines, providing low processing power, requiring relatively large instruction and data memories, and limiting the developer to a small set of tools for the specific processor. Therefore, the FPGA in the TrustNode is accompanied by a standard x86-64 sub-board for general purpose computing. The FPGA and the processor sub-board are connected via a quad-lane PCIe Gen. 2.1 interconnect that can exchange data between the processor and the FPGA with a data rate of 5 GT/s (x4). Having an integrated processor sub-board also prevents incompatibilities with hardware or drivers that might occur in case of development boards that are connected to a host PC via complex interfaces like PCI-Express.

The device was developed as a network processing node, but features buttons and 7-segment displays, internal GPIO pin headers and general purpose LEDs, which are controlled by user logic, and a micro-USB console port and a software-controlled JTAG port for debugging, event triggering, and status display, and is also designed to work with remote access and NFV orchestration.

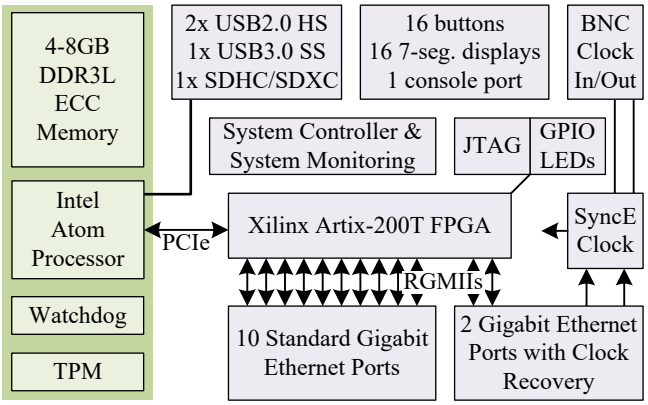

Fig. 1. Major device components, with the processor sub-board to the left

Several objectives were defined when designing the TrustNode. Besides optimizing for low-latency, low-jitter, and full line-speed, they included to use simple unified small-footprint interfaces at the edge of the FPGA and to keep the demand for processing power for the processor sub-board low.

Some tasks could be offloaded to a small non-volatile FPGA, the system controller, like reset sequencing, system status monitoring, FPGA pinout checking, scanning and decoding of the 16 front panel buttons, power management for the 16 7-segment displays of the front panel, fan control, high-speed FPGA reconfiguration, and warning sound generation via a beeper in case of overheating. The PCIe connection between the FPGA and the processor required a trade-off between the footprint of the PCIe bus master IP-Core on the FPGA and the packet forwarding performance of the processor. It was decided to create a slightly larger PCIe bus master IP-Core that directly moves packets received from the packet processing datapaths to one of 12 receive queues in the processor sub-board's DDR3L memory. There the packets are directly provided to Linux user space applications via a zero-copy driver to keep the processing effort low and the software forwarding speed high. Zero-copy and FPGA-based DMA accesses are also used to move packets from a transmit queue in the processor sub-board's memory to one of the RxProcDPs. The zero-copy driver was inspired by an open source driver for Intel Network Interface Card (*NIC*), ixgbe [7], but was simplified, yet to the processor the FPGA still looks like a NIC.

The PCIe bus master IP-Core not only provides packet interfaces, but also a memory-mapped interface (*MMI*) for single-word configuration accesses as well as an interface to generate Message-Signaled Interrupts (*MSI*) toward the processor. The MMI is a hierarchical, fully pipelined bus with decentralized address decoding that is connected to many of the system's components.

IV. Packet Processing on the FPGA

The FPGA design is based on a set of packet processing pipelines with shared memory and a fat tree topology with the network interfaces at the leafs. Packets are by default forwarded in a cut-through fashion, but switching to store-and-forward is possible on a per-packet basis by aggregating frames in the transmit FIFO (TXF).

The architecture consists of four parts, shown in Fig. 2:

- the receive datapaths, consisting of the line-speed, narrow receive network datapaths (*RxNwDP*) and the high-throughput, wide receive processing datapaths (*RxProcDP*), with a ratio of six RxNwDPs and one PCIe packet interface per RxProcDP
- the shared packet buffering
- the transmit datapaths, consisting of the high-throughput, wide transmit processing datapaths (*TxProcDP*) and the line-speed, narrow transmit network datapaths (*TxNwDP*), with a ratio of one TxProcDP per six TxNwDPs and one PCIe packet interface.
- some modules not involved in packet processing are used, e.g., for clock generation, interfacing to the system controller, datapath status monitoring, management interface to the Ethernet PHYs etc.

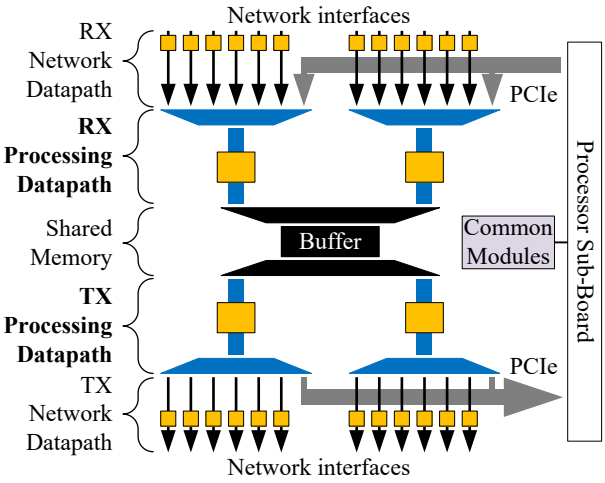

Fig. 2. FPGA Design Structure, with the respective modules (orange boxes)

Each datapath consists of one or more synchronous modules, connected in parallel, to keep the latency low, or consecutively, in the order of processing. Datapaths are designed for worst-case traffic load, both in terms of bit rate and packet/segment rate, starting with the network datapaths, which determine the processing datapath capacities and define the required memory bandwidth of the packet buffer.

All datapath modules are connected via ARM AMBA AXI4-Stream interfaces [8]. This interface has a very low overhead and allows for great flexibility. The system is currently self-similar, i.e., it consists of pipelines of modules with AXI4 interfaces that themselves consists of pipelines of register stages with AXI4 stages, but this is no necessity. Only few modules actually provide AXI4-Stream backpressure to preceding units to pause their operation, because this backpressure increases latency and jitter.

Frames arriving at one of the device's 12 Ethernet PHY ports are forwarded to the FPGA via RGMII. Each interface has its own RxNwDP, consisting of the RGMII interface IP-Core, a receive MAC unit (*RXMAC*), receive FIFO (*RXF*), and NoC Header Creation unit (*HC*). The RXMAC checks for Ethernet-level errors within received frames, removes the preamble, start of frame delimiter (*SFD*) and frame check sequence (*FCS*), segments packets into 64 Byte segments, checks if the frames have to be bridged or routed, parses Ethernet pause frames and predefined IPv6 prefixes, and adds information on SFD reception time, checksum status and others as sideband information. The segmented frame is forwarded to the RXF for clock synchronization, aggregation of incoming octets to full segments, and datapath width adaptation. During this aggregation, sideband information generated during reception is forwarded through a separate FIFO to the next stage. This stage is the HC, which counts and drops/trims segments that did not pass the checks of the RXMAC, and forwards all other segments to the connected RxProcDP, after adding a NoC header in front of each segment, which is constructed from the received sideband information and contains such information as input port, receive timestamp, and segmentation information. Following the HC unit, the segments from all connected ports are multiplexed by the RX arbitration unit (RXARB) in an interleaved scheme (cf. Fig. 3) onto the respective RxProcDP. There, the packets are parsed, classified, their output port is determined, and some additional checks are applied, which results in more information being added to the NoC header, like the output port, a flow identifer, and a frame discard flag with an additional discard reason for statistics creation. At the end of the two RxProcDPs, segments are multiplexed into a shared on-chip packet buffer.

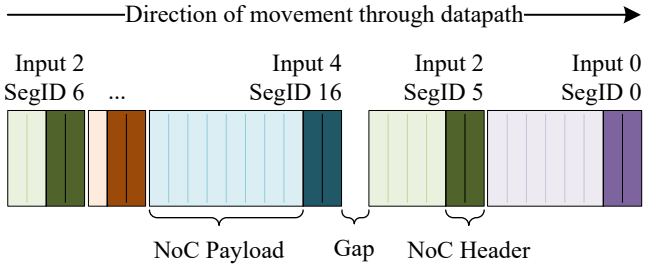

Fig. 3. Interleaved segments on processing datapath

The multiplexing to and from the buffer is based on 128 bit words, always alternating between the two RxProcDPs on the write side and between the two TxProcDPs on the read side. This buffer by default stores up to 3,000 segments, which is a reasonable size for this class of device under the assumption that links are usually not saturated with single flows. A buffer for larger amounts of data is located on the connected processor sub-board. In parallel to the buffer storage,

- a buffer management unit (BM) provides pointers to free buffers for segment storage, maintains linked lists of segments stored, and takes care of buffer protection,

- a queue manager (QM) provides de-jittering of received segments, maintains non-blocking output queues of packet heads, and checks queue length thresholds, and
- a segmentation-aware low-jitter transmit scheduler (*SCHED*) selects the next output port and queue for transmission.

After a frame is selected for transmission, it is read out of the shared packet buffer, forwarded onto one of the two TxProcDPs, modified according to its header information in the transmit routing unit (*ROUTE*), and then demultiplexed to the respective output port's header termination unit (*HT*) for NoC header stripping. At the output of the HT, it is stored in the transmit FIFO (*TXF*) of the respective output port for clock synchronization, aggregation, reassembly, and datapath width adaptation. At this point it is either directly forwarded (cut-through mode) or aggregated to a full packet (store-and-forward mode), before being provided to a transmit MAC unit (*TXMAC*) that sends the segments as a continuous frame to the respective Ethernet PHY, using an RGMII. This is done only, if no Ethernet pause backpressure is active at the start of frame transmission. The TXMAC adds preamble, SFD, FCS, and, for under sized frames, also padding. The TXMAC can also inject the current timestamp at a predefined location before calculating the FCS, to support the 1-step mode of the precision time protocol, or to enable latency measurements as used in section VI.

Instead of being received from a physical network interface and sent to one of them, a packet could also be received from the PCIe interface and sent to it, for further processing on the connected processor sub-board. The sub-board could also act as a larger packet buffer for bulk packet storage, i.e., using a share of the 4 GB DDR3L SDRAM on the processor sub-board for low-priority queues. All time slots on the processing datapaths that are not used by the network interfaces are made available to the PCIe packet interfaces. The PCIe interface can provide data faster than a single RxProcDP can process, but provides a backpressure mechanism to keep packets in the processor sub-board's memory instead of losing them, if insufficient time slots are available in the RxProcDPs.

The most basic RxProcDP unit is the Ethernet Switch module, which applies the standard learning/flooding/ageing scheme to received, to be bridged frames, using 128 hash buckets of eight entries per RxProcDP. The frame replication necessary for flooding is implemented in software on the processor sub-board, because it is not as time-critical. The learned addresses are synchronized between datapaths and with the processor sub-board. A more advanced RxProcDP unit is the Flow Cache, which currently parses the first segment of any received packet and compares the parsing results with table entries that were generated based on a flow table that is maintained in software on the processor sub-board by Open vSwitch [9]. This complex unit maps detected flows to a unique value for the FlowID field in the NoC header, which can then be used by all following units to address their respective flow action and flow counter tables, for actions like output and next hop selection, VLAN tagging, rate limiting, queue selection, and packet dropping.

The receive datapaths provide enough bandwidth to allow for packet processing without backpressure, thereby avoiding uncontrolled packet loss. In the receive datapaths parts of packets and the NoC headers might be overwritten or removed, but the traffic volume is never increased, to keep the required buffer bandwidth low and to avoid any backpressure.

Most of the RxNwDPs and TxNwDPs run at the frequency of the network interfaces, e.g., 125 MHz for Gigabit Ethernet, while the RxProcDPs and TxProcDPs operate at an independent frequency between 125 MHz and 200 MHz. The datapaths have a total net forwarding capacity of 40 Mpps, or 17.7 Gbps to 20.48 Gbps, for both the RX and the TX side, which is more than enough for the twelve Gigabit Ethernet ports. The RxProcDPs start with the receive arbiter (*RXARB*) and serves all connected network interfaces in a fair round-robin fashion. Due to an abundance of datapath capacity and relatively small segment size variations, no complex algorithm like deficit round robin is needed to enforce fairness.

V. Modules

Datapath modules, no matter in which datapath are usually similarly structured. Fig. 4 shows an example for an element on one of the processing datapaths.

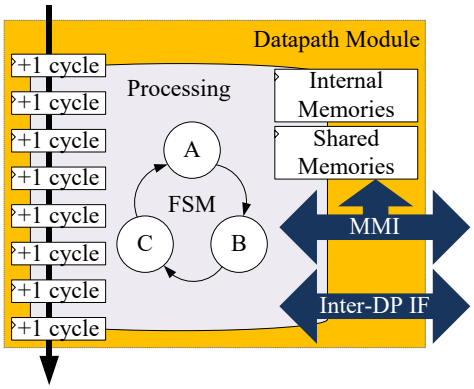

Fig. 4. Generalized processing datapath element structure

It has a register pipeline that carries the AXI4-Stream signals of the datapath, consisting, e.g., of the 8-bit or 64-bit TDATA and the single bit signals TVALID and TLAST. For network datapath modules, TDATA might be much narrower. Additional narrow sideband information carried in the signal TUSER is optional and module-specific. TREADY, a backpressure signal in the opposite direction than the other signals, is not registered and usually forms a combinatorial high-fan-out tree that expands with each additional module. It is terminated only by buffer elements, e.g., FIFOs.

The register pipeline is a delay element that provides a constant amount of time for the processing of one or several finite state machines (FSM) that use data from the upper part of the pipeline, either from the NoC header or from the packet as an input to its processing functions. The FSMs process this information, potentially looking up mappings and bit vectors, and potentially writing information back to the NoC header or some part of the packet. Packet length modification operations like removal or insertion of VLAN tags are also common, yet

they may require the realignment of all data following the point of removal or insertion in the same segment.

Modules often have additional configuration interfaces, like an interface to the MMI, an interface for synchronization to other datapaths (e.g., between RxProcDPs or between TxProcDPs), and to memory IP-Cores. In addition, connections between any two parts of the system are possible, but experience shows that it is important to keep the number of long-distance signals low to achieve sufficiently high clock frequencies and prevent routing congestion.

Most processing is done only for the first segment of a packet, if this segment is not to be discarded, while subsequent segments of the same packet might experience no processing at all or get the same (cached) information written to their NoC header. Reception of the last segment of a frame indicates whether the whole frame was valid, so internal state changes to, e.g., packet counters, are executed on module level only on successful forwarding of the last segment.

For processing datapaths, caching of information for subsequent frames requires holding state per input port (in case of RxProcDPs) or output port (in case of TxProcDPs), because RXARB and transmit scheduler interleave segments from or to different input and output ports.

Subsequent processing in modules eases development of new modules and eases keeping track of data dependencies. Yet, serial processing accumulates pipeline latencies. Therefore, several modules might also be attached in parallel to the same register pipeline, as shown in Fig. 5. The pipeline has to be long enough for the functional module with the longest latency, if several modules finish at the same time the results might have to be arbitrated, modules might have varying latencies, designers have to take care not to create consistency issues, and each memory lookup usually costs several cycles. This is, because each memory access are usually pipelined, because on-chip SRAMs, contrary to ASIC designs, have fixed locations on the die, so long on-chip lines might be needed to connect the datapath module to its respective memories. Only with input and output registers to and from the memory are synthesis tools able to achieve high clock frequencies in potentially densely packed FPGAs with non-uniform memory distribution.

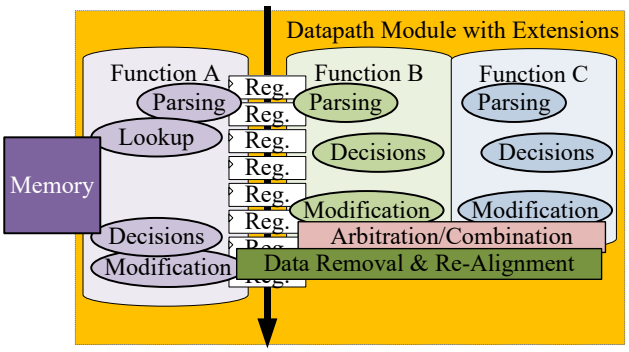

Fig. 5. Parallel functions with shared pipeline

When parsing received packets, their beginning has a fixed format, the Ethernet frame structure, but the further into the packet the parser/classification unit has to go, the more different offsets, types, and formats have to be supported. Anything different from bulk traffic and traffic that has to be handled with very low latency is usually forwarded to the processor sub-board, like ARP, ICMPv6, ESMC etc., which provides an easy and cheap way to implement complex and widely diverse protocols at a low speed. This also harnesses the steadily growing codebase for network processing on Linux.

When parsing, fields can wrap to the next data slot or even span several data slots. Depending on the protocol stack within the frame, fields might even wrap to the next segment. Wrapping to the next segment means keeping state per input port (on the RxProcDPs) or per output port (on the TxProcDPs), because segments from different ports are processed in an interleaved fashion. Therefore, for some applications it might be advantageous to implement parsers and other functional units in the 8 bit-wide RxNwDPs (in parallel to the RXMACs). This way they would have access to contiguous, uninterrupted frames with the byte offset being the only dimension, at the cost of having the respective unit implemented up to 12 times instead of just two times. Yet, functional units implemented in parallel to the RXMACs have to be robust enough to deal with input noise, because Ethernet fragments etc. are filtered out only at a later stage (*HC*).

VI. Analytic Model and Measurements

We created an analytic latency model (cf. Fig. 6) of all parts adding latency or jitter to forwarding packets to guide our development and being able to optimize for latency. The model resulted from an in-depth analysis of various implementations of the FlowEngine packet processor and was tuned every time we either found a potential for latency improvement, or detected that previously anticipated improvements resulted in extensive complexity. The latency categories are pipeline delay, arbitration delay, aggregation delay, backpressure cycles, cross-clock domain synchronization cycles, datapath width adaptation cycles, and head-of-line blocking cycles. Pipeline delay is added by any register stage in the processing modules and by sequential states in the BM/QM/SCHED cores. Arbitration delay depends on the RXARB, BM, and SCHED schemes and their utilization. Aggregation delay is observed in the RXF, where data is aggregated to full segments, and in the de-jittering part of the QM, which waits for the second segment of a frame to arrive to prevent jitter-based TXF underruns. Backpressure is seen wherever data is added to segments, in the HC, the ROUTE unit, and the TXMAC, or if the TXMAC is blocked by Ethernet pause backpressure. Synchronization delay is observed at any clock domain crossing. The datapath width adaptation adds latency only in the current RXF implementation, which can easily be updated. Head-of-line blocking can happen only in output queues, and only, if cut-through forwarding is prevented due to filled queues, or due to an already started and still ongoing frame transmission.

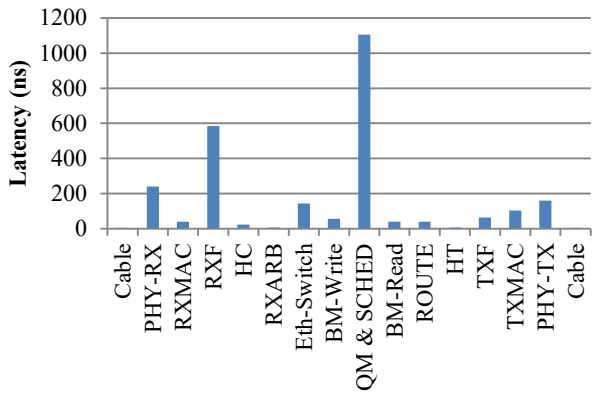

Fig. 6. Analytic Model: Contribution of each module to the latency

We started with the cables connecting to the Ethernet ports, each adding around 5 ns per meter, continuing with receive and transmit latencies of the Ethernet PHY chip, the RGMII receive and transmit IP-Cores on the FPGA, the receive and transmit Ethernet MACs on the FPGA and so on. Some parts of the latency are fixed, including all processing pipeline delays, and some parts vary based on the utilization of the respective units, and phase differences at clock domain crossings. All latencies given in the figure are worst case, i.e., including jitter, for traffic on all ports, but no head-of-line blocking through filled queues. The single highest latency is expected from the queue manager due delay scheme used to compensate for worst-case jitter in the RxProcDPs, especially the RXARB scheme. The jitter at the RXARB grows with the number of ports multiplexed. The second highest latency is exhibited by the RXFs, which have to aggregate full 64-byte segments at Gigabit line-rate, resulting in 512 ns for the aggregation alone. Having an even smaller segment size of 32 bytes would reduce this latency of the RXF alone by 256 ns, but would double the NoC header overhead in the processing datapaths and would potentially increase datapath module complexity as well.

Various measures were implemented based on the analytic model to achieve both low latency and low jitter. The most effective measure was to split frames into small segments and to interleave them on the datapaths, but it requires special queuing like Virtual Output Queues to avoid blocking of segmented frames. The segmentation size is determined on the one hand by the amount of NoC information required, i.e., the NoC header size, by the NoC overhead allowed, i.e., the expected datapath utilization, and by the depth of the frame parsing in the application, and on the other hand, by the aggregation latency that is still acceptable. Cut-through forwarding helps especially with large frames and only, if forwarding of (shortened) bad frames is allowed in the application, and if the device-internal jitter can be kept reasonably low. The effective datapath delay was minimized by parallelizing modules and by clocking the modules at the highest possible frequency, which is determined by the FPGA technology and the FPGA's speed grade. In addition, using RGMII interfaces instead of SGMII interfaces saves a considerable amount of time, both in terms of latency and jitter. Not writing full segments into the buffer and reading full segments out, but interleaving them on memory word level helped scaling to two parallel datapaths on the RX and the TX side. Other measures were using short control loops, e.g., between the TXF fill level threshold and the transmit scheduler, and fully exploiting the low-latency indicators of different FIFO types used in parallel to save a few cycles. These indicators of packet presence are used to start the RxProcDP arbitration process with the RXARBs and to start the generation of preamble and SFD in the TXMACs. Keeping datapaths narrow also helps to keep jitter low, because less padding at the end of segments and of memory words is to be taken into account. Last, but not least, no datapath should be fully saturated, because clock domain crossing are always providing a certain level of uncertainty to TDMA schemes, so timeslots might be missed.

After completing the implementation of a first version of the TrustNode FPGA design, we measured the latency of the device using the timestamp injection in the RXMAC and the TXMAC described above, with offline evaluation of the packet traces captured with Wireshark for all frame sizes from 64 bytes to 1518 bytes. The resulting latencies are shown in Fig. 7. The observed jitter is 504 ns, which is almost 512 ns. 512 ns is the reception time for one 64-byte segment on Gigabit Ethernet. This jitter arises mainly from the arbitration schemes in RXARB and transmit scheduler in combination with the receive de-jittering scheme in the QM, which delays frame forwarding until the reception of the second segment of a frame to have more than enough data for cut-through forwarding without any risk of TXF underrun. Any jitter apart from the one created in RXARB and transmit scheduler is neglible. The jitter may be reduced by tuning the de-jitter function, or by reducing the jitter introduced by the RXARB. In this early version, the datapath frequency, i.e., the clock frequency of everything between HC and HT, including the RxProcDPs and the TxProcDPs, was set to 125 MHz, which is expected to have added around 400 ns to the latency and to have almost doubled the jitter. This leaves an easy path for further improvement.

The figure also shows that the forwarding time does not depend on the frame size, which is a useful property of cut-through forwarding.

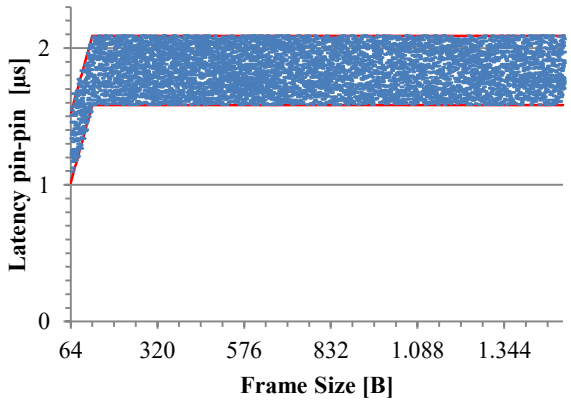

Fig. 7. TrustNode's measured low rate single port latency (slow data-path, no cross-traffic), MAC-to-MAC

The figure does not account for the latency of the RGMII connection, the Ethernet PHYs on both sides, and the cable. Those add another 411 ns, so the experienced latency for low data rate traffic is between 1,507 ns and 2,499 ns, if 1 m of Ethernet cable is used on the input and output port. This makes it the lowest unicast latency on the market for layer 2/3 processing devices with Gigabit Ethernet ports.

VII. Conclusion and Future Work

This paper introduced the TrustNode, a networking device and FPGA design for lowest latency packet processing. Measurement results show both significant latency and jitter improvement to the state of the art, while further improvements are predicted based on an analytic model created during the development of the device.

Users may add functionality in parallel to the existing modules, exploiting the existing optimizations without risking any latency or throughput penalties. More than 50 % of the logic and on-chip memory resources are available for these FPGA design extensions. Extensions can also easily be added to the Linux software stack running on the Atom processor, which has a very low idle load and memory footprint.

Next steps are updating the receive FIFOs for even lower latency and the precise evaluation of latency-determining effects at full load, i.e., maximum interface utilization, especially with updated arbiters. Other topics are the implementation of an on-chip loopback datapath from the transmit processing datapaths to the receive processing datapaths, complex flow classification, flow-level statistics, and integration with DPDK.

Acknowledgment

Part of this work has been supported by the projects TrustNode, CHARISMA, and SELFNET of the European Union's Horizon 2020 research and innovation programme, with grant agreement numbers 664078, 671704, and 671672, respectively.

References

[1] InnoRoute FlowEngine IP-Core: https://www.innoroute.com/flowengine [accessed 26.Jan.2017]

[2] NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing; John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo; MSE 2007, San Diego, June 2007

[3] Mellanox: http://www.mellanox.com/page/ethernet_cards_overview [accessed 26.Jan.2017]

[4] Nallatech: http://www.nallatech.com/solutions/fpga-network-processing/ [accessed 26.Jan.2017]

[5] Soc-e (2017). SMARTzynq Module: 5 Port Gigabit Ethernet Industrial Embedded Switch Module. Available http://soc-e.com/products/smart-zynq-module/. [accessed 2017.01.29]

[6] Lockwood, J. W. and Monga, M. (2015). Implementing Ultra Low Latency Data Center Services with Programmable Logic: 68-77

[7] Intel (2013). Linux Base Driver for the Intel Ethernet 10 Gigabit PCI Express Family of Adapters. Available https://www.kernel.org/doc/Documentation/networking/ixgbe.txt. [accessed 2017.01.29]

[8] ARM (2010). AMBA4 AXI4-Stream Protocol Specification. [accessed 2017.01.29]

[9] Pfaff, B.; Pettit, J.; Koponen, T.; Jackson, E. J.; Zhou, A.; Rajahalme, J.; Gross, J.; Wang, A.; Stringer, J.; Shelar, P. and others (2015). The Design and Implementation of Open vSwitch: 117-130

[10] Altera (2012). Altera for Arista 7124FX Platform. Available http://arista.com/assets/data/pdf/7124FX/Arista_Altera%20Paper.pdf. [accessed 2017.01.29]

[11] Gibb, G.; Lockwood, J. W.; Naous, J.; Hartke, P. and McKeown, N. (2008). NetFPGA - an Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers, IEEE Transactions on Education 51: 364-369.

[12] Lockwood, J. W.; McKeown, N.; Watson, G.; Gibb, G.; Hartke, P.; Naous, J.; Raghuraman, R. and Luo, J. (2007). NetFPGA - an Open Platform for Gigabit-Rate Network Switching and Routing: 160-161.

[13] Kay, R. (2009). Pragmatic Network Latency Engineering Fundamental Facts and Analysis, cPacket Networks, White Paper: 1-31.

[14] Pedreiras, P.; Leite, R. and Almeida, L. (2003). Characterizing the Real-Time Behavior of Prioritized Switched-Ethernet, 2nd RTLIA.

[15] Keinänen, J.; Jokela, P. and Slavov, K. (2009). Implementing zFilter based Forwarding Node on a NetFPGA.

[16] Naous, J.; Gibb, G.; Bolouki, S. and McKeown, N. (2008). NetFPGA: Reusable Router Architecture for Experimental Research: 1-7.

[17] Naous, J.; Erickson, D.; Covington, G. A.; Appenzeller, G. and McKeown, N. (2008). Implementing an OpenFlow Switch on the NetFPGA platform: 1-9.

[18] Shafer, J.; Foss, M.; Rixner, S. and Cox, A. L. The Axon Network Device: Prototyping with NetFPGA.

[19] Okafor, K.; Ezeha, G.; FU, I. A.; Okezie, C. and Diala, U. (2015). Harnessing FPGA Processor Cores in Evolving Cloud Based Datacenter Network Designs (DCCN): 1-14.

[20] Ye, A. G.; Rose, J. and Lewis, D., 2005. Field-Programmable Gate Array Architectures and Algorithms Optimized for Implementing Datapath Circuits. Library and Archives Canada = Bibliothèque et Archives Canada.

[21] De Micheli, G. and Benini, L. (2001). Powering Networks on Chips: Energy-Efficient and Reliable Interconnect Design for SoCs: 33-38.

[22] Moadeli, M.; Shahrabi, A.; Vanderbauwhede, W. and Ould-Khaoua, M. (2007). An Analytical Performance Model for the Spidergon NoC: 1014-1021.

[23] dpdk.org (2017). Data Plane Development Kit. Available http://dpdk.org/. [accessed 2017.01.29]

[24] Hu, C., Yang, J., Zhao, H., & Lu, J. (2014). Design of all programmable innovation platform for software defined networking. Power, 75(91W), 21W.