ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Roman Trüb

# Generic Functional Blocks for FPGA-based Network Nodes

Semester Thesis SA-2015-08
March to May 2015

Advisor: Dr. Markus Happe, markus.happe@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

**Acknowledgements**

I would like to thank my project advisor Dr. Markus Happe for his support and his valuable feedback during the semester thesis. Furthermore I would like to thank Prof. Dr. Bernhard Plattner for giving me the opportunity to write this semester thesis in his research group.

**Abstract**

At ETH an FPGA-based prototype platform called EmbedNet has been developed which implements a flexible protocol stack architecture for network devices. The functional blocks of this protocol stack can be dynamically interconnected and individually mapped to either hardware or software. This new approach allows to adapt the protocol stack to current communication requirements and networking conditions. This leads to more efficiency in terms of processing time and energy consumption compared to today's static protocol stack architecture of the Internet.

Because even a single functional block requires a quite high implementation effort not many functional blocks have been implemented so far. Therefore it wasn't possible to evaluate the performance of larger dynamic protocol stacks on the EmbedNet platform so far. In this semester thesis a generic functional block is developed which emulates the forwarding behavior (processing delay and drop rate) of real functional blocks. With multiple instances of this generic functional block it is possible to emulate the forwarding behavior of arbitrary protocol stacks. This thesis presents the first performance evaluation of the EmbedNet platform which uses up to six functional blocks.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

Nowadays the requirements of network devices vary heavily. For some applications the throughput is the biggest issue, whereas for other applications security is the main concern. For mobile devices high battery consumption is a key problem. Nevertheless the static protocol stack architecture (depicted in Figure 1.1 on the left) is still widely used even though it lacks flexibility to adapt to these needs. A packet in a static protocol stack needs to pass all protocol layers even if some of them aren't used in a certain scenario. This results in inefficient implementations in terms of throughput and energy consumption.

That's why there is a new approach using a dynamic protocol stack (DPS) architecture (depicted in Figure 1.1 on the right). In the DPS architecture the network functionality is split into so called *functional blocks* (FBs). A functional block for security could for example implement an encryption [7]. Another functional block providing reliability could for example implement a continuous repeat request protocol [2]. The functional blocks can be dynamically linked to form an arbitrary protocol stack. This reconfigurable protocol stack is continuously adopted to the current needs and therefore reduces the overhead. For example if a connection is known to be reliable anyway one can omit the functional block for reliability.

Furthermore each functional block can be dynamically mapped to hardware or software. For many functional blocks the processing in hardware is faster by orders of magnitude compared to software. But in most implementations the area for hardware is limited. In software the number of functional blocks is nearly unlimited but on the other hand the processing is slow and all functional blocks have to share one or a few processors whereas the functional blocks mapped to hardware run in parallel and can process the packets in a pipelined fashion.

The EmbedNet prototype platform is an FPGA-based implementation of the dynamic protocol stack architecture. It is designed to evaluate functional blocks and dynamic protocol stacks in different scenarios. The main drawback of the EmbedNet platform is the large implementation effort for a single functional block. A new functional block needs to be implemented in hardware and software separately since there is no automatic conversion process. In addition each functional block needs to be evaluated thoroughly in order that it doesn't introduce any instabilities in the EmbedNet system. These reasons explain why there exist only a few implementations of functional blocks up to now. Therefore it's not feasible to investigate larger more complex dynamic protocol stacks on the EmbedNet platform.

One possible solution to this problem is the use of a generic functional block. A generic functional block emulates the forwarding behavior (in time and drop rate) of an arbitrary functional block. Multiple instances of a generic functional block can then emulate a complex dynamic protocol stack. This allows for a performance evaluation without the need to implement more functional blocks.

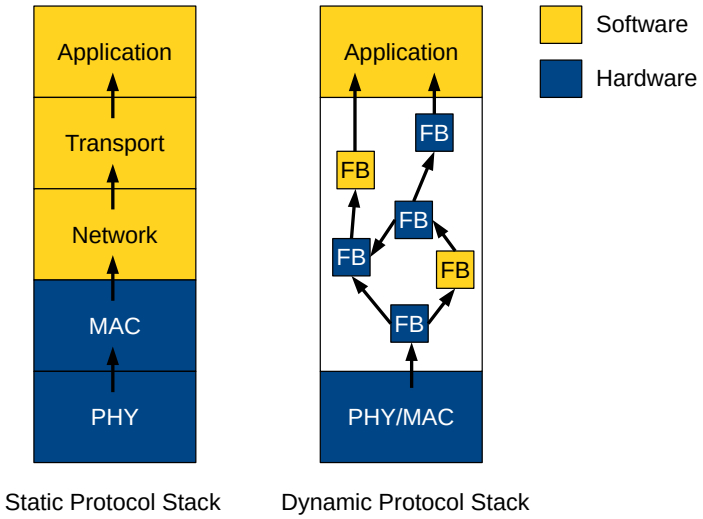



Figure 1.1: Static and Dynamic Protocol Stack Architecture

## 1.2 Goal

The goal of this thesis is to first implement a generic functional block and then to use this functional block to evaluate the performance of the EmbedNet platform. The functional block should be implemented in hardware and software featuring the same functionality.

## 1.3 Related Work

This thesis evaluates the EmbedNet platform [5]. The EmbedNet implements the DPS architecture and proposes a way to continuously adapt the protocol stack and the hardware/software mapping to the environment by using a self-aware network node architecture. In addition it shows how two network nodes negotiate a common protocol stack. An important part of the hardware implementation of the EmbedNet is the Network on Chip (NoC) which has been developed by Huber [4]. The NoC is used to forward packets inside the EmbedNet between different functional blocks mapped to hardware.

Other related work are implementations of application specific functional blocks for the EmbedNet. Deragisch [2] developed two functional blocks. One implements a computationally expensive Huffman compression algorithm and the other one implements a reliability protocol which is exemplary for flow control. For both functional blocks Deragisch compares the execution time in hardware and software. Furthermore he implemented a state transition mechanism, which is necessary to move the functional blocks between hardware and software. Kronig addresses security in his thesis [6] and presents an implementation of a functional block featuring an intrusion prevention system (IPS). Yang [7] implemented a functional block which provides AES encryption. In contrast to the described functional blocks my generic functional block does not provide any functionality useful for an application. Instead it's designed to emulate the forwarding behavior (processing delay and drop rate) of an arbitrary functional block.

An application of the DPS architecture is presented by Happe et. al. in [3]. A smart camera network represents the Internet of Things. In two case studies it is shown how to track an object over multiple views of cameras and that an adaptive hardware/software mapping can save hardware resources. However the protocol stacks are limited to a single AES functional block, an AES encryption/decryption module. In this thesis I propose a generic functional block which can be used to form a protocol stack of arbitrary complexity. This allows for an extensive evaluation of the underlaying EmbedNet platform.

## 1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 describes the concept of a performance evaluation using a generic functional block. Chapter 3 gives an overview of the EmbedNet platform and explains the design and implementation of the generic functional block. Chapter 4 shows results of the validation of the generic functional block in action. The results of the evaluation of the EmbedNet are presented in Chapter 5. Chapter 6 concludes the thesis and discusses future work. Appendix A describes how to use the generic functional block which has been developed in this thesis. The time schedule of the thesis can be found in Appendix B. The original project description is included in Appendix C.

# Chapter 2

# Methodology

## 2.1 Generic Functional Block

The generic functional block is implemented in hardware as well as in software. Both versions provide the same basic functionality. The generic functional block emulates the forwarding behavior of a real functional block in terms of delay and drop rate as shown in Figure 2.1. A generic functional block receives a packet, then stores the packet and forwards the unchanged packet after a certain time interval. This emulates the processing of a real functional block. A generic functional block can also drop an arriving packet with a predefined drop rate. This is useful for example to emulate the behavior of a functional block that implements an intrusion prevention system (IPS).

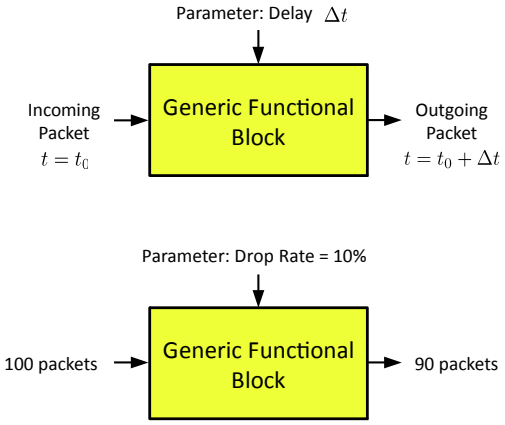



Figure 2.1: Functionality of the generic functional block

To configure the generic functional block there are three main parameters:

- Delay per Packet
- Delay per Byte
- Drop Rate

These parameters can be configured for each instance of the generic functional block separately. The delay per byte is useful if the processing time of the real functional block to be emulated depends on the packet length. The drop rate is related to all arriving packets not to a time interval in which the packets arrive.

Many additional parameters could have been selected to allow for a more realistic emulation of real functional blocks. Some ideas are described in Section 6.2. The reason why only the pa-

rameters described above have been implemented is that a bottleneck in the current EmbedNet prototype implementation would hide most of the details of further parameters.

The interconnection of the generic functional blocks is realized with an address-parameter which identifies the next generic functional block. Each generic functional block simply forwards the packets to the generic functional block corresponding to this address. More details about the addressing can be found in Sections 3.1.4 and 3.3.5.

## 2.2 Evaluating using Generic Functional Blocks

The generic functional block helps to evaluate the behavior of large complex dynamic protocol stacks with many different functional blocks. With this concept it is not necessary that all implementations of the functional blocks are available. Several instances of the generic functional block can be interconnected to form arbitrary dynamic protocol stacks. The idea is that a statistical description of the behavior (delay and drop rate) of real functional blocks is used to parametrize the generic functional blocks. With this a first rough estimation of the performance of the complex dynamic protocol stack on the EmbedNet can be obtained.

In this thesis I first investigate the maximum packet rates with simple dynamic protocol stacks consisting of up to one hardware and/or one software generic functional block. This allows to identify the performance bottleneck of the EmbedNet platform. Afterwards I measure the packet throughput of more complex settings with two competing protocol stacks as shown in Figure 2.2. In these scenarios I examine the fairness in allocation of the resources shared by both protocol stacks.

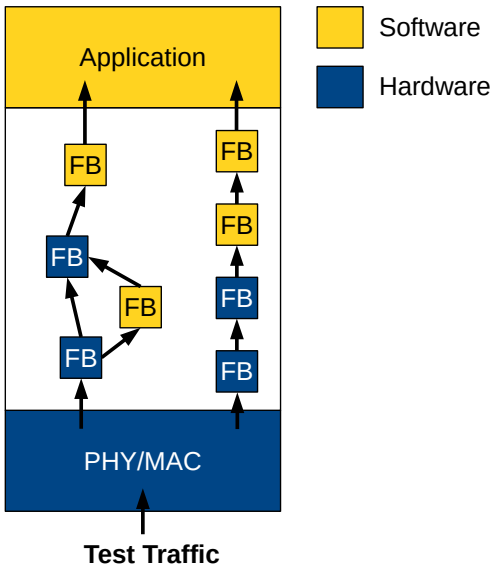



Figure 2.2: Two parallel protocol stacks

The generic functional block can also be used to act as an artificial bottleneck or a black hole. This might be useful for debugging purposes in future work for the EmbedNet but is not covered in this thesis.

# Chapter 3

# Implementation

## 3.1 EmbedNet Platform

The EmbedNet (depicted in Figure 3.1) is a FPGA-based prototype platform which implements the dynamic protocol stack architecture. For the implementation the Xilinx ML605 evaluation board with a Virtex-6 FPGA is used. The FPGA is SRAM based, i.e. the FPGA configuration is lost on power down and needs to be downloaded to the FPGA each time it is powered on. In the following sections I explain the components of the EmbedNet.

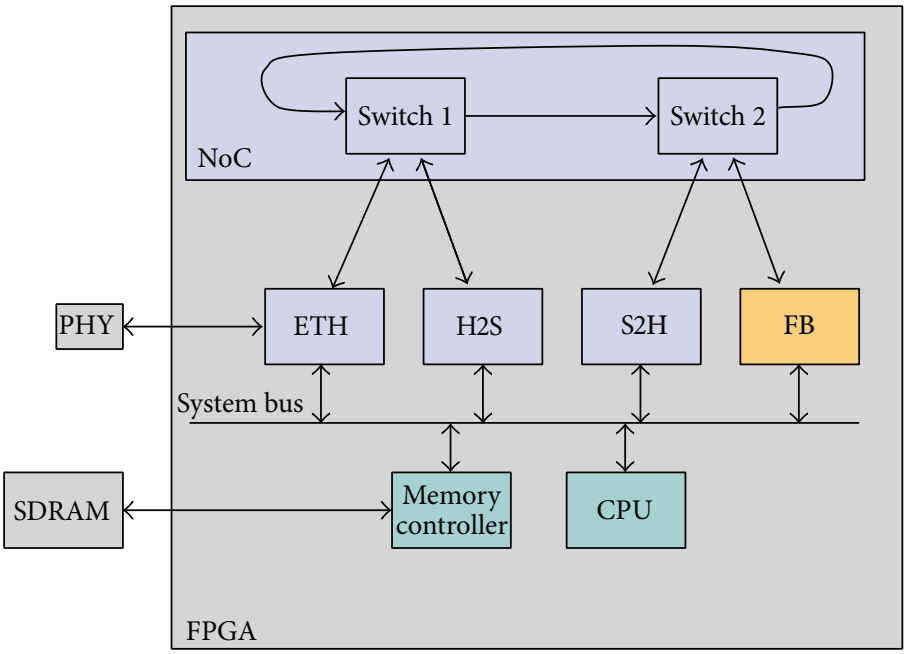



Figure 3.1: EmbedNet Design ([5], modified)

### 3.1.1 Communication

The physical Ethernet interface (PHY) provides connectivity to other network nodes.

The network on chip (NoC) connects all hardware functional blocks inside the EmbedNet node. It is implemented as a unidirectional ring network. The advantage of this topology is the simple architecture. A major drawback is however that all internal communication is broken if a single functional block blocks the network on chip.

### 3.1.2 Hardware Functional Blocks

There are three functional blocks that are always present in hardware: ETH, H2S and S2H. The ETH functional block interfaces with the physical Ethernet interface and translates the external header of a packet to the internal header and vice versa (for detailed explanation see Section 3.1.4). The H2S and the S2H block represents the connection between hardware and software. Every packet that crosses the hardware/software boundary must pass through one of them. The H2S block is used to forward packets from hardware to software, the S2H block is used to forward packets from software to hardware. In addition the S2H block acts as black hole for traffic arriving from the hardware side.

Furthermore there is a varying number of slots for functional blocks (FB). Each FB block can implement any functional block. Partial reconfiguration allows to reconfigure only the area of the functional block while maintaining the rest of the FPGA configuration intact. In my thesis I don't use this feature because I only use one type of functional block, the generic functional block. Therefore the FPGA configuration is always generated as a whole.

### 3.1.3 Software

To run software on the FPGA there is a soft core CPU called MicroBlaze. A soft core CPU is built from configurable logic blocks (CLBs) from the FPGA rather than it is physically implemented as a structure in the silicon. The MicroBlaze processor runs with a clock frequency of 100 MHz. On this soft core CPU an embedded Linux operating system is running. The CPU is connected to external SDRAM memory as well as non-volatile Compact Flash memory on the evaluation board. The Compact Flash Memory is used to store the executables of the software and the collected measurement data.

### 3.1.4 Addressing & Identifiers

Each packet arriving at the EmbedNet node consists of the payload and an external header which contains a MAC header and an 8 byte hash. The MAC header is used to address the network node. The hash identifies the protocol stack which is used to process the packet. At the ETH block this external header is then replaced by an internal header. The details of the header translation are depicted in Figure 3.2. The internal header, also called the NoC header, is used for addressing functional blocks inside the network on chip. It contains the ID of the destination switch (global address) and the ID of the functional block (local address). In addition a so called *Information Dispatch Point (IDP)* is used. Originally each IDP was meant to identify a functional block. In this thesis the IDP only identifies the protocol stack which shall process the packet. In the ETH block every hash is mapped to exactly one destination IDP. This mapping is configured by software.

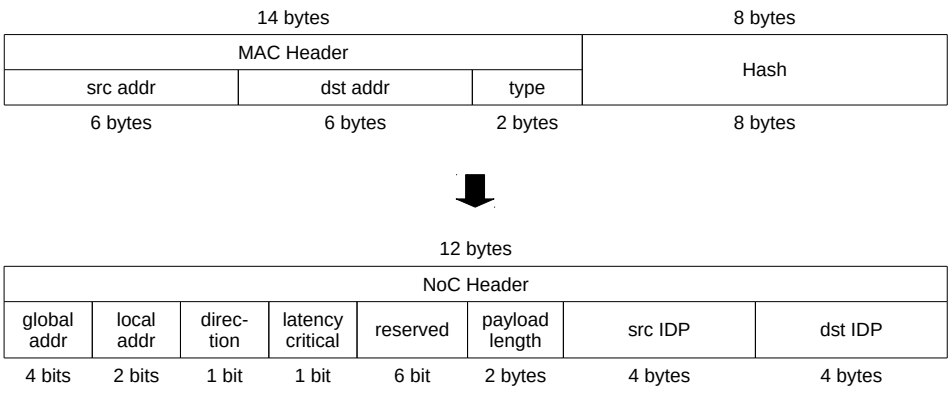



Figure 3.2: Translation of the header in the ETH block

## 3.2 Hardware Design

The hardware design of the generic functional block is implemented in VHDL and is translated into a hardware configuration for the FPGA using the Xilinx Platform Studio (xps) software.

The block diagram of the hardware generic functional block is depicted in Figure 3.3. The functional block is built from multiple finite state machines (FSMs), a FIFO and the ReconOS interface [1]. The ReconOS interface allows to set the parameters of the hardware generic functional block from software. Furthermore parameters or statistical data can be queried from software.

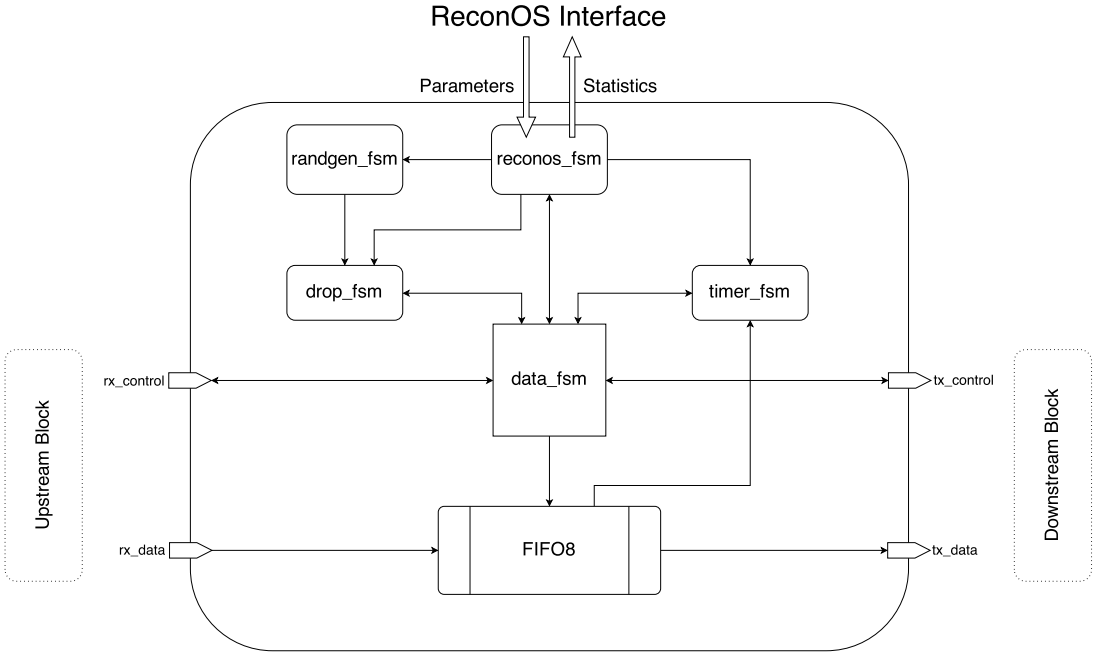



Figure 3.3: Hardware design of the generic functional block

A packet which is forwarded to this functional block announces its arrival with a control signal to the `data_fsm`. If the block is ready the `data_fsm` queries the `drop_fsm` for a drop decision. If the packet should be dropped the `data_fsm` indicates to the upstream block that the packet can be sent but does not instruct the FIFO to read in the data bytes. This means the packet is never stored. If the packet shouldn't be dropped the `data_fsm` instructs the FIFO to store the incoming data bytes and notifies the upstream block that the packet can be sent. Once the packet is fully stored in the FIFO the `data_fsm` instructs the `timer_fsm` to start counting. When the time interval defined by the parameters has elapsed the `timer_fsm` notifies the `data_fsm`. Then the `data_fsm` informs the downstream block that the packet is ready to be sent. If the downstream block acknowledges this signal the `data_fsm` instructs the FIFO to send out the packet.

### 3.2.1 ReconOS Interface

Each hardware functional block which uses an ReconOS interface is visible to other processes as a so called *delegate thread*. This delegate thread represents the hardware thread in software. For the communication with the software a message box (mbox) from the ReconOS library is used. A mbox is a message queue where multiple threads (in hardware or software) can put or get messages.

To use the ReconOS interface I implemented the `reconos_fsm`. In the FSM several commands are defined in order to set the parameters and to get statistical data to and from the hardware

generic functional block. If all necessary parameters have been set the `reconos_fsm` sets a signal which indicates to the other FSMs that it's safe to start operating.

### 3.2.2 Data FSM

The `data_fsm` implements the LocalLink interface[1] to communicate with upstream and downstream hardware modules that send and receive packets. It ensures that an arriving packet is only accepted and stored in the FIFO if the generic functional block is ready. That's the case if the parameters are set and no other (partial) packet is currently stored inside the generic functional block. Similar for the outgoing port a ready signal is set when the generic functional block has finished with emulating the processing of the packet. The packet is not sent until the downstream module indicates that it is ready to receive the packet. If that is the case the `data_fsm` instructs the FIFO to write the packet to the outgoing port.

In addition to the packet forwarding the `data_fsm` also collects statistical data. It counts the number of packets that arrived at the generic functional block and the number of packets that have been forwarded by the generic functional block. These two counters can be accessed by software via the ReconOS Interface. The difference of these two numbers gives the number of packets that have been dropped inside the generic functional block.

### 3.2.3 Randgen FSM

The `randgen_fsm` gets the seed parameter from the `reconos_fsm` and continuously generates 32 bit pseudo random numbers. For this purpose the `randgen_fsm` implements the pseudorandom number generator depicted in Figure 3.4. The random number is used by the `drop_fsm` to do randomized drop decisions. If the `drop_fsm` doesn't currently produce the drop decision using random numbers the random generator isn't active and doesn't produce random numbers.

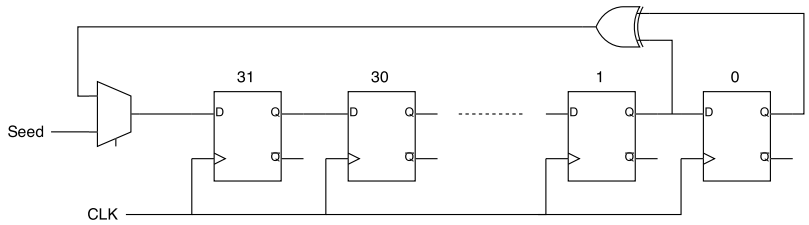



Figure 3.4: Block diagram of the pseudorandom number generator

### 3.2.4 Drop FSM

The the purpose of the `drop_fsm` is to have a new drop decision ready every time an packet arrives at the generic functional block. The drop decision simply tells the `data_fsm` to either drop or forward the packet. There are two parameters for the `drop_fsm`: the `drop_type` and the `drop_value`. There are three operating modes of the `drop_fsm` which can be selected with the `drop_type` parameter:

- For `drop_type=0` the `drop_fsm` is disabled and doesn't drop any packets.
- For `drop_type=1` every `drop_value`-th packet is dropped. This mode doesn't use randomization and therefore doesn't only hold for the mean.

[1] `http://www.xilinx.com/aurora/aurora_member/sp006.pdf` [accessed 20-May-2015]

- For `drop_type=2` every $\frac{2^{32}}{\text{drop\_value}}$-th packet will be dropped on average. In this mode the `randgen_fsm` is used to generate randomized drop decisions. If the random number is larger than the `drop_value` the packet is forwarded otherwise the packet is dropped.

For both cases `drop_type=1` and `drop_type=2` the setting `drop_value=0` is not valid and will disable the drop functionality, i.e. no packets are dropped. To drop every packet the parameters can be set to `drop_type=1` and `drop_value=1`. With this configuration the generic functional block acts as a black hole.

### 3.2.5 Timer FSM

The `timer_fsm` implements the delay. It consists of two counters which count the number of clock cycles the packet needs to be delayed. One counter counts the number of bytes, the other counter counts the clock cycles for each byte of the packet and the packet itself. The two parameters `delay_per_packet` and `delay_per_byte` as well as the packet size in bytes determines the number of clock cycles to delay the packet:

$$\text{total\_delay} = \text{delay\_per\_packet} + \text{packet\_size} \cdot \text{delay\_per\_byte}$$

### 3.2.6 FIFO

The FIFO (First In First Out) queue stores the packet inside the generic functional block. It has a fixed capacity of 1600 bytes. This is enough for any valid packet since the maximum packet length is 1500 bytes. The data width is 8 bits because the LocalLink interface has a data width of 8 bits, too. There are two signals which allows the `data_fsm` to instruct the FIFO to read one byte per clock cycle from the input port or to write one byte per clock cycle to the output port.

## 3.3 Software Design

The software is written in C and makes use of POSIX threads (Pthreads) and POSIX semaphores. The software implements a modular design to allow for arbitrary protocol stacks. Each module is represented by a Pthread. All threads are executed quasi-parallel on a single soft core CPU. There are four types of threads which are all depicted in the exemplary setup in Figure 3.5. All the software is implemented in the user space.

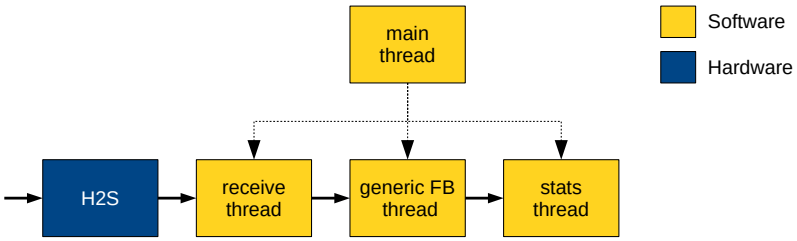



Figure 3.5: Exemplary setup of the software design with a generic functional block

### 3.3.1 Receive Thread

The receive thread is used to fetch packets from the H2S block and to distribute them to the correct software threads. There is exactly one instance of the receive thread. The H2S block stores the packet in the memory. From the H2S block the receive thread gets the address and

the size of the packet via the ReconOS interface. The thread then extracts the destination IDP from the packet. Based on the destination IDP and the configuration the receive thread forwards the packet to the corresponding software thread. This can be a software instance of the generic functional block or a stats thread which collects performance statistics.

### 3.3.2 Generic Functional Block Thread

The generic functional block thread is the software implementation of the generic functional block. It emulates the forwarding behavior according to a user defined set of parameters.

The main difference to the hardware version is the implementation of the processing delay. To correctly emulate a real software functional block the CPU must be kept busy while the packet is delayed. This is implemented using an assembler NOP (no operation) operation which is executed for a certain number of times. The NOP operation has no effect beside that it keeps the CPU busy for one clock cycle. The advantage of the assembler NOP operation is its small granularity and that it is not omitted when the code is optimized by the compiler. I made measurements to get the execution time of one NOP loop iteration on the soft core CPU of the FPGA. In Figure 3.6 the distribution of the execution time for one NOP loop iteration is depicted. The outliers are most likely due to context switches of the CPU. One NOP loop iteration takes about 110 ns which corresponds to 11 clock cycles on the CPU which is running with a clock frequency of 100 MHz (This means 1 clock cycle for the NOP and 10 clock cycles overhead for one loop iteration). This value is used to convert the requested delay into a number of NOP loop iterations.

Another small difference to the hardware implementation is the random number generator. To generate random numbers in the software generic functional block the standard C library function `rand()` is used. This random generator is seeded with the system time when the program is executed by using the function `srand()`. Therefore in software it isn't possible to initialize each instance of the generic functional block with a separate seed.

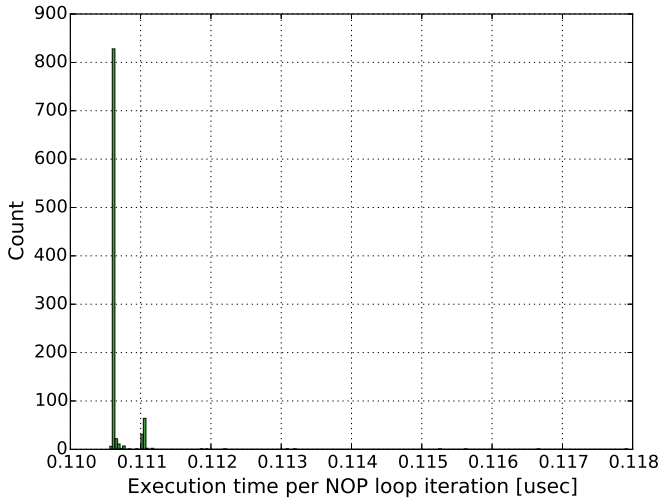



Figure 3.6: Distribution of the execution time of one NOP loop iteration (98.6% of 1000 measured samples are displayed)

### 3.3.3 Stats Thread

The stats thread is used to count and optionally display all arriving packets. There is a separate counter for every IDP. The main thread reads and stores these counter values at specific time instants to generate statistics. Once a packet has been registered in the stats thread it is discarded.

### 3.3.4 Main Thread

The main thread handles all program arguments and reads the protocol stack configuration from a file. The program arguments are used to select what to log or to display. The configuration file contains the parameters for the generic functional blocks as well as information of how the blocks in software and hardware are interconnected (for more information about the configuration file see Appendix A.2.4). With these information the main thread initializes all hardware and software threads.

Furthermore the main thread collects samples of measurements and prints statistics. For this it stores timestamps and the counter values from different software threads.

### 3.3.5 Interconnection of the Threads

All instances of the threads except for the instances of the receive and main threads have their associated shared memory as well as two semaphores which protect the shared memory. The shared memory consists of a pointer to the packet data in the memory, the packet length and the packet origin. A scheme of this design is depicted in Figure 3.7a.

In the following I explain the data transfer mechanism between two threads based on the example of the interconnection of the generic FB thread and the stats thread for the scenario depicted in Figure 3.7b. The packet signal semaphore is initialized with the value 0, the done signal semaphore is initialized with the value 1. The done signal semaphore indicates that the content of the shared memory is no longer used by the stats thread and therefore can safely be overwritten with a new packet. The generic FB thread which wants to send a packet decrements the done signal semaphore of the stats thread. This makes sure no other thread writes to the shared memory at the same time. The generic FB thread can then safely write its packet and the associate information into the shared memory. After that the generic FB thread increments the packet signal semaphore of the stats thread. This tells the stats thread that there is new data in the shared memory which can be fetched. The stats thread then processes the packet and then first decrements the packet signal semaphore and afterwards increments the the done signal semaphore. This allows the next thread to send a packet to the stats thread. With this concept it is also possible that multiple threads forward the packets to a single thread.

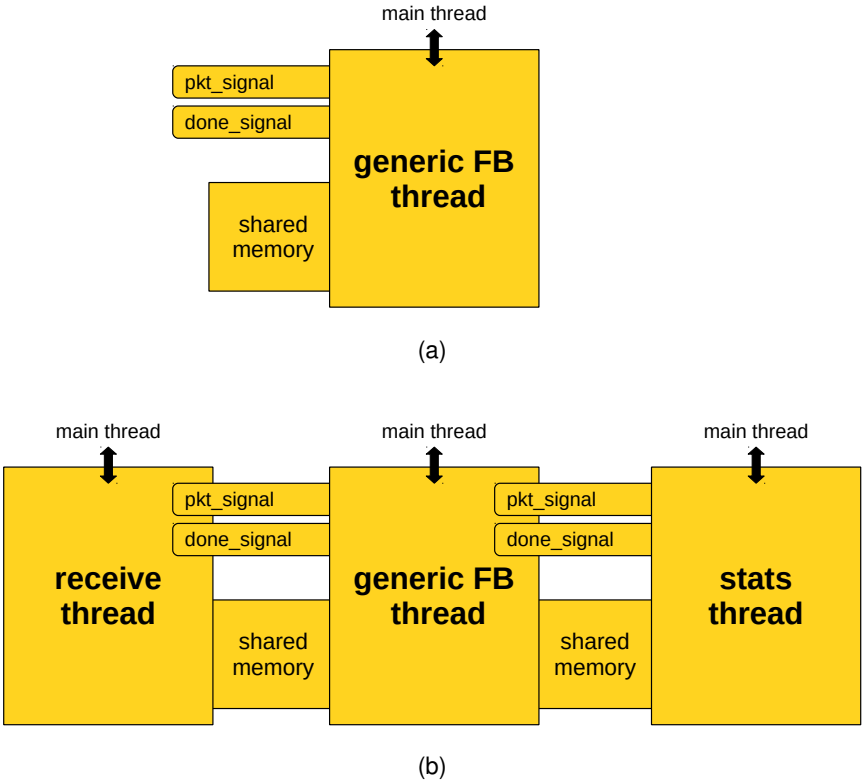



Figure 3.7: Design of a single thread (a) and a setup of interconnected threads (b)

# Chapter 4

# Validation

In this chapter I mainly focus on the validation of the hardware implementation of the generic functional block.

During the development I validated the hardware implementation of the generic functional block by simulating the behavior. For the simulation I use the Xilinx ISE Simulator (ISim)[1]. In Figure 4.1 the simulation of the processing of one packet is depicted. First the packet is read into the FIFO therefore the filling level (`fifo8_s_fill` signal) of the FIFO is incremented. Then the drop decision is used (not shown in the figure). In this case the packet isn't dropped. Afterwards the packet is delayed. First the packet is delayed according to the `delay_per_byte=2`. The packet size in this example is 6 bytes. Therefore the delay is 12 clock cycles. Then the packet is delayed according to the `delay_per_packet=4` which takes 4 clock cycles in this example. In the end the packet is written to the output port.

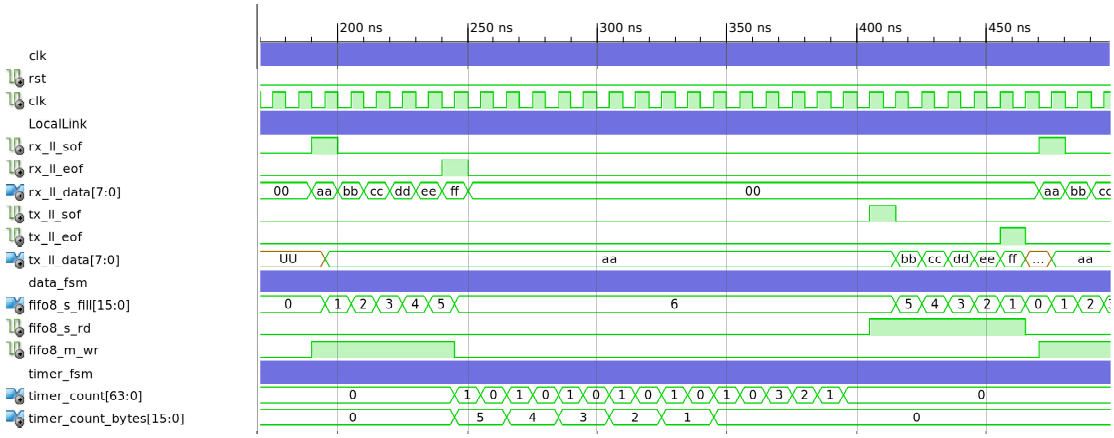



Figure 4.1: Simulation of the hardware generic functional block (ISE Simulator)

Once everything worked in the simulation I generated the bitstream (configuration for the FPGA) and tested the implementation on the FPGA. To validate and debug the implementation while running on the FPGA there is the Xilinx tool named ChipScope[2]. To use this tool it is necessary to add an additional test circuitry to the VHDL code. An exemplary measurement of the implementation running on the FPGA is shown in Figure 4.2. Again the processing of one packet is depicted. In this case the packet size is 38 bytes. The parameters are unchanged (`delay_per_packet=4` and `delay_per_byte=2`).

I tested the hardware as well as the software generic functional block in various scenarios. I validated the behavior with different packet sizes and with different combinations of parameters. For the delay parameters I used combinations of the three basic cases: zero, one or more than one cycle (e.g. eight cycles). For the drop functionality I tested the three cases: drop no packets,

[1] `http://www.xilinx.com/tools/isim.htm` [accessed 20-May-2015]
[2] `http://www.xilinx.com/tools/cspro.htm` [accessed 20-May-2015]

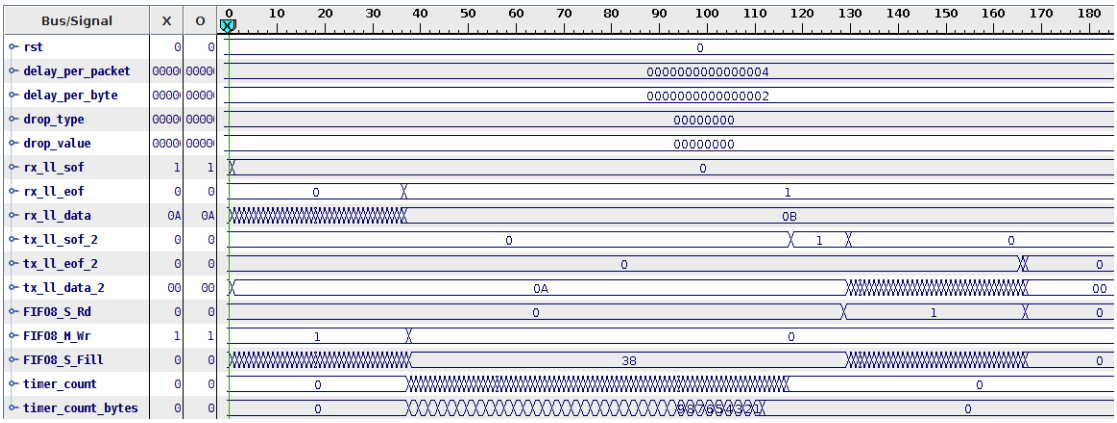



Figure 4.2: Hardware debugging of the hardware generic functional block (Screenshot of the ChipScope Software)

drop every packet and drop a fraction of all packets (e.g. a third). In addition I tested all drop types to make sure that they work as expected.

# Chapter 5

# Evaluation

## 5.1 Experimental Setup

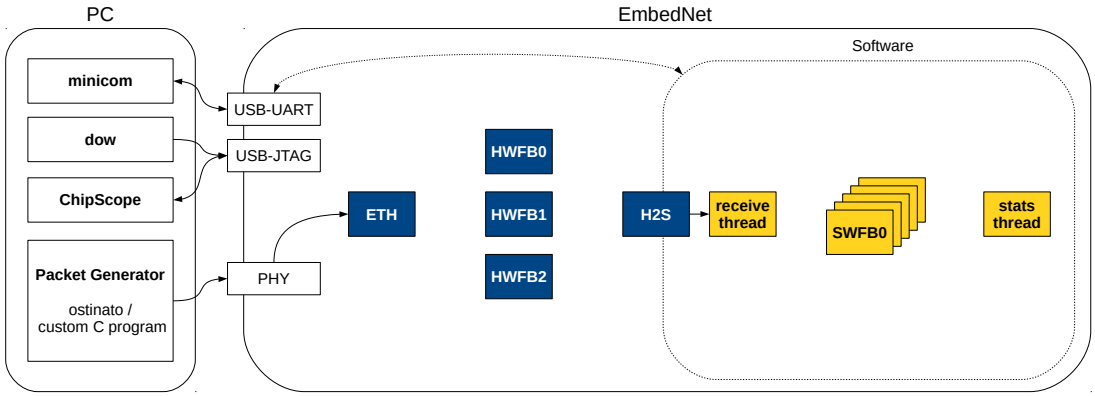



Figure 5.1: Experimental Setup

The experimental setup is depicted in Figure 5.1. For the measurements I use the Xilinx Virtex-6 FPGA ML605 evaluation board. On the FPGA the MicroBlaze soft core processor which is clocked at 100 MHz and runs an embedded Linux version 2.6.37 is used. In addition I use a Desktop PC that runs Ubuntu 14.10 with Linux version 3.16.0.

The evaluation board is connected by multiple ports to the PC. The USB-UART port is used setup a remote serial console on the Desktop PC to access the Linux which runs on the FPGA. For this the terminal emulation program *minicom* is used. To download the bitstream (FPGA configuration) and the kernel image to the FPGA the USB-JTAG port is used. For this I use a script called *dow* which calls the Xilinx tools *iMPACT* and *xmd* (Xilinx Microprocessor Debugger). ChipScope uses the same USB-JTAG port for the hardware debugging.

To send test network traffic to the EmbedNet the physical interface is used. All packets originate from the PC and end up in the software on the FPGA (except the packets that are dropped along the way). To generate the packets for the measurements I use the *ostinato* software[1] as well as a customized C program to generate packets with the required properties such as size, headers and packets per second.

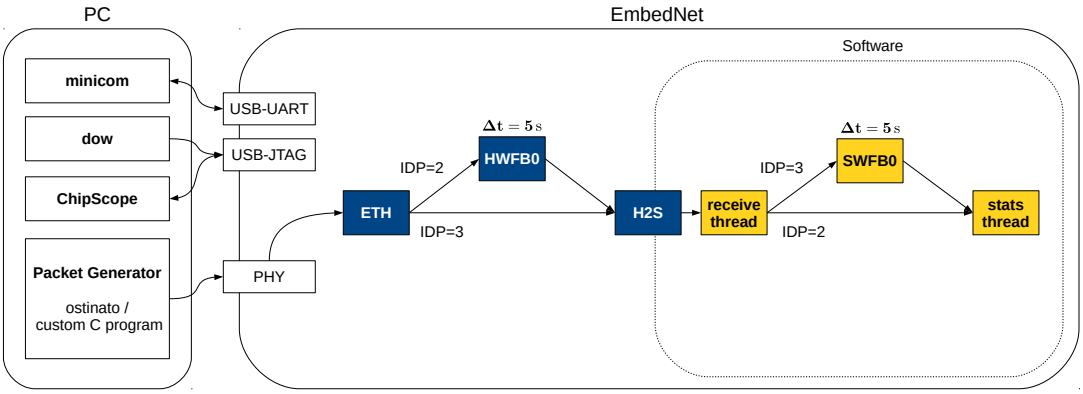
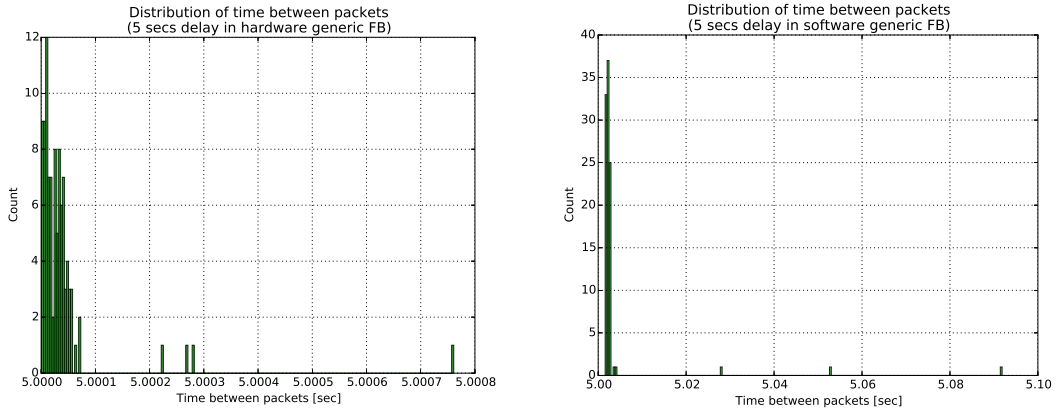



Figure 5.2: Setup to measure the accuracy of the delay in the generic functional block




(a) (b)

Figure 5.3: Distribution of the delay of the generic functional blocks in hardware (a) and software (b)

## 5.2 Accuracy of the Generic Functional Block

In order to measure the accuracy of the delay parameter of the generic functional block I used the setup with two protocol stacks (IDP 2 and IDP 3) depicted in Figure 5.2. The delay parameters of the hardware and software generic functional block are set to 5 seconds. The drop functionality is not used. With the packet generator I flooded the EmbedNet node with a packet rate larger than 1 packet every 5 seconds. In the stats thread I measured the delay between consecutive packets that passed through one of the generic functional blocks. I conducted two separate measurements, one with the hardware generic block only (IDP 2) and another one with the software generic block only (IDP 3). The distribution of the delay in the hardware generic functional block is depicted in Figure 5.3a whereas the distribution of the delay in the software generic functional block is depicted in Figure 5.3b.

The two figures show that the accuracy of the hardware implementation is about 100 times better than the accuracy of the software implementation. I actually expected a result like this since the software execution shares the CPU with other system tasks and therefore includes many uncertainties like context switches. The hardware implementation on the other hand runs independently of the CPU and is deterministic. The measurements show that both versions work as expected.

[1] `https://code.google.com/p/ostinato/` [accessed 20-May-2015]

## 5.3 Maximum Packet Rates

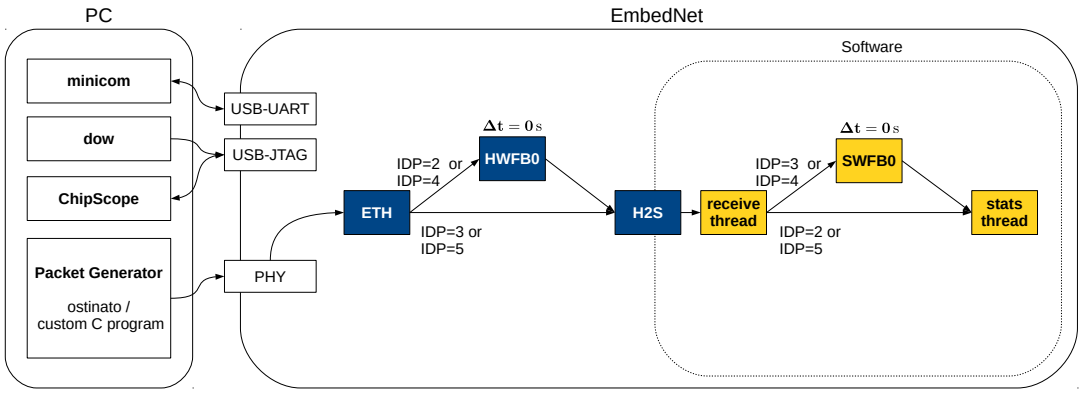



Figure 5.4: Setup to measure the maximum packet rates for four different protocol stacks

The goal of the following measurements is to measure the maximum packet rates with four different protocol stacks including up to one hardware generic functional block and/or up to one software functional block. The setup us depicted in Figure 5.4. The protocol stacks are identified by their IDP. All delays of the generic functional blocks are set to 0 and the drop functionality is not used. Each protocol stack is measured by its own (i.e. there where no packets with different IDPs in the system at the same time). Again the EmbedNet is flooded with more packets than it can handle. The packet rates (packets per seconds (pps)) measured in the stats thread are listed in Table 5.1, 5.2 and 5.3 for the three packet sizes 64 bytes, 500 bytes and 1500 bytes, respectively.

Table 5.1: Maximum packet rates with packets of size 64 bytes

| | HW FB (IDP 2) | SW FB (IDP 3) | HW + SW FB (IDP 4) | no FB (IDP 5) |
|---|---|---|---|---|
| pps | 996 | 638 | 627 | 994 |
| kbits/sec | 510 | 327 | 321 | 509 |

Table 5.2: Maximum packet rates with packets of size 500 bytes

| | HW FB (IDP 2) | SW FB (IDP 3) | HW + SW FB (IDP 4) | no FB (IDP 5) |
|---|---|---|---|---|
| pps | 993 | 634 | 638 | 992 |
| kbits/sec | 3’972 | 2’536 | 2’552 | 3’968 |

Table 5.3: Maximum packet rates with packets of size 1500 bytes

| | HW FB (IDP 2) | SW FB (IDP 3) | HW + SW FB (IDP 4) | no FB (IDP 5) |
|---|---|---|---|---|
| pps | 991 | 639 | 639 | 992 |
| kbits/sec | 11’892 | 7’668 | 7’668 | 11’904 |

The results show that the size of the packet has no influence on the maximum packet rates. However the software generic functional block has a quite significant influence. It reduces the maximum packet rate by 36% from about 990 pps to about 630 pps. That’s remarkable since the generic functional block is configured with zero delay. Possible reasons for this reduction are additional context switches and other overhead (e.g. synchronization with semaphores, conditional statements) caused by the software generic functional block.

I also measured the packet rates in the hardware generic functional block with disabling the software. This is interesting because the software generic functional block in the protocol stacks also reduces the packet rate in the hardware part. The reason for this is that the hardware version of the generic functional block only sends the packets if the downstream software components are ready to handle the packet. To disable the software I redirected all packets from the hardware generic functional block to the S2H block in the EmbedNet. The S2H block acts as a black hole for traffic arriving from the NoC. With this setup the hardware generic functional block is not slowed down by the software. The maximum packet rates of these measurements

for different packet sizes are listed in Table 5.4. I need to add that the packet rate for packets of the size 64 bytes wasn't limited by the generic functional block but by the packet generator on the PC.

Table 5.4: Maximum packet rates measured in the hardware generic functional block

| packet size | 64 bytes | 500 bytes | 1'500 bytes |
|---|---|---|---|
| pps | 429'240[a] | 104'219 | 33'793 |
| kbits/sec | 219'771 | 416'867 | 405'516 |

[a] maximum packet rate limited by packet generator

The results show that the H2S block and/or the software significantly reduce the packet rates and therefore represent a bottleneck.

## 5.4 Two Competing Protocol Stacks

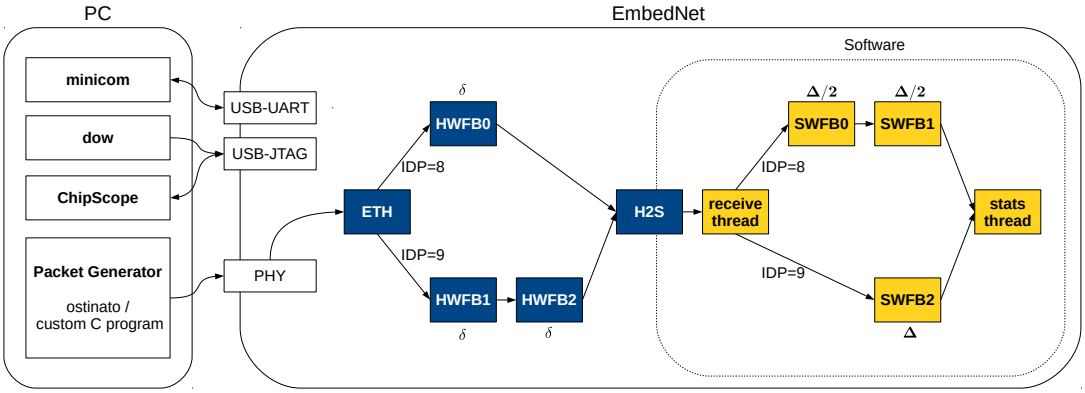



Figure 5.5: Setup with two competing protocol stacks

In the following series of measurements I investigate the behavior of the EmbedNet if two parallel protocol stacks are used at the same time. For this scenario I use the setup which is depicted in Figure 5.5. The protocol stack corresponding to IDP 8 consists of one hardware generic functional block and two software generic functional blocks whereas the protocol stack with IDP 9 contains two hardware generic functional blocks and one software generic functional block. All the hardware generic functional blocks are configured to delay every packet by $\delta = 10$ clock cycles in all measurements. The software functional blocks of the stack with IDP 8 are configured to emulate a processing time of $\Delta/2$ for every packet whereas the single software generic functional block of the stack with IDP 9 emulates a processing time of $\Delta$ for every packet. The $\Delta$ is varied for the different measurements. The dropping functionality is not used except for the last measurement. For all measurements only packets with a size of 64 bytes are used. Therefore there is no benefit in using the `delay_per_byte` parameter.

The setup implies that both protocol stacks need the same processing time on the CPU for one packet. If both protocol stacks get the same amount of processing time on the CPU they both should have the same throughput. Because the protocol stack with IDP 8 consists of 2 software threads and the protocol stack with IDP 9 only has 1 software thread it could be possible however that the protocol stack with IDP 8 gets $2/3$ of the processing time and the protocol stack with IDP 9 gets only $1/3$ of the processing time on the CPU.

### 5.4.1 No congestion

For the first measurement I configured the packet rates and the $\Delta$ such that there shouldn't be a congestion neither at the H2S block nor at the generic functional blocks. The input with a upper limit of 400 pps per protocol stack is depicted in Figure 5.6. The $\Delta$ is set to 10'000 clock cycles which correspond to 0.0001 seconds and a maximum packet rate of 10'000 pps. However the maximum packet rate which can be expected is around 630 pps as seen in the previous measurements (section 5.3). The measured result is depicted in Figure 5.7.

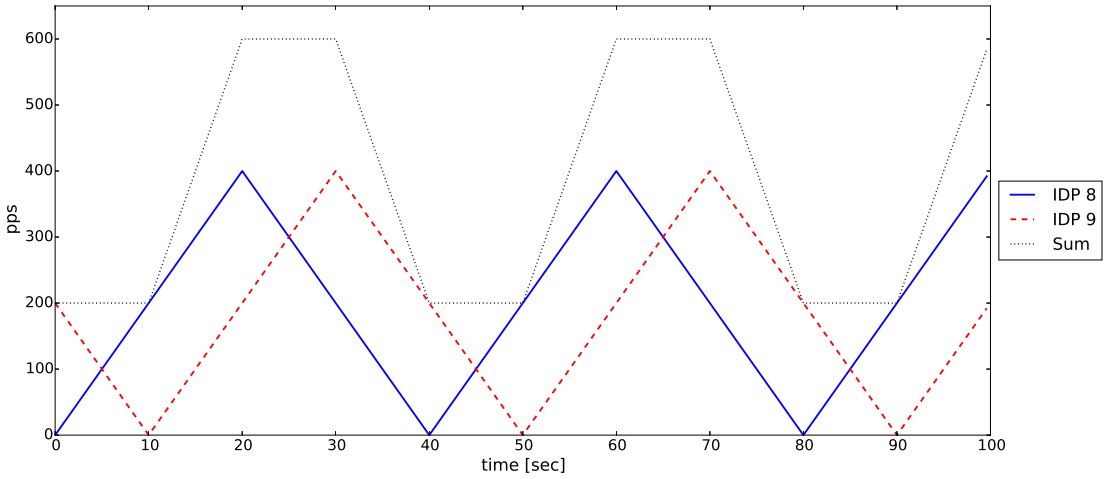



Figure 5.6: Generated packet rates with an upper limit of 400 pps per protocol stack (ideal values)

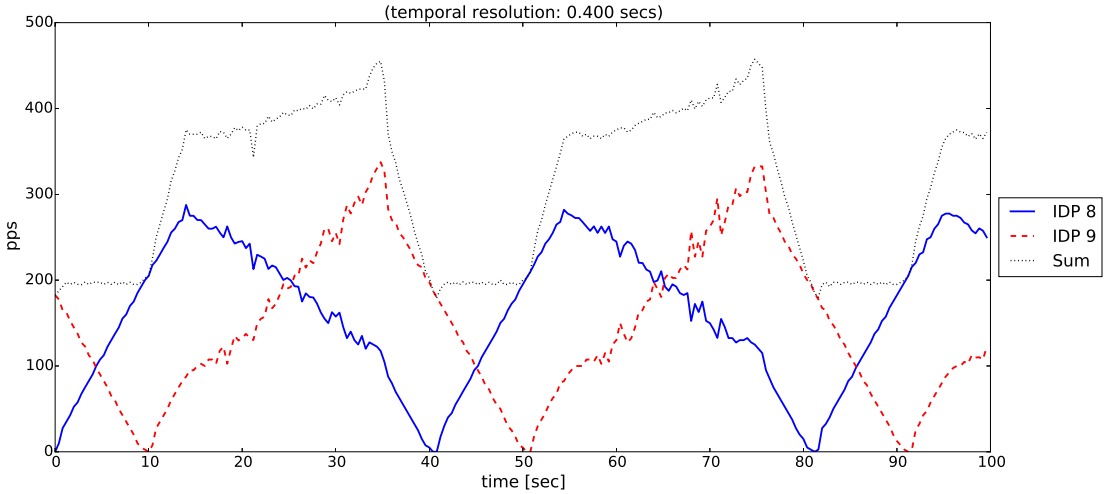



Figure 5.7: Measured packet rates in the stats thread for the setup with no congestion

The results show that for a total packet rate below about 370 pps the EmbedNet behaves almost ideally. However the EmbedNet seems not to be able to process packet rates above 370 pps in an ideal manner with this setup. That's most likely because there are conflicts between the two protocol stacks in the software. Interestingly once the limit of about 370 pps is hit the total packet rate continues to grow to about 440 pps. With input packet rates above 370 pps the distribution of resources is no longer proportional to the input packet rate but is fair in a way because the system tries to allocate to both protocol stacks the same amount of processing time. This means that the packet rates of both protocol stacks tend to get to the same level (this can be seen better in Section 5.4.4). This equilibrium level however doesn't persist because both input packet rates are changing heavily and move into different directions.

### 5.4.2 Congestion at the H2S Block

For the second measurement I configured the packet rates such that there is a congestion before the H2S block. The generated packet rates with a limit of 2'000 pps per protocol stack are depicted in Figure 5.8. The $\Delta$ is left unchanged at 10'000 clock cycles. The resulting output packet rates measured in the stats thread are depicted in Figure 5.9.

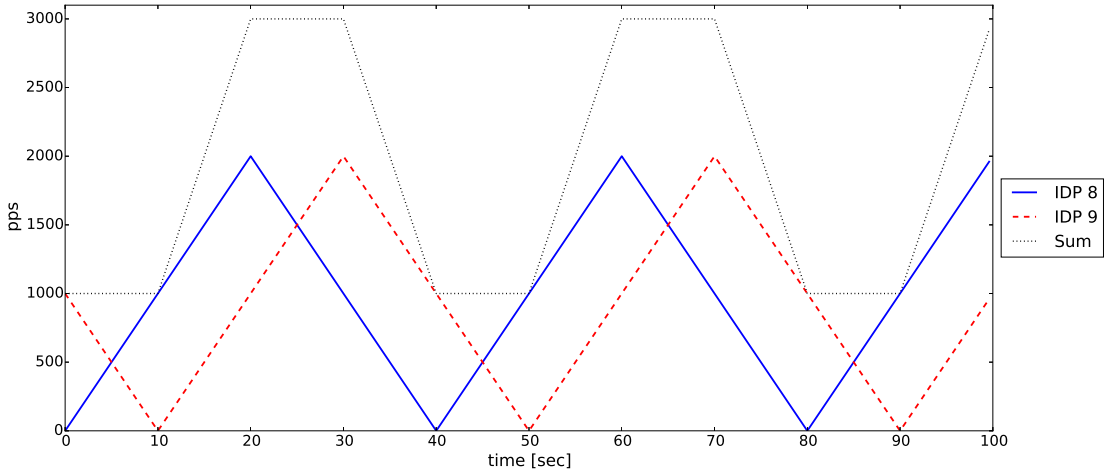



Figure 5.8: Generated packet rates with an upper limit of 2'000 pps per protocol stack (ideal values)

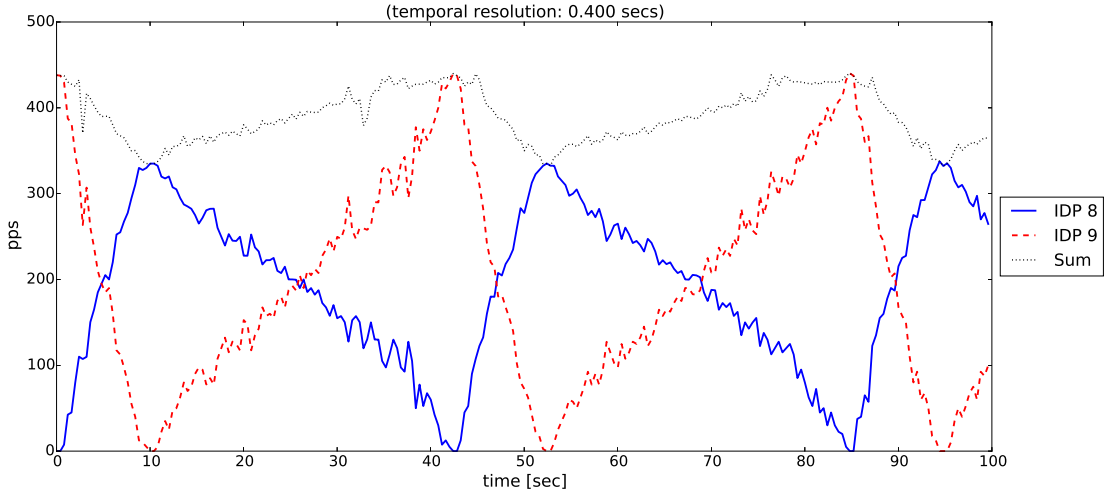



Figure 5.9: Measured packet rates in the stats thread for the setup with congestion at the H2S

The results of these measurements contain much more variations compared to the measurements without congestion in Figure 5.7. An amount of packets that are too much at a certain time instant are delayed and processed later. In addition the resources are no longer distributed in a fair way. The peak of the protocol stack with IDP 9 are significantly higher than the peaks of the protocol stack with IDP 8 even though only one protocol stack is active at the time of the peaks. A possible reason for that could be that at the H2S block not the same amount of packets from both protocol stacks are lost (due to the congestion) because the input packet rates of both stacks are not symmetrical to each other.

### 5.4.3 Congestion at the Software Generic Functional Block

For this measurement I configured the software generic functional blocks such that they cannot process all packets that arrive. This means the software generic functional blocks artificially

produces a congestion. The $\Delta$ is set to 2'000'000 clock cycles which equals 0.02 secs and allows a maximum packet rate of 50 pps if only a single protocol stack is using the CPU. For the packet generation I used the packet rates depicted in Figure 5.10. It's the same packet generation pattern I used in the first measurement (Figure 5.6). The results of the measurement in the stats thread is depicted in Figure 5.11.

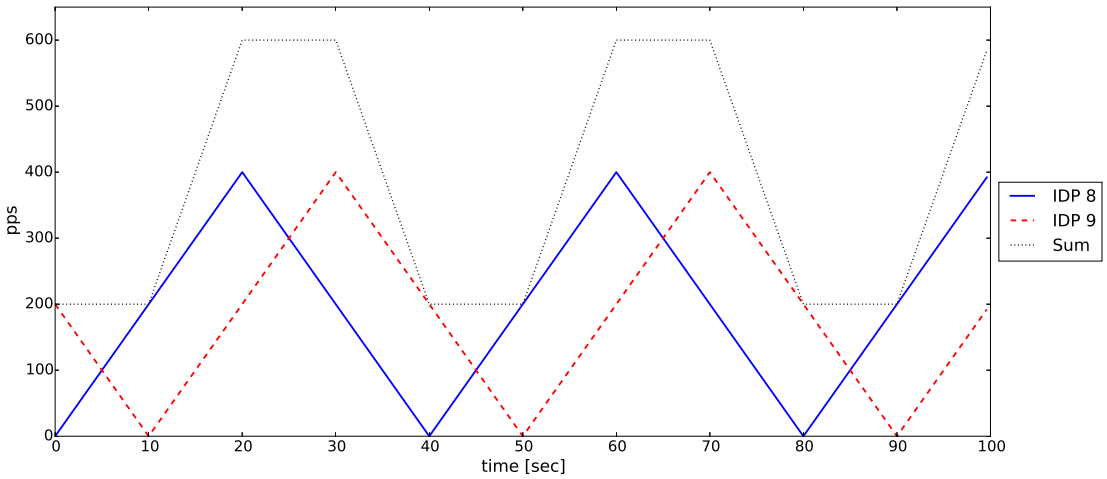



Figure 5.10: Generated packet rates with an upper limit of 400 pps per protocol stack (ideal values)

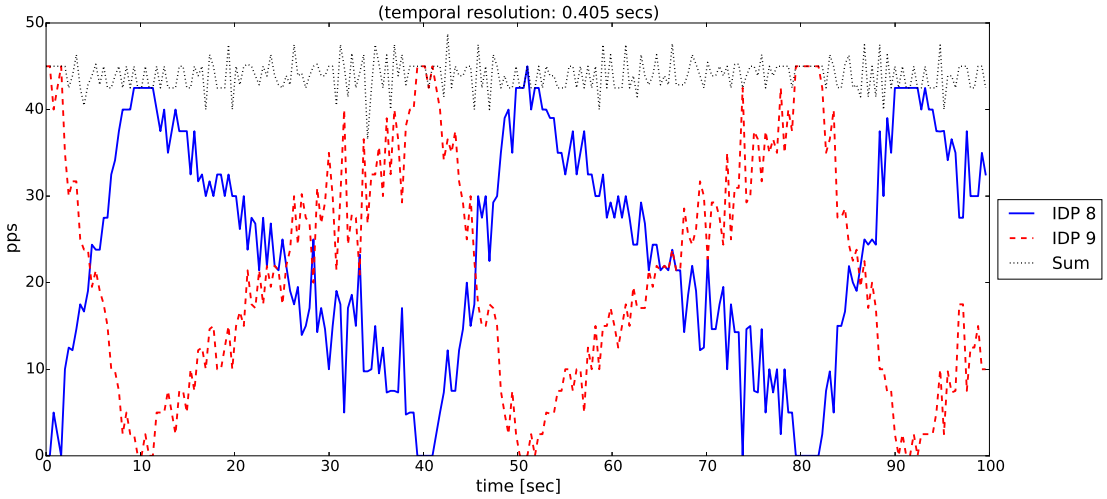



Figure 5.11: Measured packet rates in the stats thread for the setup with congestion at the software generic functional blocks

Again the results show a lot of variations which are most likely due to the congestion. The total packet rate meets the expectation of 50 pps quite well. Also the two protocol stacks seem to get a fair share of the resources since both stacks exhibit symmetrical packet rate curves to each other.

### 5.4.4 Different Packet Generation Intervals

To investigate a more varying time behavior for the two protocol stacks I did a measurement with two different intervals between the peaks of the packet generation output. The interval for the protocol stack with IDP 8 is still set to 40 seconds but the interval for the protocol stack with IDP 9 is set to 60 seconds. The generated packet rates are shown in Figure 5.12. The $\Delta$ is set to 10'000 clock cycles. The result of the measurements in the stats thread is depicted in

Figure 5.13.

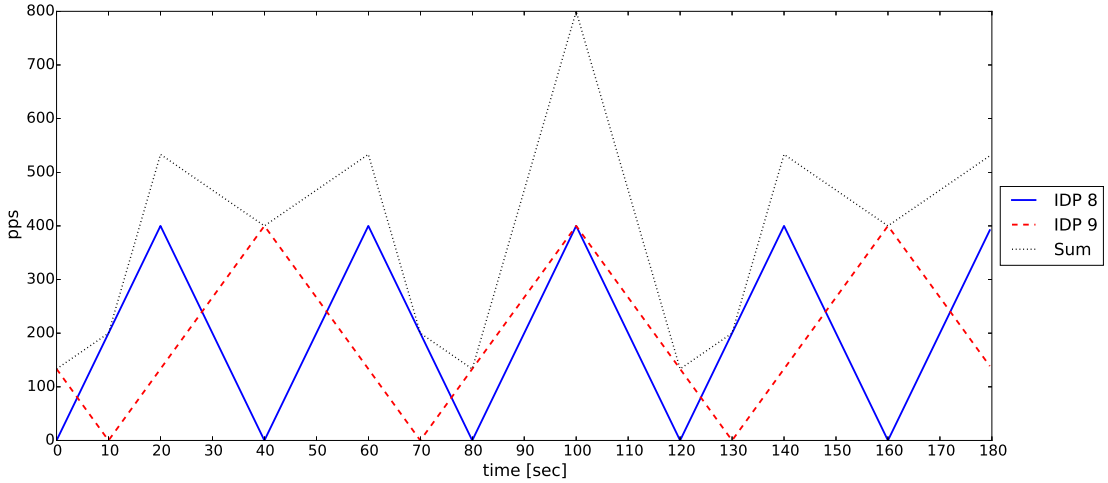



Figure 5.12: Generated packet rates with different intervals between the peaks (ideal values)

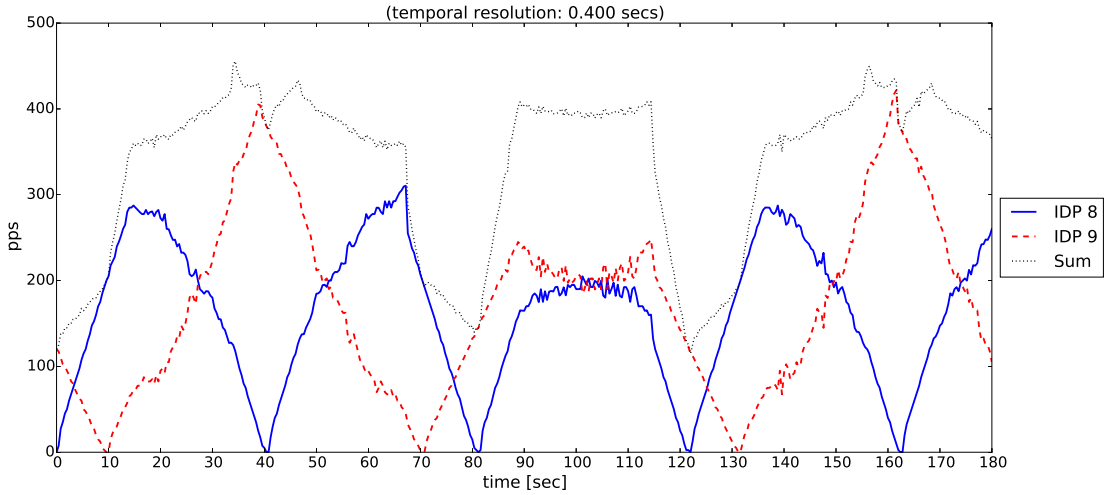



Figure 5.13: Measured packet rates in the stats thread for the setup with different packet generation intervals

In this measurement again the packet processing limit of about 370 pps is identifiable. Above that limit at around 90 seconds (in Figure 5.13) the packet rate of the protocol stack with IDP 9 suddenly decreases whereas the packet rate of the protocol stack with IDP 8 still increases even though the input packet rates of both protocol stack increases. It seems that the packet rate of both protocol stacks tend to the same equilibrium level. This is in a way fair since both protocol stacks get the same amount of processing time on the CPU. However packet rates that are proportional to the input packet rates (which is not the case here) could be considered better in terms of fairness.

### 5.4.5 Drop Rate

In this last measurement I test the drop functionality of the generic functional block. For this I configured the software generic functional block SWFB1 in the stack with IDP 8 to drop every second packet. All other generic functional blocks are configured not to drop any packets. For the packet generation the packet rates depicted in Figure 5.14 have been used. It's again the same packet generation pattern used in the first measurement (Figure 5.6). The result is depicted in Figure 5.15.

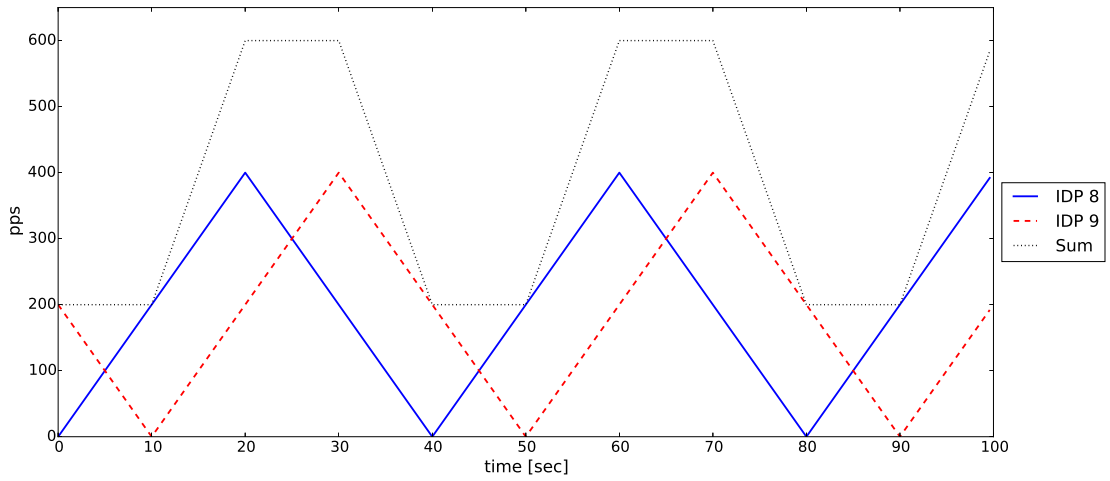



Figure 5.14: Generated packet rates with an upper limit of 400 pps per protocol stack (ideal values)

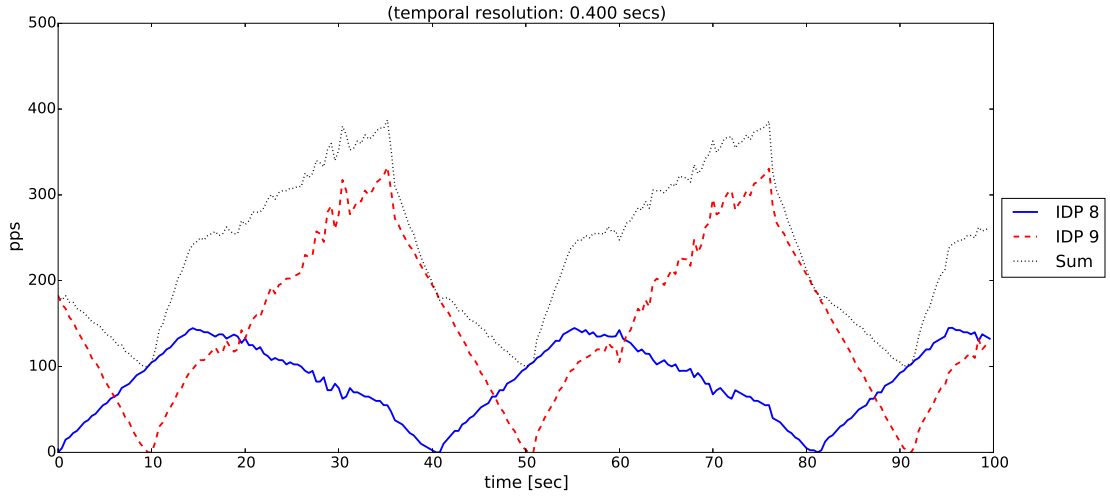



Figure 5.15: Measured packet rates in the stats thread for the setup using the drop functionality

The result shows that the drop functionality works as expected. The packet rate of the protocol stack with IDP 8 is exactly halved compared to the same measurement without using the drop functionality in Figure 5.7. The unusual low limit of about 250 pps of the total packet rate at which the EmbedNet doesn't process the packets ideally anymore may seem a bit odd at first glance. But it can be explained by the processing overhead caused by the packets that are dropped and therefore aren't counted in the stats thread.

## 5.5 Shortcomings of the EmbedNet

During the development of the generic functional block and the evaluation of the EmbedNet platform I noticed several shortcomings.

First of all there is the high implementation effort which is necessary to implement a new function block. This problem is however not specific to the EmbedNet platform. All platforms for hardware acceleration implementations suffer from it. There are new approaches which can be used to implement hardware using high-level development tools. But as of today none of them can generate efficient implementations without special hints from the person who programs the hardware implementation.

Another problem which is more specific to the EmbedNet is an "Illegal Instruction" error I encountered during the development of the software for the EmbedNet. The error only occurs in approximately 50% of all executions of the program I implemented. It seems to be connected to the initialization of the hardware. For this the ReconOS library is used. Until the end of the thesis I couldn't resolve the issue because I don't have deeper knowledge of the internal structure of that library. Obviously it must have something to do with a kind of race condition since the software works in 50% of the executions.

A shortcoming concerning the performance of the EmbedNet is the bottleneck at the H2S block. This problem is quite severe since it affects all protocol stacks and all implementations of functional blocks. Therefore there is another thesis which treats exactly this problem.

Another limitation is the low clock frequency of only 100 MHz of the CPU on the EmbedNet platform. I think a faster CPU could significantly improve the performance of the functional blocks in software. As mentioned in the paper of Keller et. al. [5] this problem maybe will be addressed in the future. They suggest to use a new FPGA evaluation board with more recent hardware.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

In this thesis I designed and implemented a generic functional block for the EmbedNet. The implementation consists of a version in hardware as well as a version in software. The generic functional block is able to emulate the forwarding behavior of a real functional block in terms of the delay and the drop rate. This allowed me to perform the first performance evaluation of the EmbedNet with up to six functional blocks.

The evaluation with two parallel protocol stacks showed that the EmbedNet behaves ideally for a packet rate below a certain limit. In my scenario this limit was at a total packet rate of 370 pps. With a total input packet rate above that limit the processing is not ideal anymore. Most likely this is because of conflicts in the software between the different protocol stacks. Above the limit the packet rates are no longer proportional to the incoming packet rates but are fair in a sense since all protocol stacks have the same throughput. This indicates that all protocol stacks get the same share of processing time.

The implementation of the generic functional block developed in this thesis can be the basis for further performance evaluations of the EmbedNet since it allows to easily build and emulate complex protocol stacks and to collect measurement data. In addition the generic functional block can be useful for debugging purposes.

The EmbedNet is a quite advanced platform to test the promising idea of a dynamic protocol stack architecture. It has many features that are necessary to implement a self-aware and self-adaptive network node which can operate more efficiently than today's network nodes. Nevertheless it also has a few shortcomings. The most important ones are the bottleneck at the H2S block and the low computing power of the soft core processor.

## 6.2 Future Work

My implementation of the generic functional block has proved to work quite well during the evaluation. However there is still a lot of room to improve it. Also the evaluation of the EmbedNet is by no means complete.

One possible improvement of the generic functional block is a more detailed emulation of real functional blocks. For example a Gaussian distribution could be used to vary the actually applied values around the values which are set by the parameters. This would emulate an unpredictable behavior more accurately. Another enhancement is the choice of different relations between the packet size and the delay. At the moment the relation is purely linear. Maybe an exponential or logarithmic relation would be useful to emulate certain real functional blocks. A further improvement is the possibility to store more than one packet inside the generic functional block. This

could be useful to emulate a functional block which implements an algorithm which needs access to more than one packet at the same time. However the effects of the described additional functionalities might be limited because a bottleneck at another place in the system could cover some details of the emulation. For example the increase of the number of packets which can be buffered in the generic functional block would in most situations not change much because the packets are buffered in other blocks anyway. But especially with low packet rates and the emulation in software the additional details would be useful.

Another aspect of the implementation of the generic functional block which could be improved is the user-friendliness. The newly introduced configuration file certainly helps to configure the generic functional block and the protocol stacks more easily. But there is no complete input validation. In addition the code contains constructs to allow for more configuration possibilities. However the additional configuration possibilities weren't used in this thesis. Therefore they haven't been tested extensively.

Also the evaluation of the EmbedNet platform could be extended. There are a lot of additional scenarios that could be useful to evaluate. For example the fairness of multiple threads on the CPU could be investigated more deeply. There is a number of features in the EmbedNet platform that can be used to influence the fairness behavior. For example it is possible to assign priorities to the software threads as well as to the hardware functional blocks. Another feature is the option to mark packets as "latency critical". In this thesis none of these features have been used. Fortunately it is quite easy to obtain further measurement results with the provided implementation and its configuration file.

Another aspect which has been neglected for simplicity in the evaluation in this thesis is to send packets from the EmbedNet to another network node. Although it is likely that the behavior is quite similar to the behavior when packets are received this could reveal undiscovered problems in the EmbedNet.

# Appendix A

# HowTo

This appendix explains how to setup and use the generic functional block with the EmbedNet.

## A.1 EmbedNet

### A.1.1 System Design

The system design with three slots for functional blocks is depicted in Figure A.1. This information is needed to set the options in the configuration file of the software.

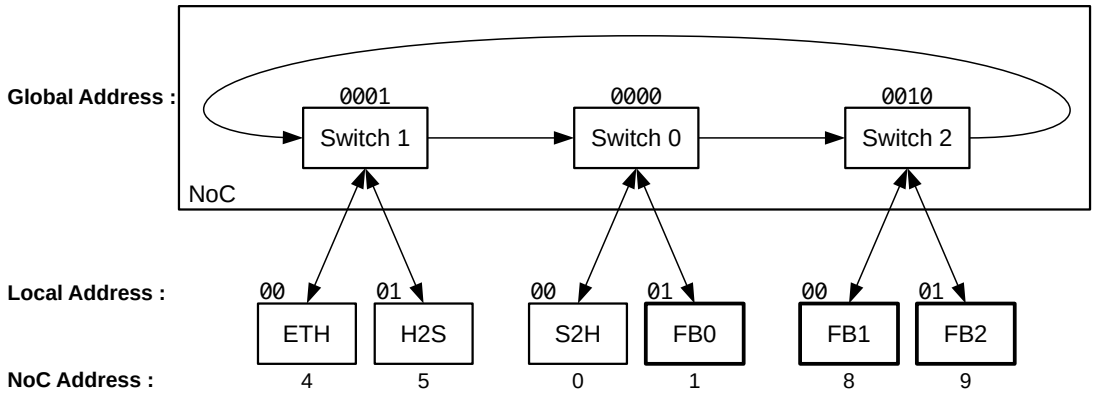



Figure A.1: System design with three FB slots

The NoC address is formed by concatenating the 4 bit global address and the 2 bit local address. E.g. for the H2S block: `0001` and `01` = `000101` = 5.

### A.1.2 How to start the EmbedNet

1. Switch on power of FPGA board
2. Wait until network interface LEDs light up
3. Download the bitstream, e.g.
   `dow system.bit`
4. Download the kernel image, e.g.
   `dow simpleImage.generic`

5. Connect to the FPGA using minicom, e.g.
   `sudo minicom -D /dev/ttyUSB0`
6. Connect the FSLs (Fast Simplex Links) and set up the memory, e.g.
   `./load_fsl.sh`
   `./load_getpgd.sh`
7. Start the software (which sets up the protocol stacks and starts the measurements), e.g.
   `./generic_fb -f<config file> <other arguments>`

## A.2 Software

The program `generic_fb` includes everything that runs in software. All the initialization (hardware and software) as well as the creation of the threads (main, receive, swfb, stats) and also the measurement is done with this program. The setup which is used for the initialization can be specified in a separate configuration file which is read by the program. With the program options it is possible to select what is displayed or measured.

### A.2.1 Compilation

The software can be compiled for the MicroBlaze soft core on the FPGA as well as for a desktop PC. On the desktop PC however there is only a subset of the functionality available because the necessary hardware is not available. Nevertheless this is useful to debug the software more efficiently.

`make dt` Compile for Desktop Ubuntu (FPGA / ReconOS specific parts will be omitted)

`make mb` Compile for MicroBlaze (FPGA soft core)

### A.2.2 Structure of the Software

generic_fb.c/.h contains the main part of the implementation, i.e. it contains all the threads (main, receive, swfb, stats), the hardware & software initialization, the part to perform measurements

args_from_file.c/.h contains all parts necessary to read in the configuration from the configuration file

timing.c/.h taken over without modification from the *app* example software

### A.2.3 Program Options

`-h <secs>` Measures the packet rate in the HW generic functional block (WITHOUT any software threads running (except the main thread)). The packet rate is measured over an interval of `<secs>` seconds.
NOTE: The IDP cannot be selected, all packets that pass through the HW FB are counted.
[double]

`-w <secs>` Measures the packet rate in the HW generic FB (WITH the software threads running). The packet rate is measured over an interval of `<secs>` seconds.
NOTE: The IDP cannot be selected, all packets that pass through the HW FB are counted.
[double]

`-r <secs>` Measures the packet rate in the stats_thread. The packet rate is measured over an interval of `<secs>` seconds.
NOTE: The IDP must be selected with the `-i` argument.
[double]

`-p <secs>` Measures the packet rate in stats_threads over time. Logs packet count and timestamp for 2 predefined IDPs (no. 8 & 9) into an array and prints it after the measurement of all samples is done (in order not to influence the measurement with `printf()`)
`<secs>` is the interval between two samples. With `-n` the number of samples is selected.
Output format: Every line consists of:
`<timestamp in usecs>, <packet count idp=8>, <packet count idp=9>`
[double]

`-t` Measures the time between 2 consecutive packets.
NOTE: The IDP must be selected with the `-i` argument.
[bool]

`-i <num>` Selects the IDP for the measurement (works with `-r`, `-t`)
[integer]

`-n <num>` Number of repetitions of a measurement (works with `-h`, `-w`, `-r`, `-p` or `-t`)
[integer]

`-c` Continuously (every second) prints the counter values for all IDPs. The values are counted in the stats_thread.
[bool]

`-d` Continuously (every second) prints the counter values of all HW generic FBs.
[bool]

`-v` Verbose (print debug infos)
[bool]

`-l` Logging mode (don't print anything except the measurements)
[bool]

`-f <file>` Filename of the file that should be used as config file. Without this option the default filename "generic_fb.config" is used.
[string]

### A.2.4 Configuration File

Every valid line has the following structure:
`<command> <option> <value0> [<value1>] [<value2>]`
All other lines are ignored (including the lines starting with a #-character).

The lines are grouped into blocks. Each blocks starts with the 'config' command which is followed by zero or more 'option' command lines which configure the entity described in the previous 'config' line. The structure of the line that starts a new entity is as follows:
`config <entity name> <entity ID>`

Note All IDs of the entities (e.g. config HWFB 0) must start from 0 and must not skip one number! The order of the blocks with respect to their IDs doesn't matter. Also the order of the options within a block doesn't matter.

Note: The behavior if one of the options below is not explicitly specified is not tested and therefore the software might not work as expected in such cases.

**Hardware generic FB (Entity name: `HWFB`)**

Example:

```
config HWFB 0
  option delay_per_packet_secs  5
  option delay_per_byte_cycles  0
  option drop_type              0
  option drop_value             0
  option randgen_seed           0x74d9a2fb
  option dst_global_addr        1
  option dst_local_addr         1
```

Options:

`delay_per_packet_<?>` In the functional block every packet is delayed by this value. The delay per packet can be specified either by a number of cycles (`delay_per_packet_cycles`) OR a number of seconds (`delay_per_packet_secs`). Internally the value is set in number of cycles as a 64 bit value.
cycles: [decimal integer] (valid range: $0$ to $2^{64}-1$)
secs: [double]

`delay_per_byte_<?>` For every byte the packet in the functional block is delayed by this value. The delay per byte can be specified either by a number of cycles (`delay_per_byte_cycles`) OR a number of seconds (`delay_per_byte_secs`). Internally the value is set in number of cycles as a 64 bit value.
cycles: [decimal integer] (valid range: $0$ to $2^{64}-1$)
secs: [double]

`drop_type` There are three drop types which must be set by specifying an integer value. The three types are:

- 0: drop disabled: no packets are dropped
- 1: drop fix: every `drop_value`-th packet will be dropped
- 2: drop random: every (($2^{32}$)/`drop_value`)-th packet will be dropped on average

For both cases `drop_type=1` and `drop_type=2` the setting `drop_value=0` is not valid and will disable the drop functionality, i.e. no packets are dropped. To drop every packet the parameters can be set to `drop_type=1` and `drop_value=1`. With this configuration the generic functional block acts as a black hole.
[decimal integer]

`drop_value` The value which is used to for the drop functionality. The exact meaning depends on the `drop_type`.
[decimal integer] (valid range: $0$ to $2^{32}-1$)

`randgen_seed` This option is used to specify the seed for the random generator which is used for the *drop random* mode of the drop functionality.
[32 bit hexadecimal integer]

`dst_global_addr` Specifies the destination switch in the NoC for the packets that are sent out of the hardware generic functional block.
[decimal integer]

`dst_local_addr` Specifies the destination block in the NoC for the packets that are sent out of the hardware generic functional block.
[decimal integer]

**Software generic FB (Entity name: `SWFB_THREAD`)**

Example:

```
config SWFB_THREAD                  0
  option delay_per_packet_cycles    500
  option delay_per_byte_secs        0
  option drop_type                  1
  option drop_value                 2
  option dst_type                   SWFB_THREAD
  option dst_id                     1
```

Options:

`delay_per_packet_<?>` In the functional block every packet is delayed by this value. The delay per packet can be specified either by a number of cycles (`delay_per_packet_cycles`) OR a number of seconds (`delay_per_packet_secs`).
cycles: [decimal integer]
secs: [double]

`delay_per_byte_<?>` For every byte the packet in the functional block is delayed by this value. The delay per byte can be specified either by a number of cycles (`delay_per_byte_cycles`) OR a number of seconds (`delay_per_byte_secs`).
cycles: [decimal integer]
secs: [double]

`drop_type` There are three drop types which must be set by specifying an integer value. The three types are:

0: drop disabled: no packets are dropped
1: drop fix: every `drop_value`-th packet will be dropped
2: drop random: every (($2^{32}$)/`drop_value`)-th packet will be dropped on average

For both cases `drop_type=1` and `drop_type=2` the setting `drop_value=0` is not valid and will disable the drop functionality, i.e. no packets are dropped. To drop every packet the parameters can be set to `drop_type=1` and `drop_value=1`. With this configuration the generic functional block acts as a black hole.
[decimal integer]

`drop_value` The value which is used to for the drop functionality. The exact meaning depends on the `drop_type`.
[decimal integer] (valid range: $0$ to $2^{32}-1$)

`dst_type` The thread type of the thread to which the packets shall be forwarded.
[valid values: `SWFB_THREAD`, `STATS_THREAD`]

`dst_id` The thread ID of the thread to which the packets shall be forwarded.
[decimal integer]

#### Stats Thread (Entity name: STATS_THREAD)

Example:

```
config STATS_THREAD 0
```

Options:
There are no options available for this entity.

#### Receive Thread (Entity name: RECEIVE_THREAD)

Example:

```
config RECEIVE_THREAD
  option idp 3 SWFB_THREAD  0
  option idp 4 STATS_THREAD 0
```

Note: For this entity there is no ID required since there is only one instance of this entity.

Options:

`idp` Specifies the routing of packets with the corresponding IDP. This option can be specified multiple times.

- `value0`: IDP [decimal integer]
- `value1`: The thread type of the thread to which the packets shall be forwarded. [valid values: `SWFB_THREAD`, `STATS_THREAD`]
- `value2`: The thread ID of the thread to which the packets shall be forwarded. [decimal integer]

#### ETH Block (Entity name: ETH_BLOCK)

Example:

```
config ETH_BLOCK
  option hash 0xabababa0abababa0 2 TO_FB0
  option hash 0xcdcdcdcdcdcdcdcd 3 TO_H2S
```

Note: For this entity there is no ID required since there is only one instance of this entity.

Options:

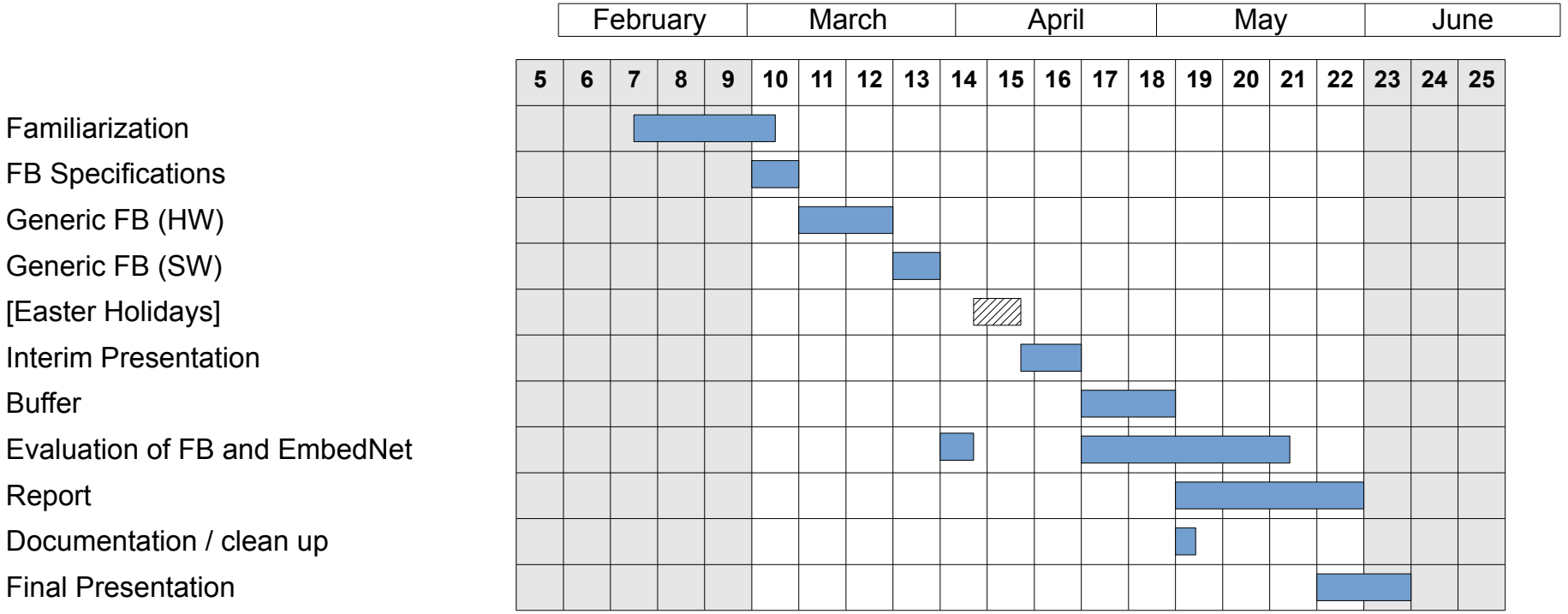
`hash` Specifies the mapping from the 8 byte hash (external header) to the IDP and the NoC address (internal header) which is used for the header translation in the ETH block. This implicitly defines the routing of the packets at the ETH block.
This option can be specified multiple times.

- `value0`: hash [hexadecimal integer]
- `value1`: IDP [decimal integer]
- `value2`: NoC address [valid values: `TO_H2S`, `TO_FB0`, `TO_FB1`, `TO_FB2`]

# Appendix B

# Time Schedule

See next page.


February
March
April
May
June
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Familiarization
FB Specifications
Generic FB (HW)
Generic FB (SW)
[Easter Holidays]
Interim Presentation
Buffer
Evaluation of FB and EmbedNet
Report
Documentation / clean up
Final Presentation
Easter Holidays: 3.4.15 – 9.4.15

# Appendix C

# Project Description

See next pages.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Semester Thesis

# Generic Functional Blocks for FPGA-based Network Nodes

Roman Trüb

Advisor: Dr. Markus Happe, markus.happe@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

1 March 2015 - 31 May 2015

## 1 Introduction

Nowadays the diversity in networked devices, communication requirements, and network conditions vary heavily, which makes it difficult for a static set of protocols to provide the required functionality. Therefore, dynamic protocol stack (DPS) architectures are investigated in which protocol stacks can be built dynamically. In contrast to the static protocol stacks that are used in today's Internet architecture, the DPS architecture splits up the networking functionality into functional blocks, which can be dynamically linked with each other to form arbitrary protocol stacks. The execution environment called EmbedNet is an FPGA-based implementation of the dynamic protocol stack architecture that allows for a dynamic mapping of such functional blocks to either hardware or software.

One major drawback of the EmbedNet platform is the high implementation effort of new functional blocks that are functionally equivalent in hardware and software. This drastically limits the complexity of dynamic protocol stacks that can be tested. However, most experiments only evaluate the packet processing performance of entire protocol stacks. Hence, arbitrary protocol stacks could be formed by interconnecting multiple instances of generic functional blocks that emulate the processing behavior of real-world protocols in time. In EmbedNet, such generic functional blocks are then mapped to hardware and/or to software and process the incoming packets for a specified time and may drop the packet at a given probability. Hence, an exhaustive performance evaluation of the EmbedNet platform can be performed using complex protocol stacks using several instantiations of the generic functional blocks in hardware and software.

## 2 Assignment

This assignment aims to outline the work to be conducted during this thesis. The assignment may need to be adapted over the course of the project.

### 2.1 Objectives

The first goal of this thesis is the development and evaluation of a generic functional block in hardware and in software. The generic functional block should be able to emulate the processing behavior of real-world protocols by delaying packets and/or dropping a packet without actually modifying the packet content. Multiple instances of the generic block (possibly with different parameter sets) have to be combinable in order to form arbitrary protocol stacks.

The second goal is an exhaustive performance evaluation of the current EmbedNet architecture using one or multiple instances of the generic functional block.

## 2.2 Tasks

This section gives a brief overview of the tasks the student is expected to perform towards achieving the objective outlined above. The binding project plan will be derived over the course of the first three weeks depending on the knowledge and skills the student brings into the project.

### 2.2.1 Familiarization

- Xilinx Design Tools (XPS, SDK, Isim, ChipScope)
- EmbedNet architecture, ReconOS execution environment and VHDL libraries
- In collaboration with the advisor, derive a project plan for your semester project. Allow time for the design, implementation, evaluation, and documentation.

### 2.2.2 Architecture and hardware/software design

- Develop a generic hardware function block (in VHDL) with the parameters such as `processing delay per packet`, `processing delay per packet byte`, and `packet drop rate`. The hardware block should not alter the packet payload, but it needs to keep the packet inside the functional block as specified by the parameters.
- Develop a generic software function block (in C) with the same parameters as the hardware functional block. The CPU should be busy for the specified processing time. The packet payload should not be modified.
- Optional: Develop a tool flow that generates an FPGA project (XILINX EDK project) that connects a user-defined number of functional blocks with each other.
- Optional: Develop a methodology that updates the hardware/software mapping of the functional blocks using partial reconfiguration.

### 2.2.3 Implementation

- Determine an appropriate version control system and set it up for further use. You might consider using git and branch the official ReconOS git repository into your git repository.
- Implement the generic hardware and software functional blocks on a Xilinx Virtex-6 ML605 board.

### 2.2.4 Validation

- Validate the correct operation of your implementation.
- Check the resilience of the implementation, including its configuration interface, to uneducated users.

### 2.2.5 Evaluation

- Do a performance evaluation of your implementation using suitable parameter sets for the generic block(s). This evaluation should include a stress test, in order to verify that your hardware thread does not introduce any instabilities into the overall system.
- Perform a performance evaluation of the EmbedNet architecture with at least one hardware and one software functional block.
- Optional: Experiment with multiple functional blocks in hardware and software, where the packets have to cross the hardware/software boundary multiple times.

#### 2.2.6 Documentation

- Provide appropriate source code documentation.
- Write a step-by-step how to that describes the compilation of your code, the loading of the code into the hardware and the execution of your code.
- Write a documentation about the design, implementation, validation and evaluation of your work.

## 3 Milestones

- Provide a project plan, which identifies the milestones.
- One intermediate presentation: Give a presentation of ten minutes to the professor and the advisor. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.
- Final presentation of 15 minutes in the Communication Systems Group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.
- Any software and hardware modules that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.
- Final report: The final report must contain a summary, the assignment, the time schedule and a declaration of originality. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.

## 4 Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps. The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.
- The student is encouraged to write all reports in English; German is accepted as well.
- The core source code will be published under the GNU general public license.

## 5 References

[1] Ariane Keller, Daniel Borkmann, Stephan Neuhaus, and Markus Happe. Self-Awareness in Computer Networks. In International Journal of Reconfigurable Computing (IJRC), Article ID 692076, 2014, Hindawi.

[2] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. „ReconOS – An Operating System Approach for Reconfigurable Computing". In IEEE Micro 34(1), Jan/Feb. 2014.

[3] Git Repository: https://github.com/EPiCS/reconos (branch: v3.0_dev)

[4] Xilinx User Guuide 360: Virtex-6 FPGA Configuration (v3.8) `http://www.xilinx.com/support/documentation/user_guides/ug360.pdf`

**Webpages:** `http://www.epics-project.eu` `http://www.reconos.de`

# Appendix D

# Declaration of Originality

See next page.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

## Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Generic Functional Blocks for FPGA-based Network Nodes

**Authored by** (in block letters):

*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| Trüb | Roman |
| | |
| | |
| | |
| | |

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*

# Bibliography


[1] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, C. Plessl, and M. Platzner. Reconos – an operating system approach for reconfigurable computing. *IEEE Micro*, 2014.

[2] F. Deragisch. Network protocols for embedded devices with dynamic hardware/software mapping. Master's thesis, ETH Zurich, 2012.

[3] M. Happe, Y. Huang, and A. Keller. Dynamic protocol stacks in smart camera networks. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Dec 2014.

[4] R. Huber. A dynamic hardware architecture for future networks. Master's thesis, ETH Zurich, 2012.

[5] A. Keller, D. Borkmann, S. Neuhaus, and M. Happe. Self-awareness in computer networks. *Hindawi International Journal of Reconfigurable Computing (IJRC)*, page 16, Jun 2014. Article ID 692076.

[6] S. Kronig. Intrusion prevention for flexible protocol stacks. Master's thesis, ETH Zurich, 2013.

[7] Y. Yang. Hardware encryption for embedded systems. Semester's thesis, ETH Zurich, 2013.