10

End2End100 – Communication Protocol Processing for Ultra High Data Rates

Steffen Büchner, Jörg Nolte

Distributed Systems/Operating Systems Group,
Brandenburg University of Technology Cottbus–Senftenberg

Alireza Hasani, Rolf Kraemer

Systems Group, Brandenburg University of Technology Cottbus–Senftenberg

Lukasz Lopacinski

System Design, IHP – Innovations for High Performance Microelectronics

CONTENTS

DOI: 10.26127/BTUOpen-5916

The theoretical data rate of wireless transmissions is pushed to 100 Gbit/s and even beyond. However, such extremely high data rates can not be processed at the receiving hosts as the following short example exemplifies.

> "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 nanoseconds. Putting that in relation with the 96.4 ns main memory access latency for a 64 Byte cache line (Intel Haswell [379]), indicates that we have to think of new protocol processing paradigms." [380]

This short example shows that the traditional protocol processing within the endpoint's kernel space is not suited for these ultra-high data rates. Consequently, one of the main questions for today's high-speed network research is how to handle the theoretical bandwidth, i.e., how to transform a single stream of protocol data into a single stream of application data. Furthermore, the challenge is increased due to a high bit error rate and changing channel qualities as to be expected for a wireless communication setup.

Figure 10.1 shows the envisioned communication system. The systems consist of the host, an embedded many-core that functions as a smart Network-Interface-Card (NIC), and custom external accelerators for the compute extensive protocol tasks, such as the FEC calculation. In this setup, the hosts are only used for producing and consuming the application data stream. The application data stream is received by the smart NIC that processes the communication protocol in parallel, the higher-level protocol processing, e.g., buffer and retransmission management are conducted on the embedded many-core because of its programmability. Parallelizing the whole protocol processing with the help of the stream processing paradigm allows providing the desired data rate [380]. However, due to the expected fluctuations in the channel qualities, it will not be sufficient to provide a static implementation for the wireless scenario. Instead, the changes in channel quality are handled by automatically reconfiguring the protocol processing. The automatic reconfiguration is applied to all processing levels.

On the link level, the link quality is continuously monitored and the most suited link is chosen for the connection. In the case that no single link can provide the desired data rate, several links are automatically combined. On the protocol level, the size of the payload per frame, the necessary redundancy and the segmentation ratio of the frame can be reconfigured to better suit the new communication links.

However, combining parallel links with different data rates and using new protocol parameters most probably changes the processing requirements. Both cases make it necessary that the protocol processing is also reconfigured, by assigning more (ore less) resources, e.g., Central Processing Unit (CPU) cores,

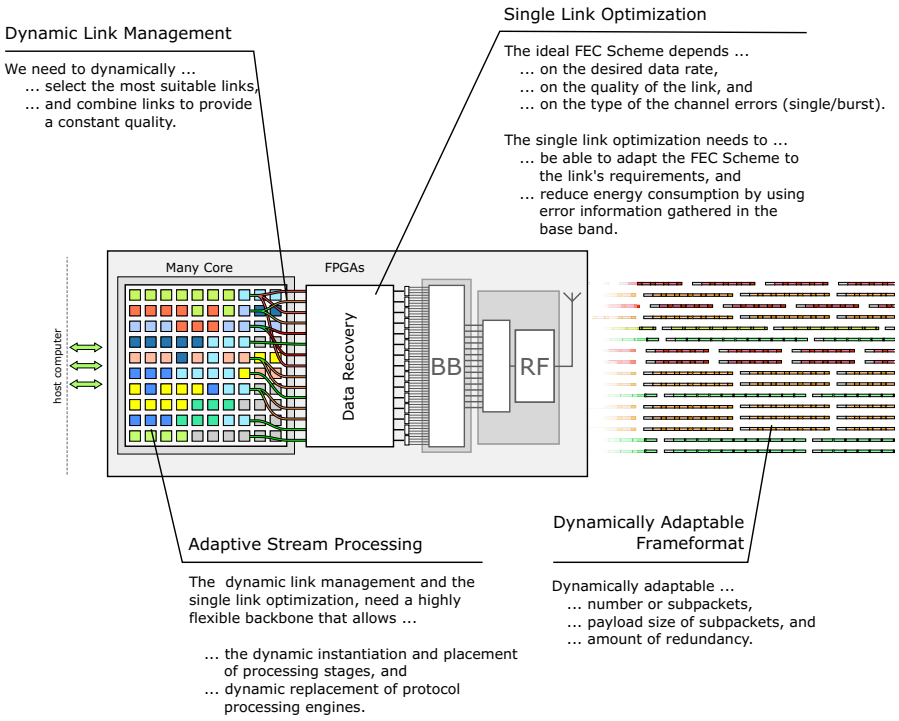


Figure 10.1
Envisioned wireless communication system. Adapted from [381]

or by switching between implementations. Consequently, the stream processing based protocol processing framework has to be extended with means of on-demand reconfiguration of the implementation. Finally, the data recovery layer must be reconfigurable in order to adapt the FEC parameters, such as the amount of redundancy, to the reconfigured protocol and the monitored channel quality.

As shown in Figure 10.1, additional accelerators are needed in the system in order to perform a number of tasks which are of higher complexity. That means certain protocol tasks have to be offloaded because their processing requirements make them unsuitable for the processing in software. Apart from complexity, not all protocol tasks are suited in nature for a software implementation. For these two reasons the presence of such external accelerators is inevitable.

In general, Error handling techniques like FEC and ARQ are somewhat inevitable, specifically at as high data-rates as 100 Gbps, to deal with transmission errors, since wireless communication suffers from a high Bit Error Rate (BER). Among these we have chosen Cyclic Redundancy Check (CRC) and FEC protocols to be offloaded into special-purpose hardware. The main reasons for adopting this strategy is as follows. FEC must perform extremely fast, within only a few nanoseconds processing time for a single frame, to support 100 Gbps communication. This includes various processing that has to be done on the packets, such as updating frame headers, calculation of checksums, splitting data into frames and segments, and more important than all complicated mathematical operations to compute the redundant bits. At the data rate of 100 Gbps for instance, TX and RX must process a frame of 1500 bytes within 120 nanoseconds, which is obviously an enormous burden on the processing device. On the other side, a review of the Ethernet systems already capable to achieve 100 Gbps on the general-purpose processors, i.e. four Intel Xeon cores, reveals that they dissipate as much as 650 Watts of power [382]. This number will be even greater in the case of wireless communication, as they need more processing power than Ethernet. This can be another reason to offload some of the tasks to a hardware like Field Programmable Gate Array (FPGA) or application-specific integrated circuit (ASIC) which facilitate more efficient implementation of the aforementioned protocols.

This chapter is used to highlight the proposed solutions on parallelization and on-demand reconfiguration of the communication system, as well as the challenges concerning the wireless medium. In section 10.1 a parallelizable data link protocol is presented. The proposed protocol can be processed parallel in several pipeline steps. The individual pipeline steps can be parallelized further in order to avoid stalling due to processing bottlenecks. However, finding such bottlenecks is not a trivial task. Section 10.2 presents a design process that allows transforming a communication protocol into a stream processing graph. This graph allows to identify possible processing bottlenecks and eliminate them by parallelization. However, some bottlenecks have such high requirements that parallelization is not an option. In Section 10.3 an FEC/CRC

accelerator with embedded link adaptation methods is presented. The accelerator is integrated seamlessly into the communication system. Finally, an evaluation of the communication system is presented in section 10.4.

10.1 A parallelizable Data Link Protocol

A protocol for ultra-high data rates has to be parallelizable, offloadable, and reconfigurable. Figure 10.2 shows the frame format of the data link layer protocol, designed for high-speed wireless communication.

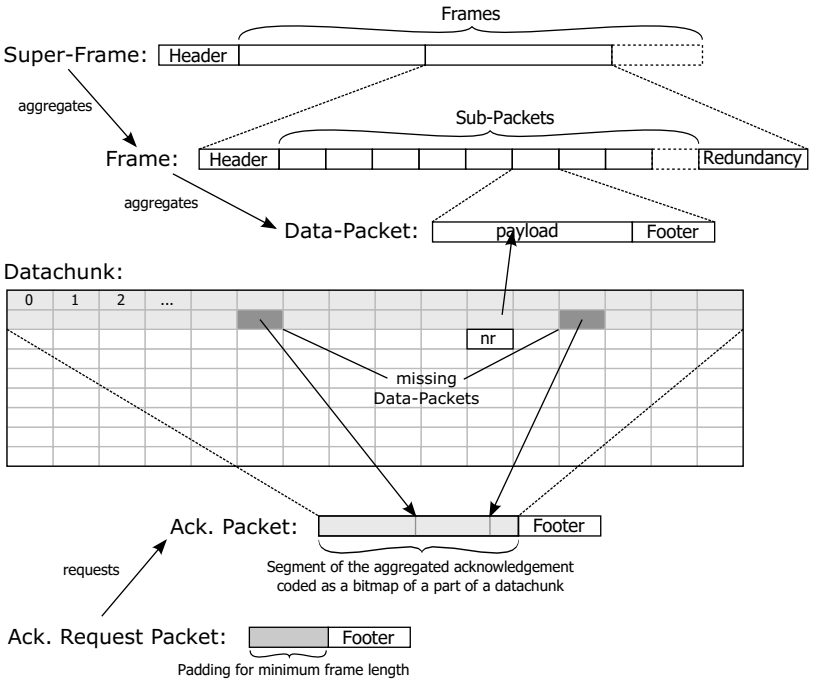


Figure 10.2
Frame format of the prototype data link protocol. From [381]

The frame format provides several layers of parallelization. At the highest level, a data stream is divided into very large packets, called data chunks. A data chunk is used for encapsulating the application level data stream. For that, it provides all necessary information, such as its size and its position within the stream. With these information it is possible to process individual data chunks in parallel.

A data chunk itself is further separated into data packets. A data packet

has an offset within the data chunk and a handle to the data chunk it belongs to. Consequently, data packets that belong to the same data chunk can be processed out of order and in parallel. The size of the data packets depends on the expected BER, i.e., the lower the BER, the larger the data packet's size [381, 383]. However, in order to avoid, that small data paket's lead to an underutilization of the communication channel, data packets are aggregated into larger frames.

Additionally, to the lower packet loss probablity, the aggregation has the effect that parts of frames can be retransmitted selectively instead of completely losing the data. To reduce the metadata overhead, a frame can aggregate only data packets that belong to the same data chunk. This can lead to a situation in with the communication channel is not completely utilized because not enough data packets of a certain data chunk are ready for transmission. One can circumvent this by further aggregating frames into superframes [381].

At this point, data packets can be processed completely in parallel, as they provide all the necessary information. A similar approach is used for the retransmission mechanism. All data packets that belong to a data chunk are coded as an aggregated acknowledgment. This ACK/NACK bitmap describes the current transmission status of the data chunk. Parallel processing is carried out by dividing the bitmap of a data chunk into "sub"-bitmaps. All sub-bitmap can now be processed in parallel.

The aggregation of acknowledgments for a full data chunk into a bitmap leads to a situation in which all data packets that were not yet received, are stated as "NACK". However, it is not clear whether these data packets were sent already. To avoid that the sender has to determine whether a data packet was already sent, the aggregated acknowledgments are sent only on demand by the sender (by sending an `AckRequest` packet) of the sender. While this may increase the latency per data chunk, it allows to request an acknowledgment after all data packets of a certain data chunk were sent, consequently, all data packets that are stated as missing in the acknowledgment were sent already and have to be retransmitted.

The proposed frame format is also highly flexible. Firstly, the size of the data chunk is configurable, which allows compromising between latency per data chunk and the number of host invocations. Secondly, the data packet size can be adjusted to the current BER, e.g., a low BER allows for larger data packets and less protocol overhead. Lastly, the amount of redundancy used for the FEC mechanism is configurable.

10.1.1 Protocol Description

The protocol works in two phases: A transmission and a retransmission phase. The transmission phase is used to initially transmit the data chunk. It starts with the separation of the data chunk into data packets, which are aggregated into data frames. Before sending any data over the communication channel,

the redundancy information for the FEC mechanism is calculated and added to the frame.

Upon receiving a frame at the receiver, the frame is firstly checked for errors which are corrected if possible by the FEC mechanism. Then the data packets of the frames are copied to the destination position in the data chunk they belong to. Each data packet that was correctly received is then marked accordingly in the aggregated acknowledgment. Once all data packets that belong to a data chunk were transmitted, the protocol switches into the retransmission phase.

The retransmission phase starts when the sender requests the first aggregated acknowledgment from the receiver. Upon receiving the aggregated acknowledgment as an answer to the request, the sender analysis the acknowledgement and retransmits data packets that are stated as missing in the acknowledgement. When all missing data packets were retransmitted, the sender requests a new acknowledgment and waits. Eventually, all data packets are transmitted correctly and the sender and receiver switch back into the transmission phase and process the next data chunk.

10.1.2 Protocol Pipelines

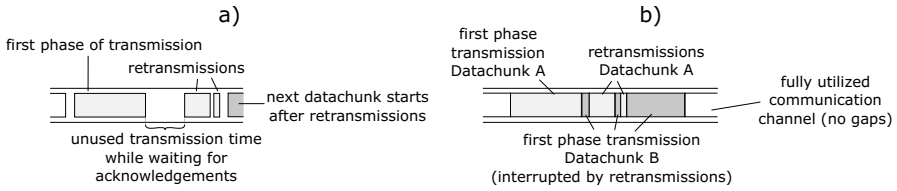


Figure 10.3
Channel utilization with single- and multi-pipeline implementations. From [381].

The presented protocol may lead to wasted transmission time due to waiting for the aggregated acknowledgement as shown in figure 10.3a). This results in a stable and low latency per individual data chunk because the communication channel is monopolized for the transmission of that data chunk, however, it also leads to a lower overall throughput due to the wasted transmission time. This can be avoided by multiplexing the simultaneous transmission of two consecutive data chunks on the communication channel as shown in figure 10.3b). Now the communication channel is completely utilized by accepting a higher latency per data chunk. What approach is best suited depends on the communication scenario.

10.2 Parallel Protocol Processing

The parallelization and implementation of communication protocols are cumbersome tasks. This has many reasons: Firstly, the protocol processing is traditionally carried out in the operating system kernel. That means that improvements and testing of communication protocols always mean changes at the kernel. Furthermore, the parallelization is hindered because of dependencies within the protocol processing, e.g., the protocol state has to be shared and therefore synchronized. Finally, the amount of parallelization is usually unknown at design time as it depends on the protocol, the desired data rate, the communication conditions, and the execution hardware.

These pitfalls can be avoided by moving the protocol processing into the userspace. However, that contradicts the requirement of freeing the communication endpoint from the protocol processing. Using a smart NIC that consists of an easily programmable embedded many-core, which also provides the necessary parallel processing power for the higher-level protocol tasks, is another way of solving that problem.

Parallelization is simplified by interpreting communication protocols as stream processing problems. Each communication protocol can be described as a stream processing graph [384], as shown for a generalized communication protocol in figure 10.4. The protocol consists of a sender, S, which consumes 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 receiver produces a stream of acknowledgments, which are consumed and used by the sender to create a stream of retransmissions.

This approach has the advantage that stream processing applications are implicitly parallelizable. The synchronization effort is thereby minimized because two stream-nodes do not share any state and depend only on the streamed items. Therefore, the processing is implicitly synchronized by the streaming of data items.

The stream processing graph of that generalized communication protocol can be augmented with the protocol's processing requirements and the processing hardware's performance characteristics. The processing requirements

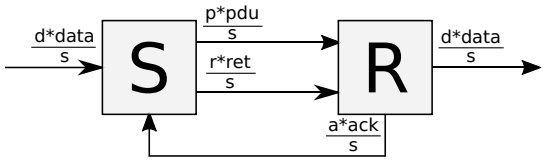


Figure 10.4
Soft real-time problems stream processing problem. From [384] © 2015 IEEE

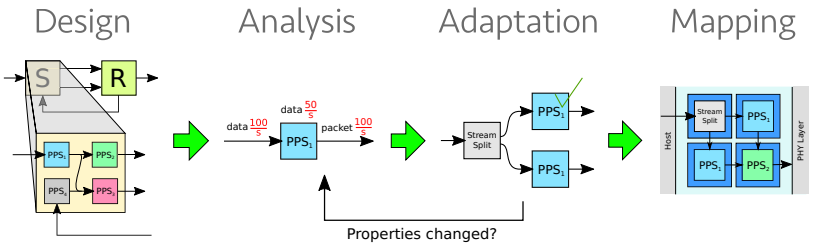


Figure 10.5
Stream processing based protocol implementation approach. Adapted from [380] © 2016 IEEE

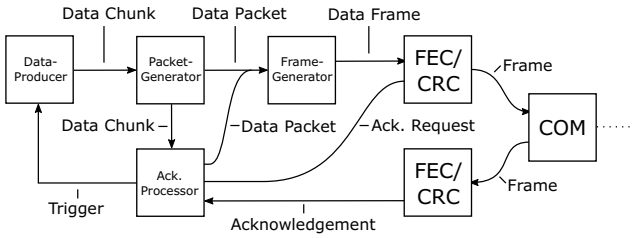


Figure 10.6
Stream processing representation of the data link protocol

are given implicitly by the data rate of the incoming data stream. For example, stage S has to process $\frac{n_{data} * data}{second} + \frac{z_{ack} * ack}{second}$. The performance characteristics, i.e., what is the highest data rate that can be processed by the hardware, can be measured individually for each stage on the processing hardware. An analysis of the ratio between requirements and performance characteristics, allows the protocol developer to identify processing bottlenecks. Moreover, the analysis's outcome can be used to adapt the stream processing graph so that soft real-time requirements correspond to the hardware's performance characteristics.

10.2.1 Stream Processing based Protocol Design

On this basis, a design process as sketched in figure 10.5 was conceived. The design process consists of 5 steps which are shortly presented in the following. A detailed explanation of the design process can be found in [381, 380].

In the first step, the communication protocol is decomposed into processing tasks (see figure 10.6). The finer the decomposition, the higher the possible parallelization. The sender side of the data link protocol was decomposed into five stages. The `Data-Packet Generator (DG)` is responsible for cutting a data chunk into smaller data packets. These data packets are streamed to the `Data-Packet Aggregator (DA)` which aggregates the data

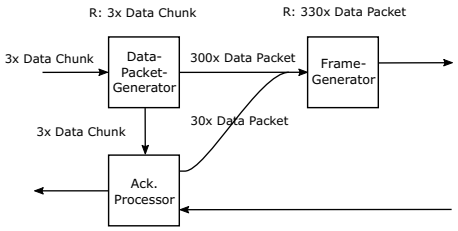


Figure 10.7
Soft real-time requirements are determined by the input data rate

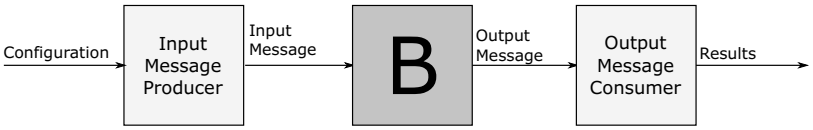


Figure 10.8
Measurement of a stage. From [381]

packets into frames. The complete frame is then given to the `Forward Error Correction (FEC)` stage which adds the redundancy that is necessary for the reconstruction of broken data packets. After the data chunk is completely processed, the `DG` notifies the `Acknowledgement Processor (AP)` that the processing is complete, and that an aggregated acknowledgment can be requested from the receiver. Upon receiving the requested acknowledgments, the `AP` retransmits the missing packets and requests another acknowledgment.

In the second step, the resulting stream graph, called a processing engine, is analyzed for its processing requirements and the performance characteristics given a certain hardware. The processing requirements are determined by applying the target data rate at the inputs of the processing engine. The established input data rate then determines the processing requirements of the stage (see figure 10.7). Additionally, the processing stage transforms the input stream into one or more output streams which now have their own data rate. These streams are fed into the following processing stages and therefore determine their processing requirements. The analysis is continued until the processing requirements for all stages are established.

The performance characteristics are measured at the target hardware and specify the maximum target data rate a group of stages can process on a certain number of processors. The measurement is straight forward because any stage only depends on its internal state and the incoming messages. Therefore a simple measuring setup as depicted in figure 10.8 can be used. In the first step, the stage is set up to be in the state of interest. Afterward, messages are streamed to the stage and the processing time is measured.

The measured processing time per message of the analysis is then used in the adaptation step for fitting the processing engine to the target hardware.

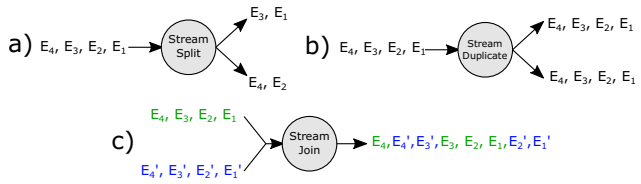


Figure 10.9
Stream operators

Since the stages are independent of each other, the processing engine can be parallelized by splitting streams into substreams and, therefore, distribute the processing load over a set of processors. The adaptation is conducted with the help of three stream operators (see figure 10.9), which are used to manipulate the data rate of a stream but not the streamed items.

The three stream operators are stream split, stream duplicate, and stream join. The stream split operator distributes streamed items round-robin over a certain number of sub-streams, consequently, the data rate of the sub-streams is reduced. The stream duplicate stream operator is used to clone a stream, i.e., each sub-stream contains the same items and has the same data rate. Finally, the stream join operator combines several streams, i.e., the resulting data rate is the sum of all input data rates.

The stream operators are used for the adaptation of the processing engine as shown in figure 10.10. In the example, the input data rate of the framing stage is too high for a single processor. Consequently, a stream split operator is used to reduce the data rate by splitting the stream into two sub-streams. However, not all stages are candidates for parallelization. For example, the `FEC` stage has high computational requirements and is better suited to be offloaded into external accelerators. Since the stages are connected by the message streams, it is only necessary to employ a message passing mechanism between the devices and marshal the message. Details about the offloaded `FEC` follow later in this chapter.

The adapted processing engine can be mapped onto the communication system. Due to the design process, the resulting processing engine can process the desired data rate on the target hardware given the expected communication conditions.

However, wireless communication systems can seldom guarantee static conditions. 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 were 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/requirements efficiently.

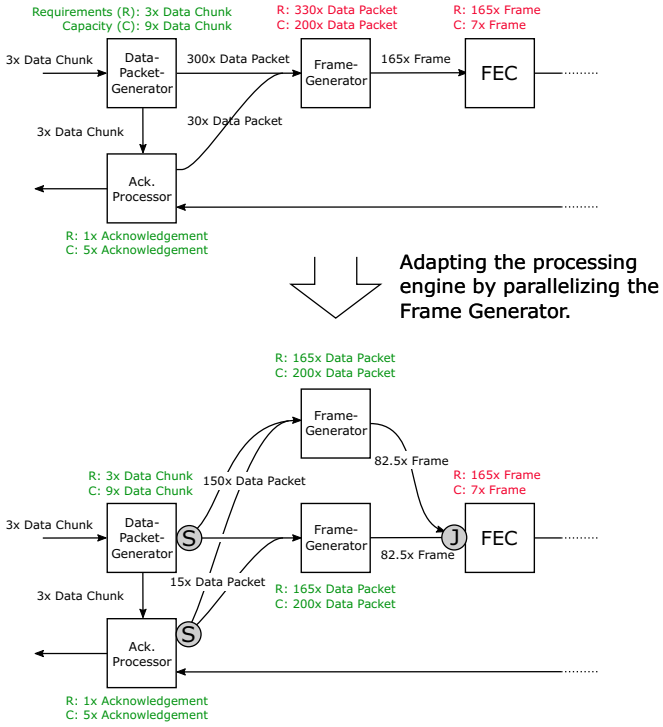


Figure 10.10
Frame Generator and the FEC stage as parallelized stages

Avoiding resource wastefulness, 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. The on-demand adaptation and replacement of processing engines that will be described in the following, use the PETL. The PETL is a graph description language that describes the stages as well as connections and provides the configuration for the individual stages.

10.2.2 Processing Engine Template Language

The stream processing approach leads to a statically designed processing engine that is specialized for a given data rate or communication condition. This is undesired in situations in which high flexibility is needed, e.g., whenever the communication conditions change, or when, even worse, the conditions are unknown at the time the protocol is being designed.

In order to provide the desired flexibility, the unadapted processing en-

gine can be regarded as a template [385]. This way, a processing engine can be adapted for different data rates, communication scenarios, as well as for changed communication conditions, on-demand with the help of a single description.

Transforming a processing engine into a template is accomplished by providing additional information that specifies how a certain adaptation has to be executed. This information is provided by regarding stages as collections and individual streams as relations. Both of these are used to construct a template processing engine, whereas a collection describes a set of stages and a relation describes a set of message streams between stages. Each processing engine template comes with a set of adaptation patterns that describe the valid template specifications. In the following, the collections, relations and adaptation patterns are explained in detail.

Collections represent the stages of a processing engine and their parallelization information, i.e., how a single stage has to be parallelized. Besides the static name, which identifies the stage, each collection is described by the following parameters:

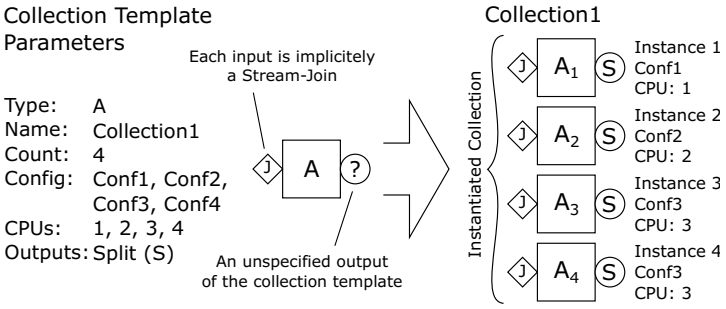


Figure 10.11
Stages of the protocol implementation described as collections

- **Stage-Type** – The Stage-Type field of a collection specifies the actual implementation of stage for that collection.
- **Count** – The count specifies the number of parallel instances in a collection.
- **Config** – The config field is an ordered list of configurations, such as a retransmission timeout, used to set up the individual stages. Configurations are specific for each stage-type and are applied in the order they are stated.
- **Groups** – The groups attribute states the number configuration groups within a collection, i.e., how many stages in a Collection share the same configuration.

- **CPUs** – The CPUs field is an ordered list that defines the CPU-mapping for a collection. The mappings are also applied in the order they are stated.
- **Outputs** – The Outputs field is an array of output configurations. An output configuration defines the stream operators that are used for each slot of the output.

With these attributes, the protocol developer is able to specify the type of the stages, the needed number of instances, their configuration, as well as the stream operators for each output for any type of processing engine. The count parameter is considered to be unspecified in the description and therefore making the description a template. However, any other of the parameters, such as the stage type, can be unspecified and be considered as a template parameter, hence, making the template more flexible. Figure 10.11 shows an example of a fully specified collection. Instantiating the collection `A` leads to four stages, each of them is being parameterized according to their configuration, CPU mapping and stream operator setup.

The message streams between the stages in a processing engine are represented by the relation primitive of the PETL. Similar to the collections, which provide the parallelization information for a stage, a relation describes the data streams between two collections. A relation has by the following attributes:

1. **Source- and Destination Collection** – The name of the source and the destination collection, respectively.
2. **Source Output/Destination Input** – Since a stage can have several outputs/inputs, these attributes specify the output and input that have to be connected.
3. **Modifier** – The modifier specifies how a stream operator, specified in a collection, is applied to the source and destination collection.
4. **Slot** – The slot specifies which output-slot of the output shall be used for the relation. This attribute was introduced for the developer's convenience and is used to enable output selection by indexing.

By defining the attributes of the relations two collections are connected. However, after defining the relation's attributes, the actual streams between stages of the source and destination collection are still ambiguous, i.e., it is not clear how the relation should connect the individual instances of the collections. These ambiguities are resolved by stating a modifier that defines a generic connection pattern for each relation. The modifiers are shown in figure 10.12 and explained in the following.

The direct, symmetric and all-to-all modifiers are the three cases of the n-to-m modifier. The main idea of the n-to-m modifier is, that it divides two arbitrary collections into subsets that are then connected in a all-to-all pattern.

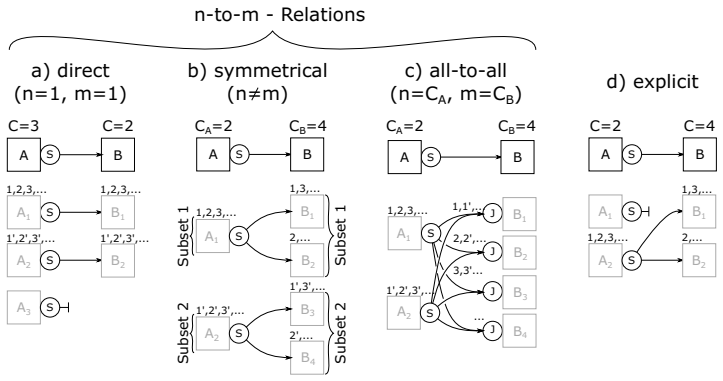


Figure 10.12
The unfolding of relations depends on the collection's count attribute and the assigned stream modifier: a) direct b) symmetrical c) all-to-all d) explicit.

When the direct modifier is used, a relation connects the x^{th} source stage with the x^{th} destination stage. This is used when it is necessary to connect the stages of a collection strictly pairwise. In the case the number of processing stages in a collection differs, some processing stages stay unconnected.

Symmetrical relation unfolding is used to separate the source and the destination collection into subsets that are connected individually. Since these subsets form distinct processing pipelines, the symmetrical modifier is used to create independent protocol pipelines. The principle is shown for the relation between the collections A and B. Both collections are logically separated into an equal amount of subsets. In the example, it results in splitting the output data stream of the processing stages of the source collection into two processing stages of the destination collection. These subsets are now connected individually.

The All-to-All modifier connects all source stages with all destination stages, i.e., the workload of all source stages is distributed over all destination stages.

Additionally, the explicit allows for the application of a certain relation to an individual stage in a collection, as shown for the relation between the collections A and B. This is used when it is necessary to connect the individual stages of a collection independently. Instead of applying the Split/Join operator to the whole source collection, the relation is constraint by the explicit modifier and only applied to stage A_2. When both, the source and the destination instance, are specified, it connects two individual stages of two collections. The explicit modifier adds the source and destination instance to a relation-definition.

Figure 10.13 shows how a PETL description of a processing engine with 5 stages is specialized into an actual processing engine with the help of relation modifiers and the collection's count attribute.

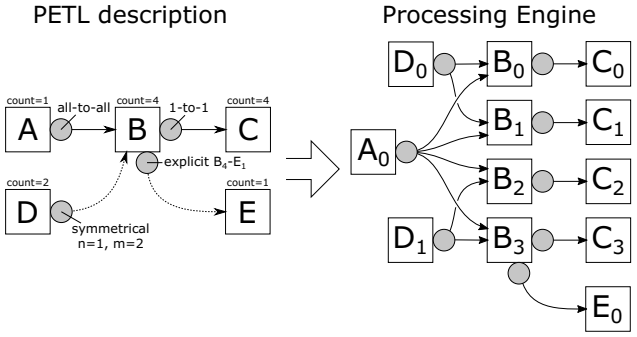


Figure 10.13
Instantiation of a PETL-description into a specialized processing engine by assigning the parallelization ratios.

10.2.3 Reconfiguration of a deployed Processing Engine

The presented description language [385] was developed during this project and allows describing template implementations of protocols. As shown, the actual protocol implementations can be derived from the PETL description by applying the necessary parallelization ratios. However, reconfiguring the implementation at runtime leads to the question at what point of time during the transmission the reconfiguration can be conducted.

Two situations have to be discussed. Firstly, increasing resources and secondly removing resources. Increasing the amount of resources is done by allocating the resources and integrating new processing stages on the fly. Firstly, new stages are created and then they are connected into the processing engine. This way all resources needed are initialized and ready to use when the first protocol messages arrive.

However, removing resources is more difficult because the resources are in use. Instead of just removing processors, the resources to be removed are marked accordingly. Then the incoming message streams are cut, i.e., no new protocol messages can reach the stages that have to be removed. Eventually, the input queues of these stages are empty, at this moment the stages are removed.

Since all connections that were directed to a removed stage were cut, no messages can reach the draining stages anymore. Therefore, all stages have to be "self-sufficient" after being removed. That means a stage that is marked as prepare-to-discard must be able to process messages. In case the completion of such a stage's immediate task is crucial[1] to the further protocol processing, but cannot be finished, the stage has to inform the remaining of the processing engine that it could not proceed.

In some cases, a completely new protocol is needed to fulfill the processing

[1]The question what is a crucial task depends on the protocol and it's implementation.

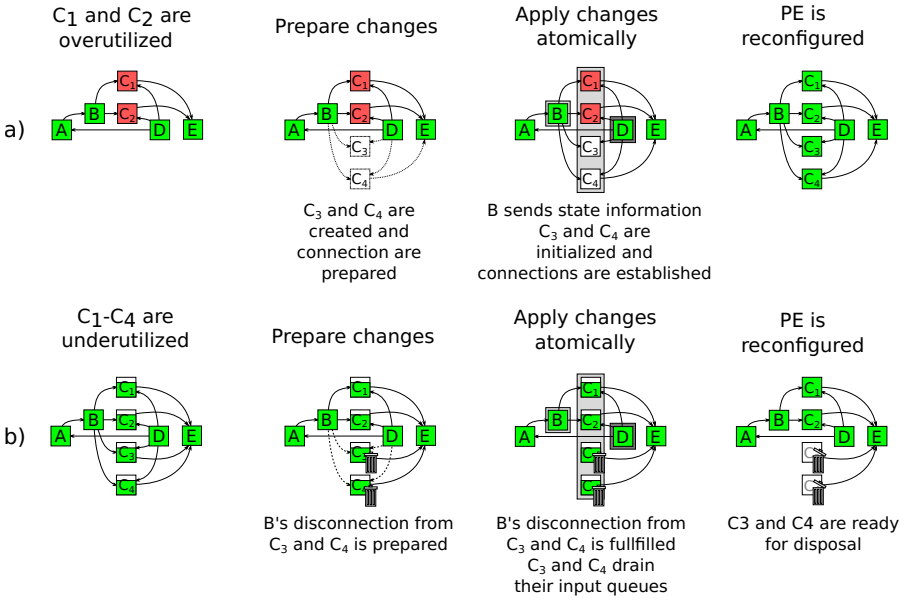


Figure 10.14
Replacement process of a protocol. a) The collection C is over-utilized, then a patch that adds two more stages is applied. b) The collection C is under-utilized, then a patch that removes two stages is applied. From [381]

requirements. However, applying changes to the underlying protocol leads to an inconsistent protocol state that has to be reinterpreted. Since the reinterpretation of states is highly dependent on the protocols between the interpretation should be done, the reinterpretation would had to be designed for each protocol pair. This is highly inconvenient because it does not allow for a generic reconfiguration process. Stopping the transmission and restarting it with the new protocol is no solution either because the resulting overhead would severely affect the transmission.

Instead, two protocols are used simultaneously for a transmission. The old protocol is used until it is drained of data, and the new protocol is used for the ongoing transmission. This allows us to seamlessly switch between two different protocols. However, this approach also means, that two protocols for the same transmission are possibly multiplexed on the same links. Consequently, the network frames received from an underlying layer have to be distributed to the correct protocol implementation.

This is achieved by virtualizing the communication interface with virtual channels as shown in figure 10.15a). A virtual channel is a protocol identification number that is transparently added to outgoing frames. Upon receiving a frame the communication systems read the virtual channel and forward it to the network frame to the correct protocol implementation.

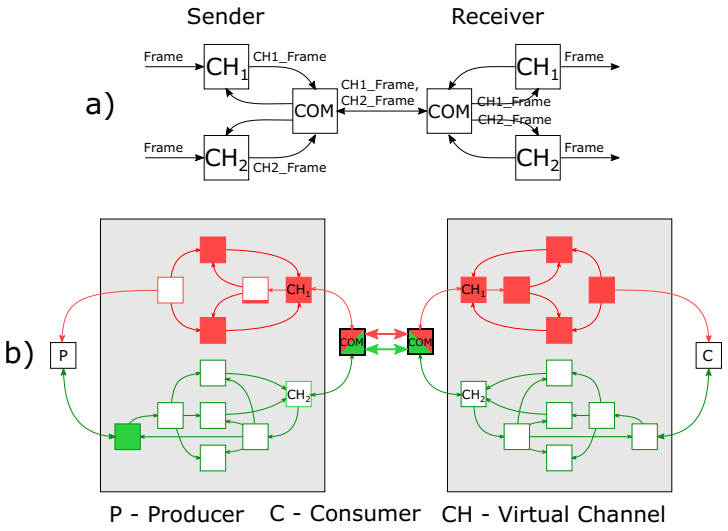


Figure 10.15
a) The virtual channel multiplexes two or more processing engines onto a communication channel. b) Replacing a processing engine on-demand. From [381]

The actual switch between the two protocols is conducted in two steps. Firstly, after choosing a suitable protocol, the new protocol is built, mapped and initialized. Once the protocol is ready, the data streams from the producer are reconnected so that the new protocol is used. Since the old and the new protocol are used simultaneously the virtual channel distributes the network frames to the correct protocol implementation as shown in figure 10.15b).

10.3 FEC

The other part of the system which is of great importance in high-throughput wireless communication systems is the one dealing with detection and correction of errors occurring during transmission. As discussed before at the beginning of section 10 such related tasks are offloaded to a different hardware like FPGA which is apart from the protocol processors. The aim of this section is to cover these tasks in more depth and reveal what strategies have been adopted in this project to realize the FEC part of the system. In addition to FEC a number of other protocols and algorithms are also needed to complement the work of FEC. These are also reviewed in this section.

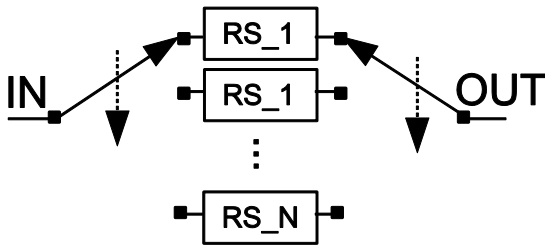


Figure 10.16
IRS coding. From [386]

10.3.1 IRS Coding

The first option for the FEC technique considered for a high-throughput wireless communication is RS code. Although RS codes are not in general as powerful as other techniques like low-density parity-check (LDPC) and Turbo codes, they have the advantage of simplicity and lower complexity. That's why they have been the first choice in this project.

A single RS decoder entity with an 8-bit symbol cannot run at a frequency higher than 250 MHz on Virtex7 FPGA. Thus, the throughput is limited to $\sim$ 2 Gbps. It means that at least 50 parallel entities are required to achieve the targeted 100 Gbps data rate. Here, a modified version of IRS codes for high-speed hardware decoding is utilized. However, the same concept can be used for construction of interleaved Bose–Chaudhuri–Hocquenghem (BCH) codes.

The general structure of the proposed IRS engine is shown in Figure 10.16. As shown in this figure, the data symbols are multiplexed between different RS decoders which causes burst errors in the incoming data-stream to be interleaved between different RS decoders. In addition, Figure 10.17 shows the proposed IRS decoder architecture optimized for the case in which the inputs and outputs accept 64-bit words and the employed RS coders are based on 4- or 8- or 12-bit symbols. According to Figure 10.17, input data is split between parallel RS structures. Each single RS entity calculates 4, 8 or 12 bits from the 64-bit word. The calculated amount of data is defined by the RS symbol size. The main reason for choosing the 64-bit architecture is hardware multiplexing supported by common serial protocols. The hardware multiplexers deserialize the data to 64 bits in most cases such as 10G Ethernet and GTH/GTX/GTZ transceivers. These transceivers can be used to interface to other devices, for example to the baseband (BB) processor. Thus, 64-bit buses are considered in the design presented in this section. This significantly reduces the complexity of the proposed data-link layer (DLL) processor.

The proposed IRS scheme has three main advantages. Firstly, interleaving improves correction of burst errors. Secondly, the interleaver can be realized as a static routing network and there is no hardware overhead for the interleaving

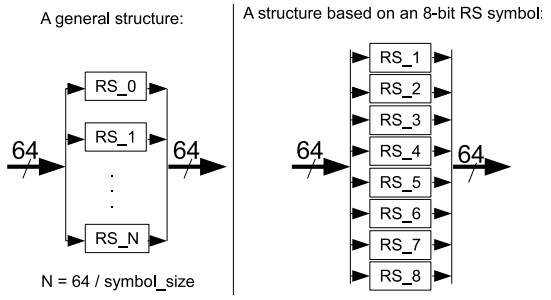


Figure 10.17
Proposed parallel RS structures for 100 Gbps IRS coding. From [386]

structure. Thirdly, IRS achieves high decoding throughput due to the parallel architecture.

The following coding schemes are considered as a base for the IRS processor: $RS(15, 13)$, $RS(255, 237)$ and $RS(4095, 4006)$ with symbol sizes 4, 8, and 12 bits. The code rates are on a similar level, and the decoders achieve optimal calculation latency in the targeted very high speed integrated circuit hardware description language (VHDL) implementation. It means that the proposed codes are selected very carefully according to practical issues and the decoding throughput has to be not lower than $1 symbol/clk$. This constraint reduces the required clock frequency, and the area required to implement the IRS engine.

In addition to FEC schemes, ARQ [387] is also one of the most important techniques used in wireless communications. It provides robustness in wireless protocols. Every time, when an incorrect frame is received, ARQ uses a return channel to inform the transmitter about the lost frame. After that, the transmitter can schedule the frame for re-transmission (Figure 10.18). The stop-and-wait-ARQ solution is inefficient. Both RF front-ends have to switch to transfer the acknowledgment (ACK) frame after each data frame transmission. Additionally, the data frame has to be fully processed and the CRC has to be recalculated before the ACK-frame can be prepared and sent. Data frame processing may introduce significant delay due to FEC and pipe-lining. That reduces transmission goodput, which can be estimated by the following formula

$$\eta = \frac{t_{data}(1 - BER)^l}{t_{overhead} + t_{data}}, \tag{10.1}$$

where l is the frame length in bits, t_{data} is the time used for payload transmission and $t_{overhead}$ is the time used for all other processing, e.g., radio switching, preamble, header and CRC transmission.

To achieve higher goodput, a different ARQ method such as selective-repeat ARQ [387] shown in Figure 10.19 has to be used. The selective-repeat ARQ repeats individual frames and uses a single block-ACK-frame [388] to

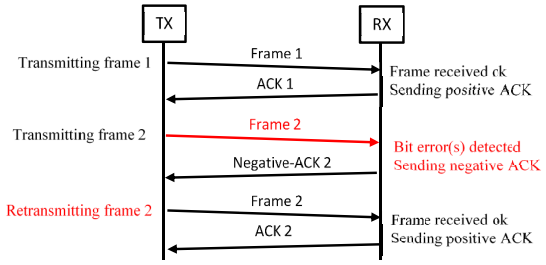


Figure 10.18
Stop-and-wait-ARQ. From [386]

acknowledge all successfully received data frames. This reduces the number of Physical Layer (PHY) turnarounds and transmitted ACK-frames.

10.3.2 Energy Efficient FEC Schemes

In addition to RS codes the possibility of using one of the other capacity-achieving codes for high-throughput wireless communications has also been examined. In an attempt to find a FEC code suitable for high-throughput communication systems operating with data-rates near 100 Gbps, various aspects of a code must be carefully inspected. Turbo codes, LDPC codes and Polar codes are three classes that are known as capacity-achieving codes, meaning that they are potentially capable of approaching Shannon capacity of the channel under some special circumstances. However, high encoding and decoding complexity of such powerful codes may appear as an obstacle for them to be used in high-throughput communication systems. Nevertheless, finding good instances of such codes and optimizing their encoding and decoding performance is an important topic and therefore the focal point of research.

Based on the extensive and advanced research conducted so far on LDPC and Turbo codes, LDPC codes better suit the requirements of high-throughput next generation networks. Here some of the advantages of LDPC codes over Turbo codes are summarized:

1. Unlike Turbo codes, LDPC codes do not require long interleaver to achieve good error performance [387];

2. LDPC codes have better block error performance [387];

3. In general, the error floor in LDPC codes occurs at higher signal to noise ration (SNR)s compared with Turbo codes [389];

4. LDPC decoders have lower latency [389] and higher adaptability to parallel architectures [389, 390] which together promise faster decoding operation.

On the other hand, the superiority of polar codes and convolutional codes

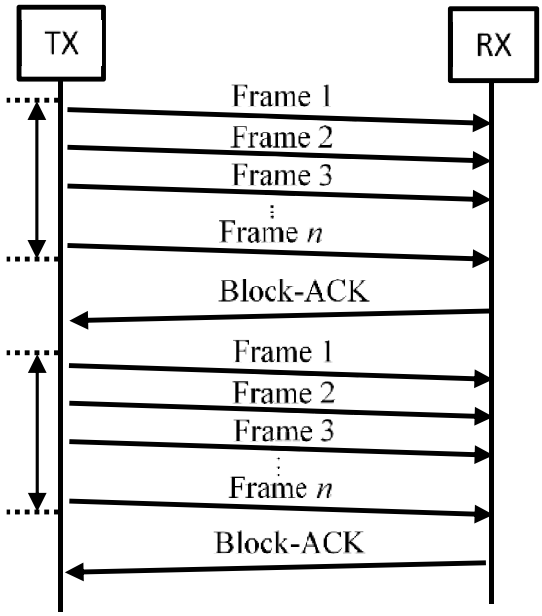


Figure 10.19
Selective-repeat ARQ. From [386]

over LDPC codes and even Turbo codes is only for short blocks [391, 392]. Relying on these conclusions, there is a seemingly good prospect for application of LDPC codes to high-throughput systems. Regarding the limited energy budget in nJ/b for wireless systems that the designer face and the high complexity that LDPC codes bring in, their entire construction, encoding and decoding operations have to be re-examined.

In particular, QC-LDPC codes [393] have been one of the examples of LDPC codes that have gained a large attention and seemt to be a suitable candidate for high-throughput communication systems. In fact, the cyclic structure of QC-LDPC codes can be exploited to simplify both encoding [394, 395, 396, 397] and decoding [398, 399, 400, 401] process of the code. Therefore, we focused on QC-LDPC codes in order to find a practical architecture tailored to high-throughput communication systems.

The generator matrix of QC-LDPC codes is not, in general, in systematic form. Therefore, finding the message corresponding to a detected codeword at the receiver is not as straightforward as it is in the case of systematic codes. In [393] two methods for finding the message from the codeword in a QC-LDPC code are proposed. The first method is a general scheme which views the problem as a set of linear equations. This method is not realizable with pure digital circuits and may be implemented by having a processor core in the design. The second method is rather an alternative scheme which makes use

Table 10.1
Non-systematic Latin-square QC-LDPC codes compliant with the proposed method for obtaining message from the codeword. From [393] © 2019 IEEE

Code	(n,k)	Code rate	#Steps
$H_{LS}(6,1,64)$	(4032,3304)	0.819	40
$H_{LS}(6,1,60)$	(4032,3308)	0.820	40
$H_{LS}(6,1,56)$	(4032,3312)	0.821	40
$H_{LS}(6,1,52)$	(4032,3316)	0.822	40
$H_{LS}(6,1,48)$	(4032,3320)	0.823	40
$H_{LS}(6,1,44)$	(4032,3324)	0.824	40
$H_{LS}(6,1,40)$	(4032,3328)	0.825	40
$H_{LS}(6,1,36)$	(4032,3332)	0.826	40
$H_{LS}(6,1,32)$	(4032,3336)	0.827	34
$H_{LS}(6,1,16)$	(4032,3448)	0.855	19
$H_{LS}(6,1,8)$	(4032,3624)	0.899	33
$H_{LS}(6,1,4)$	(4032,3792)	0.940	33
$H_{LS}(5,1,32)$	(992,750)	0.756	19
$H_{LS}(5,1,28)$	(992,754)	0.760	19
$H_{LS}(5,1,24)$	(992,758)	0.764	19
$H_{LS}(5,1,20)$	(992,762)	0.768	19
$H_{LS}(5,1,16)$	(992,766)	0.772	14
$H_{LS}(5,1,12)$	(992,790)	0.796	14
$H_{LS}(5,1,8)$	(992,814)	0.821	4
$H_{LS}(5,1,4)$	(992,878)	0.885	1
$H_{LS}(4,1,16)$	(240,160)	0.666	9
$H_{LS}(4,1,12)$	(240,164)	0.683	9
$H_{LS}(4,1,8)$	(240,168)	0.700	5
$H_{LS}(4,1,4)$	(240,188)	0.783	1
IEEE 802.16e	(1152,2304)	0.5	49

of a particular structure in generator matrices of QC-LDPC codes in order to find the message from a non-systematic codeword. The examination of a large number of non-systematic Latin squares QC-LDPC codes [402] and one of the IEEE 802.16e standard QC-LDPC codes [398] show that they all have the required structure to be usable by the algorithm. The latter method is realizable in hardware and can be implemented with a digital circuit consisting of XOR gates. Table 10.1 shows the non-systematic codes which have been examined, along with the number of required steps it takes for the method to complete. More details on the proposed method and QC-LDPC codes can be found in the corresponding reference [393].

Another proposed improvement on QC-LDPC codes is a general form of shuffling of their parity-check matrix (PCM) which can split the critical path

delay in layered decoding (LD) and therefore improve throughput by allowing higher clock rates [403]. Layered (or Turbo) decoding of LDPC codes is considered as a decoding schedule that facilitates partially parallel architectures for performing iterative algorithms based on belief propagation. It has reduced implementation complexity and memory overhead compared to fully parallel architectures and also higher convergence speed compared to both serial and parallel architectures. LD relies on the layered structure of the PCM. In LD schedule, each iteration is split into several sub-iterations, running over successive layers of the PCM. During each sub-iteration, reliability messages are exchanged between check node (CN)s of that layer and their neighbor variable node (VN)s, and at the end, the updated reliability messages are handed to the next layer. Accordingly, only a subset of CNs and VNs participate in each sub-iteration, and layers are processed successively from top to down the PCM.

The generalized shuffling method described here is to include the idea of critical path splitting of [398]. Given that the PCM consists of c layers, each with b rows, suppose that a set of integer offset values $O_1, \ldots, O_c$ related to layers of H_{qc} is selected, such that $0 \leqslant O_j < b, j = 1, \ldots, c$. For $i = 1, \ldots, b$, let $H_g^{(i)}$ be the matrix made up of $i + O_1, i + b + O_2, i + 2b + O_3, \ldots, i + (c-1)b + O_c$ rows of H_{qc}. Consequently, the new shuffled matrix $H_{qc}^{(g\_sh)}$, having matrices $H_g^{(i)}$ as its layers, will be the generalized shuffled matrix. The offset values are carefully selected according to the H_{qc} which is to be shuffled, such that in the generated shuffled PCM no column in a layer has the weight of greater than one. This will ensure the least possible path delay in the very-large-scale integration (VLSI) implementation of the LD. For example, consider the IEEE 802.16e QC-LDPC code whose base matrix has been shown in table 10.2. The PCM for this code is derived by substituting each non-negative integer with an identity matrix of size 96 cyclically shifted rightward equal to that integer [398]. The appropriate offset values for such a PCM are $[0, 8, 0, 12, 84, 0, 88, 8, 0, 16, 0, 80]$ as determined in [398]. Otherwise, when choosing all the offset values as 0 which is equivalent to performing the primary form of shuffling results in some columns of weight bigger than one in some layers and therefore longer path delay.

10.3.3 Link Adaptation

It is possible to design an algorithm that finds a trade-off between the coding overhead and the desired error correction capability. The algorithm is able to adjust the code-rate on the fly during the transmission. It takes two statistics to make a decision, whether the code-rate should be reduced, increased or remain unchanged. Specifically, the corresponding unit monitors how many segments were lost and compares the value with the code redundancy. If the efficiency degradation caused by the loss of segments is higher than the coding

Table 10.2
Base matrix of the rate-1/2 QC-LDPC code in IEEE 802.16e standard. From [393] © 2019 IEEE

-1	94	73	-1	-1	-1	-1	-1	55	83	-1	-1	7	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	27	-1	-1	-1	22	79	9	-1	-1	-1	12	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	24	22	81	-1	33	-1	-1	-1	0	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1
61	-1	47	-1	-1	-1	-1	-1	65	25	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1
-1	-1	39	-1	-1	-1	84	-1	-1	41	72	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	46	40	-1	82	-1	-1	-1	79	0	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1
-1	-1	95	53	-1	-1	-1	-1	-1	14	18	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1
-1	11	73	-1	-1	-1	2	-1	-1	47	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1
12	-1	-1	-1	83	24	-1	43	-1	-1	-1	51	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1
-1	-1	-1	-1	-1	94	-1	59	-1	-1	70	72	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1
-1	-1	7	65	-1	-1	-1	-1	39	49	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0
43	-1	-1	-1	-1	66	-1	41	-1	-1	-1	26	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0

overhead, then the code-rate is decreased. The threshold as described in [404] is derived as

$$\left(\frac{error\_segments}{all\_segmnts}\right) \leqslant \left(\frac{RS\_overhead}{RS\_block\_size}\right). \tag{10.2}$$

The reader can find more details in [404] on how the above equation can be further simplified and then realized in a hardware like FPGA.

10.4 Evaluation

All concepts presented in this chapter have been implemented and tested. This section is used to present selected results. Firstly, the protocol processing approach is evaluated concerning the protocol processing capabilities and the processing overhead. Afterward, two scenarios that use the automatic reconfiguration and protocol replacement are presented. The results corresponding to the offloaded FEC is then evaluated. The chapter ends with the results of an integration test, which combines the offloaded FEC and stream processing approach into a communication system.

10.4.1 Stream Processing Approach

The following results were obtained with the stream processing framework StrIPEs that implements the presented concept. StrIPEs was implemented on top of a zero overhead Linux that disables preemptive scheduling in order to minimize interruption from the operating system.

Table 10.3
Theoretical maximum and measured goodput, as well as the processing overhead of the protocol processing overhead (shortened from [381]).

	Data-Packet Size: 8192							
# of 10 GbE	1	2	3	4	5	6	7	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

The framework and the protocols are executed on Mellanox TileGx72 manycore boards [405], which provide 72×1 Ghz cores and 8×10 GbE interfaces. The 10 GbE interfaces are used in the following evaluation to emulate the wireless links.

10.4.1.1 Overhead and Scalability

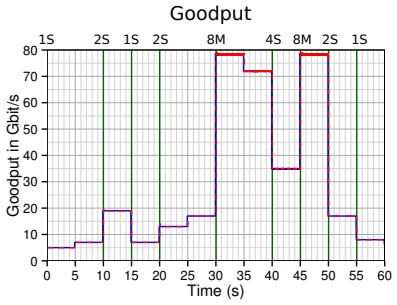
The main performance indicators for the StrIPEs framework are the maximum achievable data rate and the overhead induced by the processing concept. The theoretical maximum achievable data rate, the actually measured data rate and the processing overhead are shown in table 10.3 for a data packet size of 8192 Bytes and a stable channel without any packet loss.

The results show that the achieved data rate scale almost linear with the theoretically achievable data rate, furthermore, the processing concept introduces only insignificant overhead of maximum 0.342% for 8×10 GbE interfaces. Further evaluations of the scalability and efficiency are presented in [381, 380].

10.4.1.2 Protocol Replacement

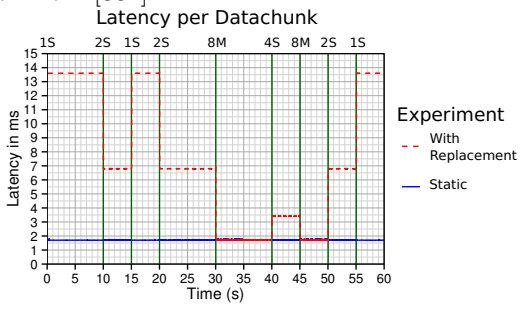
The scenarios are used to showcase the feasibility of the on-demand adaptation. The first scenario emulates a situation in which the host's desired data rate for the transmission changes over time. In this scenario the data rate is determined by the host, i.e., the host tells the communication system about the desired data rate and the communication system can change the communication protocol accordingly.

Whenever the newly desired data rate can not be fulfilled by the protocol and the number of communication channels or the number of of allotted



(a)

Figure 10.20
The goodput in Gbit/s over time of for an adapting (red) and a static (blue) processing engine. From [381]



(a)

Figure 10.21
The latency per datachunk in ms over time for an adapting (red) and a static (blue) processing engine. From [381]

resources is to high, a new better suited protocol is build and deployed. For data rates lower than 70 Gbit/s a single pipeline protocol is used. The single pipeline monopolizes the communication channels and achieves a lower and more stable latency. However, due to processing pauses, e.g., while waiting for the acknowledgement, the maximum achievable data rata is slightly below the theoretically achievable data rate. Therefore, for data rates higher than 70 Gbit/s a multi pipeline protocol is used. With the multi pipeline approach the transmission of two consecutive data items is multiplexed on the communication channel. Consequently, processing pauses of one data chunk can be hidden.

The goodput of the first 60 seconds of the transmission are shown in figure 10.20. The host starts with a desired data rate of 5 Gbit/s. After 5s the data rate changes to 7 Gbit/s, however, the communication protocol stays unchanged because it can still fulfill the processing requirements. After 10 sec-

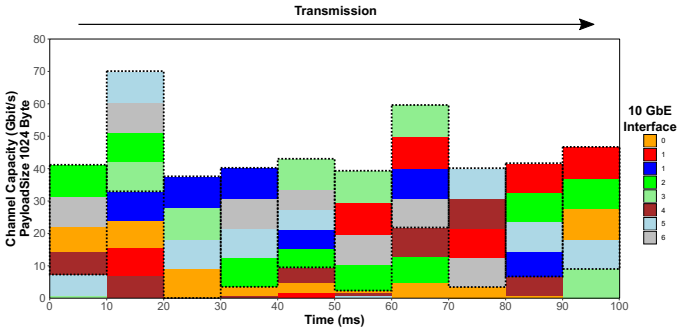


Figure 10.22
Channel capacities used for the dynamic channel bonding scenario. From [381]

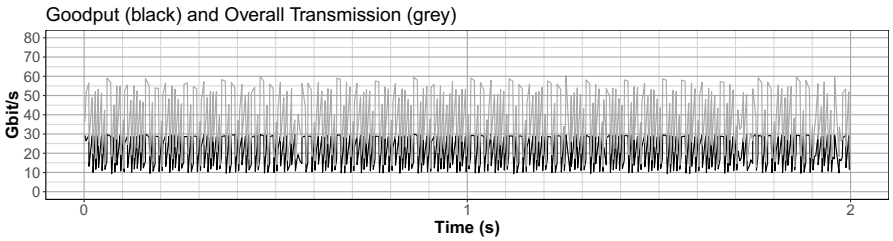
onds the protocol is replaced the first time when the desired data rate is increased to 19 Gbit/s. This continues over the course of the transmission.

For comparison the same transmission was carried with a static 80 Gbit/s multi pipeline protocol. The results show, that the on demand replacement has no impact on the achieved data rate, i.e., the on demand replacement of communication protocols can be used to react to changing data rate requirements.

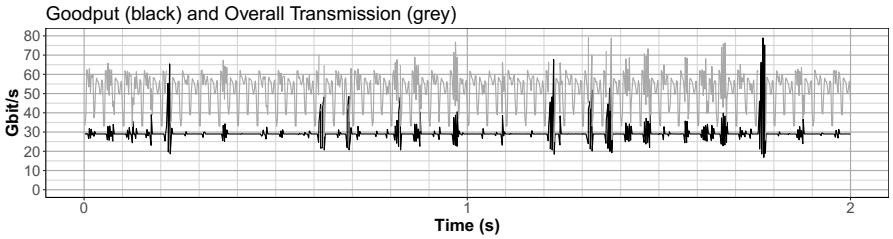
The latency results are shown in figure 10.21. One can clearly see how the latency per data chunk varies over the course of the transmission when the protocol replacement is used. This is because the number of combined channels and the allotted processing resources limit the latency. In comparison, the static 80 Gbit/s multi pipe line protocol achieves a constantly low latency. However, while wasting processing resources which are not used for the transmission.

10.4.1.3 Protocol Reconfiguration and Dynamic Channel Bonding

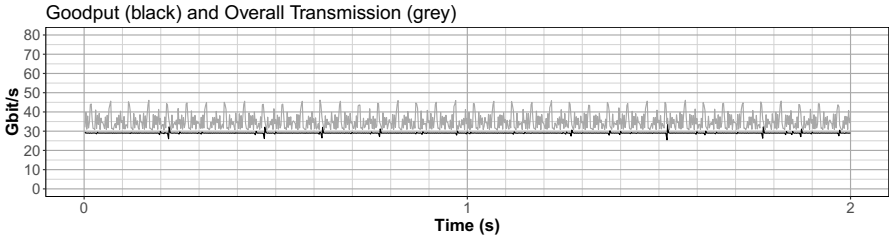
The second scenario is used to show the applicability of the protocol reconfiguration in a communication environment with unstable channels. In this scenario the host sends data with a constant data rate of 29 Gbit/s and the communication channel's qualities change in 10ms intervals. Consequently, the communication system has to reconfigure the protocol automatically whenever the achievable data rate is lower than required. This is done by dynamically bonding the minimum amount of channels that fulfill the data rate requirements and by parallelizing the protocol processing accordingly. However, this can lead to wasted resources, i.e., used processors and communication channels, in the case the quality of the bonded channel change for the better. Consequently, the protocol is also reconfigured in the case the desired data rate can be achieved with a lower resource consumption. Figure 10.22 shows the assumed channel capacities over a 100 ms timespan. The quality of the



(a) The protocol was adapted with an expected BER of 0. The resulting average goodput is 17.28 Gbit/s of desired 29 Gbit/s.



(b) The protocol was adapted for the worst case scenario. During the transmission a goodput of unstable 29 Gbit/s was achieved. The resource consumption is high due to the worst case adaptation.



(c) The protocol was adapted on-demand depending on the channel capacities. The transmission resulted in a stable goodput of 29 Gbit/s. The resource consumption was minimal depending on the channel capacity.

Figure 10.23
The goodput and throughput of three transmissions given a desired data rate of 29 Gbit/s and the channel qualities presented in figure 10.22. The transmissions shown in figure a) and b) were conducted with statically adapted protocols, the transmission shown in figure c) was conducted with a protocol that was adapted on-demand depending on the channel capacities. From [381]

channels changes every 10 ms. These channel capacities are reported to the embedded many core boards.

The scenario was evaluated with a static multi pipeline protocol that was configured for a desired data rate of 29 Gbit/s but was able to use all communication channels (S29), a statically configured multi pipeline protocol that was configured for the theoretically maximum data rate of 80 Gbit/s (S80), and a dynamically reconfigured multi pipeline protocol that uses the reported channel capacities for the selection of channels and for the estimation of the necessary parallelization (dynamic).

The results of the evaluation are shown in figure 10.4.1.3 to 10.4.1.3. The results for S29 (see figure 10.4.1.3) show an unstable goodput that does not provide the desired data rate for the application. Furthermore, the results show a high throughput, i.e., the sum of goodput and retransmissions. This is due to the high number retransmission caused by using broken channels. The results for S80 (see figure 10.4.1.3) show that the desired data rate could be reached most of the time. This is because the protocol was configured for a data rate of 80 Gbit/s, i.e., the resulting protocol was theoretically able to handle the expected packet loss. However, the transmission also shows spikes that refer to data rate breakdowns. Finally figure 10.4.1.3 shows the results for the automatic reconfiguration according to the reported channel capacities. One can see that the desired data rate was provided in a stable manner with less retransmissions. This is because the protocol was always configured to use the minimum amount communication channels. Consequently, completely broken down channels were not used, which considerably reduced the number of necessary retransmissions.

10.4.2 FEC Results

The design consisting of the protocols and techniques outlined in section 10.3 has been fully implemented and synthesized using Genus^{TM} software, and its layout has been made with Innovus^{TM}. Moreover, the following optimization measures have been performed on the Netlist and layout, in order to achieve the reported throughput of 165 Gbps with energy efficiency of 4.47 pJ/bit:

- The dual port static random access memory (RAM) memories needed for RS implementation [406] are replaced by flip-flop (FF) arrays. This solution sounds insane from the power and area point of view, but the memories are the main bottleneck of throughput in our design. Moreover, planning a chip with memories is more difficult than placing pure logic alone. In our case, we need to place 64 memories, each of the size of 256×8 bits. Replacing the memories with FF arrays increases the clock speed from 600 MHz up to 2100 MHz, which corresponds to the throughput improvement from 67 Gbps up to 235 Gbps. However, the chip area also increases from 0.57 mm^2 to 1.02 mm^2 and the power from 0.286 W up to 3.5 W.

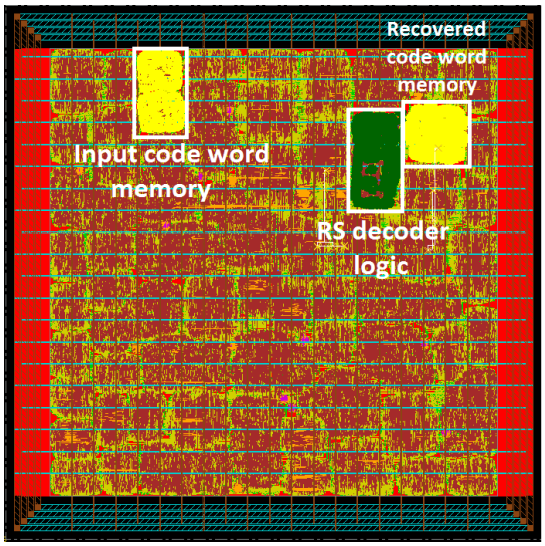


Figure 10.24
ASIC layout of the implemented processor (receiver) in 28 nm CMOS technology. From [407] © 2019 IEEE.

- In order to reduce the energy dissipated in the FF arrays emulating the memory blocks, the technique of clock gating is adopted which causes a reduction in the very high dynamic power. In each clock cycle, we read and write just a single byte to each FF memory. This means that we access only $\sim 0.19\%$ of the total memory registers in each clock cycle. Thus, we can significantly reduce the power by inserting clock gates and deactivating $\sim 99.81\%$ of the memory registers. Although the clock gates increase area by $\sim 0.02\ mm^2$ and reduce the clock by 600 MHz (from 2100 MHz downto 1500 MHz), the power is desirably reduced to 0.928 W from the initial 3.5 W.

- To reduce the static power dissipated by the chip, multi-threshold voltage optimization is performed. In short, for all critical paths, the transistors with the lowest voltage switching threshold are inserted, while for non-critical paths transistors with a high threshold and reduced leakage are used. This reduces the power from 928 mW to 602 mW. As a result, the chip area remains almost unchanged, but the clock frequency is further reduced by $\sim$ 200 MHz (from 1500 MHz downto 1300 MHz).

The layout of the chip is shown in Figure 10.24. We use a doubled VDD-VSS power ring around the placed logic. The input-output (IO) pads are excluded from the area and power analysis. We highlighted a single RS decoder entity and its belonging codeword memories. The input memory is placed close

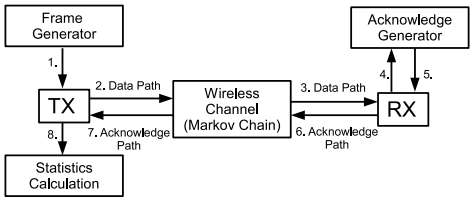


Figure 10.25
Matlab simulation model. From [386]

to the chip edge due to the input signals routing. The corrected codeword, after fixing the evaluated error is stored in the memory placed next to the decoder. It is possible to reduce the memory size by 25% by removing the bypass First In First Out (FIFO)s, which are used to shift out the originally received codeword, in the case when the decoder cannot correct all the bit errors.

10.4.2.1 DLL Frame Format

Figure 10.25 shows a Matlab simulation of the proposed DLL processor. TX and RX models use all the aforementioned techniques, including aggregation, fragmentation, FEC, and a hybrid ARQ with link adaptation. As a default scenario, a point-to-point communication between a TX and a RX is performed, as depicted in figure 10.25. TX transmits data to RX with the data-rate of 100 Gbps. Then, the sender waits for a feedback from the receiver to figure out whether the frames have been received correctly.

To support two-way communication, time division duplex (TDD) is applied. In short, the sender stops transmissions of the data frames after a predefined time, and allows the receiver to send acknowledgments. Figure 10.26 shows a Finite State Machine (FSM) that controls the transmission.

The transmitter sends a predefined number of frames, and each frame is carrying a predefined number of data-fragments (Figure 10.27). After that, a single ACK-frame is requested, and a timer is started. If timeout occurs, then the ACK request frame is retransmitted. This procedure is repeated until the ACK is received successfully. All mentioned parameters of the FSM and frame format are fully adjustable.

10.4.2.2 Energy consumption

As mentioned in the introduction, energy and power consumption are one of the most critical parameters of the high-speed transceivers. In our case, the energy and power depend on channel BER and selected FEC code. The energy is mostly consumed by the RS decoders and it is indeed related to the code-rate curve shown in figure 10.28.

Figure 10.29 shows the variation in energy consumption versus BER.

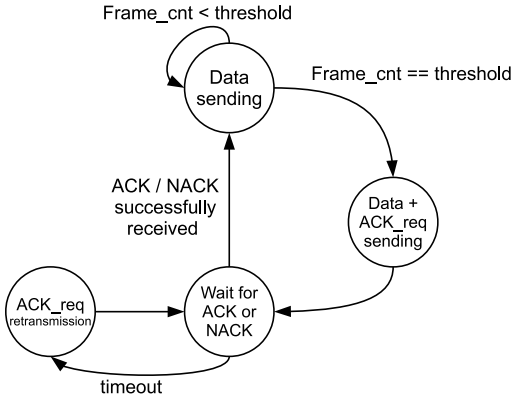


Figure 10.26
FSM of the transmitter. From [386]

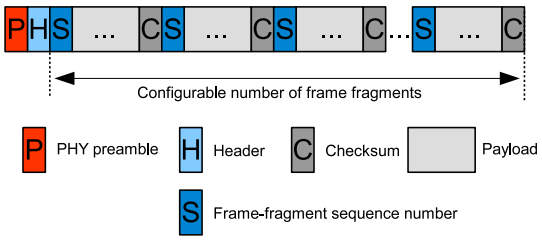


Figure 10.27
Frame format used in the simulation model. From [386]

For BER $< 1e-5$ which is correspondent to the highest code-rate of $RS(255, 253)$, the processor consumes extremely low DC-power of 29.7 mW, equivalent to 0.22 pJ/bit. With the increase of BER, the DC-power also increases and saturates at 602 mW, or equivalently 4.47 pJ/bit. This high-power mode corresponds to the lowest code-rate of $RS(255, 223)$.

It should be noted that power consumption is dependent upon the number of Galois field multiplications

$$10\lfloor\frac{255-k}{2}\rfloor^2 + 771\lfloor\frac{255-k}{2}\rfloor - 255, \tag{10.3}$$

and additions

$$6\lfloor\frac{255-k}{2}\rfloor^2 + 764\lfloor\frac{255-k}{2}\rfloor, \tag{10.4}$$

which in the case of $RS(255, k)$ decoding are asymptotically $O(n^2)$.

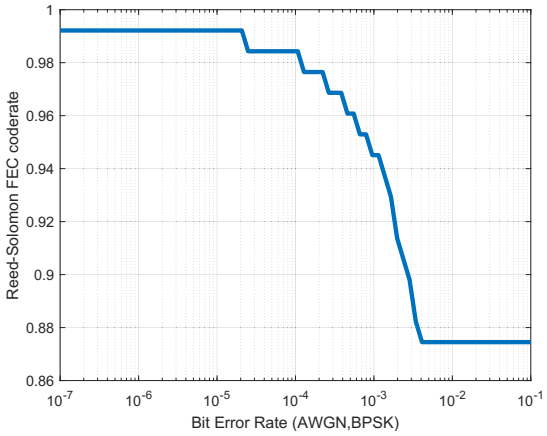


Figure 10.28
Link adaptation scheme. The algorithm changes the FEC code-rate according to the internally estimated fragment error rate. From [407] © 2019 IEEE

10.4.2.3 Voltage Scaling

Further saving of energy may be made by adjusting the throughput, clock speed, and voltage. The voltage range for the targeted process is 0.8 - 1.1 V. In our design, the clock speed scales almost linearly with the voltage (Figure 10.30), which is not the case for LDPC decoders realized in comparable technologies [408, 409]. On the other side, for the energy per bit in the targeted range of 0.8 - 1.1 V, a quadratic function fits the points more precisely. Both fitting curves are as follows:

$$Throughput\ [Gbps] \approx 313.42x - 199.81, \tag{10.5}$$

$$Energy\,per\,bit\ [pJ/bit] \approx 3x^2 + 1.22x - 0.509, \tag{10.6}$$

where $x \in [0.8, 1.1]$ represents the chip voltage.

In our case, we need to reduce the throughput to ∼115 Gbps at ∼ 1.01 V to achieve the limit of ∼ 3.8 pJ/bit at BER ≈ 6.3e-2 with $RS(255, 223)$. The BER value of 6.3e-2 is the lowest achievable BER for additive white Gaussian noise (AWGN) channel as well as the worst case from the energy consumption point of view. Assuming the lowest possible voltage of 0.8 V, the processor achieves 50.4 Gbps and consumes max. 2.38 pJ/bit.

10.4.2.4 Comparison with other Published Work

To the best of our knowledge, there is not any comparable work providing similar comprehensive functionality to our implementation. Therefore, we compare our work to existing high-speed LDPC and polar decoders (table 10.4).

Table 10.4
Performance comparison of FEC and DLL processors. From [407] © 2019 IEEE

	[410]	[411]	[412]	[409]	[413]	[414]	[408]	[415]	Current work
Technology	ST Micro. 65nm SVT CMOS	40nm G CMOS	40nm LP-CMOS	28nm UTBB FDSOI	28nm CMOS	ST Micro. 28nm FDSOI	28nm FDSOI	28nm	GlobalF. 28nm SLP CMOS
Estimation stage	Post layout	Post physical synthesis	-	-	-	Post synthesis	Post layout	Post layout	Post layout
Design	ASIC	ASIC	ASIC	ASIC	ASIP	ASIC	ASIC	ASIC	ASIC, 128-bit data bus
Voltage [v]	1.2	0.9	1.1	1.07	0.9	0.9	0.6-0.9	0.9	0.8-1.1
Freq. [MHz]	257	500	220	260	470	-	-	451	1300^c 1000^d 750^e 450^f
FEC	LDPC 802.11ad	LDPC 802.11ad	LDPC 802.11ad	LDPC 802.11ad	LDPC 802.11ad	LDPC (30000,26786)	LDPC 802.11ad	Polar code (1024,869)	IRS
Soft-decision decoding	YES, 4-bit	YES, 5-bit	YES, 5-bit	YES, 5-bit	YES	YES, 5-bit	YES	YES	NO
Code-rate	0.813	0.813	0.5	0.5	0.5	0.893	-	0.848	0.875
Eb/N0 [dB] @ BER 1e-5 AWGN	∼13 64-QAM*	-	∼3.5 BPSK*	∼3.5 BPSK	∼3.5 BPSK	∼4.5 BPSK	-	-	∼5.5*∗ (BPSK) ∼14*∗ (64-QAM)
Throughput [Gbps]	160.8	5.6	6.2	12	18.4	200	160^a 57.1^b	7.8	145.5 111.9 83.9 50.4
Chip area $[mm^2]$	12.09	0.16	0.8	0.63	0.78	3.73	2.8	0.35	1.04
Energy per bit $[pJ/b]$	32.49	17.7	32.9	30	18	1.5	6^a 2.9^b	1.41	4.47^c 3.69^d 3.04^e 2.38^f
Normalized throughput $[gbps/mm^2]$	13.9	35	6.16	19.04	23.6	53.6	57.14^a 20.39^b	22.28	139.9^c 107.6^d 80.7^e 48.5^f
Functionality	FEC decoder	FEC decoder	FEC decoder	FEC decoder	FEC decoder	FEC decoder	FEC decoder	FEC decoder	FEC dec., FEC enc., HARQ-I, link adaptation, aggregation, fragmentation

a) at 0.9 V; b) at 0.6 V; c) at 1.1 V and RS(255,223); d) at 1.0 V and RS(255,223); e) at 0.9 V and RS(255,223); f) at 0.9 V and RS(255,223); -) not specified or unclear; ∗) channel not specified, AWGN assumed; ∗∗) including 0.55 dB fragmentation gain.

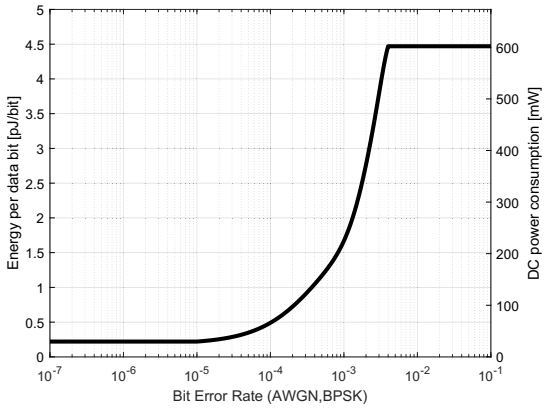


Figure 10.29
Average energy consumption per bit for the DLL receiver as a function of BER. From [407] © 2019 IEEE

As stated before, most of the chip resources are utilized by FEC, and hence, such a comparison is fair. Compared to high-speed LDPC at similar code-rates [410, 414], our hard-decision RS loses ~ 1 dB gain. It rather leads to a smaller chip area and significantly higher throughput normalized to the area. In our case, we achieve up to 140 Gbps mm^2, and this value is at least 2 times higher than for LDPC decoders. We require only 1.04 mm^2, but LDPC requires 2–6 mm^2 at the same data-rate. Moreover, we integrate a complete DLL processor, not only FEC decoders. Thus, we believe that the 1 dB gain loss of the proposed hard decision method is mitigated by the superior area efficiency.

The other very important design parameter is energy efficiency. Our implementation can work with energy as low as 2.38 pJ/bit at 0.8 V, which is a moderately good result. The LDPC [414] and polar [415] solutions require only 1.5 pJ/bit and 1.42 pJ/bit, respectively. Other published 28 nm LDPC decoders consume $2.9 - 30$ pJ/bit [409, 413, 408].

The data-rate of our solution can be additionally improved. Currently, we process 128 bits/clk and use 1.04 mm^2 chip area. Thus, there are no technical barriers to use more computation entities in parallel and process more than 165 and 145 Gbps at $RS(255, 253)$ and $RS(255, 223)$, respectively.

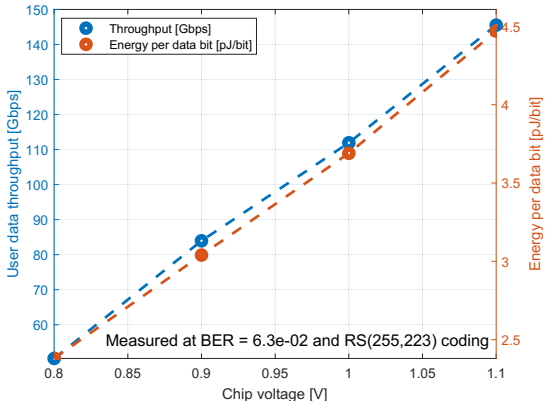


Figure 10.30
Throughput and energy per data bit as a function of chip voltage in the range of 0.8 to 1.1 V. From [407] © 2019 IEEE

10.5 Conclusion

In this chapter we presented a scalable approach for processing communication protocols with ultra-high data rates. The approach seamlessly combines a software-based protocol implementation for higher-level protocol processing tasks with efficient hardware implementations for low-level tasks.

Using a software approach for high-level protocol processing tasks eases protocol implementation and testing, while using custom hardware implementations for tasks unsuitable for software implementations reduces the stress on processing hardware. The latter mainly includes a dedicated method for FEC along with a link customization scheme to adjust the code rate in relation to the current status of the link. These tasks were specifically implemented by hardware description language (HDL) and synthesized in 28 nm CMOS technology.

After the protocol tasks have been distributed to the most suitable processing hardware, protocol processing must be parallelized at all levels. In order to reduce the complexity associated with parallelization of a communication protocol, a design process was conceived to further facilitate implementation. The design process uses the stream processing paradigm to separate the communication protocol into independent and parallelizable protocol tasks. In addition, it has been shown that communication protocol processing can be analyzed similarly to soft real-time problems, resulting in a statically parallelized protocol implementation capable of providing the desired data rate for specific communication conditions.

Such a static protocol implementation is sufficient for scenarios where communication conditions such as BER and communication requirements such as the desired data rate are fixed and known at compile-time. However, this is not the case for wireless communication scenarios. To avoid using unsuitable protocol implementations when communication conditions change, the implementations must be adapted at runtime.

This was achieved with an adaptation process that implements the changes in parallel with the transfer and then switches to the new implementation when it is ready. In this way, the impact of the protocol-adaptation on the transfer is minimal. To further reduce the impact, each adaptation can result in two implementations being used simultaneously. The first implementation is used until all data originally sent with this implementation is fully processed, while the new implementation is already used for the current transmission. This way, it can be avoided that the current transmission comes to a standstill until all old data has been transmitted.

In order to avoid the necessity of designing implicit per-protocol adaptation strategies, a new template language for describing communication protocols was developed. The language provides general means to describe communication protocols as templates that can be automatically instantiated and parallelized.

Finally, all presented concepts were evaluated with a newly designed data link protocol for ultra-high wireless communication. The protocol was implemented and parallelized with the presented design process. It was shown that the design process allows a straight-forward analysis and implementation of the communication protocol. The resulting protocol implementation and the parallelization derived from the analysis were evaluated in several scenarios [416, 380, 385, 381]. In summary, the evaluation showed that wireless communication with ultra-high data rate with low overhead can be performed with the presented approaches.

At lower levels the use of different channel coding techniques was investigated. These included both the prominent conventional RS code and a special class of LDPC codes known as QC-LDPC codes. While RS codes are not as powerful in detecting and correcting errors as LDPC codes, they are of less complexity in coding and decoding, and this gives much better results in terms of energy efficiency when used as FEC procedures. However, for the QC-LDPC codes, a shuffling method has been introduced that further reduces their decoding complexity, paving the way for their use in next generation, high throughput wireless networks. It should be noted that the low-level protocol tasks implemented separately on dedicated hardware have also been synthesized for a 28nm CMOS technology.