Applying the Stream Processing Paradigm to Ultra High-Speed Communication Systems

Von der Fakultät MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik der Brandenburgischen Technischen Universität Cottbus-Senftenberg genehmigte Dissertation zur Erlangung des akademischen Grades eines

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

vorgelegt von

Diplom Informatiker (Dipl.-Inf.)
Steffen Büchner

geboren am 11.09.1982 in Frankfurt Oder

Vorsitzender: Prof. Dr. Michael Hübner
Gutachter: Prof. Dr. J. Nolte
Gutachter: Prof. Dr. R. Kraemer
Gutachter: Prof. Dr. W. Schröder-Preikschat

Tag der mündlichen Prüfung: 03.09.2020

Abstract

In the last 30 years, communication became one of the most important pillars of our civilization. Every day terabytes of information are moved wired and wireless between computers. In order to transport this amount of data, researchers and industry increase the data rates of the underlying communication networks with impressive speed. However, such ultra-high data rates are unavailable at the communication endpoints.

One reason why ultra-high data rates are still not available for the communication endpoints is their inability to handle the protocol processing at this data rate. In order to enable communication endpoints to process high-volume data streams, the protocol processing has to be parallelized and optimized on all processing levels. However, parallelization and optimization are cumbersome tasks, which are further complicated as the protocol processing is traditionally carried out by the operating system.

This thesis aims at circumventing these problems by moving the protocol processing into external processing hardware and interpreting communication protocols as stream processing problems. In order to achieve ultra-high data rates at the communication endpoints, a protocol stream processing design approach was developed and evaluated. The design process is separated into implementation, soft real-time analysis, parallelization, and mapping steps, which allow a scalable protocol implementation without paradigm changes. Furthermore, a data link protocol for 100 Gbit/s wireless was developed and implemented with the new stream processing design concept, in order to show its feasibility. The data link protocol is configurable for different communication conditions and easy to parallelize by providing different granularities of packets. The proposed design-process has shown to be suitable for uncovering bottlenecks and helping with debugging the individual stages of the protocol.

Zusammenfassung

In den letzten 30 Jahren wurde die ständig verfügbare elektronische Kommunikation zu einer der wichtigsten Säulen unserer Zivilisation. Jeden Tag werden Terabyte an Informationen drahtgebunden und drahtlos zwischen Computern übertragen. Um diese Datenmenge zu transportieren, erhöhen Forscher und Industrie die Datenraten der zugrunde liegenden Kommunikationsnetze mit beeindruckender Geschwindigkeit. Solche ultrahohen Datenraten sind jedoch an den Kommunikationsendpunkten nicht verfügbar.

Ein Grund, warum für die Kommunikationsendpunkte immer noch keine extrem hohen Datenraten zur Verfügung stehen, ist die Unfähigkeit, die Protokollverarbeitung mit dieser Datenrate durchzuführen. Damit Kommunikationsendpunkte hochvolumige Datenströme verarbeiten können, muss die Protokollverarbeitung auf allen Verarbeitungsebenen parallelisiert und optimiert werden. Parallelisierung und Optimierung sind jedoch komplizierte und fehleranfällige Aufgaben. Diese Herausforderung wird weiter verstärkt, da die Protokollverarbeitung traditionell im Betriebssystemkern durchgeführt wird.

In dieser Arbeit werden diese Herausforderungen gelöst, indem die Protokollverarbeitung in externe Verarbeitungshardware ausgelagert wird und indem Kommunikationsprotokolle als Stream Processing Probleme interpretiert werden. Um ultrahohe Datenraten an den Kommunikationsendpunkten zu erreichen, wurde ein Designansatz für die Verarbeitung von Protokolldatenströmen entwickelt und bewertet. Der Designprozess ist in Implementierungs-, Weiche-Echtzeitanalyse-, Parallelisierungs- und Mapping-Schritte unterteilt, mit welchen eine skalierbare Protokollimplementierung ohne Paradigmenwechsel ermöglicht wird. Darüber hinaus wurde ein Datenverbindungsprotokoll für drahtlose 100 Gbit/s Kommunikation entwickelt und mit dem neuen Stream Processing Designkonzept implementiert, um dessen Leistungsfähigkeit zu demonstrieren. Das Datenverbindungsprotokoll ist für verschiedene Kommunikationsbedingungen konfigurierbar und lässt sich durch unterschiedliche Granularitäten der Pakete leicht parallelisieren. Der vorgeschlagene Entwurfsprozess hat sich als geeignet erwiesen, Verarbeitungsengpässe aufzudecken und bei der Fehlersuche in den einzelnen Phasen der Entwicklung zu helfen.

Contents

1 Introduction **1**

1.1 Disclaimer . . . 5

2 Stream Processing and Communication Protocol Processing **7**

2.1 Solutions from the Networking Community . . . 8

2.1.1 User Space Protocol Processing . . . 9

2.1.2 Hardware Acceleration by Offloading . . . 13

2.1.3 Flexible Protocol Processing . . . 18

2.2 Stream Processing Architectures and Approaches . . . 23

2.3 Real-Time Analysis . . . 29

2.4 Summary & Conclusion . . . 31

3 Soft Real-Time Stream Processing **37**

3.1 Decomposition of Protocols into Processing Stages . . . 38

3.2 Soft Real-Time Analysis . . . 40

3.2.1 Soft Real-Time Requirements . . . 40

3.2.2 Performance Measurement . . . 41

3.3 Adaptation of the Processing Engine and Offloading of Stages . . . 42

3.3.1 Stream Operators . . . 43

3.3.2 Adaptation . . . 44

3.3.3 Offloading . . . 44

3.4 On-Demand Adaptation of Processing Engines . . . 45

3.4.1 Readaptation of a Deployed Processing Engine . . . 46

3.4.2 Switching Protocols by Entirely Replacing Processing Engines . . . 47

3.4.3 Exchanging Processing Engines . . . 49

4 A 100 Gbit/s Data Link Protocol **51**

4.1 The Prototyp Data Link Protocol . . . 51

4.1.1 Protocol Description . . . 55
4.1.2 Summary . . . 57
4.2 Decomposition of the Protocol into Processing Stages . . . 58
4.2.1 Implementation of the Processing Stages . . . 61
4.2.2 Summary . . . 63
4.3 Soft Real-Time Analysis . . . 63
4.3.1 Soft Real-Time Requirements . . . 64
4.3.2 Performance Characteristics . . . 69
4.3.3 Summary . . . 78
4.4 Adaptation of the Processing Engine . . . 78
4.5 Mapping of the Processing Engine . . . 82
4.6 Testing of the Implementation and Latency Hiding . . . 84
4.6.1 Integration of external Accelerator Hardware . . . 89
4.6.2 Summary . . . 90
4.7 Evaluation . . . 90
4.7.1 Processing Overhead . . . 91
4.7.2 Scenarios . . . 94
4.7.3 Beyond Channel Bonding . . . 102

5 Conclusion **107**
5.1 Own Publications used in this thesis . . . 109

A Supplementary Technical Information **111**
A.1 Exchanging Processing Engines . . . 111
A.2 Offloading of Stages . . . 112

B Data Link Protocol **117**
B.1 Protocol Data Structure . . . 117
B.1.1 DataChunk . . . 117
B.1.2 Virtual Channel Header . . . 118
B.1.3 Frame and SuperFrame . . . 118
B.1.4 Sub-Packet . . . 119
B.2 Description of the Protocol Processing Stages . . . 121
B.2.1 Message Types . . . 121
B.2.2 Data-Packet Generator (DG) . . . 122
B.2.3 Acknowledgement Processor (AP) . . . 124

B.2.4 Data-Packet Combiner (DC) 131
B.2.5 Acknowledgement Generator (AG) 134
B.2.6 Performance measurements for The DA with incoherent memory 138
B.2.7 Service Stages 139
B.3 Full Protocol Processing Engine Description 143

Bibliography **155**

CHAPTER 1

Introduction

Fast, reliable communication is the backbone of today's life. However, the ever-increasing amount of data-traffic is raising challenges. In order to be able to handle the increasing amount of traffic, the theoretically achievable data rate of modern communication systems is pushed into new spheres of several hundreds of gigabit per second.

For example, the processing capacity of routers and switches has increased from 52 Gbit/s in the early 2000 [1] to a staggering 96 Tbit/s half-duplex in 2019 [2]. Likewise, today's networks are connected with 100 Gbit/s Ethernet, and 400 Gbit/s Ethernet is about to be deployed [3]. The Ethernet roadmap envisions line-rates of more than 1 Tbit/s for as near as mid-2020 [4], and research already showed the feasibility of wireless data rates of 100 Gbit/s and more [5, 6].

However, these data rates stress the capacities of today's processing hardware to the utmost limits and beyond. Consequently, these data rates are used almost exclusively at the network backbone because today's protocol processing approaches for the endpoints are not able to handle data streams at these line-rates. The following example from [7] shall 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 [8]), indicates that we have to think of new protocol processing paradigms." [7]

This short example outlines the problem faced nowadays in wired networks. The challenge grows when taking wireless connections into account, because individual wireless communication channels may provide a lower than necessary data rate and must be combined. Moreover, because wireless connections are much less stable than wired connections, the effort for retransmission increases drastically and can render the assigned resources insufficient. Consequently, the communication system must be reconfigured, for example, by increasing the assigned processing resources. It is quite clear that a server that is already at the edge of its processing capacity can not bear the additional processing costs.

To be able to utilize the theoretically achievable data rates available to the communication endpoints, one of the main questions for today's high-speed network research is how to overcome the mismatch of processing requirements and capacities.

Generally, the answer is threefold: We need new parallelizable protocols, the protocol processing has to be parallelized on all levels, and special purpose hardware must be intelligently integrated to decrease the protocol processing costs for the host. These general answers lead to new questions:

- How much parallelization is necessary and how to minimize synchronization overhead induced by parallel processing?
- What protocol tasks should be offloaded to dedicated hardware and how can these parts be integrated into the communication system?
- How can reconfigurable protocols be designed and processed in order to react to changing communication conditions?
- How can all this be implemented without increasing the design and implementation complexity?

The central idea proposed in this thesis is that the implementation can be handled in a structured manner by understanding communication protocols as a stream processing problem [9]. Generally, all communication systems can be described by a stream processing graph, as shown for a generalized communication system in figure 1.1 (top). Each communication system consists of a sender, `S`, which receives a stream of data and transforms it into a stream of Protocol Data Units (PDUs) and a receiver, `R`, which consumes this PDU stream and transforms it back into the original data stream. Additionally, the

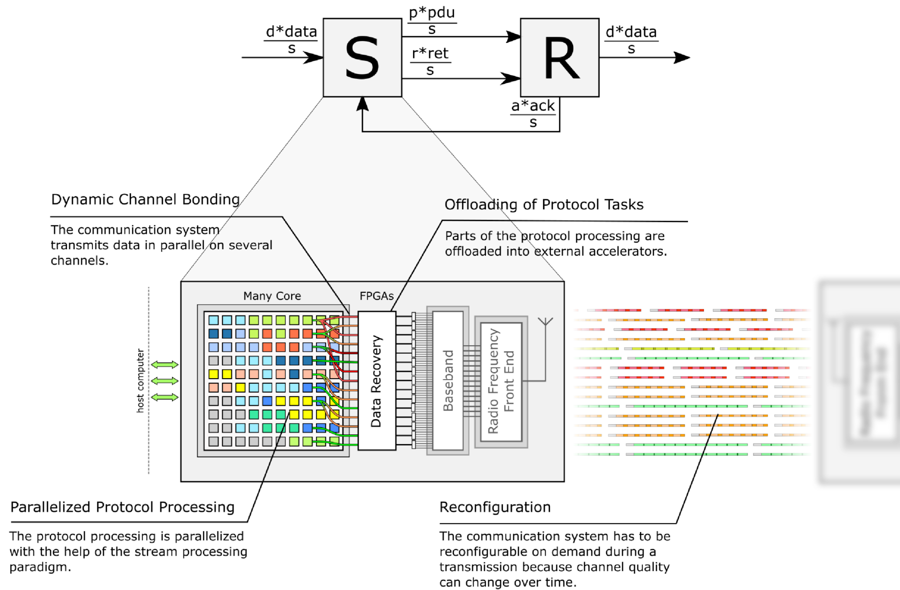


Figure 1.1: A soft real-time stream processing representation for communication protocols with retransmissions (top). A "smart" network interface card consisting of a manycore processor and Field Programmable Gate Array (FPGA) for the protocol processing (bottom).

receiver produces a stream of acknowledgments (ack), which are then consumed and used by the sender to create a stream of retransmissions (ret).

In order to avoid performance degradation, all parts of the communication system have to meet the processing deadlines dictated by the desired data rate. Consequently, the stream processing graph of this generalized communication protocol formulates a soft real-time problem that can be analyzed as such. In order to be able to estimate the necessary processing resources, the stream processing graph is augmented with the protocol's soft real-time processing requirements and the processing hardware's performance characteristics. The soft real-time requirements are given implicitly by the data rate of the incoming data stream. For example, the sender `S` has to process $d \times data + a \times ack$ per second. The performance characteristics, i.e., the highest data rate that can be processed by the hardware, can be measured individually for each stream-stage on the processing hardware. An analysis of the ratio between requirements and performance characteristics allows the protocol developer to identify processing bottlenecks and derive a static parallel processing schedule.

On that basis, a communication system is designed so that it is parallelized on all levels and offloads parts of the protocol processing into special purpose hardware, as depicted in figure 1.1 (bottom). The higher level protocol processing, e.g., the buffer- and retransmission-management, is processed in parallel by an embedded manycore, that manages and combines several communication channels for the transmission. Compute-intensive protocol tasks, such as forward error correction, are offloaded into special purpose hardware. In case the communication conditions change, the stream processing paradigm is well suited for readaptation of the protocol processing because the individual protocol tasks are connected only by streams that can be reconnected.

These corner-points are elaborated and evaluated in this thesis, leading to the following main contributions:

1. A stream processing based methodology that integrates the design, implementation, analysis, parallelizing, and offloading of communication protocols without paradigm changes.

2. A concept for on-demand protocol readaptation at runtime without increasing the protocol complexity and the demonstration of its feasibility for ultra-high speed communication.

3. A parallelizable data link protocol for wireless communication that is able to utilize a data rate of 80 Gbit/s with an overhead of less than 0.35%.

This thesis is organized as follows: In order to establish the necessary features of the structured protocol implementation approach, the current research efforts regarding the field of high-speed communication are investigated in chapter two. In addition to the need for parallelization on all levels, the domain-analysis shows that it is essential to offload the protocol processing into external accelerators. Furthermore, the investigation lead to the decision that reconfigurability of the protocol processing has to be a part of the concept.

In chapter three, the concept of the stream processing design approach is presented. This concept integrates the design, parallelization, implementation, and offloading of communication protocols without paradigm changes. Additionally, chapter three shows how reconfigurable protocol processing can be accomplished without complicating the protocol design.

The design approach is applied step-by-step in chapter four on the example of a data link protocol for wireless endpoints, in order to showcase the feasibility of the approach. The proposed prototype data link protocol is explained in detail before the design process is used to implement, adapt, and offload the protocol. Afterward, chapter four provides an evaluation that investigates the performance of the designed protocol under different communication conditions.

Lastly, the main findings and conclusions, as well as an outlook to future work are highlighted in chapter five.

1.1 Disclaimer

This thesis was written in the context of the project End2End100 (German Research Foundation Project End2End100, DFG NO 625/9-1). End2End100 is a joint project of IHP (Innovations for High Performance Microelectronics GmbH) and the Brandenburg University of Technology Cottbus–Senftenberg and is part of the DFG (German Research Foundation) priority program "100 Gbit/s Wireless And Beyond" (DFG Schwerpunkt-program SPP 1655 Drahtlose Ultrahochgeschwindigkeitskommunikation für den mobilen Internetzugriff). In the scope of the priority program, transmission technologies for 100 Gbit/s wireless communication are investigated. While the priority program focuses 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.

CHAPTER 2

Stream Processing and Communication Protocol Processing

The main goal of this thesis is to investigate how the stream processing programming paradigm can be leveraged for the protocol processing at high data rates. Stream processing applications can be represented by a graph, where edges are the streams and the vertices are the data processors. The vertices are independent of each other, i.e., the vertices only depend on the streamed data and a potential internal state. Therefore, streaming applications are inherently parallelizable. The parallelization can be done by pipelining, as well as by distributing the streamed data to parallel data-processors.

The independence of the vertexes and the possibilities of parallelization can be exploited for the processing of communication protocols: Protocols rely on communication channels that can be regarded as data streams. These streams are then processed by the protocol implementation. By dividing the protocol implementation into tasks with local-only state and treating these tasks as independent data-processors, the protocol implementation can be interpreted as a stream processing graph. Each protocol task is then only dependent on its internal state and the incoming packets. Therefore, the stream processing paradigm can be further used to parallelize the protocol processing. A coarse example[1] is the OSI-protocol stack, where each layer is responsible for a specific task, connected by the streams that transport packets between layers. The layers themselves work on the connection state and the streamed packet.

Stream processing applications are inherently distributed systems because the location where a streamed data item is processed is arbitrary, due to the independence of the vertices. This can be leveraged into offloading parts of the protocol processing, such as is common practice for the Medium Access Control (MAC) layer.

[1] The example is hampered by the fact that the OSI-Stack consists of several distinct protocols.

This high-level view is, however, idealized: In reality, layers may share states (e.g., for performance reasons), or are hard to parallelize, which may lead to bottlenecks for high data rates. Finding these bottlenecks in traditional monolithic protocol implementations can be cumbersome. The stream processing approach allows the structured analysis of the stream processing graph in order to identify these bottlenecks: Streams have a data rate and they expose the dataflow between the processing steps. The processing steps have performance characteristics, such as the time needed to process a data-item. In combination, this information can be used to analyze the application for bottlenecks and performance requirements.

The following domain-overview is used in this thesis to investigate how stream processing concepts are applied to the protocol processing, how actual stream-processing frameworks are designed, and how the stream processing approach can be used to analyze the protocol processing.

2.1 Solutions from the Networking Community

Streaming and pipelining the protocol processing is, of course, not a new idea. It has been used in order to accelerate the protocol processing since systems with multiple processors and powerful Network-Interface-Cards (NICs) appeared. The approaches and goals are diverse: Streaming packets of the same flow between kernel and user space is used in order to exploit data-locality, whereas pipelining is used in order to accelerate the protocol processing, and offloading parts of that pipeline can help to reduce the load of the communication host, to name just three. The following pages will give an insight into how stream processing concepts are adapted for the protocol processing.

The beginning of this section focuses on user space protocol processing, which allows reducing overhead by minimizing context-switches and copies between kernel and user space. In the second part, offloading approaches are investigated. Offloading is used to relieve the host from the protocol processing effort and for accelerating compute-intensive protocol tasks. This section finishes with approaches that introduce flexibility into the protocol processing, which can be used to decrease protocol's implementation complexity while increasing maintainability and preserving a high protocol processing performance.

2.1.1 User Space Protocol Processing

The communication protocol stack is traditionally located within the operating system's kernel and is controlled by a standardized API, such as Sockets in Unix [10]. This is convenient because general-purpose applications do not need detailed knowledge about the protocol's implementation, such as buffer management mechanism, but profit from an easy to use and standardized interface. Additionally, the operating system kernel isolates the protocol processing from the application, i.e., different processes can only see and access "their" data. However, changes at the kernel protocol stack are cumbersome, and in kernel protocol processing introduces processing overhead, such as system calls and buffer copying, which may exceed the actual protocol processing costs [11, 12]. In order to avoid the downsides of the in-kernel protocol processing, researchers are investigating the feasibility of user space protocol stacks.

MultiEdge

In [13], the authors argue that using standard hardware instead of special High Performance Computing (HPC) network hardware, such as InfiniBand [14], reduces costs and allows for easier maintenance. In order to compensate performance drops due to the lack of hardware support, the authors introduce MultiEdge at the communication system. The main protocol processing in MultiEdge is done within the Operation System (OS) kernel, whereas buffers are copied between kernel and user space. MultiEdge allows the transparent bonding of several physical channels, such as several 10 GbE interfaces into a single connection, as well as in-order and out-of-order packet processing. However, 100+ Gbit/s communication scenarios are not feasible without heavy optimizations and a parallel processing approach. Therefore, the authors extended MultiEdge in a parallelized fashion [15]. The authors identified the copying of payload and headers between user space and kernel as the main performance bottleneck. In the extended MultiEdge, copying buffers between kernel and user space is avoided by context-independent[2] Virtual Memory (VM) remapping. Additionally, the processing of the MultiEdge protocol can be conducted in parallel. However, parallel programming is cumbersome and traditional thread-based parallelism has its caveats: The access to the receive-queues has to be thread-safe, i.e., locking and synchronization has to be introduced. As an optimization, the authors propose the single-threaded processing of in-order packets; however, out-of-order

[2]The kernel thread that manipulates the page-table can be unrelated to the receiving process.

packets can be processed concurrently. Since all threads that are part of the processing for a single connection need access to the protocol state and other shared resources (e.g., buffers), synchronization between threads is still necessary. Their evaluation shows that parallelization of the protocol processing leads to a data rate of 38.9 Gbit/s out of maximal 40 Gbit/s one way and 57.6 Gbit/s out of maximal 80 Gbit/s bi-directional. This is achieved at the cost of an utilizing 3 Central Processing Units (CPUs) (one-way) respectively 4 CPUs (bidirectional) of overall of 8 CPUs.

NetSlices

NetSlices [16] aims to move the protocol processing into user space in order to ease the programming of packet processors, such as software switches. However, the authors found the conventional RAW-socket [17] approach insufficient due to the following reasons: Firstly, it distributes packets to all open raw sockets which introduces several unnecessary copies, and secondly, it uses system calls for each receive/send operation. The authors propose to partition the network hardware into slices, so-called NetSlices, which are tightly coupled with specific processors in order to minimize contention. A slice consists of a transmit- and a receive-queue of the NIC, a kernel thread, and one or more parallel user-level threads that are responsible for packets from the queues they are assigned to. By assigning incoming packets to their corresponding receive-queues depending on the packet-flow[3], independent flows can be processed in parallel while profiting from cache locality and reduced synchronization. By additionally assigning the kernel thread and the user-level thread to CPUs that are close to each other, the processing overhead is further reduced. In their evaluation, the authors show that their packet processing performance outperforms conventional user space packet processing and is close to in-kernel processing. However, while no system calls are necessary to send and receive packets, each packet is still copied between kernel-thread and user-level thread.

MultiStack

In [18], the authors argue that while the protocol processing within the kernel provides necessary isolation, it also makes it cumbersome to develop, test, and deploy new protocols because each change to a protocol implies modifying the operating system. In order to

[3]Packets in a packet flow that belong to the same context, such as a connection.

simplify the protocol development, the authors propose Multistack, which can be used to provide isolated parallel protocol stacks for individual applications. The isolation of protocol stacks is achieved on the basis of port-number, IP-address, and protocol type. This 3-tuple is used by an in-kernel software switch that multiplexes and demultiplexes incoming packets. In the case that no user-level protocol stack could be found, MultiStack provides a bypass to the traditional kernel protocol stack as a fall-back solution. Additionally, Multistack allows assigning multiple cores to a single port, i.e., the user space protocol stacks can be further parallelized on a per packet-flow basis. Their evaluation shows that, due to their streamlined data paths and by avoiding the (slow) socket library, the user-level protocol stack outperforms the legacy stack in all benchmarks.

InifiniBand & Message Passing Interface (MPI)

InfiniBand [14] is an open industry standard for low latency communication developed by a vendor consortium. InfiniBand is based on messages that are sent and received directly by the target application without the involvement of the operating system. This is done by connecting two applications with a queue pair that bypasses the network stack. Sending messages can be done with a send-/receive-semantics or by writing/reading directly from the remote address-space due to Remote Direct Memory Access (RDMA). The send-/receive-semantics builds on data structures in the application's address-space where the sending application can place messages that have to be sent, and the receiving application can retrieve messages that were received. These data structures are read and written by the InfiniBand sub-system, which in turn transmits and receives messages from the network. While this requires additional copies that increase the message passing latency, it has the advantage that the communication details are hidden from the application, i.e., no additional effort for the developer. In contrast, the RDMA semantics is used to write messages directly into the receivers address-space. While being faster, the destination addresses have to be negotiated before a transmission. Being able to deliver messages directly to the queue's endpoint allows parallel processing of messages without further overhead. Being a message-based communication system for HPC, it is most suitable for use with the Message Passing Interface (MPI).

MPI comes in a variety of implementations, such as OpenMPI [19] and provides point-to-point and collective communication. MPI uses two protocols for the communication: MPI Eager for message passing and Rendezvous for the transmission of larger data [20]. The

Eager protocol provides a low message passing latency by sending messages to the receiver without the need for prior negotiation by the application. Due to InfiniBand's RDMA support, the communication performance can be improved even further [20]. Since the transmission of bigger data is negotiated first with a handshake procedure that is used to prepare the receiving buffer, the Rendezvous protocol can profit straight forward from RDMA by negotiating the destination address during the handshake phase. The Eager protocol can only profit from RDMA when the addresses of the message receive-buffers are known to the receiver before the transmission takes place. This can be realized by organizing the message buffers as ring-buffers at the sender and receiver. When both ring-buffers are synchronized, the receiver always knows at what ring-buffer slot the next message should arrive, and the sender knows which slot has to be used for the next message. The flow-control is based on these ring-buffers, i.e., when the sender-buffer is full, the sender can not send messages with the RDMA approach[4]. The receiver notifies the sender about the free space by piggybacking the last possible writing slot of the ring-buffer to outgoing messages.

Summary

The presented approaches provide efficient and low-overhead network I/O that can be used to implement user-level protocol stacks, which increases flexibility and eases the protocol design process. These approaches are suitable for partitioning the protocol processing based on individual connections or protocol stacks onto distinct processors. However, none of the approaches use the full potential of the stream-processing paradigm, e.g., parallelism by pipelining the protocol processing steps. Instead, the presented approaches rely on multi-threading and fall back to explicit synchronization between the processing threads for state manipulation and access to shared resources.

MPI allows parallel processing of messages without constraints. However, it also doesn't provide a traditional communication protocol but relies on an underlying communication system for reliable communication. The combination of InfiniBand and MPI provides an interesting RDMA flow-control mechanism for the Eager protocol. A similar approach will be used in this thesis for transmitting bulk-data between the host and protocol processing system.

[4]Then the traditional (non-RDMA) InfiniBand's send/receive mode is used.

2.1.2 Hardware Acceleration by Offloading

In order to alleviate the protocol processing load of the host, and therefore freeing its resources, protocol processing offloading was proposed. Offloading has the premise that the host is either not capable of the protocol processing or that the host's resources should be used for the host's actual task and not for the protocol processing. This subsection is divided into offloading of full protocol stacks, offloading of parts of the protocol processing, and applications that use offloading for faster packet processing.

TCP Offload Engines

In [21], the authors present an FPGA implementation of a fully offloaded TCP/IP stack, which supports up to 10000 simultaneous connections and that can utilize a 10 Gigabit Ethernet (GbE) interface. The authors highlight that this is only possible by following the TCP/IP dataflow, i.e., separating transmit/receive paths and allowing lightweight access to the connection states. In [22], the same authors adapt their initial implementation to support low-latency scenarios. This is achieved with application knowledge and fine-grained optimizations. Furthermore, the authors assume that reordering incoming packets is an application task. While that assumptions eases the implementation, it also assigns work back to the host. While the authors achieve 10 Gbit/s, the connections are simulated on the NIC itself. Since some protocol tasks, such as data transfers between host and NIC, are omitted, no conclusion about the actual performance can be drawn.

General Hardware Acceleration

The offloading of the complete TCP protocol stack is seen as problematic [23] for reasons such as the cumbersome process of updating a hardware implementation or the expensive and error-prone design process. A beneficial compromise is the partial offloading of suited protocol task into external hardware. TCP Segmentation Offload (TSO) (e.g., [24]) is a sender-side optimization that offloads the segmentation of bulk data into an external accelerator, such as a NIC. Instead of segmenting the data at the host, the payload and the meta data information needed for building the headers is copied to the NIC, which creates the segments, fills the headers, and transmits them to the network. Hence, stress on the memory system is reduced because the individual packets do not have to be

touched at the host. Large Receive Offload (LRO) [25] is a hardware supported receiver side optimization that aggregates consecutively incoming packets that belong to the same flow into a larger packet. This way the processing overhead for small packets at the host is reduced. Interrupt coalescing [26] is an optimization technique that reduces the number of expensive interrupts between NIC and host by only sending an interrupt for the first packet that is copied into an empty NIC receive buffer. Interrupt coalescing is also used to avoid livelocks in systems that receive packets faster than the host can maintain [27]. In [28], the authors investigate several protocol processing optimizations related to TCP offloading. The authors show that measures such as interrupt suppression in case more data is available, checksum offloading, and zero-copy buffer management allows for higher throughput and less overhead.

PacketShader

PacketShader [29] is a software router that uses offloading of the packet processing to an external GPU. PacketShader employs a number of optimizations: Firstly, the optimized user-level packet I/O engine uses packet-batching for receiving and sending in order to reduce the offloading overhead. The transmission overhead for the batches is further decreased by using preallocated and slotted huge buffers instead of individual buffer allocation/deallocation. Moreover, the memory locations of the packet batches are chosen depending on the Non-Uniform Memory Access (NUMA) memory-system. Lastly, interrupt-coalescing (suppressing consecutive interrupts) is used to avoid receive-path livelocks.

The packet I/O engine employs the parallel processing power of Graphic Processing Units (GPUs) for the actual packet processing. PacketShader's workflow is as follows: (1) the packet I/O engine fetches batches of packets from the NIC and prepares them for the processing step by dropping malformed packets and extracting meta data, such as IP addresses, necessary for the packet processing. (2) After pipelining the batch to the thread that is responsible for the communication with the GPU, this thread copies the data to the GPU and triggers the packet processing. (3) After the processed batch is given back to the worker thread, the batch is post-processed depending on the processing results and split into packets for transmission. The evaluation shows that especially for small packets, the GPU accelerated packet processing outperforms CPU-only processing.

GASPP

GASPP [30] is a packet processing framework relying solemnly on GPUs for the packet processing. It allows programming the GPU based on module prototypes for common protocols, such as Ethernet and TCP, as well as stateful protocol processing. The programmer can then use an Application Programming Interface (API), which provides support for protocol tasks, such as pattern matching, encryption, and packet manipulation, for the implementation of the modules. The modules can be chained, i.e., sequent processing steps can be grouped together. Finally, the modules are compiled into kernels that allow the execution on a GPU. In GASPP, the GPU's parallel processing capacities are used to process one packet per thread/core. However, GPUs are organized in clusters of threads, whereas each thread in a cluster computes the same kernel.

GASPP uses a set of optimizations in order to reduce overhead and increase utilization of the GPU. In order to utilize the GPU, GASPP groups packets by length and type and schedules these groups so that they are processed within the same cluster. In order to avoid unaligned memory accesses as they happen when accessing fields in a network header, the header is decoded into meta data that is aligned before processing. Finally, in order to minimize the per-packet overhead due to copying between the NIC and the GPU, GASPP collects incoming packets into batches that are transferred between host and GPU. Additionally, GASPP allocates the same DMA memory area to both devices, i.e., the NIC copies to the memory and the GPU reads from the same memory area. The evaluation shows that GASPP outperforms any of the single-CPU implementations, however, the protocol processing performance depends on the number of batched packets whereas smaller batch-sizes decrease the performance.

Click Modular Router

The Click Modular Router (Click) [31] is a software router developed in the year 2000 that allows connecting small packet processing tasks (elements) into an application graph that implements the desired functionality, such as packet routing. The Click Modular Router used the stream processing paradigm for introducing flexibility in the (back then) purely hardware dominated network infrastructure. However, when introduced, Click fell far behind in points of performance compared with the traditional hardware approaches used at this time. The main reason was the lack of suitable processing hardware, such as

manycores or programmable GPU. Therefore, Click processed the routing application in a single thread within the kernel. However, the increase in flexibility and maintainability was seen beneficial, and researchers started adopting the concepts and improving the performance.

ClickNP [32] is a Field Programmable Gate Array (FPGA) version of the Click Modular Router. As in Click, the basic processing elements can be connected with each other. An element has a local-only state, an arbitrary number of inputs and outputs, and three functions: An initializer, a function that receives data from the inputs and processes them, and a function that receives signals from the host. Elements are connected by FIFO buffers in which the sender writes and from which the receiver reads. Additionally, elements can communicate with each other through messages. Since the elements are independent from each other, subsequent elements build a pipeline that processes data in parallel. Additionally, it is possible to split pipelines in case a single processing step has to be parallelized further. To further increase the throughput, small loops are unrolled, so that they become a pipeline themselves, and elements that combine fast and slow tasks are split up. Additionally, ClickNP offers the possibility to compile elements into the host binary. In this case, the managing thread that is also responsible for the configuration of the FPGA, creates a worker thread for the host-side element. This way, ClickNP provides parallel execution on the host and the FPGA. However, while ClickNP is flexible regarding the configuration of the host and the external FPGA, it provides no on-demand adaptation.

The Network Balancing Act (NBA) [33] is an extension of the Click Modular Router that uses batching of packets in combination with offloading work to a GPU. Additionally to the parallel processing at the external GPU, NBA provides parallelism at the host. For this, the processing pipelines are replicated and processed independently on parallel cores, which reduces synchronization overhead. The worker-thread that is responsible for a pipeline, fetches a batch of packets from the network cards and processes the batch locally according to the processing pipeline. The authors present a load balancing scheme for GPU offloading, which implicitly takes the offloading overhead into account. In order to do so, NBA provides offloadable elements that provide a host and an accelerator implementation. A load-balancer element decides whether the offloadable element should be processed locally or at an external accelerator. The communication with the accelerators is conducted with device threads that pass packet batches, sent from a worker-thread's load balancer to the GPUs. The authors show that offloading every packet is not necessarily the most efficient approach. Instead, the authors use a manually estimated threshold

which determines the percentages of work that has to be offloaded. Consequently, NBA is flexible regarding the invocation of the external accelerator, however, it does not provide on-demand adaptations of the functionality.

Network Processors

The IBM Wire-Speed Processor [34] is a 16 core processor with an integrated network interface designed for parallel protocol processing in software. Additionally, the processor is equipped with accelerators for tasks that are unsuited for parallel processing in software. Such tasks are: Non-parallelizable tasks that would lead to a single-threaded bottleneck, tasks that have a high memory-bandwidth demand because they have to be executed at the whole packet (e.g., CRC computation), and last but not least, standardized tasks with a high computational footprint. These tasks are offloaded into on-chip accelerators. One of these accelerators, the RXACC [35] (Receive Stack Accelerator), is integrated into the on-chip Host Ethernet Adapter (HEA). The RXACC reads incoming packets in 16 Byte chunks that can be interpreted individually by a programmable packet parser. The extracted information is used by the parser to determine the next parsing step and a processing rule for the packet. The processing rule defines the functional unit that implements a certain protocol processing task, such as filtering, that is executed on the packet. Additionally, the accelerator creates a meta data descriptor that summarizes the results of the hardware packet processing. This information, e.g., the offset of a certain protocol header or the connection to which a packet belongs, is used by the software stack to process the packet efficiently. Additionally, the Host Ethernet Adapter provides several queues that can be assigned to software-threads, allowing per queue parallelism depending on the results of the parser.

Summary

While offloading complete protocol stacks in special-purpose hardware is cumbersome and lacks flexibility, offloading parts of the protocol processing improves the overall protocol processing performance, as well as it significantly reduces the computational effort of the host. Generally, offloading should be conducted in a streamlined manner to avoid unnecessary copying, e.g., in the case of FEC computation, the computation should be the last processing step before directly sending the final frame without additional copies.

Therefore, offloading frameworks that use general-purpose accelerators such as GPUs are not optimal. While providing significant parallel processing power, GPUs are unsuitable for endpoint protocol processing because they lack programmability and packet streams have to be pre-processed for the GPU processing and post-processed for the transmission which introduces (avoidable) overhead.

2.1.3 Flexible Protocol Processing

Introducing flexibility into a protocol can be beneficial, as it allows the protocol implementation to react to certain communication conditions. However, flexible protocols also have disadvantages, e.g., flexibility increases the implementation complexity, which increases the possibility of errors. In order to achieve flexibility concerning the communication conditions and keep the negative effects in check, the protocol implementation itself can be changed in the best case at runtime. That would allow simple low-complexity protocols that are still tailored for a specific use case.

Multipath TCP

Multipath TCP (MPTCP) [36] is an extension of the Transmission Transport Protocol (TCP) [37] that allows combining several physical interfaces into a single connection, consequently providing multiple parallel transmission paths. This is realized in several steps: Firstly, an initial multipath-capable connection is created. Once the receiver answers that it supports the multipath option, new TCP sub-streams can be added. While the general idea and implementation are straight forward, the main problem are middleboxes, such as Network Address Translations (NATs) and firewalls, since they may alter TCP packets. For example, unknown TCP options may be stripped from the packet, or sequence numbers may be rewritten. These problems are solved by adding an explicit (global) data acknowledgment field to the TCP header that is independent of the substream sequence number. When an MPTCP connection is created successfully, additional substreams can be created in order to increase overall throughput. However, despite the parallel connections, no parallel protocol processing is conducted.

Pluginizing QUIC

QUIC [38] is a transport protocol introduced by Google that encrypts almost the complete frame. In addition to the increased security, encrypting the frame prevents network-operators to change headers at will, which was a main problem when introducing MPTCP. Consequently, QUIC can be improved and changed easily. This is leveraged by PQUIC [39], where the authors argue that QUIC should be adaptable on a per connection basis. In PQUIC the authors use extended Berkeley Packet Filter (eBPF) [40] functions to implement plugins that can be attached to the PQUIC implementation in order to change the QUIC's behavior. The plugins are attached to the protocol and executed in a lightweight virtual machine, allowing for a high portability. Furthermore, these plugins must not be present at the communication hosts because they can be dynamically loaded from trusted plugin-providers.

At the present state, the flexibility is limited to the QUIC protocol but it is foreseeable that other protocols can profit from the idea. While the benefits of that take on flexibility are obvious (e.g., fast and easy testing and deployment of protocol changes), the approach also comes with considerable overhead rendering it unfeasible for high data rates. Additionally, the configuration is limited to the start of a connection, because of (no further stated) synchronization issues during the code-injection. Lastly, no concept of parallel protocol processing or offloading of protocol tasks were envisioned in PQUIC.

DRoPS

The DRoPS framework [41] provides flexible protocols by combining predefined microprotocols. DRoPS is based on the idea that communication protocols can be decomposed into disjunct protocol tasks, such as fragmentation or acknowledgement schemes. These individual tasks can be encapsulated into microprotocols which can be combined into a communication protocol. All microprotocols implement the same interface that provides initialization, transmit, and receive functions. Decomposing protocols into individual microprotocols as such, allows to reconfigure protocols at runtime. DRoPS allows the reconfiguration at runtime by piggy-backing the changes into the outgoing frames or by creating a new connection. In the case the changes are encoded within the frame, the receiving endpoint changes the local protocol accordingly before further processing the payload. However, this means that incoming frames are stalled until the adaptation

is finished. Depending on the data rate, this may cause dropped frames and a high overhead.

VirtualStack

VirtualStack [42] is an architecture that uses virtual network interfaces to provide applications with flexible network stacks to decouple an application from the network stack. In this way, applications can benefit from different network paths without knowing the underlying network. In addition, VirtualStack provides rule-based programming to define thresholds, such as minimum throughput. When the threshold is reached, an additional channel with a new network stack is transparently created.

While providing flexibility with respect to the used protocols and communication interfaces, VirtualStack is executed in user space, therefore, using resources needed by the host. Furthermore, VirtualStack allows combining several network devices, however, the achievable goodput is limited by the processing power of a single processor, because no parallelization is conducted.

Cactus

Cactus [43] is a middleware that allows the composition and configuration of protocols based on microprotocols. In contrast to the other presented approaches, Cactus's focus is resilience against security threads. The authors argue that attacks on the communication system can be deflected by adapting the communication protocol in case of an attack. The Cactus framework is event driven and the individual microprotocols do not have knowledge of each other, but have access to a global state. That global state can be changed by all microprotocols. In order to avoid synchronization, the processing is not parallelized and the event handlers are not interruptible. Since the system can be never in an inconsistent state, the reconfiguration of the protocol can be done by reassigning an event handler for a certain protocol task, such as a new encryption algorithm. However, no parallelization disqualifies Cactus for high data rates.

In order to keep sender and receiver consistent, Cactus uses a three step approach: detection, agreement, and action. After detecting a thread, the sender and receiver have to agree on the correct adaptation, before the changes are implemented. This approach

takes time and in the case of Cactus, it either stalls the transmission or continues using a protocol for which a threat was already detected.

Flexibility in Networks

Network flexibility is used to handle networks and reduce maintenance costs by Software Defined Networks (SDNs) [44] and Network Function Virtualizations (NFVs) [45]. SDNs add flexibility to networks by separating data plane, i.e., physical hardware that follows network rules, and the control plane that is used to define these rules. The concept of SDNs emerged in the spotlight with the OpenFlow API [46], which was meant to ease protocol research within already available production networks without disrupting the regular network traffic. An OpenFlow switch consists of a flow-table that combines each flow with an action, a secure communication channel for the network controller that deploys and defines the rules, and an implementation of the OpenFlow protocol. The actions are forwarding or dropping packets of a known packet flow, sending unknown packets to the controller for identification or using the original processing pipeline of the hardware. Since OpenFlow reuses already existing hardware, it was quickly adopted also outside of research [44].

However, traditional network hardware, e.g., switches and routers, are usually limited to a certain amount of known fixed protocols and their headers that are understood by the hardware's chips. Programming Protocol-independent Packet Processors (P4) [47] is an extension for OpenFlow, that allows describing protocol headers and forwarding rules in order to increase the variety of supported protocols. P4 relies on five components for the programming [48]: header, parsers, tables, actions, and a control program. The headers describe the fields of a packet that can be checked by a parser. The parser uses the header definitions and identifies headers and stores them in the meta data. The match- and action-tables, in combination with the meta data, are used to determine the necessary actions for that packet. If a match is found, the corresponding actions, such as manipulating headers, are executed. Finally, a control program allows defining the order in which packets are matched to tables, i.e., a pipeline of tables. Additionally, conditions can be used to control the packet processing, e.g., the IP header has to be matched and found as valid before the TCP header is interpreted. While dependencies between tables serialize the processing of a single packet, the pipeline of match and action tables allows the processing of several packets in parallel.

This concept is used by [49] for an FPGA accelerated P4 parser. Furthermore, Network Processing Units (NPUs) aim for OpenFlow and P4 compliance with the goal of providing hardware-switch line rates with the flexibility of programmable switches [50]. These processors provide an architecture that supports the P4 programming model. For example, the NFP-4000 [51] provides parallel packet processing cores specially designed for the parsing and classification of incoming packets, clusters of parallel flow processing cores used for the match and actions tables, and an ARM processor used for configuration.

Network Function Virtualization (NFV) is another approach to increase the flexibility of network processing. NFV aims to provide network functions such as HTTP-caches or VPN-tunnels, as virtual machines instead of individual middleboxes [52]. These virtual machines can be grouped together on one powerful server to provide a certain service. Since no actual machine has to be deployed, the approach brings flexibility. The VMs are connected with a software switch, such as Open vSwitch [53].

Elastic Scaling (E2) [54] is a management framework for NFVs that uses SDN concepts and employs them for the description and connection of network functions. The control plane relies on the description of network functions pipelines called pipelets. Pipelets describe a directed acyclic graph whose edges describe the traffic and the nodes describe the network functions. The network functions can be further characterized by replication constraints, as well as configuration and resource requirements. This description is used for automatic placement, dynamic scaling, and configuration of the network functions. Network functions can further support each other with "rich messages" such as an already reconstructed TCP stream. That byte-stream can be used by several network functions without the additional reconstruction effort.

Summary

While flexibility in networking gained widespread adoption, flexible protocols did not receive much research attention. The reasons are low performance and a lack of necessity in general-purpose endpoints. However, in high-speed communication scenarios, simple and specialized protocols are needed to achieve the desired data rates with reasonable design and optimization effort.

2.2 Stream Processing Architectures and Approaches

Data parallelism and pipelining are used in many packet processing scenarios. However, most of the approaches are tailored for a certain use case without concern for generality. The goal of this thesis, however, is the investigation of the feasibility of a stream processing approach that integrates the complete implementation process without paradigm changes. Such stream processing approaches exist for the analysis of unbounded streams of data items. Stream processing systems can be coarsely divided into *Data Stream Processing* and *(Complex) Event Stream Processing*. Data stream processing is used to handle a stream of data items which are processed with operations such as aggregation or joining. The common architectures are Data Stream Management Systems (DSMSs), which are similar to databases and often even provide a language similar to Structured Query Language (SQL). (Complex) Event Stream Processing handles event items, such as *11:23 : Temperature in Room 2.11 > 34 °C* and is used to correlate these events with each other, in order to extract complex events.

Stream processing systems answer some questions that will probably arise when applying the stream processing paradigm to communication protocols, such as: How can a stream be efficiently parallelized? What is a convenient programming abstraction? How is the data transported and how is the processing scheduled? In the following pages, stream processing engines developed for different use cases will be presented and investigated.

Twitter Heron

Stream processing systems emerged in mid-2000s as an answer to the question of how the ever-increasing amounts of gathered data can be handled. Big data, as it is called, uses stream processing for the analysis of high-volume data streams [55]. These frameworks use clusters of computers that are "loosely" coupled and managed with a global coordinator, such as ZooKeeper [56]. The actual analyzing tasks are distributed over the cluster, where each individual cluster-node processes a partition of the data stream. The main problems are scalability, fault tolerance, and manageability. A popular large scale stream processing system is Twitter Heron [57].

Twitter Heron is a stream processing architecture that provides coarse-grained parallelism on the basis of processes. Heron is used for analytics at Twitter where it replaced Apache Storm [58], which didn't provide the needed scalability. As Heron replaced Storm, it

had to provide the same API and abstractions in order to minimize migration costs. At the core, it differentiates between data-sources, called *spouts* and data-processors, called *bolts*. Bolts are implemented by the user in Java, and a spout can be connected to any data source, such as databases or log-files. Bolts and spouts are connected by a user-defined topology. Topologies are Direct Acyclic Graphs (DAGs) that describe the logical execution plan of the stream application. A topology is created by the user and is managed by a topology master.

Each topology is annotated with information about parallelism of individual spouts and bolts and a plan that describes how the data is distributed. The distribution pattern is called grouping, as it groups items from the stream to be forwarded to a certain instance. The grouping of items can be done in several ways: Randomly, by hashing values, and by broadcasts. Random distribution follows no particular scheme and just assigns data-items to a sub-stream. Hashing is used in the case a data-item has to be processed by a certain sub-stream, i.e., the data-item is characterized, and the corresponding sub-stream is selected accordingly. Lastly, broadcast distribution duplicates all items and forwards them to all sub-streams.

The topology is instantiated by the scheduler that creates containers which encapsulate the functionality. A container provides a Stream Manager for the communication, a Metrics Manager for performance monitoring, as well as the spouts and bolts. Containers are distributed on the computer cluster depending on the parallelism and data distribution information given by the topology. Since several containers can be assigned to the same processing node, cluster nodes can be multiplexed with other containers and even with topologies. The communication between any individual bolts is handled by the stream manager, whereas every stream manager is connected with every other stream manager. This way, the stream manager is responsible for the routing within a container and between containers. While this seems to be inefficient and poses a possible bottleneck, this is done in order to ease debugging and to be able to track all performance metrics.

Heron implements a spout-based back pressure flow control mechanism in order to avoid loosing items in the case that the item data rate is too high for a bolt in a container. In this case, the stream manager of this container stops reading from the local spouts and sends a start-back-pressure command to all other stream managers. Eventually, none of the spouts can send data so that the over-utilized bolts can catch up. When the over-utilization is resolved, a stop-back-pressure command is used in order to signal that

the data-producers can produce tuples again. This information is passed to the topology master.

After scheduling the topology, users can inspect how topologies work by analyzing the performance metrics and fine-tune the deployment in order to optimize resource utilization. Furthermore, Twitter Heron can be extended with an automatic scaling mechanism [59] that reacts to over- and under-utilization by increasing or reducing the number of Heron instances.

Stream

Stream [60] is a Database Management System (DBMS) developed at Stanford University. The authors target high data rate streams with changing characteristics and loads in a resource constraint environment. Being a DBMS, Stream provides CQL (Continuous Query Language), a SQL like query-language that deals with the unboundedness of the input data by differentiating the query output. A query can output a relation that is bounded by a sliding window, i.e., a snapshot of a stream over a given time or number of tuples, or a stream, i.e., a continuous sequence of tuples.

Stream's queries are not parallelized. Instead, operators are executed in one thread and scheduled in order to minimize memory consumption. The operators that belong to a query are only dependent on the logical timestamps of the tuples. Thus, the scheduling does not affect the correctness of the query. Operators are scheduled with a chain-scheduler that the authors present in detail in [61]. The scheduling creates static chains of operators based on the expected progress of the query (i.e., how many tuples are removed from the system), whereas the operator that leads to the highest progress is scheduled first. However, a single CPU may limit the processing of a stream when the data arrival-rate is too high, i.e., when not enough CPU cycles are available to process each item of the stream. The authors propose to gracefully degrade the accuracy of the query by load-shedding. The items are then dropped by a special sampling operator. Furthermore, individual queries can be assigned to distinct threads.

Noria

Dataflow processing is also used as a backend for applications. Noria [62] is a stream processing based query backend for web applications and provides an SQL like query-language. In order to reduce the query latency, the authors propose to replace explicitly pre-computed data-base queries or cached query results with stateful in-memory database queries. Instead of selectively pre-computing often-used results, the updates at the database are stored in the operator state, leading to in-memory database-views. However, this potentially leads to high memory usage and changes at a query could lead to inconsistent states. The high memory consumption is reduced by partially evicting state. This is done by notifying the whole query about dropping a certain element when it is evicted from the operator's state. In the case a subsequent read has to access the item, the evicted data item is acquired from an operator that has the latest version of the tuple.

Noria employs parallel processing on several levels: Firstly, a dataflow can be separated by hashing the values and process dataflows with different hashes (that do not share state information) on parallel cluster nodes. Secondly, on a single server, parallelism can be provided by multiple Noria instances, as well as parallelism within a Noria instance.

StreamBox

StreamBox [63] is stream processing engine designed for low latency processing of out-of-order streams. The authors propose to organize data items in containers, depending on the ingress timestamp that is assigned to each data item. The containers basically subdivide items into buffers that belong to the same processing window, called epoch, which allows StreamBox to process data items out of order without waiting, while keeping the item order intact.

StreamBox parallelizes the processing on two levels. Firstly, consecutive processing steps form a pipeline that processes containers in parallel. Secondly, a single pipeline step can process all its currently assigned containers in parallel. A new container is created whenever the first data item of the corresponding epoch is handled by a pipeline step. A main problem is figuring out whether all items of an epoch were already received or whether more out of order data items are to be expected. The authors solve this by including watermarks that are injected at the data source. Assumed, that no tuples are

lost during the transmission, it is clear that all other data items belonging to this epoch were received when the end-watermark is seen. The container for that processing step can then be discarded. The parallel execution of pipeline steps is handled by parallel threads that are dynamically allocated from a thread-pool. These threads receive aggregated data-items on which they process the according pipeline step. The aggregation is configurable and is used to reduce the item dispatch overhead.

The scheduling scheme is based on the "next externalization moment", i.e., the moment the oldest windows (i.e., the oldest consecutive group of containers) is completely processed. Therefore, it prioritizes older data items in order to finalize the processing of pending windows.

StreamIt

StreamIt [64] is a programming language for streaming applications. The C++ like language organizes the streaming applications into processing steps called filters. Each filter has an input queue, an output queue, and a work function. Upon the arrival of a data item, the work function processes the item and pushes the results to the next filter via the output queue. Each output can be configured as a splitter that produces several streams and can be used to split or duplicate the stream. Inputs can be configured as joins that combine incoming streams. Additionally, StreamIt allows sending timed control messages from within a work-function to other filters. The timing specifies when the control message will arrive at the receiving filter relative to the streamed items. This is used, for example, to change a state of a filter with a controlled delay.

While the splitter outputs can be used for manually parallelizing the processing, the processing can also be parallelized automatically [65] based on the exposed parallelism. Instead of exploiting the inherent parallelism of each filter individually, filters that form a pipeline are automatically chained and executed sequentially. However, that may lead to purely data-parallel execution that eliminates all pipelining. In order to prevent such degenerated stream processing, the pipelines are separated into (fused) sub-chains. The ratio between data parallelism and pipelining is determined by a heuristic that tries to utilize all processors.

Summary

The presented systems are tailored for their specific use cases. Distributed stream processing systems, such as Twitter Heron, do not fit the protocol processing scenario very well. The main problems that were solved are maintainability and fault tolerance.

The SQL-like stream processing abstraction languages in Stream and Noria allow the designer to describe the problem from a very high level, however, they are unsuited for implementing communication protocols. Out-of-order stream processing organized in containers as in StreamBox can be used for hiding processing latencies and will be necessary for reaching high data rates, especially when packets are lost during transmission. However, since the communication characteristics depend on the communication scenario, such latency hiding techniques should not be an implicit part of the processing, but applied by the developer.

All systems but Stream employ parallelism in order to increase the throughput. The parallelism is employed on several levels. Firstly, coarse-grained processes are assigned to parallel cluster nodes [57, 62]. At the process-level, an application-specific scheduler could select the next data-items to be processed [60, 62, 63, 64]. Most of the investigated stream processing systems schedule the processing in order to increase the CPU utilization, as this reduces costs. While the CPU utilization is also important for communication protocol processing, the main driver is the expected throughput. Additionally, a sophisticated scheduler (as in StreamBox) can help to avoid pipeline stalling by searching scheduling lists for the oldest items. While this is important for unpredictable streams, such as data from user-interactions, it is less crucial for communication protocols in which the characteristics are commonly known or can be estimated.

In comparison, a scheduling based on a logical execution graph, as used by Twitter Heron, seems more practical. That graph can be annotated with static scheduling decisions, e.g., the grouping of messages into sub-streams, as used by Twitter Heron or StreamIt. The actual schedule can then be derived at design time in case the processing costs are already known. The resulting schedule can be carried out by a lightweight First In First Out (FIFO) scheduler.

2.3 Real-Time Analysis

Communication protocols are soft real-time problems because missing a deadline, e.g., due to lost packets, leads to performance degradation, but seldom ends in a system failure. Nevertheless, a high communication-performance is crucial for users and applications. Since protocols can become arbitrarily complicated and communication conditions vary, the estimation of the resources, needed in order to achieve a certain data rate, can be cumbersome. In the last part of the domain-analysis, three concepts for dataflow based performance analysis are briefly investigated. These concepts are used in the field of system design in order to evaluate the system's performance or to find resource boundaries such as minimum buffer sizes, as well as service guarantees, such as the worst-case throughput.

The Artemis Workbench [66] is a toolchain that allows a high-level modeling and performance analysis of platform-based embedded systems [67]. Platform-based embedded systems are built from predefined building blocks that are combined in order to form a special purpose System on Chip (SoC). Each building block fulfills a certain task of the systems, and their performance parameters are known (e.g., the ram module has 10 KB of RAM). This specification of the system can be used for analysis. The software that is executed on the specified SoC, e.g., a MATLAB script, is transformed into a process-network [68] and then mapped onto the specified hardware architecture. The specified system can then be evaluated by coarse-grained simulations. In the case the behavior is not as expected, the platform can be refined. At the end of the process, the specified hardware-software design can be handed to a manufacturer.

Synchronous Dataflow Graphs (SDFGs) [69] are used in signal processing in order to analyze application behavior. The graphs contain nodes and edges, whereas each node can have several inputs, and the edges describe the dependencies between the nodes. Each node produces a fixed number of output items whenever a fixed number of items are available on all inputs, i.e., the behavior is completely deterministic. This makes a SDFG decidable and allows deriving static schedules and calculate necessary buffer-sizes. While being suitable for a signal-processing application, the SDFG model is limited in its expressiveness: It does not allow any dependency of a node on incoming data. However, in a real system, data dependencies may define the output behavior, e.g., the number of items produced by a node at an output depends on the item at its input.

The authors of [70] argue in favor of analysis instead of simulation for the design of model-based embedded systems in order to achieve a certain service quality. They propose to use SDFG for the analysis. In order to specify the applications dynamic behavior that can lead to different requirements at different times, the authors propose Scenario Aware Synchronous Dataflows (SADFs) for the analysis instead of actually modeling the dynamic behavior. The idea is as follows: Different stages of the processing are modeled as synchronous dataflow graphs that can be analyzed according to throughput, latency, or buffer requirements. In order to reflect the dynamic behavior, a finite-state-machine that "switches" between the scenarios, is used.

The Worst-Case-Execution-Time (WCET) analysis has the goal of estimating the maximum time it takes to finish a certain task. The WCET is needed to define a schedule in hard real-time systems. In [71] a comprehensive overview of WCET analysis methods and tools is given. The WCET analysis comprises roughly two problems: Firstly, the input data that leads to the WCET has to be found, and secondly, the execution time has to be estimated. Finding the correct input data is not trivial in the general case due to the variety of possible inputs. However, common inputs are maybe known and can be used for an approximation. Furthermore, the task can also be separated into smaller tasks to decrease the complexity. Other approaches analyse the task's possible execution paths and estimate the WCET with the help of a hardware model [72]. Moreover, estimating the WCET for a task can be done by statically analysing the program code [73], or by measuring the execution time on the target hardware [74]. The analysis can be further assisted by tools such as aiT [75].

Summary

This last part of the domain overview offered a brief insight into real-time analysis techniques. SDFGs are restricted to static behavior, but allow to derive a static scheduling with accurate resource estimations. SADFs combine several SDFG in order to model different behaviour but may increase the number and complexity of the graphs significantly. Analysis tools, such as the Artemis Workbench, simulate the software and hardware in order to evaluate the system. However, software protocol processing on general purpose processors is complex and poses a lot dynamic behavior, which would increase the graph and analysis complexity to an impracticable level. Of course, the communication community has its own set of analyzing theory [76]. However, it lacks accessibility for protocol designers without a strong mathematical background.

This thesis aims for an accessible approach that allows a fast and developer-friendly estimation of the processing performance and requirements of the protocol implementation. Therefore, a simple and straightforward approach based on scenarios and simulation is needed. That approach should help the protocol developer to estimate hardware requirements solemnly based on the protocol implementation. The protocol's inherent soft real-time requirements can then be used in combination with a performance indicator similar to the WCET to derive a schedule for the protocol processing.

2.4 Summary & Conclusion

In the first part of this chapter, the research efforts undertaken in order to utilize the theoretical data rates of current and future networks were briefly investigated. A tabular summary is presented in table 2.1. The summary is categorized into parallelization, flexibility, and offloading. Parallelization is further divided into the *parallel processing paradigm* and *Channel Bonding*[5], i.e., parallelizing a transmission over several communication channels. The parallel processing paradigm is categorized into stream-processing (SP), pipelining (PL), multi-threading (MT), and single threaded (-). The flexibility category is further categorized by whether processing resources can be allocated depending on the load (*Processing Resources*), whether it is possible to change the number of channels (*Channel Bonding*), and whether the behaviour of the executed protocol is adaptable (*Protocol Behaviour*). Flexibility is rated as on-demand (O), at initialization (I), and at compile time (C). The last category, offloading, is rated as stream-lined (S), when additional copies are avoided, and disruptive (D) otherwise.

Two main directions with the goal of increasing the processing capacity are seen throughout the presented solutions: Parallel packet processing and processing offloading. Both are seen individually, as well as in combination. Parallel protocol processing is conducted by pipelining consecutive tasks, as well as with explicit thread-based parallelism. Thread parallelism is leveraged by streaming packets along a consistent path, e.g., mapping a packet flow to a certain processor, in order to reduce copying and synchronization overhead [16, 18, 21]. Parallelism is exploited on the bases of packet flows [16, 18] for explicit parallel processing of packets with GPUs [29, 30, 33], with NPUs [34, 51], and

[5]Channel bonding is only considered for endpoint protocol processing, not for packet processing application.

	Parallelization		Flexibility				
	Parallel Paradigm	Channel Bonding	Processing Resources	Channel Bonding	Protocol Behavior	Offloading	Domain
Desired combination of features	SP	✓	O	O	O	S	
MultiPath TCP[36]	-	✓	-	O	-	-	Protocol
PQUIC[39]	-	✓	-	I	I	-	Protocol
MultiEdge[15]	MT	-	-	-	-	-	Protocol
InfiniBand[14] & MPI[19]	-	-	-	-	-	S	Protocol
Virtual Stack[42]	-	✓	-	O	O	-	Protocol
DRoPS[41]	-	-	-	-	O	-	Protocol
Cactus[43]	-	-	-	-	O	-	Protocol
TCP Offload Engines [21, 22]	PL	-	-	-	-	S	Protocol
TSO[24] & LRO[25]	PL	-	-	-	-	S	Protocol
NetSlices[16]	MT[1]	/	-	/	-	-	Framework
MultiStack[18]	MT[1]	/	-	/	-	-	Framework
IBM Wirespeed[34, 35]	MT[1]	/	-	/	O[3]	S	Processor
NFP-4000[51]	MT[1]	/	-	/	O[3]	S	Processor
PacketShader[29]	PL	/	-	/	-	D	Packet Processing
GASPP[30]	MT	/	-	/	-	D	Packet Processing
Click[31]	ST[2]	/	-	/	-	-	Packet Processing
ClickNP [32]	SP	/	C	/	C	S	Packet Processing
NBA [33]	SP	/	O	/	C	D	Packet Processing
P4 [47]	PL	/	-	/	C	S	Packet Processing
P4 Parser [49]	PL	/	-	/	C	S	Packet Processing
Elastic Scaling [54]	DS	/	O	/	-	-	Packet Processing

Table 2.1: Communication solutions from the networking community. The reviewed approaches encompass communication protocols, processing frameworks, special network processors, as well as packet processing solutions. The shown categorization was conducted with respect to *Parallelization Paradigm* (SP–Stream Processing, MT–Multi-Threading, PL–Pipelining), whether *Channel Bonding* is supported, *Flexibility* (O–On-Demand, I–On-Initialization, C–Compiletime,/–Not Applicable), and *Offloading* (*S - Streamlined*, D - Disruptive). None of the reviewed concepts fulfilled all of the desired features.

[1]Depends on the protocol implementation, however, no concept for parallel processing but parallel threads.

[2]Click uses the stream-processing paradigm but not parallel execution.

[3]Only the packet parser can be configured.

for combining of several parallel interfaces for redundancy or increasing the theoretical data rate [15, 36]. Pipelining is used to overlap copying of bulk data with processing [29], for consecutive processing steps [47, 51], as well as for handling in- and egress of packets in parallel [21].

However, while allowing the handling of tremendous data rates of up to several terabits per second (in the case of network switches), most of the presented approaches focus on parallel protocol processing of many low-volume flows. The synchronization overhead can be minimized by partitioning the incoming packets depending on the flow they belong to, because two flows do not share the same state and hardware resources. The partitioning is realized by the classification of packets and assigning of packets to the correct destination queue. Approaches that also focus on high volume flows, such as MultiEdge [15], rely on optimized synchronization between the processing threads.

Offloading is used mostly in packet processing solutions. It is commonly implemented seamless, i.e., the accelerator is integrated in a way that reduces the number of copies [32, 34, 35, 47, 49, 51]. However, a disruptive integration, i.e., the packet data is firstly received from the NIC and then passed to the accelerator, is also commonly found [29, 30, 33] especially when GPUs are used as accelerators. Offloading the protocol processing into custom hardware in order to reduce the load of the communication endpoints is the exception in the reviewed solutions. This is because of the inflexibility and complexity of hardware-only solutions such as TCP Offload Engines [21, 22]. However, two seamlessly integrated TCP specific approaches, which tremendously decrease the TCP protocol processing effort by reducing the invocations of the host are LRO [25] and TSO [24].

Flexibility in protocol processing is mostly used for channel bonding [36, 39, 42] and for adapting protocols for certain communication conditions [41, 43, 39, 15, 77]. However, the flexibility focuses on the protocols (e.g., changing the acknowledgement scheme), leaving out questions about resource consumption and parallelism. Flexibility in packet processing is common nowadays after SDNs were introduced. It is mostly used to increase the maintainability of networks [31, 32, 33, 47, 49], and the introduction of new network functions [54]. The highest flexibility of the reviewed solutions is offered by P4 [47, 49], as it does not only allow to manage the network, but also permits the adaption of the processed protocols.

Summarizing, none of the reviewed processing concepts fulfill all of the desired features.

Especially the possibilities of parallel protocol processing are used only rudimentary. One of the reasons is that communication protocols were not designed with the idea of parallelization at the endpoints in mind. Furthermore, changes in established protocols are difficult to deploy, as the example of MPTCP shows. Flexibility is supported to a certain degree by several of the concepts, however, in all cases that allow on-demand reconfiguration, the transmission is stalled, and no parallel execution is possible. Offloading is widely supported, however, not in combination with on-demand reconfiguration and parallel protocol processing.

The second part of this chapter has focused on stream-processing architectures for data analysis, which showed that these systems do not fit communication protocols very well. The main reason is given by the different perspectives: For example, large scale systems [57] focus on maintainability rather than efficiency. Nevertheless, these systems provided valuable insights: Static stream distribution as a function of the outputs [64] can be used to distribute processing tasks efficiently. This can be combined with a simple back pressure flow-control [60]. However, the back pressure should be applied implicitly without the additional management overhead.

Altogether, the stream processing paradigm is generally fitting for communication protocol processing. Furthermore, it allows a structured analysis that provides performance estimations and hints for the scheduling, as shown in the last part of this chapter. Summarizing, the following conclusions will guide the remaining of this thesis:

We need to rethink the protocol processing.

In order to process high volume data streams, the protocol processing has to be parallelized. However, in contrast to processing thousands of individual streams that can be processed in parallel by partitioning the individual streams to distinct processing units, the processing of high volume streams itself must be parallelized. A simple multi-threaded approach, however, would lead to synchronization overhead due to reading/writing shared states or accessing buffers. While optimized fine-grained synchronization can help to reduce the overhead, it is error-prone and cumbersome. This thesis proposes a stream processing based protocol design process that allows implementing scalable protocols without the need for explicit synchronization.

The protocol processing should be offloaded.

While offloading the whole protocol processing to special-purpose hardware is inflexible, leaving the protocol processing to the host is no alternative either. The golden path is a heterogeneous offloading in which each part of the protocol processing is conducted on the most suitable hardware. Therefore, stateless and compute-intensive protocol tasks, such as Cyclic Redundancy Check (CRC) and Forward Error Correction (FEC) computation, should be done in special-purpose hardware. The stateful parts of the protocol processing have to be offloaded to a programmable NIC. The offloading hardware should be integrated seamlessly and take the natural protocol dataflow into account.

Protocols should be simple and specialized.

As an indirect insight from this chapter's review, a lot of research was complicated due to complex protocols and designs, or was undertaken in order to preside over existing complexity. The protocol design becomes cumbersome and error-prone when the protocol has to comply with many special cases. This can be avoided by focusing on certain communication conditions, such as a certain bit error rate, instead of designing a jack-of-all-trades. However, such narrow focus limits the applicability of the protocols. This can be circumvented by providing sets of simple and specialized protocols that are used, adapted, and combined automatically when the need arises. This thesis provides a concept for switching and adapting protocols on-demand at runtime.

CHAPTER 3

Soft Real-Time Stream Processing

This thesis investigates the assumption that interpreting communication protocols as soft real-time stream processing problems eases the protocol implementation process because the stream processing paradigm implicitly exposes parallelism, allows straight forward partial reuse of protocol implementations, and is predestined for partial offloading of protocol processing tasks. The implementation process envisioned in this thesis is, as sketched in figure 3.1, divided into five steps, each focusing on a single aspect.

In the design-step, the protocol processing is broken down into the protocol's processing tasks, such as the retransmission of lost packets. Each task is an isolated building block that is represented as a stream processing step, hereafter called *stage*. The stages are connected by message streams, which carry the necessary data and control information between the independent stages. The streams follow the protocol's inherent dataflow. The design-step results in a stream processing graph, hereafter called *processing engine*.

The analysis-step comprises the performance and soft real-time analysis of the processing engine. The analysis focuses on the processing engine's soft real-time requirements (what data rate is required per stage) and the performance characteristics (what data rate can be handled by the target hardware). The requirements and the performance characteristics are used to estimate the necessary parallelization that allow a protocol implementation to handle certain data rates.

The analysis's outcome, i.e., the processor utilization, is then used to adapt the processing engine for the target hardware in the adaptation-step. Due to the stream processing approach, the individual stages are only dependent on their inputs and their internal state. Therefore, the desired data rate can be achieved by adapting the processing engine with the help of stream operators that manipulate the path and data rate of streams by splitting, joining, and duplicating them. The result is a processing engine for the protocol that can handle the data rate on the targeted processing hardware.

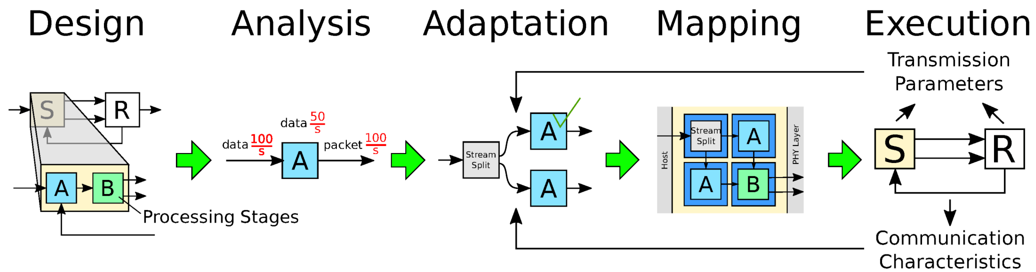


Figure 3.1: Stream processing based protocol implementation (adapted from [78]).

In the mapping-step, the adapted processing engine is mapped onto the target hardware and distributed over the communication hardware. Here again, the performance characteristics are used, this time as a cue for the mapping. In case a stage cannot be executed on the desired target hardware (e.g., due to an extremely high number of necessary Central Processing Units (CPUs)), a stage can be offloaded onto an external special-purpose accelerator, such as a Field Programmable Gate Array (FPGA) [79].

The final step is the execution of the stream processing based protocol implementation. During a transmission, the communication conditions are not static. The transmission parameters, such as the desired data rate, as well as the communication conditions, such as the channel quality, change over time. Since both affect the adaptation parameters that were established during the soft real-time analysis, a readaptation and mapping can be necessary at runtime.

The remainder of this chapter is used to describe the design process and explain the concept with the help of a simple example.

3.1 Decomposition of Protocols into Processing Stages

The first step of the design process is the transformation of the communication protocol into a processing engine by defining stages for individual protocol tasks. The granularity of the separation into stages determines the amount of exploitable parallelism, i.e., the finer the separation, the more parallelism can be exploited. Separating a protocol into stages is an experimental process, which requires knowledge about the protocol. However, a useful starting point for identifying coarse-grained processing tasks is using the protocol's

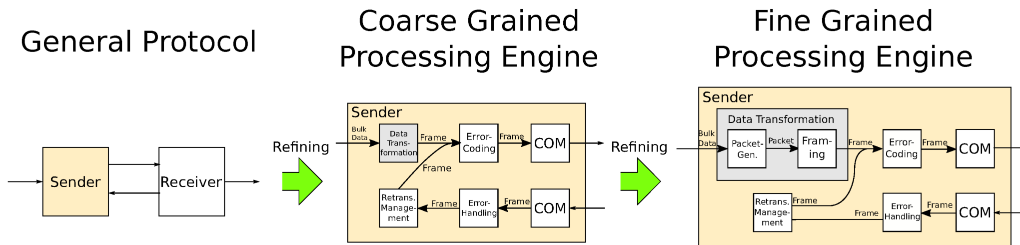


Figure 3.2: The refining process for the sender-side processing engine depending on the dataflow of the protocol from the general protocol to a coarse grained protocol separation finally resulting in a fine-grained processing engine.

dataflow. The protocol's dataflow can be extracted from the protocol description (e.g., textural or finite-state automata).

The refinement process is shown for the sender side in figure 3.2. It starts with the general protocol that is separated into coarse-grained stages. The dataflow of the example starts with a stream of bulk-data that is passed to the Data Transformation stage where the bulk-data is transformed into 10 packets. The packets are wrapped into network frames and forwarded to the `Error-Coding` stage before they are sent by the Communication (COM) interface to the receiver.

The coarse grained receiver processing engine (not shown) handles possible errors in the frames (`Error-Handling` stage) and reassembles the original chunk of Bulk Data by copying packets from the incoming frame into a local data-buffer. Packets that were transmitted correctly are forwarded to the Retransmission Management stage, which extracts retransmission information and eventually generates an acknowledgment. The acknowledgments are treated in the (receiver's) `Error-Coding` stage and then transmitted via the COM interface to the sender. In turn, the sender checks the incoming acknowledgments for errors (sender-side `Error-Handling`) and then retransmits missing frames.

However, the coarse-grained stages have to be further refined (also shown in figure 3.2). This is done by identifying compound protocol processing tasks that can be further separated. The data transformation stage of the sender is a strong candidate for further refinement because it combines two protocol tasks: Firstly, it is responsible for transforming a piece of bulk-data into a sequence of 10 packets (`Packet-Generator`) and secondly, it is responsible for copying these packets into frames (`Framing`). Neither of these tasks depend on the other and, more importantly, they have different runtime requirements.

The separation of the bulk-data into packets can be done by simple pointer arithmetic, whereas copying the generated packets into frames includes expensive memory operations (e.g., memcpy). The resulting processing engine is used in the next step for the soft real-time analysis.

3.2 Soft Real-Time Analysis

The stream processing implementation of a communication protocol provides implicit parallelism due to the pipelining of stages. However, the desired data rate may not be reached due to bottlenecks in the pipeline, e.g., if a stage is not able to process its input messages fast enough, it has to stall the pipeline or risk the loss of messages. Identifying these bottlenecks is the task of the soft real-time analysis. The analysis of a processing engine consists of the following two steps:

1. Requirements Estimation – Determining the processing requirements of all stages in a processing engine given a certain data rate.
2. Performance Measurements – Estimating the runtime of each individual stage given a certain hardware configuration.

3.2.1 Soft Real-Time Requirements

The soft real-time requirements of the processing engine are an inherent property of each communication protocol, as they describe how each part of a protocol has to perform in order to meet the desired data rate. The principle is shown in figure 3.3.

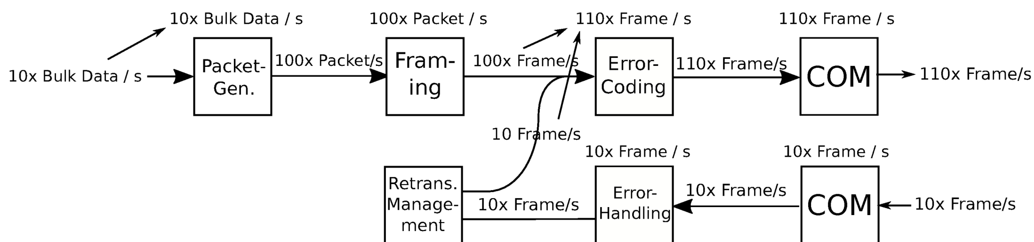


Figure 3.3: Soft real-time requirements for the example processing engine.

The soft real-time requirements are calculated by applying the desired data rate to the processing engine. In the example, the desired data rate is 10× Bulk Data per second. By applying this data rate to the input of the `Packet-Generator` stage, the input data rate becomes the soft-real time requirement, i.e., the `Packet-Generator` stage has to process 10 Bulk Data / s. The `Packet-Generator` stage produces 10 Packets for each received Bulk Data message. Consequently, the `Packet-Generator`'s output data rate is $\frac{100 \times Packet}{s}$. The output data rate is now applied to the input of the `Framing` stage. This procedure is continued until the requirements of all stages are estimated. In the case a stage has more than one input stream, such as the `Error-Coding` stage, the soft real-time requirement of that stage is calculated as the sum of the incoming data rates.

3.2.2 Performance Measurement

The goal of the performance measurement is to describe the performance of a stage on a given hardware configuration. In contrast to in-deep performance investigations, that result in a fine and accurate analysis [80], the performance characteristics are measured from a high-level that provides a coarse-grained summary of a stage's runtime behavior. Therefore, the performance characteristics provide an insight into a stage's execution time without the effort of a deep investigation[1].

Conceptionally, performance characteristics state the maximum input data rate a stage can process given that the stage has exclusive access to the investigated hardware. Each stage is measured individually as shown in figure 3.4. A producer stage provides the input

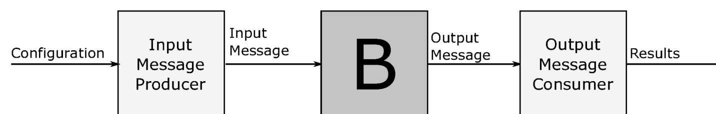


Figure 3.4: Simplified measuring setup for the estimation of the performance characteristics, e.g., necessary processing time per message, of stage `B`. The Input Message Producer sends messages with the maximum possible data rate and the Output Message Consumer receives and consumes the output messages.

[1] This is conceptually a Worst-Case-Execution-Time (WCET) problem, i.e., WCET analysis tools and methods could be also used. However, due to the soft real-time character of communication protocol processing that can tolerate missing a deadline, the accuracy demands on the performance characteristics are less strict.

stream for the measured stage and the output streams are consumed by a receiver stage. The performance characteristics, e.g., the number of processed messages per second or the execution time per message, are measured during that benchmark.

3.3 Adaptation of the Processing Engine and Offloading of Stages

After estimating the soft real-time requirements and measuring the performance characteristics of all stages, the processing engine is ready for the adaptation. Figure 3.5 shows the sender-side processing engine with soft-real time requirements and hardware capacities.

The comparison of the soft real-time requirements with the performance characteristics shows that the `Framing` stage and the `Error-Coding` stage pose bottlenecks, as the maximum achievable data rate is lower than the soft real-time requirements. These bottlenecks are removed by the adaptation of the processing engine. Consequently, the goal of the processing engine adaptation is to construct a parallelized processing engine in which all stages perform at the desired data rate, without changes to the implementation. During the adaptation step, the analysis's outcome is used to fit the processing engine to the target hardware, assuming that each stage can monopolize its own processor core. The adaptation is carried out with the help of *stream operators*.

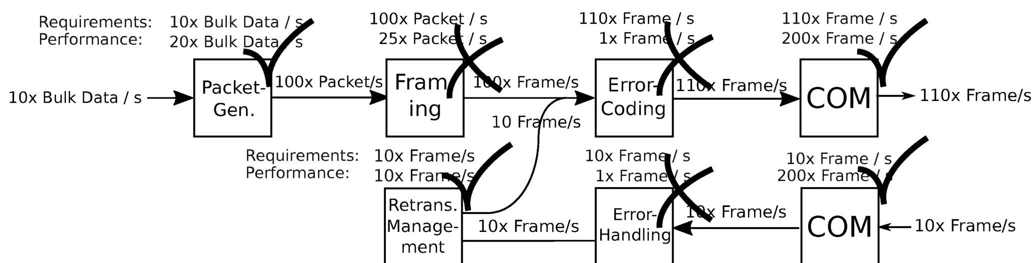


Figure 3.5: Soft real-time requirements and performance characteristics of the processing engine. The `Framing`, `Error-Coding` and `Error-Handling` stages pose bottlenecks because the soft real-time requirements are higher than the measured performance characteristics.

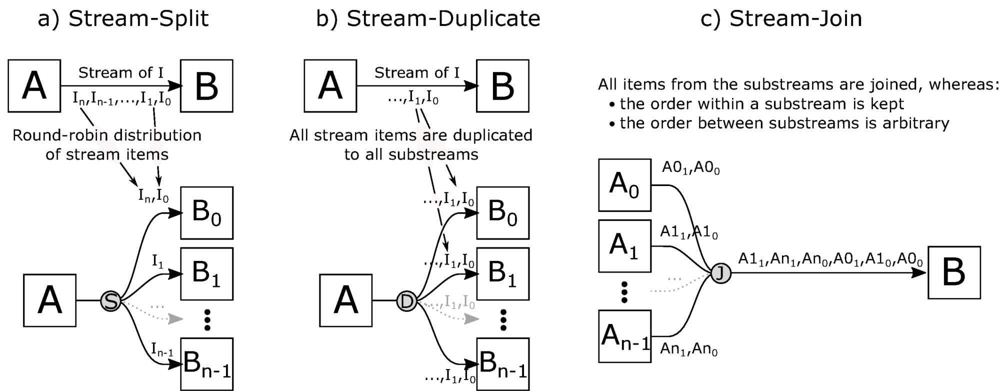


Figure 3.6: The stream operators manipulate the message stream without concern for the message's content. The possible operators are: a) Stream-Split b) Stream-Duplicate, and c) Stream-Join

3.3.1 Stream Operators

A stream operator manipulates the data rate of individual streams without concern for the streamed messages. The following three stream-operators are used for the adaptation.

- **Stream-Split** – The split-operator is used when the data rate of the stream between two stages is too high and must be reduced. The splitting of the stream is done by distributing its individual messages round-robin to different sub-streams (Fig. 3.6a). Consequently, the stream-split creates n sub-streams with a data rate of:

$$datarate_{sub-stream} = \frac{datarate_{source}}{n}$$

- **Stream-Duplicate** – The stream-duplicate operator is used to clone a stream (Fig. 3.6b), resulting in several sub-streams with the data rate of the source stream. This is used for example, when subsequent stages have to be notified about a new stage-status, or when the streamed messages shall be processed redundantly.

- **Stream-Join** – The stream-join is used when several source streams with the same message type have to be combined in a single destination stream (Fig. 3.6c). The data rate of the destination stream is:

$$datarate_{destination} = \sum_{source=0}^{\#Sourcestreams} datarate_{source}$$

3.3.2 Adaptation

The results of the soft real-time analysis states that the `Framing` stage can not handle the desired data rate on a single processor of the chosen hardware. Consequently, the `Framing` stage has to be parallelized. Parallelizing the processing of a stream is carried out with the Stream-Split operator, as it reduces the data rate, whereas the ratio between requirements and hardware characteristics states the number of sub-streams that are necessary. In the case of the example processing engine 4 sub-streams are sufficient. The new processing engine after applying the Stream-Split is shown in figure 3.7a.

3.3.3 Offloading

After applying the stream-split operator, the soft real-time requirements of the `Framing` stage fits the measured performance characteristics of the target hardware. However, the `Error-Coding` stage and the `Error-Handling` stage can still not fulfill the requirements. Furthermore, the ratio between requirements and hardware characteristics of both stages state a high parallelization count, which makes these stages candidates for offloading in special purpose hardware.

The stream processing paradigm is predestined for offloading parts of the protocol processing because two stages can only depend on the streams between them and are

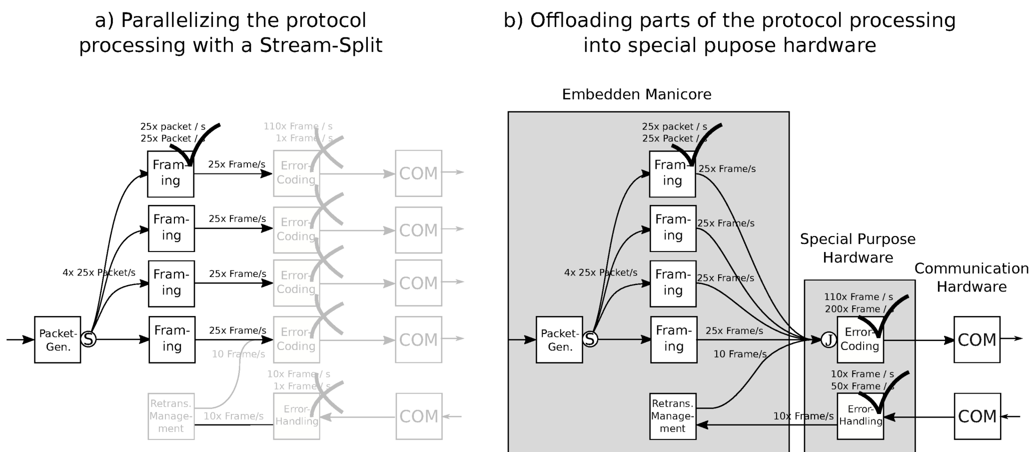


Figure 3.7: Soft real-time requirements and performance characteristics of the processing engine.

otherwise isolated. After offloading both stages, the final processing engine looks as presented in figure 3.7b.

The number of necessary `Error-Coding` stages could be reduced by offloading to one stage, consequently the output streams of the `Framing` stages are joined with a stream-join at the input of the `Error-Coding` stage.

3.4 On-Demand Adaptation of Processing Engines

The last part of this chapter focuses on how to provide flexibility to the protocol processing, while still using statically built protocols. Locating and avoiding processing bottlenecks by employing the planning approach helps to design suitable protocols for static conditions. However, actual communication systems can not assume static communication conditions without affecting the quality of service. On one side, the communication requirements, such as the desired data rate, can change. On the other side, the communication conditions, such as the channel quality, are not necessarily static either. This can lead to a situation in which a communication protocol and its implementation is carefully optimized for the wrong parameters, which, in turn, leads either to wasted resources or to performance degradation. Such a situation arises when the protocol implementation is not able to handle the new communication conditions and/or requirements efficiently.

Avoiding wasting resources, as well as performance degradation, can be achieved using a suitable processing engine at all times, which means that the protocol processing has to be changed at runtime. Depending on the situation, it can be sufficient to readapt the currently used processing engine. However, in some cases, a complete processing engine can be unsuitable and has to be replaced. Both approaches are explained in the following.

The on-demand adaptation, as well as the replacement of processing engines, use the Processing Engine Template Language (PETL). The PETL is a graph description language that describes the stages, as well as the connections, and provides the configuration for the individual stages. More information about the description language can be found in [81].

3.4.1 Readaptation of a Deployed Processing Engine

The readaptation of a currently used processing engine is used when the implemented protocol is still generally suitable, but a change in the communication conditions lead to over- or under-utilization of the assigned resources. That happens, for example, when the host changes the data rate. Readaptation can be used for adding, as well as removing stages to/from the processing engine.

Figure 3.8a presents how the parallelization count can be increased. During the preparation of the readaptation the new stages are built and connections between them are added according to the PETL description of the new processing engine. However, before the stages can actually be used, the connected stages may have to distribute state information, such as the currently used buffers, to the newly added stages, and the newly added

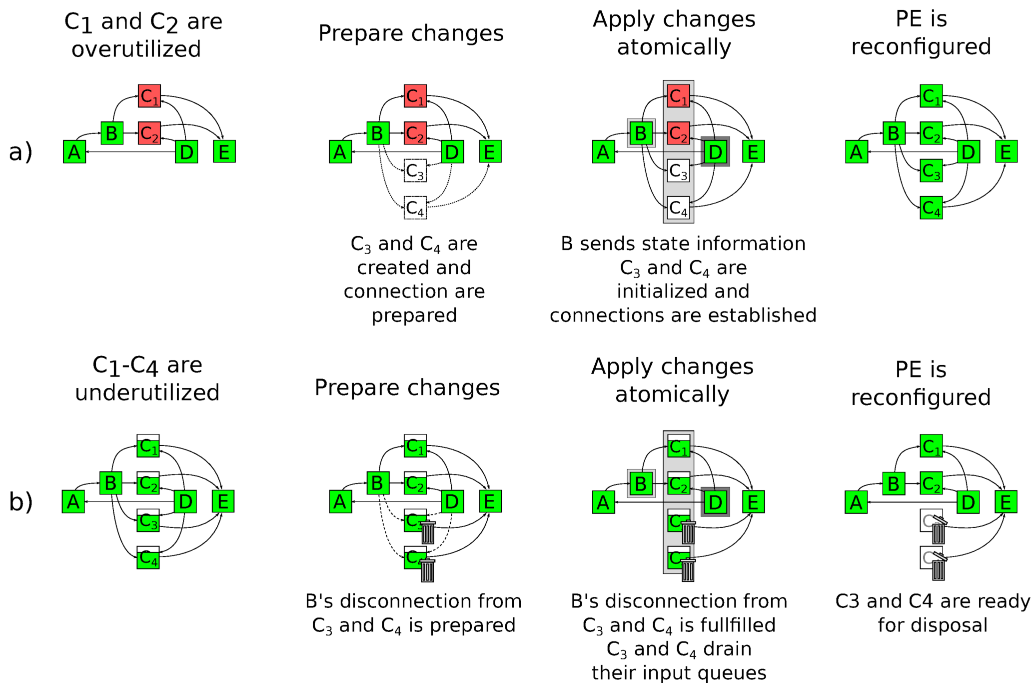


Figure 3.8: Readaptation process for a processing engine: a) When the stages C_1 and C_2 are over-utilized, the a re-adaption adds more stages of type C to the processing engine. b) When the stages C_1 to C_4 are under-utilized, then a readaptation may remove stages of type C.

stages have to be initialized. In order to avoid employing uninitialized stages, the state distribution, as well as the initialization of the new stages, are conducted atomically. Afterward, the processing engine can be used.

Removing stages from a processing engine is shown in 3.8b. The removal has to be done in two steps: Firstly, the stages that are about to be removed are marked as potentially disposable and all incoming connections are marked as to-be-deleted. However, all outgoing connections have to stay connected to the remaining processing engine. These changes are, again, carried out atomically. Since the inbound connections are deleted, the stages that will be removed have no part in the ongoing protocol processing. However, all messages that were already sent to these stages will still be processed. When the draining of the removed parts of the processing engine will be finished, i.e., when all remaining messages have been processed, the stages will be ready for disposal.

3.4.2 Switching Protocols by Entirely Replacing Processing Engines

In some cases, it is not sufficient to readapt the currently employed processing engine because the implemented protocol itself becomes unsuitable, e.g., when a different acknowledgment mechanism is better suited for the ongoing transmission. Altering the processing engine's protocol for an ongoing transmission is undesirable because the internal protocol state would then have to be reinterpreted for the new protocol. Alternatively, stopping the transmission and restarting the communication with a new processing engine that implements the new protocol is possible, but would severely impact the performance.

These problems can be drastically reduced by applying the adaptation approach to an entire processing engine that is currently used (published in [82]). Situations that require a completely different protocol are then handled by replacing the currently used processing engine with a new, suitable processing engine at runtime.

Nevertheless, employing two processing engines at the same time requires that the data streams from an underlying layer, such as the Medium Access Control (MAC) layer, are forwarded to the correct protocol processing engine. Multiplexing different processing engines for a single transmission is achieved by virtualizing the communication interfaces with virtual channels. A virtual channel, as shown in figure 3.9a, is an identification number that is added by the sender to outgoing messages, such as network frames. On the receiver side, the virtual channel is read by the abstraction of the physical interfaces

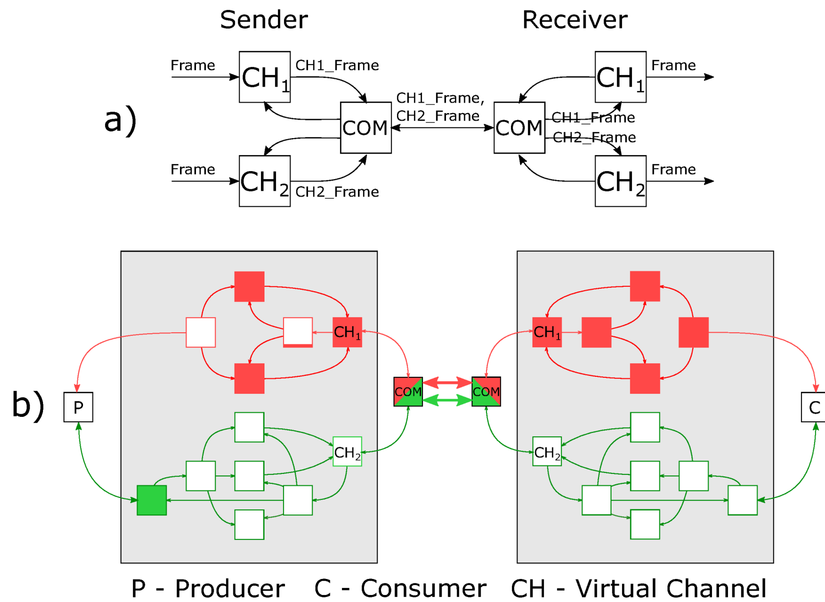


Figure 3.9: a) A virtual channel (CH) is used to distinguish the data streams between two processing engines that are using the same communication interface. b) Replacement procedure (adapted from [82]) of a processing engine. Virtual channels are used to multiplex the original processing engine (red) and the new processing engine (green) on the same communication interface. The old processing engine is drained, while the new processing engine is already used for the ongoing transmission in order to avoid disruptions in the protocol processing.

in order to determine the correct processing engine for a message. Since the virtual channel identifier does not change the protocol nor the implementation, it can be used with any processing engine. The idea is similar to the port-number field in the IP header [37]. However, the virtual channel does not multiplex different connections on a single communication interface, but different protocol implementations for a single connection. The usage of virtual channels is shown in figure 3.9b. Two different processing engines are used simultaneously for one transmission. The original processing engine (red) does not receive any more data from the producer and is only used until all stages are drained, however, the new processing engine (green) is already employed by the producer for new data. The virtual channel multiplexes both processing engines onto one COM interface.

3.4.3 Exchanging Processing Engines

A communication system always consists of the sender and the receiver. However, the trigger for a readaptation or replacement may occur only at one of the hosts. For example, the sender host requests an increase of the data rate to the extent that makes it necessary to employ an additional communication interface. While the sender can readapt its processing engine depending on the data rate information, the receiver will continue with the old processing engine. This problem is usually solved by some negotiation-process, in which the sender and receiver agree on the terms of their transmission, such as data rate or the used communication channels and interfaces.

However, the negotiation-process takes time that could be already used for building and deploying the new processing engine. Skipping the negotiation-phase and "forcing" a readapted processing engine on the other communication endpoint allows to shorten the time between adaptation decision and employing the new processing engine. This is done by sending the specialized processing engine, coded as a PETL description, to the other communication endpoint for implementation. The "handshake" follows afterward, when the forced endpoint acknowledges the successful readaptation. In the case the forced endpoint declines the PETL, it can propose another processing engine.

CHAPTER 4

A 100 Gbit/s Data Link Protocol

This chapter is used to apply the proposed design concept to a prototype wireless data link protocol. The protocol is meant to connect two wireless endpoints, in which the endpoints have exclusive access to a simplex medium, such as a line-of-sight radio connection [5]. The main objective is providing a goodput that is close to the theoretical maximum. The goodput is provided to the host as an ordered lossless data stream. Due to the simplex connection, an additional communication channel from the receiver to the sender is employed for acknowledgments and control information.

In the first part of this chapter, the 100 Gbit/s wireless data link protocol and its data structure are presented. Afterward, the protocol design process is applied to the data link protocol.

4.1 The Prototyp Data Link Protocol

The targeted communication data rate of 100 Gbit/s and beyond can only be reached when the whole protocol is designed with the communication conditions in mind. The first challenge of ultra high-speed communication is the high protocol processing costs, which easily monopolize the processing power of the communication endpoints. Consequently, the protocol processing has to be offloaded in order to free the host's resources for its actual task. Ideally, the host is only responsible for producing and consuming the payload and passing it on to the processing hardware, whereas the data-transport between host and protocol processing hardware should cause as little as possible invocations of the hosts.

The second main challenge is to use the available channel capacity efficiently. In order to reach a high transmission efficiency, the available channel capacity has to be used primarily for the transmission of payload. Consequently, meta data and preamble overhead should

be minimized. The preamble overhead can be reduced by using large frames. However, in a wireless scenario in which bit-errors are to be expected, large frames pose a severe problem due to their high frame-loss susceptibility. In order to reduce the risk of losing complete frames, frames are segmented into sub-packets. Each sub-packet has a smaller size, and therefore the risk of losing the packet is reduced. Furthermore, the protocol employs Forward Error Correction (FEC) for the correction of faulty sub-packets and headers.

Figure 4.1 shows the protocol data structures specifically designed for high-speed wireless communication. The protocol data structure[1], their relationship to each other, and their purpose are explained in the following.

Offloading the protocol processing to an external accelerator alone is not sufficient, as the number of host-invocations should also be minimized. This is realized by organizing the host's input and output data streams in large packets of bulk-data, called *datachunks*. Using large datachunks, e.g., 16 MB, allows reducing host invocations drastically, compared to the standard MTU size of 1500 Byte of TCP. A datachunk is described by a datachunk-descriptor (`DataChunkDesc`) that contains the address of the payload, the datachunk's size, and a sequence number that is used for (re-)ordering. The datachunk size is a compromise between latency and host invocations, whereas the larger the datachunk, the higher the latency, and the fewer the host invocations.

The data link protocol employs three types of sub-packets: `DataPacket`, `AckRequestPacket`, and `AckPacket`. Each sub-packet starts with the payload. The payload is followed by a sub-packet specific footer. This order was chosen because it allows a buffer-less "on-the-fly" FEC/CRC coding in hardware. Each sub-packet contains a flag that states whether the sub-packet's payload is valid. The redundancy data for the reconstruction of frames is appended at the end of the frame. However, segmenting a frame into smaller sub-packets that all have their own header increases the meta data overhead. The meta data overhead is reduced by only allowing sub-packets of the same type and size within a frame. The sub-packet type and the payload size are both stated in the frame's header, i.e., the frame-header is shared by all sub-packets. Consequently, the sub-packets have to add only sub-packet specific meta data, such as CRC, and datachunk identifiers. The semantics of the shared header depend on the sub-packet type, which is explained in the following.

[1]Further technical details, such as the specific protocol fields and sizes are omitted here, but are provided in annex B.1.

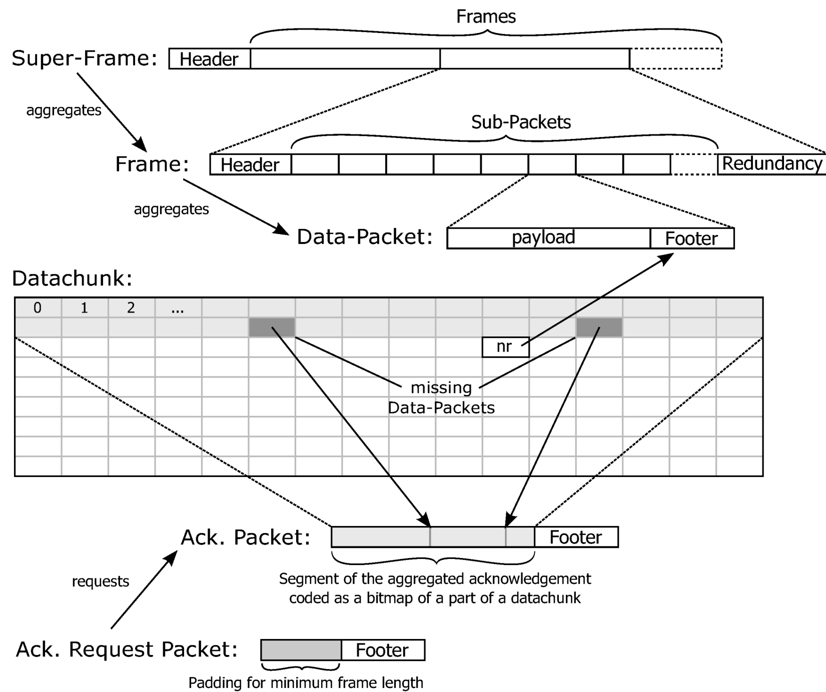


Figure 4.1: The data structures of the prototype link layer protocol (further developed from [7]). Each frame is segmented into smaller sub-packets of three types: Data-Packets carry data-payload that belongs to a datachunk, Ack-Request-Packets are used to trigger the receiver to (re-)send an aggregated acknowledgment, and Ack-Packets carry parts of an acknowledgment. Since each frame can only contain sub-packets of the same type, super-frames can be used to further aggregate frames of different types.

The `DataPacket` is used to transmit the payload. It contains a piece of the datachunk and an offset that states the position of the payload relative to the datachunk's beginning. The datachunk's id and all further datachunk related information are provided in the (shared) frame's header. A frame that transports `DataPackets` is called `DataFrame`.

The `AckPackets` are sent from the receiver to the sender by using a separate (back-)channel because switching between transmission and receiving at these high speeds takes time and should be avoided. However, due to cost considerations and in order to reduce energy consumption, the back-channel may have a significantly lower capacity than the channel used for the transmission of the payload. Consequently, the acknowledgment mechanism should minimize the transmitted data necessary for the acknowledgments.

The first measure that reduces the number of transmitted acknowledgments and minimizes acknowledgment processing-costs is sending acknowledgments only after an explicit request. This way, acknowledgments are only sent and processed when the sender actually needs them. The `AckRequest` is used to request an acknowledgment/final-acknowledgment from the receiver, and it does not contain an actual payload. However, when acknowledgments are only sent on request, a lost `AckRequestPacket` or `AckPacket` can seriously degrade the goodput. For this reason, the acknowledgment request provides a `Redundancy` field, stating the redundancy of acknowledgments that have to be sent in response to the acknowledgment request. This is especially useful when a high packet-loss rate is expected during the transmission. A frame that transports `AckRequestPackets` is called `AckRequestFrame`.

The second measure is reducing the data that has to be transmitted for an individual acknowledgment by removing internal redundancy. The protocol uses aggregated acknowledgments[2] that state all missing packets for a complete datachunk. The individual data-packets are coded as a bitmap (1 — received / 0 — missing). The bitmap is the payload of the `AckPacket`. Additionally, an acknowledgment can be flagged as *final*, which means it does not contain further missing data-packets. That allows ignoring the acknowledgment's payload data and consequently reduces the processing time and, therefore, the latency. A frame that transports `(Final-)AckPackets` is called `(Final-)AckFrame`.

Furthermore, the `AckPacket` has a field that states to which segment of the datachunk this acknowledgment packet refers. Consequently, the bitmap of a whole datachunk can be divided into "sub-bitmaps", which refer to a consecutive section of the datachunk. This approach is similar to the segmentation of data-frames and reduces the probability of losing complete acknowledgments due to bit-errors.

The proposed frame-format aggregates sub-packets of the same size and type that belong to the same datachunk. While this restriction reduces the necessary meta data per sub-packet, it can again lead to a high meta data overhead (e.g., preamble), in case the frame cannot be filled completely. This could be avoided by further aggregating frames into `SuperFrames`.

[2]The acknowledgment is a combination of ACK and NACK, however, due to readability reasons, it will be referred to as acknowledgment.

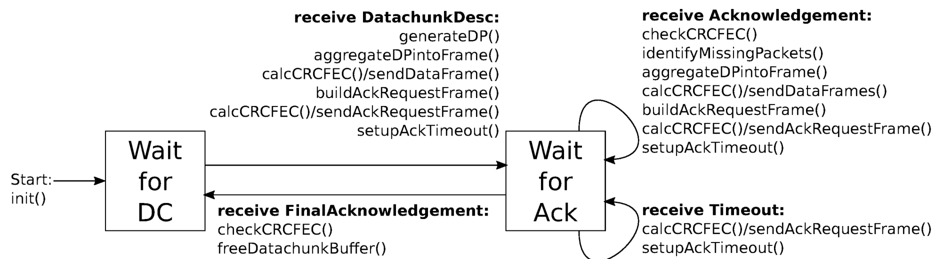


Figure 4.2: Coarse-grained EFSM of the sender-side protocol.

4.1.1 Protocol Description

The prototype data link protocol builds upon the presented data structure and is presented in the following with the help of Finite State Machines (FSMs). The transitions of the state-machines are annotated with processing tasks[3] that will be assigned to the protocol processing stages in the decomposition step.

The Sender

Figure 4.2 shows the FSM of the sender-side data link protocol. The sender starts in the *Wait for DataChunk (DC)* state after the initialization. In the *Wait for DC* state, the sender waits for a `DataChunkDesc` that describes a datachunk-buffer to be processed. When a `DataChunkDesc` is provided by the host to the sender, the corresponding datachunk-buffer is processed by generating data-packets and aggregating data-packets into `DataFrames`. For each frame the error correction data (e.g. Cyclic Redundancy Check (CRC) code and the FEC data) are calculated before the frame is sent. Finally, after all data-packets of the current datachunk are transmitted, an `AckRequestFrame` is built and sent. Afterward, the sender activates a receive-timeout and switches into the retransmission state *Wait for Ack* and sends `AckRequestFrames` until an `AckFrame` is received. The receive-timeout ensures that a new `AckRequestFrame` is sent in the case no `AckFrame` is received.

When receiving the awaited `AckFrame`, the sender identifies missing data-packets and aggregates them into frames. After processing the acknowledgment, i.e., retransmitting all missing packets, the sender requests a new acknowledgment, i.e., sends an

[3]The connection-management is carried out by a separate management protocol (see annex A.1) and is omitted from the following protocol description.

`AckRequestFrame`, and waits. Eventually, all data-packets are transmitted correctly, and the sender receives a `FinalAckFrame` that states that all data-packets of the current datachunk were received correctly. Upon receiving such `FinalAckFrame`, the sender frees all memory that is related to the now completely processed datachunk and waits for the next datachunk to be transmitted.

The Receiver

Figure 4.3 shows the FSMs of the receiver-side data link protocol. On initialization, the receiver prepares a datachunk buffer for the first expected datachunk, and switches into the *Wait for new DC* state. The receiver waits in this state for a `DataFrame` that contains the first data-packets of the awaited datachunk or an `AckRequestFrame` for the last processed datachunk. The further processing depends on the type of the received frame:

- In the case an `AckRequestFrame` is received, and it belongs to the last processed datachunk, the receiver answers with the `FinalAckFrame` of the last datachunk and stays in the *Wait for new DC* state.
- In case a `DataFrame` is received and the frame belongs to the new datachunk, the now outdated aggregated acknowledgment of the last datachunk is discarded. Furthermore, the receiver copies the correctly transmitted data-packets into the prepared datachunk buffer and marks the correctly transmitted data-packets of the `DataFrame` as *received* in the new aggregated acknowledgment. Finally, the receiver switches into the *Wait for DataPacket (DP)* state.

The processing continues similarly in the *Wait for DP* state. Upon receiving a `DataFrame`, the correctly transmitted data-packets that belong to the current datachunk are copied into the datachunk buffer and are marked accordingly in the current acknowledgment. When an `AckRequestPacket` is received, the current `AckFrame` is built and sent.

Upon receiving the last data-packets of the currently processed datachunk, a `FinalAckFrame` is sent and the completed datachunk buffer is forwarded to the host. Afterward, the receiver allocates and prepares a new buffer for the next datachunk. Finally, the receiver switches into the *Wait for new DC* state, in which it keeps answering incoming `AckRequestFrames` while waiting for the first `DataFrame` of the next datachunk.

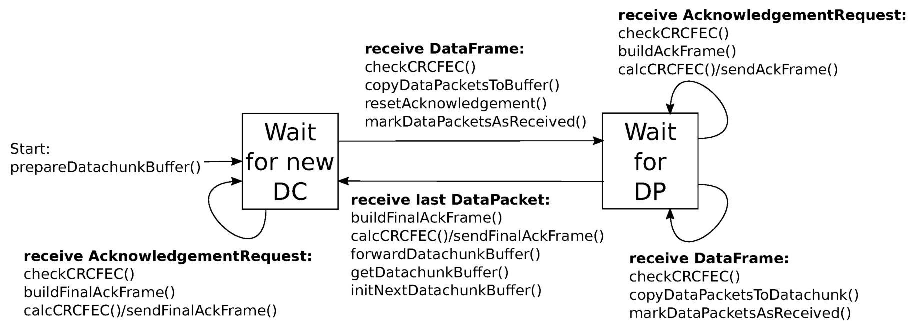


Figure 4.3: Coarse-grained EFSM of the receiver-side protocol.

4.1.2 Summary

In this section, the data link protocol that will be used in the remainder of this thesis was presented. The main advantage of the proposed protocol is its high parallelizability. The proposed frame-format allows for easy parallelization because all frames are self-sufficient, i.e., the combination of frame and sub-packets provide all meta data necessary for its processing. That means that data-packets can be aggregated in parallel into frames (and copied into the destination datachunk buffer) in any order as long as the data-packets belong to the same datachunk. For this reason, frames can be processed in parallel also because they provide the datachunk information. The acknowledgments can be processed in parallel based on the segment, i.e., one processor can process the first ack-packet in an acknowledgment frame, and a second processor can process the second ack-packet.

The remainder of this chapter will use the stream processing design approach in order to implement the protocol. The design process consists of the following steps, which are conducted in separate sections:

1. Decomposition – Identify protocol tasks, encapsulate them into stages, and connect the stages in order to form a protocol processing engine.
2. Soft Real-Time Analysis – Calculate the soft real-time requirements and measure the performance characteristics of the individual stages.
3. Adaptation – Reduce the soft real-time requirements by parallelization.
4. Mapping – Distribute stages over the communication system depending on their performance characteristics.

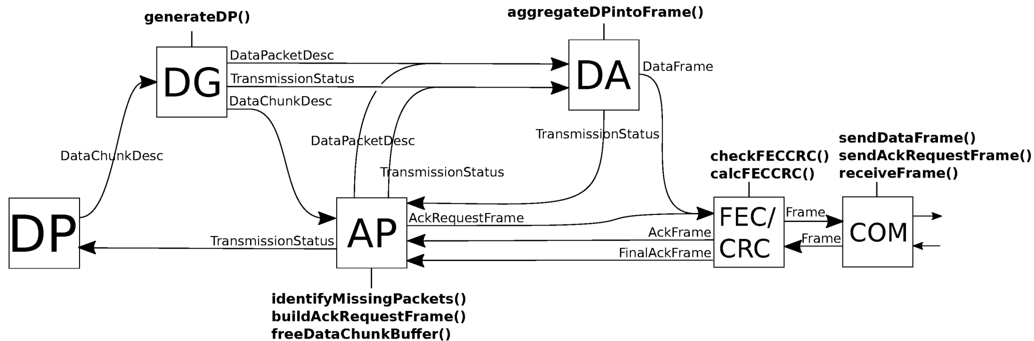


Figure 4.4: The sender side processing engine of the presented data link protocol. The processing tasks mentioned in the protocol's sender and receiver EFSM are assigned to the corresponding stages.

4.2 Decomposition of the Protocol into Processing Stages

The protocol implementation starts by wrapping the protocol tasks from the protocol's Extended Finite State Machine (EFSM) into processing stages and connecting these stages in order to form a processing engine. Figure 4.4 (sender) and 4.5 (receiver) show the processing engine for the prototype data link protocol. The individual stages and the protocol's dataflow is explained in the following.

The `Data Producer (DP)` represents the host that allocates buffers and fills the buffers with data to be transmitted. The actual protocol processing starts when the `Data-Packet Generator (DG)` receives a datachunk descriptor (`DataChunkDesc`) from the host. That descriptor is immediately forwarded by the `DG` to the `Acknowledgement Processor (AP)`, in order to notify the `AP` that the processing of a new datachunk has started. Afterward, the `DG` cuts the corresponding datachunk buffer into `DataPackets` and forwards the individual data packet descriptors (`DataPacketDesc`) to the `Data-Packet Aggregator (DA)`. After cutting the datachunk into pieces, the `DG` notifies the `DA` that all data-packets were forwarded, by sending the current status of the transmission (`TransmissionStatus`) to the `DA`.

The `DA` receives the individual `DataPacketDesc` messages and aggregates them into `DataFrames`. Upon receiving the `TransmissionStatus` notification from the `DG`, that states that the last data-packet of the current chunk was given to the `DA`, the `DA` finalizes and

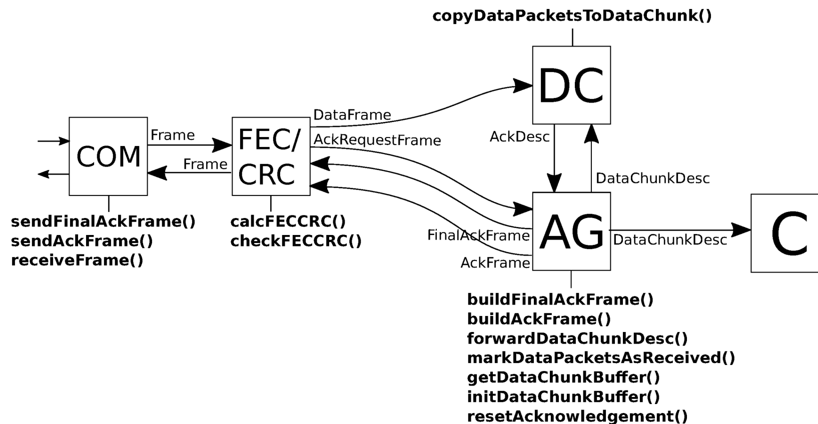


Figure 4.5: The receiver side processing engine of the presented data link protocol. The processing tasks mentioned in the protocol's sender- and receiver EFSM are assigned to the corresponding stages.

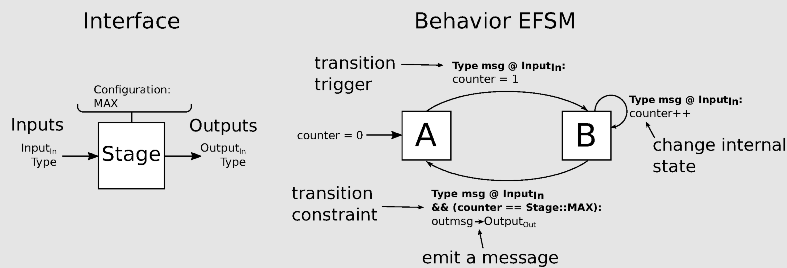
sends the current `DataFrame`, and forwards the `TransmissionStatus` notification to the `AP`. Upon receiving the `TransmissionStatus` message from the `DA`, the `AP` then knows that all packets of the current datachunk went through the `Data-Packet Aggregator`. Consequently, it sends an `AckRequestFrame` and switches into the retransmission mode in which it accepts incoming `AcknowledgmentFrames`. All outgoing frames are processed by the `Forward-Error-Correction/Cyclic-Redundancy-Check (FEC/CRC)` stage before they are transmitted by the `Communication (COM)` interfaces.

On the receiver side, each incoming network frame (`Frame`) is checked for errors which are then corrected by the `FEC/CRC` stage. Depending on the `Frame`'s type, the `Frame` is then forwarded. `DataFrames` are passed to the `Data-Packet Combiner (DC)` and `AckRequestFrames` are passed to the `Acknowledgement Generator (AG)`.

The `DC` copies the payload from the valid data-packets into the current datachunk-buffer (which was earlier received from the `AG`). The IDs of the data-packets that have to be acknowledged (`AckDesc`) are passed to the `AG` that marks each of the packets as correctly received. In case all packets of the current datachunk are received, the `AG` sends a `FinalAckFrame` to the `Forward Error Correction` stage and acquires a new datachunk buffer that is given to the `DC`. Upon receiving an `AckRequestFrame`, the `AG` answers the sender with the current `AckFrame`.

Technical Details 1: Describing Processing Stages

Each stage is only dependent on its inputs and its internal state. Therefore, the behavior of each stage can be presented by an isolated EFSM. The transitions of the EFSM are triggered by messages received over the input streams. The transitions can be further constrained by conditions, such as the message's content. Each transition can define a list of actions that are associated with that transition. An action can change stage-internal variables, call a method, or emit messages via the outputs. Furthermore, transitions are executed atomically, i.e., the processing of a message is not interruptable.



The transformation of an EFSM into a compilable stage implementation is straightforward and could be even done automatically [83]. Once the initial implementation is finished, further optimizations can be applied to the implementation. Furthermore, the description of the stages as state machines allows the use of validation approaches, such as checking for deadlock freeness.

After receiving and checking the `Frame` at the sender-side's `FEC/CRC` stage, the `(Final-)AckFrame` is forwarded to the `Acknowledgement Processor (AP)`. Upon receiving a `FinalAckFrame`, the `AP` notifies the `Data Producer (DP)` that the protocol pipeline is free and a new datachunk can be sent. In case an `AckFrame` with missing data-packets was received, the missing packets are wrapped into `DataPacketDesc` messages which are given by the `AP` to the `DA`. After all missing data-packets from the `AckFrame` are transmitted, the `AP` notifies the `DA` with a `TransmissionStatus` message that it should send the current `DataFrame`. This notification is sent back to the `AP` after the `DataFrame` was sent. In turn, the `AP` sends a new `AckRequestFrame` and waits for the response from the receiver.

4.2.1 Implementation of the Processing Stages

In order to implement the processing engine, the behavior of the stages has to be defined. The behavior definition can be done with an EFSM, as shown in Technical Details 1. The Data-Packet Aggregator (DA), which is used to aggregate data-packets into network frames, shall serve as an example[4].

Figure 4.6 shows the interface of the DA. Each DA is configured with a buffer-pool where it can allocate network Frames, the maximum number of data-packets per network frame (DPPerFrame), and the size of the data-packets (DPSize) it aggregates.

The Data-Packet Aggregator (DA) has two inputs: The $\texttt{DataPacketDesc}_{\texttt{In}}$ ($\texttt{DPD}_{\texttt{In}}$) input that receives incoming DataPacketDesc messages, and the $\texttt{FinishFrame}_{\texttt{In}}$ ($\texttt{FF}_{\texttt{In}}$) input that receives TransmissionStatus messages.

The DataPacketDesc messages describe a consecutive number of data-packets. For that, it contains a chunk identifier (ChunkNr), the number of consecutive data-packets that the descriptor refers to, and the memory address and offset in the datachunk of the first (of these consecutive) data-packet's. Describing several consecutive data-packets with a single DataPacketDesc is a measure of reducing message passing load between stages. The $\texttt{FinishFrame}_{\texttt{In}}$ ($\texttt{FF}_{\texttt{In}}$) input receives TransmissionStatus messages. Transmission-Status messages are used to forward changes in the transmission state of a datachunk.

Figure 4.7 shows the EFSM that describes the behaviour of the DA. The DA starts in the *Empty Frame* state after it allocates its first Frame and configures it (setting the number and size of data-packets per frame). Upon receiving a DataPacketDesc message at the $\texttt{DataChunkDesc}_{\texttt{In}}$ ($\texttt{DPD}_{\texttt{In}}$) input, the payload of the data-packets is copied into the current network frame. In the case the frame was empty upon receiving the DataPacketDesc message, the DataFrame's header is configured with the DataPacketDesc's ChunkNr and SequenceNr. If the DataFrame is not yet full after processing the DataPacketDesc message, the DA switches to the *Non-Empty Frame* state, otherwise it forwards the full frame via its $\texttt{DataFrame}_{\texttt{Out}}$ ($\texttt{Fr}_{\texttt{Out}}$) output, then allocates a new Frame, configures the corresponding DataFrame, and remains in the state *Empty-Frame*.

[4]The implementation of the other protocol stages as EFSMs is provided in annex B

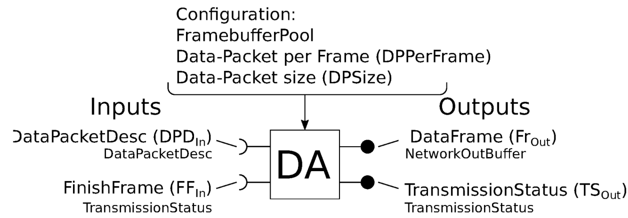


Figure 4.6: The interface of the `Data-Packet Aggregator (DA)`.

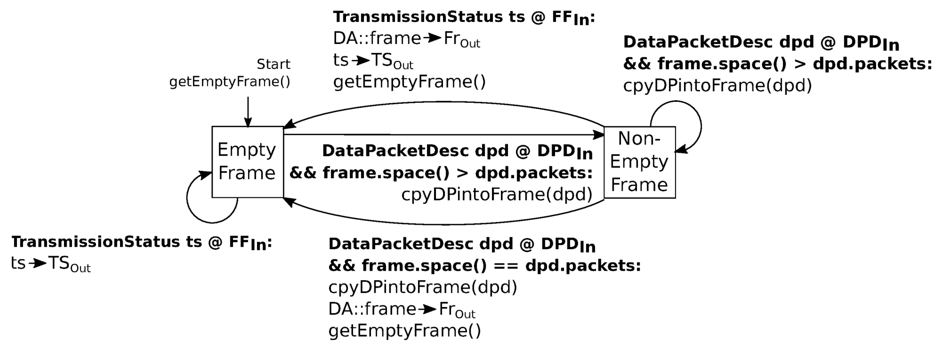


Figure 4.7: The EFSM of the `Data-Packet Aggregator (DA)`. The EFSM describes the behaviour of the `DA` depending on its internal state and the messages received at its inputs.

In the *Non-Empty-Frame* state, incoming `DataPacketDesc` messages are processed until the `DataFrame` is full. When the maximum number of data-packets is reached, the current data-frame is sent per $\texttt{DataFrame}_{\texttt{Out}}$ ($\texttt{Fr}_{\texttt{Out}}$) output, and a new `Frame` is allocated and configured. This continues until all data-packets, referred to by the `DataPacketDesc`, are copied into network frames. In the case the current `DataFrame` is empty after processing the message, the `DA` switches back into the *Empty Frame* state, otherwise it remains in the *Non-Empty-Frame* state.

The $\texttt{FinishFrame}_{\texttt{In}}$ ($\texttt{FF}_{\texttt{In}}$) input is used to forcefully finish a `DataFrame` that has not been filled completely, e.g., when no more data-packets are expected. The $\texttt{FF}_{\texttt{In}}$'s behavior depends on the `DA`'s state. In the case the `DA` is in the *Empty-Frame* state, the `TransmissionStatus (TS)` message is only forwarded via the `DA`'s $\texttt{TransmissionStatus}_{\texttt{Out}}$ ($\texttt{TS}_{\texttt{Out}}$) output. In case the `DA` receives a `TransmissionStatus` message while it is in the *Non-Empty Frame* state, the current `DataFrame` is sent, and a new empty `Frame` for future data-packets is allocated and configured. Finally, the `DA` switches into the *Empty Frame* state.

4.2.2 Summary

In this section, the data link protocol was transformed into a processing engine. The stage decomposition was done based on the protocol's dataflow and the previously identified protocol tasks. The high level implementation of the individual stages is presented by state-machines, which allows an easy transformation into compilable code, as well as for static protocol analysis such as deadlock freeness.

4.3 Soft Real-Time Analysis

After the decomposition of the protocol into a processing engine and implementing the stages, the soft real-time analysis can be performed. The analysis depends on the implementation and configuration of the protocol, as well as on the communication conditions. However, at this moment, only the implementation of the stages is known. The target data rate is given by the application and the configuration of the protocol depends on the expected channel conditions. Consequently, the first steps are choosing a target data rate and estimating the channel conditions. The target data rate for the following analysis is 40 Gbit/s. The channel conditions can be estimated given the transmission technology with applied error correction measures.

In this thesis, the expected Bit Error Rate (BER) of the channel is in a range from 10^{-5} and 10^{-6} after the FEC processing [84]. Since the protocol has to provide the desired data rate under the full range of expected channel conditions, the worst-case BER of 10^{-5} will be used for the configuration of the protocol.

The data link protocol is highly configurable, e.g., the size and number of data-packets per frame can be adjusted to the expected communication conditions. In order to determine the best protocol configuration, the actual data loss depending on the BER has to be calculated. Table 4.1 shows the packet loss probability, as well as the accumulated protocol overhead per datachunk and the accumulated data-loss per datachunk (overhead + lost packets), depending on the data-packet size given a BER of 10^{-5} (see Technical Details 2). Four data-packet sizes from 1024 Bytes[5] to 8192 Bytes were selected as candidates.

[5]For smaller packet sizes, the random number generator used to simulate the BER during the evaluation becomes a bottleneck.

Data-Packet size	Packet Loss Probability	Protocol Overhead per 16MB Datachunk	Lost Data per 16MB Datachunk
1024 Byte	8.2 %	139kB	1462kB
2048 Byte	16.4 %	77kB	3210kB
4096 Byte	32.8 %	48kB	7985kB
8192 Byte	65.6 %	46kB	31155kB

Table 4.1: Packet loss probability, protocol overhead (per datachunk), and lost data (per datachunk), depending on data-packet size for a Bit Error Rate (BER) of 10^{-5}.

The lowest overall data-loss is seen for the 1024 Byte data-packet size. Consequently, a data-packet size of 1024 Byte is chosen, which leads to a maximum of 8 data-packets per frame (the maximum number of 1024 Byte data-packets that fit in a 9000 Byte Ethernet[6] jumbo-frame [85]). Finally, a datachunk has a size of 16 MB, i.e., it consists of 16384 data-packets.

Since each data-packet is acknowledged by a single bit, the overall acknowledgment size is 2048 Byte. An acknowledgment is organized in 2× 1024 Byte segments. Since there is still space for 6 additional `AckPackets` in the `AckFrame`, each `AckPacket` is provided redundantly four times within an `AckFrame`. In order to handle completely lost `AckFrames`, each `AckFrame` is provided with a redundancy of two frames.

The analysis is performed in two steps: First, the soft real-time time requirements per input and stage are established. Second, the performance characteristics are measured.

4.3.1 Soft Real-Time Requirements

The soft real-time requirements state how many messages a particular stage has to process given a desired data rate for the transmission. The requirements can be calculated by following the processing engine's dataflow and depends on the implementation of the stages, the desired data rate, the configuration of the protocol, and the expected channel conditions.

[6] Since no real 100 Gbit/s wireless communication interface exists yet, 10 GbE interfaces are used as a replacement.

Technical Details 2: Bit Error Rate and Packet Loss

The Bit Error Rate (BER) states the percentage of erroneous bits in a transmission. In order to estimate the impact of an equal distribution of single bit-flips on a transmission, the BER can to be used to calculate the amount of lost data. This is done here by transforming the BER into a packet loss probability by multiplying the packet size in bit with the BER.

$$P_{loss} = Packet_{size} * BER$$

P_{loss} states the probability a packet is lost. This can be used to calculate the actual expected packet loss. The initial transmission is round 0.

$$Packets_{send}^{Round0} = Packets_{initial}$$

The lost packets of the initial transmission have to be retransmitted, i.e., the packet loss probability has to be applied to the retransmissions:

$$Packets_{send}^{Round1} = Packets_{send}^{Round0} * P_{loss}$$

This leads to the geometric series:

$$Packets_{all} = \sum_{Round=0}^{\inf} Packets_{initial} * {P_{loss}}^{Round}$$

Which converges to:

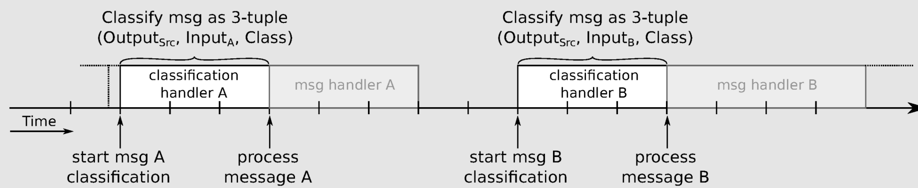
$$Packets_{all} = \frac{Packets_{initial}}{1 - P_{loss}}$$

In which $Packets_{all}$ is the overall number of transmitted packets. The number of bytes transmitted can now be calculated by multiplying the number of lost packets with the packet's size. Finally, after adding the protocol meta data bytes, the overall data that has to be transmitted is known, given the BER and protocol configuration.

The estimation of the soft real-time requirements is basically counting the number of messages each stage has to process. However, a message can have different runtime requirements depending on the stage's state and message's content. For example, the effort for processing incoming acknowledgments depends on the number of data-packets that have to be retransmitted. In order to be able to analyze the different runtime requirements, the soft real-time requirements are further refined by a classification of messages (see Technical Details 3). The classification depends on the stage's state and

Technical Details 3: Classification of Messages

The classification of messages is used to identify the actual input data for the stages during a transmission, as it is necessary for an accurate performance analysis (e.g. Worst-Case-Execution-Time (WCET) analysis). The stage implementation is instrumented with a classification handler that is transparently called after receiving the message and before the message is actually processed. Since the classification handler can be automatically removed it does not induce any overhead in the final implementation.



The classification handler has full read access to the message's content, as well the stage's state, but is not allowed to make any changes to the stage or the message. The classification itself is implemented by the protocol designer and depends on the behavior of the stage. The classification results in a 3-tuple $(Output_{Src}, Input_{Dest}, Class)$.

The number of 3-tuples for the same destination input and classification, i.e., $(*, Input_{Dest}, Class)$, also determines the soft real-time requirements of an input.

the message's content. The estimation of the message-classes can be done by analyzing the protocol's dataflow, or by analyzing the EFSM of the stages.

While it is possible to manually calculate the requirements by following the protocol's dataflow, it can become cumbersome depending on the complexity of the protocol implementation, the number of message classes, and the channel conditions. Instead of calculating the requirements manually, the soft real-time requirements are estimated in this thesis by simulating a transmission with the expected channel conditions. The channel conditions are modeled by a channel abstraction that replaces the actual communication interfaces. During the simulation, the protocol implementation is instrumented for counting the number of messages per stage, input, and message class automatically.

Figure 4.8 shows the soft real-time requirements according to the identified message classes of the `DG`, `DA` and `AP` of the processing engine subset. Since the `DG` is a straightforward

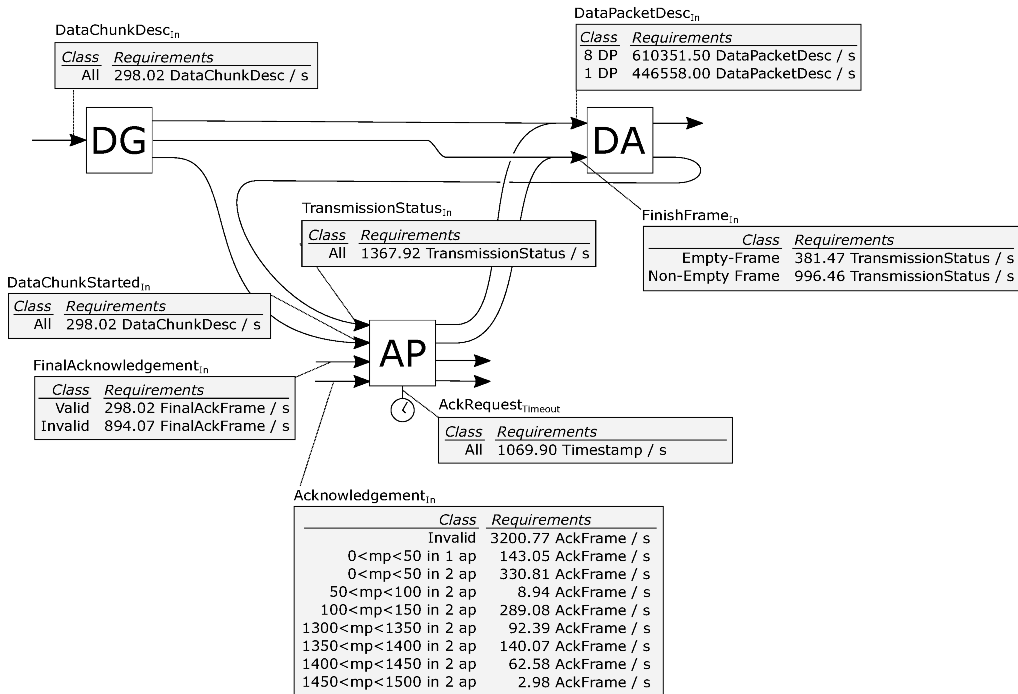


Figure 4.8: The results of soft real-time requirement estimation. The processing engine was used in a simulated transmission with a BER of 10^{-5}.

stage that always reacts the same way to any incoming message, the classification of incoming messages was not necessary. The message classes of the `DA` and the `AP` are explained in the following.

Data-Packet Aggregator (DA)

The `DA` has to process `TransmissionStatus (TS)` messages received by its `FinishFrame`$_{\texttt{In}}$ input and `DataPacketDesc` messages received at its `DataPacketDesc`$_{\texttt{In}}$ input.

The `DataPacketDesc`$_{\texttt{In}}$ input's task depends on the number of consecutive packets that the received `DataPacketDesc` message refers to. Consequently, the number of data packets that have to be aggregated is used as the classification property. For example: When the incoming message refers to 8 consecutive data-packets, the message is classified as `8 DP`.

During the simulation, two classes of DataPacketDesc messages are counted, 8 DP and 1 DP. The 8 DP messages were received from the DG, and the 1 DP messages were sent by the AP.

The FinishFrame$_{\text{In}}$ input is used to trigger the finalization and emission of a data-frame independent of the number of currently aggregated data-packets. Messages received by the FinishFrame$_{\text{Input}}$ input are classified given the current state of the DA. In case the current frame is empty, the TransmissionStatus message is just forwarded, and no data-frame is emitted. In this case, the message is classified as Empty Frame. In the case the current frame has already aggregated at least one data-packet, the data-frame is sent, a new empty frame is allocated, and the TransmissionStatus message is forwarded. Therefore, the message is classified as Non-Empty Frame.

Acknowledgement Processor (AP)

The AP has 4 inputs and one timeout: The Acknowledgement$_{\text{In}}$ input receives Ack-Frames that state which data-packets were not transmitted correctly. The Final-Acknowledgement$_{\text{In}}$ input receives FinalAckFrames that state that the current datachunk was completely transmitted. The DataChunkStarted$_{\text{In}}$ input is used to reset the AP and store the new DataChunkDesc. The TransmissionStatus$_{\text{In}}$ input is used to reset the AcknowledgementRequest$_{\text{Timeout}}$ and finally, the AcknowledgementRequest$_{\text{Timeout}}$ is used to transmit an AckRequestFrame.

The messages received at the FinalAcknowledgement$_{\text{In}}$ input are classified as Valid and Invalid. Invalid final acknowledgments are outdated, i.e., either their sequence number was already received, or the final acknowledgment does not belong to the current datachunk.

The Acknowledgement$_{\text{In}}$ input receives AckFrames that state the data-packets that have to be retransmitted. The acknowledgment frames are classified as Invalid when the acknowledgment frame is outdated or when the AckFrame does not belong to the current datachunk. Valid AckFrames are classified by the number of missing data packets for the current datachunk and the number of ack-segments[7].

[7] For example, an acknowledgment that states 1325 missing packets (mp) in two ack-packets (ap) is classifies as "1300 < mp < 1350 / 2 ap"

The $\texttt{Acknowledgement}_{\texttt{In}}$ input and the $\texttt{FinalAcknowledgement}_{\texttt{In}}$ input show a very high amount of Invalid messages. The reason is that the (Final-)AckFrame messages are sent redundantly because the timely receiving of acknowledgments, acknowledgment-requests, and final acknowledgments is paramount in order to achieve a stable throughput because in case they are lost, they stall the transmission. However, only the first (final) acknowledgment is considered valid. The remaining are invalid due to their outdated sequence number.

Finally, the $\texttt{AP.DataChunkStarted}_{\texttt{In}}$ and the $\texttt{AP.TransmissionStatus}_{\texttt{In}}$ inputs always react the same and no further classification is needed.

4.3.2 Performance Characteristics

The calculation of the performance characteristics is carried out on the Mellanox TileGx72 manycore board [86]. The manycore board is equipped with 72×1 GHz general-purpose cores and 4× 100Gbit/s memory controllers. Additionally, the TileGx72 provides 8× 10 GbE interfaces, and a PCIe 3.0 interface.

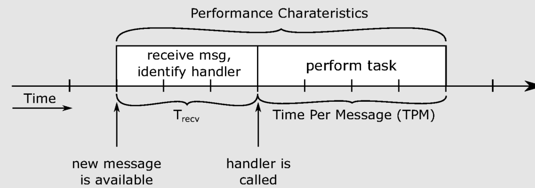
The memory controllers are configured to transparently stripe the memory and provide a virtual 400 Gbit/s memory interface for the applications. This distributes the memory load, and it should reduce Non-Uniform Memory Access (NUMA) effects that would complicate the analysis. While the latency per memory access may be increased, the combined capacity of the memory-controllers allows for more straightforward implementation. The protocol processing framework is built upon a Zero-Overhead Linux that allows to disable the timer-interrupt and prevent preemptive scheduling, giving the protocol processing framework exclusive access to the computation hardware, while providing a convenient programming environment.

The measurement of the performance characteristics starts with the measurements regarding the message-passing subsystem, followed by the stage-specific benchmarks for the stage.

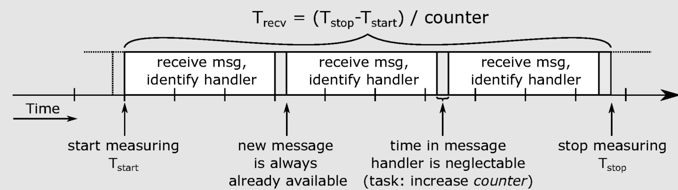
In the presented graphs, the outliers were removed by focusing on the 95 percentiles of the measurement in order to increase the readability. Since the outliers stem mostly from cold caches that do not reflect the situation during a transmission, they can be ignored.

Technical Details 4: Receiving and Scheduling Costs of a Message

The performance characteristics of a stage are the sum of the receive- and the scheduling costs (T_{recv}) of a message and the Time-Per-Message (TPM) needed for a message to be processed.



The receive- and scheduling-costs are measured with a benchmark that saturates a processor with messages, i.e., the processor never has to wait for a new message. The handler for these messages does not perform any work but counts the incoming messages. Therefore, the number of received and processed messages can be used to estimate the receive- and scheduling-costs T_{recv} for a message.



Message Passing

The performance characteristics estimation starts with measuring the receive-cost per message (see Technical Details 4). In order to estimate the receive-costs, a join-benchmark is used, i.e., two (or more) message producers send their messages to a single consumer until the consumer is fully saturated. Since the per-message receive- and scheduling costs (T_{recv}) are dependent on the message's size, the receive-costs have to be measured for all message sizes used by any stage in the processing engine. The implementation of the data link protocol uses messages of sizes up to 72 Bytes. Table 4.2 shows the receive-costs per message and the message-size (two producers were able to saturate the consumer fully during the benchmark).

Size in Byte	T_{recv}(size) in ns	Size in Byte	T_{recv}(size) in ns
8	73.35	48	84.69
16	79.55	56	84.46
24	77.47	64	92.45
32	80.17	72	91.17
40	85.34		

Table 4.2: Receive and scheduling costs for a message depending on its size. Two message producers are necessary to fully utilize a message consumer.

DataPacketAggregator (DA)

The `Data-Packet Aggregator (DA)` shall serve as an example[8] for the performance characteristic measurement. The `DA` is configured with a data-packet payload size of 1024 Bytes and a `DataFrame` that can aggregate eight data-packets. Furthermore, the `DA` can use a network-frame buffer pool for each 10 GbE interface.

The `DA` has two inputs, which both have two message classes (MC). The $\texttt{DataPacketDesc}_{\texttt{In}}$ ($\texttt{DPD}_{\texttt{In}}$) input has to process `DataPacketDescs` that contain 8 data-packets (MC: 8 DPs), as well as messages that contain 1 data-packet (MC: 1 DP).

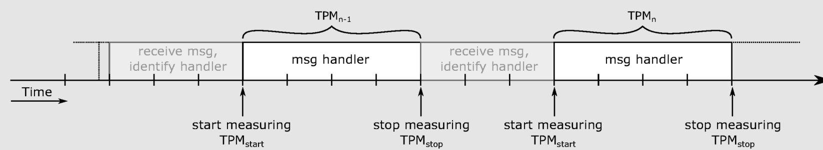
The $\texttt{FinishFrame}_{\texttt{In}}$ input has to process `TransmissionStatus` messages of two classes: The `Empty Frame` message-class states that the `DA`'s current frame is still empty, and the `Non-Empty Frame` message-class states there is already a data-packet aggregated in the current frame.

The performance characteristics estimation generally consists of two different measurements: Firstly, the TPM is measured depending on the mapping of the stage in order to estimate the sensitivity of the stage on the mapping. Secondly, the established best mapping and worst mapping is used to parallelize the stage in order to measure the TPM in a contention situation.

[8] The benchmarks for the remaining stages are omitted here for the sake of readability but are provided in annex B.

Technical Details 5: Time-Per-Message

The main part of the performance characteristics is the Time-Per-Message (TPM) that states how long it takes a stage to process a message given that the stage has exclusive access to the hardware. The measurement itself is done in the framework's backend by measuring the time spent in the handler-function of the measured input. This way, all implicitly used resources are measured, except for the message-passing subsystem, which was measured separately. Measuring the TPM isolated from the message-passing subsystem, minimizes the influence of the relative mapping of the message producer and the message receivers during the benchmark.



The TPM is measured in two steps. Since a stage can be sensitive to the mapping, e.g., distance to memory controllers, the stage's sensitivity to its mapping is measured first. Secondly, in order to investigate possible contention effects, the TPM is measured with increasing parallelization.

$$\mathrm{TPM}_{s,e} = \frac{\sum_{i=s}^{e} TPM_i}{e - n}$$

Due to the possibly small TPMs, the measurement may impose a significant overhead. In order to reduce the induced measurement overhead, a single measurement run is divided into snapshots and only the average TPMs for these snapshots is reported. Furthermore, instead of using the WCET, the median of these averages is then used for the analysis. This is possible due to the soft real-time character of communication protocol processing, i.e., missing a deadline does not lead to a failure, and it avoids too pessimistic performance estimations.

Stage Sensitivity on the Mapping

The sensitivity of the stage to its mapping is established by measuring the (mapping) Time-Per-Message (TPM) (see Technical Details 5) for the dominant input and using the message class that produces the highest processing costs. The dominant input is usually the input that is responsible for the primary processing-costs[9] during an actual

[9]In the case an educated guess is not possible, all inputs have to measured.

transmission. In the case of the DA, this is the $\texttt{DPD}_{\texttt{In}}$ input in combination with the message class 8 DPs.

The measurement itself is done individually on all processors, i.e., the producer stage first measures the TPM for the CPU 1, then for CPU 2, and so on. The mapping of the producer stage and the sender stage is shown in figure 4.10. CPU 0 is used for the initialization of the manycore (I), CPU 1 is used to set up the benchmark (B), the message producer (P) is mapped to processor 2, and the consumer is mapped to processor 71.

The benchmarking starts by preparing the stage for the measurements and deciding about the content of the messages. Since the message class 8 DPs requires the DA to be in the initial *Empty Frame* state, the measurement of the DA's $\texttt{DPD}_{\texttt{In}}$ input does not need a particular setup.

However, the messages used for the benchmarking have to be constructed carefully. First, in order to measure the TPM for the 8 DPs message class, the DataPacketDesc message has to be configured with 8 consecutive packets. Additionally, the DataPacketDesc messages have to refer to an actual memory address because the DA will read the packet's payload from the specified memory address. Furthermore, the TPM for the $\texttt{DPD}_{\texttt{In}}$ input is higher for payload that has to be fetched from memory than for payload that is available in the cache.

During the transmission phase, any individual data-packet's payload will most probably not be available in the processor cache. Therefore, it has to be avoided that the DA can read the desired payload directly from the cache, which would not reflect the actual runtime-behavior. In order to avoid that the DA can fetch payload-data from the cache, the message producer simulates the characteristics of a transmission, i.e., the DataPacketDesc message's payload points into an actual datachunk buffer and the message producer emulates the separation of that datachunk. Therefore, it provides DataPacketDesc messages with different payload pointers that do not repeat during a single benchmark.

The TPM depending on the mapping of the DA is shown in figure 4.9. The median TPM is in a range of min. $10878ns$ to max. $11305ns$, i.e., a span of $427ns$. Consequently, the DA is slightly dependent on the mapping. Furthermore, the results also show an interquartile range between $125ns$ and $379ns$, i.e., the measurement results vary between repetitions, even on the same Central Processing Unit (CPU).

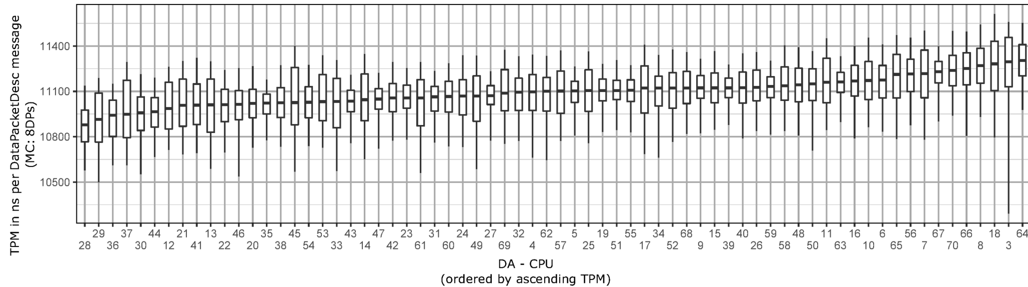


Figure 4.9: TPM in *ns* per `DataPacketDesc` message with the message class 8 DPs depending on the mapping of the `DAs`.

Figure 4.10 shows the median TPM depending on the CPU's position in the CPU-grid as a heat-map. One can see that the lowest TPMs are measured in the "middle" of the manycore. Consequently, `DA`s benefit when the distance to the four memory-controllers is similar. Therefore, memory-bound stages should be mapped to cores in the center of the CPU grid first.

Contention Measurements

The mapping-TPM is not sufficient for the analysis because unnoticed contention effects caused by the parallel execution can lead to wrong adaptations. The contention effects can be estimated when the investigated stage is measured parallel on several cores. Since the `DA` is sensitive to its mapping, the parallel performance is analyzed individually for their best-/worst-case mapping. The benchmarks are carried out for all inputs and all message classes, not only the dominant ones because all inputs and all identified message-classes increase the processor utilization.

`DataPacketDesc` input ($\texttt{DPD}_{\texttt{In}}$)

The contention measurements for the $\texttt{DPD}_{\texttt{In}}$ have to be carried out for the two message classes 8 DP and 1 DPs. Figure 4.11 shows the TPMs for the message class `DPD (8 DPs)`, given the best-case mapping (left) and the worst-case mapping (right). The impact of the mapping onto the parallel TPM can be seen only for small parallelization counts up to five parallel `DAs`. For higher parallelization counts, the impact can be neglected, as

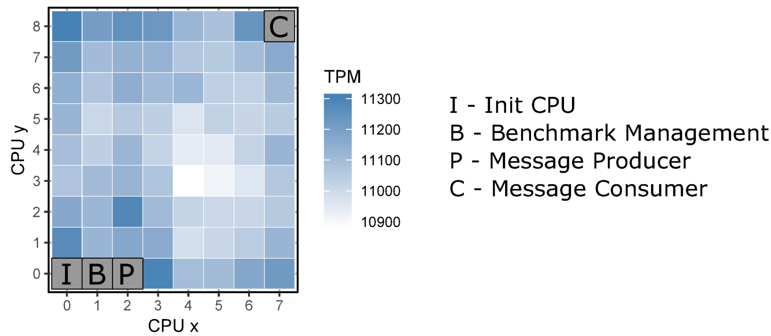


Figure 4.10: Heat-map of the TPM in *ns* per `DataPacketDesc` message with the message class 8 DPs showing the effect of the memory striping. The lowest TPMs were measured in the case the distance to the memory controllers is similar, i.e., in the middle of the CPU grid.

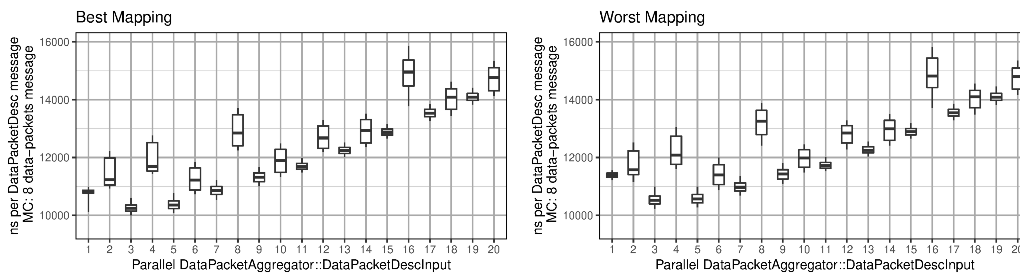


Figure 4.11: Performance Characteristics of the $\texttt{DataPacketDesc}_{\texttt{In}}$ input of the `DA` for the message class `DataPacketDesc (8 data-packets)`, depending on the mapping.

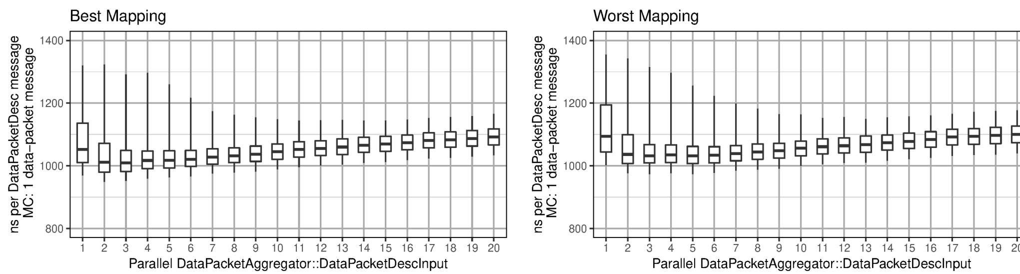


Figure 4.12: Performance Characteristics of the $\texttt{DataPacketDesc}_{\texttt{In}}$ input of the `DA` for the message class `DataPacketDesc (1 data-packet)`, depending on the mapping.

the results for best- and worst-case mapping are nearly identical. However, the impact of the contention due to parallel execution can be clearly seen, as the TPM increases for the best-case mapping from around 10241ns for a single DA to 14766ns for 20 parallel DA (worst-case: 10519ns for one DA, 14821ns for 20 DAs). Furthermore, the TPM does not increase monotonously but shows spikes for even numbers of parallel DA, especially for powers of two. For example, the highest median TPM (14955ns) was measured for 16 parallel DAs.

That behaviour is most probably caused by data cache contention due to evicting (and immediately re-fetching) cache lines that were already prefetched by one core because another CPU prefetches its own payload to the same cache lines. This occurs because the data-packets are sent to the DAs with increasing addresses. The effect can be theoretically avoided by using non-coherent memory for the datachunk buffer (see annex B.2.6). However, since the Direct Memory Access (DMA) mechanism of the Tilera Gx72 boards relies on coherent memory, the buffer memory would have to be copied from the coherent DMA buffer to the incoherent datachunk buffer first. This would negate the positive effect of using incoherent memory.

The contention benchmark for the message class DPD (1 DP) is realized with a random order of data-packets. The random order emulates an actual transmission because DPD (1 DP) messages are only sent by the AP in order to retransmit (randomly) lost data-packets. Figure 4.12 shows the TPMs for the message class DPD (1 DP) given the best-case mapping (left) and the worst-case mapping (right). The cache contention effect, i.e., the spikes, is not seen for the message class DPD (1 DPs). The reason is that the data-packets (and therefore the payload) handled by the DAs follow a random order, i.e., the probability that parallel DAs prefetch data to the same cache lines is lower compared to sequentially distributed data-packets.

FinishFrame input (FF_{In})

The FF_{In} input has two message classes: Empty Frame and Non-Empty Frame. The measurement for the Empty Frame message class is straight forward, as no special setup is necessary. When receiving a TransmissionStatus message while the current frame is still empty, the DA's task is only to forward the received message via its TransmissionStatus

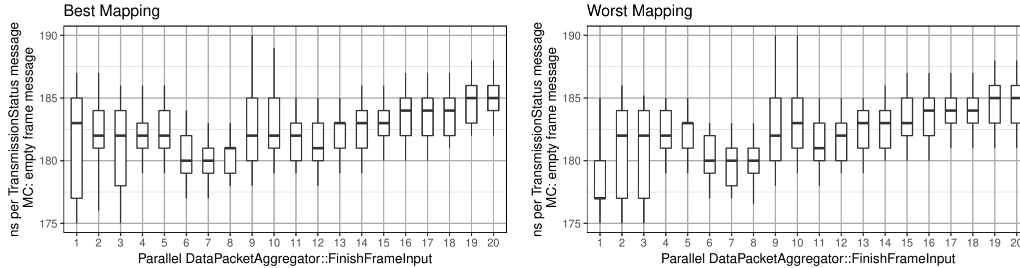


Figure 4.13: TPM of the `FinishFrame` input of the `Data-Packet Aggregator (DA)` for the message class `TransmissionStatus (Empty Frame)` (Best case mapping right and worst case mapping left).

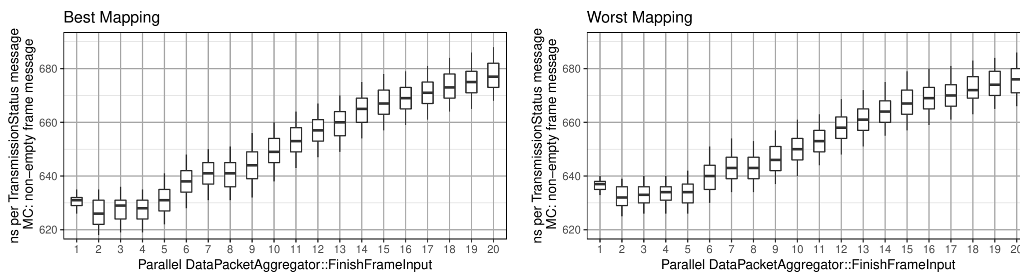


Figure 4.14: TPM of the `FinishFrame` input of the `Data-Packet Aggregator (DA)` for the message class `TransmissionStatus (Non-Empty Frame)` (Best case mapping right and worst case mapping left).

output, i.e., no real work has to be done. Therefore, the processing time is expected to be low, without contention effects, and without an impact of the mapping. This is reflected in the results shown in figure 4.13. The median TPMs show no sign of contention, as they are in a range of 180 ns to 185 ns for the best mapping and 177 ns to 185 ns for the worst mapping.

In order to measure the TPM for Non-Empty Frame messages at the $\mathtt{FinishFrame_{In}}$ input, the `DA` has to be in the *Non-Empty Frame* state. This is done by sending a valid `DataPacketDesc` to the $\mathtt{DPD_{In}}$ before each measurement, so that the current frame is not empty anymore. The `DA` now performs three tasks: First, it sends the non-empty frame via the $\mathtt{Frame_{Out}}$ output. Second, it acquires a new network frame. Third, it forwards the received transmission status via the `TransmissionStatus` output. The results of the measurement are shown in the figure 4.14.

The TPM for the Non-Empty Frame message class stays almost constant for less than five parallel DAs and then increases moderately for higher numbers of parallel DAs. The moderate contention effect stems from competition for the network buffer pools, as each of the DAs tries to allocate new empty frames.

4.3.3 Summary

This section presented the approach for estimating the soft real-time requirements and the measurement of the performance characteristics for a given processing engine. It furthermore showed how message classes are used to increase the accuracy of the general analysis. The different TPMs of the DA's $\texttt{DataPacketDesc}_{\texttt{In}}$ input for the message classes 8 DP and 1 DPs showed that ignoring message classes may lead to unreliable benchmark results.

Furthermore, the benchmarks helped to increase the efficiency of the implementation and understanding of the target hardware. In the case of the DA, the use of cache-incoherent datachunk buffers would increase the efficiency as cache invalidation could be controlled fine-grained. However, due to the DMA subsystem of the Tilera manycore board, cache-coherent memory has to be used. The benchmarks also helped to debug the implementation of the stages, as the benchmarks rely on the correct implementation of the interface and the stage's behavior. In the case the implementation does not reflect the desired stage-behavior, the benchmark will fail.

The analysis's results are used in the following to estimate the necessary parallelization of the processing engine and its adaptation.

4.4 Adaptation of the Processing Engine

The soft real-time analysis provides the protocol designer with insights into processing requirements and capacities for the whole processing engine. These results are now used for the adaptation of the processing engine. The adaptation's goal is to reduce the soft real-time requirements of all stages by parallelization until the processing capacities of the hardware are sufficient to process the resulting data rate. The soft real-time requirements and the performance characteristics are used to calculate the *processor utilization*, which is used to estimate the necessary parallelization count of the inputs (see Technical Details 6).

Technical Details 6: Calculating the Processor Utilization

The soft real-time requirements and the performance characteristics are used for identifying processing bottlenecks in the processing engine by determining the necessary adaptation ratios with the equations 4.1 and 4.2.

The functions multiply and sum up the soft-real-time requirements $\mathrm{R}_{\mathrm{Input}}^{\mathrm{Class}}$ and the performance characteristics $\mathrm{P}_{\mathrm{Input}}^{\mathrm{Class}}$ for all inputs and all message-classes of a stage. The performance characteristics P are the sum of the TPM and the message receive costs T_{recv}.

$$Dup_{Input}(\#CPU) = \sum_{m=0}^{\#Class} R_{Input}^{Class_m} \times P_{Input}^{Class_m}(\#CPU) \tag{4.1}$$

$$Split_{Input}(\#CPU) = \frac{1}{\#CPU} \times \sum_{m=0}^{\#Class} R_{Input}^{Class_m} \times P_{Input}^{Class_m}(\#CPU) \tag{4.2}$$

The result is the estimated processor utilization of that input given a parallelization count $\#CPU$, the soft real-time requirements, and the performance characteristics when all input streams of $Input$ are split or duplicated. Consequently, the equations calculate the processor utilization for the input, including contention effects.

In the case $Dup_{Input}(\#CPU) <= 1$, the input streams can be duplicated. In the case that $Dup_{Input}(\#CPU) > 1$, the processor utilization is too high and has to be reduced by a stream-split or changing the implementation.

In the case that $Split_{Input}(\#CPU) < \#CPU$, the desired data rate can be processed by the input when a stream-split with $\#CPU$ is used. In the case that $Split_Input(\#CPU) > \#CPU$, the streams have to be split more $\#CPU$ times.

The adaption depends on the processor-utilization, which takes the results of the soft real-time analysis and the performance measurements into account. The processor-utilization states how often a stream that is connected to an input has to be split. Figure 4.15a shows the processor-utilization for the two inputs of the `Data-Packet Aggregator (DA)`. The `DA`'s `DataPacketDesc`$_{\texttt{In}}$ input has to be parallelized nine times in order to reduce its processor utilization to 0.8284. The second input shows a processor-utilization of 0.0008 for 1 CPU, i.e., it is most probably neglectable.

This information is used in figure 4.15b for the adaptation of the `DA`, which is carried out with the Stream-Split, Stream-Duplicate, and Stream-Join operators. What operator

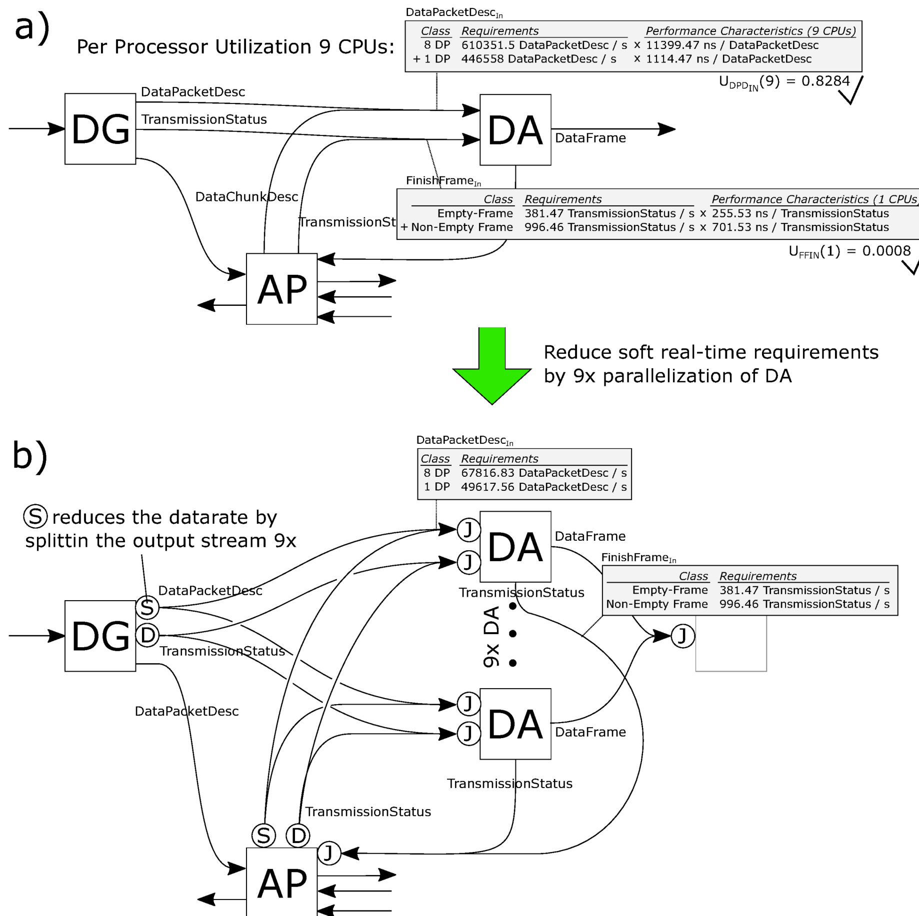


Figure 4.15: The analysis focuses on the processing engine subset Data-Packet Generator (DG), Data-Packet Aggregator (DA), and Acknowledgement - Processor (AP).

can be used depends on the semantics of the streams. In the case of the DA, the input streams of the $\texttt{DataPacketDesc}_{\texttt{In}}$ input can be split, whereas the input streams of the $\texttt{FinishFrame}_{\texttt{In}}$ input have to be duplicated.

The stream-split operator can be used for the $\texttt{DataPacketDesc}_{\texttt{In}}$ because the DG and AP only expect that data-packets, which were sent over that stream, are aggregated into a frame. Which DA processed a data-packet and the order of data-packets does not matter. Consequently, a stream-split can be used to distribute the DataPacketDesc messages in order to reduce the soft real-time requirements. By splitting the DataPacketDesc streams from the DG and AP, the soft real-time requirements of the $\texttt{DA::DataPacketDesc}_{\texttt{In}}$ input are reduced by a factor of nine.

In contrast, the TransmissionStatus messages, sent by DG and AP, are directed to all DAs, because the DG and the AP expect that all DAs receive the TransmissionStatus message in order to finish and send their current DataFrame. Consequently, the TransmissionStatus stream has to be duplicated. By duplicating the TransmissionStatus streams, the data rate was not reduced, i.e., the soft real-time requirements of the $\texttt{TransmissionStatus}_{\texttt{In}}$ input are unaffected by the stream-operator.

Stage	#CPUs	Per-Processor Utilization in %	
Sender			
Data-Packet Generator (DG)	1	6.16	✓
Data-Packet Aggregator (DA)	9	82.92	✓
Acknowledgement Processor (AP)	1	7.86	✓
PCIe (Sender)	**2**	**147.00**	⚡
Both			
Communication (COM)	5	88.00	✓
Receiver			
Data-Packet Combiner (DC)	5	94.33	✓
Acknowledgement Generator (AG)	1	38.11	✓
PCIe (Receiver)	2	70.00	✓

Table 4.3: Predicted per-processor-utilization and parallelization ratios of the stages for a desired data rate of 40 Gbit/s and a BER of 10^{-5}.

The processor utilization of the DA ($U_{\texttt{DA}}$) after the adaptation (see figure 4.15b) is:

$$U_{\texttt{DA}} = Split_{\texttt{DPD}_{\texttt{In}}}(9) + Dup_{\texttt{FF}_{\texttt{In}}}(9) = 0.8284 + 0.0008 = 0.8292$$

That is, the ninefold parallelization of the $\texttt{DataPacketDesc}_{\texttt{In}}$ input and the duplication of the $\texttt{FinishFrame}_{\texttt{In}}$ input lead to an utilization per processor 83%. Consequently, the adaptation was successful. However, sometimes the processor utilization of a stream that has to be duplicated is too high. In this case, the split has to be carried out earlier in the protocol pipeline.

All remaining stages are handled in the same way, depending on their processor-utilization, as shown in table 4.3. However, when the adaptation procedure reached the PCI Expresss (PCIes) interfaces, the analysis revealed that 40 Gbit/s are not achievable with the given system, because the sender-side PCIe interface is not able to provide the desired data rate ($U_{\texttt{PCIe (Sender)}}(2) = 1.47$). The reason is that the sender host is equipped with a PCIe 2.1 interface, which can reach a maximum theoretical throughput of 32 Gbit/s [87]. The bottleneck is caused by the datachunk transfer between host and embedded manycore board. Therefore, the actual transfer of the payload is omitted in the following benchmarks. An example that allows reaching the 40 Gbit/s by parallelizing the protocol processing over several devices follows later in this chapter.

Adaptation and Scheduling

The assigned stream operators define the parallelization, as well as the distribution of messages. Since messages have to be processed in the streamed order, the scheduling of message can be done locally per CPU with a simple FIFO scheduler. In order to avoid message drops caused by short term over-utilization of CPUs, a back pressure flow control mechanism is used to automatically limit the data rate in such a case.

4.5 Mapping of the Processing Engine

The last step of the protocol implementation is the mapping of the processing engine onto the communication system. The processing engine will usually be mapped on several devices, such as the host, an embedded manycore, and an additional external

Figure 4.16: A possible mapping of the processing engine to the processing hardware. The FEC/CRC stages are offloaded to external FPGAs.

accelerator. The choice of the employed devices depends on the specifics of the offloaded stages. The higher-level protocol processing, such as retransmission management and data manipulation, is best suited to be offloaded to a programmable embedded manycore due to its flexibility. An embedded manycore employs the stream processing framework to execute the offloaded processing engines and connect itself transparently to the host's stages via the available communication channels, such as PCIe. Technical details about the offloading of the protocol processing into the embedded manycore and the transparent integration of the host can be found in annex A.2.

On the contrary, some stages, such as CRC/FEC calculations, are best processed in specialized hardware and will be further offloaded, e.g., on special-purpose FPGAs or Application-Specific Integrated Circuits (ASICs). Such special-purpose hardware will not implement a stream processing framework but can be seen as a particular parallelized remote stage. In order to employ such hardware, a marshaling stage is needed to communicate with the device, i.e., exchange data and control information. The mapping of stages and the communication system is shown in figure 4.16.

In the first step, the stages are assigned to their processing hardware. The communication system consists of three parts: The host, that should only provide and receive the payload, an embedded manycore for the high-level protocol processing, and an external FPGA that acts as a special-purpose accelerator for the FEC/CRC calculation.

After the coarse grained distribution of the processing engine over the processing hardware, the stages that are assigned to the manycore have to be further mapped to CPUs. However,

finding such an optimal mapping is a research topic on its own, and was not investigated in this thesis, but a possible approach is outlined in the following.

After offloading the `FEC/CRC` stages, the remaining stages are mapped to dedicated CPUs on the embedded manycore. All stages that abstract hardware resources, such as `COM` or `PCIe` interfaces, are assigned to a CPU where they can access the hardware the fastest. The protocol processing stages are mapped to CPUs with the help of the mapping sensitivity, measured during the performance analysis, i.e., the mapping starts with assigning CPUs to stages according to their best mapping. Without any claim to generality, the stages with the highest parallelization count are mapped to a CPU first. Furthermore, the mapping of stages to CPUs follows the same considerations as in other high-performance applications, such as NUMA-awareness [88].

4.6 Testing of the Implementation and Latency Hiding

The final step in the protocol design process is testing the implementation. For the testing and the evaluation, the following restrictions were applied due to shortcomings of the evaluation hardware.

1. The offloaded FEC stages were not used for the majority of the evaluation. Instead, the reduction of the BER that would result from the integration of the `FEC/CRC` stages was implicitly assumed. This measure was taken due to the lack of hardware. Besides, the FEC/CRC calculation adds a small overhead that does not influence the conclusions of this thesis. However, an evaluation in which two external FPGAs are integrated in communication system is provided at the end of this section.

2. The wireless communication technology was replaced by 8×10 GbE interfaces and the expected BER was simulated. This measure was taken due to the lack of 100 Gbit/s wireless transmission technology.

3. The transport of payload from the host to the embedded manycore and vice-versa is omitted. Unfortunately, one of the hosts of the communication system provides only a PCIe 2.1 interface, which limits the throughput to around 32 Gbit/s. A benchmark that actually transmits the payload end-to-end is provided later in this chapter.

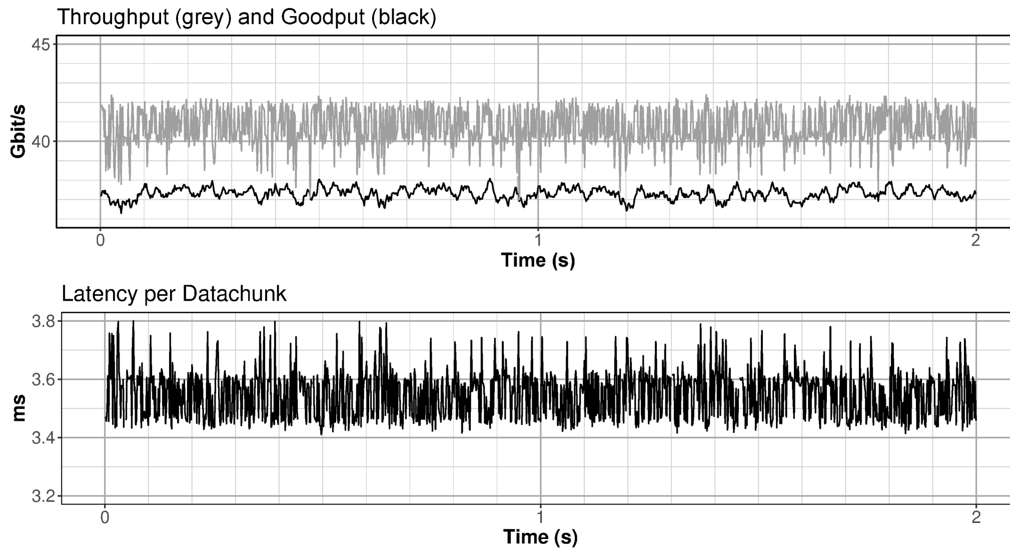


Figure 4.17: Goodput, throughput, and latency of the adapted processing engine, given a BER of 10^{-5}. The grey line shows the throughput, i.e., the complete channel utilization with transmission and retransmission, the black line shows the goodput per datachunk. The average goodput per transmission is 37.29 Gbit/s.

Figure 4.17 shows the achieved goodput and throughput of the processing engine given a BER of 10^{-5}. As one can see, the goodput, i.e., net- or application-level throughput, does not actually reach the desired 40 Gbit/s.

The lower-than-expected goodput is caused by the protocol itself. Since the protocol separates the transmission-phase and the retransmission-phase as described earlier, there exists a processing gap after all packets were transmitted. In that gap, the sender waits for the requested acknowledgment (see figure 4.18a), which reduces the actual time in which packets can be transmitted.

The processing gap between two datachunks can be closed by overlapping the transmission of two datachunks, as shown in figure 4.18b. Instead of waiting for the acknowledgment, the transmission of the second datachunk is started after all sub-packets of the first datachunk are transmitted once.

The flow-control of datachunks to protocol pipelines is handled by a `Channel Manager (CM)` stage, as shown in figure 4.19. In the case a new datachunk should be transmitted, the `CM` selects a free pipeline and forwards the datachunk to that protocol pipeline

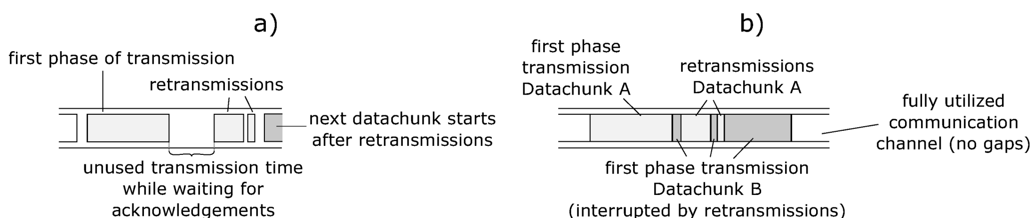


Figure 4.18: a) The singlechannel protocol monopolizes the communication-channel for transmission of single datachunk at a time. b) The multichannel protocol interleaves the communication-channel with the transmission of two datachunks simultaneously.

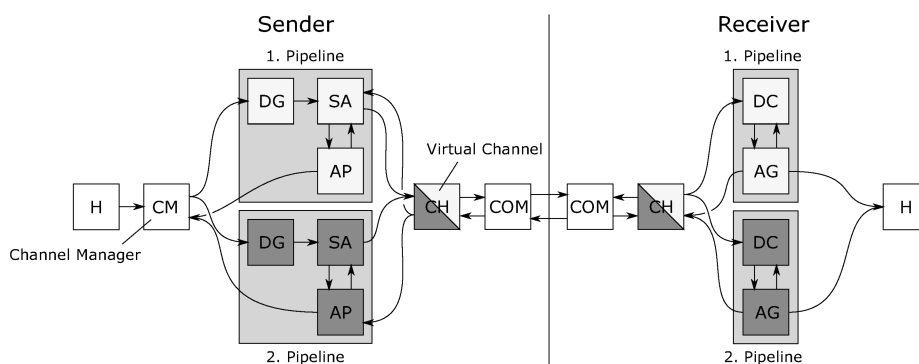


Figure 4.19: The multichannel protocol interleaves the communication channel with two parallel transmissions. The transmission of a datachunk on one virtual channel is interleaved with the retransmissions of the second virtual channel, therefore fully utilizing the communication channel's capacity.

for transmission. In the case that all pipelines are still transmitting a datachunk, the forwarding is suspended until one of the pipelines is ready. This approach multiplexes two transmissions on the same physical channel, which have to be separated. The separation is achieved by leveraging the virtual channels. The virtual channel has two tasks: Firstly, it adds the corresponding virtual channel number to every outgoing network-frame and secondly, it reads the virtual channel number of each incoming frame and forwards the network-frame to the correct protocol pipeline. This way several parallel transmissions can be multiplexed on the same communication channel without changes to the protocol.

Finally, the additional stages for the second protocol pipeline have to mapped onto the communication system, but the adaptation of the processing engine was conducted with the assumption that each stage utilizes its own CPU for the processing. However, the analysis estimated the processor utilization for 40 Gbit/s, whereas it does not matter whether

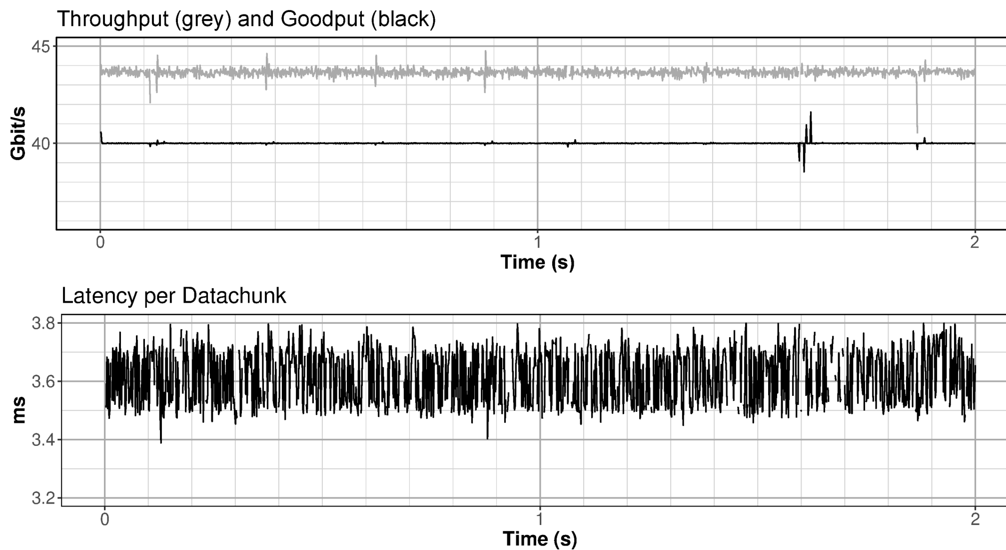


Figure 4.20: Goodput, throughput, and latency of the double-pipeline processing engine given a BER of 10^{-5}. The grey line shows the throughput,i.e., the complete channel utilization with transmission and retransmission, the black line shows the goodput per datachunk. The average goodput per transmission is 39.99 Gbit/s.

messages are processed by one or more stages on a processor, as long as the message load per processor does not increase. Therefore, the new stages do not increase the load on the processors and can be multiplexed on the processors used for the first pipeline.

Figure 4.20 shows the goodput and latency of the transmission after adding a second protocol pipeline. The throughput now reaches the desired 40 Gbit/s by fully utilizing the communication channel. Furthermore, the additional channel leads to a stable throughput, despite the packet loss. The spikes that can be seen every 200ms stem from overlapping retransmission phases, i.e., the channel is not completely used because the sender and receiver wait for acknowledgments. However, while pipeline parallelism increases the channel utilization, it also negatively affects the latency per datachunk negatively because two transmissions now compete for the physical channel.

The original protocol version which employs a single protocol pipeline is in the following called *singlechannel*, and the version that employs two parallel protocol pipelines will be called *multichannel*.

Stage	Predicted Utilization (%)	Measured Utilization (in %)	
		Singlechannel	Multichannel
$\mathtt{DG.DCD}_{\mathtt{In}}$	6.16	54.41	58.40
$\mathtt{DA.DPD}_{\mathtt{In}}$	82.85	76.32	81.98
$\mathtt{DA.FinishFrame}_{\mathtt{In}}$	0.01	0.07	0.09
$\mathtt{AP.AckFrame}_{\mathtt{In}}$	7.57	11.44	12.52
$\mathtt{AP.FinalAckFrame}_{\mathtt{In}}$	0.04	0.18	0.22
$\mathtt{AP.AckRequest}_{\mathtt{Timeout}}$	0.2	0.32	0.36
$\mathtt{AP.DCS}_{\mathtt{In}}$	0.00049	0.02	0.02
$\mathtt{AP.TS}_{\mathtt{In}}$	0.04	0.08	0.08

Table 4.4: Predicted processor utilization for a desired data rate of 40 Gbit/s and a bit error rate of 10^{-5}

Accuracy of the Analysis

The predicted and measured processor utilization for the 40 Gbit/s adaptation of the data link protocol is shown in table 4.4.

The $\mathtt{DataPacketGenerator.DataChunkDesc}_{\mathtt{In}}$ ($\mathtt{DG.DCD}_{\mathtt{In}}$) input and the $\mathtt{AckProcessor.AckFrame}_{\mathtt{In}}$ ($\mathtt{AP.AckFrame}_{\mathtt{In}}$) input show a far higher processor utilization than predicted. This is a result of the back pressure flow-control, which stalls the `DG`'s and `AP`'s processing in the case the receive-buffers of the `Data-Packet Aggregator (DA)` are full.

Furthermore, the effect of the transmission gap for the singlechannel version of the protocol can be seen, as the measured processor utilization of the DA is 5.66% lower for the singlechannel version, compared to the multichannel version. The lower processor utilization of the singlechannel protocol leads to a calculated goodput of 37.26 Gbit/s ($\frac{81.92\%}{77.32\%} = \frac{40Gbit/s}{XGbit/s}, X = 37.26Gbit/s$) compared to a measured singlechannel goodput of 37.29 Gbit/s.

4.6.1 Integration of external Accelerator Hardware

The FEC stages that were meant to be offloaded into external FPGAs have a vital role in wireless communication as they decrease the BER from an unusable BER of 10^{-3} to a manageable BER of 10^{-5}. The FPGAs for the FEC were experimentally integrated into the communication system via the ETH-Service (ETH) interfaces, which allowed to use existing drivers and thus reduced the implementation effort. The stream processing approach allowed to separate the FEC calculation from the rest of the protocol processing, i.e., FPGA, and the software implementation for the embedded manycore could be developed and tested completely independent after the message format was established. The forward error correction was developed by Dr.-Ing. Lukasz Lopacinski for his dissertation [84].

Figure 4.21 shows the goodput and latency depending on a simulated packet-loss for a prototype integration using one 10 GbE interface [7]. One can see how the integration of the FPGA slightly decreased the goodput and increased the latency, respectively. This is due to some lost frames between the embedded manycore and the FPGA, as well as the additional processing step. Some further optimization of the Ethernet implementations of the FPGA and embedded manycore would have reduced the losses, however, that was not the scope of this thesis. Furthermore, these results were achieved with multiple singlechannel pipelines [7] that were not able to fully overlap the transmission of datachunks.

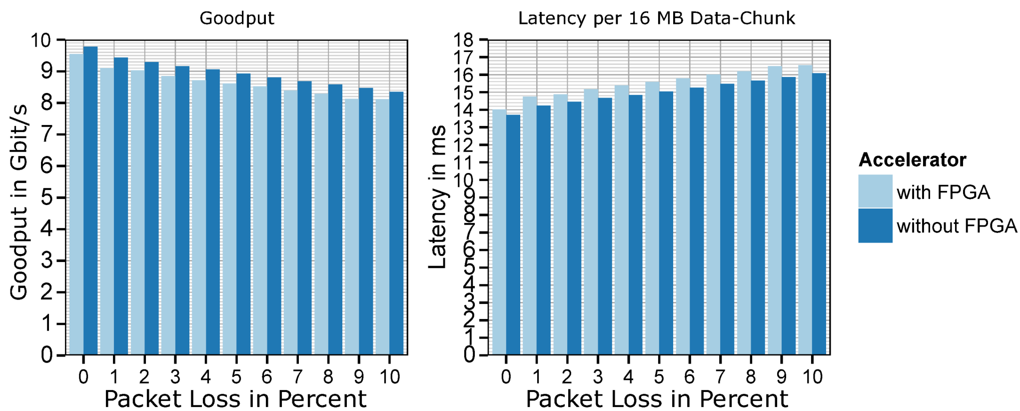


Figure 4.21: Goodput and latency of the communication system with and without attached FPGAs [7]. The stated packet-loss was simulated at the FPGA.

At the time of the evaluation, only two FPGAs were available. Therefore, only the goodput for one 10 GbE interface could be measured. However, due to the implementation as an individual and independent stage, the goodput scales linearly with the number of employed 10 GbE interfaces and FPGAs.

All remaining benchmarks presented in this thesis, are performed without FEC stages and the decreased BER that would result in using FPGAs is accepted as given.

4.6.2 Summary

This section showed how processing latencies, e.g., due to the acknowledgment mechanism, can reduce the actual goodput. This can lead to a situation in which processors are idle because the system is waiting for the correct transmission of the last missing packets that allows them to continue their work. However, during the soft real-time analysis, it was assumed that all stages are constantly processing messages. These processing-gaps can be hidden with parallel processing engines that are used in an overlapped manner. This is achieved by providing parallel protocol pipelines that share a channel by virtualization. In order to estimate whether an additional parallel processing engine is necessary, the achievable data rate of the adapted processing engine has to be measured.

The experimental integration of the external FPGAs, responsible for the FEC calculation showed that the integration via standard 10 GbE interfaces is feasible. Using standard hardware allowed for using existing drivers without additional implementation effort, i.e., tremendously reducing time and costs.

4.7 Evaluation

The final part of this chapter evaluates the approach with respect to overhead, flexibility, and scalability. The overhead of the processing approach is measured in subsection 4.7.1 by setting the BER to zero and measuring the goodput for different data-packet sizes and numbers of interfaces. The flexibility is evaluated with two real-world scenarios. The first scenario adapts the processing engine automatically to changing channel conditions, by dynamically selecting and combining the channels depending on the BER as well as the desired data rate. This is particularly useful for wireless communication as channel

conditions are known to change over time. In the second scenario, the user has different data rate requirements over the cause of the transmission. Finally, the scalability of the approach is shown by parallelizing the protocol processing over two manycore boards.

4.7.1 Processing Overhead

An interesting question is how much overhead introduces the processing approach depending on the desired data rate and the number of stages. Table 4.5 shows the static protocol overhead that stems from the protocol headers (without taking redundancy into account). In the worst case ($8 \times$ 1kB `DataPackets` per `DataFrame`), 1.074 % of the transmitted data is static protocol overhead. This can be reduced to 0.391% in the case only one 8kB `DataPacket` is used to fill the frame. The maximum possible Ethernet throughput for 9000 Byte jumbo frames is 9.97 Gbit/s. The difference of 0.03% to 10 Gbit/s stems from the 14 Byte MAC header, 5 Byte inter-frame-gap, 8 Byte preamble, and 4 Byte CRC. The protocol overhead and the maximum Ethernet throughput are used to calculate theoretical max achievable goodput, also shown in table 4.5.

Communication Interfaces	
Raw-Ethernet throughput	9.97 Gbit/s

ProtocolItem	Size	
`VirtualChannelHeader`	8 Byte	
`FrameHeader`	16 Byte	(without FEC redundancy)
`SubPacket`	8 Byte	

ProtocolItem	Protocol Overhead	Payload	Overhead in percent	Max. 10 GbE Goodput (Gbit/s)
`DataFrame (1kB)`	88 Byte	$8 \times 1kB$	1.074 %	9.863
`DataFrame (2kB)`	56 Byte	$4 \times 2kB$	0.684 %	9.902
`DataFrame (4kB)`	40 Byte	$2 \times 4kB$	0.488 %	9.921
`DataFrame (8kB)`	32 Byte	$1 \times 8kB$	0.391 %	9.931

Table 4.5: Protocol overhead of the data-frame for different payload-sizes and maximum raw Ethernet throughput.

.

Additional to the static protocol overhead, the processing itself imposes overhead that can reduce the goodput. The processing overhead was measured individually for all four packet sizes (1kB, 2kB, 4kB, 8kB), with 1 to 8 10 GbE interfaces and a BER of 0. Since the goal was to fully reach the theoretical data rates, the multichannel version of the protocol was used.

Table 4.6 shows the number of configured `DAs` and `DCs` (all other stages did not need to be parallelized), the measured goodput, the theoretical maximum throughput depending on the number of interfaces, as well as the overhead in Gbit/s and percent. As to be expected, the overhead increases in general with a higher desired data rate and larger processing engines, i.e., the parallelization has an impact on the protocol processing. However, the measured processing overhead stays consistently below 0.4% of the theoretically possible throughput, i.e., it is neglectable.

The measurements were carried out with the parallelization counts established during the analysis. In case the results were lower than expected, the predicted processor utilization was investigated.

For a payload size of 1024 Bytes, this was the case for 7×10 GbE interfaces. For 7×10 GbE interfaces, the analysis proposed 14 parallel `DAs` with a processor utilization of 99.29%, which resulted in a goodput of 67.247 Gbit/s. Increasing the number to 15 parallel `DAs` reduced the processor utilization to 92.22% and increased the goodput to 68.994 Gbit/s.

For the payload size of 8192 Bytes, the adaptation was corrected for 8×10 GbE interfaces, where the `DA` processor utilization was stated as 98.97% for 13 `DAs`. The corresponding adaptation of the processing engine resulted in a goodput of 78.414 Gbit/s. By increasing the amount of parallel `DAs` to 14, the processor utilization was reduced to 95.66% and the achieved goodput increased to 79.176 Gbit/s.

Summary

The theoretical data rate was consistently reached in all cases. The maximum protocol processing overhead of 0.342% was measured for the protocol configuration with 8192 Bytes payload for a target data rate of 80 Gbit/s. In two cases, the analysis lead to an adaptation that did not allow to utilize the provided communication channels fully. In

Data-Packet Size: 1024

# of 10 GbE	1	2	3	4	5	6	7	8
# of DA	2	4	5	7	9	11	15	17
# of DC	2	3	4	6	7	8	10	11
Measured Goodput (Gbit/s)	9.858	19.72	29.588	39.447	49.303	59.155	68.994	78.828
Theo.Max. Goodput (Gbit/s)	9.863	19.726	29.589	39.452	49.315	59.178	69.041	78.904
Overhead (Gbit/s)	0.005	0.006	0.001	0.005	0.012	0.023	0.047	0.076
Overhead in %	0.051	0.030	0.003	0.013	0.024	0.039	0.068	0.096

Data-Packet Size: 2048

# of 10 GbE	1	2	3	4	5	6	7	8
# of DA	2	3	5	7	9	10	13	15
# of DC	1	2	3	4	6	6	8	8
Measured Goodput (Gbit/s)	9.896	19.785	29.702	39.6	49.465	59.351	69.245	78.972
Theo.Max. Goodput (Gbit/s)	9.902	19.804	29.706	39.608	49.51	59.412	69.314	79.216
Overhead (Gbit/s)	0.006	0.019	0.004	0.008	0.045	0.061	0.069	0.244
Overhead in %	0.061	0.096	0.013	0.020	0.091	0.103	0.100	0.308

Data-Packet Size: 4096

# of COM interfaces	1	2	3	4	5	6	7	8
# of DA	2	3	5	6	9	9	11	14
# of DC	1	2	3	3	4	6	7	8
Measured Goodput (Gbit/s)	9.915	19.836	29.739	39.675	49.581	59.403	69.261	79.12
Theo.Max. Goodput (Gbit/s)	9.921	19.842	29.763	39.684	49.605	59.526	69.447	79.368
Overhead (Gbit/s)	0.006	0.006	0.024	0.009	0.024	0.123	0.186	0.248
Overhead in %	0.060	0.030	0.081	0.023	0.048	0.207	0.268	0.312

Data-Packet Size: 8192

# of 10 GbE	1	2	3	4	5	6	7	8
# of DA	2	3	5	6	7	9	11	14
# of DC	1	2	3	3	4	5	6	8
Measured Goodput (Gbit/s)	9.925	19.855	29.788	39.701	49.601	59.467	69.346	79.176
Theo.Max. Goodput (Gbit/s)	9.931	19.862	29.793	39.724	49.655	59.586	69.517	79.448
Overhead (Gbit/s)	0.006	0.007	0.005	0.023	0.054	0.119	0.171	0.272
Overhead in %	0.060	0.035	0.017	0.058	0.109	0.200	0.246	0.342

Table 4.6: The achieved goodput when fully utilizing up to 8×10 GbE interfaces. The overhead is measured without any packet-loss and for different data-packet sizes. Additionally, the table shows the number of `Data-Packet Aggregators (DAs)` and `Data-Packet Combiners (DCs)` that were used for the transmission.

both cases, the predicted processor utilization of the original adaptation was close to 100%. In such cases, a safety margin could be applied to the processor utilization in order to avoid the performance drop.

4.7.2 Scenarios

In any real-world scenario, the communication parameters, e.g., BER and the desired data rate, will not be static, as assumed until now. In order to achieve the desired goodput, the employed processing engine could be designed for the worst-case scenario, e.g., the desired data rate of 80 Gbit/s and a BER of 10^{-5}. However, that would waste resources in case the channel conditions change for the better or the data rate decreases. Instead, in this thesis, it was proposed to readapt/replace the employed processing engine on-demand at runtime in order to fit to the communication parameters.

This is possible because the stages are only dependent on their inputs and internal state, and the parallelization of a processing engine is independent of the streamed items. Therefore, by adding or removing stages, the processing engine can be readapted at runtime without actually changing the implementation.

The adaptation of the following two scenarios are handled by the `EndpointManager` (for more details please refer to annex A.1), which monitors the communication parameters. After the endpoint manager notes that the desired data rate cannot be reached under the current conditions, the endpoint manager selects the necessary channels for the transmission. After the channels have been selected, the processing engine is automatically adapted. The automatic adaptation is based on the Processing Engine Template Language (PETL)-description and the adaptation pattern (see Technical Details 7).

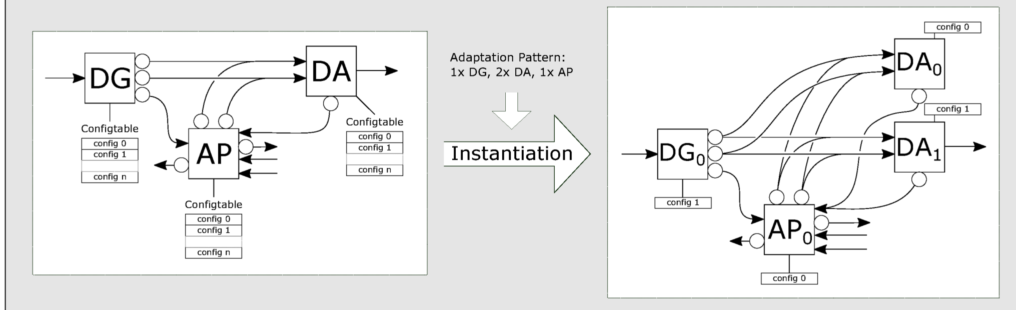
Changing Channel Conditions

Whenever the BER reduces the channel capacity to a service quality that results in the desired data rate not being met anymore, the processing engine has to be adapted in order to fit the new conditions. To be able to adapt the processing engine to fit the channel conditions, the channel BER has to be monitored. Therefore, the current BER for all channels is continuously reported to the `EndpointManager`, which compares the

Technical Details 7: Processing Engine Template Language

A Processing Engine Template (PET) is a PETL [81] description of the unadapted processing engine, extended with possible configurations of the stages. It allows deriving adaptations by assigning parallelization counts to the stages, which, when unfolded, build the desired processing engine layout and also provide the per stage configuration.

The PET is the blueprint, however, it does not give information about how a processing engine should be adapted for a certain communication situation. This information is given with *adaptation patterns*. After applying the communication characteristics, e.g., the desired data rate, the adaptation pattern provides the necessary adaptation counts for the individual stages.



desired data rate with the current capacity of the currently used channels. In case the channel capacity is insufficient, the `EndpointManager` sorts the channels with increasing BER. From this ordered list, the best x channels are selected so that x is as small as possible, and the combined capacity is higher than the desired data rate. The number of combined communication-channels is used to calculate the theoretical possible data rate, which is then used to generate the adaptation pattern, i.e., how many stages are necessary to utilize the assigned channels. Finally, the Processing Engine Template (PET) is used to instantiate the actual processing engine, given the adaptation pattern.

The channel capacities used for this scenario are shown in figure 4.22. The capacity of each channel changes every 10 ms and is calculated based on the BERs (see annex table B.1) and a data-packet payload size of 1024 Byte. After 100 ms, the pattern repeats. The desired data rate in this scenario is 29 Gbit/s.

The evaluation results are presented for three different cases: Figure 4.23 shows the goodput and total channel utilization[10] for a static processing engine that was adapted for

[10]The channel utilization results from the overall amount of transmitted data, including

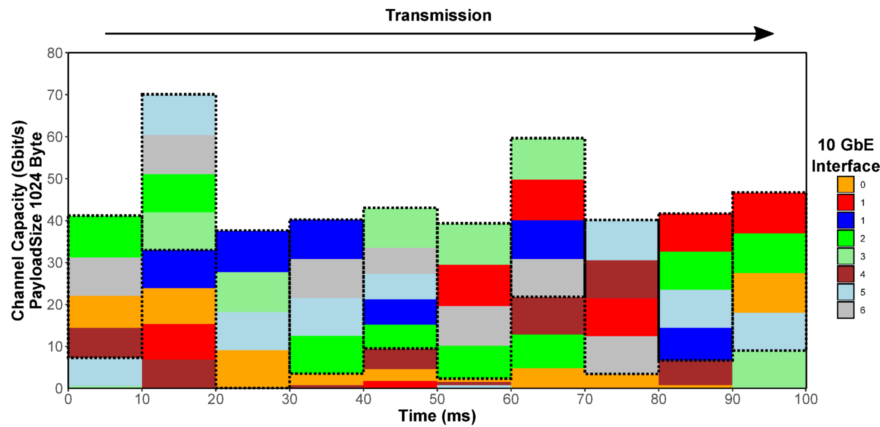


Figure 4.22: Per-channel capacity in Gbit/s given a data-packet payload size of 1024 Bytes and the BER from table B.1. The BER changes every 10ms, the pattern repeats after 100 ms. The interfaces are ordered descending by their capacity. The interfaces that are framed in the graph are used (channel bonded) by the adapting processing engine.

a desired data rate of 29 Gbit/s and a BER of 10^{-5} (*static PE29*). Figure 4.24 shows the goodput and total channel utilization for a processing engine that was statically adapted to fully utilize 8×10 GbE interfaces (*static PE80*). Figure 4.25 shows the channel utilization and goodput which was achieved when the processing engine is adapted to the channel conditions (*adapting PE*).

The *static PE29* processing engine achieved a goodput of 17.28 Gbit/s, even if it was able to use all available channels. However, since it was adapted for a data rate of 29 Gbit/s (i.e., five `Data-Packet Aggregator (DA)` and four `Data-Packet Combiner (DC)`), it was not able to utilize all channels because the assigned processing resources were limiting the protocol processing. This was avoided with the processing engine *static PE80* (see figure 4.24) that was adapted to utilize 8×10 GbE interfaces fully. As one can see, the desired data rate of 29 Gbit/s could be achieved, at the cost of always using all communication interfaces and 17 CPUs for the `DAs` on the sender side and 12 CPUs for the `DCs` on the receiver side. Additionally, one can see that the channel utilization fluctuates between 39 Gbit/s and almost 70 Gbit/s. This is due to the high number of retransmissions caused by using completely broken channels. This also leads to jitter in the goodput seen by the host.

The adapting processing engine *adapting PE* (see figure 4.25) also provided 29 Gbit/s,

retransmissions.

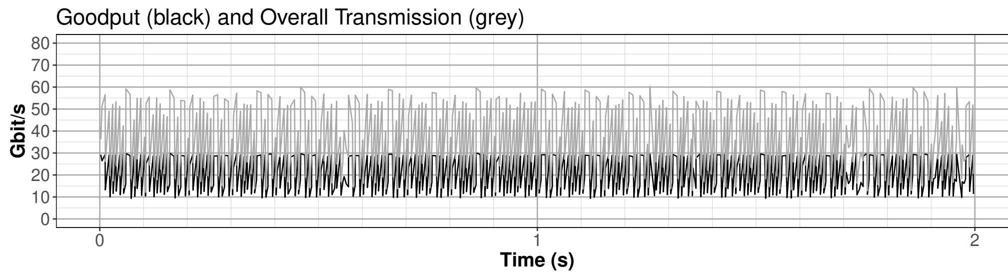


Figure 4.23: Throughput and goodput of a static version of the multichannel protocol that was adapted (5× `Data-Packet Aggregator (DA)`, 4×`Data-Packet Combiner (DC)`) for 30 Gbit/s, but was also given access to all channels. The desired data rate for the transmission was 29 Gbit/s.

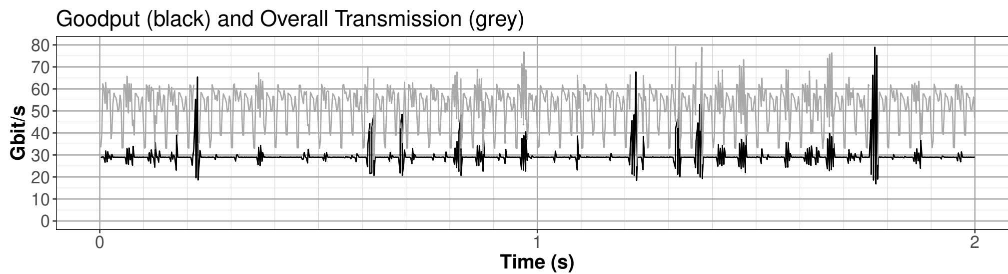


Figure 4.24: Throughput and goodput of a static version of the multichannel protocol that uses all eight 10 GbE interfaces and was also adapted (17× `Data-Packet - Aggregator (DA)`, 12×`Data-Packet Combiner (DC)`) in order to be able to utilize all interfaces. The desired data rate for the transmission was 29 Gbit/s.

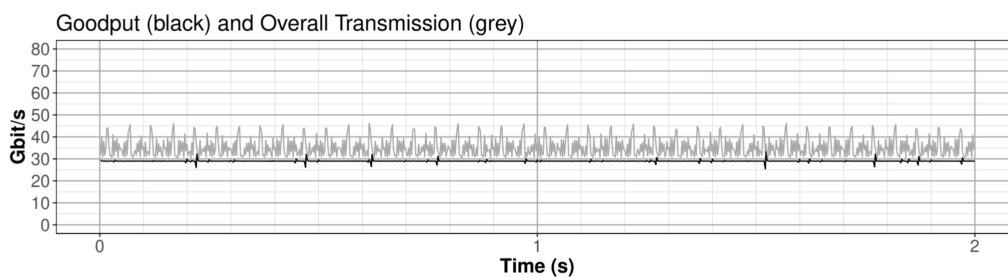


Figure 4.25: Throughput and goodput of the multichannel protocol that was continuously adapted to the channel conditions. The protocol is adapted and deployed every 10ms, when the changed BER is noted by the `EndpointManager`. The adaptation is based on the minimum number of necessary channels and the resulting theoretical data rate. The desired data rate for the transmission was 29 Gbit/s.

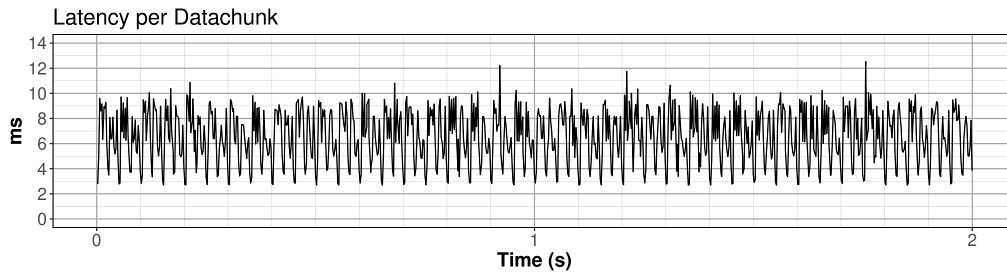


Figure 4.26: Latency per datachunk of the static multichannel protocol that uses all eight 10 GbE interfaces and was also adapted (17× `Data-Packet Aggregator (DA)`, 12×`Data-Packet Combiner (DC)`) in order to be able to utilize all interfaces.

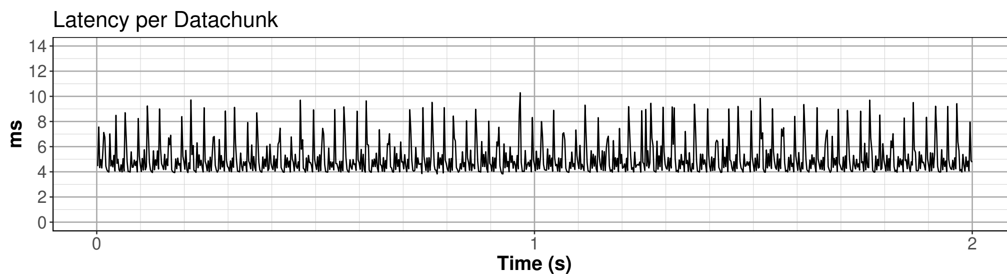


Figure 4.27: Latency of the adapting multichannel protocol. The protocol is adapted and deployed every 10ms, when the changed BER is noted by the `EndpointManager`. The adaptation is based on the minimum number of necessary channels and the resulting theoretical data rate.

however, with a lower channel utilization and less jitter compared to *static PE80*. Due to the immediate adaptation to the new channel conditions, the resource consumption could be significantly reduced. Table 4.7 shows the resources used by the adapting processing engine depending on the capacity of the communication interfaces. Given that the non-adapting variant uses all eight 10 GbE interfaces and 30 CPUs at the receiver and 38 CPU at the sender, the resource consumption can be reduced significantly.

Additionally to the goodput, the latency per datachunk also profits from the on-demand adaptation as seen in 4.26 (static) and figure 4.27 (adapting). This is because of the higher amount of retransmission when using the *PE80* processing engine, which lead to a higher amount of necessary acknowledgments and therefore, to a higher latency.

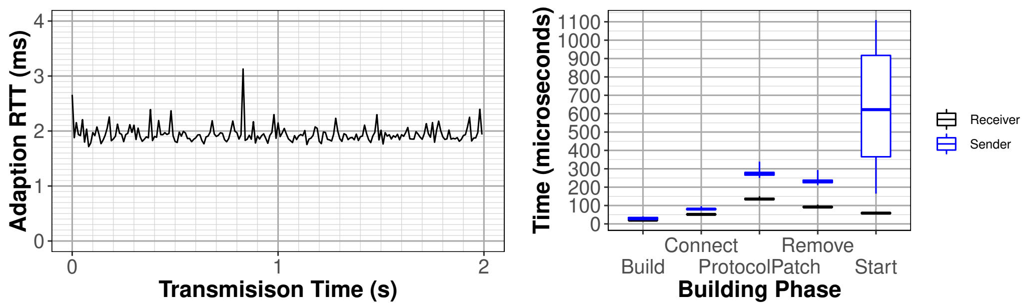
In order to be able to profit from changing the communication interfaces, the adaptation

Time → Resource ↓	0ms	10ms	20ms	30ms	40ms
Receiver CPUs	20	21	20	21	23
Sender CPUs	24	25	24	25	28
10 GbE Interfaces	0,3,5,7	2,3,6,7	0, 2, 4, 6	2, 3, 6, 7	2, 3, 4, 6, 7

Time → Resource ↓	50ms	60ms	70ms	80ms	90ms
Receiver CPUs	21	21	21	21	20
Sender CPUs	25	25	25	25	24
10 GbE Interfaces	1, 3, 4, 7	1, 2, 4, 7	1, 5, 6, 7	1, 2, 3, 6	0, 1, 3, 4

Table 4.7: CPUs and COM interfaces used by the *adapting PE* on the sender- and receiver-side, depending on the channel quality. The channel quality changed every 10 ms (please refer to figure 4.22 for the corresponding channel capacities).

has to be fast. Figure 4.28a shows the duration of a full adaptation cycle, i.e., from the moment the receiver decides to adapt the processing engine until it receives the acknowledgment from the sender that it finished its adaptation. As one can see, this adaptation Round Trip Time (RTT) is around 2ms, which accounts for roughly 20% of the timespan during which channels are stable. Consequently, one would expect to see an



(a) Time between start and acknowledgement of protocol adaptation in ms.

(b) Building times of the protocols in μs.

Figure 4.28: The costs of the on-demand readaptation of the processing engine.

impact on the achieved goodput. The reason why the adaptation RTT does not affect the overall goodput is threefold:

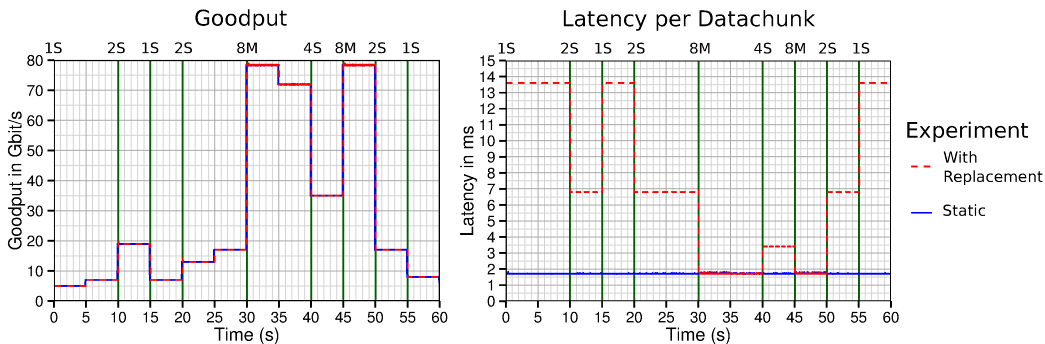
Firstly, the combined communication interfaces have (usually) slightly higher capacity than necessary for the transmission. This has the effect that losing transmission time due to the adaptation can be caught up in the remaining 8 ms (before the interface's capacity changes again).

Secondly, the communication channels do not always have to be changed entirely, i.e., while one channel may break down completely, the other channels can be used further. That further reduces the negative impact of the adaptation RTT.

Thirdly, the adaptation RTT is the time until the adaptation has been acknowledged. However, neither the receiver nor the sender waits for an acknowledgment. Instead, after sending the adaptation command, the receiver already starts its own adaptation, assuming that the sender will be able to follow immediately. Furthermore, the old processing engine is still available until all stages are idle. Consequently, the actual time spent for adaption on the sender/receiver is crucial.

The adaptation costs[11] (separated into the adaptation phases) for the sender and receiver are shown in figure 4.28b. As one can see, the actual adaptation costs are significantly lower than the adaptation RTT suggests, meaning that the adapted processing engine is employed before the acknowledgment arrives at the receiver. Please note that the phases *ProtocolPatch*, *Build*, *Remove*, and *Connect* are carried out in parallel to the transmission, i.e., only the commit phase may stall the transmission. The comparably high costs for the start-phase at the sender are caused by the readaptation of the `DA`. Every time a `DA` is configured, it frees the currently used network-buffers, acquires a new network-buffer pool and a new network-buffer. These costs could be avoided by preparing DAs before the transmission. However, that would complicate the protocol building process because instead of just building a new `DA`, a fitting `DA` with the correct configuration would have to be found.

[11] Benchmarks and technical details for the replacement and readaptation were published in [82].



(a) The goodput in Gbit/s over time of for an adapting (red) and a static (blue) processing engine.

(b) The latency per datachunk in ms over time of for an adapting (red) and a static (blue) processing engine.

Figure 4.29: The sending host changes the desired data rate over time. This leads to readaptation of the singlechannel protocol as long as the desired data rate is below 70 Gbits/s. In the case, the desired data rate increases over 70 Gbit/s, the whole protocol is replaced with the multichannel version. The goodput-graph is annotated with the number of combined interfaces and the protocol version (e.g., 2s – 2 interfaces/singlechannel, 8m – 8 interfaces, multichannel).

Changing Data-Rates

This scenario focuses on a situation in which an application's data rate requirements vary over time so that the theoretically achievable data rate of a single physical communication interface is not sufficient. When the data rate changes, the new data rate is reported to the endpoint manager that compares the currently achievable data rate with the required data rate. In the case that the currently achievable theoretical data rate is not sufficient (or higher than necessary), the endpoint manager solves that situation by building a new processing engine, which combines the necessary number of communication interfaces and stages. In this scenario, the current protocol is not just adapted, but actually replaced by another protocol. What protocol the endpoint manager chooses depends on the required data rate. The singlechannel protocol is selected for data rates below 70 Gbit/s, so that the application can profit from the lower latency. Otherwise, the multichannel protocol is used because it can provide a higher maximum data rate than the singlechannel protocol.

The results for this scenario are shown in figure 4.29a and 4.29b. The sender starts with a data rate of 5 Gbit/s and changes the data rate every 5 seconds. For comparison, measurements were also carried out using the multichannel protocol with 8×10 GbE

interfaces without protocol replacements. The results show that the measured goodput for the experiments with replacement and without replacements are basically identical, i.e., the replacement has no negative effect on the transmission. Furthermore, it shows that the approach can be used to react to changing communication parameters without a significant delay.

Summary

Both scenarios took the possibility of changing communication conditions over the course of a transmission into account. The adaptation and replacement showed to be a better alternative to the worst-case adaptation as the resource utilization can be reduced significantly by on-demand readaptation.

The first scenario focused on changing channel conditions for which a constant data rate of 29 Gbit/s was desired. The goal was to provide the desired data rate, while minimizing the resource consumption during the transmission, however, without touching the protocol implementation or increasing the protocol's complexity. This was achieved by adapting the protocol processing engine automatically to combine the necessary number of communication channels. The results showed that readapting the processing engine spontaneously is possible at runtime for high data rates without degrading the quality of service.

The second scenario focused on changing data rate requirements that were solved by the replacement of the complete protocol. In this example, the transmission was optimized for latency when data rates lower than 70 Gbit/s were required. However, when the desired goodput increased over 70 Gbit/s, this requirement was dropped in favor of a higher maximum throughput. Again, the on-demand replacement did not show negative side effects, while reducing the resource consumption.

4.7.3 Beyond Channel Bonding

A single device may not be sufficient for the protocol processing. In this thesis, homogeneous, as well as heterogeneous systems, are supported. In order to be able to employ several devices, the processing engine has to be distributed over several parts of the communication system.

The distribution of processing engines is used when a single piece of processing hardware is not capable of providing the desired data rate due to the lack of resources. Since the stages in a processing engine communicate only by message streams and are only dependent on their internal state, they can theoretically be distributed arbitrarily over a given communication system, as long as the stream processing paradigm is followed, the soft real-time requirements are fulfilled, and streams can be routed between the stages that are located on different devices.

All the previous transmissions omitted the transfer of payload between host and embedded manycore. The reason was that the available PCIe 2.1 interfaces at one of the hosts could not provide the necessary throughput. This problem can be solved by combining two embedded manycores, which allows scaling up the theoretical data rate with the number of devices. Each of the combined embedded manycores is connected to the host via an individual PCIe 2.1 interface.

Figure 4.30 shows a simplified multi-device processing engine. The two devices are

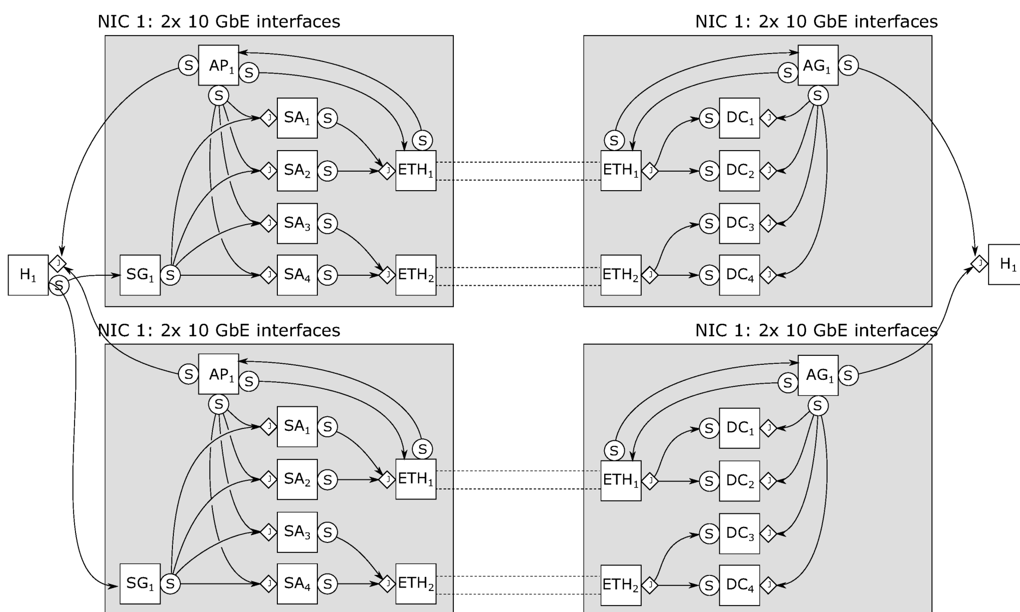
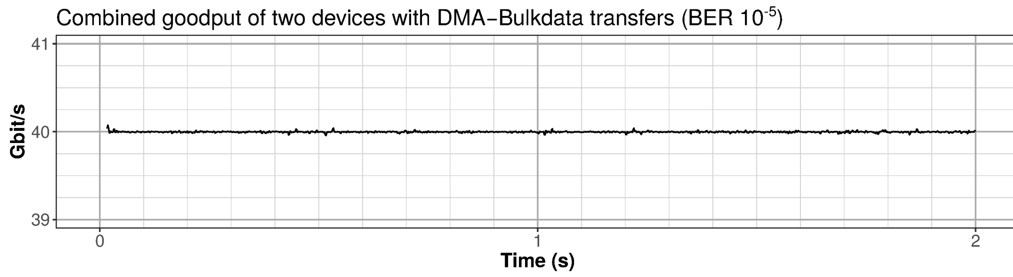
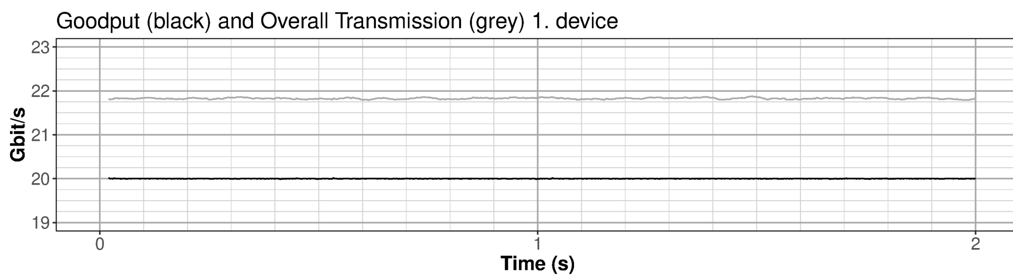


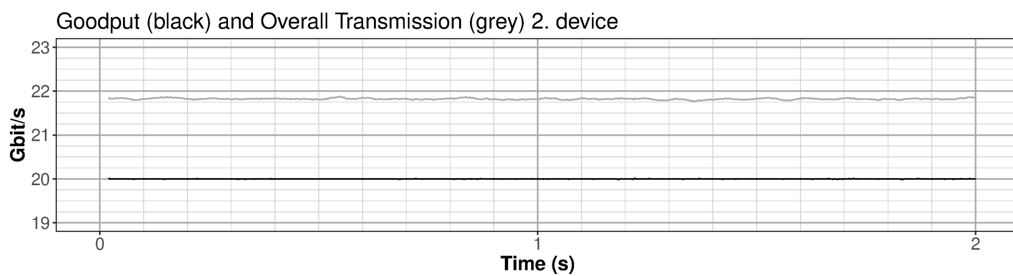
Figure 4.30: The protocol processing engine is distributed over two devices per endpoint.



(a) Goodput of a transmission with two combined devices as seen by the host.



(b) Goodput of a transmission with two combined devices as seen by first device.



(c) Goodput of a transmission with two combined devices as seen by second device.

Figure 4.31: The goodput and throughput of a multi device transmission.

combined by providing an additional data stream to the second device, whereas the necessary modifications to the host are minimal: It only needs the additional stages for the PCIe interfaces in order to connect to two devices. The processing engines on the embedded manycores do not need to be changed either, as both of them work individually and see only their part of the transmission.

This concept is used to build a communication system that is able to transmit 40 Gbit/s with a BER of 10^{-5}. In prior experiments, the maximum throughput of a single PCIe 2.1 interface was measured with 27.84 Gbit/s [87]. Consequently, the combined devices should be able to deliver a goodput of 55.68 Gbit/s. Each of the devices will have to process a data rate of 20 Gbit/s. Consequently, the adaptation has to be redone. After adapting and testing, the individual processing engines were adapted with 5×`DA`, 3×`DC`, and 3×`COM`.

The combined goodput as seen by the host and the goodput/channel utilization of the two devices given a desired data rate of 40 Gbit/s and a BER of 10^{-5} is shown in the figures 4.31a-4.31c. Firstly, one can see that the usage of two devices leads to a stable goodput of 40 Gbit/s as seen by the host. Furthermore, the figures show that the transmission was equally distributed between the two embedded manycore, each transmitting 21.9 Gbit/s, which lead to a goodput of 20 Gbit/s for each.

Summary

This scenario showed how to use the stream processing protocol processing approach to scale up the protocol processing beyond the capacities of a single device. Since individual devices work independently, the combined goodput scales linear and is only limited by the capacity of host's memory- and PCIe interfaces.

CHAPTER 5

Conclusion

This thesis proposed that communication protocols should be understood as soft real-time stream processing problems. The protocol processing was separated into independent processing stages that were connected to a processing graph, which can be analysed for its soft real-time requirements given a certain data rate. However, such analysis can be cumbersome when done manually, mainly because stages behave differently depending on the message's content and the stage's state. Therefore, a simulation environment was conceived that allows for a straightforward simulation of the protocol implementation. During the simulation, messages can be classified depending on their content and the stage's state, in order to retrieve fine-grained, soft real-time requirements.

In order to identify bottlenecks and gather insights on the necessary parallelization count of the individual stages, these stages were benchmarked on the target hardware. While the individual benchmarks increased the implementation effort, it was shown that by including the target hardware at an early design stage, the stream processing analysis allowed the identification of processing bottlenecks. Furthermore, the benchmarks gave insights into the parallelization of the protocol and also helped to find bugs in the implementation.

It was shown that the analysis leads to useful performance predictions that could be further used for the parallelization of the protocol processing. The parallelization itself proved unusually straightforward. Due to the stream processing approach which decomposes the protocol into independent stages, the parallelization was possible by stream operators. The stream operators split, duplicated, or joined the streams in order to distribute individual messages and consequently parallelize the processing. Since all protocol processing tasks are isolated from each other and only triggered by receiving a message, no further synchronization was necessary.

Following the stream processing paradigm, up to the mapping step, allowed to easily distribute protocol processing tasks over different processing hardware, such as host,

embedded manycore, and external Field Programmable Gate Arrays (FPGAs). Furthermore, it was shown that the theoretically achievable data rate could be increased over the possibilities of a single device by adding more devices and mapping the protocols accordingly.

However, the static schedule also presented itself as a caveat in the context of end-to-end communication, since the communication conditions and requirements are usually not static in a wireless scenario. Either the schedule was based on the worst-case requirements and wasting resources, or more stressing communication conditions could not be met. That problem was alleviated by extending the concept with the ability to alter processing engines at runtime, while they are heavily used. It was shown that the presented approach was able to react to requirement changes as fast as in a 10 ms interval without impacting the quality of service.

The main goal of the thesis, i.e., processing ultra-high volume data streams, was achieved. It was shown that data rates up to the processing hardware's theoretical maximum of 80 Gbit/s could be processed. The stream processing based protocol processing imposed a maximum overhead of 0.342%. Furthermore, a multi-device parallelization strategy was presented that allows to process communication protocols on disjunct devices in order to further increase the possible data rate.

However, more research questions arose. Firstly, the mapping of the processing engines in this thesis was static but, in a multi-user scenario, dynamic resource management is needed. While first investigations on dynamic resource management for protocol stream processing were conducted in the master thesis Zuzana Gabonayová [89], more research on the integration has to be conducted.

Secondly, the concept developed in this thesis may allows the generation of hardware implementations with the same design process. While there are already approaches to synthesize hardware descriptions from Extended Finite State Machines (EFSMs), the feasibility of hardware implementation should be further investigated. Primarily the question, how can performance characterization, extracted from a software implementation, be used in order to conceive parallelized hardware implementation?

Thirdly, this thesis focused on high-volume data streams, however, in concerns of most internet communication, the connections are short-lived and have a low volume. The presented protocol would not be suitable for short connections because it focuses on large

packets. It would be interesting to investigate a stream processing protocol for short connections.

Finally, protocols are traditionally not designed with parallelization of the processing in mind. Therefore, it would be interesting to investigate the applicability of the design process to a general-purpose protocol such as TCP/IP.

5.1 Own Publications used in this thesis

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", 2015 IEEE 40th Conference on Local Computer Networks (LCN), Clearwater Beach, FL, 2015, pp. 398-401

In this publication, I developed the initial idea of using the stream-processing paradigm for protocol processing. Furthermore, the concept of stream-operators and adaptation of the processing engine was presented in this publication. Finally, the initial frame-format idea that is the bases of the protocols presented in this thesis was introduced as joint work with Dr.-Ing. Lukasz Lopacinski.

S. Büchner, L. Lopacinski, J. Nolte and R. Kraemer, "100 Gbit/s End-to-End Communication: Designing Scalable Protocols with Soft Real-Time Stream Processing", 2016 IEEE 41st Conference on Local Computer Networks (LCN), Dubai, 2016, pp. 129-137.

In this publication, I used the concept to design scalable communication protocols. The protocol I presented in this paper is an early version of the protocol presented in this thesis. Furthermore, two different versions of the protocol were developed. A low-latency version, similar to the singlechannel protocol presented in this thesis, and a multilane protocol version, that provided a distinct pipeline for each communication interface. Both protocols were evaluated in order to show the feasibility of the processing approach. Furthermore, the integration of the FPGAs was presented in this paper as a joint work of Dr.-Ing. Lucasz Lopacinski and my self.

Büchner, S., Lopacinski, L., Kraemer, R., et al. (2017). "Protocol Processing for 100 Gbit/s and Beyond – A Soft Real-Time Approach in Hardware and Software.", Frequenz, 71(9-10), pp. 427-438.

In this journal paper the authors jointly described a full communication system meant for 100 Gbit/s wireless communication. The main aspects such as the stream processing concept and Forward Error Correction (FEC) calculation and parallelization were summarized. Furthermore, the integration of the external accelerator was discussed in this publication.

S. Büchner, J. Nolte, A. Hasani and R. Kraemer, "100 Gbit/s End-to-End Communication: Low Overhead On-Demand Protocol Replacement in High Data Rate Communication Systems," 2017 IEEE 42nd Conference on Local Computer Networks (LCN), Singapore, 2017, pp. 231-234.

In this paper, I introduced the concept of virtual channels as a tool for the on-demand replacement of complete communication protocols. Furthermore, the general replacement process was presented here. The concept was evaluated by replacing the currently used protocol with a newly build version of itself.

S. Büchner, A. Hasani, L. Lopacinski, R. Kraemer and J. Nolte, "100 Gbit/s End-to-End Communication: Adding Flexibility with Protocol Templates", 2018 IEEE 43rd Conference on Local Computer Networks (LCN), Chicago, IL, USA, 2018, pp. 263-266.

In this publication, I introduced the Processing Engine Template Language (PETL) as a tool to describe a family of communication protocols with protocol templates. I used PETL to describe a template of the multilane protocol presented in 2016. The feasibility of the idea was evaluated with a channel hopping hopping protocol that switched to a new channel in case the channel quality became insufficient. The channel hopping was achieved by dynamically building a new protocol that uses a different channel and replacing the old protocol.

APPENDIX A

Supplementary Technical Information

A.1 Exchanging Processing Engines

All parts of the communication system need to implement their part of the desired processing engine. Since the layout of a suitable processing engine depends on the application and communication conditions and is, therefore, not known before the transmission, the communication system is firstly initialized with a minimal processing engine called `ProtocolStub`. The protocol stub, shown in figure A.1, consists of an `EndpointManager`, a host-to-NIC interface, such as PCIe, a management protocol, and additional communication interfaces, such as Ethernet. The `EndpointManager` is responsible for managing the local `ProtocolStub`, for the communication between host and NIC, and for monitoring the communication parameters, such as the desired data rate. The management protocol is responsible for the management-communication with the remote communication endpoint.

The `ProtocolStub` has two main objectives: Firstly, establishing a connection between the host and the NIC and setting up a communication interface for the management protocol. Secondly, the `EndpointManager` waits for a PETL description. Upon receiving a PETL description, the `ProtocolStub` is patched with the new processing engine. The PETL connection can originate from the local host or from the remote endpoint.

The protocol management is responsible for exchanging states, PETL descriptions, and commands between two endpoints. The management protocol is a simple retransmission protocol that employs its own virtual channel to separate itself from other processing engines.

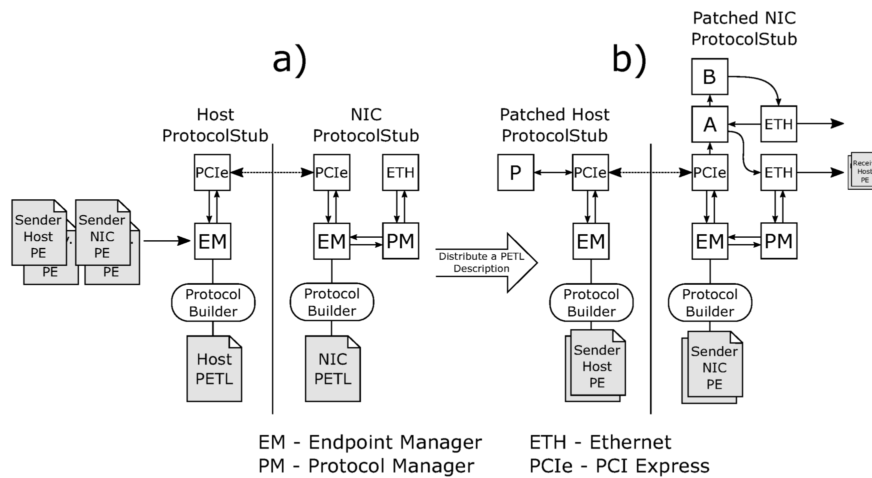


Figure A.1: a) The host and the NIC start with a minimal `ProtocolStub`. After connecting the stubs via a communication interface, such as PCI Express, the host sends the PETL description to the other `ProtocolStub`. b) Upon receiving a PETL description, the `ProtocolStub` patches itself with the new processing engine.

A.2 Offloading of Stages

The offloading approach needs a general interface, so that message streams can be spun arbitrary over device/address space boundaries. The interface has to connect stages transparently over these devices/address space boundaries, i.e., it has to transparently transport bulk data and messages. However, in the general case, there will be fewer actual communication channels between the devices than data streams between the remote processing engines. Therefore, the data streams have to be multiplexed over the communication channels. The communication channels are used for bulk copying of data as well as transferring messages between host and embedded manycore.

The offloading is based on proxy-stages. Each message stream that crosses an address space border is routed through a proxy-stage. This proxy-stage mimics all inputs of the stage it replaces. The stage to be replaced has a unique identifier that is used for a lookup-process and configuration process, as shown in figure A.2 (1)-(6).

The lookup-process is initiated with a configuration message (1) to the proxy stage `B'`. The proxy `B'` then sends a lookup-request to the local `LookupServer` (2). The `LookupRequest` contains the unique identifier of the stage the proxy `B'` shall replace. The

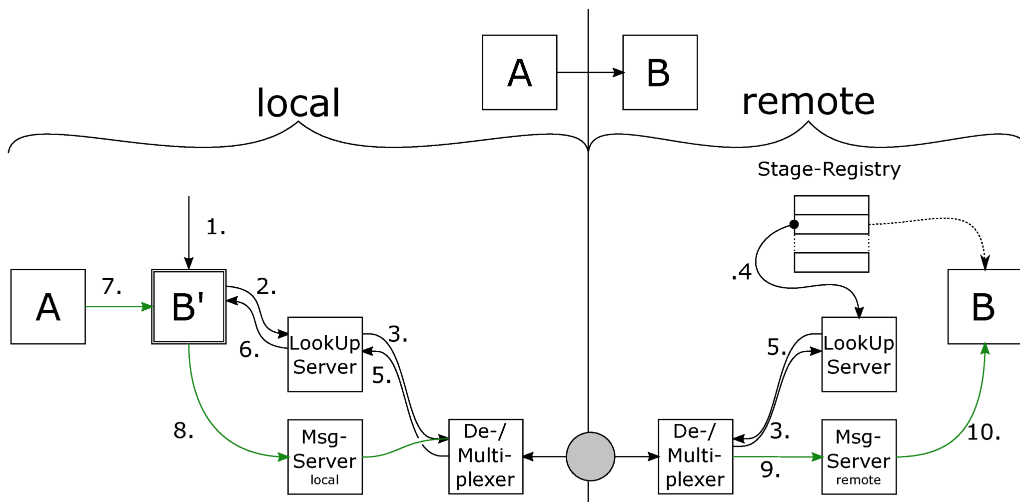


Figure A.2: Address information lookup and cross-device message delivery with the help of proxy messages and proxy-stages. The steps are as follows: 1. Configure message. 2. Send lookup-request. 3. Forward lookup request to remote lookup server. 4. Retrieve stage B's address information. 5. Send lookup-answer. 6. Forward lookup answer to proxy B'. 7+. Send message from A to B by using the configured proxy B'.

local `LookupServer` forwards the request to the remote `LookupServer` (3), that requests the addressing and marshaling information from the `Stage-Registry` (4) and sends back the lookup-answer to the proxy `B'` (5+6). After configuring itself according to the lookup answer, proxy `B'` is ready to forward messages to the remote stage `B`.

When the proxy-stage receives a message at one of its inputs (7), it will marshal the payload and the addressing information into a `ProxyMessage` that is passed to a local $\texttt{MsgServer}_{local}$ (8). The $\texttt{MsgServer}_{local}$ sends the `ProxyMessage` via the communication interface, such as PCIe, to a $\texttt{MsgServer}_{remote}$ on the remote device. Since, the address information is coded in the `ProxyMessage`, the receiving $\texttt{MsgServer}_{remote}$ has only to transform the `ProxyMessage` into the locally used message format and hand the message over to the message passing subsystem (9). Each input of the `StageProxy` can be configured as a buffer input, that can be used to copy buffers between devices transparently.

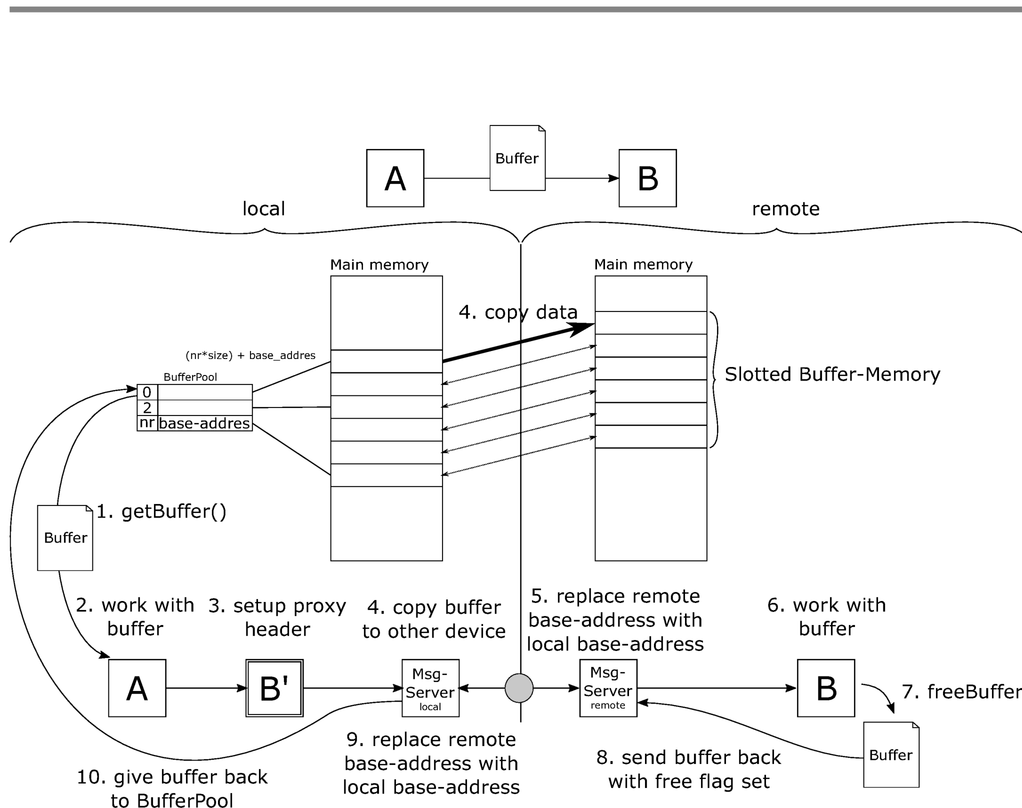


Figure A.3: Transparent zero-copy buffer transport using slotted buffer-pools and a proxy concept.

Zero-Copy Bulk Data Transport

The bulk-data transport is implemented with the help of preallocated slotted huge pages. Within address space boundaries, e.g., within the embedded manycore, buffers are passed between stages by forwarding the buffer descriptor and accessing the buffer by its memory address. However, between devices, the buffer's content has to be copied. In order to minimize the copying overhead, a zero-copy buffer transport approach is employed, as shown in figure A.3.

Each buffer has an identifier corresponding to its slot in the hugepage. Such slotted memory-area is provided at the local and the remote devices and serves as the zero-copy target memory. Slot n in the hugepage refers on both devices to the same buffer.

Consequently, when the buffer-descriptor is located at the remote side, the corresponding buffer-memory is unavailable on the local side and vice versa. While this approach halves the effectively available buffer-memory, it allows for a fast and straightforward zero-copy

strategy because the remote copy-destination is known at all times. The zero-copy buffer transport approach is depicted in figure A.3 and explained in the following.

After acquiring a buffer, stage `A` will work with the buffer until eventually passing the buffer-handle to (remote) stage `B` on the remote device. The copy-process starts by sending the buffer-handle from stage `A` to proxy-stage `B'`. The receiving input (configured as a buffer input), will marshal the message into a special `BufferMessage` before passing the message to the $\texttt{MsgServer}_{local}$. Upon receiving the message, the $\texttt{MsgServer}_{local}$ will firstly copy[1] the buffer's memory-area to the remote side's buffer slot of this buffer and will secondly transmit the `BufferMessage` to the remote device. On the remote-side, the $\texttt{MsgServer}_{remote}$ will replace the base address of the buffer with the local base address, before passing the buffer-handle to the actual stage `B`. From now on, stage `B` can work with the buffer until it is eventually freed. Freeing the buffer is accomplished by sending a `FreeBufferMessage` back, which causes $\texttt{MsgServer}_{local}$ to pass the buffer back to the BufferPool.

[1]The DMA buffer transport was developed in the master thesis of Leonard Förster in the context of this dissertation [87].

APPENDIX B

Data Link Protocol

The following pages are used to explain details of the data link protocol that were omitted in the main body of this thesis for sake of readability.

B.1 Protocol Data Structure

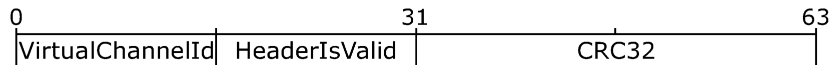
The prototype data link protocol uses the following frame format. The sizes of payload, headers, and footers are chosen to be a multiple of 64 bit, which is the memory bus width of the targeted processing hardware (Mellanox TileGx72 [90]).

B.1.1 DataChunk

The datachunk is used for the transport of bulk-data between the protocol processing engine and the host. The latency per datachunk and the number of host invocations[1] depend on the size of the datachunk. Considering that a large datachunk causes fewer host invocations but also a higher latency and vice versa, the size has to be a compromise. A datachunk is described by a datachunk descriptor `DataChunkDesc` that contains the address of the payload, the payload's size, and a sequence number that is used for reordering.

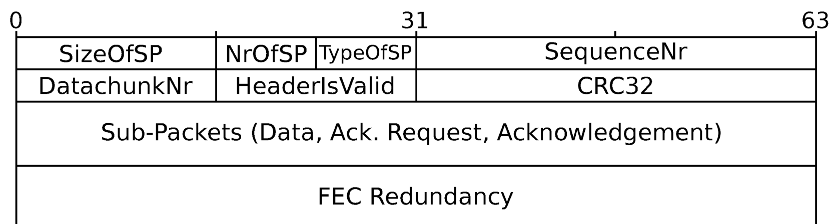
[1] Handling incoming and outgoing packets.

B.1.2 Virtual Channel Header



The `VirtualChannelHeader` is used to multiplex different protocol-generations on a physical channel by means of a virtual virtualization (sec. 3.4.2). It is provided by the processing framework and is prepended to any outgoing network-frame. The `VirtualChannelHeader` contains the `VirtualChannelId`, which is used to distinguish processing engines that are multiplexed on the same physical channel. Additionally, the virtual channel header provides a CRC field used for error-detection. The remaining 16 Byte (`HeaderIsValid`) are used to state whether the header contains errors. The `HeaderIsValid` field is filled by a special purpose accelerator, such as a Field Programmable Gate Array (FPGA), in order to mark the `VirtualChannelHeader` as unrecoverable.

B.1.3 Frame and SuperFrame



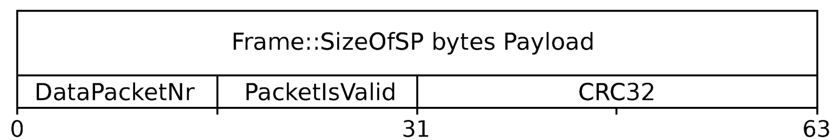
A `Frame` is used to aggregate `NrOfSP` $\times$ sub-packets of the type `TypeOfSP` with each having the size stated in the field `SizeOfSP`. The type of a sub-packet can be `Data`, `Acknowledgement`, `AcknowledgementRequest`, or `FinalAcknowledgement`.

The `SequenceNr` interpretation depends on the sub-packet type (explained in the next sub-section). All sub-packets in a frame belong to the datachunk with the number `DatachunkNr`. Finally, the header contains the CRC (`CRC32`), and a `HeaderIsValid` field. The `HeaderIsValid` field states whether the frame's header is correct (`HeaderIsValid == 0`), i.e., correctly transmitted or successfully reconstructed. If the header could not be reconstructed, the `HeaderIsValid` field is set to a non-zero value and the frame has to be dropped. The redundancy data for the reconstruction of an erroneous header can be appended after the last sub-packet.

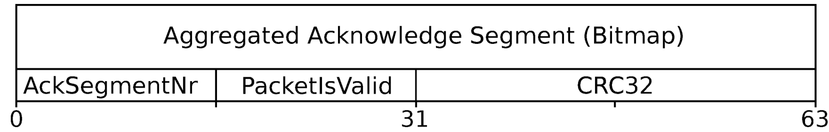
The proposed frame-format can only aggregate sub-packets of the same size and type that belong to the same datachunk. While this restriction reduces the necessary meta data per sub-packet, it can lead to underutilization of the communication channel, in case the frame cannot be filled completely. This physical-channel underutilization can be avoided by aggregating network-frames into `SuperFrames`. However, the `SuperFrame` is not used in this thesis.

B.1.4 Sub-Packet

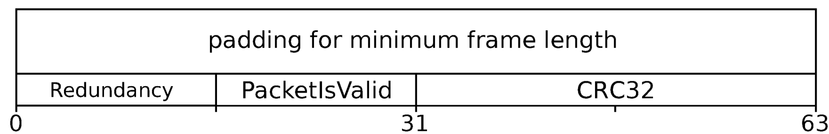
All data-, acknowledgment-, and acknowledgment-requests-packets are wrapped as payload in sub-packets. Each sub-packet starts with the payload (the payload size is stated in the frame's header). The payload is followed by a sub-packet specific footer. This order was chosen because it allows an "on-the-fly" FEC/CRC (de-)coding in hardware. Each sub-packet contains a `PacketIsValid` field and a CRC. The CRC is used to detect sub-packets that need to be reconstructed by the Forward Error Correction (FEC). In the case a sub-packet could not be reconstructed, the `PacketIsValid` field is set to a non-zero value, which indicates that it has to be ignored in the further processing of the frame. The three types of sub-packets are explained in the following.



The footer of the `Data` sub-packet contains the `DataPacketNr` that states the position of the payload relative to the datachunk's beginning, as well as the mentioned `PacketIsValid`/`CRC` fields. The frame-header's `SequenceNr` is interpreted as the position of the corresponding datachunk in the overall transmission. The frame's `TypeOfSP` field has to be set to `Data`.



The protocol uses aggregated acknowledgments[2], that state all missing packets for a whole datachunk. The individual sub-packets are coded as a bitmap (1 — received / 0 — missing). The bitmap is the payload of the `Acknowledgement` sub-packet. Additionally, the `Acknowledgement` sub-packet has a field that states to which segment of the datachunk this acknowledgment packet refers. Consequently, the bitmap of a whole datachunk can be divided into "sub-bitmaps" which refer to a consecutive section of the datachunk. This approach is similar to the segmentation of data frames and reduces the probability of losing acknowledgments due to bit-errors. The frame's `SequenceNr` field of the frame is used to suppress outdated acknowledgments. The frame's type can be set to `Acknowledgement` or `FinalAcknowledgement`. `Acknowledgement` implies that at least one data packet is missing, and `FinalAcknowledgement` indicates that the whole datachunk was transmitted correctly.



The `AcknowledgementRequest` is used to request an acknowledgment/final-acknowledgment from the receiver, and it does not contain an actual payload. However, it can contain padding bytes because the underlying hardware layers may expect a minimum frame-length (e.g., 64 Byte for Ethernet [91]). Additionally, the acknowledgment request provides a `Redundancy` field, stating the number of acknowledgments that have to be sent in response to the acknowledgment request. This is used when a high packet-loss rate is expected during the transmission. Because the acknowledgment request can be sent redundantly, too, the frame's `SequenceNr` field is used to identify and suppress duplicate acknowledgment requests.

[2]The acknowledgment is a combination of ACK and NACK, however, due to readability reasons, it will be referred to as acknowledgment.

B.2 Description of the Protocol Processing Stages

The following pages are used to describe the behaviour of the processing stages. The section starts with an explanation of the used message types, and is followed by the behavior and performance measurements of the individual protocol processing stages. This section finishes with an explanation of the complete processing engine of the prototype data link protocol.

B.2.1 Message Types

Additionally to the protocol data types (that also act as messages), the following message types are used by the implemented stages.

TransmissionStatus

The `TransmissionStatus` contains information about the transmission status of a datachunk. The most important states are:

1. `AllPacketsTransmittedOnce` – The data packets of the datachunk were transmitted and the acknowledgement phase can start.
2. `AcknowledgementProcessed` – The acknowledgement was completely processed, all missing data packets were retransmitted.
3. `WaitForNewChunk` – A protocol pipeline is ready to process a new datachunk.

DataChunkDesc

The `DataChunkDesc` describes a datachunk, i.e., the chunk number (`ChunkNr`), the sequence number (`SequenceNr`), and size (`SizeOfChunk`), and it provides information about the assigned buffer, e.g. the buffer's pointer and its size.

QueueStatus

The QueueStatus is used to to coordinate the communication between host and DataChunk Distributor (DD). It provides the number of started and finished datachunks as well as the remaining space in the datachunk queue.

DataPacketDesc

The DataPacketDesc describes a data sub-packet, i.e., the chunk number (ChunkNr) and the sequence number (SequenceNr) of the datachunk it belongs to. Additionally, it provides a pointer to the data within the datachunk buffer (Address), and the data packets id (Id).

AckDesc

The AckDesc message describes data packets that were copied into data chunk buffer. Additionally to the data packet ids, the AckDesc contains the ChunkNr it refers to.

B.2.2 Data-Packet Generator (DG)

Figure B.1 shows the interface and the state-machine of the DG. It is configured with the data packet size and the number of consecutive data packets to which a DataPacketDesc message refers. The behavior of the DG's $\texttt{DataChunkDesc}_{\texttt{In}}$ ($\texttt{DCD}_{\texttt{In}}$) input is described by the state-machine. The input behavior is explained in the following.

The DG's task is to transform a datachunk into a stream of data packets. Upon receiving a DataChunkDesc message, the datachunk counter DG:ChunkNr of the DG is stored in the ChunkNr field of the DataChunkDesc, i.e., the datachunk's internal identifier is assigned. After forwarding the DataChunkDesc via the DG's $\texttt{DataChunkStarted}_{\texttt{Out}}$ output, the datachunk-buffer is cut into data packets according to the configured data packet size (DG::DPSize). Each DataPacketDesc message is sent over the $\texttt{DataPacketDesc}_{\texttt{Out}}$ ($\texttt{DPD}_{\texttt{Out}}$) output. Finally, a transmission status message with the status AllPackets-TransmittedOnce is sent via $\texttt{TransmissionStatus}_{\texttt{Out}}$ ($\texttt{TS}_{\texttt{Out}}$) output and the DG's datachunk counter DG::ChunkNr is increased.

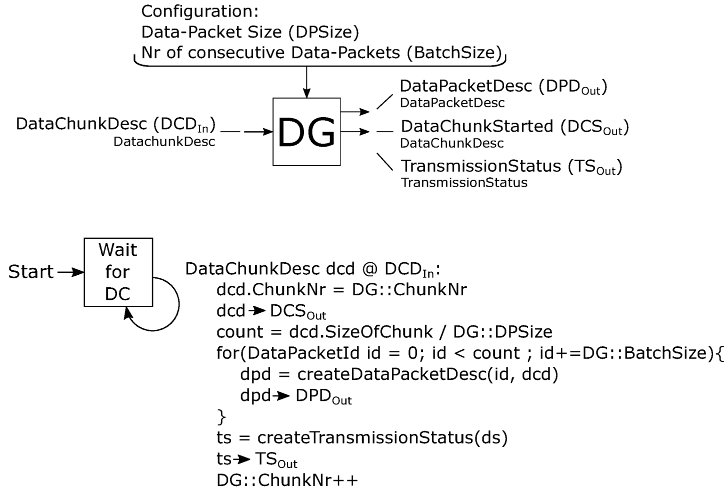


Figure B.1: The interface (i.e., inputs and outputs) and EFSM of the `Data-Packet Generator (DG)` stage. The `DG` is responsible for cutting a datachunk into data packets, and forwarding the corresponding `DataPacketDesc` messages via its $\texttt{DataPacketDesc}_{\texttt{Out}}$ output.

Performance Characteristics

The `DG` is configured with a data packet size of 1024 Byte and batch-size of 8 data packets. Measuring the performance characteristics for the $\texttt{DataChunkDesc}_{\texttt{In}}$ input of the `DG` is straight-forward because the only identified message class `ALL` (a plain `DataChunkDesc` message) does not require any setup of the `DG`. Therefore, the $\text{TPM}^{\texttt{All}}_{\texttt{DataChunk}_{\texttt{In}}}$ (read as: Time consumed per `DataChunkDesc` Message received at the $\texttt{Datachunk}_{\texttt{In}}$ input) can be measured by simply sending `DataChunkDesc` messages to the `DG`.

Figure B.2 shows the $\text{TPM}^{\texttt{All}}_{\texttt{DataPacketDesc}_{\texttt{In}}}$ depending on the mapping of the `DG` to a certain core whereas the cores are ordered by ascending Time-Per-Messages (TPMs). The medians of the measured TPMs are in a range of min. 201577 *ns* to max. 205030, so the mapping seems to have an effect on the TPM. However, the high inter-quartile range of the measured TPMs indicates that random jitter in the processing of the message is the reason for the seen mapping sensitivity. A conclusive mapping cannot be drawn from the results.

Due to the `DG`'s high message output of 2048 `DataPacketDesc` messages per received

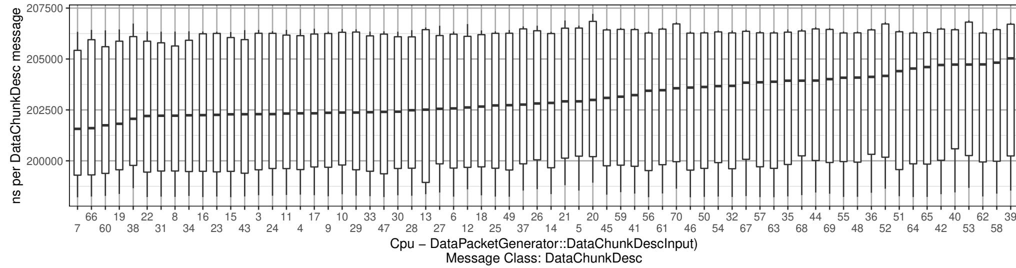


Figure B.2: Time-Per-Message (TPM) in *ns* per `DataChunk` message depending on the mapping of the `DGs`.

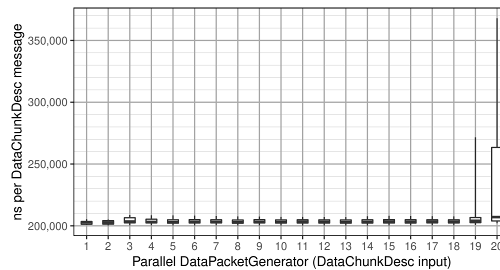


Figure B.3: Contention TPM of the `DataChunkDesc`$_{\mathtt{In}}$ input of the `Data-Packet - Generator (DG)`.

`DataChunkDesc` message, the contention measurements are conducted with 8 consumer-stages. The results for the contention measurements are shown in figure B.3 for the (inconclusive) best case mapping. As the `DG` is purely CPU bound and does not compete for any resources but the message passing subs-system, the `DG` does not show any contention effects. However, the dependency of the DG on a sufficient amount of consumers can be seen clearly, as the TPM stays constant for up to 18 parallel DGs, but once all message-consumers are utilized the TPM starts increasing significantly due to the back pressure flow control.

B.2.3 Acknowledgement Processor (AP)

The behavior of the `AP` is described by the interface and the state-machine shown in figure B.4. The `AP` combines several tasks. However, retransmitting missing packets is its main responsibility. Additional tasks are: Requesting acknowledgements from the receiver,

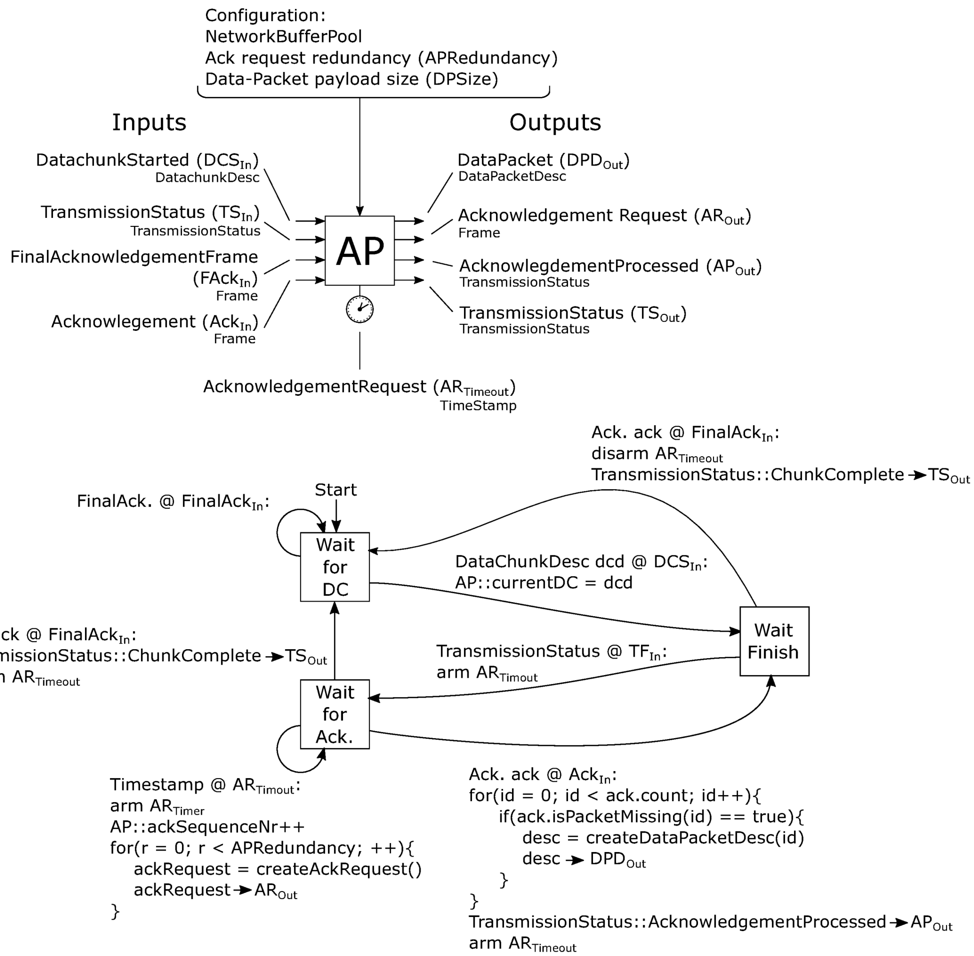


Figure B.4: The interface and state machine of the `Acknowledgement Processor (AP)`. The `AP` is used to process segmented acknowledgements and retransmit any missing data packet it encounters.

as well as notifying other stages about the complete transmission of a datachunk. The AP is configured with a bufferpool of network frames, the amount of redundantly sent AckRequestFrame messages (APRedundancy), and the data packet payload size (DPSize). The input behaviour is as follows:

The AP starts in the *WaitForDC* state in which it waits for a DataChunkDesc. In this state all messages but a DataChunkDesc message received at $\texttt{DataChunkStarted}_{\texttt{In}}$ ($\texttt{DCS}_{\texttt{In}}$) input are ignored. Upon receiving a DataChunkDesc message at the $\texttt{DCS}_{\texttt{In}}$ input, the DataChunkDesc is stored and the AP switches into the *WaitFinish* state.

In the *WaitFinish* state the AP waits for a message that signals that a prior (re)-transmission of data packets or a whole data chunk was finished. This happens either by a TransmissionStatus message received at the $\texttt{TransmissionStatus}_{\texttt{In}}$ ($\texttt{TS}_{\texttt{In}}$) input, or a FinalAckFrame received at the $\texttt{FinalAckFrame}_{\texttt{In}}$ ($\texttt{FAck}_{\texttt{In}}$) input, that states that the current datachunk was completely transmitted. In the case of a received Transmission-Status::AllPacketsTransmittedOnce or TransmissionStatus::AcknowledgementPro-cessed message at the $\texttt{TS}_{\texttt{In}}$, the AP arms the $\texttt{AR}_{\texttt{Timeout}}$ and switches into the *Wait For Ack* state. In the case a FinalAckFrame is received at the $\texttt{FAck}_{\texttt{In}}$ input, the AP switches into the *Wait For Data Chunk (DC)* state, disarms the $\texttt{AR}_{\texttt{Timeout}}$, and sends a Transmission-Status::DataChunkComplete per its $\texttt{TransmissionStatus}_{\texttt{Out}}$ ($\texttt{TS}_{\texttt{Out}}$) output.

In the *Wait For Ack* state, the AP waits for AckFrame, FinalAckFrame, and Timestamp messages, where FinalAckFrame messages are handled the same as in the *Wait Finish* state. Upon receiving a Timestamp message at the $\texttt{AR}_{\texttt{Timeout}}$, APRedundancy $\times$ AckRequestFrame are sent over the $\texttt{AcknowlegdementRequest}_{\texttt{Out}}$ ($\texttt{AR}_{\texttt{Out}}$) output and the $\texttt{AR}_{\texttt{Timeout}}$ is rearmed. When an AckFrame is received at the $\texttt{Acknowledgement}_{\texttt{In}}$ input, the acknowledgement is scanned for missing packets. For each missing packet a DataPacketDesc is created and sent via the $\texttt{DataPacketDesc}_{\texttt{Out}}$ output. After sending all missing packets, a TransmissionStatus::AcknowledgementProcessed message is sent per $\texttt{AP}_{\texttt{Out}}$.

Performance Characteristics

The AP has four inputs and one timeout. While the measurement of the DataChunk-$\texttt{Started}_{\texttt{In}}$, TransmissionStatus¸ and $\texttt{AckRequest}_{\texttt{Timeout}}$ inputs is straight forward, the $\texttt{Acknowledgement}_{\texttt{In}}$ and $\texttt{FinalAcknowledgement}_{\texttt{In}}$ inputs need a special measurement

setup because their inputs expect network `Frame` messages that have to be emulated. During the measurements, the network interfaces are simulated for incoming network-frames, i.e., the network-frame's memory addresses are set up as they would have been by the network interface. Additionally, the costs for freeing a network-frame are simulated by waiting for 144 cycles (as it would take to free a network-frame) instead of actually freeing a network frame. Furthermore, the framework's timeout mechanism is disabled in order to avoid unwanted influence by timeouts.

Sensitivity to the mapping

The dominant input of the `AP` is the $\texttt{AckFrame}_{\texttt{In}}$ input, in combination with the `AckFrame (1500 mp/2 ap)` message class, i.e., 1500 missing packets in two ack-packets.

In order to measure the $\texttt{AckFrame}_{\texttt{In}}$ input, the acknowledgment frames have to be prepared with the corresponding number of missing packets and valid ack-packets, as classified during the simulation. The missing packets are evenly distributed over all valid ack-packets. Before the $\texttt{AckFrame}_{\texttt{In}}$ input can be measured, the `AP` has to be set up with a valid `DataChunkDesc`, and then switched into the *Wait For Ack* state by sending a `TransmissionStatus::AllPacketsTransmittedOnce` message.

The $\text{TPM}^{\texttt{AckFrame (1500 mp/2 ack-packets)}}_{\texttt{AckFrame}_{\texttt{In}}}$ sensitivity to the mapping is shown in figure B.5. The results show a minor increase of the median TPM from 401041 *ns* to 401568 *ns* , which means that the `AP` is not sensitive on its mapping to a certain Central Processing Unit (CPU). Additionally, the results also show a high interquartile range, which is caused most probably by different memory-access costs due to the location of the received network

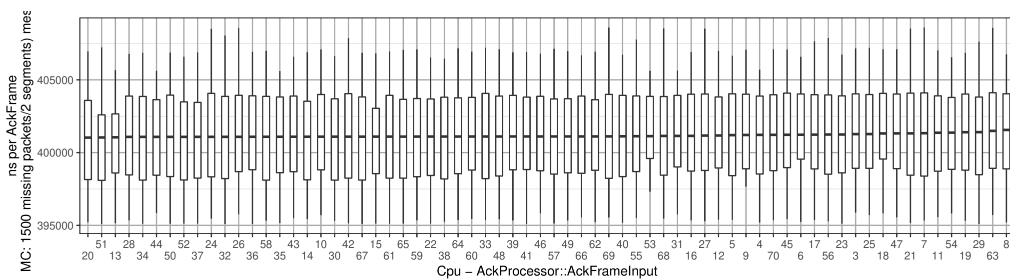


Figure B.5: TPM in *ns* per `AckFrame (1500 mp/2 ack-packets)` message depending on the mapping of the `AP`s.

frame. Consequently, the AP is sensitive to the mapping, however, on a per-message basis. Since the APs will have to process network frames whose payload are possibly located on all memory controllers, no conclusive answer on the mapping-sensitivity can be given.

Acknowledgement input (Ack_{In}):

The $AckFrame_{In}$ input receives messages of 10 different classes, depending on the number of missing packets and valid ack-packets, as discussed earlier for the simulation results. The following performance analysis shows only the results for AckFrame (50 missing packets (mp)/1 ack-packets (ap)), i.e., 50 missing packets in 1 ack-packet, and AckFrame (1500 mp/2 ap), i.e., 1500 missing packets in two ack-packets. However, the analysis was, of course, performed for all message classes.

In order to avoid that the message consumers become a bottleneck, seven parallel consumers will be used during the measurements. The TPMs for parallel processing are shown in figure B.6 for the message classes AckFrame (50 mp/1 ack-packet) and AckFrame (1500 mp/2 ack-packets). The results show that up to 20 AP are not able to saturate 12 consumers, i.e., the results are not affected by contention due to the back-pressure flow-control.

The median TPMs for both message classes follow the same pattern, however, have different ranges. The results for the message class AckFrame (1500 mp/2ap) are in a range from 441114 ns to 450489 ns , whereas the range for the message class AckFrame (50 mp/1ap) is from 23478ns and 26392ns . The results are nearly constant, with an exception for just 1 AP for which the TPM is lower than for 2+ AP. Taking only actual parallel execution into account, i.e., parallelization counts of greater than 1, the range shrinks to medians between 438580 ns and 441664 ns (AckFrame (1500 mp/2ap)) and 25025 ns and 25212 ns (AckFrame (50 mp/2ap)), respectively. This means, no actual contention exists.

FinalAcknowledgement input ($FinalAck_{In}$):

The $FinalAcknowledgement_{In}$ ($FinalAck_{In}$) input receives messages of two classes: FinalAckFrame (Valid) and FinalAckFrame (Invalid). In the case the FinalAckFrame is

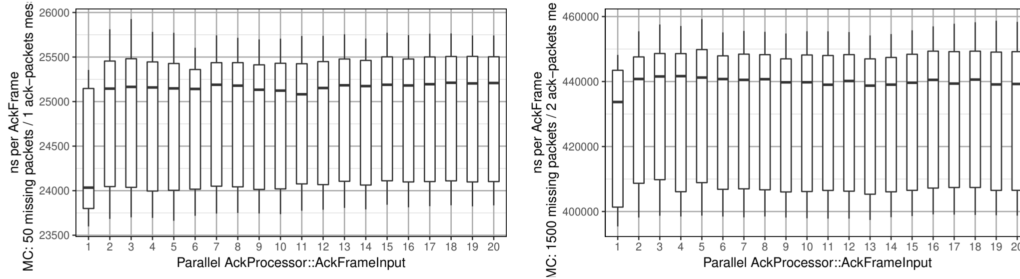


Figure B.6: Performance Characteristics of the `AckFrame` input of the `Acknowledgement Processor (AP)` for the message classes: `AckFrame (50 mp/1 ap)` (left), and `AckFrame (1500 mp/2 ap)` (right).

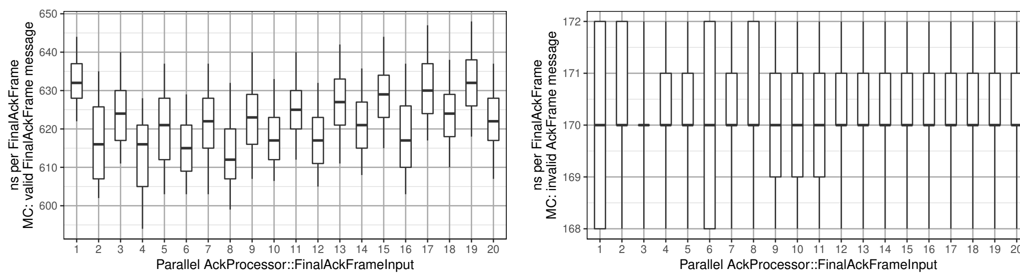


Figure B.7: TPM of the `FinalAck` input of the `Acknowledgement Processor (AP)` for valid (left) and invalid (right) `FinalAckFrames`.

valid, i.e., it belongs to the current datachunk and the `AckFrame`'s sequence number is not outdated, the `AP` forwards the current datachunk via the $\texttt{DataChunkDone}_{\texttt{Out}}$ output and switches into the *Wait for DC* state. In order to measure the $\texttt{FinalAck}_{\texttt{In}}$ input for the message class `FinalAckFrame (Valid)`, the `AP` has to be brought into the *Wait For Ack* state before each measurement.

The TPM for parallel execution of the `AP` is shown in figure B.7 (left). The measured median $\text{TPM}^{\texttt{FinalAckFrame (Valid)}}_{\texttt{FinalAckFrame}_{\texttt{In}}}$ span from 612ns to 633 ns per message, whereas no influence of the parallelization on the TPM is seen.

The measurement for the `FinalAckFrame (Invalid)` message class does not need any prior setup. The `AP` is initialized in the *Wait For Datachunk*, which will render any incoming `FinalAck` message invalid. The resulting medians (see B.7/right) are constant at 170ns , i.e., the parallelization does not have any impact on the TPM.

$\texttt{DataChunkStarted}_{\texttt{In}}$ ($\texttt{DCS}_{\texttt{In}}$) input and $\texttt{TransmissionStatus}_{\texttt{In}}$ ($\texttt{TS}_{\texttt{In}}$) input:

The $\texttt{DataChunkStarted}_{\texttt{In}}$ input and the $\texttt{TransmissionStatus}_{\texttt{In}}$ input are used to alter the state of the AP, such as bringing the AP into the *Wait For Ack* state in which it receives and processes incoming acknowledgements.

In the case a DataChunkDesc message is received at the $\texttt{DCS}_{\texttt{In}}$ input, the DataChunkDesc is stored and the AP switches to the *Wait Finish* state. Since this is done for any incoming DataChunkDesc message regardless of the current state, no setup of the AP is necessary. The $\text{TPM}_{\texttt{DCS}_{\texttt{In}}}^{\texttt{DataChunkDesc}}$ for the parallel execution are shown in figure B.8 (left). The results show low TPMs that are not affected by higher parallelization counts. That was to be expected, because changing the internal state and storing the new DataChunkDesc in a local variable is neither costly, nor exists any competition for shared resources.

The $\texttt{TS}_{\texttt{In}}$ input is used to change the AP's state from *Wait Finish* into *Wait For Ack*, as long as the received TransmissionStatus::AllPacketsTransmittedOnce message belongs to the currently stored DataChunkDesc. Therefore, in order to measure the $\text{TPM}_{\texttt{TS}_{\texttt{In}}}^{\texttt{TransmissionStatus}}$, the AP has to be set up with a valid DataChunkDesc by using the $\texttt{DCS}_{\texttt{In}}$ input. The results are shown in figure B.8 (right). The results show slightly higher TPMs, which are caused by forwarding the incoming TransmissionStatus message. Again, the results were to be expected due to the low processing costs of the performed task.

$\texttt{AcknowledgementRequest}_{\texttt{Timeout}}$ ($\texttt{AR}_{\texttt{Timeout}}$)

The $\texttt{AcknowledgementRequest}_{\texttt{Timeout}}$ is technically not an input but a timeout. However, each timeout can also be used as an input, and be measured as such. The $\texttt{AR}_{\texttt{Timeout}}$ is used to trigger the sending of an AckRequestFrame to the receiver in the case the AP is in the *Wait For Ack* state. In order to measure the TPM for a timeout, the AP is set up with a DataChunkDesc and a TransmissionStatus::AllPacketsTransmittedOnce message. The timeout itself is triggered manually by sending a Timestamp message to the $\texttt{AR}_{\texttt{Timeout}}$. Additionally, in order to prevent contention at the message consumer due to freeing the outgoing AckRequestFrames, this benchmark uses 8 consumers. The $\text{TPM}_{\texttt{AR}_{\texttt{Timeout}}}^{\texttt{Timestamp}}$ is shown in figure B.9.

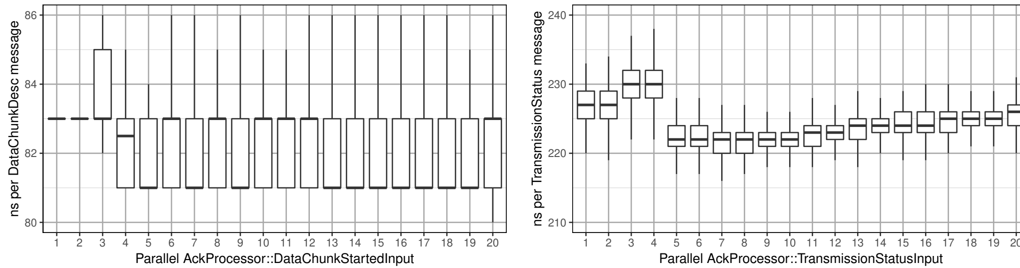


Figure B.8: TPM of the `DataChunkStarted` input (left) and the `TransmissionStatus` input (right) of the `Acknowledgement Processor (AP)`.

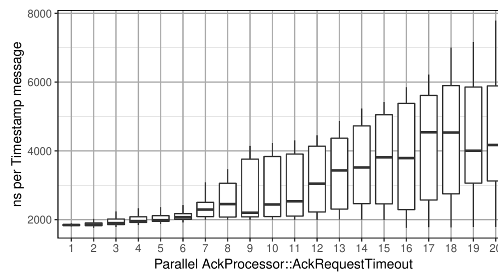


Figure B.9: Performance Characteristics of the `AcknowledgementRequest Request` timeout.

The parallelization shows a clear contention effect on the TPM when 6 or more `AP` process `Timestamp` messages in parallel. This is due to contention on the message passing sub system and the `NetworkOutBuffer` pools which have to provide 2 network-buffers (2 because of `AckRequest` redundancy) for each incoming `Timestamp` message. Furthermore, these network-frames are freed immediately afterward by one of the consumers which increases the contention. The additional work done by the consumers lead to over-utilization that stall the processing due to the back-pressure flow control. However, the analysis stated that only one `AP` is needed, therefore, it was not necessary to repeat the measurements.

B.2.4 Data-Packet Combiner (DC)

The `Data-Packet Combiner (DC)` is responsible for copying data packets from a data frame into the destination position within the datachunk. The interface and EFSM are shown in figure B.10.

When a data frame is received at the $\texttt{DataFrame}_{\texttt{In}}$ ($\texttt{DF}_{\texttt{In}}$) input, the frame is checked whether it belongs to the current datachunk, if not the whole frame is ignored. If the frame is valid, the payload of the individual data packets is copied into the datachunk buffer and the data packet ids are stored in `AckDesc` messages. When the frame was completely processed, the `AckDesc` message is sent via the $\texttt{AckDesc}_{\texttt{Out}}$ output.

During the processing of a `DataFrame` message, the `DC` tries to receive waiting `DataFrame` messages in order to avoid uncontrolled frame drops in case of an over-utilization. In the case another `DataFrame` was received, the new frame is stored in a local waiting queue, only if also this queue is full, the frame is dropped by the `DC`. The dropping mechanism is disabled during the following measurements.

When a `DataChunkDesc` is received at the $\texttt{DataChunkDesc}_{\texttt{In}}$ ($\texttt{DC}_{\texttt{IN}}$) input, it is stored as a buffer for incoming Frames. Any previously received datachunk buffer is discarded.

Sensitivity to the Mapping

The `DC`'s dominant input is the $\texttt{DF}_{\texttt{In}}$ input. The $\texttt{DF}_{\texttt{In}}$ has 9 message classes, depending on the number of valid data packets in the network-frame, whereas a full frame with 8 valid data packets causes the highest processing costs. Consequently, the mapping sensitivity is measured with the help of the $\texttt{DF}_{\texttt{In}}$ and the message class `DataFrame (8 DPs)`.

In order to measure the TPM for the $\texttt{DF}_{\texttt{In}}$ input, each `DC` has to be set up with a valid `DataChunkDesc`. Furthermore: In order to mimic the situation during an actual transmission, the individual `DataFrames` have to be filled with valid data packets before

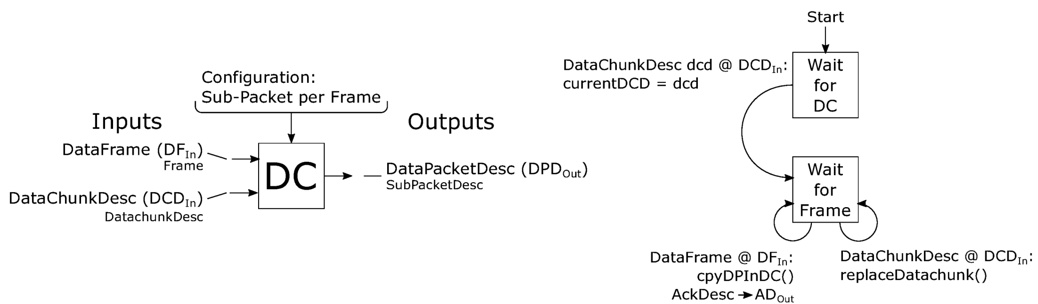


Figure B.10: The interface and EFSM of the Data-Packet Combiner.

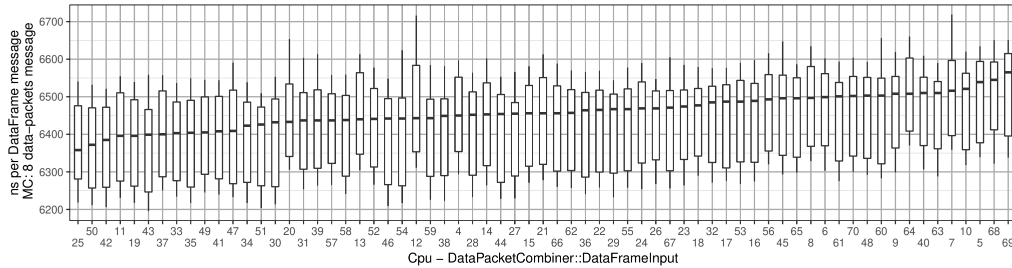


Figure B.11: Time-Per-Message (TPM) in ns per `DataFrame (8 DP)` message depending on the mapping of the `DCs`.

sending. The results of the mapping analysis are shown in figure B.11. The results show that the mapping has a minor effect on the processing time as the median TPMs are in a range between min. 6121 ns and max. 6300 ns .

`DataFrame`$_{\texttt{In}}$ input

The contention measurements employ 16 `DataFrame` producers because building a `DataFrame` is expensive, which reduces the maximum message output of the data producer. In the case the message output is to low the benchmark does not create a contention situation and the measurements are maybe faulty.

The measured contention TPMs for the message classes `DataFrame (8 data packets (DP))` (up) and `DataFrame (1 DP)` (down) of the `DC` are shown in figure B.12. The results for the message class `DataFrame (8 DP)` show no sign that the mapping has an impact on the TPM. However, they also show clearly that the `DC` suffers from contention, which was to be expected because all parallel `DCs` compete for the memory controllers.

The results for the message class DataFrame (1 DP) show the contention effect only up to 6 parallel `DCs`. However, this is no reason for celebration, as it is a consequence of the used benchmark. The message producers need more time to build a `DataFrame` than 7 `DCs` need to process them, consequently, the latency between two `DataFrame (1 DP)` messages increases until no contention exists anymore. Generally, this is a reason to increase the number of producers, however, in the case of the 40 Gbit/s example (see 4.6) five parallel `DC` are sufficient to fulfill the soft real-time requirements. Consequently, there is no need for contention results, for parallelization counts higher than 5. Furthermore,

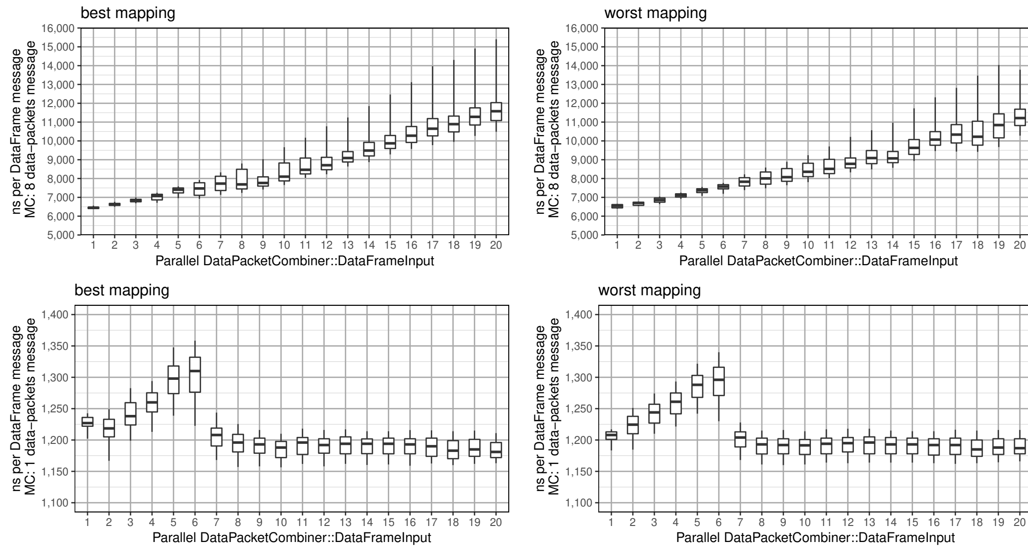


Figure B.12: TPM for the $\mathtt{DC.DataFrame}_{\mathtt{In}}$ input in *ns* under parallel execution. The best mapping is shown left and the worst mapping is shown right.

the simulation for the overhead measurements in section 4.7.1 identified only `DataFrames` of the message class DataFrame (8 DP) for which the latency problem does not exist.

$\mathtt{DataChunkDesc}_{\mathtt{In}}$ input

Measuring the TPM for the $\mathtt{DataChunkDesc}_{\mathtt{In}}$ ($\mathtt{DCD}_{\mathtt{IN}}$) input is straightforward and does neither need a special setup of the `DataChunkDesc` message, nor any setup of the `DC`. The results of the contention measurements are shown in figure B.13. As expected (the `DataChunkDesc` message is only stored), no contention effects or influence of the mapping can be seen.

B.2.5 Acknowledgement Generator (AG)

The `AG` is configured with the number of data packets per datachunk, the number of acknowledgement segments per `AckFrame`, and a buffer pool for network frames. The `AG`

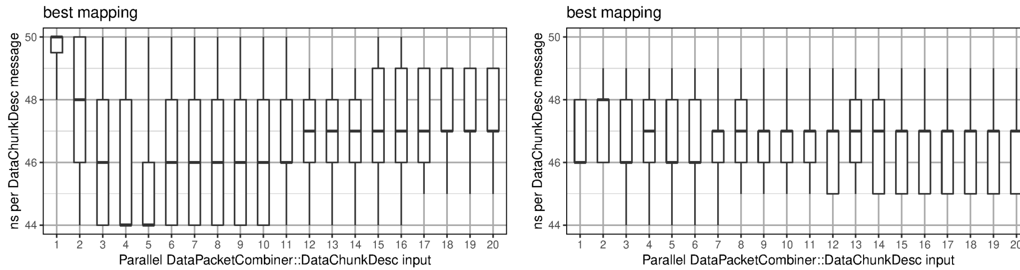


Figure B.13: The TPM for the $\texttt{DataChunkDesc}_{\texttt{In}}$ input given the best mapping (left) and the worst mapping (right).

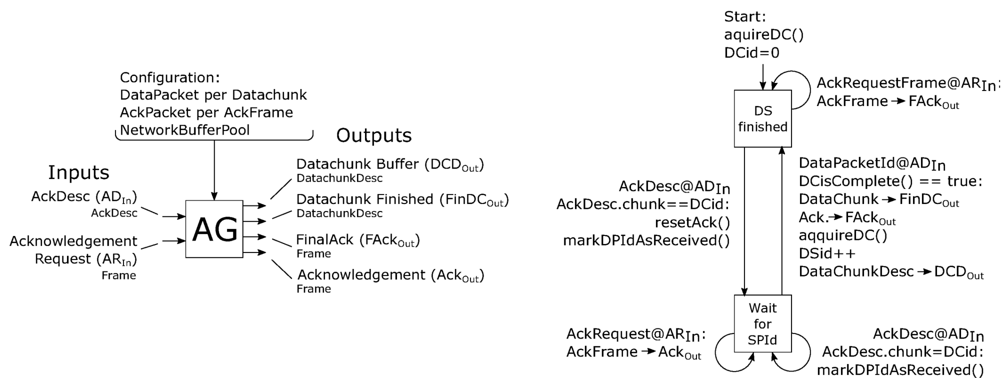


Figure B.14: The interface and behaviour EFSM of the `Acknowledgement Generator (AG)`.

is responsible for tracking the correctly received data packets, preparing and distributing datachunk buffers for the next datachunk, and answering to acknowledgement-requests with acknowledgements.

On initialization the next expected datachunk number is set to 0, and the initial `DataChunkDesc` messages, referring to the corresponding buffer, are distributed per $\texttt{DataChunkDesc}_{\texttt{Out}}$ output. Afterward, the `AG` switches in the *Datachunk (DC) finished* state, i.e., it waits for the first `AckDesc` message of the new datachunk.

Upon receiving the first `AckDesc` message at the $\texttt{AckDesc}_{\texttt{In}}$ input that belongs to the new datachunk, the `AG` switches into the *Wait For AckDesc (AD)* state and marks the data packets, stated in the `AckDesc` message, as received. Additionally, the data packet counter is initialized.

The AG is now in the *Wait for AD* state and waits for further AckDesc messages, as well as AckRequestFrame messages. For each received AckDesc message that is received at the $\texttt{AckDesc}_{\texttt{In}}$ input, the AG checks whether the data packet was already marked as received. The new data packets are marked in the acknowledgement and the data packet counter is increased.

In the case the data packet counter reaches the number of data packets per datachunk (DPPerDC), a FinalAckFrame is sent via the $\texttt{FinalAck}_{\texttt{Out}}$ output. Additionally, the current DataChunkDesc of the finished datachunk is forwarded via the $\texttt{DataChunkFinished}_{\texttt{Out}}$ output, and a new DataChunkDesc is prepared and distributed via the $\texttt{DataChunkDesc}_{\texttt{Out}}$ output. Finally, the AG switches back into the *DC finished* state. However, the current acknowledgement is kept for future AckRequestFrame messages, until the first AckDesc of the new datachunk arrives.

Upon receiving an AckRequestFrame message at the $\texttt{AckRequest}_{\texttt{In}}$ input, the current acknowledgement is sent no matter the current state of the AG.

Sensitivity to the Mapping

The dominant input of the AG is the $\texttt{AckDesc}_{\texttt{In}}$ input that receives AckDesc messages. Measuring the $\texttt{AckDesc}_{\texttt{In}}$ input is straight-forward because the only requirement is that the AckDesc messages belongs to the current datachunk and that the data packets, referred to by the AckDesc are not yet marked as received. Therefore, the message producer provides unique AckDesc messages for each measurement.

The sensitivity of the AG on its mapping is shown in figure B.15. The median TPM are in a range from 200 ns to 215 ns , i.e., there is no sensitivity to the mapping.

$\texttt{AckDesc}_{\texttt{In}}$ input

The parallel measurements for the $\texttt{AckDesc}_{\texttt{In}}$ input are shown in figure B.16. As to be expected no difference between best- and worst-case mapping can be identified. Furthermore, there is no measurable contention under parallel execution. That also was expected

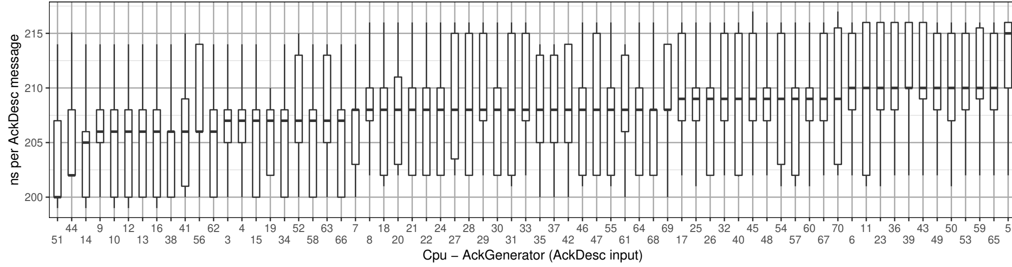


Figure B.15: Processing time in *ns* per `AckDesc` message depending on the mapping of the `AGs`.

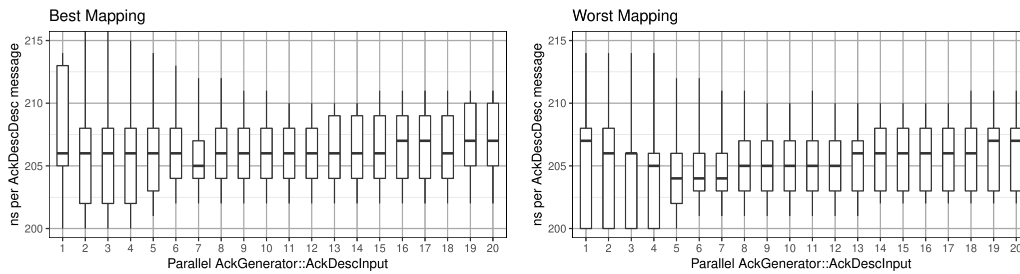


Figure B.16: Contention measurements for the `AckDesc`$_{\mathtt{In}}$ input for the best mapping (left) and the worst mapping (right).

since marking newly received data packets in the acknowledgement is done on the local acknowledgement that is most probably available in the local cache.

`AckRequest`$_{\mathtt{In}}$ input

Two message classes were identified for the `AckRequest`$_{\mathtt{In}}$ input: `AckRequest (Invalid)` and `AckRequest (Valid)`. Measuring the TPM for the message class `AckRequest (Invalid)` does not need any setup or special attention to the `AckRequest` message. All incoming `AckRequest` messages with an outdated sequence number or `AckRequests` that do not belong to the current datachunk are considered invalid. The results of the measurement for the message class `AckRequest (Invalid)` are shown in figure B.17. The results show no difference between best case and worst case mapping and no sign of contention.

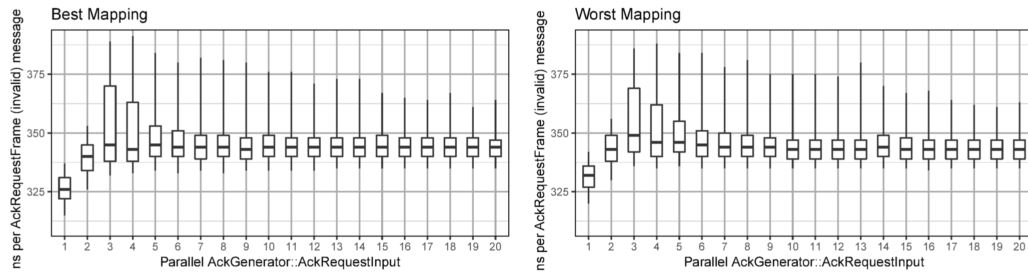


Figure B.17: Performance Characteristics of the $\texttt{AckRequestFrame}_{\texttt{In}}$ input of the `Acknowledgement Generator (AG)` for the message class `AckRequestFrame (Invalid)`.

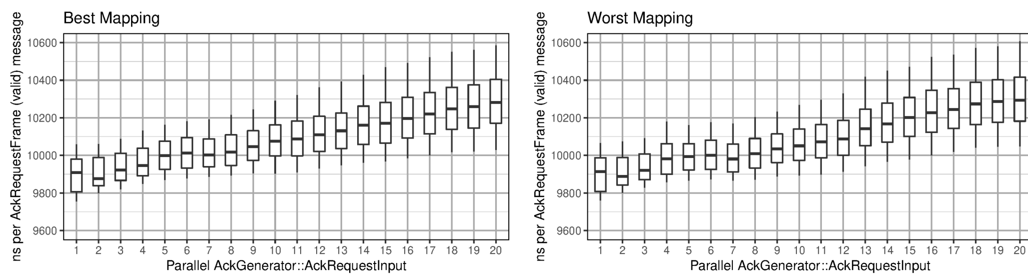


Figure B.18: Performance Characteristics of the $\texttt{AckRequestFrame}_{\texttt{In}}$ input of the `Acknowledgement Generator (AG)` for the message class `AckRequestFrame (valid)`.

In order to measure the parallel TPM for the message class `AckRequest (Valid)`, the acknowledgement requests have to refer to the current datachunk and the sequence number must be increased for each `AckRequest` message. The results for the measurements are shown in figure B.18. The results show no influence of the mapping as the TPMs for the best case mapping and the TPMs for the worst case mapping are basically identical. The contention effect that leads to increasing TPMs is caused by allocating network frames from the frame buffer pools.

B.2.6 Performance measurements for The DA with incoherent memory

Figure B.19 shows the contention measurements when coherent/incoherent memory is used as payload for the `DataPacketDesc` messages that are processed by the `Data-Packet Aggregator (DA)`. The spikes that were seen for the contention measurements (see 4.3.2) disappeared and the TPM decreased. However, since the Direct Memory Access (DMA)

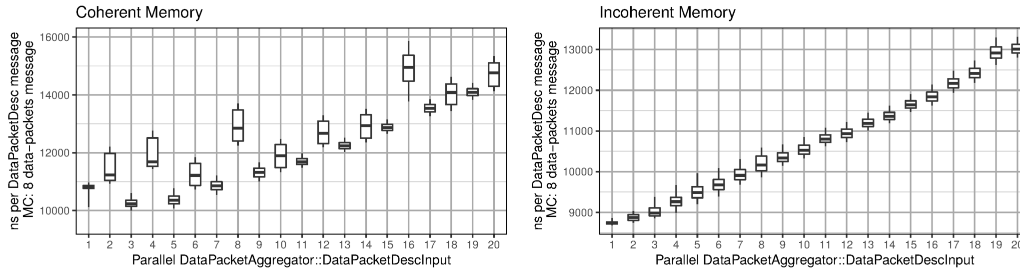


Figure B.19: Performance Characteristics of the $\texttt{DataPacketDesc}_{\texttt{In}}$ input of the `DA` for the message class `DataPacketDesc (8 DPs)` with coherent (left) and (incoherent) memory as payload for the `DataPacketDescs`.

interfaces of the TileGX72 manycore board rely on coherent memory, the incoherent memory could not be used.

B.2.7 Service Stages

The following stages provide services that do not belong to the protocol processing but provide simple management features. Furthermore, none of these stages have to be parallelized. Therefore, a detailed description and analysis is omitted.

`Output Channel (OUT-CH)`

The `OUT-CH` has an interface with one input and one output (see figure B.20a). Its only task is to write the configured virtual channel id into the `VirtualChannel` header of the frames it receives at its $\texttt{Frame}_{\texttt{In}}$ input, and to afterward forward the frames via its $\texttt{Frame}_{\texttt{Out}}$ output.

`Input Channel (IN-CH)`

The `IN-CH` (see figure B.20b) has an input that receives network frames and an array of outputs for the protocol pipelines. The `IN-CH` stage is responsible for demultiplexing the physical channel depending on the virtual channel id that is stated in the `VirtualChannel` header of the received `Frame`. The `IN-CH` needs no configuration because all necessary information are stated in the received frames.

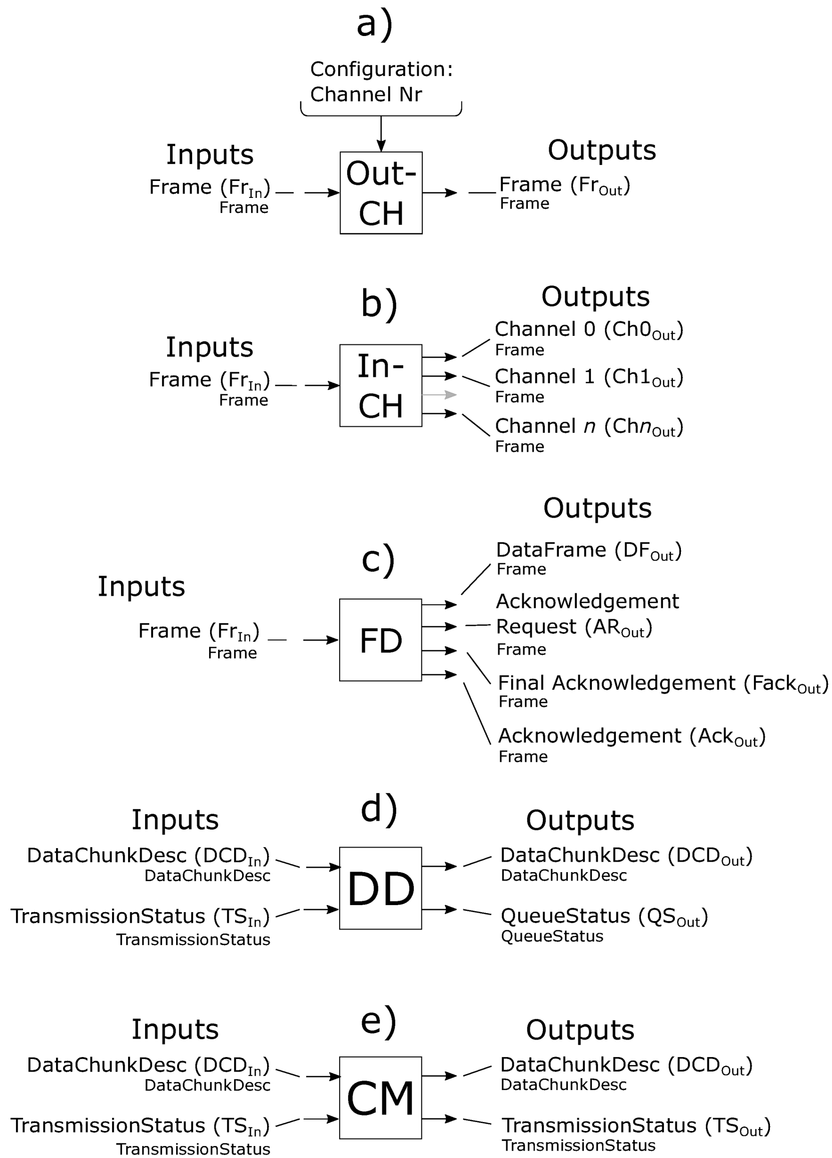


Figure B.20: The interfaces of the a) `Output Channel` (`OUT-CH`), the b) `Input Channel` (`IN-CH`), c) `Frame Distributor` (`FD`), d) `Datachunk Distributor (DD)`,and e) `Channel Manager (CM)`.

Frame Distributor (FD)

The FD needs no configuration because all necessary information are stated in the network-frame. The FD's interface (figure B.20c) has one input and four outputs, one fore each frame type. Upon receiving a valid Frame on its $\texttt{Frame}_{\texttt{In}}$ input, the frame is forwarded to the specified output.

DataChunk Distributor (DD)

The DD (figure B.20d) is responsible for the flow control between host and processing engine. This is done by buffering datachunks that are received at the $\texttt{DataChunkDesc}_{\texttt{In}}$ ($\texttt{DCD}_{\texttt{In}}$) input when the processing engine is not yet ready for a new datachunk.

In order to determine whether the processing engine is ready, the DD receives TransmissionStatus messages on its $\texttt{TransmissionStatus}_{\texttt{In}}$ ($\texttt{TS}_{\texttt{In}}$) input and monitors the status of the transmission. Upon receiving a TransmissionStatus::WaitingForDatachunk message, the oldest buffered datachunk is sent by using the $\texttt{DataChunkDesc}_{\texttt{Out}}$ ($\texttt{DC}_{\texttt{out}}$) output. Additionally, it distributes the current QueueStatus, i.e, the number of waiting datachunks, of the datachunk queue via the $\texttt{QueueStatus}_{\texttt{Out}}$ ($\texttt{QS}_{\texttt{out}}$) output.

Channel Manager (CM)

The CM (figure B.20e) is responsible for selecting a virtual channel for a datachunk that is received at the $\texttt{DataChunkDesc}_{\texttt{In}}$ input. In order to determine the status of the channels, the CM monitors the status of all connected protocol pipelines with the help of incoming TransmissionStatus messages. Once, a protocol pipeline (i.e., virtual channel) is free, the CM sends a TransmissionStatus::WaitForNewChunk message via its $\texttt{TransmissionStatus}_{\texttt{Out}}$ ($\texttt{TS}_{\texttt{Out}}$) output. Additionally, the CM stores the DataChunkDescs that are currently processed by a protocol pipeline, and frees the corresponding datachunks upon receiving the TransmissionStatus::WaitForNewChunk message.

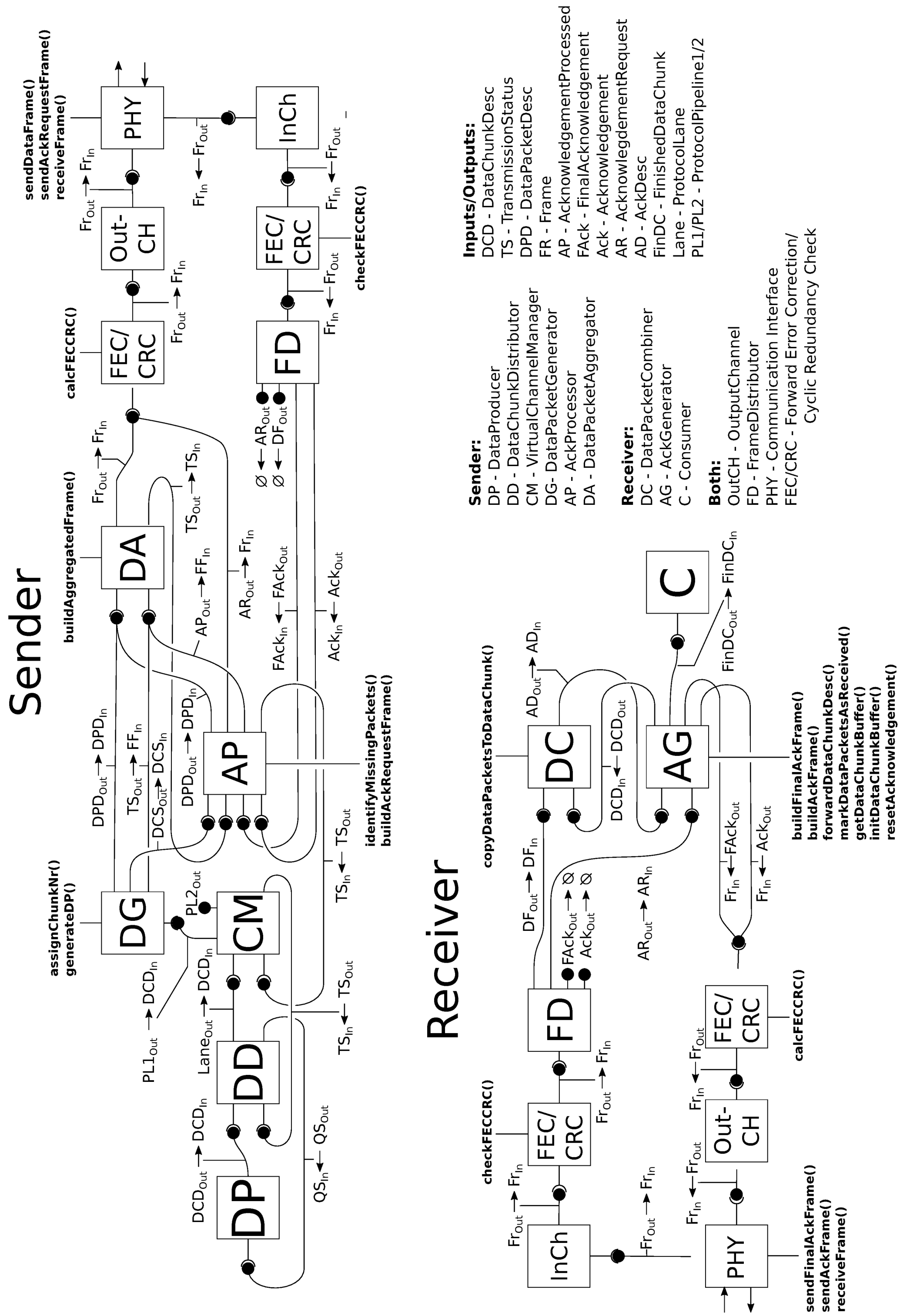


Figure B.21: The fully connected processing engine.

B.3 Full Protocol Processing Engine Description

The `Data Producer (DP)` represents the host and allocates buffers and fills the buffers with data to be transmitted. These buffers are passed to the `DataChunk Distributor (DD)`, which provides a buffer-queue that contains datachunks that have to be transmitted. The `DD` informs the `DP` about the fill-level of the queue. Each buffer in the queue is passed to the `Channel Manager (CM)` which chooses a free virtual channel for that buffer. For that purpose, the `CM` notifies the `DD` when a virtual channel is free to process the next datachunk.

The actual protocol processing starts with the `Data-Packet Generator (DG)` that receives a `DataChunkDesc` and cuts the corresponding buffer into `DataPackets` and forwards the individual data packets to the `Data-Packet Aggregator (DA)`. After cutting the datachunk into pieces, the `DG` notifies the `DA` that all data packets were forwarded. Additionally, the `DG` notifies the `Acknowledgement Processor (AP)` what datachunk is currently processed. The `DA` receives the individual data packets and aggregates them into a network frame. Upon receiving the notification from the `DG` (that all data packets are processed), the `DA` finishes the current network frame and forwards the notification to the `AP`, which then knows that all packets of the current datachunk went through the `DA`. Consequently, it sends an `AcknowledgementRequest` and switches into the retransmission mode in which it accepts incoming acknowledgments. Before any frame is transmitted by the `Communication (COM)` interface, the `Forward-Error-Correction/Cyclic-Redundancy--Check (FEC/CRC)` stage processes the frame and the `Output Channel (OutCH)` assigns the virtual channel number to the `VirtualChannelHeader`.

On the receiver side, the `Input Channel (InCH)` inspects the `VirtualChannelHeader` of the network frame and forwards the frames to the corresponding protocol pipeline. Afterward, each incoming network frame is checked for errors that are corrected if possible by the `FEC/CRC` stage. The corrected network frame is then passed to the `Frame Distributor (FD)`. The `FD` reads the frame-type and forwards `DataFrames` to the `Data-Packet Combiner (DC)` and `AcknowledgementRequests` to the `Acknowledgement Generator (AG)`. The `DC` copies the payload from the individual data packets into the current datachunk-buffer, which was received from the `AG`. The IDs of the data packets are passed to the `AG` that marks each of the packets as correctly received. In case all packets are received, the `AG` forwards a `FinalAcknowledgement` to the `FEC` stage and acquires a new datachunk buffer that is given to the `DC`. Upon receiving an `AcknowledgementRequest` the `AG` answers with the current `Acknowledgement`.

After being forwarded to the corresponding protocol pipeline (`InCH`) and checked by the sender sides `FEC/CRC` stage, this `FinalAcknowledgement/Acknowledgement` is finally received by the `FD` on the sender side. The sender-side `FD` reads the frame-type and forwards `AcknowledgementFrames` to the `AP`'s acknowledgement input and `FinalAcknowledgements` to the final-acknowledgement input. Upon receiving a `FinalAcknowledgement`, the `AP` notifies the `CM`, which frees the corresponding datachunk buffer, marks the virtual channel as free and notifies the `DD`. The `DD` can now forwards the first waiting datachunk to the `CM`. In case an `Acknowledgement` with missing data packets was received, the missing packets are forwarded by the `AP` to the `DA`. After all missing data packets from the `Acknowledgement` were transmitted, the `AP` notifies the `DA` that it should finish the current frame. This notification is sent back to the `AP` which in turn requests a new acknowledgement.

Time → Channel ↓	0ms	10ms	20ms	30ms	40ms
Channel 0	2.3×10^{-4}	1.4×10^{-4}	9.5×10^{-5}	7.1×10^{-4}	7.0×10^{-4}
Channel 1	2.4×10^{-2}	1.5×10^{-4}	3.6×10^{-3}	2.1×10^{-1}	8.0×10^{-4}
Channel 2	8.5×10^{-3}	9.6×10^{-5}	1.5×10^{-5}	6.0×10^{-5}	3.9×10^{-4}
Channel 3	7.9×10^{-6}	8.8×10^{-5}	1.0×10^{-3}	9.5×10^{-5}	4.2×10^{-4}
Channel 4	9.2×10^{-4}	9.3×10^{-5}	4.0×10^{-5}	9.9×10^{-4}	5.0×10^{-5}
Channel 5	2.8×10^{-4}	3.0×10^{-4}	2.1×10^{-3}	9.0×10^{-4}	5.0×10^{-4}
Channel 6	3.2×10^{-4}	3.0×10^{-5}	9.0×10^{-5}	9.3×10^{-5}	3.8×10^{-4}
Channel 7	8.0×10^{-5}	7.1×10^{-5}	9.7×10^{-4}	7.1×10^{-5}	3.7×10^{-4}
Time → Channel ↓	50ms	60ms	70ms	80ms	90ms
Channel 0	9.0×10^{-4}	5.0×10^{-4}	6.4×10^{-4}	9.0×10^{-4}	5.6×10^{-5}
Channel 1	2.2×10^{-5}	3.2×10^{-5}	9.4×10^{-5}	9.0×10^{-5}	2.7×10^{-5}
Channel 2	5.5×10^{-3}	7.2×10^{-5}	6.4×10^{-3}	2.2×10^{-4}	1.6×10^{-3}
Channel 3	2.1×10^{-4}	2.0×10^{-4}	4.0×10^{-3}	9.0×10^{-5}	5.0×10^{-5}
Channel 4	1.2×10^{-5}	1.7×10^{-5}	3.4×10^{-3}	2.2×10^{-2}	9.6×10^{-5}
Channel 5	9.0×10^{-4}	9.9×10^{-5}	9.0×10^{-5}	4.0×10^{-4}	6.0×10^{-3}
Channel 6	9.0×10^{-4}	4.0×10^{-3}	4.4×10^{-5}	9.0×10^{-5}	9.3×10^{-5}
Channel 7	5.0×10^{-5}	9.2×10^{-5}	9.4×10^{-5}	4.2×10^{-3}	5.6×10^{-2}

Table B.1: Channel BER for the dynamic channel bonding scenario.

List of Figures

1.1 A soft real-time stream processing representation for communication protocols with retransmissions (top). A "smart" network interface card consisting of a manycore processor and Field Programmable Gate Array (FPGA) for the protocol processing (bottom). 3

3.1 Stream processing based protocol implementation (adapted from [78]). . 38
3.2 The refining process for the sender-side processing engine depending on the dataflow of the protocol from the general protocol to a coarse grained protocol separation finally resulting in a fine-grained processing engine. . 39
3.3 Soft real-time requirements for the example processing engine. 40
3.4 Simplified measuring setup for the estimation of the performance characteristics, e.g., necessary processing time per message, of stage `B`. The Input Message Producer sends messages with the maximum possible data rate and the Output Message Consumer receives and consumes the output messages. 41
3.5 Soft real-time requirements and performance characteristics of the processing engine. The `Framing`, `Error-Coding` and `Error-Handling` stages pose bottlenecks because the soft real-time requirements are higher than the measured performance characteristics. 42
3.6 The stream operators manipulate the message stream without concern for the message's content. The possible operators are: a) Stream-Split b) Stream-Duplicate, and c) Stream-Join 43
3.7 Soft real-time requirements and performance characteristics of the processing engine. 44
3.8 Readaptation process for a processing engine: a) When the stages C_1 and C_2 are over-utilized, the a re-adaption adds more stages of type C to the processing engine. b) When the stages C_1 to C_4 are under-utilized, then a readaptation may remove stages of type C. 46

3.9 a) A virtual channel (CH) is used to distinguish the data streams between two processing engines that are using the same communication interface. b) Replacement procedure (adapted from [82]) of a processing engine. Virtual channels are used to multiplex the original processing engine (red) and the new processing engine (green) on the same communication interface. The old processing engine is drained, while the new processing engine is already used for the ongoing transmission in order to avoid disruptions in the protocol processing. 48

4.1 The data structures of the prototype link layer protocol (further developed from [7]). Each frame is segmented into smaller sub-packets of three types: Data-Packets carry data-payload that belongs to a datachunk, Ack-Request-Packets are used to trigger the receiver to (re-)send an aggregated acknowledgment, and Ack-Packets carry parts of an acknowledgment. Since each frame can only contain sub-packets of the same type, super-frames can be used to further aggregate frames of different types. 53
4.2 Coarse-grained EFSM of the sender-side protocol. 55
4.3 Coarse-grained EFSM of the receiver-side protocol. 57
4.4 The sender side processing engine of the presented data link protocol. The processing tasks mentioned in the protocol's sender and receiver EFSM are assigned to the corresponding stages. 58
4.5 The receiver side processing engine of the presented data link protocol. The processing tasks mentioned in the protocol's sender- and receiver EFSM are assigned to the corresponding stages. 59
4.6 The interface of the `Data-Packet Aggregator (DA)`. 62
4.7 The EFSM of the `Data-Packet Aggregator (DA)`. The EFSM describes the behaviour of the `DA` depending on its internal state and the messages received at its inputs. 62
4.8 The results of soft real-time requirement estimation. The processing engine was used in a simulated transmission with a BER of 10^{-5}. 67
4.9 TPM in *ns* per `DataPacketDesc` message with the message class `8 DPs` depending on the mapping of the `DAs`. 74
4.10 Heat-map of the TPM in *ns* per `DataPacketDesc` message with the message class `8 DPs` showing the effect of the memory striping. The lowest TPMs were measured in the case the distance to the memory controllers is similar, i.e., in the middle of the CPU grid. 75

4.11 Performance Characteristics of the DataPacketDesc$_{\texttt{In}}$ input of the DA for the message class DataPacketDesc (8 data-packets), depending on the mapping. 75
4.12 Performance Characteristics of the DataPacketDesc$_{\texttt{In}}$ input of the DA for the message class DataPacketDesc (1 data-packet), depending on the mapping. 75
4.13 TPM of the FinishFrame input of the Data-Packet Aggregator (DA) for the message class TransmissionStatus (Empty Frame) (Best case mapping right and worst case mapping left). 77
4.14 TPM of the FinishFrame input of the Data-Packet Aggregator (DA) for the message class TransmissionStatus (Non-Empty Frame) (Best case mapping right and worst case mapping left). 77
4.15 The analysis focuses on the processing engine subset Data-Packet - Generator (DG), Data-Packet Aggregator (DA), and Acknowledgement Processor (AP). 80
4.16 A possible mapping of the processing engine to the processing hardware. The FEC/CRC stages are offloaded to external FPGAs. 83
4.17 Goodput, throughput, and latency of the adapted processing engine, given a BER of 10^{-5}. The grey line shows the throughput, i.e., the complete channel utilization with transmission and retransmission, the black line shows the goodput per datachunk. The average goodput per transmission is 37.29 Gbit/s. 85
4.18 a) The singlechannel protocol monopolizes the communication-channel for transmission of single datachunk at a time. b) The multichannel protocol interleaves the communication-channel with the transmission of two datachunks simultaneously. 86
4.19 The multichannel protocol interleaves the communication channel with two parallel transmissions. The transmission of a datachunk on one virtual channel is interleaved with the retransmissions of the second virtual channel, therefore fully utilizing the communication channel's capacity. . 86
4.20 Goodput, throughput, and latency of the double-pipeline processing engine given a BER of 10^{-5}. The grey line shows the throughput,i.e., the complete channel utilization with transmission and retransmission, the black line shows the goodput per datachunk. The average goodput per transmission is 39.99 Gbit/s. 87

4.21 Goodput and latency of the communication system with and without attached FPGAs [7]. The stated packet-loss was simulated at the FPGA. 89
4.22 Per-channel capacity in Gbit/s given a data-packet payload size of 1024 Bytes and the BER from table B.1. The BER changes every 10ms, the pattern repeats after 100 ms. The interfaces are ordered descending by their capacity. The interfaces that are framed in the graph are used (channel bonded) by the adapting processing engine. 96
4.23 Throughput and goodput of a static version of the multichannel protocol that was adapted (5× `Data-Packet Aggregator (DA)`, 4×`Data-Packet Combiner (DC)`) for 30 Gbit/s, but was also given access to all channels. The desired data rate for the transmission was 29 Gbit/s. 97
4.24 Throughput and goodput of a static version of the multichannel protocol that uses all eight 10 GbE interfaces and was also adapted (17× `Data-Packet Aggregator (DA)`, 12×`Data-Packet Combiner (DC)`) in order to be able to utilize all interfaces. The desired data rate for the transmission was 29 Gbit/s. 97
4.25 Throughput and goodput of the multichannel protocol that was continuously adapted to the channel conditions. The protocol is adapted and deployed every 10ms, when the changed BER is noted by the `EndpointManager`. The adaptation is based on the minimum number of necessary channels and the resulting theoretical data rate. The desired data rate for the transmission was 29 Gbit/s. 97
4.26 Latency per datachunk of the static multichannel protocol that uses all eight 10 GbE interfaces and was also adapted (17× `Data-Packet - Aggregator (DA)`, 12×`Data-Packet Combiner (DC)`) in order to be able to utilize all interfaces. 98
4.27 Latency of the adapting multichannel protocol. The protocol is adapted and deployed every 10ms, when the changed BER is noted by the `EndpointManager`. The adaptation is based on the minimum number of necessary channels and the resulting theoretical data rate. 98
4.28 The costs of the on-demand readaptation of the processing engine. 99

4.29 The sending host changes the desired data rate over time. This leads to readaptation of the singlechannel protocol as long as the desired data rate is below 70 Gbits/s. In the case, the desired data rate increases over 70 Gbit/s, the whole protocol is replaced with the multichannel version. The goodput-graph is annotated with the number of combined interfaces and the protocol version (e.g., 2s – 2 interfaces/singlechannel, 8m – 8 interfaces, multichannel). 101
4.30 The protocol processing engine is distributed over two devices per endpoint. 103
4.31 The goodput and throughput of a multi device transmission. 104

A.1 a) The host and the Network-Interface-Card (NIC) start with a minimal `ProtocolStub`. After connecting the stubs via a communication interface, such as PCI Express, the host sends the Processing Engine Template Language (PETL) description to the other `ProtocolStub`. b) Upon receiving a PETL description, the `ProtocolStub` patches itself with the new processing engine. 112
A.2 Address information lookup and cross-device message delivery with the help of proxy messages and proxy-stages. The steps are as follows: 1. Configure message. 2. Send lookup-request. 3. Forward lookup request to remote lookup server. 4. Retrieve stage B's address information. 5. Send lookup-answer. 6. Forward lookup answer to proxy B'. 7+. Send message from A to B by using the configured proxy B'. 113
A.3 Transparent zero-copy buffer transport using slotted buffer-pools and a proxy concept. 114

B.1 The interface (i.e., inputs and outputs) and EFSM of the `Data-Packet Generator (DG)` stage. The `DG` is responsible for cutting a datachunk into data packets, and forwarding the corresponding `DataPacketDesc` messages via its $\texttt{DataPacketDesc}_{\texttt{Out}}$ output. 123
B.2 Time-Per-Message (TPM) in *ns* per `DataChunk` message depending on the mapping of the `DGs`. 124
B.3 Contention TPM of the $\texttt{DataChunkDesc}_{\texttt{In}}$ input of the `Data-Packet - Generator (DG)`. 124
B.4 The interface and state machine of the `Acknowledgement Processor (AP)`. The `AP` is used to process segmented acknowledgements and retransmit any missing data packet it encounters. 125

B.5 TPM in *ns* per `AckFrame (1500 mp/2 ack-packets)` message depending on the mapping of the `APs`. 127
B.6 Performance Characteristics of the `AckFrame` input of the `Acknowledgement Processor (AP)` for the message classes: `AckFrame (50 mp/1 ap)` (left), and `AckFrame (1500 mp/2 ap)` (right). 129
B.7 TPM of the `FinalAck` input of the `Acknowledgement Processor (AP)` for valid (left) and invalid (right) `FinalAckFrames`. 129
B.8 TPM of the `DataChunkStarted` input (left) and the `TransmissionStatus` input (right) of the `Acknowledgement Processor (AP)`. 131
B.9 Performance Characteristics of the `AcknowledgementRequest Request` timeout. 131
B.10 The interface and EFSM of the Data-Packet Combiner. 132
B.11 Time-Per-Message (TPM) in *ns* per `DataFrame (8 DP)` message depending on the mapping of the `DCs`. 133
B.12 TPM for the $\texttt{DC.DataFrame}_{\texttt{In}}$ input in *ns* under parallel execution. The best mapping is shown left and the worst mapping is shown right. 134
B.13 The TPM for the $\texttt{DataChunkDesc}_{\texttt{In}}$ input given the best mapping (left) and the worst mapping (right). 135
B.14 The interface and behaviour EFSM of the `Acknowledgement Generator (AG)`. 135
B.15 Processing time in *ns* per `AckDesc` message depending on the mapping of the `AGs`. 137
B.16 Contention measurements for the $\texttt{AckDesc}_{\texttt{In}}$ input for the best mapping (left) and the worst mapping (right). 137
B.17 Performance Characteristics of the $\texttt{AckRequestFrame}_{\texttt{In}}$ input of the `Acknowledgement Generator (AG)` for the message class `AckRequestFrame (Invalid)`. 138
B.18 Performance Characteristics of the $\texttt{AckRequestFrame}_{\texttt{In}}$ input of the `Acknowledgement Generator (AG)` for the message class `AckRequestFrame (valid)`. 138
B.19 Performance Characteristics of the $\texttt{DataPacketDesc}_{\texttt{In}}$ input of the `DA` for the message class `DataPacketDesc (8 DPs)` with coherent (left) and (incoherent) memory as payload for the `DataPacketDescs`. 139
B.20 The interfaces of the a) `Output Channel` (`OUT-CH`), the b) `Input Channel` (`IN-CH`), c) `Frame Distributor` (`FD`), d) `Datachunk Distributor (DD)`,and e) `Channel Manager (CM)`. 140

B.21 The fully connected processing engine. 142

List of Tables

2.1 Communication solutions from the networking community. The reviewed approaches encompass communication protocols, processing frameworks, special network processors, as well as packet processing solutions. The shown categorization was conducted with respect to *Parallelization Paradigm* (SP–Stream Processing, MT–Multi-Threading, PL–Pipelining), whether *Channel Bonding* is supported, *Flexibility* (O–On-Demand, I–On-Initialization, C–Compiletime,/–Not Applicable), and *Offloading* (*S - Streamlined*, D - Disruptive). None of the reviewed concepts fulfilled all of the desired features. [1]Depends on the protocol implementation, however, no concept for parallel processing but parallel threads. [2]Click uses the stream-processing paradigm but not parallel execution. [3]Only the packet parser can be configured. 32

4.1 Packet loss probability, protocol overhead (per datachunk), and lost data (per datachunk), depending on data-packet size for a Bit Error Rate (BER) of 10^{-5}. 64
4.2 Receive and scheduling costs for a message depending on its size. Two message producers are necessary to fully utilize a message consumer. . . 71
4.3 Predicted per-processor-utilization and parallelization ratios of the stages for a desired data rate of 40 Gbit/s and a BER of 10^{-5}. 81
4.4 Predicted processor utilization for a desired data rate of 40 Gbit/s and a bit error rate of 10^{-5} 88
4.5 Protocol overhead of the data-frame for different payload-sizes and maximum raw Ethernet throughput. 91
4.6 The achieved goodput when fully utilizing up to 8×10 GbE interfaces. The overhead is measured without any packet-loss and for different data-packet sizes. Additionally, the table shows the number of `Data-Packet Aggregators (DAs)` and `Data-Packet Combiners (DCs)` that were used for the transmission. 93

4.7 CPUs and COM interfaces used by the *adapting PE* on the sender- and receiver-side, depending on the channel quality. The channel quality changed every 10 ms (please refer to figure 4.22 for the corresponding channel capacities). 99

B.1 Channel BER for the dynamic channel bonding scenario. 145

Bibliography

[1] Alcatel 7652 PDR (datasheet), Alcatel, 2001.

[2] Nokia 7950 Extensible Routing System (Release 19) (datasheet), Nokia, 2019.

[3] DE-CIX first Internet Exchange worldwide to offer 400-Gigabit Ethernet access technology (press release), DE CIX, 2019.

[4] Ethernet Roadmap 2019 (press release), Ethernet Alliance, 2019.

[5] I. Kallfass, F. Boes, T. Messinger, J. Antes, A. Inam, U. Lewark, A. Tessmann, and R. Henneberger, 64 Gbit/s Transmission over 850 m Fixed Wireless Link at 240 GHz Carrier Frequency *Journal of Infrared, Millimeter, and Terahertz Waves*, vol. 36, pp. 221–233, Feb 2015.

[6] T. Nagatsuma, Y. Fujita, Y. Yasuda, Y. Kanai, S. Hisatake, M. Fujiwara, and J. Kani, Real-time 100-Gbit/s QPSK transmission using photonics-based 300-GHz-band wireless link in *2016 IEEE International Topical Meeting on Microwave Photonics (MWP)*, pp. 27–30, Oct 2016.

[7] S. Büchner, L. Lopacinski, J. Nolte, and R. Kraemer, 100 Gbit/s End-to-End Communication: Designing Scalable Protocols with Soft Real-Time Stream Processing in *41st Annual IEEE Conference on Local Computer Networks (LCN 2016)*, (Dubai, United Arab Emirates (UAE)), Nov. 2016.

[8] 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*, pp. 739–748, Sept 2015.

[9] 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), pp. 607–610, Oct. 2015.

[10] I. Vessey and G. Skinner, Implementing Berkeley Sockets in System V Release 4 in *Proceedings of the Winter 1990 USENIX Conference*, pp. 177–193, 1990.

[11] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, An analysis of TCP processing overhead *IEEE Communications Magazine*, vol. 27, pp. 23–29, June 1989.

[12] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier, TCP performance re-visited in *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pp. 70–79, IEEE, 2003.

[13] S. Karlsson, S. Passas, G. Kotsis, and A. Bilas, MultiEdge: An Edge-based Communication Subsystem for Scalable Commodity Servers *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–10, 2007.

[14] P. Grun, Introduction to InifiniBand for End Users tech. rep., InifiniBand.

[15] S. Passas, K. Magoutis, and A. Bilas, Towards 100 Gbit/s Ethernet: Multicore-based Parallel Communication Protocol Design in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 214–224, ACM, 2009.

[16] 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*, ANCS '12, (New York, NY, USA), pp. 27–38, ACM, 2012.

[17] S. McCanne and V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 1993.

[18] 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, pp. 52–58, Apr. 2014.

[19] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (D. Kranzlmüller, P. Kacsuk, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 97–104, Springer Berlin Heidelberg, 2004.

[20] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, High performance RDMA-based MPI implementation over InfiniBand in *ICS*, 2003.

[21] 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*, pp. 36–43, May 2015.

[22] D. Sidler, Z. István, and G. Alonso, Low-latency TCP/IP stack for data center applications *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, 2016.

[23] J. C. Mogul, TCP Offload Is a Dumb Idea Whose Time Has Come in *HotOS*, 2003.

[24] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey, Server Network Scalability and TCP Offload in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 15–15, USENIX Association, 2005.

[25] L. Grossman, Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver in *Ottawa Linux Symposium*, 2005.

[26] P. Druschel, L. L. Peterson, and B. S. Davie, Experiences with a High-Speed Network Adaptor: A Software Perspective in *SIGCOMM*, 1994.

[27] J. C. Mogul and K. K. Ramakrishnan, Eliminating Receive Livelock in an Interrupt-driven Kernel *ACM Trans. Comput. Syst.*, vol. 15, pp. 217–252, Aug. 1997.

[28] J. S. Chase, A. J. Gallatin, and K. G. Yocum, End-System Optimizations for High-Speed TCP 2000.

[29] S. Han, K. Jang, K. Park, and S. Moon, PacketShader: A GPU-accelerated Software Router *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 195–206, Aug. 2010.

[30] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, Design and Implementation of a Stateful Network Packet Processing Framework for GPUs *IEEE/ACM Transactions on Networking*, vol. 25, pp. 610–623, Feb 2017.

[31] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, The Click Modular Router *ACM Trans. Comput. Syst.*, vol. 18, pp. 263–297, Aug. 2000.

[32] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware in *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, (New York, NY, USA), pp. 1–14, ACM, 2016.

[33] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon, NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, (New York, NY, USA), pp. 22:1–22:14, ACM, 2015.

[34] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson, Introduction to the wire-speed processor and architecture *IBM Journal of Research and Development*, vol. 54, p. 3, 2010.

[35] F. Abel, C. Hagleitner, and F. Verplanken, Rx Stack Accelerator for 10 GbE Integrated NIC *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, pp. 17–24, 2012.

[36] C. Raiciu, C. Paasch, S. Barré, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, How hard can it be? designing and implementing a deployable multipath TCP in *NSDI 2012*, 2012.

[37] W. R. Stevens, TCP/IP Illustrated, Volume 1: The Protocols 1994.

[38] M. Thomson and J. Iyengar, QUIC: A UDP-Based Multiplexed and Secure Transport 2019.

[39] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, Pluginizing QUIC in *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, (New York, NY, USA), pp. 59–74, ACM, 2019.

[40] "1.3. bpf() syscall for ebfp virtual machine programs." `https://kernelnewbies.org/Linux_3.18#bpf.28.29_syscall_for_eBFP_virtual_machine_programs`. Accessed: 2019-11-17.

[41] R. S. Fish, J. M. Graham, and R. J. Loader, DRoPS: kernel support for runtime adaptable protocols in *Proceedings. 24th EUROMICRO Conference (Cat. No.98EX204)*, vol. 2, pp. 1029–1036 vol.2, Aug 1998.

[42] J. Heuschkel, A. Frömmgen, J. Crowcroft, and M. Mühlhäuser, VirtualStack: Adaptive Multipath Support through Protocol Stack Virtualization in *Eleventh International Network Conference, INC 2016, Frankfurt, Germany, July 19-21, 2016. Proceedings*, pp. 73–78, 2016.

[43] M. A. Hiltunen, R. D. Schlichting, C. A. Ugarte, and G. T. Wong, Survivability through customization and adaptability: the Cactus approach in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, vol. 1, pp. 294–307 vol.1, 2000.

[44] N. Feamster, J. Rexford, and E. Zegura, The Road to SDN: An Intellectual History of Programmable Networks *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–98, Apr. 2014.

[45] "Network Functions Virtualization— Introductory White Paper" tech. rep., ETSI, Oct 2013.

[46] N. McKeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner, OpenFlow: enabling innovation in campus networks *Computer Communication Review*, vol. 38, pp. 69–74, 2008.

[47] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, P4: Programming Protocol-independent Packet Processors *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, July 2014.

[48] The P4 Language Specification Version 1.0.5 tech. rep., The P4 Language Consortium, 2018.

[49] J. Santiago da Silva, F.-R. Boyer, and J. P. Langlois, P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, (New York, NY, USA), pp. 147–152, ACM, 2018.

[50] I. Netronome Systems, Programming NFP with P4 and C tech. rep., Netronome Systems, Inc, 2017.

[51] I. Netronome Systems, NFP-4000 Theory of Operation tech. rep., Netronome Systems, Inc, 2016.

[52] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, Network Function Virtualization: State-of-the-Art and Research Challenges *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 236–262, 2016.

[53] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, The Design and Implementation of Open vSwitch in *NSDI*, 2015.

[54] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, E2: A Framework for NFV Applications in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 121–136, ACM, 2015.

[55] W. Wingerath, F. Gessert, S. Friedrich, and N. Ritter, Real-time stream processing for Big Data *it - Information Technology*, vol. 58, pp. 186–194, 2016.

[56] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, ZooKeeper: Wait-free Coordination for Internet-scale Systems in *USENIX Annual Technical Conference*, 2010.

[57] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, Twitter Heron: Stream Processing at Scale in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 239–250, ACM, 2015.

[58] Q. L. Anderson, Storm Real-Time Processing Cookbook 2013.

[59] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, Dhalion: Self-regulating Stream Processing in Heron *Proc. VLDB Endow.*, vol. 10, pp. 1825–1836, Aug. 2017.

[60] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, STREAM: The Stanford Data Stream Management System in *Data Stream Management*, 2016.

[61] B. Babcock, S. Babu, M. Datar, and R. Motwani, Chain : Operator Scheduling for Memory Minimization in Data Stream Systems in *SIGMOD Conference*, 2003.

[62] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. Morris, Noria: dynamic, partially-stateful data-flow for high-performance web applications in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 213–231, USENIX Association, 2018.

[63] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, StreamBox: Modern Stream Processing on a Multicore Machine in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, (Berkeley, CA, USA), pp. 617–629, USENIX Association, 2017.

[64] W. Thies, M. Karczmarek, and S. Amarasinghe, StreamIt: A Language for Streaming Applications in *Compiler Construction* (R. N. Horspool, ed.), (Berlin, Heidelberg), pp. 179–196, Springer Berlin Heidelberg, 2002.

[65] M. I. Gordon, W. Thies, and S. Amarasinghe, Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs *SIGPLAN Not.*, vol. 41, pp. 151–162, Oct. 2006.

[66] A. D. Pimentel, The Artemis workbench for system-level performance evaluation of embedded systems *IJES*, vol. 3, pp. 181–196, 2008.

[67] G. Smith, Platform Based Design: Does It Answer the Entire SoC Challenge? in *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, (New York, NY, USA), pp. 407–407, ACM, 2004.

[68] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. F. Deprettere, System design using Khan process networks: the Compaan/Laura approach *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 340–345 Vol.1, 2004.

[69] E. A. Lee and D. G. Messerschmitt, Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing *IEEE Transactions on Computers*, vol. C-36, pp. 24–35, 1987.

[70] S. Stuijk, M. Geilen, B. D. Theelen, and T. Basten, Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 404–411, 2011.

[71] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[72] Y. . S. Li and S. Malik, Performance analysis of embedded software using implicit path enumeration *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 1477–1487, Dec 1997.

[73] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm, Static Timing Analysis for Hard Real-Time Systems in *Verification, Model Checking, and Abstract Interpretation* (G. Barthe and M. Hermenegildo, eds.), (Berlin, Heidelberg), pp. 3–22, Springer Berlin Heidelberg, 2010.

[74] S. Edgar and A. Burns, Statistical analysis of WCET for scheduling *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, pp. 215–224, 2001.

[75] J. Souyris, E. Pavec, G. Himbert, G. Borios, V. Jégu, and R. Heckmann, Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 01 2005.

[76] C.-S. Chang, Performance guarantees in communication networks *European Transactions on Telecommunications*, vol. 12, pp. 357–358, 2001.

[77] M. Jung, E. W. Biersack, and A. Pilger, Implementing network protocols in java - a framework for rapid prototyping in *In International Conference on Enterprise Information Systems*, pp. 649–656, 1999.

[78] S. Büchner, L. Lopacinski, R. Kraemer, and J. Nolte, Protocol Processing for 100 Gbit/s and Beyond - A Soft Real-Time Approach in Hardware and Software *Frequenz*, vol. 71, pp. 427–438, Sept. 2017.

[79] L. Lopacinski, J. Nolte, S. Büchner, M. Brzozowski, and R. Kraemer, Parallel RS Error Correction Structures Dedicated for 100 Gbps Wireless Data Link Layer in *15th IEEE International Conference on Ubiquitous Wireless Broadband 2015:*

Special Session on Wireless Terahertz Communications (IEEE ICUWB 2015 SPS 02), (Montreal, Canada), oct 2015.

[80] M. Dobrescu, K. Argyraki, and S. Ratnasamy, Toward Predictable Performance in Software Packet-Processing Platforms in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, (San Jose, CA), pp. 141–154, USENIX, 2012.

[81] S. Büchner, A. Hasani, L. Lopacinski, R. Kraemer, and J. Nolte, 100 Gbit/s End-to-End Communication: Adding Flexibility with Protocol Templates in *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, pp. 263–266, Oct 2018.

[82] S. Büchner, J. Nolte, A. Hasani, and R. Kraemer, 100 Gbit/s End-to-End Communication: Low Overhead On-Demand Protocol Replacement in High Data Rate Communication Systems in *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, pp. 231–234, Oct 2017.

[83] N. Bombieri and F. Fummi, Automatic Transactor Generation in TLM by Exploiting EFSMs 2007.

[84] L. Lopacinski, *Improving goodput and reliability of ultra-high-speed wireless communication at data link layer level.* Doctoral thesis, BTU Cottbus - Senftenberg, 2017.

[85] IEEE, Ethernet Jumbo Frames (standard), IEEE Computer Society, 3855 SW 153rd Drive, Beaverton, OR 97006, 11 2009.

[86] Mellanox, 350 Oakmead Parkway, Suite 100, Sunnyvale, CA 94085, *TILEncore-Gx72 Intelligent Application Adapter*, 53461pb/rev 1.5 ed.

[87] L. Förster, *Optimising DMA Streams for Soft Real-Time Stream Processing.* Master thesis, BTU Cottbus - Senftenberg, 2017.

[88] M. Diener, E. H. M. da Cruz, and P. O. A. Navaux, Locality vs. Balance: Exploring Data Mapping Policies on NUMA Systems *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 9–16, 2015.

[89] Z. Gabonayová, *Resource Management for Soft Real-Time Stream Processing on Many-Core Systems.* Master thesis, BTU Cottbus - Senftenberg, 2019.

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

[91] IEEE, IEEE Standard for Ethernet - IEEE Std 802.3(TM)-2012 (standard), IEEE Computer Society.