100 Gbit/s End-to-End Communication: Designing Scalable Protocols with Soft Real-Time Stream Processing

Steffen Büchner*, Lukasz Lopacinski‡, Jörg Nolte*
*Distributed Systems/Operating Systems Group,
‡Systems Group
Brandenburg University of Technology Cottbus–Senftenberg,
Cottbus, Germany
Email: {steffen.buechner,lukasz.lopacinski,joerg.nolte}@b-tu.de

Rolf Kraemer
System Design Group
Innovations for High Performance Microelectronics GmbH
Frankfurt/Oder, Germany
Email: kraemer@ihp-microelectronics.com

***Abstract*—With the recent roll-out of 100 Gbit Ethernet technology for high-performance computing applications and the technology for 100 Gbit wireless communication emerging on the horizon, it is just a matter of time until non-high performance computing applications will have to utilize these data rates. Since 10 Gbit/s protocol processing is already challenging for current server machines and simply upscaling the computing resources is no solution, new approaches are needed. In this paper, we present a stream processing based design approach for scalable communication protocols. The stream processing paradigm enables us to adapt the communication protocol processing for a certain hardware configuration without touching the protocol's implementation. We use this design technique to develop a prototype communication protocol for ultra-high throughput applications and we demonstrate how to adapt the protocol processing for a *Stable Throughput* as well as for a *Low Latency* scenario. Last but not least, we present the evaluation results of the experiments, which show that the measured throughput respectively latency of the adapted protocol, scales nearly linear with the number of provided interfaces.**

***Index Terms*—100 Gbit/s Wireless, Soft Real-Time Stream-Processing, End2End100**

I. Introduction

Today's applications and services become more depending on fast wireless communication. Every day, terabytes of data are transferred over wireless networks with HD video streaming being only one example. To satisfy the hunger for even higher data rates, many companies and research groups focus on improving existing technologies, e.g., IEEE 802.11 (WiFi).

Once the problems on the physical layer, which are currently addressed by many researchers, are solved, higher communication layers also need to be changed to meet the desired data rate. This problem is addressed in the project End2End100[1]. The project End2End100 is a project of IHP[2] and BTU[3] and is related to the DFG (German Research Foundation) priority program "100 Gbit/s Wireless And Beyond"[4], in which transmission technologies for 100 Gbit/s wireless communication are investigated. While the priority program is focusing on the transmission technology, the overall goal of End2End100 is to integrate the results of the priority program within an end-to-end communication solution and to achieve a wireless throughput of 100 Gbit/s between two endpoints.

While the necessity of new concepts for high data rate communication concepts is widely acknowledged, research regarding new higher level concepts for the communication protocol processing at high data rates is still hardly an issue in the research community.

While it is important to exploit existing approaches, it will not enable us to use future data rates of 100 Gbit/s and beyond. The following example will emphasize the problem. A server in a data center is equipped with a 100 Gbit/s network interface and a state-of-the-art processor, such as the Intel Haswell. To be able to fully utilize the network interface, the server has to process 100 Gbit/s = 12.5 GB/s of packet data per second. Assuming the packets have a size of 1500 Bytes, the server has to process 8,333,333.33 raw packets per second respectively a new packet every 120 nano seconds. Putting that in relation with the 96.4 ns main memory access latency for a 64 Byte cache line (Intel Haswell [1]), indicates that we have to think of new protocol processing paradigms.

This example leads to five necessities, which have to be met for enabling 100 Gbit/s protocol processing. We need communication protocols, which are (1) easy to parallelize, and the entire protocol processing must be (2) performed in parallel with (3) close to zero parallelization overhead. Computation intensive parts of the protocol, like CRC/FEC calculation must be (4) offloaded to special purpose external accelerator hardware, e.g., FPGAs, and the host machines (5) should only produce and consume data, but should not take part in the protocol processing as such.

In this publication, we show how the stream processing programming paradigm can incorporate these necessities into a single design flow, and how it enables us to efficiently scale up the communication protocol processing regarding the desired

[1]German Research Foundation Project End2End100, DFG NO 625/9-1
[2]Innovations for High Performance Microelectronics GmbH
[3]Brandenburg University of Technology Cottbus–Senftenberg
[4]DFG Schwerpunktprogram SPP 1655 Drahtlose Ultrahochgeschwindigkeitskommunikation für den mobilen Internetzugriff

DOI 10.1109/LCN.2016.25

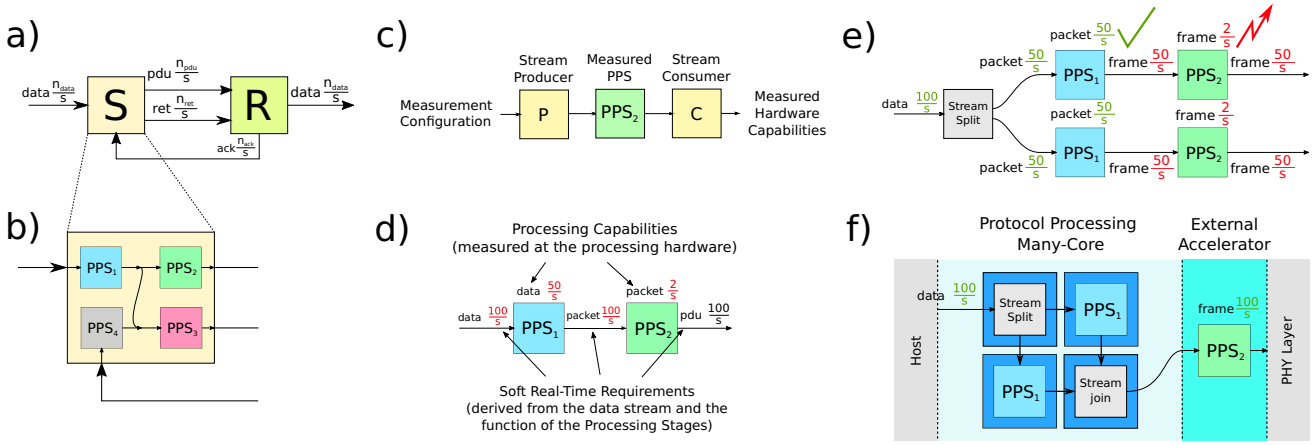

Fig. 1. Stream processing based protocol design flow. a) Protocol design. b) Protocol implementation based on data independent Protocol Processing Stages (PPS). c) Measuring the hardware processing capabilities. d) Deriving the soft real-time requirements. e) Adapting the Protocol Processing Graph with the help of Stream Operators, i.e., parallelization. f) Mapping the Protocol Processing Graph onto the processing hardware, e.g., assign PPS to cores of an embedded many-core and further offload PPSs to more suitable processing hardware.

data rate without changing the protocol as such.

The remainder of this paper is organized as follows: In section II, we describe how the stream processing approach can be used for the protocol design and the protocol processing. Section III gives an overview of our communication protocol for high data rates and its representation as a stream processing graph. Section IV focuses on the adaptation of the protocol processing for a low latency, and a high and stable throughput scenario. The evaluation results are presented and discussed in section V, followed by the presentation of some related work in section VI. The paper is finished with our conclusions and an outlook in section VII.

II. Stream Processing of Protocols

As mentioned in the introduction, upcoming transmission technologies need a new way of thinking about protocol processing. We propose a new protocol processing concept based on soft real-time stream processing, which enables the protocol developer to design scalable and parallelizable protocols. Stream processing is a data-flow oriented programming approach, in which the control flow follows the data flow, rather than being defined by function calls. The stream processing [2] approach is widely used for tasks that have to process a continuous flow of data, e.g., deep packet inspection.

In principle, all communication protocols can be described as a simple stream processing graph. Consider the *Protocol Processing Graph (PPG)* of the generalized communication protocol shown in figure 1a). The protocol consists of a sender *S* and a receiver *R*. The sender consumes a stream of data and transforms it into a stream of protocol data units (PDUs). This PDU stream is consumed by the receiver, which transforms it back into the original data stream. Additionally, the receiver produces a stream of acknowledgements. These are consumed and used by the sender to create a stream of the possibly needed retransmissions.

Understanding communication as a stream processing problem allows us to manage the whole protocol development process, i.e., *design, implementation, adaptation for hardware* and *execution on that hardware*, without any paradigm shift. The first step of the design process is to subdivide the protocol into separate tasks, so-called *Packet Processing Stages (PPSs)*, as shown in figure 1b). These PPSs are predefined building blocks, which implement data independent steps of the protocol processing. The desired protocol behavior is achieved by connecting the PPSs according to the data flow. The data flow is represented by directed edges. In [3] we showed how the combination of the stream processing programming approach, *soft real-time properties*, *hardware processing capabilities* and *data stream manipulation operators*, helps with the protocol design process.

The processing capabilities of a PPS indicate the number of streamed elements the PPS can process given a particular hardware. They are obtained by measuring the maximum achievable data rate on the hardware, which will process the PPS. That is possible because each PPS only depends on their internal state and the data at the input, which we can simulate. Figure 1c) shows a PPS measurement setup. The measured PPS is mapped onto the investigated execution hardware, e.g., a general purpose processor, and is allowed to utilize the hardware to the full extent. Each input of the PPS is connected to a stream producer, and each output is connected to a stream consumer. The stream producer generates a very high data rate stream, which fully utilizes the measured PPS. The processing capability of the PPS is the maximum data rate it was able to process.

The soft real-time requirements are an inherent property of each protocol, as they state how each part of a protocol has to perform to meet the desired data rate. The soft real-time requirements are estimated by analyzing the protocol given the desired data rate, which is either given by the use case, e.g., the data rate of a high definition video stream, or by the theoretical

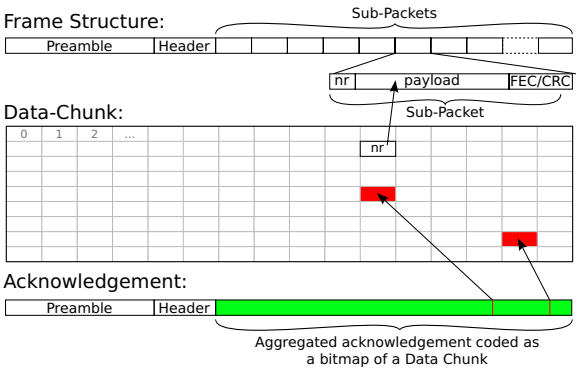

Fig. 2. The data structures for our prototype link layer protocol.

maximum data rate of the used communication technology. However, hidden data and processing dependencies can make the real-time analysis a cumbersome procedure. The stream processing approach eases the real-time analysis significantly because the real-time requirements for the building blocks can be derived directly from the data rate of the data flow (Fig. 1d).

By comparing the real-time requirements and the processing capabilities, shown in figure 1d), it is clear that the destination hardware is not able to process the protocol. The stream processing approach can be used to conveniently parallelize the protocol processing without any changes to the protocol, as it is shown in figure 1e). The parallelization is carried out with the help of *Stream Operators (SOs)*. A SO is used to manipulate the data stream, e.g., splitting the stream into substreams.

However, some tasks, such as a compute intensive Forward Error Correction (FEC), are too demanding for a certain general purpose CPU. In such a case a possible solution is to offload the task onto a better suited external accelerator as shown in figure 1f). The stream processing approach allows convenient offloading of PPSs. The only requirement for offloading a task is that an interface exists, which manages the data format transformation and the transport of the streamed data.

After the design process is finished, the protocol is already adapted for the target hardware and can be used on the hardware without further modifications.

III. Protocol

In this section, we will present our high throughput link layer protocol, which was developed by means of the aforementioned stream processing approach. The basic idea for the protocol was presented in [3]. It is designed for scenarios which need (wireless) communication at ultra high data rates of up to 100 Gbit/s and beyond in a streamed manner. Such scenarios include the transmission of high volume media streams, and the connection of two data-centers. However, the design and processing approach proposed in section II can be used to design protocols for any other scenario as well.

The protocol is based on *data-chunks* as shown in figure 2. A data-chunk is a large amount of memory, i.e., several

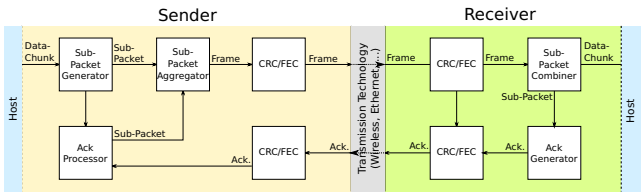

Fig. 3. Protocol Graph of our prototype Ultra High Throughput Data Link Layer Protocol.

megabyte. Using data-chunks reduces the management effort for the host to a feasible amount of a few hundred data-chunks, compared to several million of Ethernet frames or TCP packets. Each data-chunk has a sequence number and is cut into a stream of payload pieces, which are coded as *sub-packets*. These sub-packets are aggregated into larger frames, as shown in figure 2, which solves the contradiction between the need for large frames (reduced overhead ratio), and the need for small frames (minimize possible data loss)[3]. Each sub-packet combines the payload and some metadata, e.g, the sub-packet's position in the data chunk. Additionally, each sub-packet contains a CRC/FEC code for error discovery and recovery. By adding the metadata to each sub-packet, we can use partially corrected frames. Furthermore, it enables us to use very fine grained selective retransmissions using aggregated acknowledgements, as well as avoiding reordering at the sub-packet level because each sub-packet can be copied to its final position in the data-chunk.

The protocol processing is separated into a transmission phase and a retransmission phase, to avoid the over-utilization of the communication channels. Over-utilization could happen, because we already use the entire capacity of the communication channel for the initial transmission of a data-chunk. Therefore, each additional sub-packet due to retransmissions would exceed the channel's capacity. The two phases are explained in the following.

Transmission phase – In the first step a data-chunk, provided by a producer, e.g., the host, is split into sub-packets. Each sub-packet is given a data-chunk sequence number and a sub-packet sequence number, which determines the position of the sub-packet in the data-chunk. The sub-packets are then aggregated into frames. Before the frames are handed over to the PHY-Layer, the CRC and the FEC are calculated for each sub-packet. When all sub-packets are transmitted the sender switches to the retransmission phase, i.e., incoming aggregated acknowledgements are not ignored anymore.

On the receiver side, each incoming frame is checked for erroneous sub-packets, which are repaired if possible. For each usable sub-packet, i.e., correctly transmitted or repaired, the according bit in the aggregated acknowledgement bitmap is set to one and the acknowledgement transmission timeout is reset. Finally, the sub-packet's payload is copied to its position in the receiver's data-chunk buffer.

The transmission of the aggregated acknowledgement is triggered when a data-chunk is completely received or when the acknowledgement transmission timeout has fired. The acknowledgement transmission timeout stays active, issuing

acknowledgements for the data-chunk, until a new data-chunk is transmitted or the end of the transmission is signaled.

Retransmission Phase – When all sub-packets of a data-chunk are transmitted once, the sender switches to the retransmission phase, i.e., incoming acknowledgements are processed. Each incoming acknowledgement contains the bitmap of the currently processed data-chunk. This bitmap is scanned for missing sub-packets. For each bit which is zero, the corresponding sub-packet is created. As in the transmission phase, the sub-packets are aggregated into frames and enhanced with CRC and FEC codes. When an acknowledgement signals the complete transmission of a data-chunk, the sender switches back to the transmission phase, and starts the transmission of the next data-chunk.

A. Protocol Processing Stages

The protocol is implemented in a stream processing fashion, as shown in figure 3. The corresponding Protocol Processing Graph is assembled with the following PPSs.

- **Sub-Packet Generator** - The Sub-Packet Generator receives a data-chunk at the input. The data-chunk is cut into sub-packets, which are forwarded to the output. Each sub-packet is assigned a sequence number.
- **Sub-Packet Aggregator** - The Sub-Packet Aggregator aggregates sub-packets into frames.
- **Ack Processor** - The Ack Processor receives Acknowledgements (Ack) at its input. For each sub-packet, that could not be repaired by the FEC stage, i.e., a zero in the acknowledgement bitmap, a sub-packet is generated and sent. When all sub-packets are transmitted correctly, the sending host is informed.
- **Sub-Packet Combiner** - The Sub-Packet Combiner receives frames, and copies the faultless sub-packets into a data-chunk buffer. Each faultless sub-packet is forwarded to the Ack Generator.
- **Ack Generator** - The Ack Generator receives sub-packets at its input. For each sub-packet, the corresponding bit in the acknowledgement bitmap is set to one, and the acknowledgement timeout is reset. When either a timeout occurs or all sub-packets are received, the aggregated acknowledgement is sent.
- **CRC/FEC Stage** - The CRC/FEC Stage receives frames. It calculates the CRC/FEC for all sub-packets in the frame and forwards the frame to the output.

IV. Scaling the Protocol

In this section, we will show the feasibility of our protocol processing approach for high throughput communication. We will scale the aforementioned protocol for our high throughput protocol evaluation system and demonstrate the flexibility of the stream processing approach with two scenarios. The motivation for different scenarios stems from the idea that different classes of applications have different requirements on the communication system. For example, a video conference application requires a low communication latency, while a storage system needs a high and stable throughput. The common approach to solving these differences in the requirements is to develop application specific protocols. Our approach allows us to employ the same protocol and the already developed PPS to fulfill different requirements just by rearranging the PPS into a new PPG. As example requirements we choose (1) *Stable Throughput* (Fig. 4a) and (2) *Low Latency* (Fig. 4b), which are explained in the following.

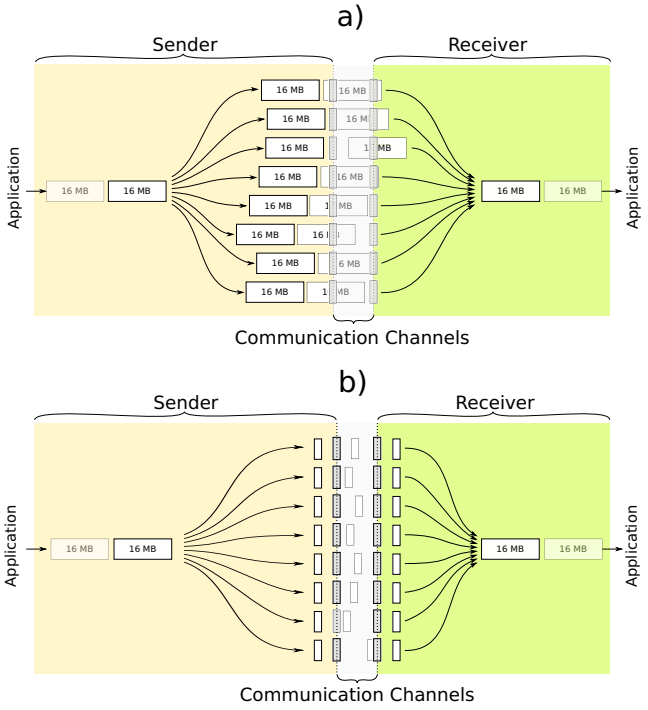

Fig. 4. a) *Stable Throughput* scenario: Each data-chunk is assigned to a transmission lane (i.e., an interface). All transmission lanes transmit different data-chunks in parallel. b) *Low Latency* scenario: Each data-chunk is assigned to the one lane, which has control over all transmission interfaces, i.e., one data-chunk is transmitted in parallel.

A. Scenarios

The idea behind the *Stable Throughput* scenario is that we can reduce the impact of the acknowledgement overhead, and also reduce the influence of unstable channels by transmitting several data-chunks in parallel, as shown in figure 4a). That can be achieved by providing one protocol processing pipeline per communication channel. This way an erroneous channel does not interfere with the transmission on other channels, which makes the overall transmission more robust and stable. The parallel protocol processing pipelines also help to hide the per data-chunk management overhead, which is expected to further increase the throughput. A downside of this approach is that the latency per data-chunk will stay constant, instead of decreasing for a higher number of interfaces.

The *Low Latency* scenario uses the available communication channels in a different manner. Instead of providing a protocol pipeline for each communication channel, the physical channels are combined into one logical channel by splitting the transmission of a data-chunk as shown in figure 4b). The data-chunk splitting can be achieved by using all communication

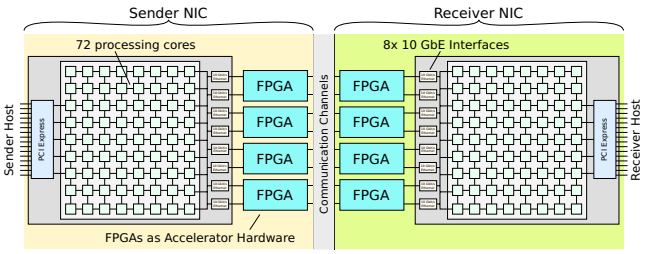

Fig. 5. Schematic view of the protocol processing embedded many-core.

channels for one protocol processing pipeline, which implicitly binds the physical channels together. In the case that all physical channels provide the same data rate and latency, the theoretical per data-chunk latency is expected to decrease by the factor of the number of combined channels. The downside of using only one protocol processing pipeline is, that an erroneous physical channel slows down the overall protocol processing, which makes this approach highly depending on low error rates. Additionally, the per data-chunk management overhead, e.g., waiting for the acknowledgement that the chunk was completely transmitted, is not hidden by the parallel transmission of data-chunks, as it is the case in the *Stable Throughput* scenario.

B. Real Time Requirements and Hardware Processing Capabilities

As mentioned in section II, the real-time requirements depend on the desired data rate and the protocol parameters. The target data rate and the protocol parameters are chosen with respect to our evaluation system.

Our evaluation hardware system configuration is shown in figure 5. It is separated into a sender side Network-Interface-Card (NIC) and a receiver side NIC, each consisting of an embedded 72 core TILEncore Gx72 [4] many-core board, which is responsible for the higher level packet processing, and four FPGAs for compute-intensive tasks, such as CRC and FEC calculation. The TILEncore Gx72 boards are equipped with eight 10 GbE Ethernet interfaces, which are used as communication channels.

Since we are focusing on achieving the highest possible throughput the evaluation hardware system can deliver, we define the maximum theoretical gross data rate of 80 Gbit/s (8 x 10 GbE) as the target data rate. This data rate can only be achieved if we do not lose any packets.

The protocol parameters needed for the calculation of the real time requirements are explained shortly in the following:

- **Sub-Packet –** 1216 Byte + 16 Byte (Header + CRC/FEC) = 1232 Byte
  The payload size of 1216 Byte is a multiple of the cacheline size of the TILEncore Gx72 many-core, which minimizes the memory access overhead. Additionally, 1216 Byte are close to the optimal block size of the used Reed-Salomon Forward Error Correction (FEC) code.
- **Frame Size –** 8 $\times$ sub-packets + 24 (Byte Header + CRC) = 9880 Byte

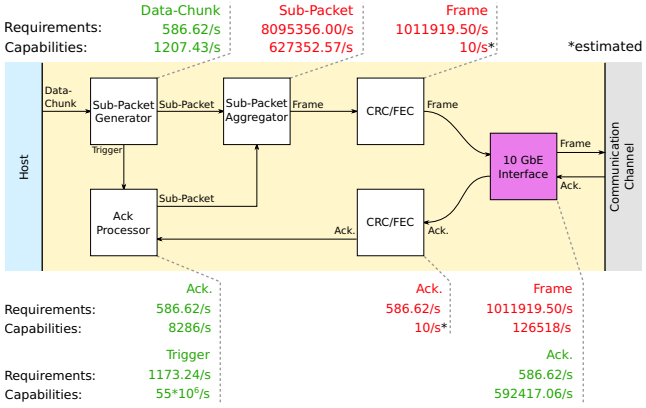

Fig. 6. The PPG for the Sender with the Soft Real-Time **Requirements** and the Processing **Capabilities** of the hardware.

  Here we are bound to the Ethernet hardware capabilities. We chose the largest frame size we were able to transmit.
- **Sub-Packets per Data-Chunk –** $13800\,\frac{\text{sub-packets}}{\text{data-chunk}}$
  The number of sub-packets is a multiple of 8. This is convenient for our aggregated acknowledgements, which are coded as a bitmap, as a multiple of 8 bit (1 Byte) simplifies the implementation of the Ack Processor and the Ack Generator Protocol Processing Stages. Using 13800 sub-packets per data-chunk $\times$ 1216 Byte payload per sub-packet leads to a data-chunk size of 16780800 Byte (~16Mb), i.e., the host has to manage less than 600 data-chunks per second. We found this a reasonable[5] tradeoff between the management effort for the host and the overhead induced by the aggregated acknowledgements.
- **Protocol Structure Overhead –** Given these parameters we have 13800 sub-packets $\times$ 16 Byte metadata + 13800 / 8 frames $\times$ 24 Byte = 262200 Byte protocol overhead per data-chunk. Thus, the protocol structure overhead is 1.56%, i.e., the maximum achievable net data rate for a 10 GbE Ethernet interface is 9.844 Gbit/s.

After defining the protocol parameters, the protocol's soft real-time requirements can be calculated. As the procedure is the same for every protocol, we focus only on the sender side. The soft real-time requirements are calculated beginning with the inputs of the PPG. In the case of the sender's protocol graph, these are the inputs of the Sub-Packet Generator and the PHY-Layer, which is represented by the 10 GbE interface. Due to the maximum achievable net data rate of 78.752 Gbit/s (8$\times$10 GbE - 1.56% protocol structure overhead), the Sub-Packet Generator has to process 586.62 data-chunks per second (78.752 Gbit = 78.752*10^9 Bit = 9.844*10^9 Byte / 16780800 Byte = 586.62). The Sub-Packet Generator produces 13800 sub-packets for every data-chunk. Therefore, the sub-packet output data rate of the Sub-Packet Generator is $8095356\,\frac{sub-packets}{s}$ (586.62 * 13800 sub-packet = 8095356 sub-packets). The trigger output sends two triggers for each

[5] A reasonable tradeoff keeps the involvement of the host as low as possible, while not affecting performance factors, such as latency and throughput.

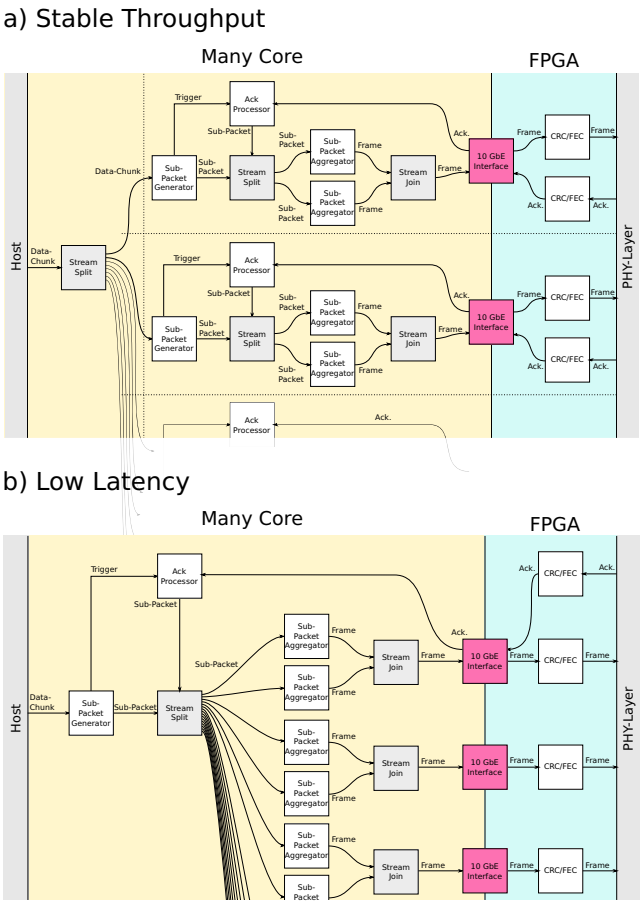

Fig. 7. a) *Stable Throughput* scenario: Parallel Lanes transmitting different chunks of data in parallel. b) *Low Latency* scenario: One data-chunk is split over several wireless channels.

data-chunk. Thus, the output data rate is $2{\times}586.62\frac{triggers}{s}$. The other input into the protocol graph is the PHY-Layer. As we estimated the packet loss rate with 0%, we will get exactly one acknowledgement for each data-chunk, thus the 10 GbE Interface has to process $586.62\frac{acks}{s}$, which is also its output data rate. The output data rates are now used as the input data rates for the remaining PPSs the same way, until all real time requirements are calculated.

The hardware processing capabilities for all PPSs but the CRC/FEC stage[6] were measured on our evaluation hardware as stated in section II. The hardware processing capability of the communication interface was calculated given its data rate of 10 Gbit/s. The soft real-time requirements and hardware capabilities for the sender side are shown in figure 6.

C. Adapting the Protocol Processing Graph

Comparing the soft real-time requirements with hardware's processing capabilities indicates that the processing hardware in combination with the protocol can not provide the desired data rate without further modification. The adaptation of the protocol graph starts with estimating the needed amount of parallelization. While not all PPSs have to be parallelized, we see that we need a minimum of 13 Sub-Packet Aggregators and eight 10 GbE interfaces, which is within the limits of the target communication system (72 Cores). However, the parallelization factor of 101191.95 for the CRC/FEC Stage is far beyond limits. Thus, we have to offload the processing of the CRC and FEC into an external accelerator hardware, in our case an external FPGA. In the following, we show briefly how the PPG of our protocol can be adapted to meet the soft real-time requirements, with respect to the aforementioned scenarios.

1) Stable Throughput: Figure 7a) shows a part of the PPG for the *Stable Throughput* scenario. The robust throughput is achieved by sending several data-chunks in parallel (see Fig. 4a). That behavior can be accomplished by splitting the data stream and parallelizing the Protocol Processing Graph (PPG) for each communication channel, i.e., we create independent protocol processing pipelines for each of the 10 GbE interfaces. Additionally, we have to split the output stream of the Sub-Packet Generator within each protocol processing pipeline, because one *Sub-Packet Aggregator* is not able to provide a data rate of 10 Gbit/s, needed to fully utilize a 10 GbE interface. As mentioned before, the CRC/FEC processing stage has to be offloaded. Due to the stream processing approach, the offloading of the CRC/FEC into an external FPGA is a matter of having a PPS implementation and a communication interface (10 GbE) for the accelerator hardware. More information about the FPGA and the FEC implementation can be found in [5] and [6].

2) Low Latency: Figure 7b) shows a part of the PPG for the *Low Latency* scenario arrangement. As mentioned before, low latency is achieved by transmitting one data-chunk in parallel over several communication channels (see Fig. 4b). We can obtain this behavior by using one Sub-Packet Generator and splitting its sub-packet output into several substreams, which are processed in parallel. The sub-packets transmitted over these substreams are aggregated into frames by parallel Sub-Packet Aggregators. To be able to fully utilize a 10 GbE interface, we join two sub-packet streams for transmission over one 10 GbE interface. As in the other scenario, the CRC/FEC calculations are offloaded into FPGAs.

D. Implementation and Hardware Mapping of the Protocol Processing Graph

We implemented the presented scenarios with the help of our low latency stream processing framework StRIPEs - Stream Interconnected Processing Engines. StRIPEs combines the PPSs and SOs and provides a lightweight execution environment, which manages the streaming of the data between the PPSs and adds support for memory management. The framework is executed on the embedded TILEncore Gx72 many-cores. The mapping of the PPGs is shown in the figures 8 and 9. The Protocol Processing Stages (PPSs) are assigned

[6]The value of the CRC/FEC PPS's processing capability is an estimation. Since it is conceivable that calculating CRC/FEC codes in software is not reasonable for high data rates, we did not implement a software PPS for the CRC/FEC calculation.

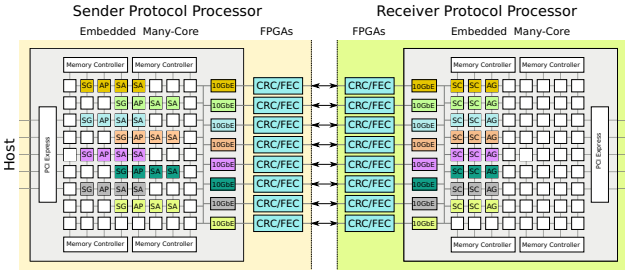

Fig. 8. The hardware mapping of the PPG for the *Stable Throughput* scenario. (Sub-Packet Generator (SG), Sub-Packet Aggregator (SA), Sub-Packet Combiner (SC), Ack Processor (AP), Ack Generator (AG)

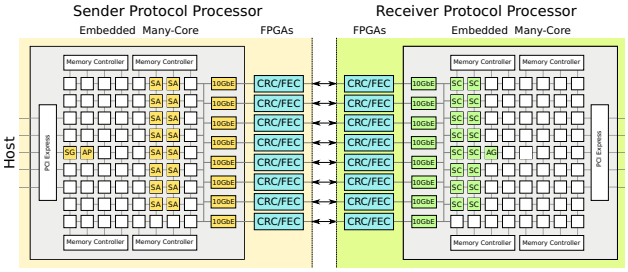

Fig. 9. The hardware mapping of the PPG for the *Low Latency* scenario. (Sub-Packet Generator (SG), Sub-Packet Aggregator (SA), Sub-Packet Combiner (SC), Ack Processor (AP), Ack Generator (AG)

to the many-core according to the data stream to minimize the probability of interferences between the data flows.

V. Scenario Evaluation

In this section, we present the evaluation results for the two scenarios. The results were obtained on our evaluation system with a throughput and a latency benchmark. The benchmarks measure the influence of lost packets by marking a defined percentage of the sub-packets as unrecoverable. The measurements were made with and without the external FPGA (CRC/FEC calculation).

The experiments do not include the host, because the maximum throughput between the host and the embedded many-core is limited to ~64 Gbit/s, which is lower than our target data rate of 80 Gbit/s. Instead we transmit test data that is provided by the embedded many-core itself.

A. Throughput and Latency Without FPGA (CRC/FEC)

The achieved throughput for both scenarios is shown in figure 10. The figure shows the results depending on the number of used 10 GbE interfaces and the simulated packet loss. As one can see, the throughput for the *Stable Throughput* scenario is close to the theoretical maximum of the 10 GbE interfaces. The theoretical maximum for the *Stable Throughput* scenario is missed by a total overhead of 0.205 Gbit/s when one interface is used, and missed by 1.685 Gbit/s when all eight interfaces are used (Tab. I). The total overhead is composed of the protocol structure overhead and the protocol processing overhead. The absolute protocol structure overhead (in Gbit/s) can be calculated considering the net throughput and the relative protocol structure overhead of 1.56%. Subtracting the

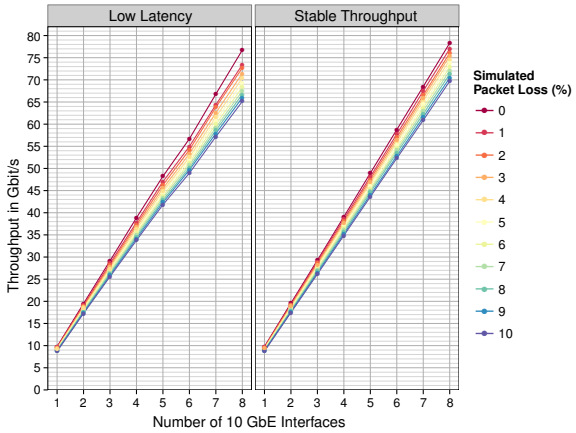

Fig. 10. Throughput comparison for the *Low Latency* and the *Stable Throughput* scenario, depending on the number of used 10 GbE interfaces.

TABLE I
THROUGHPUT AND LATENCY RESULTS FOR THE *Stable Throughput* SCENARIO WITH 0% PACKET LOSS

Number of Interfaces	Theoretical Max (Gbit/s)	Net Throughput (Gbit/s)	Total Overhead (Gbit/s)	Protocol Overhead (1.56%) (Gbit/s)	Processing Overhead (Gbit/s)	Processing Overhead (%)	Latency (ms)
1	10	9.795	0.205	0.155	0.050	0.500	13.705
2	20	19.591	0.409	0.310	0.099	0.495	13.702
3	30	29.355	0.645	0.465	0.180	0.600	13.715
4	40	39.011	0.989	0.618	0.371	0.927	13.758
5	50	48.963	1.037	0.776	0.261	0.522	13.705
6	60	58.755	1.245	0.931	0.314	0.523	13.704
7	70	68.494	1.506	1.085	0.421	0.601	13.714
8	80	78.315	1.685	1.241	0.444	0.555	13.109

TABLE II
THROUGHPUT AND LATENCY RESULTS FOR THE *Low Latency* SCENARIO WITH 0% PACKET LOSS

Number of Interfaces	Theoretical Max (Gbit/s)	Net Throughput (Gbit/s)	Total Overhead (Gbit/s)	Protocol Overhead (1.56%) (Gbit/s)	Processing Overhead (Gbit/s)	Processing Overhead (%)	Latency (ms)
1	10	9.789	0.211	0.155	0.056	0.560	13.714
2	20	19.497	0.503	0.309	0.194	0.970	6.881
3	30	29.142	0.858	0.462	0.396	1.320	4.604
4	40	38.859	1.141	0.616	0.525	1.313	3.366
5	50	48.232	1.768	0.764	1.004	2.008	2.782
6	60	56.693	3.307	0.898	2.409	4.015	2.366
7	70	66.796	3.204	1.059	2.145	3.064	2.008
8	80	76.73	3.27	1.216	2.054	2.568	1.748

calculated protocol structure overhead from the total overhead, results in the protocol processing overhead, which is a measure for the scalability. The values are shown in table I.

The protocol processing overhead (acknowledgement overhead and stream processing) accounts for 0.5% of the theoretical data rate for 1 interface (0.05 Gbit/s of 10 Gbit/s) and 0.555% for the theoretical data rate of 8 interfaces (0.444 Gbit/s of 80 Gbit/s). That is a difference of only 0.055%. These numbers indicate that the stream processing approach causes little overhead for the *Stable Throughput* scenario and that the protocol throughput of the *Stable Throughput* scenario scales nearly linear. However, a closer look at the protocol processing overhead reveals that it does not grow monotonously, but fluctuates between a minimum of 0.495% (2 interfaces) and a maximum of 0.927% (4 interfaces). This indicates further, that the parallelization overhead is not the dominant part of the processing overhead. The most probable reason for the fluctuating overhead is, that additional hardware parameters, e.g., a varying memory access latency, have a bigger impact on the overhead, than the protocol processing itself. Finally, the results confirm the prediction, that the *Stable Throughput* scenario is robust against packet loss.

As expected, the throughput of the *Low Latency* scenario is lower than the *Stable Throughput* scenario. While the throughput for one interface can be considered the same, the protocol processing overhead of the *Low Latency* scenario increases steeper than the *Stable Throughput* scenario (compare Tab. I and Tab. II). The first reason for the higher increase is that the acknowledgement overhead cannot be hidden by transmitting several chunks in parallel. On the contrary, the ratio between acknowledgement overhead and transmission grows, due to the protocol's current implementation, as more interfaces are used for the transmission, which is also the reason why the *Low Latency* scenario is more affected by lost packets than the *Stable Throughput* scenario. Additionally, the current implementation of the protocol does not employ a sophisticated memory allocation strategy for the data-chunks. Therefore, its more likely that several Protocol Processing Stages access the same memory controller at the same time when reading or writing to the data-chunk.

The achieved latency per data-chunk is shown in figure 11. The figure shows the results depending on the number of used 10 GbE interfaces and the simulated packet loss. One can see that the latency of the *Low Latency* scenario scales indirectly proportional with the number of used interfaces (Tab.II). Considering the latency per data-chunk for one communication interface as the base value, the perfectly scaled latency for eight interfaces would be $13.714ms/8 = 1.714ms$. The measured latency for eight interfaces is $1.748ms$, which is a difference of only $34\mu s$. As expected, the latency of the *Stable Throughput* scenario can be considered as constant (see table I).

B. Full System with FPGA

At the time this publication was written, we had only one FPGA for measurements available. However, designing

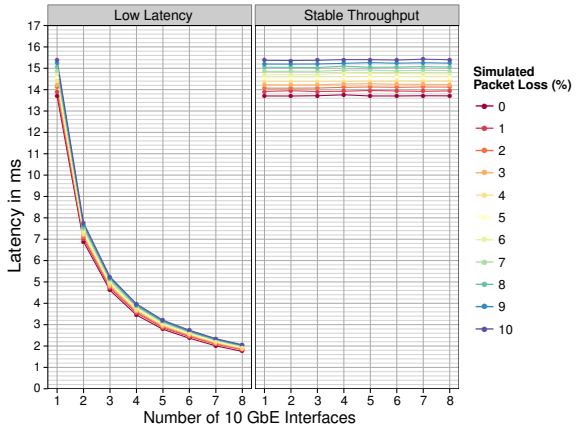

Fig. 11. Latency comparison for the *Low Latency* and the *Stable Throughput* scenario, depending on the number of used 10 GbE interfaces.

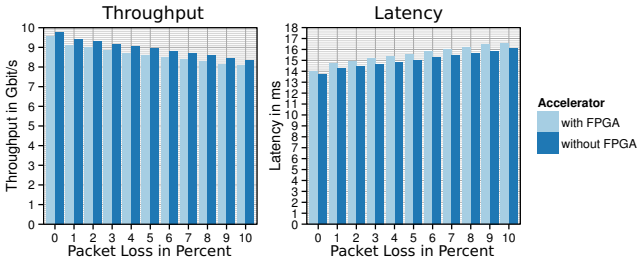

Fig. 12. Throughput of one lane for the complete protocol processing system (with FPGA) depending on packet loss and the number of used interfaces.

the CRC/FEC as an offloaded Protocol Processing Stage, the results will scale with the number of used FPGAs. The achieved throughput and latency for the complete protocol processing system, i.e., with external FPGA for Cyclic Redundancy Check (CRC) and FEC calculation, is shown in figure 12. The results show, that the throughput is lower, and the latency is higher when the FPGA is used. This is caused by the latency added due to the FEC processing on the FPGA, as it increases the time it takes the sender to realize that a data-chunk is completely transmitted. The impact can be lowered by either choosing larger data-chunks or by overlapping the transmission of data-chunks with the transmission of acknowledgements.

VI. Related Work

This work was inspired by COPRA [7]. COPRA is a COmmunication PRocessing Architecture for wireless sensor networks. In COPRA, different predefined protocol building blocks are combined to form a protocol. The modularity of the building blocks enables the developer to connect exactly the needed protocol parts for each application, which offers a high flexibility. However, in contrast to our approach, COPRA was designed for 16 bit micro controllers and a lossy event flow, which is acceptable for hard real-time problems, but not for flow controlled communication.

The communication protocol stack is traditionally located

within the operating system's kernel. That means applications have to switch into the kernel space for every network operation, which makes it unfeasible for high data rate communication. Additionally, it makes the protocol stack inflexible, as each change implies modifications at the operating system's kernel. In [8] the authors propose to move the protocol stack into the user space to ease the deployment of new protocols, protocol improvements and extensions. We agree with the authors that the protocol processing has to move from the kernel, but doubt that a simple user space protocol stack will enable data rates of 100 GBit/s end-to-end communication, because the protocol processing still uses the host's resources, e.g., CPU time and memory, which are lost for the host's actual task, e.g., performing database queries.

RouteBricks [9] is an extension to the Click Modular Router [10] which introduces parallel processing to exploit the processing power of multi-processor off-the-shelf server systems. Like ours, their approach was twofold: Firstly, they showed that a throughput requirement, which cannot be fulfilled by a single server, can be fulfilled by a cluster of servers and that the throughput scales linearly with the number of used machines. Secondly, they made clear that the full packet processing capability of a single server can only be exploited when the packet processing is parallelized. Besides the use case (RouteBricks is a router, we introduced an integrated parallel NIC approach), the authors still use the host computers processing power to process packets, while we argue that new protocols are needed, and that the packet- / protocol-processing should be offloaded to specialized hardware.

NetSlices [11] is an approach to make the packet processing more efficient by replacing the Unix raw socket with a parallel approach. The authors have three main contributions: 1. They state that the actual packet processing should not be done within the kernel because memory placement, CPU affinity, and the process isolation are hard to control. 2. The packet processing should be parallelized. The authors propose a mapping where the user- and kernel-space packet processing is coupled closely, to reduce cache misses. 3. As long as the packet processing application has no control over the hardware, it will not be able to come close to the theoretically available performance. We agree with all the stated arguments. However, we think packet processing should neither be carried out in the kernel nor within the host machine's user space. In fact, the authors approach can be sufficient as long the host is a dedicated protocol processing machine with one task, e.g., routing or deep packet inspection. In cases the packet processing is only a service for the host machines actual task, e.g., managing a big database, the authors approach will take processing power and memory bandwidth, leaving only a fraction of it to the actual tasks.

VII. Conclusion and Future Work

In this paper, we presented a new approach for the design of scalable high performance communication protocols. We showed how our soft real-time stream processing approach can be used to determine the needed amount of parallelism and how a Protocol Stream Graph can be adapted with the help of Stream Operators. Additionally, we presented an easy way to use external accelerator hardware to offload compute intensive tasks. Finally, we presented the evaluation results, which showed that our approach works in practice and gives the expected throughput and latency behavior.

The next step is to integrate more processing parameters, e.g., memory bandwidth, into the planning approach, which will allow the protocol designer to adapt the protocol processing graph according to these hardware capabilities. Additionally, we plan to replace the manual mapping of PPSs onto CPUs with a dynamic approach. This will allow us to react to changing environment parameters, such as the Bit Error rate (BER), by increasing or decreasing the amount of parallelism. In the future, we also plan to implement a high throughput MAC-Protocol for wireless transmission technology and provide an accurate simulation of the channel behavior.

Acknowledgement

This work is part of the End2End100 Project which started in 2013 and is funded by the German Research Foundation (DFG), DFG NO 625/9-1.

References

[1] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel, "Cache coherence protocol and memory performance of the intel haswell-ep architecture," in *Parallel Processing (ICPP), 2015 44th International Conference on*, Sept 2015, pp. 739–748.

[2] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.

[3] S. Büchner, J. Nolte, R. Kraemer, L. Lopacinski, and R. Karnapke, "Challenges for 100 gbit/s end to end communication: Increasing throughput through parallel processing," in *40th Annual IEEE Conference on Local Computer Networks (LCN 2015)*, Clearwater Beach, USA, Oct. 2015, pp. 607–610.

[4] *TILEncore-Gx72 Intelligent Application Adapter*, Mellanox® Technologies, Product Brief Rev 1.4, 2016.

[5] L. Lopacinski, M. Brzozowski, R. Kraemer, J. Nolte, and S. Buechner, "Design and implementation of an adaptive algorithm for hybrid automatic rrepeat request," in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), 263 (2015)*, 2015.

[6] L. Lopacinski, J. Nolte, S. Buechner, M. Brzozowski, and R. Kraemer, "100 gbps wireless - data link layer vhdl implementation," in *Proc. of the 18th Conference on Reconfigurable Ubiquitous Computing 2015, (2015)*, 2015.

[7] R. Karnapke and J. Nolte, "Copra - a communication processing architecture for wireless sensor networks," in *Euro-Par 2006 Parallel Processing*. Springer, 2006, pp. 951–960.

[8] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo, "Rekindling network protocol innovation with user-level stacks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 52–58, Apr. 2014.

[9] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 15–28.

[10] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[11] T. Marian, K. S. Lee, and H. Weatherspoon, "Netslices: Scalable multi-core packet processing in user-space," in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '12. New York, NY, USA: ACM, 2012, pp. 27–38.