Department of Physics and Astronomy

Ruprecht-Karls-Universität Heidelberg

Bachelor's Thesis

in Physics

submitted by

Lukas Pilz

born in Berlin, Germany

Towards Fast Iterative Learning On The BrainScaleS Neuromorphic Hardware System

This Bachelor's Thesis has been carried out by Lukas Pilz at the

Kirchhoff Institute for Physics

Ruprecht-Karls-Universität Heidelberg

under the supervision of

Prof. Dr. Karlheinz Meier

Towards Fast Iterative Learning On The BrainScaleS Neuromorphic Hardware System

To fully exploit the accelerated operation of the BrainScaleS neuromorphic hardware system, fast configuration, reconfiguration and read-out of experiments is crucial. In particular, many offline learning experiments require frequent modifications of the neuronal network topology and its input. This thesis presents the implementation and integration of a communication module using the FPGA's Playback Memory in the custom software stack supporting the BrainScaleS system, which enables a fast hardware reconfiguration. The full reconfiguration of all synaptic weights now takes 1.4 µs per synapse row, a $\mathcal{O}(10^3)$ improvement compared to the previous busy-wait communication scheme. The correct functionality of the communication module was verified using low-level hardware tests. The executed benchmarks show a significant increase in performance at larger configuration sizes compared to the currently used asynchronous communication. Measurements of parameters required for correct Playback Memory operation have been executed and are discussed. Further suggestions for improvements of the communication module and for future measurements are provided.

Schritte hin zu schnellem iterativen Lernen auf der neuromorphen BrainScaleS Hardware

Um den beschleunigten Betrieb des neuromorphen BrainScaleS Systems vollständig ausnutzen zu können ist eine schnelle Konfiguration, Rekonfiguration und Datenauslese des Experiments essentiell. Insbesonderte benötigen viele Offline-Lernexperimente häufige Änderungen des neuronalen Netzwerks und seines Inputs. Diese Arbeit präsentiert die Implementierung und Integration eines Kommunikationsmoduls, welches das Playback Memory des FPGAs benutzt, in die existierende Software um das BrainScaleS System. Hiermit wird eine schnelle Rekonfiguration der Hardware ermöglicht. Eine komplette Rekonfiguration aller synaptischen Gewichte braucht nun 1.4 µs pro Synapsenzeile, was die Zeit des bisherigen Kommunikationsschemas um $\mathcal{O}(10^3)$ verbessert. Die korrekte Funktionalität des Kommunikationsmoduls wurde mit hardware-nahen Tests überprüft. Mittels eines Benchmarks konnte, insbesondere für große Konfigurationen, eine signifikante Leistungsverbesserung gegenüber der momentan verwendeten asynchronen Kommunikation demonstriert werden. Die für einen stabilen Betrieb nötigen Parameter wurden gemessen und die Ergebnisse diskutiert. Abschließend werden in dieser Arbeit weitere Optimierungsmöglichkeiten für das Kommunikationsmodul und zukünftige Messungen vorgestellt.

Contents

1 Introduction

Neuroscience as a scientific discipline with the aim to enhance our understanding of the brain, is in need of tools allowing a detailed investigation of the complex processes governing brain functions. Because complex systems like the human brain are difficult to understand analytically, simulation is one of the most efficient tools to test hypotheses and observe the development of brain-inspired networks. With the rise of computers at the dawn of the 21st century, computer simulation gained importance in the description of neural dynamics and the solution of differential equations abstracting the behavior of single neurons. However, it soon became apparent that this method of observing neural dynamics does not scale well on bigger networks, not only in the sense of simulation time but also in the realm of energy consumption. Even the most advanced supercomputers are merely capable of simulating small fractions of the brain's 10^{10} neurons and 10^{15} synapses at energy costs orders of magnitude higher and simulation speeds orders of magnitude lower than its biological equivalent.

One alternative to computer simulation is implementing the aforementioned neural networks in neuromorphic hardware. These systems are especially designed for neuroscience applications and aim to improve in key characteristics like low energy consumption, parallel execution and high scalability. Combining conventional compute units (ARM cores) and a custom interconnect technology allows the SpiNNaker system (*Painkras et al.* [2012]) to increase its efficiency compared to running simulations on supercomputers. SpiNNaker provides a fully programmable framework for large-scale simulations at the drawback of higher power usage compared to other more specialized neuromorphic systems.

The TrueNorth chip manufactured by IBM tackles this issue by using Complementary Metal-Oxide-Semiconductor (CMOS) technology to implement neurons with a fixed digital neuron model in a custom architecture. At the cost of flexibility, TrueNorth provides a platform for energy-efficient simulation of Leaky Integrate-and-Fire (LIF) networks at biological time-scales (cf. *Furber* [2016]).

In contrast to the previously mentioned systems, the BrainScaleS system focuses on analog neuron and synapse circuits. This allows neuronal network emulations running at a speedup factor of 10^4 with lower energy consumption per action potential than digital systems like SpiNNaker. It also incorporates further key features from biology like a temporally continuous development of neuron state variables, as analog circuits evolve according to their design continuously over time. To increase the connection density, decrease the energy needed for signal transmission and minimize the amount of post-processing steps, these circuits are integrated using wafer-scale technology. All of these traits allow for the implementation of neural networks, which are near to the biological

equivalent in a scalable environment at the aforementioned speedup, making this system ideal for learning applications.

A custom software stack manages experiment setup and execution on the BrainScaleS wafer system by generating, among other things, the configuration of on-chip components. These are sent via Gigabit Ethernet to the respective Kintex Field-Programmable Gate Arrays (FPGAs), which are located on the wafer module itself and control data streams between the host computer and groups of on-chip neurons and components. The chip configuration data consists of commands, which then can be either routed directly to the chip or stored in the FPGA's Playback Memory (PbMem) module. When using the PbMem module, the correct inter-command timing is guaranteed by the PbMem itself, as it sends the commands at previously specified timestamps.

The goal of this thesis is the implementation of a PbMem-based communication module in the software stack, granting a high degree of robustness and control over experiments on the wafer system. This is important for experiments, which rely on iterative reconfiguration of the setup, e.g. offline learning experiments. In particular, the so-called In-the-loop experiment, which employs machine learning techniques can benefit from the high reconfiguration speeds granted by PbMem configuration.

In chapter 2 the hardware and software environment are presented to describe the context of the PbMem communication module. Chapter 3 then describes the actual implementation of the communication module and some prerequisite structures. In chapter 4 the characterization of parameters belonging to different hardware abstraction classes and components is presented. Finally chapter 5 sums up the results and provides suggestions for further improvements.

2 Platform

2.1 Hardware

The Neuromorphic Physical Model version 1 (NM-PM1) system consists of a conventional computing part, which entails a cluster of 20 nodes and two front ends and supports the neuromorphic part, comprised of 20 BrainScaleS wafer modules. On these wafer modules the wafer itself and its communication infrastructure are located, which consists of on-chip as well as off-chip components. The Xilinx Kintex 7 FPGAs are the most important off-chip components, as they provide an interface to the wafer via the Joint Test Action Group (JTAG) protocol and the high-speed serial link of the Digital Network Chip (DNC) (which was integrated into the FPGA).

At its core, the BrainScaleS wafer-scale module's building blocks are 384 High-Input Count Analog Neuronal Network Chips (HICANNs), each containing 512 neurons and 512×224 synapses in a synapse array (in HICANNv2[1]; 512×220 in HICANNv4). The analog neuron circuits follow the Adaptive Exponential Integrate-and-Fire (AdEx) model, which was developed by Brette and Gerstner (*Brette and Gerstner* [2005]) as an expansion of the basic LIF point neuron model. The input spikes are filtered out of the data stream and processed by the synapse drivers. These convert the spike information from the presynaptic neurons into an excitatory or inhibitory postsynaptic potential (EPSP / IPSP). If the neuron fires, it generates a digital spike, which is then sent to the respective target neurons via the communication infrastructure.

There are multiple different on-chip components, which support the neuron circuitry in distinct ways. To supply the biological neuron model with parameters and for calibration of the hardware, four floating gate blocks with 24 columns and 129 rows each are located on every HICANN. The individual floating gate cells are standard Positive Metal-Oxide Semiconductor (PMOS) transistors with a non-connected gate, where a specific amount of charge is stored to represent a given parameter. A hardware state machine called the floating gate controller takes care of loading and unloading of the cells and their connection to the Analog-to-Digital Converters (ADCs).

The Layer 1 (L1) bus is a system of differential lanes, which span the wafer horizontally and vertically, carrying pulses from HICANN to HICANN. Six blocks of L1 switches are used to link horizontal to vertical connections and vertical connections to the synapse drivers. This allows pulses, which are injected by the Synchronous Parallel Layer 1 (SpL1) repeaters onto the horizontal lines to take complex paths across the wafer. The SpL1 repeaters mentioned above are a special type of L1 repeater and differ mainly in their ability to inject signals into the L1 bus. The common

[1] All of the following statements refer to HICANN version 2 (HICANNv2), as this was the chip revision used during development and testing.

L1 repeaters are grouped in six blocks, which are located on the edges of one HICANN. Because the signal timing and amplitude deteriorate with the amount of wire traversed (due to the wire acting like a low-pass filter), repeaters are used to resample and reinject the signal onto the L1 bus. They also provide cross-talk compensation by being able to connect neighboring lines via far-end crosstalk (FEXT) capacitors.

All of these components include memory, which stores information concerning their configuration and a memory controller. The L1 switches' and floating gate controllers' memories are registers implemented as Static Random Access Memories (SRAMs) with standard cells, which in theory have a read and write access time of one controller clock cycle[1]. In simulation however, the effective write and read access times were different (cf. chapter 4.3.2). The memories of the L1 repeaters in contrast are implemented as full-custom memory arrays, which have significantly longer access times but use less space than the standard SRAM cells. Their access times are one controller cycle for a write and 12 controller cycles for a read command (cf. *Grübl* [2016]).

All of the timings given above refer to one of the three time domains relevant in the context of this work. The aforementioned HICANN clock drives the digital infrastructure on the chip and is generated by a Phase-Locked Loop (PLL). The PLL generates a base frequency of 50 MHz, which can be modified by using a multiplier and a divisor to tune the resulting frequency to values of 50, 83, 100, 125, 150, 200, 225, 250 or 266 MHz. The second time domain is the DNC time, which is fixed at a frequency of 250 MHz and is used for releasing the packets from the FPGA to the HICANNs and vice versa. Lastly there is the FPGA time, which is also fixed, albeit at 125 MHz, and which drives all FPGA operations. The FPGA, DNC and HICANN time are counted in FPGA cycles (FC), DNC cycles and HICANN cycles respectively.

The configuration data is provided to the on-chip components via an Open Core Protocol (OCP) bus called the HICANN bus. This bus is constructed as a tree-like structure with three levels, one clock cycle[2] additional latency for each level and the components as leaves. Its structure can be seen in figure 1. Within the tree, a blocking handshake protocol is implemented, which requires the root controller to wait for the reaction of a leaf controller, verifying the command's reception, before sending the next command. The command data is provided to the root via the HICANN ARQ protocol (HICANNARQ), however the handshake information (whether or not the leaf controller received a given command) stays internal to the tree (cf. *Grübl* [2016], *Karasenko* [2016]).

[1] The controller clocks are generated from the HICANN clock and operate at $\frac{1}{4}$ of its frequency.

[2] These cycles refer to the HICANN bus clock, whose frequency is also defined as $\frac{1}{4}$ of the PLL (HICANN clock) frequency.

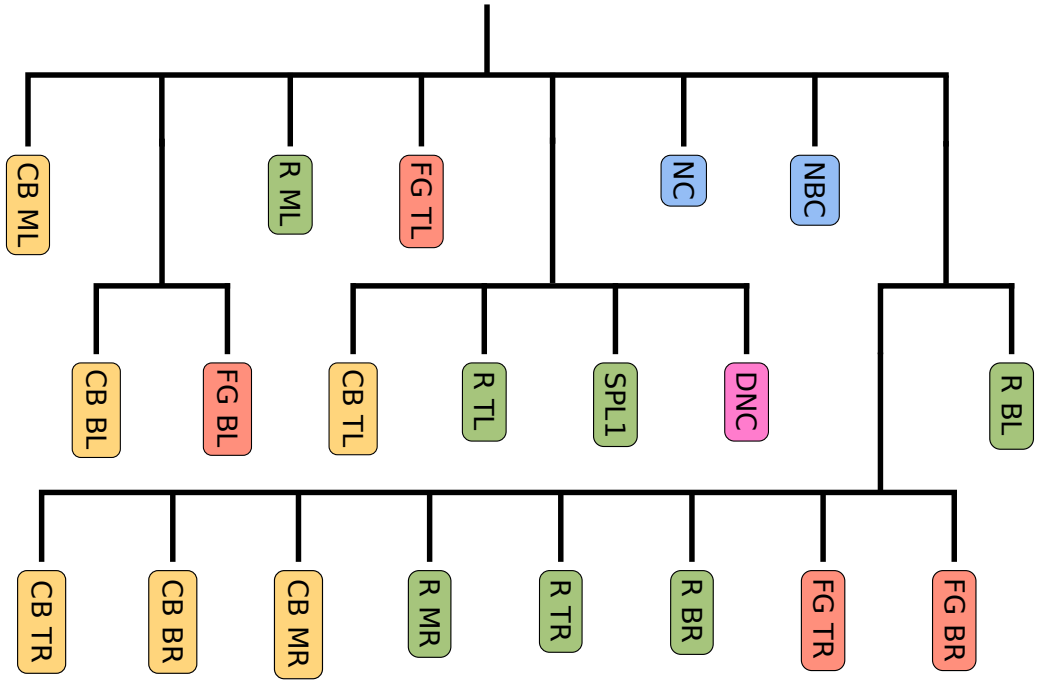

Figure 1: Schematic of the HICANN bus structure. The different colors encode different controller groups. Yellow represents L1 crossbar switches, green repeaters, red floating gate controllers, blue neuron control and neuronbuilder control and pink represents the DNC channel (used for spike output). The letters following the abbreviated description refer to different positions of the controller groups. The first letter encodes whether the group is at the top (T), in the middle (M) or at the bottom (B) of the HICANN. The second letter encodes, whether it is the left (L) or the right (R) group. In addition to the SpL1 repeaters there are six different repeater groups on one HICANN. The top and bottom groups drive the vertical L1 lanes and the middle repeaters drive the horizontal L1 lanes. The middle crossbar switches are concerned with connecting the horizontal and vertical L1 lanes and the top and bottom switches get the data from the vertical L1 lanes to the synapse drivers. Neuron control and neuronbuilder control are used for configuration of the synapse drivers and the Spike Timing Dependent Plasticity (STDP) controller

In the following paragraphs, the HICANNARQ will be explained in a simplified manner, allowing the analysis of its effects on the experiments considered in chapter 4. A more detailed and precise description can be found in *Karasenko* [2014] and *Debus* [2016].

The HICANNARQ manages the FPGA - HICANN communication via the FPGA's highspeed TX- (transmit port; FPGA to HICANN) and its RX-port (receive port; HICANN to FPGA). Due to the counterparts of the FPGA's TX and RX port being the HICANN's RX and TX port, the following names will be adapted for the sake of clarity. The TX port of the FPGA (transmitting data to

the HICANN) will be called FPGAOut and its RX port will be called FPGAIn. In analogy, the HICANN's TX port (transmitting data to the FPGA) will be named HICANNOut and its RX port will be called HICANNIn.

All of the port controller instances have a first in, first out (FIFO) queue, where up to 16 commands can be temporarily stored. All words (64 bit packets with 49 bits payload) sent in between the two sides have a header, which contains their sequence (an individual number, which identifies the packet) and an acknowledgement information (ACK) number to verify the communication. Within one side, connected to the receiving instance, a register exposes status information e.g. the last received sequence number to the transmitting instance. This number gets sent back in the ACK field of the next packet headed to the other side confirming the reception of the data, allowing the next word there to be released.

When sending a single word of 64 bits from the FPGA to the HICANN, the HICANNARQ on the FPGA side starts the `tx_timeout` (cf. figure 2). When the packet is received by the HICANNIn instance, the sequence information gets exposed to the HICANNOut instance and the `rx_timeout` is started at the same time. The ACK is not sent yet to avoid using two different packets for transmitting the ACK information and possible HICANN response-data. Upon now receiving the command, the HICANN bus sends it to its leaf controllers, which in case of a read command generate a response and send it to the HICANNOut instance, interrupting the `rx_timeout`. The sequence information exposed from the HICANNIn instance is then written in the ACK field of the payload's header and this packet is now sent back to the FPGA-side of the HICANNARQ, hereby (upon reception) interrupting the `tx_timeout`. If the original command however has been a write command, the HICANN will not return any data. In this case, upon finishing the `rx_timeout` the last word is resent with the new ACK and its payload is invalidated by turning off a specific bit in its header. This again after being received by the FPGAIn instance terminates the `tx_timeout`. If no ACK from the HICANN is received at all until this timeout ends, the original packet will be considered lost and therefore be resent.

This entire structure exists twice for two different communication channels named Tag 0 and Tag 1. They share one input from the PbMem, but send the commands to two different HICANN structures. Tag 0 supplies the aforementioned HICANN bus (and all components connected to it) with data, whereas Tag 1 sends data to a pipelined OCP slave which connects to the STDP and synapse controllers for example. This pipelined slave consists of multiple separate controllers on different levels, which implement a handshake protocol with their parent and child, but not with the root controller (as in Tag 0). The bus in between these controllers operates with the same packet rate as the HICANN bus, namely $\frac{1}{4}$ of the PLL frequency. These two different Tags are implemented to separate the (relatively fast) register SRAM accesses from the slower controller

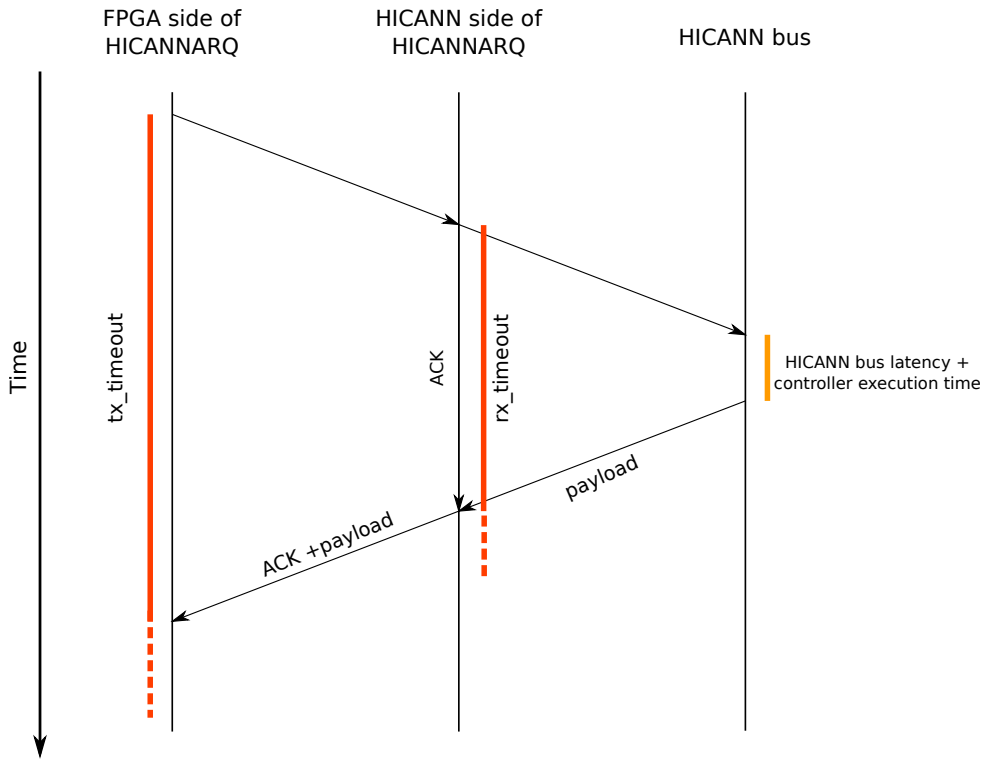

Figure 2: Schematic of the HICANNARQ protocol for a read command with long `tx_timeout` and `rx_timeout`. When this read command is sent from the FPGA to the HICANN, the `tx_timeout` gets started on the FPGA side. Upon receiving the packet at the HICANNIn controller, the `rx_timeout` is started. Then the packet is sent to the HICANN bus and the hardware controllers, where a response is generated. This response is received by the HICANNOut instance, interrupting the `rx_timeout` and then gets immediately sent to the FPGA with the ACK in its header. Upon receiving the packet at the FPGAOut instance, the `tx_timeout` is interrupted.

accesses into two distinct communication channels. Because one reticle consisting of 8 HICANNs is provided with data from one FPGA, 8 data streams (consisting of both Tags) are processed in parallel there.

As seen in figure 3, the chip data is either directly sent to the host from the FPGAIn instance (as currently with HICANN configuration data) or stored in the trace module of the FPGA (as spikes are currently handled).

The data arriving at the HICANNARQ's FPGAOut instance originates either from the frame decoder located directly behind the UDP controller or the PbMem module. There are two kinds of commands: pulses, which are always stored in the PbMem and HICANN configuration packets,

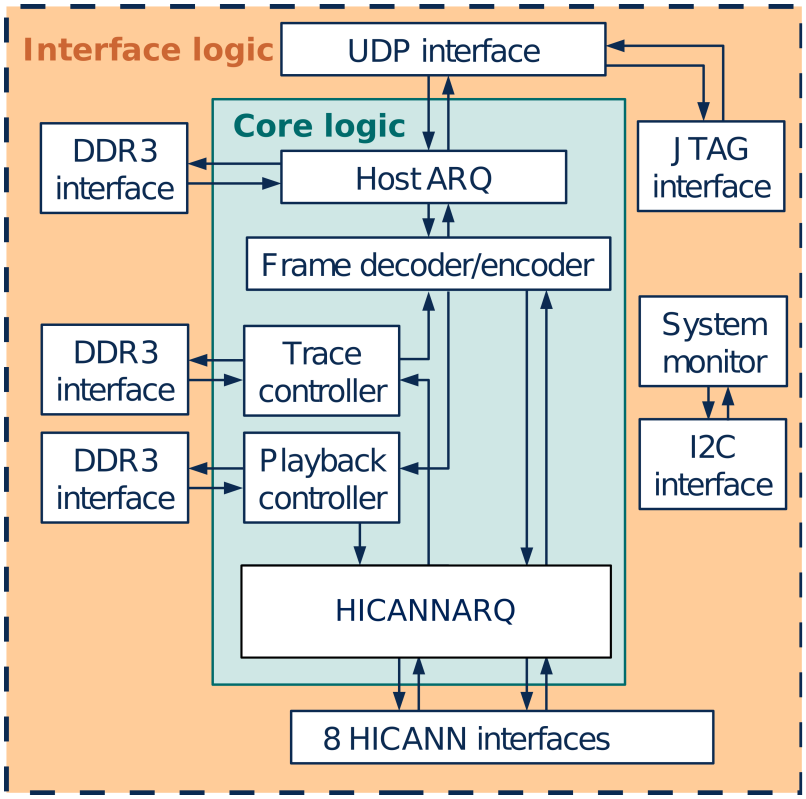

Figure 3: Schematic of the FPGA-internal modules. Coming from the host, the data is received at the User Datagram Protocol (UDP) controller, which forwards it to the frame en- and decoder. From there on it can either be sent directly to the HICANN (e.g. asynchronous configuration) or be stored in the PbMem's DDR3-SDRAM memory (cf. Playback Controller). In either way it ends up at the HICANNARQ, from where it is sent to the HICANN. The HICANN's return data can either be stored in the trace module's DDR3-SDRAM memory (cf. Trace Controller) or be sent directly to the host. This figure was modified from [*HBP SP9 partners*, 2016].

which can be sent via PbMem or directly to the HICANNARQ. PbMem configuration packets are sent in groups with a header stating the group size and its release time, whereas non-PbMem packets merely consist of the HICANN configuration data. After loading all desired pulses and configuration commands into the 512 MiB Double Data Rate Synchronous Dynamic Random Access Memory (DDR3-SDRAM), the PbMem can be started. From this moment on the PbMem controller guarantees the release of the command groups at their prespecified times. After having decoded the commands (which takes 6 FC) they are written into a FIFO queue from where they are pushed to the HICANNARQ. If the FIFO is full, commands get dropped and an error flag is raised, which is currently not visible to the host.

The data arriving at the PbMem module originates from the frame decoder, which gets its data in turn from the Host ARQ protocol (HostARQ) module. This module implements the HostARQ protocol structuring the communication between FPGA and host computer and is situated directly below the UDP interface.

2.2 Software

To keep within the scope of this thesis, the software stack will be presented in a simplified manner. For a complete software architecture description see *Müller* [2014].

The setup of experiments is being described in PyNN (*Davison et al.* [2008]), a common tool in neuroscience. This provides an Application Programming Interface (API) front end for many different back end solutions e.g. software simulators like NEST or NEURON or neuromorphic hardware systems like BrainScaleS or SpiNNaker. For BrainScaleS a custom PyNN back end called `PyNN for the BrainScaleS Hybrid Multiscale Facility (PyHMF)` was implemented to expose the populations, projections and parameters defined in PyNN to the software stack. This information has to be mapped onto the hardware and the neural connections have to be routed on the wafer, which is the task of the next layer called `marocco` (cf. *Jeltsch* [2014], *Klähn* [2016]). After having generated a chip configuration and having defined the course of the experiment, the `Stateful Hardware Abstraction Layer (stHAL)` (cf. *Koke* [2016]) is used to represent this configuration by using `Hardware Abstraction Layer Backend (HALbe)` containers and coordinates (cf. *Jeltsch* [2014], *Müller* [2014], *Koke* [2016]). It then uses the functions exposed in `HALbe` for the actual configuration of the hardware. `HALbe` itself is used to abstract hardware properties by providing component and coordinate abstractions. The configuration functionality it provides is concentrated in `FPGABackend` and `HICANNBackend`, which expose functions from `hicann-system`. The last layer before the Ethernet and HostARQ implementation is `hicann-system`, which abstracts hardware control units and the hardware access. Its functionality is mainly defined by three major classes named `ControlModule`, `Stage2Ctrl` and `Stage2Comm`.

`ControlModule` is a mixed hardware abstraction and control class and its derivatives can be split into three parts.

Firstly there is `ReticleControl`, which is the only access point for HALbe and enables it not only to use all hardware abstraction classes but also the communication classes which are going to be discussed later.

The second group is mostly deprecated and contains `DNCControl`, `FPGAControl` and `HICANNCtrl`. At the moment `DNCControl` and `FPGAControl` are only used in old low-level hardware tests and

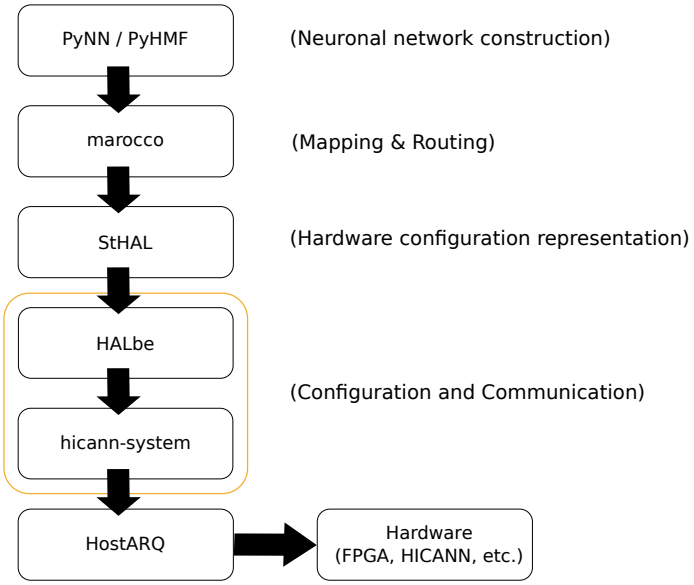

Figure 4: Schematic of the Neuromorphic Physical Model (NM-PM) software call stack (further details can be found in the text). The user constructs neural networks using the PyNN API, which are translated into a hardware configuration by `marocco`. This is held in `stHAL`, which uses `HALbe`'s hardware abstraction and functions for communication to the hardware. This communication is using `hicann_system` and the host-side HostARQ, which then in turn sends the data to the FPGA (and subsequently to the HICANN).

don't play a role in the newer code. The `HICANNCtrl` class however provides `ReticleControl` access to all hardware abstraction classes.

Lastly, there is the real hardware component abstraction as for example in `RepeaterControl` or `L1SwitchControl` (referred to as hardware control classes). These classes provide an API for the configuration of the corresponding on-chip components e.g. by transforming coordinate parameters or exposing functions which provide access to subcomponents. The arguments given to these functions are then processed and used to call the base class methods `write_cmd()` and `read_cmd()`. From there they are passed on to `Stage2Ctrl`.

`Stage2Ctrl` is used to provide the hardware abstraction classes mentioned above with access to the correct communication class. It combines information about the HICANN number and the communication channel to be used.

Lastly there are the communication classes, which are all derived from `Stage2Comm` and provide different communication protocols on a per-reticle basis. The currently available communication modes include JTAG-based and HostARQ-based access, which are implemented in the

base class and the derived `S2C_JtagPhys2Fpga_Arq` class. The latter one uses `HostALController` to implement its communication functions and provides a wrapper for the methods implemented there. `HostALController` is implementing the packet-formatting and provides asynchronous (non-PbMem) access for HICANN configuration commands and PbMem based access for pulses.

3 Implementation of the communication module

3.1 Introduction

Because there are several different ways to set up experiments on the BrainScaleS wafer system, a flexible and well defined construct for abstracting PbMem programs is needed. The communication module presented in this chapter tries to provide a high-performance framework with a clean API, which is able to support existing asynchronous as well as PbMem optimized experiments. It uses well-structured data containers to future-proof the module for tasks like changing single commands in an already assorted program. Parts of the implementation were already started in *Mauch* [2016] for pulse handling.

3.2 Prerequisites

3.2.1 Implementation of a time container

A very important - but before the start of this thesis still missing - feature is a container providing a universal description of time. This container has to support any of the multiple units being used in the code like DNC cycles, FPGA cycles or nano seconds. Up to now, in every time-dependent function untyped integers were used and the variable's name or a comment next to its definition had to provide information regarding its unit, which was error-prone. This issue was addressed by implementing two containers, which aim to provide a more robust framework for measures of time.

These two containers named `global_time` and `global_time_diff` are data structures, which describe absolute points in time as well as durations. A clear separation between both is essential to avoid confusion and to prevent accidental mixups in their usage. Factory create methods (cf. *Stroustrup* [2013]) are used to enforce time unit specific object construction and are (next to the default constructor needed for compatibility to some of the serialization and deserialization functions within `HALbe`) the only creation methods being exposed. Access to the respective numerical values is limited via unit-specific getter member methods.

The arithmetic operators connecting `global_time` and `global_time_diff` are loosely based on `C++11`'s `std::chrono` library, where similar objects are being described (cf. table 1).

However, because the time points of entries in the PbMem program are not absolute like the ones in `std::chrono`, an explicit conversion operator is required to allow operations like the one described in excerpt 1.

left hand side	operator	right hand side	result
global_time	+	global_time_diff	global_time
	-	global_time	global_time_diff
	-	global_time_diff	global_time
global_time_diff	+	global_time_diff	global_time_diff
	+	global_time	global_time
	-	global_time_diff	global_time_diff
	-	global_time	global_time

Table 1: Relationship between time containers and the return values of different arithmetic operators. Because the addition of two time points does not make sense, the `global_time::operator+(global_time)` was not implemented. The operators describing the addition and subtraction of `global_time` objects to and from `global_time_diff` objects were implemented according to `std::chrono`'s implementation of `std::chrono::time_point` and `std::chrono::duration`.

```
1  PbTraceEntry packet;
2  global_time max_command_time_last_chunk;
3  global_time_diff fpga_hicann_delay;
4  global_time release_time;
5  release_time = packet.getTime() - max_command_time_last_chunk - fpga_hicann_delay;
6
7  // This gets interpreted by the compiler as the following:
8
9  release_time = (packet.getTime() - (max_command_time_last_chunk + fpga_hicann_delay));
10 release_time = (packet.getTime() - temp);
11
12 // unit of temp is global_time and difference of two global_times is global_time_diff -> compiler error
13 // -> explicit conversion operator is needed
14
15 release_time = (global_time)(packet.getTime() - temp);
```

Excerpt 1: Illustration of the need for an explicit conversion operator. The `release_time` of an entry is calculated from its event time in the PbMem program minus the maximum command time of the last chunk already executed and the FPGA-HICANN delay. After the compiler has collapsed this equation, the unit of the resulting object is `global_time_diff`, which has to be casted to `global_time`.

3.2.2 Implementation of data types for the PbMem Program

Necessarily, the PbMem communication module has to store the PbMem program, which is produced by calling `HALbe` back end functions. These in turn generate write and read commands to the HICANN. To properly implement this, one container for the individual entries of the program and another for the program itself is needed. The `struct PbTraceEntry` seeks to provide the former by using a `union` of raw 64 bit unsigned integers for storage of the payload information containing four different members for multi-purpose usage. Base members are the raw data, which is just an unsigned 64 bit integer and the generic instance, which holds information common to all types of entries (like the HICANN or DNC address and packet type information). Additionally there are the pulse data representing the information storage for software-generated pulses and the HICANN configuration storage, holding the information of the configuration packets.

The `PbTraceEntry` objects are created by explicit constructors, whose interfaces are based on the existing `HostALController` interface for pulses and the `Stage2Ctrl` interface for configuration packets. One constructor for HICANN configuration packets and two constructors for pulse events were implemented to provide support for either assembling pulses from the raw information or using a preformatted 16 bit label, which incorporates this information.

The access to this container is designed to be open, to ensure that every parameter is modifiable even after the construction through the explicit single-purpose constructors. In the future this might allow selectively altering individual entries without having to rewrite the whole program. This would be very useful during parameter sweeps, as long as the FPGA and its firmware do not provide this functionality (cf. chapter 5).

The `PbTraceEntry` objects are stored in the `struct EventChunk`, which uses a `std:vector` to hold this data. Some of the standard functions like `std::vector::at()` or `std::vector::{begin,end}()` are exposed for convenient access. It also implements an optional sorting functionality based on `std::sort()`, which prepares the program for sending to the FPGA.

Currently, single pulses get abstracted using the `PulseEvent` container and pulse groups are stored in the `PulseEventContainer` object, which will be replaced by `PbTraceEntry` and `EventChunk`. However, due to time constraints this change was not completed as of now because this would require major changes to `HALbe` and `stHAL`.

3.2.3 Delay

Due to finite hardware controller access times on component SRAMs and infrastructure latencies it is necessary to implement delays, which belong to a specific command and define when the following command can be sent at the earliest. For PbMem communication these delay parameters become increasingly important, as the latencies between FPGA and HICANN are much smaller than the host - HICANN latency dominating in the case of asynchronous usage. Thus hardware controller execution times get relevant. Because of different controller implementations, SRAM access times and the HICANN bus topology, the delay is hardware component-specific and was therefore implemented in the base class `CtrlModule` as a protected member. Its derivatives are able to modify the delay value in their respective constructors if necessary. The default delay is set to a minimal value which guarantees error free execution.

A previously existing (but non-functional) delay argument was removed from high-level access methods in `hicann-system` and can now only be passed on to `CtrlModule`'s `write_cmd()` and `read_cmd()` methods. It is necessary to expose this argument here to ensure backwards-compatibility to the `hicann-system` testmodes, which for historical reasons use `CtrlModule`'s low-level methods with explicitly specified delays rather than the derivative's access methods. Additionally a public getter method was added, which allows the user to look at (but not modify) this variable. In chapter 4.3, delays for most of the control modules in `hicann-system` are characterized and default values are defined.

To grant the user a high degree of control over the program whilst not changing the API for backwards compatibility reasons, it is now also possible to define the exact event time of a packet. This is achieved by introducing a `global_time event_time` variable to all access functions in `hicann-system` and in HALbe, which is defaulted to a non-valid value. The value of the `event_time` parameter gets checked in the `write_cmd()` and `read_cmd()` methods in `hicann-system` and the respective implementation of `Stage2Ctrl`'s `issueCommand()` method is called.

3.3 Implementation

The starting point of the implementation was `S2C_JtagPhys2Fpga`, as this class already provided the necessary initialization functionality and JTAG access to the derived asynchronous module `S2C_JtagPhys2FpgaArq`. A decision was made to change the current naming scheme and to call the new class, which would also be derived from `S2C_JtagPhys2Fpga`, `S2C_HostArq_PbMem`. It is intended to change the name of the asynchronous module to `S2C_HostArq_Async` in the future.

```
1 // Interfaces for configuration packets
2 void issueCommand(uint jtag_hicann_nr, uint tagid, ci_payload *data, global_time_diff del);
3 void issueCommand(uint jtag_hicann_nr, uint tagid, ci_payload *data, global_time_diff del, global_time
      event_time);
4 // Interfaces for pulse events
5 void issueCommand(global_time event_time, uint8_t dnc, uint8_t jtag_hicann_nr, uint8_t gbit_link, uint8_t
      l1address);
6 void issueCommand(global_time event_time, uint16_t label);
```

Excerpt 2: Interfaces of the communication module's `issueCommand()` member method. These four implementations show the different possibilities of adding entries to the PbMem program. Next to the payload and addressing information commands can either only provide their delay or additionally the time, at which they are to be inserted into the PbMem program. Pulse events always have to provide their `event_time`, but their addressing information can either be passed on individually or compressed into the 16 bit label.

The delay mentioned in chapter 3.2.3 is implemented by storing the time of the last packet inserted into the PbMem program and the time the next packet is allowed to be sent. However, because one communication module serves one FPGA (i.e. eight different HICANNs on the current system) these two variables are held HICANN-wise in two `std::arrays`, `last_event_times` and `next_event_times`. This information is used in the multiple overloaded `issueCommand()` methods that are implemented in `S2C_HostArq_PbMem`, to check if the `event_time` of a command is valid and if not, to shift the command to the next valid time (not without warning the user of course).

The interface of these `issueCommand()` methods is based on the constructors of `PbTraceEntry` and is shown in Excerpt 2. As shown there, HICANN configuration packets can be inserted into the PbMem program without explicitly stating when they are to be released. In this case the `issueCommand()` method generates the next possible event time from `next_event_times` and the HICANN number. However, if an `event_time` is given, the method checks whether the given time is allowed (i.e. larger or equal than the `next_event_time` for this HICANN). Should this not be the case, it shifts the packet to the next allowed event time whilst warning the user.

One tricky part of the implementation was the need to have full backwards compatibility to the existing asynchronous tests. This is difficult because in the asynchronous case the configuration is done incrementally and data is requested from the chip whilst reconfiguring it. It is possible to provide this compatibility by modifying the existing `recvData()` method to check whether all commands in the program were sent to the chip. Should this be the case, data is just requested in the usual way from the data buffers. However, if there are unsent commands in the software

PbMem program, they have to be sent down before trying to receive data. To check this condition, the index of the last command sent to the chip is stored and compared to the total size of the PbMem program. To retain a logically (and chronologically) consistent program it now becomes important to ensure that the `event_time` of new commands, which are to be added is larger than the maximum `event_time` of the last part. If this is the case, it is possible to resend the program being assembled in its entirety. This prerequisite is enforced by setting the `next_event_time`s of all HICANNs to their maximum value after the program is sent and using the already implemented checks of the aforementioned `next_event_time`s in the `issueCommand()` methods.

After having checked the given `event_time` against the `next_event_time` of the specified HICANN and having generated the correct event time from the `next_event_times` and the HICANN number (if necessary), the payload gets assembled from the provided information. Then the `PbTraceEntry` is constructed and stored in the `EventChunk pbmem_program` member variable of the communication module. Now the `last_event_time` and `next_event_time` variable of the given HICANN have to be updated to store the packets delay information.

The `sortAndSend()` method is the most complex method of the communication module, as it has to convert the abstracted PbMem program into correctly formatted groups and overflow packets within one Ethernet frame according to the specification (as seen in figure 5). These groups are consisting of up to 128 64 bit words, which corresponds to a maximum of 255 pulse events (à 32 bit) or 127 configuration packets (à 64 bit). Each group has its own header (64 bit for configuration and 32 bit for pulse packets), which stores the release time of the group from the FPGA's PbMem module and the number of its pulses or configuration commands. Because the HostARQ provides a streaming interface, i.e. is agnostic to the distribution of the group within the respective frames, it is possible that the data is spread out across two frames. Overflow packets are added, if the distance between two consecutive events is larger than one half of the width of the FPGA's clock counter.

The implementation of `sortAndSend()` is based on the `addPlaybackPulse()` method from `S2C_JtagPhys2FpgaArq`'s `HostALController`, which adds a single pulse event to a preexisting buffer. The combination of the algorithm parsing the PbMem program, filling the Ethernet buffer element-wise and the possibility of commands occurring in groups creates the necessity of updating the group header when a new word is to be appended to a group. This requires `sortAndSend()` to use a double buffering scheme, as one group might stretch over two frames. After the first Ethernet frame is completely assembled (apart from maybe updating some of the group headers) it is swapped with the second frame buffer, which then is being assembled as well. When this is finished, the old frame immediately gets sent to the HICANN and the buffers get swapped again,

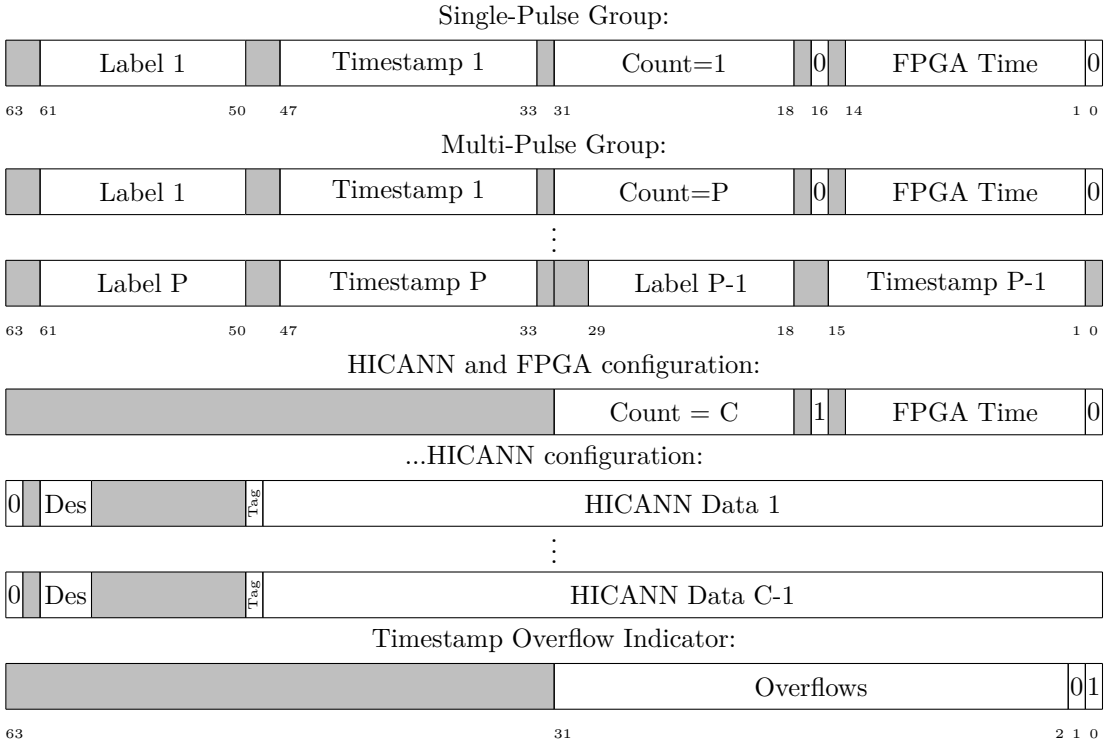

Figure 5: Specification of the FPGA Playback Data's format, taken from [*HBP SP9 partners*, 2016]

overriding the old frame's buffer. Some ideas concerning the improvement of this process are being discussed in chapter 5.

The FPGA's PbMem module can be started by using the `startPbMem()` method, which resets the HICANN time counters via JTAG and subsequently primes and starts the FPGA system time counters to synchronize them. This is unfortunately very slow[1], which has a major impact on the communication module's performance, as the PbMem program needs to be started relatively often in non-PbMem optimized cases. The FPGA functionality was changed over the course of this thesis, as suggested by the author, but due to time constraints these changes could neither be thoroughly tested nor used. Further information on this subject can be found in chapter 5.

To receive data after an experiment has been started, the `recvData()` method was adapted from `S2C_JtagPhys2Fpga_Arq`'s implementation. As already discussed, this sorts and sends the unsent commands in the PbMem program and then starts the PbMem module of the FPGA. Although it

[1] With the use of `C++11`'s `std::chrono` library, a duration of $(2146 \pm 25)\mu s$ for the function's execution time was measured.

has not been investigated in detail, the success of the floating gate writing seems to be sensitive to an added sleep of $\approx 2000\mu s$ after the PbMem start was executed. This has to be investigated in the future, as this may reflect some behavior of the controller, which is not yet fully understood. The rest of the implementation was left unchanged, as time constraints did not allow implementing the new data containers or tweaking the receive functions' efficiency.

To give the user a high degree of control over the PbMem program, some access functions were built into the communication module. The `get_eventtime()` method allows the user to get either the `last_event_time` or the `next_event_time` of the given HICANN e.g. to use it as the timing-offset of a spike train, which is to be inserted in the PbMem program. The `increase_eventtime()` method can be used to increase either of the `event_time`s of a given HICANN and is used e.g. in the busy wait methods of `SynapseControl` or `Syn_trans` to increase the existing delay of a write or read command to the `controller_timeout`, which was characterized in chapter 4.3.

For debugging purposes the `print_pbmem_program()` method was introduced to enable streaming the PbMem program's content in human-readable form into any given `std::ostream`. If no argument is given, the default is `std::cout` (the command line). However, if the stream given is a `std::ofstream` (for file output), the output is CSV-formatted. Because of time constraints the HALbe integration of this feature was not finished, as this would require a serialization of `std::ostream`. Lastly the `Reset()` method was implemented to completely reset the entire communication module and all associated parameters to their default values and delete the PbMem program, which is useful when executing multiple experiments consecutively.

4 Testing of the communication module

4.1 Introduction

The aim of the tests presented in this chapter is to demonstrate the benefits of the PbMem communication module and to characterize the delays needed for the hardware abstraction classes' operation. Unfortunately it was out of scope of this thesis to build a software verification testbench for the communication module and as no HICANN configuration loopback exists in the FPGA, a constrained random testing approach was not possible. The verification of the PbMem module's functionality was achieved by executing low-level tests like `tmag_switchramtest`, which writes random values in the L1 switch SRAMs of the HICANN and reads them out again as well as additionally observing the Ethernet packets directly with the tool `wireshark`.

4.2 Benchmark

The benchmark compares a typical asynchronous test to a PbMem optimized program, namely the `tmag_switchramtest` to the `tmlp_pbmemcommtest`. On one HICANN, there are 112 SRAM cells in the top left L1 switch block, which in the asynchronous case get written one after another and then read out in groups of four. The `tmlp_pbmemcommtest` stores a write command followed by a corresponding read command for each SRAM cell in one PbMem program. After this has been sent to the FPGA and the PbMem module has been started, the data is received at once. The scaling of the tests in the number of commands sent is provided in the asynchronous case by looping the write, read commands and receive calls a certain number of times determined by a command-line argument. As for the PbMem case this is done by first assembling one program with the same amount of write and read commands as in the asynchronous case and then executing the receive calls in one block at the end. The benchmark executes the tests with 1, 10, 100, 1000 and 10000 loops and repeats every measurement 10 times to gather statistics. The measured time is the duration from start-time to end-time of the program.

One would expect to see the asynchronous test's execution times rising faster than the PbMem's, because each command is a blocking access to the chip whereas the commands in the PbMem usecase only have to be sorted and sent. The PbMem test's execution times should be dominated by `std::sort()`'s complexity when increasing the number of commands.

Figure 6 shows the results of this benchmark. As one can clearly see, the PbMem test scales asymptotically significantly better than the asynchronous one. At 10000 repetitions the asynchronous execution time is $(40 \pm 1)s$ vs the PbMem's execution time of $(8.0 \pm 0.1)s$. Initially however, the mean

execution time of the PbMem test is slightly higher than for the asynchronous test, but remains within the 1σ interval of the PbMem's data. A reason for the error of the first PbMem execution being significantly larger than every other one is probably the FPGA being power cycled before the test execution, as the benchmark starts with the PbMem experiment. It may be caused by the very first high speed initialization taking longer than the following ones, but this was not investigated in detail and could not be reproduced reliably.

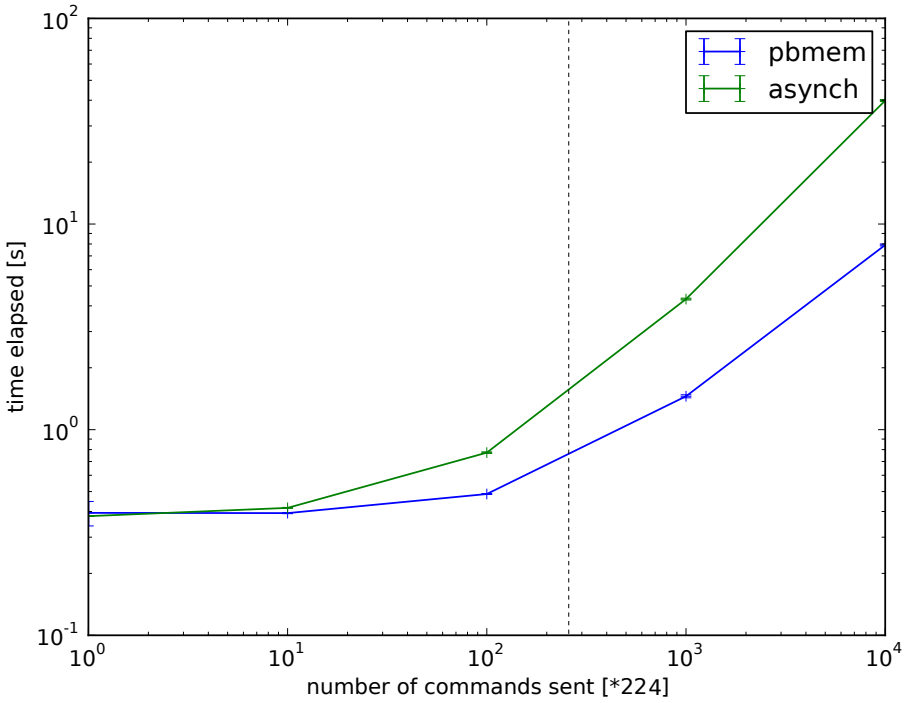

Figure 6: Results of benchmark comparing asynchronous and PbMem test. The elapsed time of each program in dependency on the number of commands sent (displayed in increments of 224) is mapped. Mean values and their standard deviations for 10 measurements per data point are shown. The green data points depict the mean times for the asynchronous `tmag_switchramtest`, whereas the mean times measured for the `tmlp_pbmemcommtest` using the PbMem are displayed in blue. The dashed vertical line marks the typical size of a complete HICANN configuration (cf. *Karasenko* [2014]).

The scaling however is not exactly as theoretically assumed. A short analysis showed that this data does not match `std::sort()`'s theoretical complexity of $\mathcal{O}(Nlog(N))$. Even with an offset to compensate the initialization time of the program (which of course is independent of the number of commands sent afterwards) the fits to these measurements did not converge. Further analysis

using the Linux performance measurement tool `perf` reveals that the complexity of `std::sort()` does not exclusively dominate the total execution time as initially assumed.

With a contribution of 67.73% to the total execution time of the program, the method `recvData()` is dominating every other method. This is the case because in its first call, the PbMem program is sorted, sent to the FPGA and the PbMem module is subsequently started. A portion of 59.74% of this time in turn is spent in the `sortAndSend()` method, which is thus (as expected) the main contributor. Its execution time in turn is mainly dominated by `std::sort()` (57.42% vs. the next function with 9.01%). However, 24.35% of the `recvData()`'s time is also spent in the `getReceivedHICANNConfig()` method which is one of the non-optimized methods from `S2C_JtagPhys2Fpga_Arq`. This means `std::sort()` is not exclusively dominating the test execution time, but `getReceivedHICANNConfig()` also has a significant influence.

In conclusion it can be stated that even with the non-optimized receive functions from `S2C_JtagPhys2Fpga_Arq` the PbMem communication module offers a significant speed advantage compared to the asynchronous one, especially at a high number of commands sent. As expected, the two perform nearly equivalent at very low numbers of commands. The PbMem communication module's advantage can additionally be increased by optimizing the receive functions.

4.3 Delay Characterization

4.3.1 Introduction

The following section describes the quantification of the delay parameters needed for the different hardware abstraction classes. Classes considered were `L1SwitchControl`, `RepeaterControl` and `FGControl`, which provide access to different register SRAMs and `SynapseController` as well as `SynTrans`, which provide access e.g. to the synapse controller. Register SRAMs are used in components like L1 switches, repeaters and floating gate controllers to store configuration data as for example the switch connection, repeater direction or parameters for the floating gate access. More complex controllers like the synapse controller use a buffer register SRAM to preload data for the controller to be written into a matrix. The synapse controller has access to a weight matrix, but can configure the synapse drivers as well. In the asynchronous usecase, after the write or read command is sent to the controller, its status register is continuously polled until the busy bit is 0. This access time dominates in experiments, where synapse weights are frequently updated e.g. the offline learning experiment mentioned earlier, which is currently being run asynchronously.

The `wait_while_busy()` and the `arraybusy()` methods implement the polling of the respective controller's busy bits in the `Syn_Trans` and `SynapseControl` classes respectively. The former

method uses a while loop around the read command for the busy bit, whereas the latter one only implements the readout without the while loop. Because these methods are called directly after the controller access, it is guaranteed that this access command is the last in the PbMem program. Thus, the functionality of these methods was changed to modify the delay of the last command in the PbMem program to the time the controller needs for this access for experiments using the PbMem communication module. This decreases the synapse controller access time from timescales dominating the digital chip reconfiguration ($\mathcal{O}(ms)$ per row) to merely the actual controller access times within the PbMem program ($\mathcal{O}(\mu s)$ per row).

The tests implemented to characterize the delays are based on the existing `tm_wafertest` and are split into the register SRAM tests and the controller timeout tests.

4.3.2 Register SRAM tests

The basic structure of the SRAM tests is to write data into the respective storage and to read it out again. The read out data is then compared to the written one. To gather statistics, this process is repeated 20 times at a predefined delay value before moving on to the next delay. The number of correctly read entries as well as the number of wrong response data is then written into files and later evaluated in software.

As explained in chapter 2.1, if the commands arrive at the HICANNARQ FIFO queues faster than the hardware controllers can process them, it is possible that they get dropped when the FIFOs are full. If write commands get dropped, the data read back is not correct. If read commands get dropped, the `recvData()` method throws an exception because the packet with a certain address was not received. This causes the sum of correctly and incorrectly read data to be lower than the total number of expected packets. Hereinafter, the first delay from which on all data is read back correctly is considered stable.

It was investigated, whether the order of write and read commands sent to the HICANN affects the measurement. First all write commands were sent down in succession and then all read commands were sent, whereas in the second execution write and read commands were interleaved by sending a write command to one SRAM cell, followed immediately by the corresponding read command. Figure 7 shows the difference in outcome between these two procedures.

Said figure is composed of the subpart (a) showing write and read commands being sent consecutively and part (b) showing write and read commands being interleaved per SRAM cell. Each of the subparts is again consisting of the three tests for the L1 switch, the repeater and the floating gate SRAMs. For every subpart, the number of correctly read entries is depicted in green and the number of incorrectly read back data in red. Because this measurement was repeated 20 times

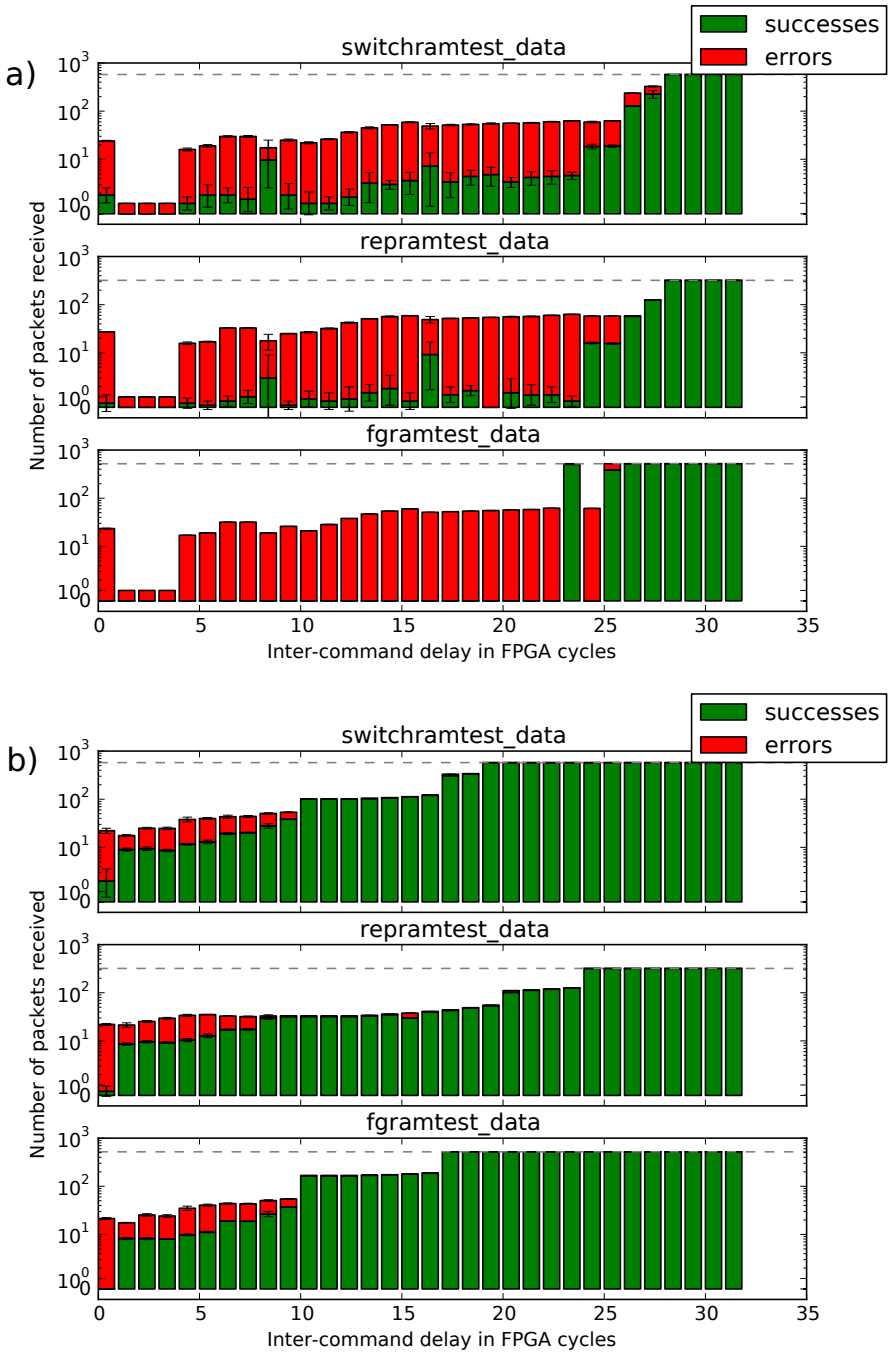

Figure 7: Test of switch, repeater and floating gate SRAM at PLL frequency of 250 MHz showing the difference between consecutively sending write and read commands (a) and interleaving them (b). The y-axis in each of the plots shows in green the number of correctly and in red the number of incorrectly received data on a logarithmic scale (linearized from one to negative one). The values are averaged over 20 repetitions and depicted with their standard deviation. On the x-axis the respective delays in FPGA cycles are shown. The dashed line represents the number of read commands sent and thus the expected number of returned data.

per delay, the values shown are averages with their respective errors. The x-axis represents the different inter-command delays in FPGA cycles at which this test was executed. If the sum of correctly and incorrectly read back data is lower than the dashed line (representing the number of expected commands), at least one read command was dropped and thus no data was received for this address. A dropped write command is visible as incorrectly read back data, depicted in red. As seen in figure 7a), the number of dropped write commands and the values of the first stable delays are much higher for consecutively sent commands than for interleaved commands, shown in figure 7b).

This difference is caused by the HICANN side of the HICANNARQ. As stated in chapter 2.1 the data generated by a read packet at the component controllers immediately transmits the ACK to the FPGA side by terminating the `rx_timeout` (cf. figure 2). However, when only sending write packets, the component controllers do not generate response data and hence the `rx_timeout` defines the ACK rate. Because all tests in this experiment were executed with standard values for the HICANNARQ parameters, the `rx_timeout` is very long and thus causes the FIFO queue of the FPGAOut instance (cf. chapter 2.1) to stall, resulting in packet loss. Consequently, to grant more stability and avoid measuring only the `rx_timeout`, all register SRAM tests considered in this thesis were executed with interleaved write and read commands.

All tests (also in chapter 4.3.3) were repeated for PLL frequencies of 250, 200, 150, 125, 100 and 50 MHz and were executed on different FPGAs (i.e. different HICANNs) on wafer 20.

By simulating the L1 switch SRAM test in a software testbench it was shown that at the PLL frequency of 250 MHz the inter-command delay for which no HICANNARQ resends occurred was 18 FPGA cycles (FC) for write and 22 FC for read commands (cf. *Grübl* [2016]). The experiment should yield the maximum value, as only one delay variable was used for both types of commands. Due to the repeaters using full-custom memory arrays, which are very slow[1] compared to the standard SRAM cells used in the L1 switch and floating gate controller SRAM, it would be expected to see that the stable delay of this component is systematically larger. Furthermore one would expect seeing a dependency of the number of read back commands on the location of the respective component in the HICANN bus, as the latency increases with the depth of this component.

Figures 8 and 9 show the results for the PLL frequencies of 250 and 125 MHz (results for the other PLL frequencies can be seen in appendix A.2). At a PLL frequency of 250 MHz, the observed delay for the switch SRAMs is 19 FC, which is smaller than the expected 22 FC. The delay of 24 FC for the repeater SRAM is considerably larger than the switch SRAM's, but again smaller than the value of 32 FC, which was measured in simulation. The difference between the observed and

[1] The HICANN bus Round Trip Time (RTT) of a read command and its return data to the top right repeater block was measured as 32 FC in simulation. This should be the maximum value for the repeaters, as this block is located at the lowest level of the HICANN bus.

expected value could be caused by the FIFOs of the HICANNARQ and the PbMem module in the following ways:

Firstly, it is possible that the FIFOs are deep enough to provide space for the complete program to execute at a smaller delay than the maximal controller execution time. Because too small delays are only visible, if a command gets dropped when trying to push it into the HICANNARQ FIFOs queues, these FIFOs can just fill up slowly due to the difference between the command input rate of the PbMem and the rate the controller accepts commands. This will not be visible, as the HICANN can continue processing the remaining commands stored in the FIFOs after the PbMem controller has finished sending the PbMem program. This problem was investigated by increasing the amount of write commands sent per cell access and comparing this to the data gathered before (cf. figure 20 in A.2). Interestingly the measured delays decreased even further as opposed to rising as expected. The reason for this might be the amended HICANNARQ frame usage. Because three to four write commands and one read command are sent to one address consecutively, the answer of the read command triggers the sending of the frame consisting of the write and read command ACKs. This might eliminate a possible residual influence the HICANNARQ has on the result.

Figure 10 illustrates the second effect possibly influencing the measured delays. If the sum of delays of consecutive read and write accesses is greater or equal to the time the controller needs for execution of the commands, it is possible for both commands to be executed without blocking the FIFO. If e.g. the delay of the read command is smaller than the execution time the controller needs for this command, the HICANN bus blocks the following write command until this read command is finished. This pushes the write command's execution time slightly backwards, but if its delay is large enough for it to be finished before the next command arrives, the controller is again ready to accept the new one.

This problem can be circumvented by splitting the delay into a write and read command delay and characterizing the two independently of each other (or only one, as the correct value of their sum is already known). However, for this the HICANNARQ parameters have to be controlled to ensure real hardware access times being measured. Further thoughts on this subject can be found in chapter 5.

The HICANN bus latency's effect on the measurements is visible in figures 8 and 9 by the step-like increases in correctly read back data. These steps occur when the inter-command delay is large enough to account for the increased latency to the leaf controller. Its handshake with the root controller can then be completed before another command arrives at the FIFO queue. Because the HICANN bus is then ready again for the next command, these don't get backed up causing them to get dropped.

Figure 8: Results of register SRAM delay characterization for PLL frequency of 250 MHz. This measurement is the same as in figure 7 (a) and (b). The stable delays of the control module tested are 19 FC for the L1 switches, 24 FC for the repeaters and 17 FC for the floating gate controllers.

Figure 9: Results of register SRAM delay characterization for PLL frequency of 125 MHz. Here the same tests as in figure 8 were executed for a different PLL frequency. The stable delays for the L1 switches, repeaters and floating gate controllers are 37 FC, 44 FC and 34 FC and are thus significantly larger than for 250 MHz.

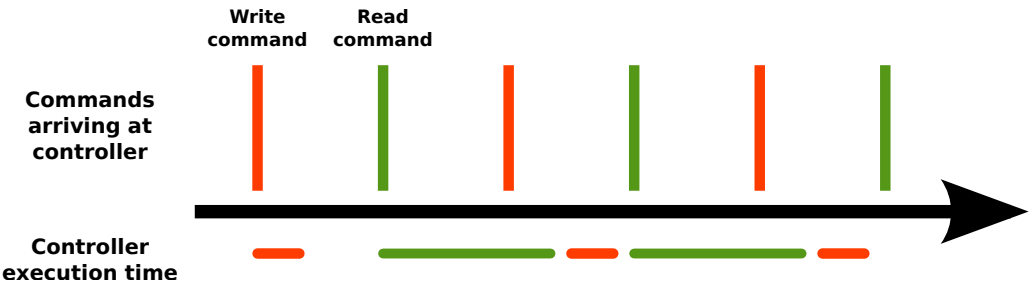

Figure 10: Illustration on measured mean delay. The upper row shows the commands arriving at the controller with a fixed delay and in the lower row the time the controller needs for execution of the commands is depicted. If the sum of delays for read and write commands is larger or equal to the sum of execution times of these commands by the controller, they can be executed without blocking the FIFO.

Considering the measurements for the PLL frequency of 125 MHz it can be demonstrated that the delays are significantly higher than for the PLL frequency of 250 MHz. However, their general order stays the same. Compared to the PLL frequency of 250 MHz, the inter-controller differences are higher than before, the repeater delay for example is now 7 FC greater than the switch SRAM delay. The additional discrepancy between the delay differences for the 125 and 250 MHz repeater and switch delays ($7 - 5 = 2$ FC) is probably caused by asynchronous clock domain crossing (cf. *Grübl* [2016]). This can cause the signal, which is edge aligned with the clock in one domain to arrive in the middle of one clock cycle in the other domain. Its influence on the measured delay is dependent on the ratio of the frequencies at the domain borders, which changes when changing the PLL frequency.

Because currently the PLL frequency is not stored as a variable in `hicann-system` it is not possible to implement the PLL frequency-dependent delays as such. The default delay would be set per PLL frequency as the maximal measured delay to avoid data loss. When now choosing a default delay for all PLL frequencies, one would take the maximum of all these values to ensure the correct timing. Further thoughts on this subject can be found in chapter 5. The aforementioned maximal delay value (per PLL frequency) is also the one used in the analysis of the PLL frequency-dependency of the delays, which is carried out in chapter 4.3.4.

4.3.3 Controller access time tests

The tests for measuring the delays used in `SynapseControl` and `SynTrans` send data to the controller's buffer register SRAMs to be written and then send the write and read commands to the controller itself. A specific time after each one of these controller commands is sent, the status

register is queried to check if the controller is still busy. Should this be the case, this iteration of the test is aborted. Because the execution of these tests took much longer than the register SRAM tests (dominated by the many JTAG accesses needed to start the PbMem module), they were only repeated 5 times per value for the controller access time. The time between the controller's write or read command and the busy check of the controller (named `controller_timeout`) is then varied.

Because the measurements for the controller access times are not based on commands getting dropped and not showing up in the return data and the fact that the controllers are connected to a pipelined OCP slave, they are not affected by the HICANNARQ problems mentioned in chapter 4.3.2.

In *Schemmel et al.* [2015] the theoretical worst case latency for a synapse array operation (the worst case operation being `START_READ`) is documented to be 34 controller cycles. Because the controller's clock is generated as $\frac{1}{4}$ of the PLL frequency (which at a PLL frequency of 250 MHz equates to $62,5$ MHz) one controller cycle equates to 2 FPGA cycles (because the FPGA clock has a frequency of 125 MHz; cf. chapter 2.1). Therefore, the value of the worst case controller latency is equivalent to 68 FPGA cycles. Because the test only has one `controller_timeout` parameter for all commands, it is expected to measure a delay value close to this theoretical worst case expectation.

In figures 11 and 12 the results of the measurements of controller access times are shown. Comparing the measured value for 250 MHz of 70 FC for `SynapseControl` to the theoretical value of 68 FC it is clear that the measurements are very close to the expectation. The discrepancy of 2 FC can probably be explained again by asynchronous clock domain crossing. However, it is not immediately clear why there should be delays, at which only some of the data is read back correctly and not all of it. This could be explained by the duration of this operation being dependent on the location of its target. Because multiple different rows are read out one by one in this experiment, one could assume that some rows for which this operation does not take as long are read out first and return valid data. Afterwards others (for which this operation takes longer) are read out, which causes the controller to still be busy when checked.

As can be seen in figure 11, the values for the delays of `SynapseControl` and `SynTrans` are identical and the rest of the data is very similar as well. Upon further investigation of this phenomenon and continued communication with A. Grübl it was discovered that the two hardware abstraction classes `SynapseControl` and `SynTrans` actually access the same controller. It therefore would be expected to obtain similar results.

The value of the measured minimum `controller_timout` for stable execution is clearly PLL frequency dependent, as the value for 125 MHz is at 140 FC significantly higher than the one for 250

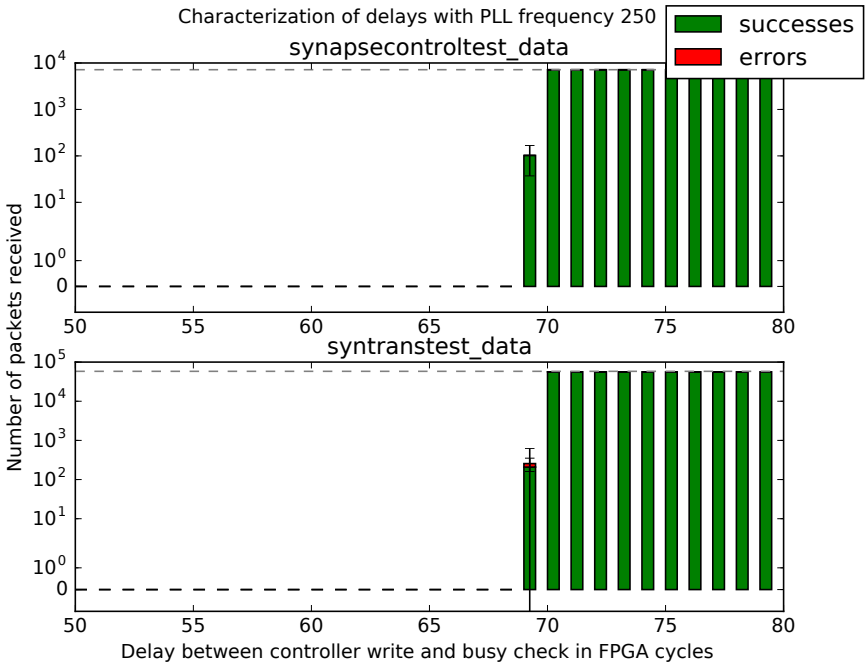

Figure 11: Results of the controller access time characterization for PLL frequency of 250 MHz. The basic structure of the plot is the same as in figures 8 and 9. The data that is read back correctly is shown in green, the incorrect data in red and all values were averaged over the 5 repetitions and are displayed with their error. Values for the stable delays are in both cases 70 FC.

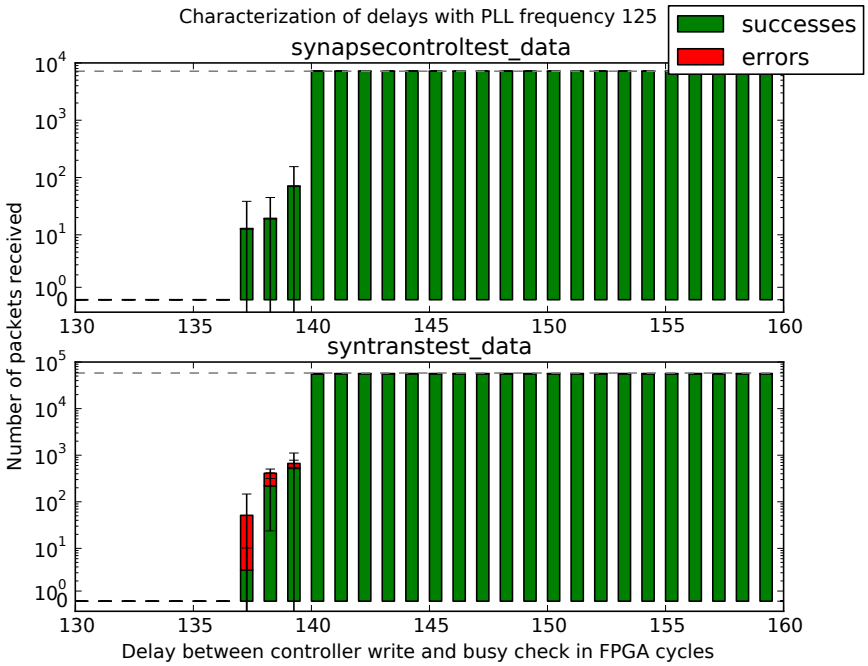

Figure 12: Results of the controller access time characterization for PLL frequency of 125 MHz. Here the same tests as in figure 11 were executed for a different PLL frequency. The stable delays for both controllers increase to 140 FC, which is double the value of 70 FC measured for 250 MHz.

MHz. This is because the controller's clock frequency is (as mentioned above) directly derived from the PLL frequency. The linearity of these values is analyzed in chapter 4.3.4.

4.3.4 Linearity

Due to both, the packet rate on the HICANN bus and the clock frequency of all controllers being defined as $\frac{1}{4}$ of the PLL frequency, it would be expected to observe a linear correlation between the two delays and the PLL frequency.

Figure 13 shows the maximal delay and `controller_timeout` values for each PLL frequency. When taking the clear outliers at 50 MHz into account, in both cases no linearity at all is visible. These deviations are caused by the HICANNARQ parameters being far away from their optimum. In particular the `tx_timeout` is too small for the large RTTs at such a small HICANN bus transfer rate and causes unnecessary resends, which stall the FPGAOut FIFO and cause commands to be dropped. This should increasingly worsen with higher RTT and indeed, when looking at the measurements for the repeaters at 50 MHz (cf. figure 19 in appendix A.2), it is evident that the repeater test (which has the maximum RTT due to the long controller access) is never completely stable. This is an indication to avoid using the frequency of 50 MHz in experiments.

However, even after discarding the values for 50 MHz it is difficult to see a clear linear relationship in either case, as all values systematically deviate upwards of a hypothetical linear function. This can possibly be explained by HICANNARQ parameters like the `tx_timeout` staying constant whilst the PLL frequency changes. Because the RTT of the HICANN bus is proportional to the increasing PLL frequency, these parameters move progressively away from their optimal (RTT dependent) value. This results in more resends by the HICANNARQ, which causes the value of the first stable delay to increase. However, to effectively back up this hypothesis further measurements would be required.

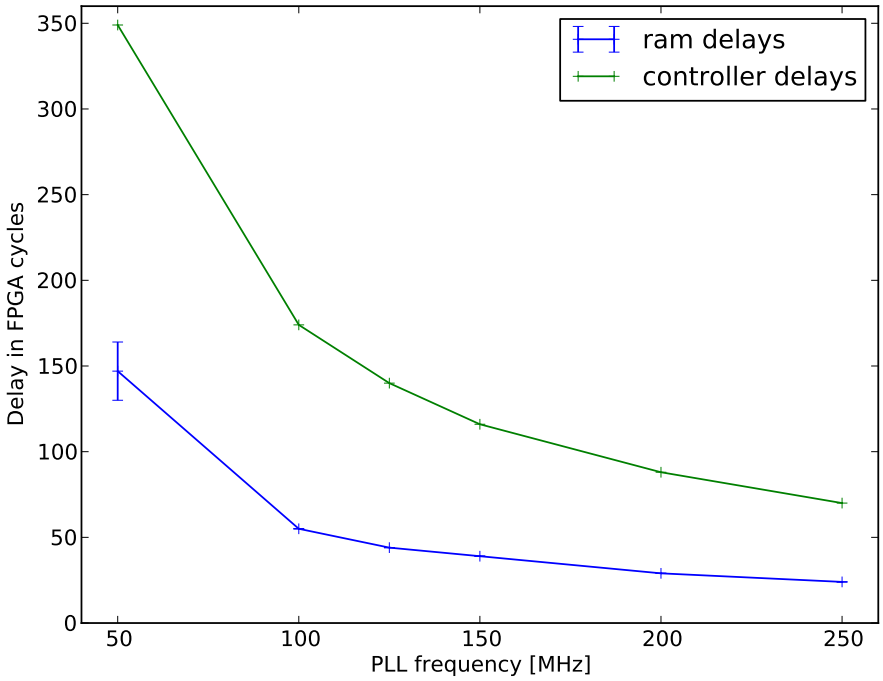

Figure 13: Analysis of the PLL frequency dependency of the delays and controller access times. The maximum delay of the register SRAMs and the controller timeouts is plotted against the PLL frequencies. The values for 50 MHz are clearly anomalous (cf. text for further details).

5 Discussion & Outlook

The goal of the first part of this thesis as presented in chapter 3 was to implement a PbMem communication module and integrate it into the software stack. This goal was achieved, as the integration into `hicann-system`, `HALbe` and `stHAL` was completed. To ensure correct configuration, the communication module was implemented with the delays determined in chapter 4. Its functionality and stability have been tested using multiple low-level hardware tests from `hicann-system` and `HALbe`. In the near future, the resulting PbMem communication module will replace the asynchronous communication module as the standard for `HALbe` and `stHAL` tests. This will speed up experiments significantly whilst making them more robust and reproducible.

The implementation of the communication module's `sortAndSend()` method could be further optimized by first parsing the PbMem program for grouped commands and then inserting these into the frame buffers as a group instead of parsing it on a per-command basis. This would supersede the backtracking to update the group header (as the size is known beforehand) and would eliminate the necessity of holding two frame buffers simultaneously. Another optimization might be, to look for alternative sorting algorithms, which converge faster for lists very likely to already be in the right order (as is the case with the PbMem program). A possible candidate might be `insertion sort`.

Based on a suggestion made by the author, the FPGA functionality was changed to allow starting of the PbMem module without having to synchronize the system time counters and reset the HICANN time counters every time. This now is performed once in the main initialization method. Afterwards only the experiment start will be triggered without the necessity of the slow JTAG access. However, this modification to the FPGA was only just ready before the end of this thesis and so this functionality was implemented, but could neither be thoroughly tested nor used in the tests mentioned in chapter 4.

For the future it is planned to improve the receive functions, making them more time-efficient and robust by for example using the new data containers. These new containers will also replace the currently existing `PulseEvent` and `PulseEventContainer` classes to avoid multiple representations of essentially the same information. Long-term, it is planned to improve the architecture of `hicann-system` by separating `reticle_control` and `hicann_control` from all hardware control classes and deleting `[fpga,dnc]_control`. New functionalities like a user-friendly interface for setting and storing the current PLL frequency will be integrated to enable PLL frequency-adaptive delays for the hardware control classes.

The second part, presented in chapter 4, focussed on benchmarking the PbMem communication module and characterizing the hardware component-dependent inter-command delays. The results

from the benchmarks (cf. figure 6) were as expected, the PbMem communication module was asymptotically significantly faster than the asynchronous module. It seems possible, to further increase the benefit of the PbMem communication module by optimizing its receive functions.

The results of the experiments conducted to characterize the delays and controller timeouts were ambivalent. The measured values for the controller timeouts were very close to the expectations, which proves the PbMem very useful for exact and timing-critical measurements. The achieved changes speed up the configuration of the synapse controller from ≈4 ms per row using the asynchronous communication to 1.4 µs per row[1] in the PbMem program, a $\mathcal{O}(10^3)$ improvement. When characterizing the memory delays however, it became apparent that the HICANNARQ and its parameters have a major impact on these measurements. After all, even though the HICANNARQ was never built for real-time experiments and measurements it performed surprisingly well. To reduce the number of static, user-defined variables, it would be advantageous to implement a self-adjusting timeout which is dependent on the current transmission rate and expected HICANN bus RTT into the HICANNARQ of the newer HICANN versions. This is currently being investigated by V. Karasenko. However, to improve the characterization in the current HICANN version, the register SRAM delays have to be split into a write and read command delay and have to be controlled in connection with the HICANNARQ timeout parameters.

When decreasing the PLL frequency, the experimentally determined values deviated upwards from the theoretically expected linear relationship. This can be explained by the non-PLL dependent HICANNARQ parameters (e.g. `rx_timeout`), which go increasingly out of optimum and cause many resends, increasing the measured delays. In the case of a PLL frequency of 50 MHz it actually causes significant instabilities of the repeater SRAM test, suggesting to avoid this specific frequency or else to change HICANNARQ parameters. However, since tuning these parameters was not the focus of this thesis, this remains open for further investigation. Because the delays cannot be implemented depending on the PLL frequency (as stated above), for now all default delays have been set to the values for 100 MHz, as these were the highest values for a stable execution.

For future optimization, the `controller_timeout` variable could be split into multiple different ones based on the worst case latencies of the controllers for these operations. Other improvements concerning the PbMem operation are on their way. The FPGA PbMem module will be enhanced to include FPGA configuration packets as a new type, which can not only trigger certain actions like stopping the PbMem and trace module but can also mark the end of the program. These 'End Of Program' packets get parsed when they are loaded into the PbMem by the memory controller and trigger a 'ready' packet to be sent to the host. This is useful to ensure the complete program has been loaded into the PbMem module before starting it, which could currently be a source of

[1] The standard delay for `SynapseControl` is set to the value for 100 MHz (175 FC). As one FPGA cycle equates to 8 ns, the delay for one command (writing one row) is 1.4 µs.

random errors. To enable constrained random tests of the software PbMem module, a HICANN configuration loopback should be implemented in the FPGA. To structure the return data from the chip, HICANN configuration responses should also be timestamped and stored in the trace module instead of being directly sent to the host as currently done. Looking further into the future it is also planned, to implement random access to the PbMem module to support the execution of multiple experiments in succession, as well as parameter sweeps, where only one packet in the program is being changed. All this of course requires software support and continued development, for which the PbMem communication module is well equipped.

These changes to the FPGA also have to go hand in hand with the new generation of HICANN called HICANN-DLS. As a completely new software stack is required for this revision, it would be very beneficial to exclusively support PbMem operation, because this mode of communication is robust and increases the reproducibility of experiments. The techniques employed in this thesis could then be carried over and lay a solid foundation for the new software.

Appendix

A Appendix

A.1 Repository Listing

The following section aims to provide a complete overview about the code used and implemented in the course of this thesis. There are two different workspaces. One workspace is used for testing and the other one is the final version, which can be integrated in the software stack. The repositories can be found at: `https://brainscales-r.kip.uni-heidelberg.de/`. As all changes are still under review, their Gerrit changesets can be found at: `https://brainscales-r.kip.uni-heidelberg.de:9443/`.

Repository	SHA 1 ID
bitter	`da89a7ece461ad6e97a6d7f9e9c0a5b9fccafbff`
calibtic	`68b6ef203cca6eb687d2d05743808997a604ddb9`
euter	`ae12b24ed5e73cff22616296a552efb365743e43`
lib-boost-patches	`1bbccae8107e511f94650c8e13857df5eb7506ef`
lib-rcf	`de6fa72d55c186bd89aefd0e1c6b90e4c99323a0`
logger	`3da08bd6fb00b239a59abde42983018fe0386bd5`
pygccxml	`8ae9e19ae00c4152fa5a381eb9e663561c07345f`
pyplusplus	`2fe3b869191acb547afaa33e2112fa83e41e79d5`
pythonic	`f2f162e34d7b024e0de79c55133fef00e82fffe0`
pyublas	`9f707f60320e20e5a7714d920ba0530d092e1310`
pywrap	`c375009b6950d6b41b59d71b7f4b202df99f8117`
rant	`4a8acd076fb9531ce61a990ef0f414935886d85d`
redman	`42d28037a0bcdf1e70c7bbb6734411de63174b3f`
sctrltp	`42c6d0ea49f91dc44a7aae1017cf4cd935246834`
sthal	`b6a4f583b318ff639dce5e782ea073f9d3fa52fb`
vmodule	`210885997f0124975cb17cb6710d5fffec585513`
ztl	`068c18233337711e40027aa51dc667f7ed6cdcd8`

Table 2: Git Hashes of HEAD state for all supporting repositories. Both workspaces are based on these versions.

The following Gerrit changesets provide the working repository state used for testing.

- hicann-system: change ID `Ief8afd94c20d757dd369e0f08d5e6b21d26fae23`
  - working standard tests: commit hash `07f25ea00775e4edc5ea60fd7e72b8f1cafe9868`
  - tests provoking FIFO overflow: commit hash `c7f0df791b84cb663580d0e7c5497e4c9ceb8656`

The results of these tests and scripts used for benchmarking and data analysis can be found in `/ley/users/lpilz/results_bachelor/`.

These Gerrit changesets provide the working repository state planned for upstream integration

- hicann-system:
  - without fast experiment start: change ID `I4b49d1f042a5b0994a4b1356c7bb95fb57afd654`
    - commit hash `01f9ee251ca15e16f4c2aeed842e9fbd598b1a36`
  - with fast experiment start: change ID `Iacc6d1343147bb929becddc6d59270418dfb09c6`
    - commit hash `db3d887a2cb073a891ed7682436dc5d93e9583c9`
- halbe: change ID `Ib3c9c1d94e6344b5ac31ed0be5f0d51ac090cb0d`
  - commit hash `4a3f547222f0351079dc0269f42f26746191e294`
- sthal: change ID `d8c7fc3efb59283f3ea4cb5937dbf8c5dde4e075`
  - commit hash `38ea09bb33dc2495b2fd5155f09ecd72cc40959d`

A.2 Image Appendix

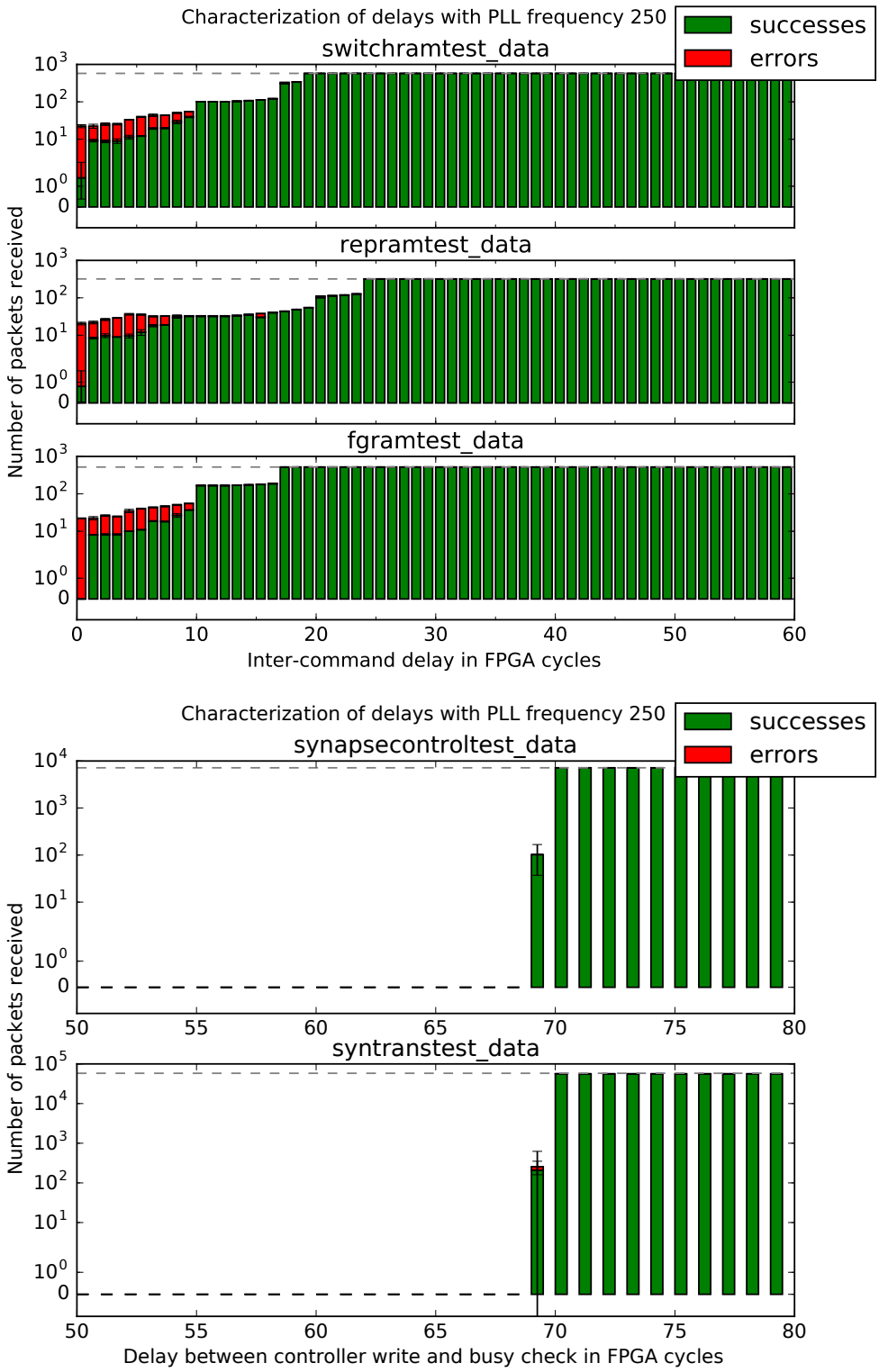

Figure 14: Results of delay characterization for PLL frequency of 250 MHz

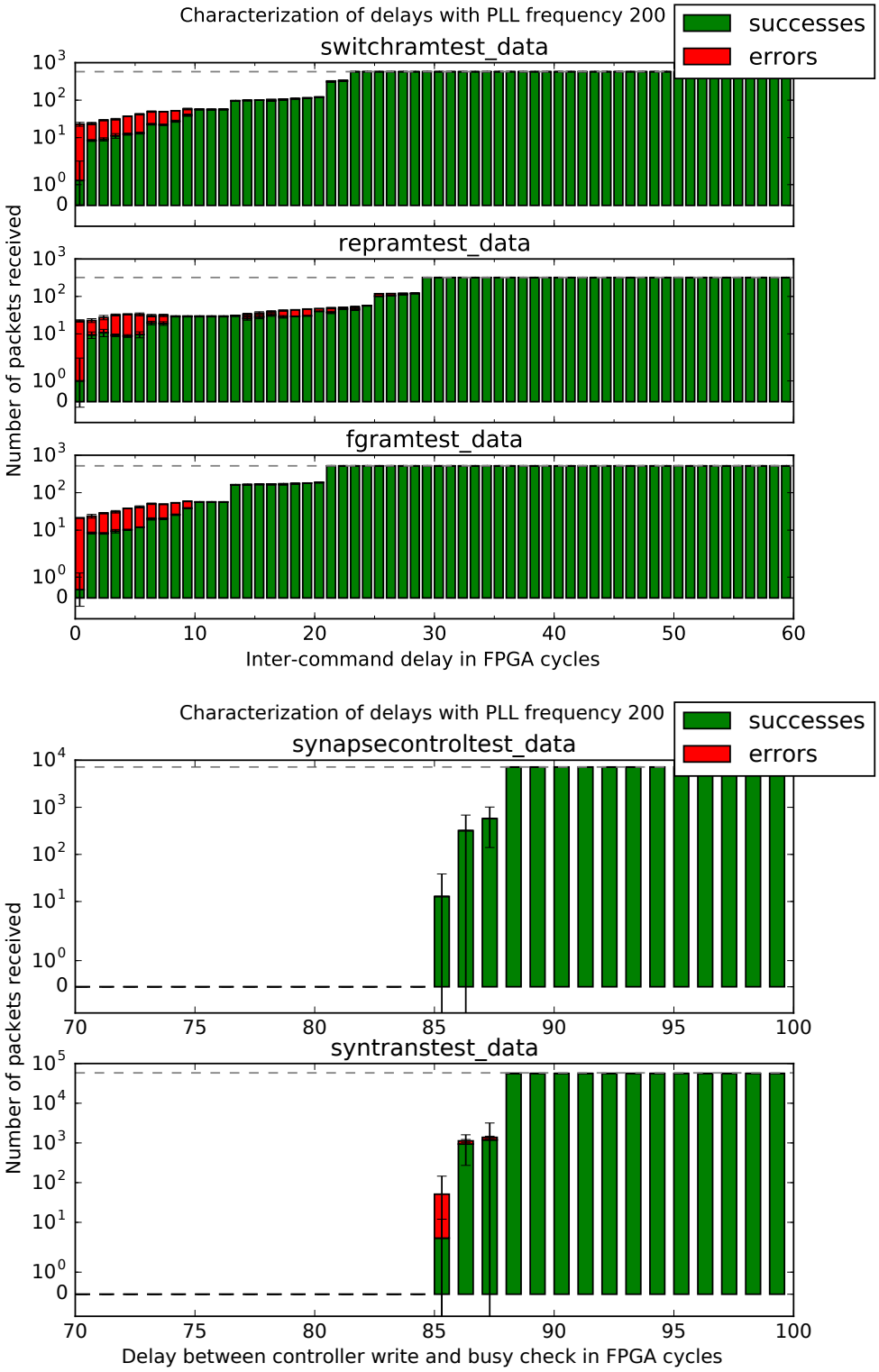

Figure 15: Results of delay characterization for PLL frequency of 200 MHz

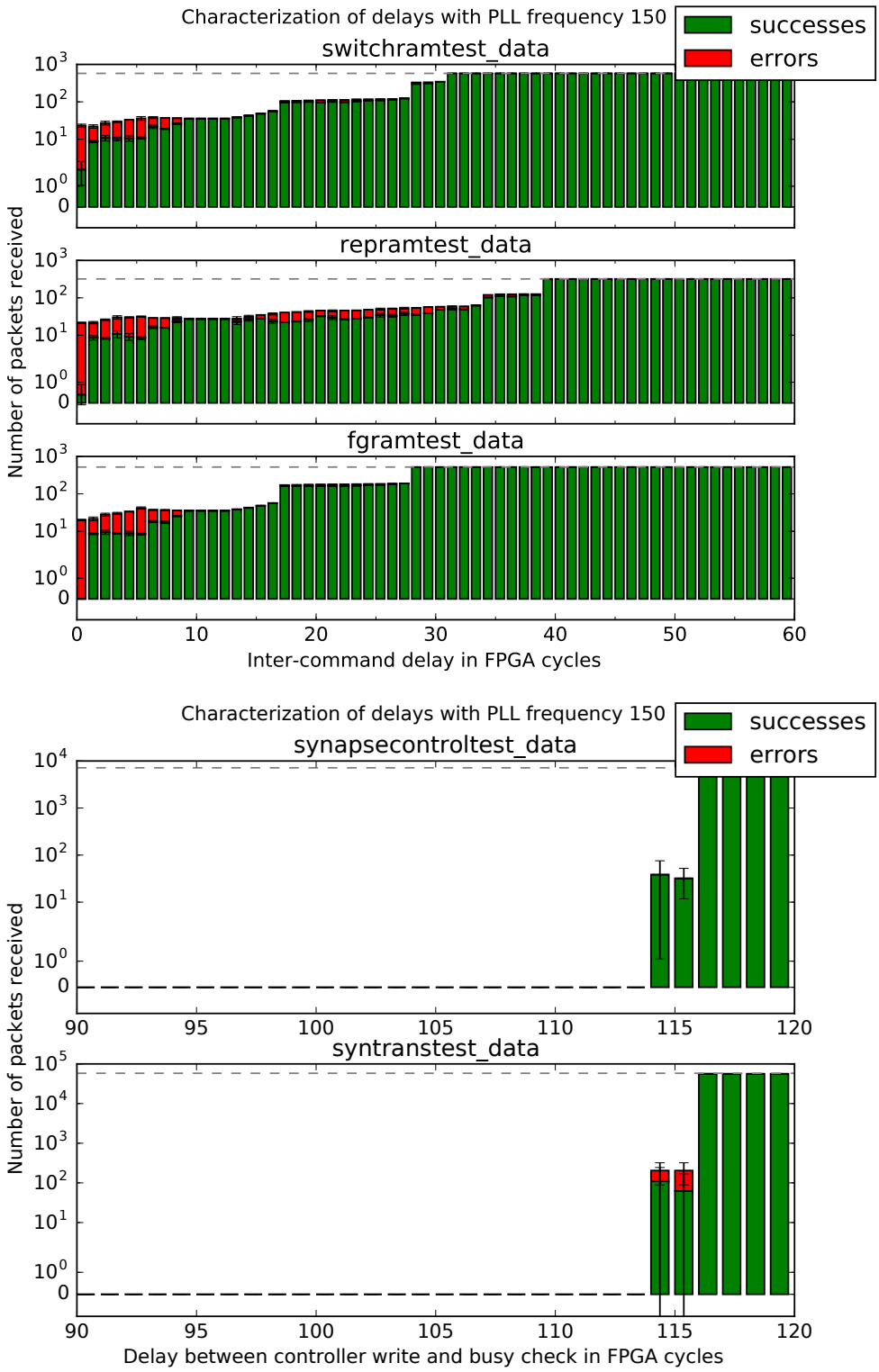

Figure 16: Results of delay characterization for PLL frequency of 150 MHz

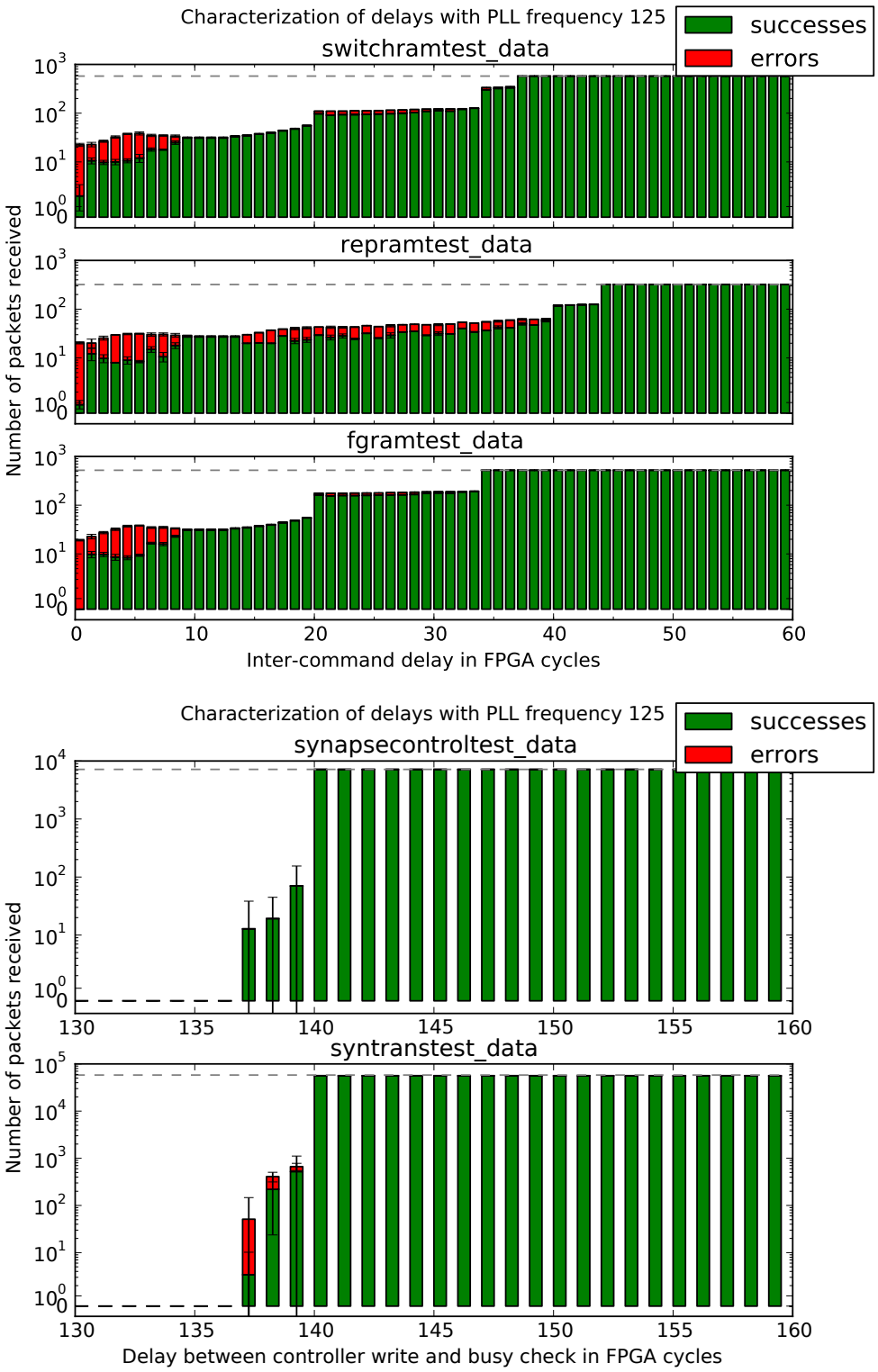

Figure 17: Results of delay characterization for PLL frequency of 125 MHz

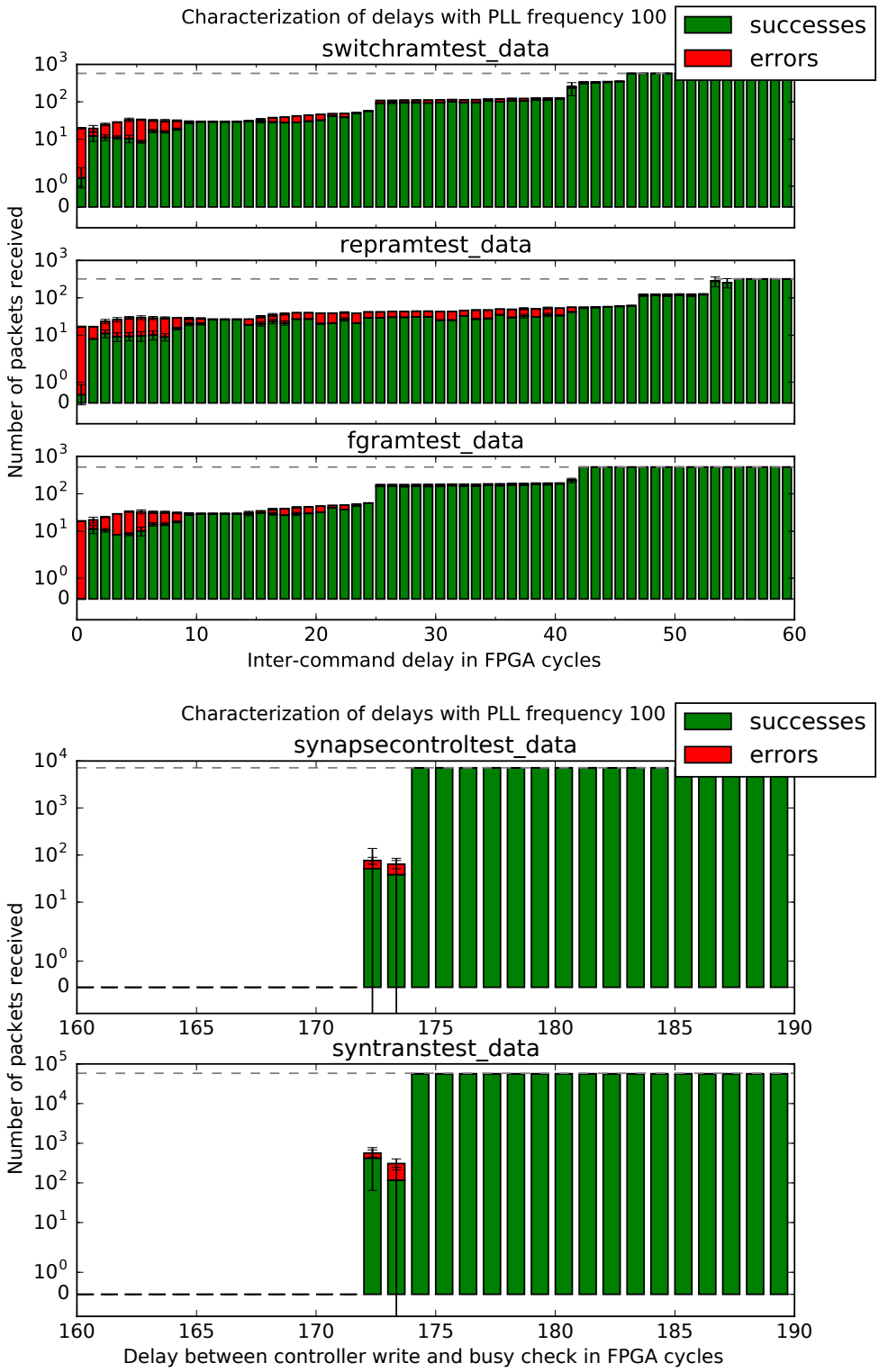

Figure 18: Results of delay characterization for PLL frequency of 100 MHz

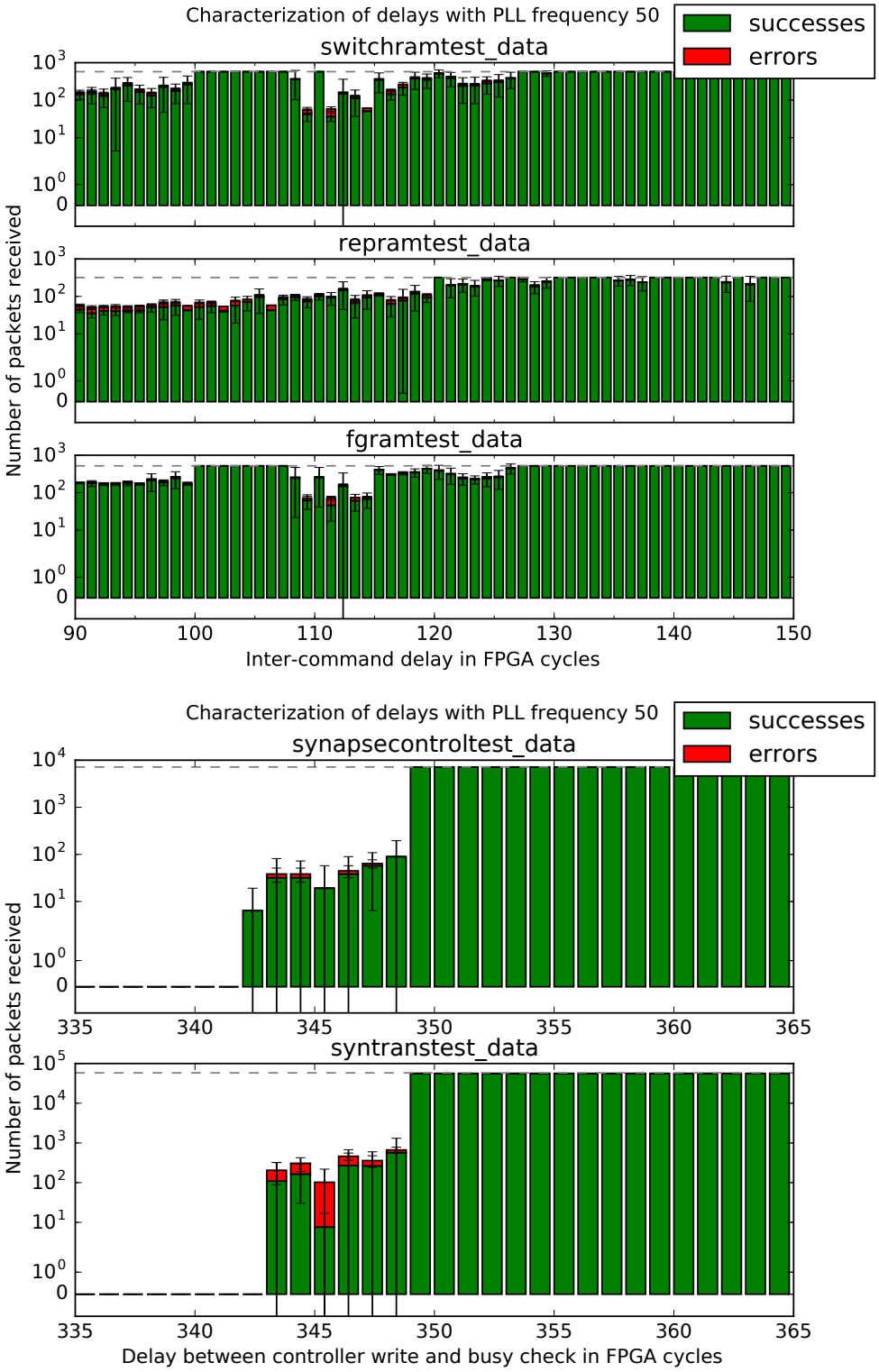

Figure 19: Results of delay characterization for PLL frequency of 50 MHz

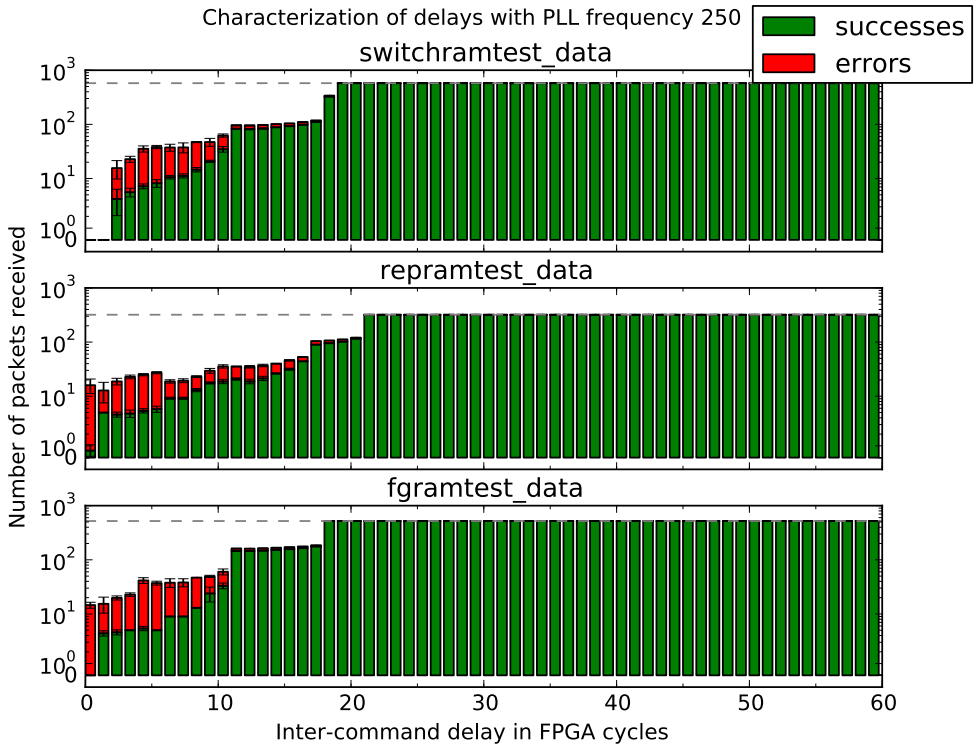

Figure 20: This test was executed at the PLL frequency of 250 MHz and differs from the other tests in so far as three or four write commands were sent to one SRAM cell (depending on the amount of cells per block). This was done in order to reliably overflow the HICANNARQ FIFOs by increasing the overall number of commands sent. More details concerning this experiment can be found in chapter 4.3.2

Glossary

ACK acknowledgement information. 5–7, 21, 23

ADC Analog-to-Digital Converter. 4

AdEx Adaptive Exponential Integrate-and-Fire. 4 4.

API Application Programming Interface. 8, 10, 13

. 21, 23, 27, 28, 30, 48

CPU Central Processing Unit. 2

DC DNC cycles. 5, 23

DDR3-SDRAM Double Data Rate Synchronous Dynamic Random Access Memory. 7

DNC Digital Network Chip. 4–7, 10, 12

FC FPGA cycles. 5, 7, 23, 26, 27

FEXT far-end crosstalk. 5

FIFO first in, first out. 6, 7, 21, 23, 26, 48

FPGA Field-Programmable Gate Array. 3–5, 7, 10–16, 18, 19, 21, 23, 29–31

HALbe Hardware Abstraction Layer Backend. 8, 10, 12, 13, 17, 29

HICANN High-Input Count Analog Neuronal Network Chip. 3–7, 9, 11–18, 21, 23, 26, 29–31

HICANN-DLS HICANN-DLS. 30

HICANNARQ HICANN ARQ protocol. 5, 7, 13, 21, 23, 30

HostARQ Host ARQ protocol. 7–9, 15

JTAG Joint Test Action Group. 4, 9, 13, 16, 19, 29

L1 Layer 1. 4–6, 18, 20, 21, 23

LIF Leaky Integrate-and-Fire. 2, 4

marocco marocco. 8

NM-PM Neuromorphic Physical Model. 9

NM-PM1 Neuromorphic Physical Model version 1. 4

OCP Open Core Protocol. 5, 23

PbMem Playback Memory. 3, 7, 9–20, 23, 26, 29, 30

PLL Phase-Locked Loop. 5, 21–30, 42–48

PMOS Positive Metal-Oxide Semiconductor. 4

PyHMF PyNN for the BrainScaleS Hybrid Multiscale Facility. 8

PyNN PyNN. 8

RAM Random Access Memory. 3

RTT Round Trip Time. 7, 27

SpL1 Synchronous Parallel Layer 1. 4, 6

SRAM Static Random Access Memory. 5, 6, 13, 18, 20–23, 26, 28, 30, 48

STDP Spike Timing Dependent Plasticity. 6, 20, 21

stHAL Stateful Hardware Abstraction Layer. 8, 12, 29

UDP User Datagram Protocol. 7

B Bibliography

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.

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.

Debus, J., Comissioning of an FPGA-based prototyping environment for neuromorphic hardware, Bachelor thesis, Ruprecht-Karls-Universität Heidelberg, 2016.

Furber, S., Large-scale neuromorphic computing systems, *Journal of Neural Engineering Std 1741-2560*, doi:10.1088/1741-2560/13/5/051001, 2016.

Grübl, A., personal communication, 2016.

HBP SP9 partners, *Neuromorphic Platform Specification*, Human Brain Project, 2016.

Jeltsch, S., A scalable workflow for a configurable neuromorphic platform, Ph.D. thesis, Universität Heidelberg, 2014.

Karasenko, V., A communication infrastructure for a neuromorphic system, Master's thesis (English), University of Heidelberg, 2014.

Karasenko, V., personal communication, 2016.

Klähn, J., Tuning of functional networks on neuromorphic hardware, Masterarbeit, Universität Heidelberg, in preparation, 2016.

Koke, C., Device variability in synapses of neuromorphic circuits, Ph.D. thesis, Universität Heidelberg, in preparation, 2016.

Mauch, C., Commissioning of a neuromorphic computing platform, Masterarbeit, Universität Heidelberg, 2016.

Müller, E. C., Novel operation modes of accelerated neuromorphic hardware, Ph.D. thesis, Ruprecht-Karls-Universität Heidelberg, hD-KIP 14-98, 2014.

Painkras, E., L. A. Plana, J. Garside, S. Temple, S. Davidson, J. Pepper, D. Clark, C. Patterson, and S. Furber, Spinnaker: A multi-core system-on-chip for massively-parallel neural net simulation, in *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference*, pp. 1–4, doi:10.1109/CICC.2012.6330636, 2012.

Schemmel, J., A. Grübl, S. Millner, and S. Friedmann, Specification of the HICANN microchip, FACETS and BrainScaleS project internal documentation, 2015.

Stroustrup, B., *The C++ programming language*, 4th ed ed., Online-Ressource (xiv 1346 p.) pp., Addison-Wesley, Upper Saddle River, NJ, 2013.