RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Moritz Schilling

A Highly Efficient Transport Layer for the Connection of Neuromorphic Hardware Systems

Diplomarbeit

HD-KIP-10-09

KIRCHHOFF-INSTITUT FÜR PHYSIK

Faculty of Physics and Astronomy

University of Heidelberg

Diploma thesis

in Physics

submitted by

Moritz Schilling

born in Marsberg, Germany

January 2010

A Highly Efficient Transport Layer for the Connection of Neuromorphic Hardware Systems

This diploma thesis has been carried out by Moritz Schilling at the

Kirchhoff Institute for Physics

University of Heidelberg

under the supervision of

Prof. Dr. Karlheinz Meier

A Highly Efficient Transport Layer for the Connection of Neuromorphic Hardware Systems

Neuromorphic hardware is a highly innovative and very promising tool in the ongoing neuroscientific endeavour of understanding and replicating the extremely complex information processing device known as the human brain. The highly accelerated neuromorphic system used throughout this thesis can produce a multitude of neuronal events during periods of high activity, which can easily fully load or even exceed the bandwidth of modern computer networks. This work delves into the efficient connection of such a system to the controlling software layers. The developed software includes state of the art principles of software engineering, such as concurrent threads and communication mechanisms without copy processes. These methods have been used to develop an essential, novel software component which has been successfully integrated into the already existing framework. Several experiments have been performed to prove the functionality of the implemented software, which is shown to be able to provide a significant speed-up for future neuronal experiments.

Eine hocheffiziente Transportschicht zur Anbindung neuromorpher Hardware

Neuromorphe Hardware ist ein hoch innovatives und vielversprechendes Werkzeug im Bestreben der Neurowissenschaft ein hochkomplexes Rechenwerk wie das menschliche Gehirn zu verstehen und nachzubilden. Bereits existierende, stark beschleunigte neuromorphe Systeme können große Mengen neuronaler Ereignisse produzieren, welche die Bandbreite auch moderner Rechnernetze schnell aus- oder überlasten können. Die vorliegende Arbeit befasst sich eingehend mit der möglichst effizienten Anbindung eines solchen Systems an die kontrollierenden Softwareschichten. Die entwickelte Software macht dabei Gebrauch von aktuellen Prinzipien der Softwareentwicklung, wie nebenläufige Threads oder Kommunikationsmechanismen, die ohne Kopiervorgänge auskommen. Diese Prinzipien wurden benutzt, um einen neuen, essenziellen Softwarebaustein zu entwickeln, der sich perfekt in die bereits bestehende Struktur einfügt. Verschiedenste Experimente wurden durchgeführt um zu zeigen, dass die entwickelte Software in der Lage ist, zukünftige neuronale Experimente erheblich zu beschleunigen.

Contents

1 Introduction

Neuroscientists explore the structure, development and working principles of nervous systems. These complex structures consist of a huge number of interconnected processing elements, the neurons, which operate in parallel. Neurons communicate via action potentials, the so-called *spikes*. Once emitted by a pre-synaptic neuron, these quasi-digital pulses travel along organic cables, the axons, which end in synaptic connections to other neurons. Via synapses the spikes affect the state of post-synaptic cells. Thus, the output of each neuron contributes to the input of many other neurons. While the communication is based on simple-structured pulses, neurons process and integrate their input in a much more complex manner. When a cell is being sufficiently excited by its input, it broadcasts this information to other neurons by emitting a spike itself.

Today, the dynamics of individual building blocks of biological nervous systems are quite well understood. Already in 1952, *Hodgkin and Huxley* [1952] developed a detailed model describing membrane dynamics. More recent research addressed a more abstract formulation of essential properties of neurons and synapses [*Brette and Gerstner*, 2005; *Markram et al.*, 1998]. Nevertheless, the impressing computational power and adaptability of highly developed nervous systems originates from network effects which arise from the complex interplay of their components. Since the versatile interaction of large networks is difficult to monitor and decipher, less knowledge on this important topic has been extracted so far. Accordingly, the study of network phenomena is subject to intensive research.

Instead of gaining insight into network dynamics directly through measurements, an alternative approach is to calculate the behaviour of networks and to compare the predicted properties with experimental results on a higher level. This is achieved in different ways.

One way is to use mathematical models based on the *analytical* treatment of equations representing components of neural networks. While elegant, the involved equations make calculations difficult due to the time-continuous nature of membrane dynamics in contrast to the binary communication via spikes. In order to overcome this obstacle, software simulators allow to *numerically* calculate the time-development of the system. Common tools like NEURON [*Hines and Carnevale*, 2003] or NEST [*Gewaltig and Diesmann*, 2007a] offer complete control over all parameters and tracking of the entire network state. This way, network experiments in the order of seconds can be performed.

Nevertheless, many important features of self-organisation in neural networks occur on much longer time-scales. Examples are long-term learning and the formation of memory. Furthermore, large networks exhibit a huge amount of parameters which define the system. In order to establish a comprehensive understanding of the network dynamics, it is necessary to explore high-dimensional parameter spaces. This requires the repetitive execution of almost identical experiments. Both of the above described types of experiments are limited by the computational power of present computers in case of large network models.

Due to these obstacles a fundamentally different approach for modelling neural networks emerged in the 1980s [*Mead and Mahowald*, 1988; *Mead*, 1989]. Neuromorphic hardware devices physically implement neuron and synapse models which follow similar dynamics as their biological counterparts. Consequently, neuronal systems on such hardware devices evolve both in parallel and time-continuously. Therefore, the simulation of neural networks via neuromorphic hardware devices is called *emulation*. Additionally, the dimensioning of the circuitry allows a highly accelerated operation compared to biology. Due to the inherent parallelism, the high emulation speed remains widely unaffected when increasing the network size. Today, an active community develops analogue or mixed-signal VLSI[1] models of neural systems [*Vogelstein et al.*, 2007; *Merolla and Boahen*, 2006; *Häfliger*, 2007; *Serrano-Gotarredona et al.*, 2006; *Renaud et al.*, 2007; *Schemmel et al.*, 2007, 2008; *Ehrlich et al.*, 2007].

All of the above described types of neuroscientific research are addressed within the FACETS[2] [*FACETS*, 2009] research project. The goal of this project with members across Europe is to find novel computing paradigms inspired by biological neural systems. The hardware system utilised throughout this thesis is being developed within the FACETS project.

In order to make neuromorphic hardware devices valuable research tools, the technical challenges coming along with this technology have to be faced. One of the major obstacles imposed by the speed-up factor in highly accelerated hardware systems is to handle the huge amounts of data generated during experiments and for configuration. Ideally, the communication framework is able to transport all information to and from the hardware system. In contrast to that, the bandwidth of links in the communication framework is limited in practice. Consequently, either the network size or the event rate has to be constrained in order to prevent the loss of information. Hence, the communication software has to maximize the data throughput.

An additional requirement to the communication software is the minimisation of latency, i.e. the processing or routing delay occurring during information transmission. This will be important for an interaction between the neural network emulated by the hardware and a virtual environment. Another application where low latency is desired is the routing process for internet backbones.

For these reasons, the optimisation of both, throughput and latency, is essential for exploiting the advantages of neuromorphic hardware devices. The development of a highly efficient transport software layer between a neuromorphic hardware device and other software layers, is the goal of this thesis.

Outline

In advance to the presentation of the new transport layer in chapter 3 and the results of various measurements with it in chapter 4, the current software and hardware environment of the FACETS stage 1 system will be described. To what extent the developed software fulfils the requirements of being an efficient transport layer for the connection of a neuromorphic hardware device is discussed afterwards in chapter 5.

[1]Very Large Scale Integration
[2]Fast Analog Computing with Emergent Transient States

2 FACETS Neuromorphic Hardware Device & Software Framework

In this chapter, a short overview of the currently available software framework and the utilized neuromorphic hardware device is given. First the neuromorphic chip *Spikey* containing the analogue neuron circuits is described, followed by the controlling software stack, starting from the highest and down to the lowest layer. Afterwards the previously used connection between software framework and neuromorphic device is presented. Its shortcomings are outlined to motivate the implementation of the new transport protocol. For the sake of completeness, the planned successor of the stage 1 hardware system, the FACETS stage 2 wafer-scale integration system, is briefly described in the last section.

2.1 Stage 1

The FACETS stage 1 hardware environment consists of a printed circuit board, the so-called backplane [*Philipp et al.*, 2007] which hosts up to 16 *Nathan*-Cards [*Grübl*, 2007] via a $40-80$ MBit/s serial bus[1]. Each card is equipped with an additional FPGA, the *Nathan*, which controls a buffer memory and *Spikey*, the neuromorphic chip.

2.1.1 Spikey – the Neuromorphic Chip

The neuromorphic chip implements 384 leaky integrate and fire neurons with conductance based synapses [*Destexhe et al.*, 1998]. The neuron membrane is emulated by a capacitance. Charge from incoming spikes flows onto the capacitor, changing its voltage. If a certain threshold is reached, an outgoing spike is detected and the voltage is set to a reset potential for a short period of time. The spike of the pre-synaptic neuron is first routed through synapse drivers and then through the corresponding synapse circuits to its targets – the post-synaptic neurons. The change in the membrane potential of the target depends on the type (excitatory or inhibitory) and the weight of the synapse involved. For an excitatory synapse, the membrane potential temporarily increases and for an inhibitory synapse, it temporarily decreases. The weight parameters are implemented in the synapse circuit and control the impact of a spike on its target. In most biological neural networks, synaptic weights may vary over time. In order to achieve such behaviour, various biologically inspired plasticity mechanisms have been implemented onto the hardware (*Schemmel et al.* [2006]; *Schemmel et al.* [2007]). For example, *STDP* is realized by changing the weight of a synapse depending on the causal or acausal correlation between a pre- and a post-synaptic spike.

A generated spike can also be routed off-chip to the corresponding *Nathan*. This inter-Spikey communication is realised via a MGT[2]-based network implementing isochronous con-

[1] the uncertainty results from slightly different clock rates across the backplanes
[2] Multi-Gigabit Transceiver

nections [*Philipp et al.*, 2007]. Data dedicated to the host computer is buffered in local SDRAM and, upon request, is transferred via backplane to the host computer.

2.1.2 Software Framework

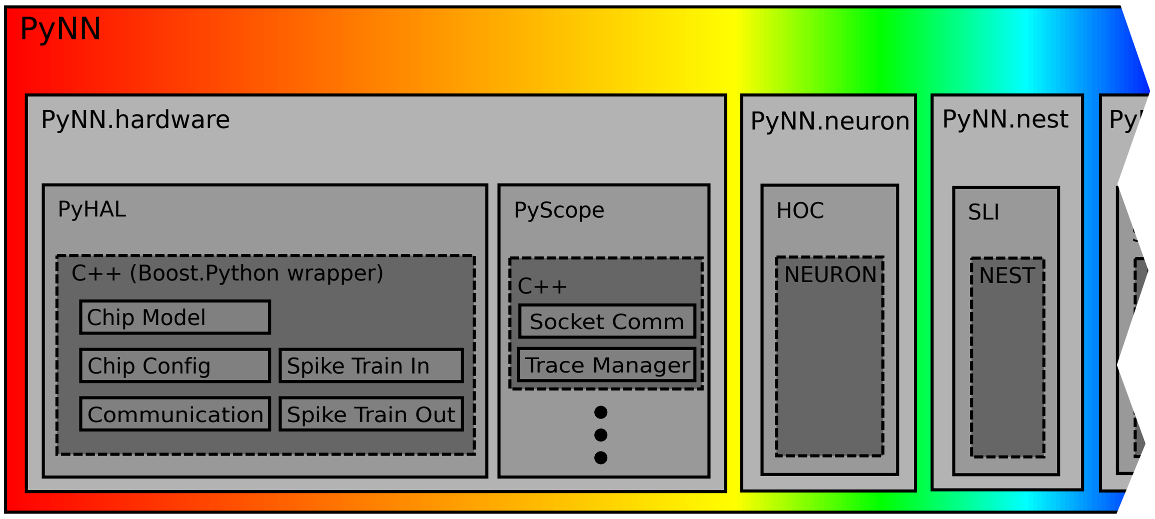


Figure 2.1: Software stack which includes PyNN, PyHAL and the wrapper functions to SpikeyHAL. The transport layer is not explicitly shown but indicated by the box *Communication*. Figure based on *Brüderle et al.* [2009], with friendly permission from Eric Müller.

The highest level of the software environment is represented by PyNN [*Davison et al.*, 2008], a meta language for an unified description of neuronal network experiments based upon the scripting language Python [*Rossum*, 2000]. The main aim of PyNN is to provide a unified interface for accessing various neuronal simulation tools (e.g. NEST [*Diesmann and Gewaltig*, 2002; *Gewaltig and Diesmann*, 2007b; *Eppler et al.*, 2008], NEURON [*Hines and Carnevale*, 2006; *Hines et al.*, 2009]) as well as neuromorphic hardware devices. The model-specific parameter set is translated by corresponding PyNN modules to the intrinsic parameter space of the chosen simulator or neuromorphic hardware device. Thus, the portability and comparability of experiments across multiple back-ends is achieved. This is necessary to identify behaviour which is generated by the simulator itself and to have access to the various feature sets (e.g. different neuron models or parameter values) when checking the model for its universality.

When using PyNN to access the FACETS stage 1 hardware device, the PyNN.hardware.-stage1 [*Brüderle et al.*, 2009] module first needs to be imported. This module links the unified interface to the specific interface of PyHAL. PyHAL provides functions to place and interconnect neurons of a neural network on a *Spikey*. Furthermore, all biological parameters are translated into the hardware-specific domain and vice versa. The configuration parameters and spike-patterns are send to SpikeyHAL [*Grübl*, 2007] where they are implemented into hardware-specific commands or Slow Control[3] commands [*Fieres et al.*, 2004]. Every command is transmitted to the neuromorphic device where it is executed and a response, de-

[3]Slow control means in this context that such commands are executed more slowly than neuronal events occur within the neuromorphic device

pending on whether its execution was successful or not, is generated. Aside from this response read commands provide the value stored at the requested location in the memory.

2.1.3 Former Data Flow

The software framework which accesses the backplane and its neuromorphic chips utilizes a low-level program called *dwserver*, which provides the connection via a PCI card installed in the hosting computer, the so called Darkwing PCI card [*Schürmann et al.*, 2002]. The *dwserver* program polls incoming messages from a user program out of its message queues. Once a message is received, the included Slow Control commands are passed to the Darkwing PCI card, which is handled by a proprietary driver called *WinDriver*. The Darkwing PCI card hosts the connection with the backplane via a Low-Voltage-Differential SCSI[4]. After the execution of the command by the hardware system, *dwserver* passes the answer back to the dedicated user program.

Originally, the Darwing PCI card was intended to be a test environment for newly developed mixed-signal chips. Now it is mainly used to provide a connection to a backplane. It works well with the software framework. Experiments with neural networks were successfully set up and performed [*Kaplan*, 2008; *Bill*, 2008; *Müller*, 2008]. However, there are some disadvantages. The most important is the proprietary driver *WinDriver* developed by Jungo Ltd., 2007. Although the driver supports enhancements like DMA[5] to relieve the computer's processing units and to speed up communication, it is not possible to use them without malfunctions. For example, if the *WinDriver* kernel module is loaded on multi-core architectures, the proprietary kernel module deadlocks frequently. Another disadvantage is the fact that for every backplane which shall be accessed, a separate Darkwing PCI card plus SCSI cable are necessary and have to be installed. Throughput measurements performed in this thesis will show that the bus between the Darkwing PCI card and the Nathan FPGAs is not fully saturated. While the bus bandwidth theoretically allows a throughput of about 5-10 MB/s, in reality only a small fraction of this is achieved (see section 4.3.3). This unnecessarily increases the time for setting up an experiment and the retrieval of the buffered results.

There are several possibilities to increase the bandwidth and throughput of the link between the neuromorphic hardware system and the controlling and processing software. For example, the existing Token-Ring-based connection could be replaced with a different bus system e.g. *HyperTransport*. But this would need extra hardware components on the backplane and on the computer system. Another potential solution would be an USB[6] link; All modern computers are equipped with such ports. However, a disadvantage of USB is the complex control of the software drivers which exacerbates the development of higher software layers.

Therefore and because of the already existing Ethernet connectors for the FPGA on the backplane, a 1 GBit/s Ethernet link was selected to replace the SCSI link as the communication channel between the software framework and hardware environment. A properly installed 1 GBit/s link is more than sufficient to saturate the bus on the backplane. It is *not* sufficient for the bus between *Spikey* and *Nathan* which has a bandwidth of $\simeq 3-4$ GBit/s, but Ethernet standards already exist with much higher bandwidths. Obstacles, like the mentioned driver problems, rarely exist in modern networking drivers.

[4]Small Computer System Interface
[5]Direct Memory Access
[6]Universal Serial Bus

2.2 Stage 2

The FACETS stage 1 hardware system supports 384 neurons and 10^5 synaptic connections per *Spikey* and up to 16 *Spikeys* per backplane. A simulation of the human brain's neocortex needs about 10^{10} neurons [*Pakkenberg and Gundersen*, 1997]. It would need thousands of interconnected backplanes in order to just reach the number of neurons not to mention the required communication bandwidth, which is impracticable. Even for the emulation of a single hyper-column in the cortex, which consists of about 10^4 neurons, more than one backplane would be necessary [*Johansson and Lansner*, 2007]. In addition to that, the FACETS stage 1 hardware system can not be used in an interactive mode. The results or output spikes are recorded and saved in the memory on the backplane and can be read out only after the experimental run is finished. But neuronal experiments in an interactive operation mode need to exchange neuronal event data with the processing layers instantaneously.

In order to move a significant step towards the emulation of larger neuronal network experiments, the FACETS stage 2 wafer-scale system is under development [*Schemmel et al.*, 2008]. Instead of cutting a wafer into single chips all circuits are left as a single wafer and become connectable by adding additional buses afterwards. A single system is designed to support up to $1.63 \cdot 10^5$ neurons and $4 \cdot 10^7$ synapses, while the acceleration factor of 10^4 compared to biological real-time is retained. Even bigger neural networks can be set up by interconnecting multiple of those neuromorphic systems. Such large numbers of neurons and synapses together with the speed-up factor of the system can lead to a comparatively large neuronal event traffic of about 10^2 GEvents/s, which eventually have to be routed off the chip to the controlling computer(s). The connection will be established via $12\times$ or $24\times$ 1 GBit/s Ethernet links, providing theoretically a maximum bandwidth per wafer-scale system of about 12-24 GBit/s. How to handle such large event rates or data packet rates will be discussed in this thesis (see chapter 5).

3 Development of a New Transportation Layer

This chapter describes the development of the new transportation protocol for the interface between the FACETS stage 1 hardware system and the attached host computers. At first, a short overview of operating systems and their interface for networking applications is given. The decision to develop a new protocol instead of using existing ones will be motivated thereafter. Finally, the requirements on the protocol are introduced and, considering these, the implemented software is discussed.

3.1 Operating System Issues

Before delving into more profound issues of software engineering, the environment has to be selected in which the software framework will operate in. Several possibilities exist, such as the Microsoft Windows operating system (OS), Apple MacOS X and different Unix derivatives. For the following reasons Linux, an Unix derivative, has been selected:

- **compatibility:** PyNN, the neuronal modelling language, including the module PyNN.-hardware.stage1 is interfacing lower software layers all of which were developed for Linux (see chapter 2).
- **open OS:** The source code of this OS is available for everyone to look at and modify it. This is especially useful when writing low-level code.
- **OS for free:** Linux is free of charge because it is released under the GNU GPL[1] [*GPL 2009*].
- **standardized API[2]:** Linux supports most of the POSIX standard[3]. This ensures portability across different platforms [*IEEE*, 2004].

When porting user space code to kernel space or writing a new device driver, this selection becomes even more obvious. How to develop a Linux device driver is very well understood and documented [*Corbet et al.*, 2005]. More details about this will be revealed in section 3.3.1.

3.1.1 Sockets

Sockets [*Stevens et al.*, 2003] are the interface between an application and the operating system when accessing networking devices, on Unix and non Unix-like operating systems. Although they have been developed by Berkeley University as early as 1983, they still are widely used. Like regular files, sockets are first opened, then accessed and finally closed with the standard system API. If opened, the system allocates buffer space, sets up all necessary administrative structures and prepares the network stack for the transmission and retrieval

[1]GNU General Public License
[2]Application Programming Interface
[3]Portable Operating System Interface (for Unix)

Layer	**Protocol types**	
Application	HTTP	SpikeyHAL
Transport	TCP	Slow Control Transport Protocol
Network	IP	
Physical	Ethernet	

Table 3.1: Position of the Slow Control Transport Protocol (SCtrlTP) in a simplified networking layer model. For comparison the positions of HTTP, TCP, IP are also given.

of data. In general, this data is not transmitted and received contiguously, but in chunks of a maximum size that depends on the capabilities of the networking device.

3.1.2 Network Stack

When data chunks arrive at the networking device or are going to be sent to a remote host they traverse the so-called network stack. At the different layers of the stack (as shown in table 3.1), different protocol handlers process them according to options set in an incoming chunk or the assigned socket. For example, if a TCP/IP-over-Ethernet[4] packet arrives at the networking device the system first calls the Ethernet protocol handler [*Enck*, 1994]. This handler checks the destination MAC address[5] and discards the packet – the so called Ethernet frame – if it is not addressed to the host. The same happens if the checksum of the frame is not equal to the checksum computed by the handler (although nowadays this is already done by the networking device itself). Last but not least, the Ethernet handler determines the next protocol handler by the type entry of the Ethernet frame and passes the payload to the corresponding handler; in this example to the IP protocol handler. The IP handler itself performs other tests and routes the encapsulated TCP packet to the TCP handler. All these handlers strip off some amount of data which should be of no further interest to the higher layers. Finally, the encapsulated data will be read by single or multiple applications. Analogously, data which is going to be sent traverses from the highest layer to the lower layers and is ultimately transmitted by the physical networking device.

3.1.3 TCP/IP

TCP/IP [*Braden*, 1989] is a combination of two protocols which are working very closely together. In the following, the main focus lies on the TCP protocol, as it is similar to the developed protocol. TCP is a connection oriented and lossless protocol. Connection oriented means that before any transmission/retrieval can be started, the sending part and the receiving part synchronize themselves including a parameter exchange and ubiquity test. Only if the connection was successfully established data transfer is allowed. Lossless means that any amount of data which is going to be sent will be received from the other part without loss and reordering. This is also known as the conservation of data principle which is a general feature of ARQ protocols[6] including TCP.

Despite the fact that TCP is widely used and very well implemented in every modern operating system available, it is not an option for the tasks within the scope of this thesis.

[4]Transmission Control Protocol/Internet Protocol
[5]Media-Access-Control address
[6]Automatic Repeat reQuest protocols

Several limitations do not allow the usage of TCP and/or IP:

First of all, the most important limiting factor is the size of the utilized FPGA in the FACETS hardware systems. An FPGA-Chip consists of many unified block units called slices – but of course the amount of slices is limited. The currently used FPGA, a Virtex4, is too small to implement a full TCP design in addition to the other logic units. The current occupancy of this FPGA is above 90% of the available slices. Besides that, most of the features of TCP are designed to deal with large and unknown networks. These features consume space in packets and – more crucially – need significant execution time which increases overall latency. Two different checksums have to be calculated, one for the IP header and one for the encapsulated TCP packet. Even if the resulting latencies could be neglected, a custom header has to be transmitted to the hardware device to reset it, define payload etc. in any case. But the overhead resulting from TCP/IP is not necessary for the developed protocol, as discussed in the following.

3.2 Functional Requirements

In this section an introduction of the most important aspects of information theory with respect to information exchange over unreliable communication channels will be given. The previously mentioned TCP-protocol embodies all aspects which ensures reliable information exchange while the IP-protocol assures the routing of packets to their destinations in large networks. To what extent SCtrlTP, the developed protocol, has to incorporate those aspects, is discussed throughout this section.

3.2.1 Routing Traffic

At first, it is necessary to establish the routing capabilities required from the new protocol. The network topology is very simple. It consists of two nodes (or endpoints): the host PC and the FACETS stage 1 hardware system. A third node may be added in between, representing an interconnected switch which routes traffic from the source to its destination. Therefore, the advanced routing capabilities of IP, which are useful if operating in larger networks, are of no need. The fact that IP is not required has two advantages:

- Space is saved in the resulting frame, since there is no IP header.
- There is no need of an additional underlying protocol named ARP[7] which maps MAC addresses to IP addresses by sending/receiving request/response frames.

For SCtrlTP presented in the next section, only MAC addresses are needed to ensure a correct routing of frames through the network. In order to route traffic to the distinct chips on the backplane, an extra "address" (this corresponds to the DEST_NAT entry in tab. 3.2) in the protocol is used, which is unique per backplane. Uniqueness of these addresses over multiple backplanes has to be ensured in a layer different from the transport protocol itself.

3.2.2 Conservation of Data Flow

As mentioned before, lossless transmission of data over a generally unknown and unreliable network requires a suitable protocol implementation. It should be capable to fulfil the following minimum requirements:

[7]Address Resolution Protocol

- Data has to be kept in buffer space until it is successfully transmitted to the other side. This is only possible if there is a return path via which the other side acknowledges a successful transmission.
- If packets got lost - even acknowledgement packets - the sending side has to retransmit them.
- Contiguous data has to be transferred in correct order, although it is sent in form of chunks. An underlying transport protocol must not modify the data itself. That means, it has to be transparent for the utilizing software.

A protocol class which meets at least these three requirements is the class of ARQ protocols [*Fairhurst*, 2002]. This protocol class is divided into several subclasses of which two will be described. Every protocol in this class uses an enumeration system to keep ordering of data chunks - the so called sequence number. The same number is used in return to acknowledge the corresponding frame(s) if received successfully. On the other hand, no return packet or a negative acknowledgement is sent if a chunk was not received successfully. The sending side has to retransmit all frames at the time when it must be assumed that frames have been lost or after a negative acknowledgement has been received. In the most simple realisation of this protocol class, the Stop-and-Wait algorithm, the sending side transmits one frame and waits until an acknowledgement is received or retransmits this frame after a certain time. Only then it proceeds in transmitting other frames.

This is not very efficient if it is unlikely to loose a frame by physical transmission errors. In the following, η denotes the efficiency of an ARQ protocol compared to a simple transmission of all frames in the absence of acknowledgements. This efficiency depends on the bandwidth β, the amount of packets N, the size of one data packet P as well as on the minimum size of an acknowledge frame A. The following equations define a rough efficiency estimate of a Stop-and-Wait protocol:

$$\eta = \frac{\Delta t_{min}}{\Delta t_{snw}} \tag{3.1}$$

$$\Delta t_{min} = \frac{NP}{\beta} \tag{3.2}$$

$$\Delta t_{snw} = N\left(\frac{P}{\beta} + \frac{A}{\beta}\right) \tag{3.3}$$

$$\implies \eta_{snw} = \frac{P}{P+A} \tag{3.4}$$

The minimum delay Δt_{min} is the time it takes to physically transmit the data of size $N \times P$ with the bandwidth β. The efficiency results from the comparison of the minimum delay with the delay Δt_{snw} of the Stop-and-Wait algorithm which results from waiting for every single acknowledgement. The time it takes for any software to process an incoming frame and prepare the transmission of an acknowledging frame as well as the length of the cable are neglected. When taking all delays into account, such as the time for the transmitting/waiting/receiving-cycle on the sending side and the receiving/processing/transmitting-cycle on the recipient, η_{snw} becomes worse for Stop-and-Wait.

An optimal protocol implementation tries to keep the link saturated and does not wait for an acknowledgement of every single frame. Such an algorithm, known as the Sliding Window

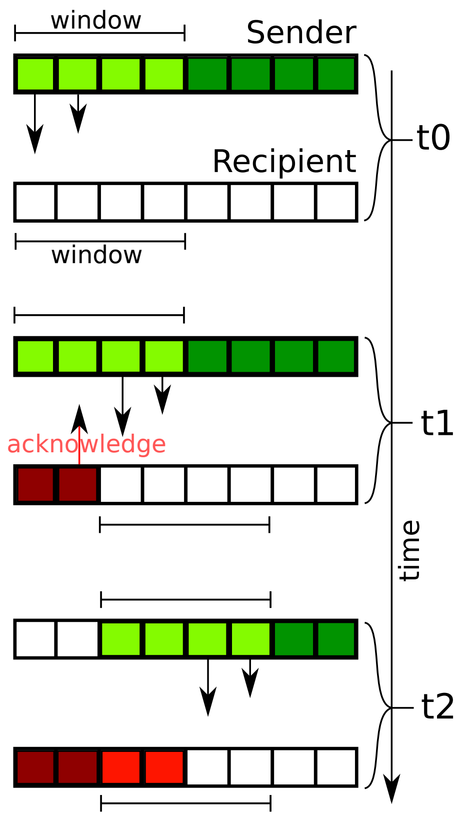


Figure 3.1: Example of unidirectional transmission of data packets from one buffer to another by using the sliding window algorithm. The sender window represents the packets a sender is allowed to send without receiving an acknowledgement from the receiver. The receiver window is defined by the number of packets coming out of order the receiver is able to buffer without overflow.
The current states of the sender window, the receiving window and the connecting link are shown at some intermediate timesteps **t0, t1, t2**. At time **t0** the transmission of four of eight packets is in progress but none was received by the recipient yet. After two packets have been received, the window is moved ahead and an acknowledgement is send at time **t1**. If the acknowledgement arrives at the sender its window will also be moved forward and new packets transmitted. Meanwhile, at time **t2**, the other two packets of the initial window have been received by the recipient. The window position was not yet changed.

algorithm[8] exists and can reach the upper throughput limit [*Peterson and Davie*, 2003]. This algorithm preserves a certain amount of data units in buffer and transmits only a fraction of it in one burst of packets. Only then an acknowledgement has to be received which results in discarding those packets acknowledged and fetching new data into the buffer. The number of packets or amount of data being transferred in a burst is defined by the so called sliding window (see figure 3.1).

$$\eta_{gbN} := 1 \tag{3.5}$$

$$\Longrightarrow \frac{\eta_{snw}}{\eta_{gbN}} = \frac{1}{1+\frac{A}{P}} \tag{3.6}$$

The efficiency estimate of the sliding window or Go-Back-N in comparison to the Stop-and-Wait algorithm is given in equation 3.6. In the case that no packet loss occurs and all acknowledgements arrive during active transmission, the efficiency η_{gbN} of that algorithm reaches the theoretical maximum of 100% efficiency. That means, that the window size has to match the overall capacity[9] of the link. The overall capacity is the sum of all available buffer sizes and the product of bandwidth and minimum signal propagation delay. Furthermore,

[8]the corresponding protocol subclass is known as the Go-Back-N class
[9]the overall capacity is known as the bandwidth-delay-product

both, sending and receiving side, have to handle the packets at the same speed or else the drop rate increases significantly. Hence, in reality much more effort has to be put into the design of software to avoid the mentioned insufficiencies.

3.2.3 Conservation of Equilibrium

Although the mentioned algorithms ensure a reliable data transmission, they are not sufficient to keep the throughput constant in non-ideal systems. In general, the transmitting or receiving side has no knowledge about bandwidth, Round Trip Time, buffer spaces and state of the remote side, not to mention the topology of the underlying network. The Round Trip Time (RTT) is the delay between the transmission of a packet and the retrieval of the corresponding acknowledgement. Retransmitting eventually lost packets too early or too late results in an unnecessary waste of bandwidth. If a packet is retransmitted too late, the recipient waits on a lost packet too long. If a packet is retransmitted too early (before the arrival of the acknowledgement), this packet is dropped because the initial one was, in fact, not lost. Adjusting the time-out of packets already sent by measuring the RTT is necessary to predict a real packet loss.

In general, the window size has to match the overall capacity of a link. Otherwise, the overall latency or RTT increases due to waiting delays while the overall throughput decreases. Time in which new packets could be transmitted is wasted. Even if an algorithm keeps track of the current RTT and the window size is properly chosen, there are cases where data gets lost because the recipient itself is not fast enough to handle those bursts of packets. The internal buffers of the recipient would fill up if it handles packets systematically slower. In such a worst case situation in which these buffers can not be emptied, this leads to an oscillating behaviour of the system with high packet loss due to constantly congested buffers. Without adapting the transmission rate or the window size to systematic packet loss, throughput could suffer significantly[10]. For detailed information of utilized congestion control algorithms see *Nagle* [1984] and *Jacobson and Karels* [1988].

In the case of the stage 1 system, the network topology and the buffer space sizes are well-known. Therefore a fixed maximum window size and a quite stable Round Trip Time are assumed because there are no different routes for our packets to take. Hence, SCtrlTP has to support the following minimum requirements:

- Simple measurement of the Round Trip Time to assess a necessary retransmission of a packet.
- Counting to a maximum number of retransmission attempts to inform the user(s) that a backplane might not be accessible.

3.3 SCtrlTP – an Efficient Transport Protocol Implementation

Keeping in mind the discussed requirements, the format of the protocol defined by Dr. Stephan Philipp and the author is presented in table 3.2. All of the features provided by the hardware device are adjustable by certain protocol entries. First of all, the entries are at least byte aligned to ensure fast processing by software. Byte alignment means that every offset of a distinct entry in the protocol is a multiple of one byte. Otherwise, all bitwise operations

[10]those prevention mechanisms in actual TCP implementations are known as congestion avoidance algorithms

Byte offset	Size	Name	Description
0	6	DEST_MAC	MAC address of the destination
6	6	SRC_MAC	MAC address of the source of the packet
12	2	TYPE	A number which identifies the encapsulated protocol SCtrlTP: 0x07ff
14	1	ARQ_FLAGS	Bit #0: Reset the whole FPGA Bit #1: Reset protocol handler Bit #2: Reset the Ethernet handler & PHY Bit #1: Reset statistics
15	1	PL_TYPE	Defines the type of payload Slow Control: 0
16	1	SEQ	Bit #0-6: Sequence number of packet Bit #7: sequence number is valid
17	1	ACK	Bit #0-6: Acknowledgement number Bit #7: unused equals to the last valid sequence number received from DEST_MAC
18	1	PL_FLAGS	Bit #0-3: type of payload entries Slow control commands: 0 Configuration data: 1 Slow control response: 2 Bit #4-7: Configuration flags
19	1	#ENTRIES	Amount of entries in payload
20	2	DEST_NAT	Defines destined Nathan(s)
22	...	PAYLOAD	Begin of entry #1

Table 3.2: Header of a SCtrlTP packet with a short description of the entries.

in software would be much more expensive in matters of CPU load. The first 14 Bytes of a SCtrlTP packet define the Ethernet header which defines where the packet comes from, where it has to be transmitted to and what protocol handler has to process it. The next 4 Bytes control the bidirectional data flow (sequence number and a piggyback accumulative acknowledge number), define what payload is transmitted and contain flags to reset distinct parts of the SW/HW protocol handler. Depending on the type field, there is an additional Slow Control specific header, also 4 Bytes long, which defines payload type, configuration flags, number of entries in payload and, most importantly, tells the protocol handler with whom to communicate with. The format of all currently valid payload entries is shown in the appendix in section A.3.

3.3.1 Kernel Space vs. User Space

As mentioned before, the protocol handlers which are integrated in Linux reside at the system's heart – the kernel. In kernel space the protocol handlers are processed as fast as possible if properly programmed. Moreover, they can make use of the full instruction set the CPU and the kernel provide. On the other hand, any software residing in user space has a lower priority than kernel code and is therefore slower in execution. Nevertheless, there are certain advantages for software in user space. In user space the software does not have to deal with memory management in its entire complexity. Errors generated by user space code are less drastic, whereas flaws in kernel software have far-reaching consequences even leading to a crash of the operating system itself.

For that reasons, the protocol software was programmed in user space while allowing the option of implementing this code as a kernel module[11] in the future. Kernel space variable definitions[12] and as few system calls as possible were inserted. For the FACETS stage 1 hardware this approach is more than sufficient because modern computer systems are fast enough to saturate even a 1 Gbit Ethernet link from user space. For the FACETS stage 2 wafer-scale hardware as presented in chapter 2 though, the transport protocol has to be ported to kernel space. This will be further discussed in chapter 5.

3.3.2 Interface Definition

In chapter 2, the whole software framework operating with the FACETS stage 1 neuromorphic hardware has been presented. The existing interface to the low-level communication code via the Darkwing PCI card is reused to provide back-ward compatibility with former software. However, there is no need to still use message queues as the communication channel between higher software layers and SCtrlTP. Every access to a message queue is a system call which is followed by a context switch[13] and significant kernel work until switching back. A more efficient IPC-mechanism[14], POSIX shared memory [*IEEE*, 2004] with synchronisation and protection mechanisms in user space, is used. Shared memory is a piece of system memory which can be mapped into the address space of multiple related or unrelated processes. When writing to it, the changes can be instantaneously visible to the other processes without copying data from process A to process B (this requires further effort though). Of course, memory corruption has to be prevented in case of multiple processes trying to access the same location in memory at the same time.

The main interface to the software protocol handler is realized by shared memory based FIFO queues[15]. Everything pushed into it is read from it again in the order of arrival. This ordering is desired as it should be a property of the protocol itself (see section 3.2.2).

The interface to shared memory based FIFO queues is designed to be generic as possible and safe with regard to multiple, concurrent accesses. This interface preventing data corruption has to be

- **correct:** Situations must not occur in which two processes could access the same data. Even multiple read accesses have to be prevented since the shared queues have no knowledge about the content of the entries. For example, if the content of an entry is a pointer and two processes acquire that pointer this will probably corrupt the data which is referenced by it. Therefore, a single read operation on a queue destroys the validity of one element. Also, any suspended processes which are waiting for a queue to contain valid data or free space have to be notified[16].

[11]a kernel module is a system driver loadable at runtime when needed

[12]a variable in kernel space has a well defined bit width while user space variable definitions could vary on different architectures

[13]a context switch means, that one process is suspended with all states being saved and another process is executed afterwards

[14]Inter Process Communication mechanisms

[15]first-in-first-out queue

[16]otherwise this could result in a classical deadlock situation

- **fair:** Every process which wants to access a queue must eventually be allowed to do so.[17]
- **flexible:** There shall be methods to initialize, destroy, read and write data to/from a queue in many customized ways.
- **fast:** Any access to the queue shall be as fast as possible.
- **capable of suspending/waking user processes:** If a process waits for data to arrive on an empty queue and has nothing else to do, it shall not consume CPU time. The same shall be true for waiting for a full queue.

To support routing data by Nathan-ID, there is the possibility to supply multiple queues assigned with each available Nathan. Every data stream to a certain Nathan becomes independent of the other ones. Prevention of interleaving streams to one Nathan from different users is also ensured by the interface functions excluding each other by flipping bits in a shared mask.

3.3.3 Synchronisation Mechanisms

Three processes are assumed some of which, process A and C, want to insert data into a shared buffer and one process which wants to remove data from it simultaneously. In that case, many undesired effects could happen in an environment which lacks the implied mechanism. Process A writes to the same location as process C without knowing about each other; this corrupts both information units, as seen in figure 3.2a. Process B removes data before any valid information has been inserted into the buffer. This would also happen if process B is removing data before a write access has been completed. Preventing those simultaneous accesses in space and/or time is ensured by:

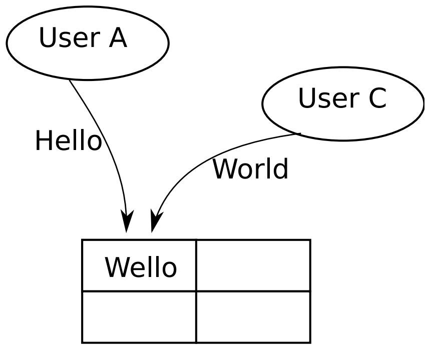
- having two distinct indices to removable and writeable parts in buffer. As mentioned before, a read access consumes an element while a write access produces a new element. So, the indices must not overtake each other.
- designing access operations to be atomic[18] with respect to each other.

Distinct indices solve the issue on reading/writing to the same location while atomicity of the operations itself solves the problems occurring when reading/writing at the same time. Excluding two possibly parallel operations from accessing shared informations is called mutual exclusion (see figures 3.2b and 3.2c). Hence, additional information is needed for process A, B, C to decide whether or not they can read or write data to/from the buffer.

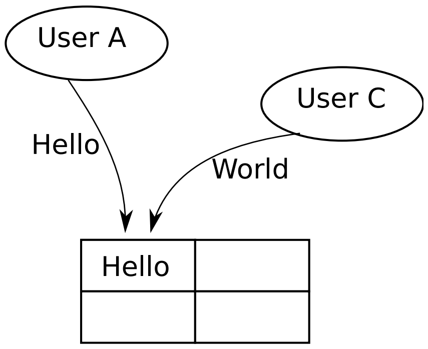
Let N be the total number of independent information units, W an index to the next writeable element and R an index to the next consumable element containing useful information. Read accesses take the value of R, reading the content of the buffer at that position and finally modifying R to point to the next valid element. Write accesses analogously treat the W value. To prevent the possibility of concurrent accesses to that indices they have to be read and modified atomically. Otherwise, process A and C may write to the same location or the increment of W would become inconsistent. This is also true for the data read or written at those locations.

[17]this property is commonly known as starvation freedom

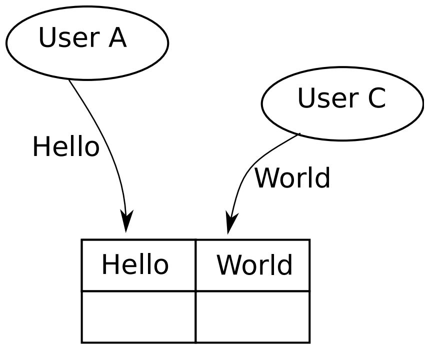
[18]in this context an atomic operation is a code section which can not be executed concurrently with another



(a) without mutual exclusion



(b) with mutual exclusion in time



(c) with mutual exclusion in time and space

Figure 3.2: Effects on a shared bounded buffer in case two user processes attempt to insert data simultaneously. In 3.2a access to the buffer is mutually exclusive, so the information of both processes will be corrupted.In 3.2b a locking mechanism is used to prevent simultaneous accesses to an element. The final content of the element is either from A or from C. In 3.2c the loss of information is prevented if both processes insert their informations at different locations as well.

The Producer-Consumer-Problem Obviously, R must not overtake W and vice versa because nothing must be consumed that has not been inserted before and must not be overwritten in case it has not been consumed before. One possible solution to this so-called producer-consumer-problem [*Herlihy and Shavit*, 2008] is to define R and W to be absolute indices instead of relative ones and to check if R is less than or equal to W. But this approach would need expensive extra care when those indices are about to overflow. When using relative indices to refer to the different types of elements, a check if $R < W$ is not sufficient to determine the state of the elements in the buffer (which is motivated in fig. 3.3). Relative indices operating on a bounded buffer are modular arithmetic integers, so R can become greater than W without validating the prohibition of overtaking. This can be solved by additional shared counters, E the amount of empty elements and F the amount of full elements, to distinguish whether or not R could overrun W and vice versa[19].

Shared Memory Based FIFO Queues The shared memory based FIFO queue operations *push* and *pop* incorporate the mentioned solutions by excluding each other via so-called spin-

[19]one counter would be sufficient though, but for the sake of better understandability two counters were used

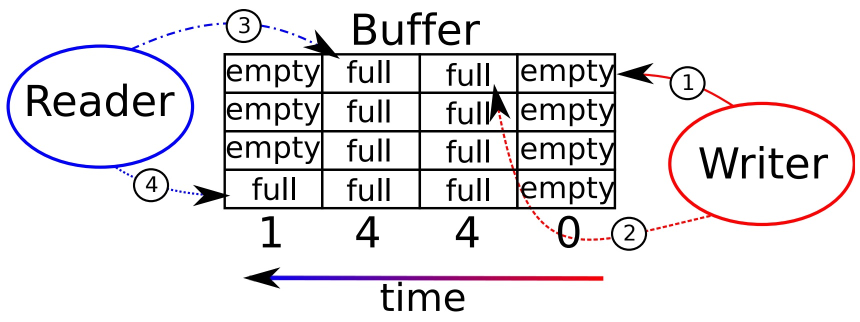


Figure 3.3: The producer-consumer problem with relative indices: The blue arrows denote the relative index R, the red arrow W. Below the indicated buffer states the value of an additional shared counter indicating the number of valid/full elements. At state **1** the writer starts to insert valid data into an empty queue until the queue is full (state **2**). Without extra counters the reading process would have no information whether the first entry is full or not. If there is at least one counter the reader knows the state of all elements. Therefore, it can proceed and consume three valid elements (states **3** and **4**)

locks[20] which protects the indices W and R, the shared counters as well as the buffer entries itself. Forestalling the explanation of the functionality of the queue operations, the presentation of spinlocks and possible implementations of such will be explained later.

```
1: $spin\_lock()$
2: while $E \leq 0$ do
3:     $spin\_unlock()$
4:     $wait\_for\_signal()$                    ▷ uses the FUTEX system call
5:     $spin\_lock()$
6: end while
7: $E \leftarrow E - 1$
8: $Element[W] \leftarrow NewElement$                ▷ Buffer access
9: $W \leftarrow (W + 1) \mod N$
10: $F \leftarrow F + 1$
11: $spin\_unlock()$
12: $dispatch\_signal()$                          ▷ uses the FUTEX system call
```

Algorithm 1: Push operation of shared memory based FIFO queue.

A call to *push* has the following effects: First of all, the spinlock is acquired which protects access to E and F, W and the corresponding buffer location. If successful, E is checked whether it is zero and the user has to wait or not. Waiting due to a full or empty queue is performed by exploiting the FUTEX-syscall[21] available in Linux [*Drepper*, 2009]. A FUTEX-syscall allows code being executed in user space to bypass normal blocking synchronisation mechanisms of the OS which would be much slower. When space for writing becomes available the buffer is accessed, E is decreased by 1 and the entry pointed to by W is overwritten with a new element. Before releasing the lock and returning to the caller, W is updated and a signal to possibly suspended *pop*-calls has to be dispatched. This is done by increasing F by

[20]similarly to quantum particles with spin 1/2 a spinlock has two possible states
[21]Fast User space muTual EXclusion system call

one and by waking possibly suspended processes by another FUTEX-syscall (see algorithm 1).

Besides the normal *push* and *pop* operations there exist two other operations called *try push* and *try pop*. If a check on E or F has revealed that the buffer is currently empty or full, these operations immediately release the lock and return the check result. This makes non-blocking or non-waiting I/O possible so processes could perform other useful work in the meantime.

Simple Spinlock Implementations A spinlock is a shared memory variable which has two states: locked and unlocked [*Herlihy and Shavit*, 2008]. The value of a spinlock is only modified by a single atomic instruction provided by the CPU.

```
1: while $exchange_{atomic}(lock, 0) \neq 1$ do
2:     $wait()$
3: end while
4: <critical code section>                              ▷ $lock \equiv 0$
5: $lock \leftarrow 1$
```

Algorithm 2: Simple spinlock. $1 \equiv unlocked, 0 \equiv locked$

```
Ensure: atomicity
1: $result \leftarrow lock$
2: $lock \leftarrow x$
3: return $result$
```

Algorithm 3: The $exchange_{atomic}(lock, x)$ function.

Once a lock operation is performed, the state of a previously unlocked spinlock changes to the locked state in an atomic fashion. Any other later lock operation waits (*spins*) on the locked state until it changes to the unlocked again. Algorithm 2 shows a possible implementation of a spinlock protecting a critical code section in which shared resources could be accessed simultaneously[22]. The presented implementation however would suffer from great performance loss in reality. If a spinlock is under heavy usage and the locked state becomes highly probable, there would be many atomic exchange operations performed unnecessarily. So, before trying to atomically exchange two values the algorithm should first check whether this exchange would result in leaving the loop (see algorithm 4) [*Herlihy and Shavit*, 2008].

Furthermore, one has to take care when thinking about where a shared variable for locking purposes should be placed in memory [*Drepper*, 2007]. Modern computer architectures posses fast memory called cache to decrease latency when accessing frequently used data. These cache memories are split up in so called cache lines with a certain size. Every time a program currently executed wants to access data in the memory the processor looks in the cache memory first. If the desired data is present and valid in the cache the processor reads from or writes to the corresponding cache line instead of reading/writing from/to the corresponding location in the main memory. A cache controller then frequently updates the main memory, and if more than one cache memory unit exists, all locations in the system referring to the

[22] in reality a so called compiler barrier has to be inserted before changing *lock* to the unlocked state; otherwise compiler optimizations could corrupt protection by instruction reordering

```
1: repeat
2:     while $lock \neq 1$ do
3:         $wait()$
4:     end while
5: until $exchange_{atomic}(lock, 0) = 1$
6: <critical code section>
7: $lock \leftarrow 1$
```

Algorithm 4: Enhanced spinlock.

same location in main memory are kept coherent. Hence, every frequently-used shared object, such as the mentioned spinlock, has to reside in its own cache line (how shared objects are arranged in memory is shown in listing A.3). Otherwise, modifications to data in the same cache line would cause the cache being invalidated and re-read from main memory, which would result in a vast performance decrease of the above spinlock code.

A More Sophisticated Spinlock Implementation Although algorithm 4 is correct, the competition of entering the critical section is not fair. It suffers from a possible starvation of processes which are never allowed to proceed. If fairness is desired and critical sections are not short compared to lock acquiring and releasing, other spinlock designs have to be used e.g. ticket based spinlocks or queue based spinlocks [*Herlihy and Shavit*, 2008].

```
1: $myplace \leftarrow increase_{atomic}(lastplace)$   ▷ Get the place in the queue of waiting processes
2: while $queue[myplace] \neq 1$ do                  ▷ Wait myplace's turn
3:     $wait()$
4: end while
5: <critical code section>
6: $queue[myplace] \leftarrow 0$                     ▷ Re-lock the current place
7: $myplace \leftarrow (myplace + 1) \mod MAX$       ▷ Pass access to the next spinning process
8: $queue[myplace] \leftarrow 1$
```

Algorithm 5: Queue based spinlock.

A queue based spinlock as shown in algorithm 5 has an additional advantage compared to other solutions: There remains no single location on which concurrent spinlock code *spins* on. There is only one counter increased once by all processes to calculate their place in queue. Therefore, the traffic of the cache controllers to ensure cache coherence is reduced and mutual cache line invalidations[23] are avoided. In modern computer architectures these invalidations are avoided by default because a smart cache coherence protocol like MESI[24] is used [*Drepper*, 2007]. In SCtrlTP, only the enhanced spinlock (algorithm 4) is used because it is faster than queue based spinlocks if the critical sections are short. Almost every critical section is tried to be entered by two threads maximum, which means that the starvation probability is very low. For more details of the software implementation of spinlocks, see appendix A.1.

[23]this is known as cache line ping pong
[24]Modified Exclusive Shared Invalid

3.3.4 Performance Optimisation

A big part in software engineering is the optimisation of algorithms and their efficient implementation. In the following, the most important aspects which were considered during the transport layer development are explained.

Multi-threading

As already mentioned before, SCtrlTP is capable of getting data from multiple users in parallel. SCtrlTP should therefore make use of multiple concurrent code blocks to handle that parallel data effectively. The operating system provides so-called *POSIX-threads*-code areas which are executed in parallel [*Drepper and Molnar*, 2005]. Such threads are most useful if they are related to (almost) independent code and the computer system is equipped with more than one processing unit. Nowadays, most personal computers are, in fact, equipped with so-called multi-core processors.

Obviously, a bidirectional protocol implementation, as the developed SCtrlTP, has two nearly independent parts: the part which handles the transmission (TX) and the part which handles the retrieval (RX) of data. The only informations being passed from RX to TX are a) the last valid sequence number received and b) the last acknowledgement number corresponding to TX's sliding window. If RX has recently updated any of those values it sends a wake-up-signal to TX which transmits an acknowledging packet or slides its window forward resulting in new packets being sent. These two parts have been designed in such a way that they can be executed concurrently.

Theoretically, more code sections could be executed in parallel, but there is a trade-off between the overhead which results from context switches and the performance gain from parallelisation. In general, there should be as many threads as there are processing units and each thread should perform as much work as they can without waiting. For example, if an almost independent code section of a thread is too short regarding to execution time such thread will spend most of the time waiting for events to occur.

Zero-Copy Policy

The space which was reserved for the frames to be transmitted or received is passed to those threads via pointers. Performance of software is greatly increased when all possible copy processes are kept at a minimum[25]. In the current solution, data is only copied by the kernel when socket system calls are performed – all other accesses to packet space are carried out by dereferencing the corresponding pointer. If a user process wants to send multiple packets or RX has received more than one valid packets in a row, a further software improvement allows them to collect packet pointers before inserting them in a single queue entry. This reduces blocking probability and the amount of expensive atomic exchange operations per packet when performing those queue operations.

[25]commonly referred to as *zero-copy* policy

Round Trip Time Estimation

An estimation of the mean RTT is calculated by TX if the transmission window has been slid.

$$\begin{aligned} Error_{RTT,i} &= RTT_{measured,i} - RTT_{average,i} && (3.7)\\ RTT_{average,i+1} &\leftarrow RTT_{average,i} + \alpha Error_{RTT,i} && (3.8)\\ e_{RTT,i+1} &\leftarrow e_{RTT,i} + \beta\left(|Error_{RTT,i}| - e_{RTT,i}\right) && (3.9) \end{aligned}$$

First, an error value $Error_{RTT}$ between the current RTT measurement and the predicted average is calculated. After that $RTT_{average}$ is updated by the addition of a fraction α of the error value. This recursive prediction value converges to the real average of RTT after a certain number of updates [*Ljung and Soderstrom*, 1983].

$$\begin{aligned} RTT_{average,i+1} &= (1-\alpha)\, RTT_{average,i} + \alpha RTT_{measured,i} && (3.10)\\ \implies w\left(RTT_{measured,i-n}\right) &= (1-\alpha)^n \alpha && (3.11) \end{aligned}$$

The weighting of a single measurement $w\left(RTT_{measured,i-n}\right)$ gets smaller for every new update of the average estimate. So, new measurements are more conductive to that prediction of the average than previous measurements. Instead of calculating the standard deviation σ of RTT, the mean deviation e is estimated which is an upper bound of σ. Otherwise, an expensive squaring of $(|Error_{RTT}| - \sigma_{RTT})$ would be necessary. The time-out, after which a retransmission of a packet is initiated, is the sum of $RTT_{average}$ and a multiple of e_{RTT}. The probability of unnecessary retransmissions is thereby decreased.

Window Corruption Prevention

Retransmissions are performed by a thread different from the TX thread. If two threads are about to send packets independently, the data flow can be corrupted without any protection. For example, if TX is about to send an acknowledgement and is suspended by the operating system and a retransmission with a recently updated acknowledgement number is performed the window of the recipient will slide forward. The continued transmission of the acknowledgement by TX afterwards will be out-dated and possibly corrupt the window of the recipient. Therefore, every packet transmission is protected by mutual exclusion provided by an additional spinlock.

Acknowledgement Strategy

Currently, the acknowledgement strategy is quite simple: All incoming packets are acknowledged either by sending an acknowledging packet or by back-packing the current acknowledgement number in a normal data packet. This can be optimized though: Instead of transmitting an acknowledging packet instantaneously (assumed that there are no data packets to be transmitted) this transmission can be delayed to keep the fraction of throughput for acknowledging overhead constant. Another strategy could be to transmit acknowledgement packets only if the remote station expects such a packet [*Landström and Larzon*, 2007]. However, this is not supported by the current hardware design.

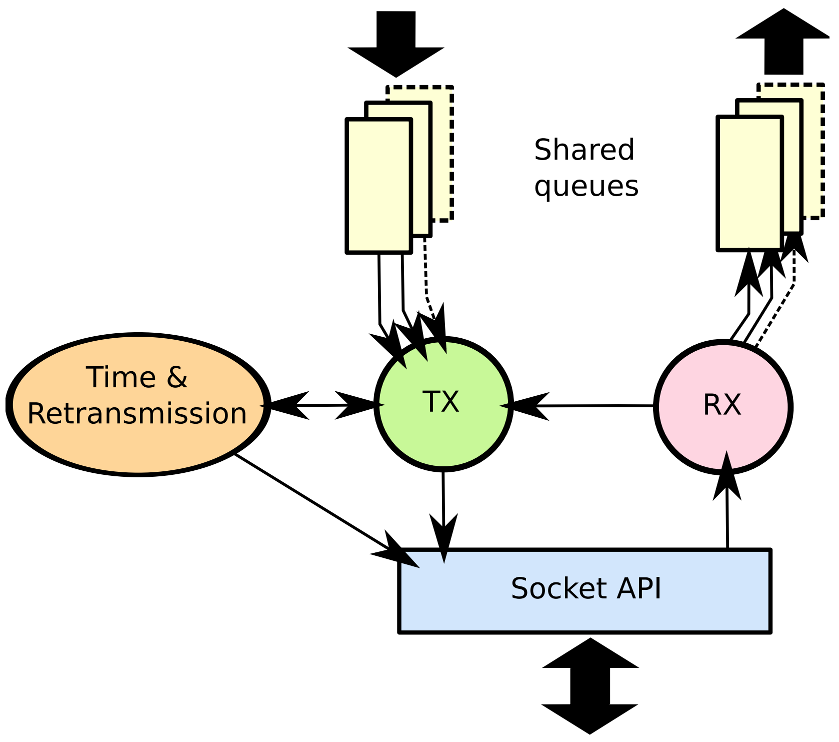


Figure 3.4: Functional structure of the Slow Control Transport Protocol software implementation. The data and signal flow is denoted by the thin arrows. The circles stand for the three concurrent threads. The user processes which insert or remove data from the shared memory based queues are not shown explicitly.

Packet Filtering

If there are other data flows, those frames which are not destined to the flow of interest have to be dropped. This can be achieved in two ways: RX could filter those frames by itself which would be very slow since every frame to be dropped is first copied from kernel space to user space and would consume RX's processing time. On the other hand, there is the possibility to tell the operating system which frames have to be dropped. A so called Berkeley Packet Filter program can be passed to the kernel which allows RX to never receive uninteresting frames [*Mccanne and Jacobson*, 1993]. Such a program which is used by SCtrlTP is shown in listing A.4.

3.3.5 Resulting Protocol Implementation

If a user process wants to send data to the backplane, it first acquires a pointer to an empty frame buffer space. After that, it puts all necessary information in it and places that pointer in a shared FIFO queue. The TX-thread is then woken up and eventually fetches that pointer off the queue and tries to register it in its sliding window buffer, thus assigning it a unique sequence number. If the assignment was successful, the current acknowledgement number and other information (e.g. the Ethernet header) are also assigned before the frame is physically transmitted by the system. On the remote side, the backplane (or another transport protocol instance) receives the frame and processes its content. Meanwhile, a third thread updates a time variable and keeps track of how long an unacknowledged packet that has already been sent was kept.

Time is measured by *sleeping*[26] a certain time (imprecise but maybe the only option on older architectures) or by setting up a HPET[27] and increasing a time-stamp variable accordingly. Any delay is therefore the difference between two timestamps, for example one taken before transmitting the packet and one after receiving an acknowledgement for it. If a packet is unacknowledged after about RTT milliseconds, the same thread assumes that this packet was lost and will retransmit it.

The RX-thread eventually receives one or more packets e.g. one acknowledgement packet and one data packet filled with responses. If the packets fits into the retrieval window it might be slid forward and the pointer to all valid packets received in order will be passed to another shared FIFO queue. Furthermore, if the acknowledgement and/or the data packet have transported a new acknowledgement number this number is passed to TX. All valid data packets received by RX will be acknowledged by TX either by backpacking the last valid sequence number received or by transmitting a corresponding acknowledgement packet.

Figure 3.4 shows the structure of the final protocol implementation. All of the requirements and features discussed in this chapter are incorporated into the software. Almost every feature can be enabled or disabled before compiling for better flexibility. In the next chapter, measurements with this protocol implementation will be presented. Afterwards, in chapter 5, some open issues of the software regarding performance increase will be discussed.

[26]a process inactively waits for a condition to become fulfilled
[27]High Precision Event Timer

4 Measurements

The developed code was tested by establishing and examining communication with both a simulated and a real hardware device. The simulated environment consists of a simple program that operates on local memory and provides the functionality necessary for all applied tests. Read and write commands are supported, hence simple RAM test can be performed. The hardware environment consists of multiple Nathan cards, some of which are equipped with *Spikey* chips (see chapter 2). Therefore, all possible functions that are supported by both the protocol and the hardware device can be and have been utilized and tested.

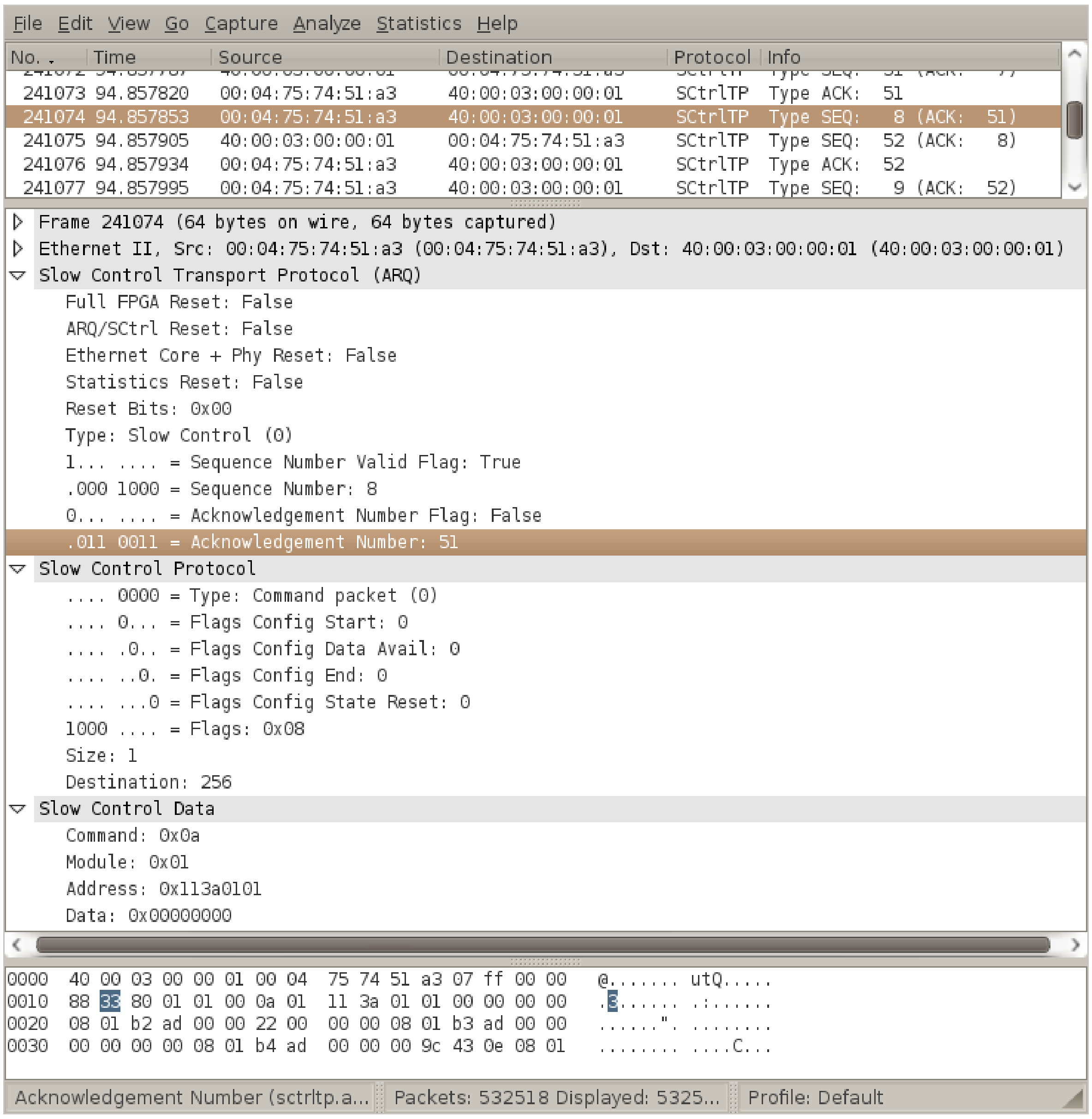
4.1 Software Tools

Prior to the actual measurement results, the software tools which were used shall be briefly introduced. During development, the programs *tcpdump* (command line interface, *Fuentes and Kar* [2005]) and *Wireshark* (graphical user interface, *Orebaugh et al.* [2006]) were utilized to test the basic protocol functionality and to unravel most of the hardware and software flaws. These tools detect both incoming and outgoing packets on a network interface and display their contents. Additionally, *Wireshark* provides a graphical user interface, many protocol dissectors and other functions assisting in protocol analysis. For the ease of debugging, a *Wireshark* plug-in which dissects the Slow Control Transport Protocol was implemented in cooperation with Eric Müller (an exemplary dump can be seen in fig. 4.1).

Tests of the correctness and performance of the underlying synchronization mechanisms discussed in chapter 3 were accomplished with dedicated benchmarking software written by the author. The software *locktest* spawns a customizable number of threads which compete with each other for increasing a shared counter. During the test the consistency of the counter value and the preservation of mutual exclusions are checked. After a predefined number of increases, the threads shut down and the results are presented. Another tool, *fifotest*, measures the throughput of the shared memory based queues as a function of the size of one single element.

Basic functionality of both the hardware device and the simulated software device was tested with already existing software tools. For example, there are tools for reading and writing to various registers or memory locations on different parts of the hardware environment. Especially important is the *dconfig* tool which configures the different Nathan cards to become operational and accessible. Besides that, the *ramtest* [*Philipp*, 2008] tool was heavily utilized which performs a consistency test of the installed memory of a Nathan card. This tool works in both the simulated and the real hardware environment.

Performance measurements were carried out with another system tool. The *ifstat* program reads out the statistics of a certain networking device over time and calculates the corresponding throughput. All throughput measurements were done with this tool. However, this tool dumps the gross throughput only, so every measurement had to be executed in the absence of any other data flow.



```
No. .   Time       Source              Destination         Protocol  Info
241073 94.857820  00:04:75:74:51:a3   40:00:03:00:00:01   SCtrlTP   Type ACK:  51
241074 94.857853  00:04:75:74:51:a3   40:00:03:00:00:01   SCtrlTP   Type SEQ:   8 (ACK:  51)
241075 94.857905  40:00:03:00:00:01   00:04:75:74:51:a3   SCtrlTP   Type SEQ:  52 (ACK:   8)
241076 94.857934  00:04:75:74:51:a3   40:00:03:00:00:01   SCtrlTP   Type ACK:  52
241077 94.857995  00:04:75:74:51:a3   40:00:03:00:00:01   SCtrlTP   Type SEQ:   9 (ACK:  52)
```

```
▷ Frame 241074 (64 bytes on wire, 64 bytes captured)
▷ Ethernet II, Src: 00:04:75:74:51:a3 (00:04:75:74:51:a3), Dst: 40:00:03:00:00:01 (40:00:03:00:00:01)
▽ Slow Control Transport Protocol (ARQ)
    Full FPGA Reset: False
    ARQ/SCtrl Reset: False
    Ethernet Core + Phy Reset: False
    Statistics Reset: False
    Reset Bits: 0x00
    Type: Slow Control (0)
    1... .... = Sequence Number Valid Flag: True
    .000 1000 = Sequence Number: 8
    0... .... = Acknowledgement Number Flag: False
    .011 0011 = Acknowledgement Number: 51
▽ Slow Control Protocol
    .... 0000 = Type: Command packet (0)
    .... 0... = Flags Config Start: 0
    .... .0.. = Flags Config Data Avail: 0
    .... ..0. = Flags Config End: 0
    .... ...0 = Flags Config State Reset: 0
    1000 .... = Flags: 0x08
    Size: 1
    Destination: 256
▽ Slow Control Data
    Command: 0x0a
    Module: 0x01
    Address: 0x113a0101
    Data: 0x00000000
```

```
0000  40 00 03 00 00 01 00 04  75 74 51 a3 07 ff 00 00   @....... utQ.....
0010  88 33 80 01 01 00 0a 01  11 3a 01 01 00 00 00 00   .3...... .:......
0020  08 01 b2 ad 00 00 22 00  00 00 08 01 b3 ad 00 00   ......". ........
0030  00 00 00 00 08 01 b4 ad  00 00 00 9c 43 0e 08 01   ........ ....C...
```

```
Acknowledgement Number (sctrltp.a...   Packets: 532518 Displayed: 5325...   Profile: Default
```

Figure 4.1: *Wireshark* shows a dump of traffic on a link between a host computer and the hardware system. The contents of single packets can be seen in the bottom panel. An overview of the packet flow is displayed in the top panel. For the Slow Control Transport Protocol a special *Wireshark* plug-in was developed in cooperation with Eric Müller which dissects the packets. The dissection is shown in the centre panel.

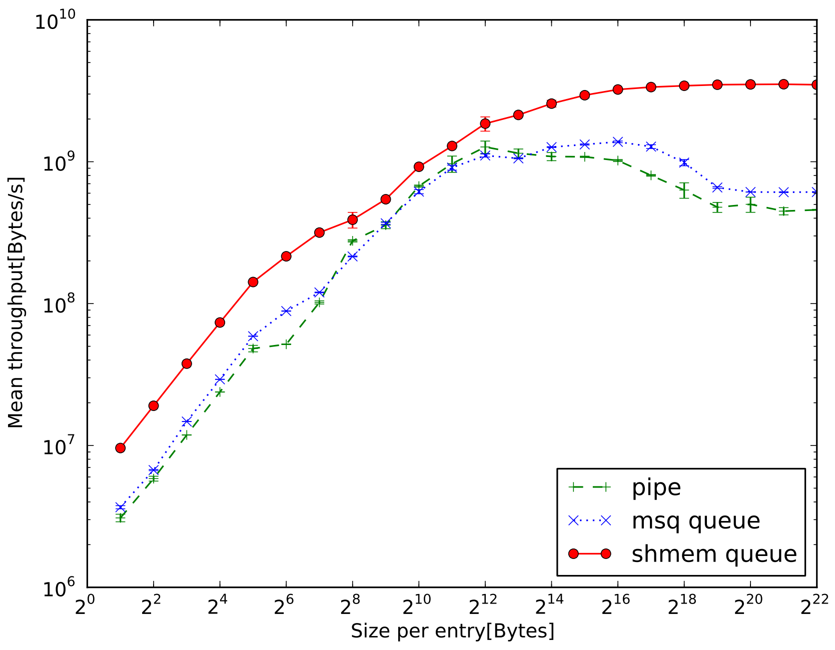


Figure 4.2: For three different IPC-mechanisms, the mean throughput is plotted as a function of the size of one element being sent at a time.

4.2 Measurements in a Simulated Environment

The simulated environment consists of two computers equipped with multi-core processors. A 1 Gbit/s Ethernet link connects both hosts. During the measurements no other data flow was active on that link.

4.2.1 Performance of Shared-Memory-Based FIFO Queues

In section 3.3.2, message-queue-based communication was considered not recommendable because of the amount of system function calls and copy processes between kernel- and user space it would need. Comparing different IPC-mechanisms, figure 4.2 shows the mean throughput over chunks of exchanged data. All measurements were performed with the same premises.

- Every element which is about to be transferred is filled with data in a non-temporal fashion. Non-temporal write accesses are bypassing the internal cache. This prevents the influence of cache effects on the measurement results.
- The element size is increased in the same way. It is doubled before a new measurement is carried out.
- Attention was paid to distribute the producing and the consuming threads over the system's processing units; otherwise, throughput decreases due to expensive context switches.

All data passed to the so-called *named FIFOs* or *pipes* will be copied into kernel space, buffered and copied again to the receiving process afterwards. Multiple writers and readers

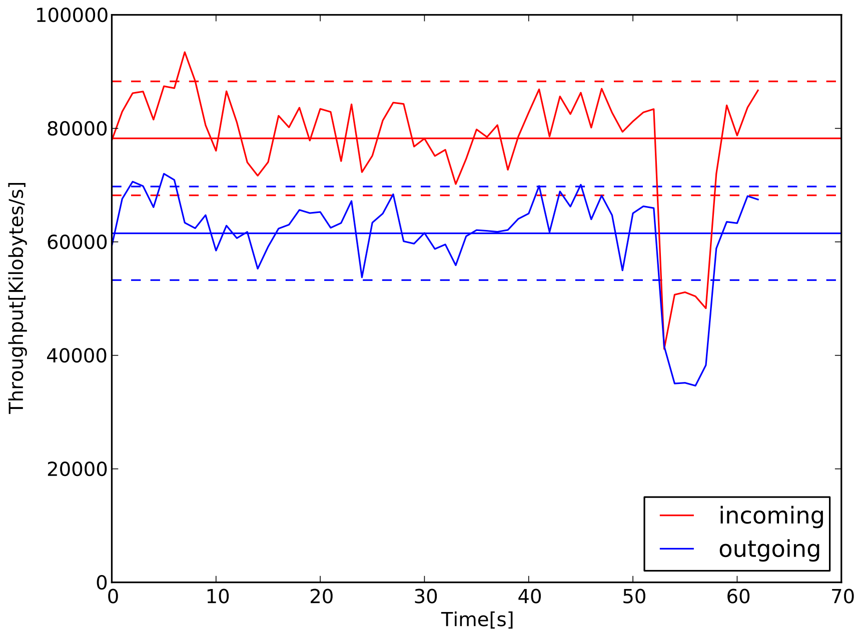


Figure 4.3: Throughput measurement with the simulated device and hardware constraints applied. A similar mean throughput is expected to be achieved in the FACETS stage 1 environment. Among the current hardware constraints are a window size of 32 packets and a maximum payload size of 1010 bytes or 101 slow control commands. The horizontal straight lines denote the mean throughput. The mean throughput for incoming packets is 78 ± 10 MB/s while the mean throughput for transmitted packets is 62 ± 8 MB/s. During the period of time $50 - 60$s the throughput has broken in due to temporarily congested buffers.

have to use separate pipes because accesses to a pipe are not guaranteed to be atomic based on a certain size. Message queue system calls trigger the same amount of context switches and copy processes, but the messages can not interleave. Therefore, the performance of both message queues and pipes is similar, although message queues outperform the pipes slightly in most areas. On the contrary, the performance of shared-memory-based queues is unmatched. Instead of copying the elements, only the pointer to an element is exchanged and context switches between kernel- and user space are reduced to a minimum. In summary, the measurements indicate that shared-memory-based FIFO queues with zero-copy policy have to be utilized to maximize the overall performance of the interface between SCtrlTP and the other software layers.

4.2.2 Throughput of the Slow Control Transport Protocol

Figure 4.3 shows the throughput measured between two hosts. One host was executing a two-threaded program which generated packets filled with commands and received the responses without further processing. On the other host, the previous mentioned simulation program and *ifstat* were running. The communication exhibits therefore an unidirectional behaviour because the return path is not independent from the incoming path. This is visible in the figure: The throughput of outgoing data follows the one of incoming data. Moreover, the

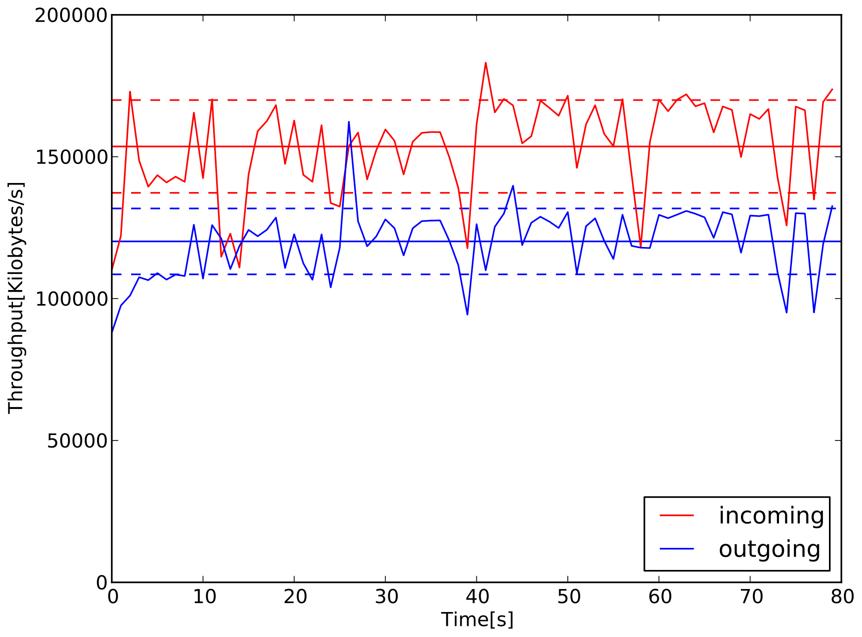


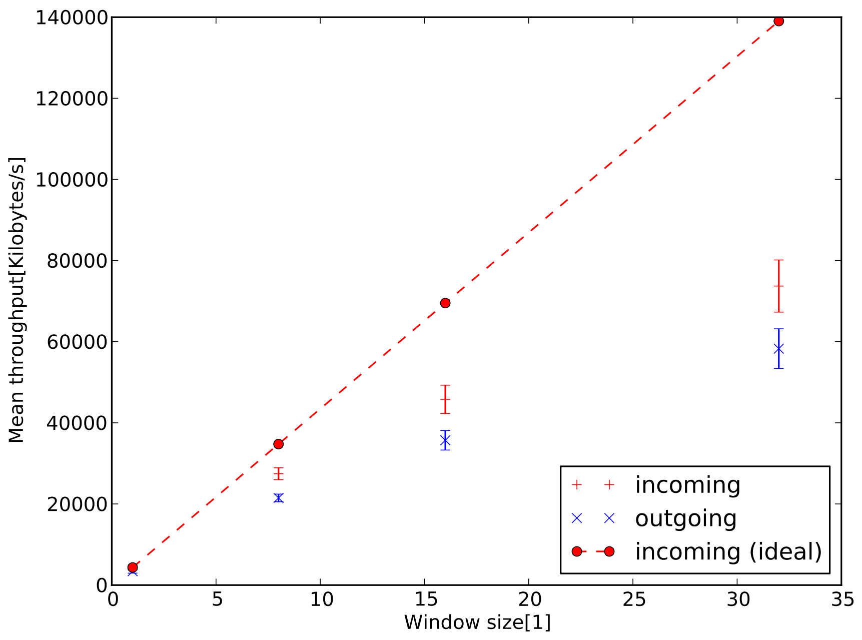
Figure 4.4: This measurement with the simulated device determines the throughput that could be achieved if both window size and payload size were adjusted to higher values than the hardware currently supports. The window size was doubled to 64 packets and the payload size was increased to 1.6 Kbytes or 160 slow control commands. The mean throughput is 154 ± 16 for incoming packets and 120 ± 12 MB/s for outgoing packets.

throughput of the received data is always greater than the other one, since the size of a result entry is less than the size of the corresponding command entry: The size of a result entry is 8 bytes; the size of the corresponding command is 10 bytes. For every additional command the asymmetry increases by 2 bytes per packet.

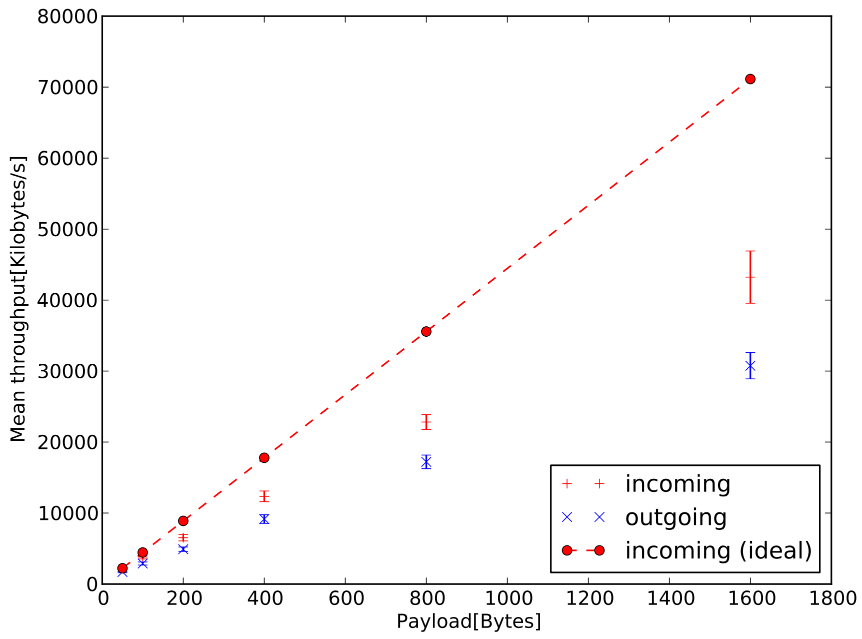
An interesting situation took place after 50 seconds: Congestion of internal buffers. During this period, one or more queues filled up and therefore packets were dropped. Congestion occurs for many reasons. For example, if the operating system preempts[1] one or more threads for a time which is greater or equal to the time it takes for another thread to fill up a queue, this queue could get congested. The probability of congestion can be reduced by activating the congestion avoidance algorithm (see section 3.2.3) and by increasing the queue sizes to filter the latencies of context switches between the threads. During the following measurements, congestion avoidance was activated, resulting in a decrease of overall throughput by the processing overhead.

Figure 4.4 shows the throughput measured with increased window size and payload size of SCtrlTP. Because the throughput becomes too high for a 1 GBit/s Ethernet link a 10 GBit/s link was used to measure the maximum theoretical performance achievable under the new conditions. The mean throughput of incoming packets is ~ 150 MB/s. An 1 GBit/s link is saturated if the throughput reaches the 125 MB/s barrier. Obviously the SCtrlTP is capable

[1]Within the context of operating systems, the term preemption means that a thread can be forced by the kernel of the system to be suspended at any time.



(a) Throughput over window size (payload size: 1010 Bytes)



(b) Throughput over payload size (window size: 8 packets)

Figure 4.5: The most important throughput dependencies are the window size and the payload size. In an ideal system the throughput should be proportional to both variables.

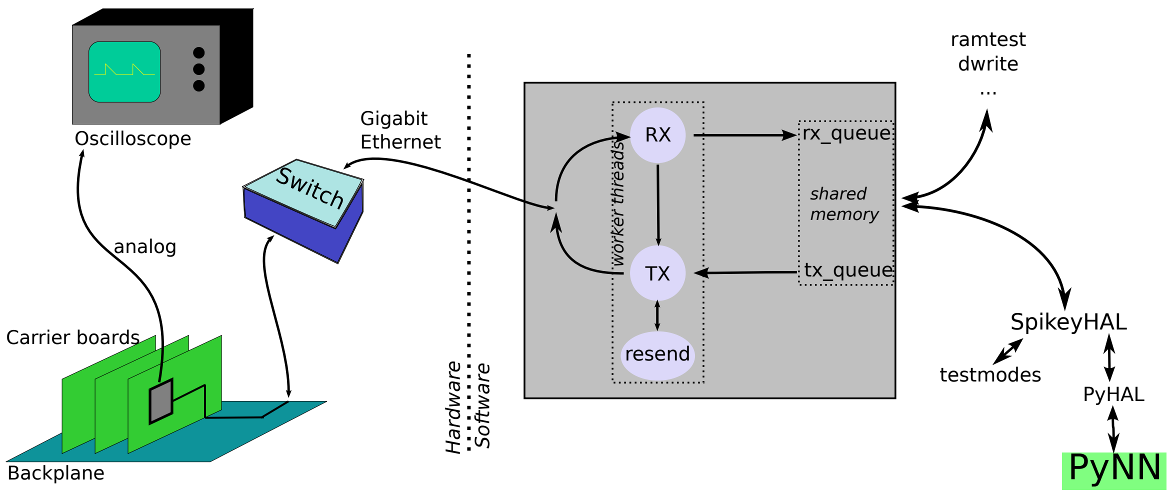


Figure 4.6: Experimental setup including the FACETS stage 1 hardware environment and SCtrlTP software (in the grey box). Other software layers are also indicated, e.g. PyNN.

of saturating an 1 GBit/s Ethernet link if both window size and payload size are increased. Therefore this protocol implementation satisfies the demands which have to be fulfilled to saturate the link between the host PC and the backplane theoretically.

Figures 4.5a and 4.5b show the dependency of the mean throughput on window size and payload size, respectively. The red dashed straight line in both figures represents a linear estimate of the ideal throughput. It is extrapolated from the coordinate system origin and the mean throughput at window size 1 or payload size 50 bytes, respectively. Due to the additional processing overhead and non-deterministic preemption by the kernel, the red curve is never crossed. Nevertheless, the mean throughput is, at a first glance, expected to be proportional to both window size and payload size. While this is almost true for the payload size, it is not the case for the window size. Increasing the payload size of a packet only increases the time it takes for the kernel to copy the packet to and from user space. This has no or just a very small effect on SCtrlTP itself. The increase of the window size, though, has numerous effects:

- The greater the window size gets, the more packets will be transmitted or received per second. This increases the probability of congested queues.
- The more packets per second arrive or have to be transmitted, the larger the processing overhead gets.
- The greater the total execution time for transmitting one window or receiving one window gets, the higher the probability of preemption by the kernel gets, which results in greater latency fluctuations.

Because of these non-deterministic dependencies, the mean throughput is not proportional to the size of the window.

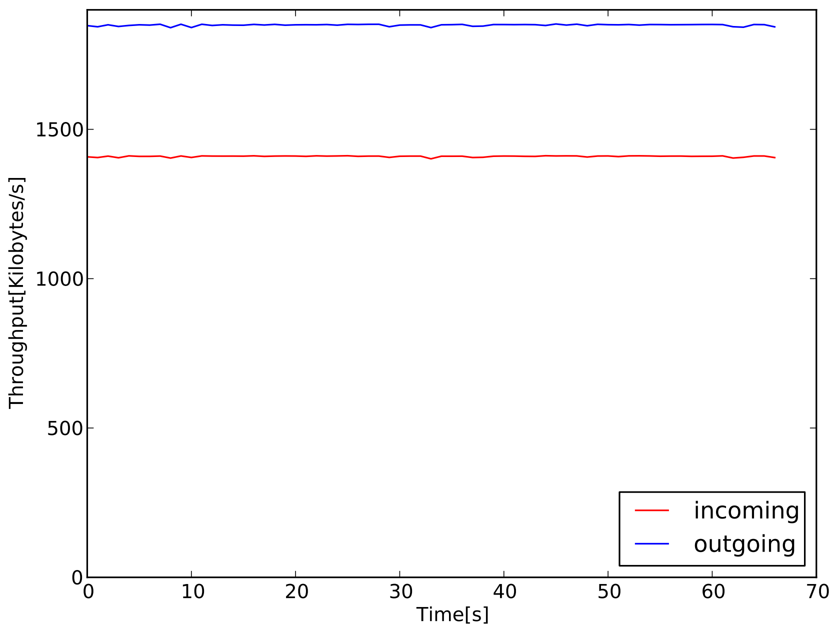


Figure 4.7: Throughput achieved during RAM test with one configured Nathan. The window size was reduced to one packet. A mean throughput of 1409 ± 2 KBytes/s for incoming and 1849 ± 3 KBytes/s for outgoing packages was measured.

4.3 Measurements on the Hardware System

As can be seen in figure 4.6, the hardware device or backplane is connected via a standard Ethernet cable to a switch. All hosts which can be used for communication with the hardware device also have to be connected to that switch. The switch receives incoming packets and decides, depending on the destination address, where those packets have to be directed to.

4.3.1 RAM Tests on the Backplane

As mentioned in section 4.1, the *ramtest* tool is one of the most important tools for testing the main functionality of the hardware device. Initially, the ETH-Core[2] of the backplane is in promiscuous mode, in which it answers any valid packet from any source. When SCtrlTP is started, it sets some important registers of the functional parts of the main FPGA and changes the mode of the ETH-Core to non-promiscuous (for a detailed description of the ETH-Core see *Gutmann* [2007]). Now the backplane can be accessed by the host and communication is possible.

First of all, the Nathan-FPGAs of the installed cards have to be configured. Afterwards, unique identification numbers are assigned to all properly configured Nathans. A RAM test can now be performed on any of the configured Nathans. Figure 4.7 shows the throughput measured during such a RAM test. Although the measured values are not particularly impressive, they remain remarkably stable. The protocol had to be reduced to a Stop-and-Wait

[2]refers to the functional part of the FPGA which is responsible for Ethernet packet handling

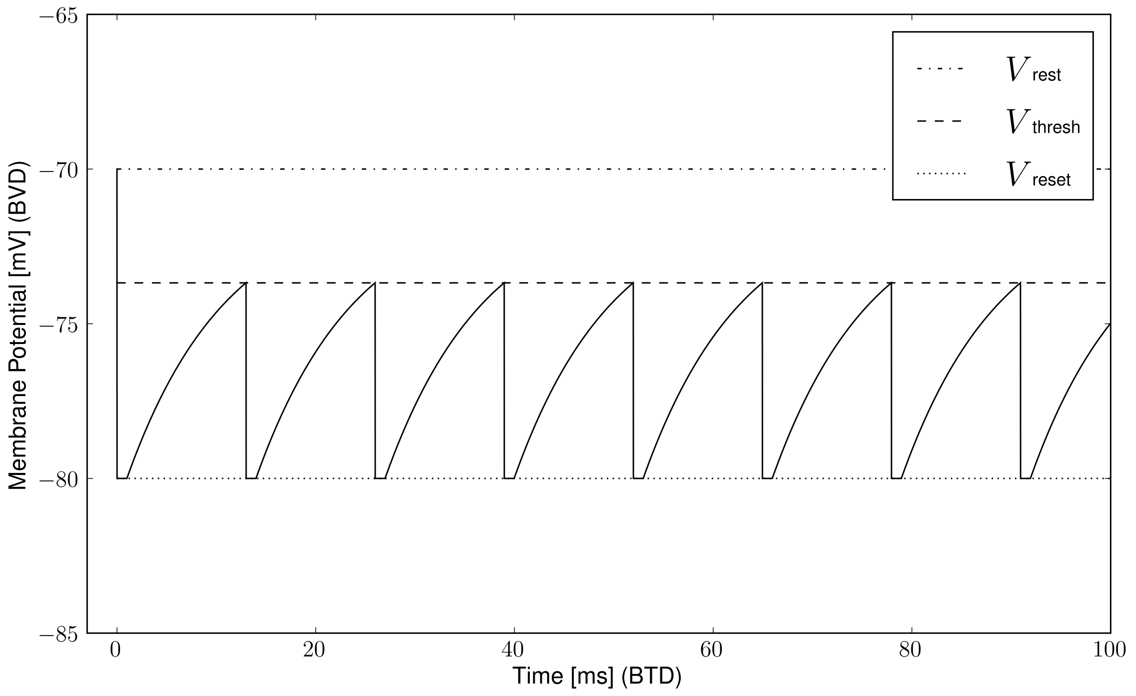


Figure 4.8: Schematic to illustrate the membrane potential time course. The threshold voltage has been lowered below resting potential. This setup is used to rapidly measure the membrane time constant τ_{mem} of a hardware neuron (cf. [*Brüderle*, 2009, p. 90]). The axes are scaled in biological units. With friendly permission of Daniel Brüderle.

protocol with a window size of 1 packet. The bandwidth between the main FPGA and one Nathan card is about 40 − 80 MBit/s or 5 − 10 MB/s respectively. So, more throughput should be possible if larger window sizes could be used. However, this measurement was possible only by means of workarounds for some serious bugs of the hardware device, which will be described in section 4.4.

4.3.2 Neuronal Experiment with PyNN

Before performing a simple neural experiment, the general operability of SCtrlTP and hardware system was tested. SpikeyHAL provides a variety of so-called testmodes, some of which

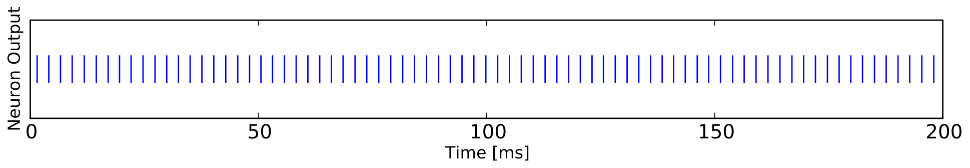


Figure 4.9: Spike train recorded during the membrane time constant calibration routine. As expected, the neuron fired with a constant frequency from which the value of τ_{mem} could be derived. The firing rate of the neuron returned from PyNN is 486.5 ± 17.5 Hz.

check if the prerequisites for a working experiment are fulfilled. For example there is a testmode designated to check the functioning of the connection between a configured Nathan and the Spikey chip, that dispatches a neuronal event from the Nathan to the Spikey, where it is returned to the former. Every testmode necessary for testing the capability of the whole system was successfully accomplished.

Before any experiment can be set up on the neuromorphic device, the properties of the neuron circuits, which differ due to the manufacturing process, have to be determined. As a first step, the membrane time constant τ_{mem} of a hardware neuron can be measured. This time constant is indirectly measurable by setting the spike threshold of the neuron to a special voltage below its resting potential. This results in the neuron firing with a certain frequency from which the value of τ_{mem} can be derived (cf. fig. 4.8). The frequency arises out of a record of the output spikes and their incidence in time. The details of the measurement procedure can be found at page 88ff. in *Brüderle* [2009].

As seen in figure 4.9, this calibration routine was successfully accomplished. The neuron fired with a constant frequency of 486.5 ± 17.5 Hz. A value of 1.06 ± 0.32 ms for the membrane time constant τ_{mem} was determined (τ_{ref}, the absolute refractory period, is 1 ms). Therewith the author showed that neuronal experiments can be successfully done with SCtrlTP as the transport layer embedded in the whole software framework.

4.3.3 Previous Performance

Figure 4.10a shows the performance of the previous data flow via the Darkwing PCI card compared to the data flow via SCtrlTP. Although two different experiments were performed, it is possible to compare the maximum throughput achieved in both experiments. The first experiment was set up such that the link between PyNN and the hardware device was under heavy load. A comparison reveals that the maximum throughput currently achieved with SCtrlTP is about 1.4 times the throughput achieved via the Darkwing link.

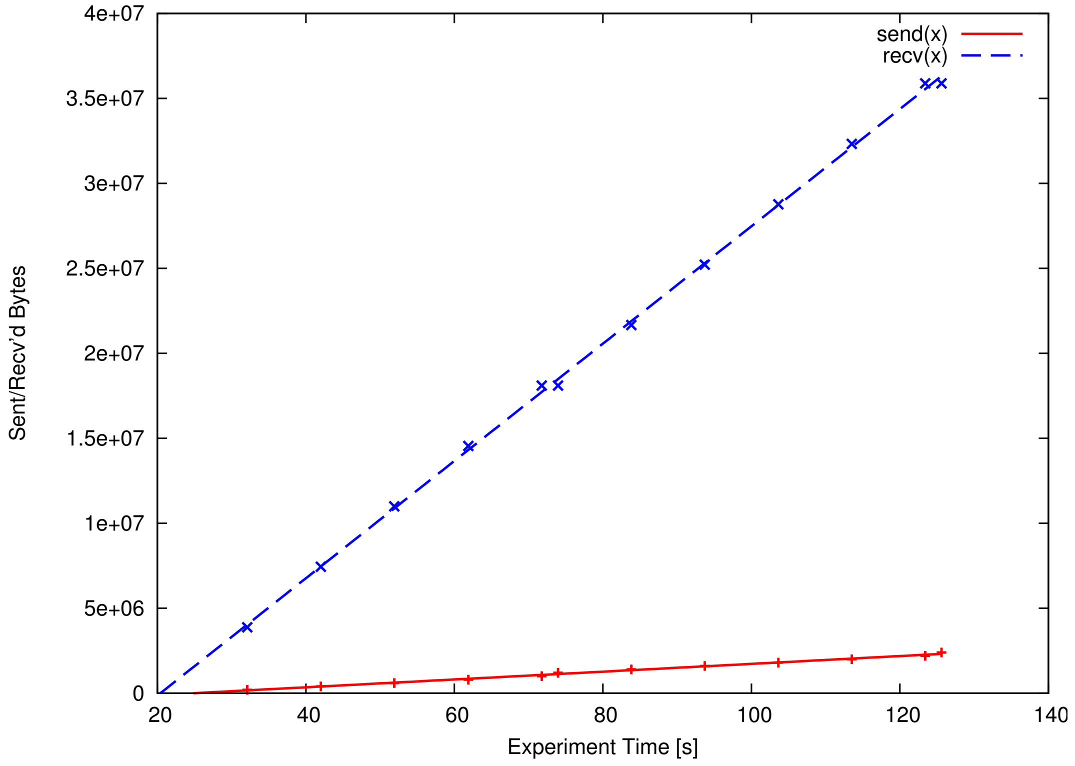
4.4 Known Bugs and Workarounds

During the testing and measurement procedures, many serious bugs of the hardware and software implementations were revealed. Most of the software bugs were related to a lack of protection of concurrently accessed variables. All of these bugs are now fixed and during all of the tests presented in this chapter no corruption of data was encountered.

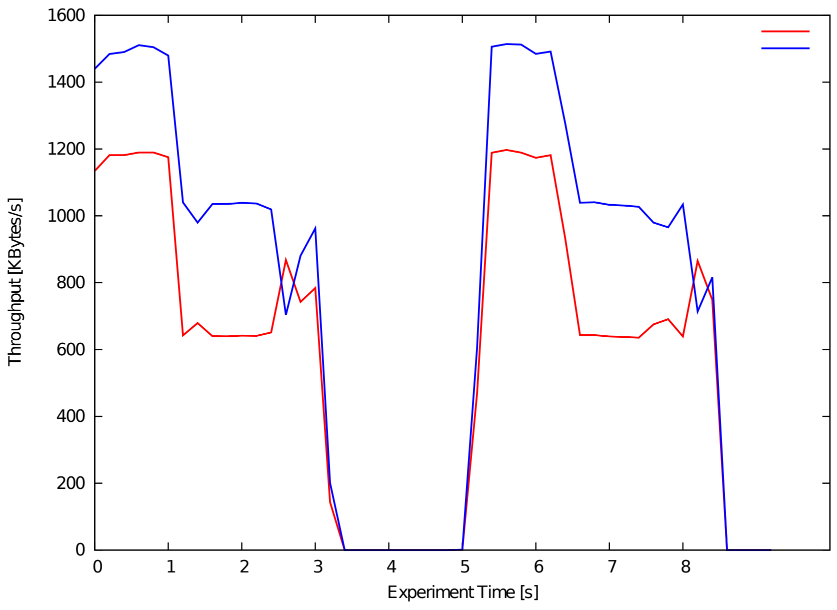
However, there are several remaining bugs in the hardware implementation. If a packet which embodies slow control commands has no trailer of at least 8 zero-bytes the hardware device will return packets containing invalid data in some of the response entries . In software, all packets are currently padded with such a trailer of zero-bytes to prevent the occurrence of this bug. A similar bug, which might be linked to the previous one, occurs if a packet contains more than 200 *double words*[3] of configuration data. In that case, every configuration attempt fails.

The previously mentioned flaws have been worked around by software but there are two major faults which are currently not understood. When performing, for example, a RAM test with a window size larger than 1 and an amount of more than one packet per test, the throughput decreases vastly. Vastly means that throughput suffers in three orders of

[3]a double word is equal to four bytes of data



(a) Measurement of the total number of bytes sent or received by *dwserver* over time using a link over the Darkwing PCI card. The slope of the fitted straight lines determines the mean throughput. The maximum gross throughput for receiving data is 862 KBytes/s.



(b) Throughput measurement during an experiment with PyNN over SCtrlTP. The maximum gross throughput for receiving is $\sim$ 1200 KBytes/s. The down time of $\sim$ 1 second results from software delays between two single experiment runs.

Figure 4.10: Comparison of the throughput measured during two different dummy-experiments with PyNN utilizing the stage 1 hardware device with data flow via the Darkwind PCI card or via SCtrlTP.

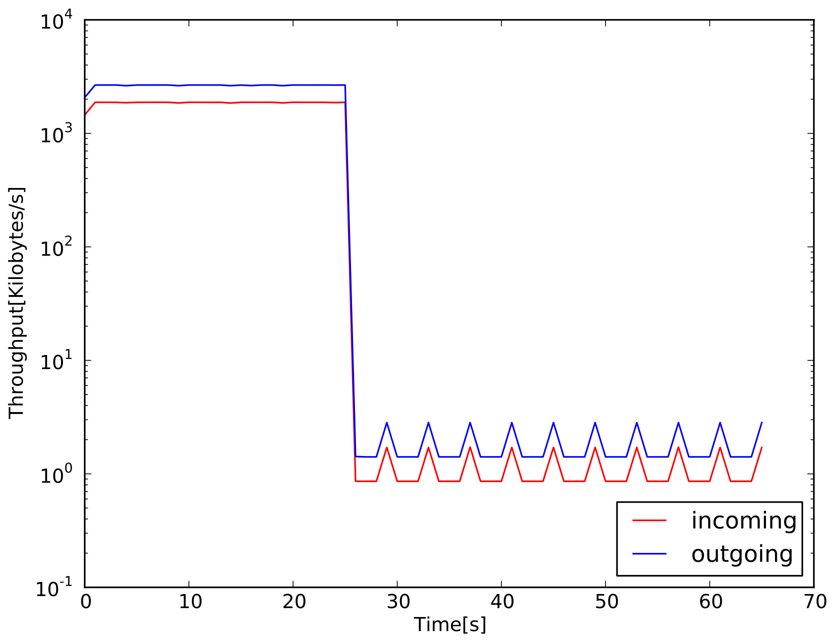


Figure 4.11: Decrease of throughput which occurs if the window size is increased to more than one packet.

magnitude as seen in figure 4.11. The source of that bug is unknown and needs further investigation.

Furthermore, on rare occasions the FPGA resets itself, although no such request was sent. If the FPGA resets itself the connection between host-PC and backplane is lost and the whole procedure has to be restarted. Many of the placed and routed hardware designs were unusable: They simply did not work or they reset themselves very often. The software used to route and place the VHDL[4] design shows many problems when the usage of the slices of the FPGA exceeds a certain threshold. It is also very probable that some of the mentioned malfunctions are related to this major problem.

[4]Very High Speed Integrated Circuit Hardware Description Language

5 Discussion & Outlook

In this diploma thesis the implementation of the new FACETS stage 1 transport protocol was presented. This *Slow Control Transport Protocol* was integrated into the existing software stack. The stack comprises the meta-language *PyNN* developed within the FACETS project which represents a neuronal simulator[1]-agnostic user-interface, the Python-based abstraction layer *PyHAL* and the low level hardware abstraction layer *SpikeyHAL*. These layers form the software stack above the implemented transport protocol.

The main goal was to thereby establish a reliable and efficient communication channel with special attention to kernel space portability and parallel operability. In a high event rate environment like a spiking neuronal network running on an accelerated neuromorphic hardware system fast data transmission is crucial. In the following it is discussed to what extent the developed software and hardware designs fulfil these demands and what is left to be done.

5.1 Achievements

For the first time, the FACETS stage 1 neuromorphic hardware device has become accessible over Gigabit-Ethernet. Compared to the previous connection via the proprietary Darkwing PCI card (cf. chapter 2) this implies an improved usability. Connections can be established with cheap and easy-to-use hardware components of the widely-used *1000Base-T* Ethernet standard. Multiple backplanes can be accessed via multiple instances of the transport protocol handler and thus a single host can be connected to many backplanes.

Measurements based on software simulations have shown that *SCtrlTP* can saturate a 1 Gbit/s Ethernet link if payload size and window size are adjusted to higher values than the hardware device currently supports (cf. chapter 4 and figure 4.3 on page 27).

Measurements between host computer and hardware system revealed that even with the current hardware constraints a better throughput compared to the old Darkwing card-based system can be achieved. The maximum throughput obtained with SCtrlTP is about 1.4 MB/s while the one realized via Darkwing is about 0.9 MB/s. During those and all other measurements, no concurrency flaws or other malfunctions of SCtrlTP were encountered. SCtrlTP conserves the information which has to be exchanged and keeps the connection in a stable regime. However, there remain critical bugs in the hardware FPGA design of which some have been circumvented by changing software constraints and behaviour.

It has been shown, that neuronal experiments described in the top-level meta-language PyNN are still executable. The new transport protocol is fully transparent to the upper software layers and required no further changes to source code in other software layers. As a first test, a calibration routine measuring the membrane time constant τ_{mem} of a single hardware neuron was executed.

[1]and emulator

Provided that the stability and functionality of the FPGA source code will be improved in the future, a significant decrease of the overall experiment's runtime can be achieved.

Still there are many obstacles with respect to the user-friendliness, performance and flawlessness of the whole system which have to be overcome. In particular, a simplification of the software stack used in the operation of the hardware system, e.g. SpikeyHAL, will contribute to an increased acceptance of neuromorphic hardware systems within the neuroscientific community.

5.2 Open Issues

The elimination of the current hardware FPGA design flaws (a description of them is given in section 4.4) is of utmost importance. In particular, the problems with the place & route algorithm which seem to be responsible for most of the detected malfunctions have to be solved. The performance of SCtrlTP measured in software stands and falls with the correctness of the FPGA source code. It would be possible to strip down the FPGA source code of the ETH-Core since most of its functionality is obsolete. The only essential parts are controlling the physical link layer, the calculation of the checksum and the processing of the Ethernet header. Major components that implement memory mapped I/O and adapt the rate of generated interrupts are not used. These parts consume a large fraction of the ETH-Core and are of no avail for the downstream protocol handler.

For example, the bandwidth between the backplane FPGA and the Nathan FPGA could be increased by implementing double data rate [*Hennessy and Patterson*, 2007] transfers. Currently, all Nathans are arranged in a single chain similar to a Token-Ring: commands to one specific Nathan are passed through the ring until they arrive at the targeted Nathan. It would be desirable for independent Nathans to be accessible in parallel without reducing the bandwidth per Nathan.

In order to further improve the usability, more effort has to be put into the whole software framework. This includes SpikeyHAL and SCtrlTP (as described in chapter 2). Especially SpikeyHAL suffers from insufficiencies. For example, SpikeyHAL is currently unusable on 64-bit architectures. Data corruption occurs if it is executed on 64-bit machines. The interface between SpikeyHAL and SCtrlTP has to be redefined to fully support an explicit mapping of multiple backplanes and Nathans. It is also desirable for all SCtrlTP instances, the dedicated backplanes and Nathans to be automatically detected, configured and mapped.

Although SCtrlTP achieves a quite impressive performance in pure software to software measurements there are some possibilities to further increase its performance. It is conceivable to remove the remaining system calls to the socket functions by exchanging data with the kernel via shared memory. There are mechanisms provided by recent Linux kernels, namely PACKET_RX_RING and PACKET_TX_RING [*Baudy*, 2009], which can be used to achieve a zero-copy policy which includes the kernel. A preliminary implementation on the receiving side that utilizes PACKET_RX_RING to receive data has already been integrated into the stack. The lack of the PACKET_TX_RING in previous kernel versions has prevented its exploitation during the work of this diploma thesis. Every packet which shall be sent or received passes through a memory region which is shared between kernel space and user space. Data generated by the control software gets written to the shared memory region where the kernel waits for data to be sent. On the way back, the kernel writes to the shared memory region and sends an event to the user space handler. Thus, redundant system calls can be avoided, too.

However, if SCtrlTP or a modified variant is intended to be utilized for the FACETS stage 2 hardware environment, some part of the software stack have to be ported to kernel space. If 20 Gbit/s of data arrive at the networking device, the correct and rapid handling of that data is absolutely necessary. Additionally, high latency acknowledge handling requires large buffer sizes in the hardware system. Last but not least, a minimised delay caused by communication becomes important in interactive and closed-loop operation modes (cf. chapter 1).

A Source Code

A.1 Locking Mechanisms

```
___s32 xchg (volatile ___s32 *variable, ___s32 new_value) {
        ___s32 old;

        /* Inline assembler instruction xchg
           xchg [src],[dest]
           [src] is the register ax
           [dest] is a location in memory */
        ___asm___ ___volatile___
        ("xchgl␣%1,␣%2;",
        :"=a"(old)
        :"a"(new_value), "m"(*variable)
        :"memory"                          );

        /*Return the previous value of variable*/
        return old;
}
```

Listing A.1: Atomic function used by spinlock code

```
void spin_lock (volatile ___s32 *lock) {

        /*Prevent double locking*/

        /*Try to acquire lock atomically*/
        do {
                /*Wait until lock seems to be free*/
                while (*lock == 0);
        } while (xchg(lock, 0) == 0);

        /*We have acquired the lock and can enter a critical section*/
        /*Register me as the owner of the lock*/
}

___s32 spin_try_lock (volatile ___s32 *lock) {
        ___s32 old;

        /*Prevent double locking*/

        /*Try to acquire lock once and return previous value*/
        old = xchg(lock, 0);

        /*Check if successful and register me as new owner of the lock*/
        return old;
}

void spin_unlock (volatile ___s32 *lock) {

        /*Check if i am the owner of the lock; if not return*/

        /*Prevent instruction reordering above critical section*/
        compiler_barrier();

        /*Release the lock*/
        *lock = 1;
}
```

Listing A.2: Spinlock code which is used by SCtrlTP (some parts are not shown)

```
struct sctp_fifo {
        /*0-63*/
        /*Includes protecting spinlock and counter value*/
        struct semaphore nr_full;

        /*64-127*/
        /*Offset to element, which was last consumed*/
        ___u32 last_out;
        /*Padding to seperate last_out, last_in from same cacheline*/
        ___u8  pad0[L1D_CLS-4];

        /*128-191*/
        /*Offset to element, which was last produced*/
        ___u32 last_in;
        ___u8  pad1[L1D_CLS-4];

        /*192-255*/
        /*Number of total elements*/
        ___u32 nr_elem;
        /*Size of one element*/
        ___u32 elem_size;
        ___u8  pad2[L1D_CLS-8];

        /*256-4096*/
        /*Pointer to buffer of elements*/
        ___u8  *buf;
        /*Keep page size alignment*/
        ___u8  pad4[PAGE_SIZE-4*L1D_CLS-PTR_SIZE];
} ___attribute___ ((packed));
```

Listing A.3: Shared memory based FIFO structure with cache line size padding and page size alignment.

A.2 Berkeley Packet Filter

```
...
struct bpf_program filter;
struct bpf_insn prog[] = {
        /*Check on protocol number*/
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 11),
        /*Check if my MAC is the destination address*/
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 0),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 9),
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 4),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 7),
        /*Check remote MAC*/
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 6),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 5),
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 10),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 3),
        /*Check on protocol type (should be equal to zero)*/
        BPF_STMT(BPF_LD+BPF_B+BPF_ABS, 15),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SCTP_TYP_DEFAULT, 0, 1),
        /*accept packet by returning -1*/
        BPF_STMT(BPF_RET+BPF_K, -1),
        /*ignore/drop packet by returning 0*/
        BPF_STMT(BPF_RET+BPF_K, 0),
};
/*Register instructions in program structure*/
filter.bf_insns = prog;
/*Insert arguments to set up filter properly*/
prog[1].k = proto;

prog[3].k =
txmac[0]<<24 + txmac[1]<<16 + txmac[2]<<8 +txmac[3];

prog[5].k = txmac[4]<<8 + txmac[5];

prog[7].k =
rxmac[0]<<24 + rxmac[1]<<16 + rxmac[2]<<8 + rxmac[3];

prog[9].k = rxmac[4]<<8 + rxmac[5];
/*Calculate number of instructions in prog*/
filter.bf_len = sizeof(prog) / sizeof(struct bpf_insn);
...
```

Listing A.4: The Berkeley Packet Filter program for SCtrlTP.

A.3 Payload Formats

Byte offset	Size	Name	Description
0	1	SC_COMMAND	Identifier of the Slow Control command
1	1	MODULE	Number of the module to access
2	4	ADDRESS	Address of a certain location in the module
6	4	VALUE	A value to be stored (e.g. for WRITE commands)

Table A.1: Format of a Slow Control command transmitted to the backplane.

Byte offset	Size	Name	Description
0	4	STATUS	Returned status value of the executed Slow Control command
			1: Command was executed successfully
			2: An error occurred during execution
4	4	VALUE	A returned value (e.g. for READ commands)

Table A.2: Format of a response of the backplane to a Slow Control command.

Byte offset	Size	Name	Description
0	4	CFG_DWORD	A double word of configuration data

Table A.3: Format of a configuration entry. Such payload is only acknowledged.

Bibliography

Baudy, J., Linux Kernel 2.6.31 – net: TX_RING and packet mmap, Linux Kernel 2.6.31 – commit 69e3c75f4d541a6eb151b3ef91f34033cb3ad6e1, 2009.

Bill, J., Self-stabilizing network architectures on a neuromorphic hardware system, Diploma thesis (English), University of Heidelberg, HD-KIP-08-44, 2008.

Braden, R. T., RFC 1122: Requirements for Internet hosts — communication layers, 1989.

Brette, R., and W. Gerstner, Adaptive exponential integrate-and-fire model as an effective description of neuronal activity, *J. Neurophysiol.*, *94*, 3637 – 3642, doi:NA, 2005.

Brüderle, D., Neuroscientific modeling with a mixed-signal vlsi hardware system, Ph.D. thesis, 2009.

Brüderle, D., E. Müller, A. Davison, E. Muller, J. Schemmel, and K. Meier, Establishing a novel modeling tool: A python-based interface for a neuromorphic hardware system, *Front. Neuroinform.*, *3*(17), 2009.

Corbet, J., A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed., O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2005.

Davison, A. P., D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, PyNN: a common interface for neuronal network simulators, *Front. Neuroinform.*, *2*(11), 2008.

Destexhe, A., D. Contreras, and M. Steriade, Mechanisms underlying the synchronizing action of corticothalamic feedback through inhibition of thalamic relay cells, *Journal of Neurophysiology*, *79*, 999–1016, 1998.

Diesmann, M., and M.-O. Gewaltig, NEST: An environment for neural systems simulations, in *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, *GWDG-Bericht*, vol. 58, edited by T. Plesser and V. Macho, pp. 43–70, Ges. für Wiss. Datenverarbeitung, Göttingen, 2002.

Drepper, U., What Every Programmer Should Know About Memory, Red Hat, Inc., 2007.

Drepper, U., Futexes Are Tricky, Red Hat, Inc., 2009.

Drepper, U., and I. Molnar, The Native POSIX Thread Library for Linux, Red Hat, Inc., 2005.

Ehrlich, M., C. Mayr, H. Eisenreich, S. Henker, A. Srowig, A. Grübl, J. Schemmel, and R. Schüffny, Wafer-scale VLSI implementations of pulse coupled neural networks, in *Proceedings of the International Conference on Sensors, Circuits and Instrumentation Systems (SSD-07)*, 2007.

Enck, J., Ethernet/802.3 and token ring/802.5, pp. 265–295, 1994.

Eppler, J. M., M. Helias, E. Muller, M. Diesmann, and M.-O. Gewaltig, PyNEST: a convenient interface to the NEST simulator, *Front. Neuroinform.*, *2*(12), 2008.

FACETS, Fast Analog Computing with Emergent Transient States – project website, `http://www.facets-project.org`, 2009.

Fairhurst, G., RFC 3366: Advice to link designers on link Automatic Repeat reQuest (ARQ), 2002.

Fieres, J., A. Grübl, S. Philipp, K. Meier, J. Schemmel, and F. Schürmann, A platform for parallel operation of VLSI neural networks, in *Proc. of the 2004 Brain Inspired Cognitive Systems Conference (BICS2004)*, University of Stirling, Scotland, UK, 2004.

Fuentes, F., and D. C. Kar, Ethereal vs. tcpdump: a comparative study on packet sniffing tools for educational purpose, *J. Comput. Small Coll.*, *20*(4), 169–176, 2005.

Gewaltig, M.-O., and M. Diesmann, Nest (neural simulation tool), *Scholarpedia*, *2*(4), 1430, 2007a.

Gewaltig, M.-O., and M. Diesmann, NEST (NEural Simulation Tool), *Scholarpedia*, *2*(4), 1430, 2007b.

GPL 2009, GNU General Public License 2.0, `http://www.gnu.org/licenses/gpl-2.0.html`.

Grübl, A., VLSI implementation of a spiking neural network, Ph.D. thesis, Ruprecht-Karls-University, Heidelberg, document No. HD-KIP 07-10, 2007.

Gutmann, C., Implementation einer Gigabit-Ethernet-Schnittstelle zum Betrieb eines Künstlichen Neuronalen Netzwerkes, Diploma thesis (German), University of Heidelberg, HD-KIP-07-08, 2007.

Häfliger, P., Adaptive WTA with an analog VLSI neuromorphic learning chip, *IEEE Transactions on Neural Networks*, *18*(2), 551–72, 2007.

Hennessy, J. L., and D. A. Patterson, *Computer architecture: a quantitative approach*, Morgan Kaufmann, Amsterdam, 2007.

Herlihy, M., and N. Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

Hines, M., and N. Carnevale, *The NEURON simulation environment.*, pp. 769–773, M.A. Arbib, 2003.

Hines, M. L., and N. T. Carnevale, *The NEURON Book*, Cambridge University Press, Cambridge, UK, 2006.

Hines, M. L., A. P. Davison, and E. Muller, NEURON and Python, *Front. Neuroinform.*, 2009.

Hodgkin, A. L., and A. F. Huxley, A quantitative description of membrane current and its application to conduction and excitation in nerve., *J Physiol*, *117*(4), 500–544, 1952.

IEEE, Standard for information technology - portable operating system interface (POSIX). shell and utilities, *Tech. rep.*, IEEE, 2004.

Jacobson, V., and M. J. Karels, Congestion avoidance and control, 1988.

Johansson, C., and A. Lansner, Towards cortex sized artificial neural systems., *Neural Networks*, *20*(1), 48–61, 2007.

Kaplan, B., Self-organization experiments for a neuromorphic hardware device, Diploma thesis (English), University of Heidelberg, HD-KIP-08-42, 2008.

Landström, S., and L.-A. Larzon, Reducing the tcp acknowledgment frequency, *SIGCOMM Comput. Commun. Rev.*, *37*(3), 5–16, 2007.

Ljung, L., and T. Soderstrom, *Theory and Practice of Recursive Identification (Signal Processing, Optimization, and Control)*, The MIT Press, 1983.

Markram, H., Y. Wang, and M. Tsodyks, Differential signaling via the same axon of neocortical pyramidal neurons., *Proceedings of the National Academy of Sciences of the United States of America*, *95*(9), 5323–5328, 1998.

Mccanne, S., and V. Jacobson, The bsd packet filter: A new architecture for user-level packet capture, 1993.

Mead, C. A., *Analog VLSI and Neural Systems*, Addison Wesley, Reading, MA, 1989.

Mead, C. A., and M. A. Mahowald, A silicon model of early visual processing, *Neural Networks*, *1*(1), 91–97, 1988.

Merolla, P. A., and K. Boahen, Dynamic computation in a recurrent network of heterogeneous silicon neurons, in *Proceedings of the 2006 IEEE International Symposium on Circuits and Systems (ISCAS 2006)*, 2006.

Müller, E., Operation of an imperfect neuromorphic hardware device, Diploma thesis (English), University of Heidelberg, HD-KIP-08-43, 2008.

Nagle, J., RFC 896: Congestion control in IP/TCP internetworks, 1984.

Orebaugh, A., G. Ramirez, J. Burke, and L. Pesce, *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*, Syngress Publishing, 2006.

Pakkenberg, B., and H. Gundersen, Neocortical Neuron Number in Humans: Effect of Age and Sex, *J Comp Neurol.*, *384*(2), 312–320, 1997.

Peterson, L. L., and B. S. Davie, *Computer Networks: A Systems Approach, 3rd Edition*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

Philipp, S., Design and implementation of a multi-class network architecture for hardware neural networks, Ph.D. thesis, Ruprecht-Karls Universität Heidelberg, 2008.

Philipp, S., A. Grübl, K. Meier, and J. Schemmel, Interconnecting VLSI spiking neural networks using isochronous connections, in *Proceedings of the 9th International Work-Conference on Artificial Neural Networks (IWANN'2007)*, vol. LNCS 4507, pp. 471–478, Springer Verlag, 2007.

Renaud, S., J. Tomas, Y. Bornat, A. Daouzli, and S. Saighi, Neuromimetic ICs with analog cores: an alternative for simulating spiking neural networks, in *Proceedings of the 2007 IEEE Symposium on Circuits and Systems (ISCAS2007)*, 2007.

Rossum, G. V., *Python Reference Manual: February 19, 1999, Release 1.5.2*, iUniverse, Incorporated, 2000.

Schemmel, J., A. Grübl, K. Meier, and E. Muller, Implementing synaptic plasticity in a VLSI spiking neural network model, in *Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN'06)*, IEEE Press, 2006.

Schemmel, J., D. Brüderle, K. Meier, and B. Ostendorf, Modeling synaptic plasticity within networks of highly accelerated I&F neurons, in *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS'07)*, IEEE Press, 2007.

Schemmel, J., J. Fieres, and K. Meier, Wafer-scale integration of analog neural networks, in *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*, 2008.

Schürmann, F., S. Hohmann, J. Schemmel, and K. Meier, Towards an Artificial Neural Network Framework, in *Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware*, edited by A. Stoica, J. Lohn, R. Katz, D. Keymeulen, and R. Zebulum, pp. 266–273, IEEE Computer Society, 2002.

Serrano-Gotarredona, R., et al., AER building blocks for multi-layer multi-chip neuromorphic vision systems, in *Advances in Neural Information Processing Systems 18*, edited by Y. Weiss, B. Schölkopf, and J. Platt, pp. 1217–1224, MIT Press, Cambridge, MA, 2006.

Stevens, W. R., B. Fenner, and A. M. Rudoff, *UNIX Network Programming, Vol. 1*, Pearson Education, 2003.

Vogelstein, R. J., U. Mallik, J. T. Vogelstein, and G. Cauwenberghs, Dynamically reconfigurable silicon array of spiking neuron with conductance-based synapses, *IEEE Transactions on Neural Networks*, *18*, 253–265, 2007.

Acknowledgments (Danksagungen)

Ich möchte mich bei allen bedanken, die zum Gelingen dieser Arbeit beigetragen haben. Dies sind insbesondere:

Herrn Prof. Dr. Karlheinz Meier für die freundliche Aufnahme in die Arbeitsgruppe.

Dem Zweitkorrektor meiner Arbeit.

Herrn Dr. Johannes Schemmel, der immer ein offenes Ohr für die vielen Fragen seines Diplomanden hatte.

Herrn Dr. Stephan Philipp für die gute Zusammenarbeit und Kommunikation. Er ist maßgeblich am Erfolg dieser Arbeit beteiligt.

Meinem Betreuer Eric Müller für die sehr fundierte und schnelle Hilfe in allen Fragen zur Softwareentwicklung.

Herrn Dr. Daniel Brüderle und Herrn Mihai Petrovici für die hilfreichen Einblicke in die Neurowissenschaften und die Hilfe bei der sprachlichen Korrektur dieser Arbeit.

Dres. Grübl, Millner für die Antworten auf meine vielen Fragen zur Hardware.

Allen Visionären für ihre außerordentliche Hilfsbereitschaft und die stets angenehme und freundschaftliche Arbeitsatmosphäre.

Vim

Dem M. C. für zackige Mittagessen.

Des weiteren möchte ich mich in besonderem Maße bei folgenden Personen bedanken, die mich während meiner Diplomarbeit außerordentlich unterstützt haben:

Meiner Freundin Gabi, ohne deren Liebe und Halt viele schwierige Stunden kaum zu meistern gewesen wären.
Meinen Eltern für ihre Hilfe bei der Überwindung aller Hürden während des Studiums.