RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Sebastian Jeltsch

# Computing with Transient States on a Neuromorphic Multi-Chip Environment

Diplomarbeit

HD-KIP-10-54

KIRCHHOFF-INSTITUT FÜR PHYSIK

# Department of Physics and Astronomy
## Heidelberg University


**Diploma thesis**

in Physics

submitted by


**Sebastian Jeltsch**

born in Ludwigsburg, Germany


**October 2010**

# Computing with Transient States on a Neuromorphic Multi-Chip Environment

**This diploma thesis has been carried out by Sebastian Jeltsch at the**
Kirchhoff Institute for Physics
Ruprecht-Karls-Universität Heidelberg
**under the supervision of**
**Prof. Dr. Karlheinz Meier**

**Computing with Transient States on a Neuromorphic Multi-Chip Environment**

The work presented in this thesis establishes a complete framework for the exploration of arbitrary neural architectures on an accelerated neuromorphic hardware device beyond single-chip boundaries. By building upon the event distribution network of the system, the developed methods close the gap between low-level functionality and the high-level neural network description language PyNN. For this purpose, algorithms have been implemented which perform three basic steps: A mapping of PyNN neurons to appropriate hardware circuits in the multi-chip environment, a resource-optimized intra and inter-chip routing of synaptic connections as well as a parameter translation between the biological and the hardware domain. The correct functionality of the presented setup is demonstrated experimentally by the successful realization of Synfire-Chains spanning multiple chips. Furthermore, an attractor-free computing paradigm based on a self-stabilizing neural network architecture is investigated on the neuromorphic system. A spike-based classifier adapted from the tempotron scheme is trained to evaluate the emerging transient network dynamics. Although this training is performed in a pure software environment merely mimicking hardware-specific constraints, a direct mapping of the tempotron to actual hardware yields remarkable qualitative and quantitative matching with the software prototype. The documented work provides a foundation for the fully hardware-integrated realization of continuous computing and classification concepts like Liquid State Machines.

**Berechnungen mittels transienter Zustände in einer neuromorphen Multi-Chip-Umgebung**

Die vorliegende Arbeit schafft die methodischen Voraussetzungen, welche erstmals erlauben, über die Grenzen einzelner Chips hinaus beliebig strukturierte neuronale Netzwerke auf einem beschleunigten neuromorphen Hardwaresystem zu untersuchen. Dazu wird, aufbauend auf einer Transportschicht zur Weiterleitung von Aktionspotentialen zwischen solchen Chips, eine Verbindung geschaffen zwischen der hardware-spezifischen Ansteuerung dieser Funktionalität und PyNN, einer abstrakten Beschreibungssprache für neuronale Netzwerke. Zu diesem Zweck wurden Algorithmen entwickelt, die vornehmlich drei Aufgaben erfüllen: Das Abbilden von PyNN-Neuronen auf entsprechende Schaltkreise der Multi-Chip-Plattform, die ressourcenoptimierte Verschaltung der synaptischen Verbindungen auf und zwischen den einzelnen Chips, sowie die Übersetzung aller Parameterwerte von der biologischen Beschreibung ins mikroelektronische Modell. Die korrekte Funktionsweise des kompletten Systems wird anhand einer Synfire-Chain, die sich uebere mehrere Chips erstreckt, experimentell belegt. Außerdem wird die neuromorphe Implementierung eines attrakorfreien Berechnungsparadigmas untersucht, das auf einer sich selbst stabilisierenden Netzwerkstruktur basiert. Dafür wird ein von einem Tempotron abgeleiteter, nur mit Aktionspotentialen arbeitender Klassifizierer für die Auswertung der instationären Netzwerkzustände trainiert. Obwohl das Training in einer reinen Softwareumgebung stattfindet, in der bestimmte Hardwarebeschränkungen nachgeahmt werden, zeigt eine direkte Übertragung des trainierten Klassifizierers auf Hardware eine bemerkenswerte qualitative und quantitative Übereinstimmung mit der Softwarevorlage. Damit schafft diese Arbeit die Grundlagen für eine vollständig hardware-integrierte Umsetzung von kontinuierlich arbeitenden Berechnungs- und Klassifizierungskonzepten wie zum Beispiel Liquid State Machines.

# Contents

# Introduction

## Towards Neuromorphic Hardware

> So that I may perceive whatever holds
> The world together in its inmost folds.
>
> (*Faust*, J. W. v. Goethe 1808)

Long before the cornerstone was laid for modern science, man tried to understand his surrounding nature. It is not merely by chance that the protagonist of Goethe's drama is a scientist struggling to reason the basic principles holding the world together. Despite continuously accelerating advances towards understanding our universe, to this day we still remain remarkably nescient about the sapience of Homo Sapiens itself.

With the dawn of the early computer age in the late '30s and early '40s of the past century, the answer to all these questions seemed within reach. Artificial "thinking machines" were envisioned in the endeavor of understanding the original thinking machine, the human brain. Pioneers like John von Neumann, Alan Turing or John McCarthy, encouraged by this ambitious goal, led the field evolving at a tearing pace. By building on the constant developments in technology, enormous progress was made in various fields like game solving, map navigation or speech recognition. However, when it comes to real world tasks like object recognition or complex body movement, even the most sophisticated machines fall behind their biological rivals. While software and algorithms are continuously improving, the miniaturization of hardware, described impressively by Moore's law, is expected to reach a natural barrier soon. With the individual in-silico layers of transistors reaching thicknesses of only a few atoms, the production of reliable components becomes increasingly difficult. While the search for alternative architectures and manufacturing processes is ongoing, evolution has found an extremely efficient solution to this problem long ago: the brain. If we were able to re-engineer the structure of the biological template, this would certainly revolutionize our contemporary concept of computing devices. In any case, it would be a breakthrough in the way of testing and implementing our hypotheses about neural algorithms.

Neuromorphic hardware development attempts a bottom-up approach at reverse-engineering the brain. The basic biological building blocks are translated into real physical circuitry, leading to a system of artificial neurons and synapses evolving simultaneously and asynchronously in time. With the system computing itself, the time required for the emulation of a given network does not scale with its size. Furthermore, due to the size of the components and the intrinsic properties of silicon, neuromorphic hardware can offer a significant speed-up factor with respect to biological real-time. This becomes particularly relevant when studying for example long term plasticity phenomena. Last but not least, neuromorphic devices can easily excel modern processors in a very brain-like fashion when it comes to power consumption.

## The FACETS Project

With the neuromorphic concept in mind, the FACETS (Fast Analog Computing with Emergent Transient States) project has united scientists from various fields of research in an effort to create a foundation for the realization of novel computing paradigms inspired by observations of biological neural systems. Enabled by a strong theoretical and experimental background offered by the formulation of models on various scales, from individual neurons and synapses up to large, cortically-inspired networks, based on in-vivo and in-vitro measurements, different types of neuromorphic hardware have been developed within the FACETS community. While one approach concentrates on a very accurate reproduction of biological structures, with the possibility of interfacing the system to real, biological neural networks, the other approach emphasizes versatility and scale. The neuromorphic hardware developed by the Electronic Vision(s) group in Heidelberg together with the TU Dresden aims towards the wafer-scale integration of multiple building blocks containing neurons and synapses, with a particular focus on enabling almost arbitrary connection patterns between all the neurons across the wafer. While this system is still in its development phase, individual chip prototypes are currently available for experimentation.

## The Problem of Scaling

Although neuromorphic hardware offers many advantages for the emulation of neural networks, several drawbacks exist. Many of them relate to distortions caused by imperfections in the hardware manufacturing process or by design-inherent limitations. However, one of the strongest limiting factors is represented by the physical size of the hardware itself, which in turn limits the number of available neural components. In order to overcome this barrier and enable the emulation of arbitrarily large networks, realizable scaling solutions need to be explored. The thesis at hand focuses primarily on resolving the scaling problem for a chip-based FACETS prototype neuromorphic system by integrating multiple such chips in a single, coherent, easy-to-use setup, while maintaining the versatility of individual chips at the level of the entire setup. The realizability of the approach is validated by comparing results from hardware emulations to equivalent runs on established software simulators. Additionally, as a secondary focus, a particular type of network model, the Liquid State Machine, is implemented – using exclusively spiking neurons for all components, including the output classifier – on a single-chip environment. The feasibility of a multi-chip solution is analyzed, experimentally tested and discussed.

## Outline

Chapter 1 sketches the technological prerequisites for the realization of a multi-chip system. Additionally, the theoretical foundations necessary for liquid computing are provided. In the beginning of Chapter 2 the fundamental concepts and necessary steps for the realization of the multi-chip framework are described. The following part, namely Section 2.2 and Section 2.3, explain the actual implementation of the mapping flow and the realization of the Liquid State Machine. Results gathered from the framework analysis and the experiments performed with the Liquid State Machine (LSM) on the hardware system are presented in Chapter 3. This includes the realization of a spike-based classifier on the actual hardware system.

# 1. Materials and Methods

This chapter gives a brief introduction to the current state of the available software and hardware components. It highlights the topics which are of special interest for the thesis at hand.

The FACETS chip-based and FACETS wafer-scale system are introduced, which are both general purpose neuromorphic hardware platforms for the exploration of neural networks. Additionally, the available wafer-scale software framework is described. It provides functionality to map arbitrary network descriptions onto large-scale neuromorphic hardware systems.

In the end the fundamental concepts for transient computation are outlined together with the basic building blocks of the Liquid State Machine.

## 1.1. The Neuromorphic Hardware System

This section describes the FACETS chip-based hardware currently available for neuromorphic experiments within the Electronic Vision(s) Group. Figure 1.1 shows the setup with its main building blocks: the Spikey chips residing on carrier boards plugged into one backplane which is connected to an operating host computer.

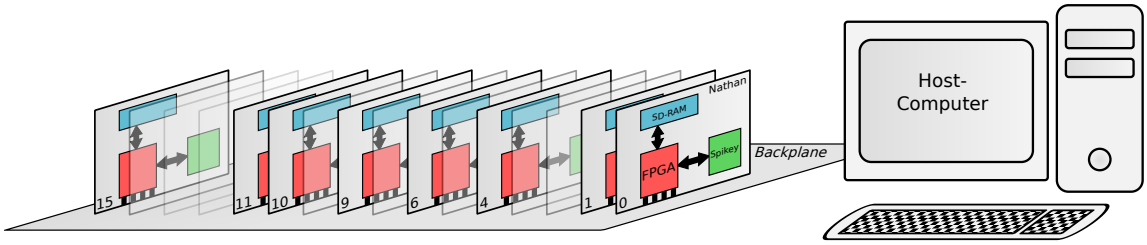


**Figure 1.1.:** The FACETS chip-based system consisting of several Spikey Artificial Neural Network chips with Nathan carrier boards on the backplane. The whole setup is operated from a host computer.

### 1.1.1. The Mixed-Signal Neural Network Chip Spikey

The mixed-signal Application-Specific Integrated Circuit (ASIC) Spikey was developed in the Electronic Visions(s) group and is available in its 4th version. The chip is fabricated in a 180 nm Complementary Metal-Oxide-Semiconductor (CMOS) process and has an area of $5 \times 5\,\mathrm{mm}^2$. A photograph of the chip is shown in Figure 1.2. The behavior of the 384 implemented Leaky-Integrate-and-Fire (LIF) neurons is emulated by analog circuits, allowing the neurons to operate in a continuous time regime. Their time evolution is accelerated by a factor of $a = 10^4$ compared to biological real time due to their shorter intrinsic time constants.

The dynamics of the membrane are described by the following set of differential equations for LIF neurons with conductance-based synapses:

$$\begin{aligned} -C_m \frac{\mathrm{d}V}{\mathrm{dt}} &= g_l(V - E_l) + \sum_j g_j(t)(V - E_e) + \sum_k g_k(t)(V - E_i) \\ V &= V_{\text{reset}} \text{ if } V > V_{\text{thresh}} \qquad . \end{aligned} \tag{1.1}$$

$C_m$ represents the overall membrane capacitance, $E_l$ the leakage reversal potential, $E_e$ the excitatory reversal potential and $E_i$ the inhibitory reversal potential. The conductance $g_l$ models a constant discharge that polarizes the membrane towards the leakage reversal potential $E_l$. The two sums on the right hand side of the equation run over each afferent synapse with its characteristic time course $g_j(t)$ for an excitatory synapse and $g_k(t)$ for an inhibitory synapse respectively. In case the membrane potential crosses a threshold $V_{\text{thresh}}$ a spike is emitted and the membrane is set back to its reset potential. The membrane time constant is given by $\tau_m = {}^{C_m}/_{g_l}$ with $\tau_m^{\text{hw}} \cdot a \approx \tau_m^{\text{bio}}$.

Each of the 384 neurons can have up to 256 inputs, resulting in a total of 98304 synapses. Each synapse implements a weight with a resolution of 4 bit and provides Spike-Timing-Dependent Plasticity (STDP) functionality [*Schemmel et al.*, 2007], [*Schemmel et al.*, 2006]. They are aligned in two grids, splitting up the chip into two cores. The axonal inputs, the so-called *synapse drivers*, occupy the space between the grids, while the neurons are aligned alongside the bottom of each grid. Each row of synapses can be individually configured to mimic facilitating or depressing efficacies by means of a shared Short-Term Plasticity (STP) feature [*Schemmel et al.*, 2007; *Tsodyks and Markram*, 1997]. Within a single core, neural connections can be arbitrarily defined and flexibly programmed. However, since every synapse driver is limited to receive input from the neuron on the same or adjacent core with the corresponding address or from an external event memory (see Section 1.1.2) the overall connection density is limited (see *Brüderle* [2009] for further detail).

The analog part is supported by a digital controller circuit residing on the lower part of the chip (see Figure 1.2). It is clocked with twice the frequency provided by the support infrastructure described in Section 1.1.2 and 1.1.3. In the following the clocks will be referred to as the fast and the slow clock, respectively.

Threshold crossings of the membrane potentials in the analog neural circuits are registered and recorded digitally. To offer a higher spike time resolution, the fast chip clock is subdivided into 16 time bins by a ring buffer locked onto the rising edge of the clock signal. For a 100 Mhz system clock (slow), this corresponds to a resolution of about $\Delta t = 3\,\mu$s biological time. However, the resolution may decrease for high event rates on the chip as only one spike can occur per time bin, so that multiple almost coincident spikes are distributed over subsequent time bins.

As a general purpose neuromorphic hardware device and due to its high configurability, Spikey can be employed as a neuroscientific modeling tool. A Python-based backend for the simulator-independent network description language PyNN, as described in Section 1.2.1, has been implemented to simplify its operation.

A complete list of chip parameters can be found in the Appendix A. For further details about the chip see *Grübl* [2007].

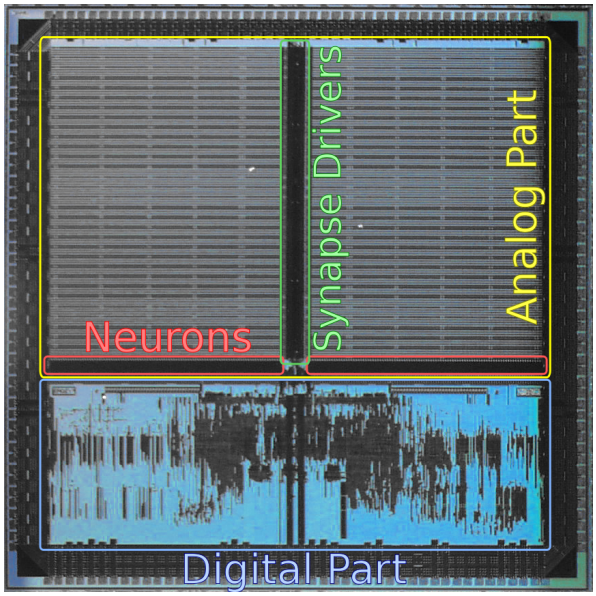


**Figure 1.2.:** A micro photograph of a Spikey (version 2) mixed-signal neural network chip. One can see the two large synapse arrays in the upper left and right, and the neurons as a dark row below each array. The irregular textured area underneath belongs to the digital support circuit.

### 1.1.2. The Carrier Board Nathan

The Nathan board is designed to support the Spikey chips with power, memory and connectivity. It is responsible for the host communication and further enables the chips to communicate with each other. The central units on the boards are Xilinx Virtex-II Pro Field Programmable Gate Arrays (FPGAs) and an off-the-shelf memory module, also referred to as *playback memory*. The memory module can stores program sequences to configure the Spikey chips and inject external stimulation. Besides, the FPGA implements communication infrastructure to the host PC and to other Nathan boards.

### 1.1.3. Backplane and Connectivity

The backplane is a custom Printed Circuit Board (PCB), which provides power, a global clock signal and connectivity for up to 16 Nathan modules. It has been developed within the Electronic Vision(s) group to host support infrastructure for chip-based neuromorphic hardware systems [*Schemmel et al.*, 2004]. Figure 1.3 shows a photograph of a backplane with three Nathan boards. The design fits in a standard 19 inch rack and is powered by a standard ATX power supply. Besides, an FPGA is provided to control the communication between Nathans and host computer. Three different bus systems are implemented: SlowControl, Gigabit Ethernet and Multi-Class Gigabit Network (MCGN), which are described in the following.

#### The SlowControl Host Communication Protocol

The *SlowControl (SC)* protocol was developed in the Electronic Vision(s) group to interconnect backplane and Nathans. The prefix “Slow” in SlowControl indicates that the protocol

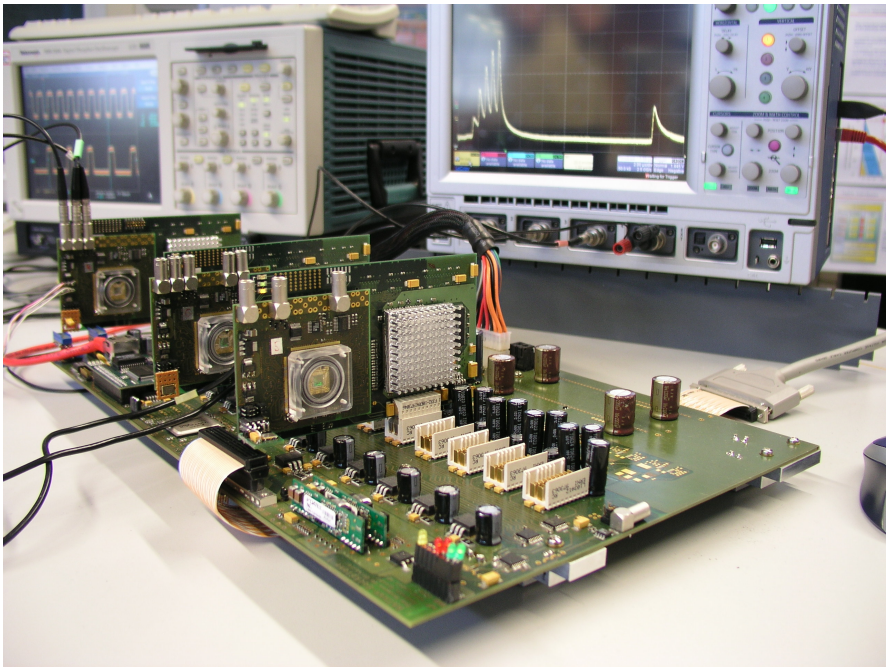

**Figure 1.3.:** A photograph of a backplane carrying three Nathan boards. On each Nathan board, below the acrylic glass lid, a Spikey chip can be seen.

was never intended for live spike distribution between different Nathan units [*Philipp*, 2008, Section 2.4]. This low-level protocol provides an interface to access FPGA registers and memory segments directly.

### The Gigabit Ethernet Host Communication and ARQ Protocol

The *Gigabit Ethernet* interface has been implemented to increase the host communication throughput and to simplify the access to the system. The hardware was until then only accessible via the custom Peripheral Component Interconnect (PCI) board in a dedicated host computer. On top of the Ethernet link layer the system uses an optimized Automatic Repeat reQuest (ARQ) protocol to ensure packet delivery and to provide low latency communication [*Schilling*, 2010] with the backplane FPGA. Consequently, in the backplane FPGA the payload is repacked in SC compliant packets. These packets are forwarded to their destination Nathan modules. Thanks to the Gigabit Ethernet interface, the chip-based system can be connected to any off-the-shelf computer or notebook equipped with a standard network interface adapter. The ARQ protocol will also be used for the communication and live interaction with the FACETS wafer-scale system (see Section 1.1.5).

### The Multi-Class Gigabit Network

The MCGN was developed in the Electronic Vision(s) group to interconnect the Nathan modules on the backplane. It is provided by means of the FPGA Multi-Gigabit Transceiver (MGT) links and is a time multiplexed network with Quality of Service (QoS) [*Philipp*, 2008, Section 1.4 and Chapter 3] functionality:

**Best-Effort Traffic** Data ready for transmission is delivered as soon as possible. Depending on the traffic load on the network nodes, this results in a variable bit rate. Features akin to lost traffic recovery [*Philipp*, 2008, Sections 3.7.1, 3.7.2] are not implemented. Hence, the network operates more efficiently and adding new nodes is inexpensive in terms of transmission overhead. This leads to a general good scalability with increasing network size.

**Priority Traffic** Spike events require fixed transmission delays and bounded jitter. The requirements are motivated by e.g. STDP neural network experiments, where reliable delays with a precision of a few ms are considered to be more critical than actual spike loss [*Morrison et al.*, 2008]. This feature is achieved via offline resource reservation and time division multiplexing. Thus the routing requires only simple table look-ups with a time complexity of $\mathcal{O}(1)$.

These two QoS features are sufficient to ensure that the transport of neural events is reliable and delays between chips are deterministic.

The fixed topology of the network on the backplane has a toroidal structure which is illustrated in Figure 1.4. The link layer is realized by impedance controlled Low-Voltage Differential Signaling (LVDS) transmission lines. Only four of the available eight MGT links connect a Nathan to its nearest neighbors. Hence not every pair of nodes shares a point-to-point connection, so that traffic between those nodes needs to be forwarded by intermediate nodes. The effective delay is proportional to the number of hops necessary to reach the final target node. This delay per hop has been found to be in the order of 180 ns [*Friedmann*, 2009].

The two sides, FPGA user logic and physical network, are accessed by means of the so-called user ports and the MGT ports. In each *time slot* within the periodic transmission frame of the time multiplexed network each input user port can be arbitrarily connected to any output MGT port, and each input MGT port can be freely interconnected to any output user port. On a fully occupied backplane any two Nathans have a maximum distance of four hops, while this distance may increase for setups with sparsely distributed Nathans on a backplane. The MCGN also provides the functionality necessary to synchronize each individual Nathan clock to a global time, allowing for a coincident triggering of experiments. The corresponding trigger signal is referred to as Global Start Signal (GSS) [*Philipp*, 2008].

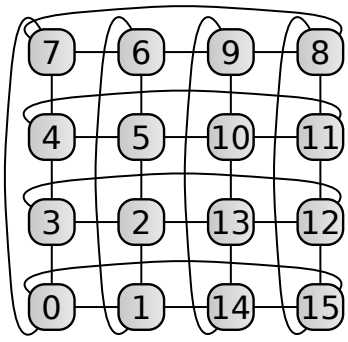


**Figure 1.4.:** Illustration of the fixed MCGN topology with toroidal structure. The available Nathan modules are represented as nodes and their MGT links as edges.

### 1.1.4. Inter-Chip Spike Distribution Network

The Event Network (EVNET) is implemented in the FPGA logic on top of the isochronous MCGN Network, to route neural events with programmable delays digitally from one chip to another [*Friedmann*, 2009]. The network provides the necessary low-level infrastructure and functionality for the multi-chip operation of arbitrary network architectures described in the work at hand.

Each digital event routed through the network consists of a reduced address to identify the pre-synaptic sender neuron and a time stamp to mark its time of occurrence. Each logic connection belongs to exactly one so-called connection bundle. The mentioned reduced address, also referred to as `subnr`, identifies the pre-synaptic neuron within its connection bundle. The assembly on the sender side and the interpretation of the event on the receiver side are performed by two separate logic blocks, that are schematically illustrated in Figure 1.6 and Figure 1.7. Each side requires several LookUp Tables (LUTs) for the operation. Due to their complexity, the sender and receiver logic are discussed below in greater detail.

FPGA user logic has access to the MCGN via the user ports. Physical links are interfaced through the MGT ports provided by the FPGA (see Figure 1.5). Every Nathan module possesses eight MGT ports. Four of them are hard-wired on the backplane in the toroidal shape described in Section 1.1.3. For every time slot within a transmission frame of the MCGN, the ports can be freely inter-connected. Forwarding traffic necessary for multi-hop connections requires routing from an MGT input port to an MGT output port, while start or terminal connections require user port input or output routing, respectively.

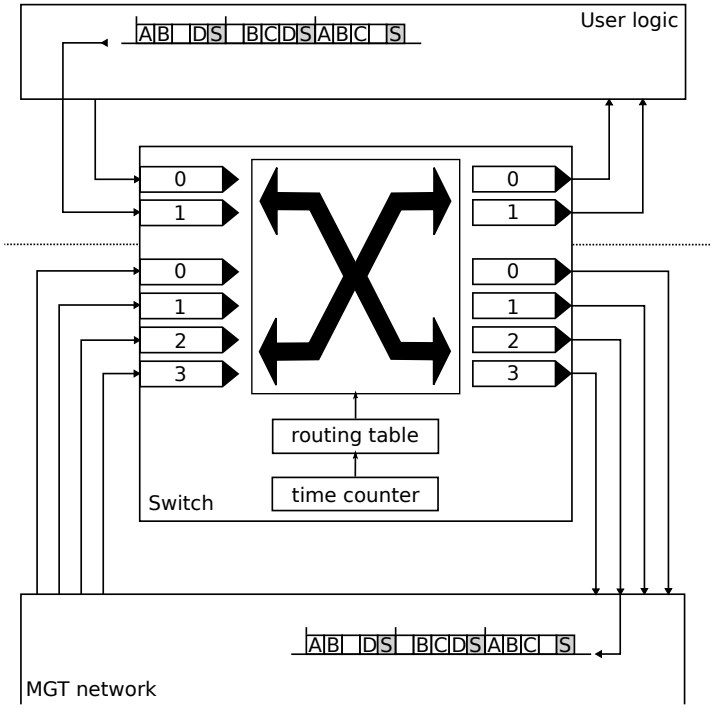


**Figure 1.5.:** Access to the MCGN network is provided via so-called user ports on the FPGA logic side and via the available MGT ports on the network side. The switch table can be programmed for every transmission slot individually. Figure by S. Friedmann [*Friedmann*, 2009].

**Sender Logic** Output events produced on Spikey are initially processed in an FPGA logic block – the so-called `source_gen` – where local events are filtered. Events not determined for transmission are dropped, while all the rest is translated into network events by providing the so-called multi FIFO index (mfi) to identify the connection bundle and looking up the `subnr` for the corresponding source neuron within the bundle. Furthermore, the event is provided with the address of the sender user port and its time stamp is extended. This user port is stored in a LUT called `Send_lut`. The subsequent switch block – referred to as `Send_switch` – routes the events to the so-called `Transmit_buffer`, a First In, First Out (FIFO) buffer that belongs to each user port on the sender side. Based on their priorly obtained mfi the events are pushed into one of multiple queues in the buffer, where they are stored until a matching MCGN time slot occurs. In each time slot, one of the queues is exposed to a sender user port and events are transmitted over the network (see Section 1.1.3). To determine which FIFO buffer has to be forwarded in which time slot, another LUT, the so-called `Mfi_lut`, is provided. It stores the assignment of time slot to mfi for a specific user port. An overview of the sender data path is illustrated in Figure 1.6.

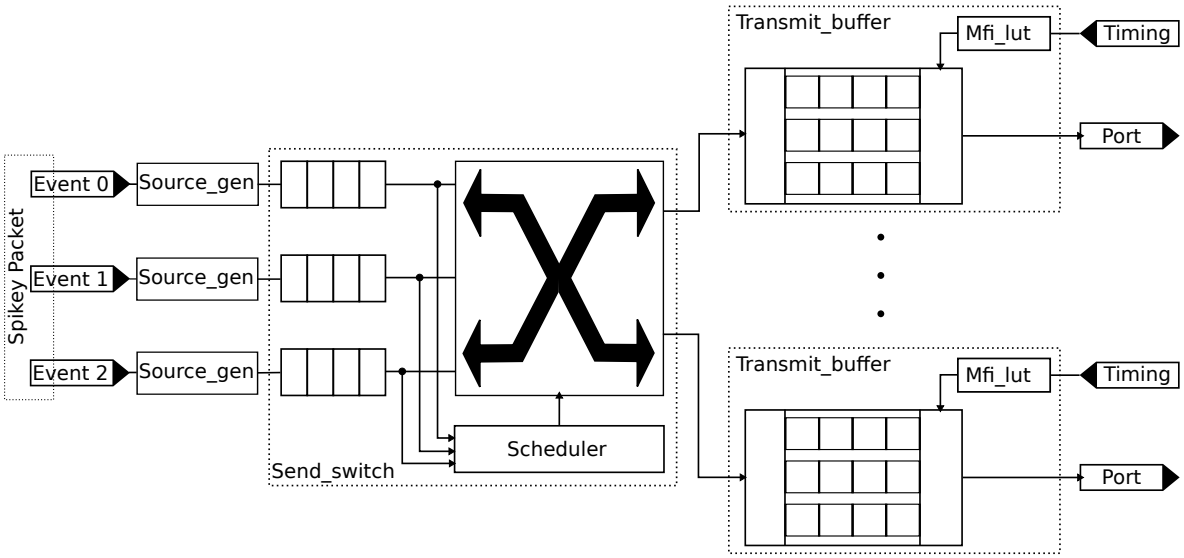


**Figure 1.6.:** An illustration of the sending FPGA logic with the pre-synaptic events flowing from left to right. The `Send_switch` connects the Spikey interface to the network interface. Events are inserted into one queue of the `Transmit_buffer` depending on their remote destination and periodically inserted into the MCGN. Figure by S. Friedmann [*Friedmann*, 2009].

**Receiver Logic** Figure 1.7 shows a schematic of the logic on the receiving side of the network. In the depicted configuration two receiver user ports are connected to the event input buffer, but only one event stream from the playback memory is connected. Therefore the overall bandwidth of input spikes from the playback memory is reduced by a factor of three compared to single-chip operation. On the receiver side, the events are picked up from the MCGN and mapped by the `make_target` onto a receiver-side connection bundles according to the so-called Logical Virtual Connection (lvc) number. The `make_target` module further resolves the target synapse driver and synaptic delay of the event by looking up the concatenation of lvc and the transmitted `subnr` in the `Recv_lut`. At this point the interpretation and retranslation

of network events back to Spikey events is complete. The subsequent `Recv_queue` and `Sorter` provide functionality to delay events from each sending Nathan by a global value and by an additional individual time delay for each target synapse driver. Finally, depending on their target synapse driver address, the `Reduce_events` module merges the event streams and distributes them to their appropriate input buffer (see *Friedmann* [2009] for further detail on the implementation of each module).

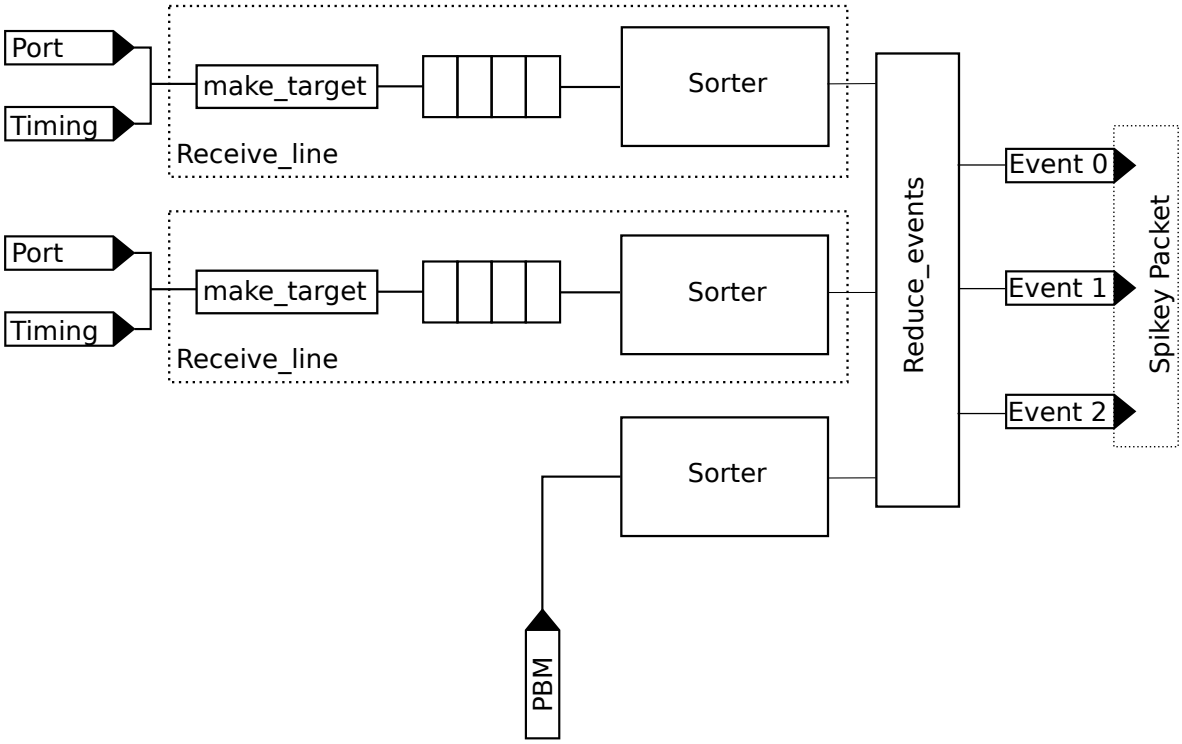


**Figure 1.7.:** An illustration of the receiving FPGA logic with the incoming data propagating from left to right. Events inserted from the MCGN and input from the playback memory (labeled "PBM") are merged to one stream. The stream is finally presented to the Spikey interface controller illustrated on the right side. Figure by S. Friedmann [*Friedmann*, 2009].

The number of programmable connections is soft-bounded by the bandwidth constraints given by the MCGN-network on the one hand and hard-bounded by the number of available logic units in the FPGA for the realization of LUTs on the other. If event rates exceed the limits of the MCGN, some events are randomly dropped, which results in an artificial network distortion that is difficult to characterize.

For the FPGA design used in the thesis at hand, two user ports are implemented, limited by the capacity of the FPGA. Furthermore, in the employed setup, the number of pre-synaptic neurons per connection bundle is limited to 64 due to the 6 bit wide `subnr`. The number of connection bundles on the sender side, is limited to eight per user port because of the 3 bit wide mfi. On the receiver side the number of connection bundles per user port is limited to 16 due to the 4 bit wide lvc.

### 1.1.5. The FACETS Wafer-Scale Hardware System

Beyond the hardware realization, the FACETS wafer-scale system provides a software tool-kit for the mapping of arbitrary neural architectures. Since the work of the thesis at hand is embedded into the wafer-scale software the hardware system is introduces shortly. The corresponding software is described in greater detail in Section 1.3.

The FACETS wafer-scale hardware is the next-generation neuromorphic hardware system developed within the FACETS project. Neuromorphic chips on an intact wafer are connected by a post-processing step instead of cutting the wafer into individual chips. The goal is to realize significant larger connection densities than one could achieve by conventional network technologies. One wafer fabricated in a 180 nm process will host 384 so-called High Input Count Analog Neuronal Network (HICANN) chips with a total of 200.000 Adaptive Exponential Integrate-and-Fire (AdEx) [*Brette and Gerstner*, 2005] neurons and up to 50 million synapses [*Millner et al.*, 2010; *Schemmel et al.*, 2010]. Each HICANN provides up to 512 single neurons and offers the functionality to combine adjacent neurons. Such a merged circuit can receive stimuli from up to 16.000 inputs. The asynchronous on-wafer spike communication is provided by the so-called Layer1 bus. The off-wafer communication to a host computer or other wafers is established by the embedded Digital Network Chips (DNCs) over the so-called Layer2 bus. Each DNC connects eight HICANNs to an external DNC FPGA module. Furthermore, each of the external DNC FPGA modules interconnects four DNCs. Hence, the system requires 12 external DNC FPGA modules in total to support all HICANN chips. An exploded view drawing of the complete wafer-scale system is shown in Figure 1.8.

To cope the complex task of mapping arbitrary neural architectures on such large systems a framework – the so-called *MappingTool* – has been developed. Section 1.3.1 provides more details concerning its basic building blocks.

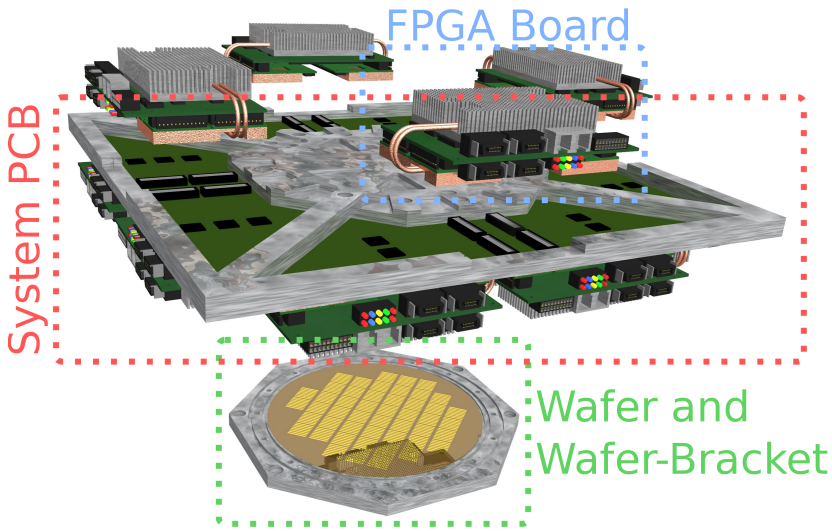


**Figure 1.8.:** An exploded view of the FACETS wafer-scale system. The lowermost module represents the actual wafer resting in its cooling plate. On top of it, one can see the wafer-scale system PCB and its mechanical support frame with FPGA communication modules below and above. Figure by D. Husmann.

## 1.2. Software Framework for Single Chip Operation

This section provides insight into the layers of currently available software used to operate the chip-based system described in Section 1.1. In the following, they are described from top (user interface) to bottom (Hardware Abstraction Layer).

### 1.2.1. The Modeling Language PyNN

The *PyNN*-language is aimed at modelers offering them a python-based Application Programming Interface (API) for simulator-independent network description. Network models described in PyNN can be ported to a large variety of software simulator backends only by changing a single line of code (*Davison et al.* [2010], *Brüderle et al.* [2009]). For a list of supported backends see Figure 1.9. PyNN also offers neuron models with standardized parameter sets and units.

By issuing different simulators to produce and reproduce experimental data one can more easily verify results and identify distortions produced by the software itself. Having the ability to choose the simulator freely can be beneficial. Each simulator has its individual strengths and weaknesses. Hence different parts of the same task can be solved by means of different simulators, according to their suitability. Besides the flexibility it offers PyNN is free, open source and well documented. By providing the necessary interfaces to operate the FACETS chip-based system and the FACETS executable wafer-scale system specification [*Vogginger*, 2010], it allows these systems to emulate nearly arbitrary neural network architectures. In this context, PyNN enables the hardware developers to analyze their system and the modelers to use the hardware with a simulator-like interface. Having thus rendered the knowledge of hardware-specific detail obsolete, from a modeler's perspective, the systems can be used as a general-purpose neuroscientific modeling tool.

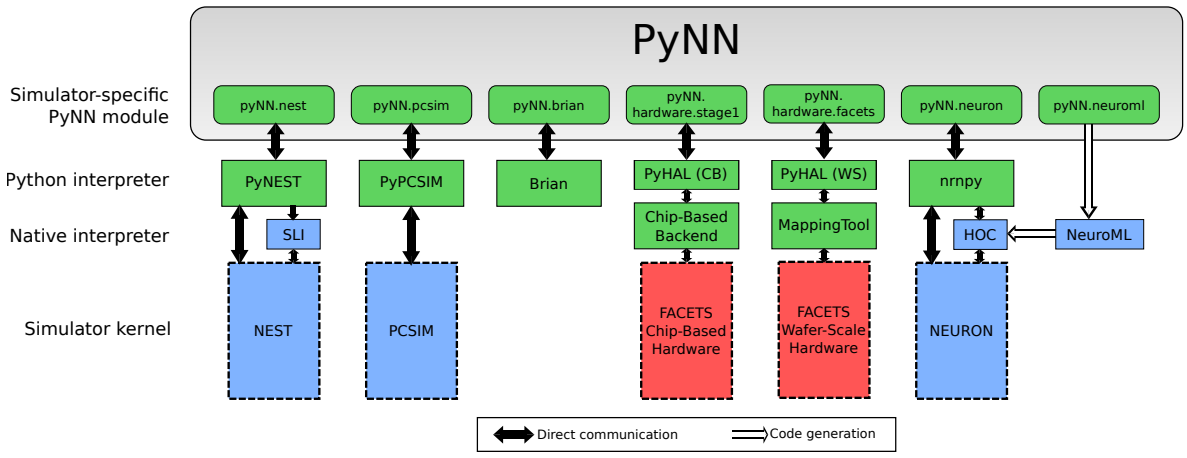


**Figure 1.9.:** The simulator-independent network description language PyNN adds an abstraction layer on top of established neuro-simulators to unify their operation. A specific translation backend for every supported simulator is necessary. The FACETS chip-based system and the FACETS wafer-scale system can both be accessed via PyNN, but require different backends despite their similarities.

### 1.2.2. PyNN Backend for the Chip-Based System

The translation of the abstract network description onto the single-chip hardware backend requires the software to distribute the neurons to their hardware counterparts and to configure the hardware parameters appropriately with respect to the model parameters. Therefore, a framework completely implemented in Python has been established by *Brüderle et al.* [2009]. The software maps the neurons sequentially by their internal PyNN ID with a configurable offset onto the available hardware neurons while checking for conflicting configurations. Parameters are translated from the biological domain to the corresponding electric hardware values. The translation is subdivided into two steps: a base transformation, which is identical for every hardware entity on chips of the same revision, and an individual calibration step. Calibration data needs to be determined and archived into an Extensible Markup Language (XML) file once for every chip and is from then on available for an experiment (for more information see *Brüderle* [2009]). Alongside the parameters, the input spike trains are converted into a hardware compatible format in terms of time stamps and hardware neuron addresses. Finally, the prepared data is transmitted and written into the chip and into the playback memory. Subsequently, the stimulation – and thus the experiment – is triggered (Section 1.1.2). After the last input spike, which determines the end of experiment, the results are collected and delivered to the PyNN interface.

In its current state the software does not provide optimization functionality for the mapping of architectures exceeding single-chip capacity. It simply checks for violating configurations and, in case it finds such, it stops. However, in a multi-chip system with e.g. limited input counts per neuron, it becomes less obvious when and why a certain configuration exceeds the hardware capacity. Therefore, it is fundamental to perform an optimization step in terms of realizing intelligently as many requested neurons and synapses as possible. The goal of the thesis at hand is to provide this very optimization by trying to reflect the original network structure as close as possible.

**Language Adaption Wrapper** The so-called *PyHAL* is a thin layer closing the gap between the low- and the high-level API. It is designed to add as little complexity as possible to the software flow but mediate between the two worlds of Python or PyNN and the low level `C++` API of the so-called *SpikeyHAL* (see below, Section 1.2.3). The software module described above builds upon these classes exposed to Python.

### 1.2.3. Low-Level Hardware Interface

The software layer called *SpikeyHAL* is responsible for interfacing the hardware. It provides classes and methods representing the configuration of the Spikey chip, the communication structures and the Synchronous Dynamic Random Access Memory (SDRAM) module close to the FPGA. All configuration data and input spike trains can technically be understood as a sequence of commands carrying the respective data. There is no difference between commands responsible for the configuration and those driving the experiment. Functionality provided to the user by the SpikeyHAL is internally translated into correctly ordered and scheduled command sequences. The actual experiment starts when the dynamics of the neural architecture come to life driven by the input played from playback memory.

## 1.3. Software Framework for Wafer-Scale Operation

The highly complex wafer-scale system offers a significantly larger configuration space as described in Section 1.1.5. Although the new system shall also be interfaced via a convenient PyNN interface, the idea of extending the pure Python backend was dropped due to performance considerations. The estimated mean event rates [*FACETS D7-13*, 2010; *FACETS M7-5*, 2010] require parallelized data structures and a strict zero-copy policy from top to bottom of the software. Furthermore, the software needs to be more versatile to handle the increased complexity. This led to the development of a new framework completely written in `C++`, the so-called *MappingTool*.

### 1.3.1. MappingTool

The operation of a neuromorphic hardware system requires several algorithmic tasks to be solved in advance. The MappingTool, as a software framework, provides all the necessary tools. It provides data structures and algorithms to place biological neurons onto hardware neurons, to route synaptic connections, to translate parameters from the biological to the hardware domain and vice versa as well as to configure hardware functionality. It has been developed to operate the FACETS wafer-scale system in one consistent flow [*Ehrlich et al.*, 2010]. More details on each step are given further below. The goal was to establish a framework that is as flexible and as modular as possible, so that it is easily adaptable to the still evolving hardware platform. Because performance was another major concern, the MappingTool has been developed to be scalably operated on parallelized unified memory machines [*Savage and Zubair*, 2009] as well as on distributed cluster architectures [*Pacheco*, 1996]. Its modularity further enables the system to swap single algorithms depending on the available computational power and the type or level of complexity for a certain problem.

The procedure of mapping arbitrary network descriptions to the hardware substrate is a highly complex algorithmic task. Therefore the design of the MappingTool intends to approximate the solution of the task by dividing it into three less complex steps which are assumed to be mostly independent. These conceptual steps are: the placing of neurons, the routing of synaptic connections and the translation of parameters into the hardware domain. They are further illustrated in Figure 1.10. The multi-chip environment in focus of the work at hand was embedded into the MappingTool due to its flexibility and availability. Hence, the next sections will give a short introduction to the components and basic concepts require to perform the mapping of neural network descriptions onto a neuromorphic hardware system.

#### The GraphModel Data Structure

The underlying data structure of the MappingTool is the so-called *GraphModel*, a container class dedicated to store all relevant biological and hardware information in a hierarchical graph structure. The fundamental concept behind the GraphModel is to have a complete representation of the topological information from network description to hardware configuration. This model is necessary and sufficient to store and restore an experiment for the purpose of reproducibility.

The basic building blocks of the graph structure are basic node entities and several types of edges. Each node carries a label that can be employed in a *key-value* manner to archive arbitrary data. The edges can provide additional information depending on their type, which is one of the following:

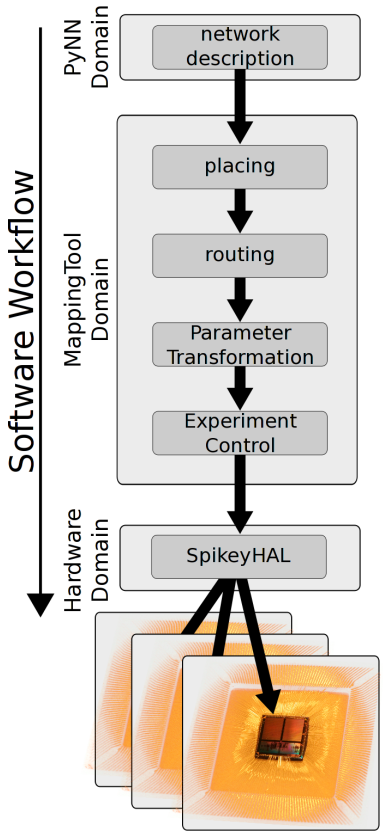


**Figure 1.10.:** Illustration of the complete mapping flow and the conceptual split of the network mapping task into three less complex problems to obtain an optimized solution.

**Hierarchical Edges:** The basic hierarchical tree structure builds upon the hierarchical edges. Each node has a parent and may carry an arbitrary number of related child nodes. The root node is characterized by being its own hierarchical parent.

**Named Edges:** The named edges enable the GraphModel to carry cycles, so that the tree structure is extended to a more general graph structure. Named Edges posses a tag member chosen from a finite set. The tag is intended to express the respective purpose of the edge (e.g. `MAPPING` edges from biological to hardware neurons). Named edges can be created between any two nodes, even between otherwise distinct trees like the representations of the biological and hardware model.

**Hyper Edges:** Named edges themselves can have a connecting edge to an extra node realized via the so-called hyper edges. These edges from edges to nodes can be used – similarly to the tag member – to provide additional information but in a more generic fashion. For example, synaptic connections represented as `NEURO_NEURO` named edges can therefore carry information about whether they have excitatory or inhibitory character.

**The Graph Navigational Language GMPath** To simplify navigation through the large GraphModel structures, the *GMPath* query language has been developed by *Wendt et al.* [2010]. Based on so-called navigational steps one can reach arbitrary nodes and edges inside the graph. Wildcards support the acquisition of recursive sets of nodes. Single queries can be combined to form more complex requests obtaining intersections of query sets. Furthermore the available hardware in a given experimental setup is described via a path language description, that needs to be supplied by the user.

### Specialized GraphModel Containers

Beside the basic GraphModel container, there exist further specialized containers with assisting functionality to hold and process context-specific data consistently.

**Biological Model** The so-called `BioModel` is derived from the GraphModel and enriches it with methods to consistently build, modify or remove neurons, synapses and biological parameter sets. It is created at runtime from the PyNN network description to hold all hardware-independent data. Stimulus data is carried by the corresponding neuron representations.

**Hardware Model** The hardware model is another specialization of the basic GraphModel. Compared to the biological graph, which is dynamically created from the network description, the structure of the corresponding hardware graph is known *a priori*. The idea is, that the complete structure of nodes and hierarchical edges is determined by the available hardware setup and represents the complete configuration space. Therefore the mapping only adds connections via named edges or modifies values of parameter nodes. The additional functionality provided by the hardware graph mainly consists of methods to build sub-tree structures fixed by the hardware design, e.g. hardware neurons with their corresponding electrical parameters. Consequently, by merging the single sub-components, the unfolded hardware representation is created.

### Assignment of Biological Cells to Hardware Neurons

In the given context, *placing* describes the procedure of mapping neurons from the biological graph onto neuronal nodes in the hardware graph. A good placing is crucial for the performance of subsequent algorithms and the routing process in particular. For example neurons with a high local connection density need to be grouped together on one HICANN chip for an efficient routing that avoids massive bus allocation. The most mature placing algorithm developed for the wafer-scale system so far is the so-called *N-Force-Cluster (NFC)* algorithm implemented by [*Ehrlich et al.*, 2008]. Depending on their similarity, the algorithm applies virtual forces onto the neurons. Neurons with similar afferent and efferent connections attract each other in a high-dimensional property space. The exact shape of the forces strongly depends on the objectives taken into consideration for the optimization process. After convergence into a stable state, the neurons are clustered via a *k-means clustering algorithm* [*Kanungo et al.*, 2002] and then distributed by drawing named edges to their hardware counterparts.

#### Routing of Synaptic Connections

The neural network architecture combined with the placing determines the desired synaptic hardware connections. Therefore the routing procedure needs to allocate hardware resources for their realization. In case the amount of connections exceeds the capacity of the available hardware, the routing ultimately needs to decide which connections to drop with respect to a set of objectives, like for example minimal synaptic loss [*Fieres et al.*, 2008].

#### Hardware Parameter Translation

Finding a suitable translation from biological to hardware parameters is not in all cases trivial. Not every parameter in the biological model of a neuron or a synapse has a corresponding hardware counterpart and the other way round. A single parameter may be emulated by a set of hardware parameters and some parameters are shared by multiple entities. Additionally, the output of the experiment needs to be translated back into the biological domain to close the loop of hardware abstraction necessary for the PyNN layer.

In the most simple case, a single parameter from one domain is translated to one value in the other by means of a simple translation function that possibly involves some calibration data. For hardware parameters shared by multiple entities on the substrate the translation process needs to find a compromise in terms of minimizing the overall distortion of the neural architecture. The potential quality of a parameter translation heavily relies on the placement performance. A compromise for a set of mapped hardware entities with shared parameters, where the required target translation values differ significantly, yields a higher distortion. In some cases, like for example the translation of continuous biological weights into the discrete hardware weight space, such compromises are inevitable.

For both chip-based and wafer-scale systems, the effective synaptic efficacy is given by:

$$g_{\mathrm{hw}}^{\mathrm{eff}}(t) = g_{\max} \cdot w \cdot \beta_{\mathrm{stdf}}(t) \qquad ,$$

where $g_{\max}$ is a base efficacy shared by all synapses in the same row, $w$ is the discrete 4 bit hardware weight and $\beta_{\mathrm{stdf}}(t)$ is the impact of the STP mechanism and is also shared for one row of synapses. For the initial configuration, $\beta_{\mathrm{stdf}}$ has no impact and can therefore be treated as a scaling factor to be set to 1. The parameter translation needs to find suitable values for the complete set of parameters for all synapses with respect to a minimal distortion of the general synaptic efficacy. This distortion originates from several causes. The most obvious one is the discretization of the weights, but it is additionally amplified by tuning the synapse driver parameter $g_{\max}$ shared a row of synapses.

## 1.4. The Liquid Computing Paradigm

Computation on continuous input is still a problem. The concept of the LSM offers a particular but generic ansatz to such problems by projecting the input into a high dimensional space before presenting it to a classifier, a trick well known from machine learning. It empowers neuromimetic classifiers to perform beyond their common characteristics.

### 1.4.1. Concept of the Liquid State Machine

In traditional computer science Finite State Machines (FSMs) have emerged as a useful generic tool to build algorithms based on transitions between stable states [*Hopcroft and Ullman*,

1969]. A similarly unifying concept, but fundamentally different in its architecture, is realized by the *Liquid State Machine*, as proposed by *Maass et al.* [2004] and by *Jaeger* [2001] in a very similar form. It offers a generic method for the approximation of arbitrary, time-continuous filters which operate on continuous input streams. In contrast to the FSM the LSM does not require convergence into stable states but is inherently designed to work with transient states.

The mathematical model of the LSM, illustrated in Figure 1.11, acts as a filter that maps an input function $\vec{u}(t)$ onto an output function $\vec{y}(t)$ continuously in time. *Maass et al.* [2002] have shown that such LSMs have *universal power in computation with fading memory on functions in time* and can therefore approximate any suitable filter with arbitrary precision. An LSM consists of: the liquid, which translates the input stream into the so-called *liquid state*, and a memoryless readout. The liquid state is a non-stable but transient state which reflects not only the current input but also a decaying representation of its past. This *fading memory* enables the memoryless readout to make a decision based on the recent input history by looking only at the current state of the liquid. To achieve that, the LSM is required to satisfy two properties:

**Separation Property** The underlying liquid does not necessarily need to be constructed as a neural network but it needs to be sufficiently complex that at any point it can provide a mapping of two different input functions $\vec{u}(t)$ and $\vec{v}(t)$ to two different liquid states $\vec{x}_u(t)$ and $\vec{x}_v(t)$ that are significantly different while staying in a non-chaotic regime in terms of conserving the relative distance of two different input streams. An illustration of the separation property can be found in *Maass et al.* [2002] in Figure 2).

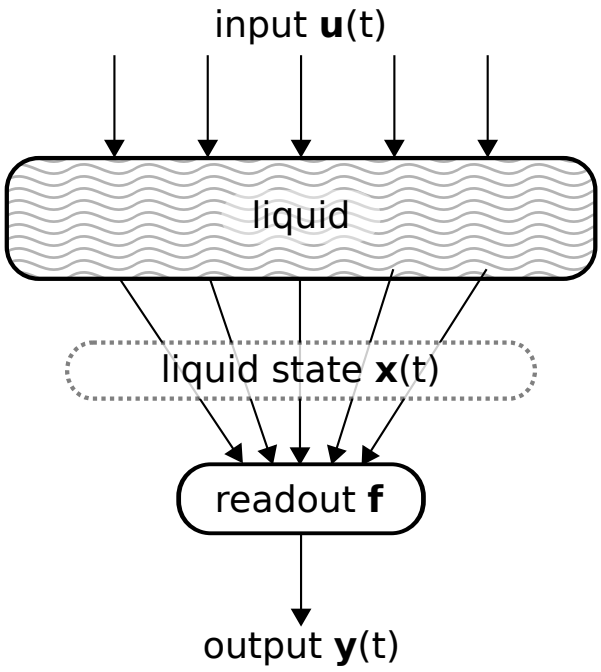


**Figure 1.11.:** The mathematical model of the *Liquid State Machine* transforms the input $\vec{u}(t)$ into the liquid state $\vec{x}(t)$ which is then mapped by the readout function $\vec{f}$ onto the output $\vec{y}(t)$

**Approximation Property** The readout needs to have a sufficient resolution to extract the required target output for a given task from the arising internal liquid states. Still it does not need to provide a memory as this is provided by the liquid itself.

### 1.4.2. Motivation and Biological Relevance

In machine learning the projection of data into a high dimensional space is known as the *kernel trick* e.g. in support vector machines [*Meyer et al.*, 2003] or support vector networks [*Cortes and Vapnik*, 1995]. In biological neural networks this concept is believed to be exploited as well. For example the olfactory system of insects is assumed to first decorrelate the sensory input in the so-called glomeruli and to then project this pre-processed information stream it on a highly interconnected network of neurons to ease the subsequent classification [*Schmuker et al.*, 2008].

Although it is not yet perfectly clear how the wiring of the mammalian brain emerges on different scales, it is reasonable to assume the wiring in very local volumes of the cortex to be statistical [*Braitenberg and Schüz*, 1991]. Consequently also such structures could be suitable for computation similar to the liquid computing paradigm.

### 1.4.3. Output Classification

A common task is to divide a set of input patterns into different groups, based on a small set of characteristic features. Finding the proper feature set to perform the separation is often a difficult task on its own. One ansatz is to provide a training set consisting of input and desired output, where the classifier tunes itself to a proper feature set during an initial learning period. Over the years a large variety of algorithms has been developed to perform the task of trained classification [*Alpaydin*, 2004]. Often they obtain their inspiration from neural circuits found in natural systems.

#### The Perceptron: A Linear Readout

The perceptron is a binary classifier developed in 1957 by *Rosenblatt* [1958]. It consists of a simple feed-forward network architecture that is able to perform linear separation tasks. Linear separation refers to the fact that the classification decision is based upon a linear combination of the input characteristics. An illustration of such a separation on a two dimensional input space can be found in Figure 1.12. Multiple layers of perceptrons can overcome the limitation of linear separation and can consequently separate arbitrary convex sets in the input space.

The output of the perceptron is the result of the weighted sum over the input compared to an arbitrary threshold and is given by Equation 1.2.

$$f(\vec{x}) = \begin{cases} 1 & \text{if} \quad \vec{w} \cdot \vec{x} + c_{\text{thresh}} > 0 \\ 0 & \text{else} \end{cases} \quad . \tag{1.2}$$

The weights can for example be tuned via the so-called *delta learning rule*. This rule has been shown in *Novikoff* [1963] to converge to the correct classification if the input is linearly separable.

$$\omega_{ij}^{n+1} = \omega_{ij}^{n} + \Delta\omega_{ij}^{n} \tag{1.3}$$

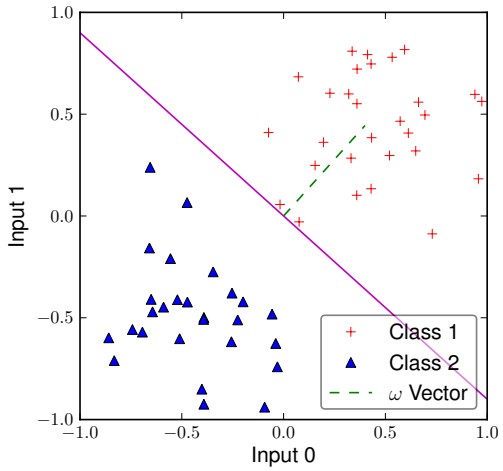


**Figure 1.12.:** The input space is divided by a hyperplane which is orthogonal to the weight vector $\vec{w}$ and passes through the origin of the coordinate system (since $c_{\text{thresh}} = 0$). For any point $\vec{x_p}$ in class 1, it holds that $\vec{w} \cdot \vec{x_p} > 0$ and for any point in class 2 it holds that $\vec{w} \cdot \vec{x_p} < 0$, respectively.

$$\Delta\omega_{ij}^{n} = \alpha(n) \cdot (t_j - o_j) \cdot x_i \qquad . \tag{1.4}$$

$\omega_{ij}^{n}$ represents the weight for the connection of the $i$th neuron to the $j$th perceptron after the $n$th learning step. $\alpha(n)$ is the learning rate and can decay for an increasing number of learning steps. The input of the $i$th afferent neuron $x_i$ presented to the $j$th perceptron yields the output $o_j$. For correct classifications $o_j$ equals the target output $t_j$.

Since the original perceptron design was not based on spiking neurons it can not directly be implemented on the FACETS chip-based system. All results presented within this work that are based on perceptrons have been obtained by means of software implementations. This limitation may be overcome by the *tempotron* described in the following.

### The Tempotron: A Spike-Based Non-Linear Readout

In contrast to the perceptron the spike-based tempotron accounts for the spatio-temporal structure of the input. Hence it is offered richer information to perform its tasks. The basic integrative element is a common leaky integrate and fire neuron. Therefore the tempotron is a candidate for implementation on the FACETS chip-based hardware. Although the hardware provides synaptic plasticity in terms of STP and STDP (see Section 1.1.1), the originally proposed learning rule given by Equation 1.8 cannot be applied to the hardware directly. Consequently the learning logic needs to be run offline on the host computer for the modification of weights according to the output of the classifier. Directly implementing the classifier on the hardware is useful beyond merely having replicated the entire LSM architecture on neuromorphic hardware. Processing the output directly on chip and reading out only the result of the classifier would be beneficial at least in two ways: the acceleration factor of the neuromorphic hardware system (see Section 1.1.1) and the reduction of required host PC communication bandwidth.

In the original tempotron proposed by *Gütig and Sompolinsky* [2006], the afferent synaptic weights are trained in a way that the tempotron neuron emits no spike within a certain time window while being exposed to one class of input and exactly one spike while being exposed to the other. The latter is enforced by blocking all incoming spikes after the first output spike has been emitted within the time window. However it is stated that the learning rule is still quite stable without this addition. Since this is neither very biologically plausible nor possible on the neuromorphic system this feature is explicitly neglected in the following considerations.

The originally proposed tempotron employs a LIF neuron with current-based synapses. The voltage course of its membrane given by the respective differential equation can be solved analytically and can be written as the weighted sum over the afferent synapses and spikes biased by the resting potential $E_l$:

$$V(t) = \sum_i \omega_i \sum_{t_i} K(t - t_i) + E_l \qquad . \tag{1.5}$$

The contribution $K(t - t_i)$ to the post-synaptic potential for current-based exponential synapses is:

$$K(t - t_i) = V_0 \left( \exp\left( -\frac{t - t_i}{\tau} \right) - \exp\left( -\frac{t - t_i}{\tau_s} \right) \right) \cdot \Theta(t - t_i) \qquad . \tag{1.6}$$

$\tau$ and $\tau_s$ denote the time constants related to membrane integration and synaptic currents respectively. The index $t_i$ of the inner sum runs over each spike of the $i$th input while the outer sum runs over all inputs.

To deduce a weight update rule for the training process, consider a cost function $E_\pm = \pm(V_{\text{thresh}} - V(t_{\max}) \cdot \Theta\left(\pm(V_{\text{thresh}} - V(t_{\max}))\right)$. $t_{\max}$ denotes the time where $V(t)$ is maximal during the classification window and the Heaviside step function ensures that only false classifications are affected. This cost function measures the distance between $V(t_{\max})$ and the threshold voltage for erroneous patterns or by how far the classification missed the correct output. While $(+)$ applies for patterns which should elicit a spike but did not, $(-)$ applies for patterns which erroneously emitted a spike. Now, we want to minimize the overall cost to maximize the correct classification. In a first-order approximation, this requires shifting the $i$th weight by $\omega_i$ into the opposite direction of the respective gradient component of $E$:

$$-\frac{\mathrm{d}\, E_\pm}{\mathrm{d}\omega_\mathrm{i}} = \pm \sum_{t_i < t_{\max}} K(t_{\max} - t_i) \pm \frac{\partial V(t_{\max})}{\partial t_{\max}} \frac{\mathrm{d}\, t_{\max}}{\mathrm{d}\omega_\mathrm{i}} \qquad . \tag{1.7}$$

Where $\partial V(t_{\max})/\partial t_{\max} = 0$ holds by definition. Furthermore, the maximum voltage $V(t_{\max})$ always corresponds – by design – to the first erroneous threshold crossing. To ensure this, all further spikes are blocked, as previously described.

Consequently, we have deduced a learning rule by implementing a *gradient-descent* method with $\Delta\omega \sim -\,\mathrm{d}\, E_\pm / \,\mathrm{d}\omega_\mathrm{i}$:

$$\Delta\omega_i^n = \alpha(n) \sum_{t_i < t_{\max}} K(t_{\max} - t_\mathrm{i}) \qquad . \tag{1.8}$$

Where $\Delta\omega_i^n$ is the change in weight corresponding to the $i$th afferent neuron after the $n$th learning step and $\alpha(n)$ represents a learning rate, which may decrease over the learning period. For patterns which erroneously did not elicit a spike in in the $n$th iteration the $i$th weight is increased by $\Delta\omega_i^n$ and decrease respectively if the spike was erroneously emitted.

### Decay of Learning Rates

Constant learning rates over time can lead to significant variations in classification performance from learning step to learning step even after long periods. To make the readout more robust against these variations, a decaying learning rate proves to be useful and can further improve the general readout performance. Training with constantly high learning rates – especially when close to the optimal weight configuration – can yield oscillations in the order of the applied learning rate.

Different decay functions $\alpha(n)$ (see Equation 1.4 and Equation 1.8) with different characteristics and parameters can be applied depending on the classifier and the input sets. Too short decay time constants can make achieving a close-to-optimal result impossible, while too large time constants can make achieving the same performance last arbitrarily long. Hence it is the experimentalist's responsibility to choose the function and parameters carefully.

The employed decay function for this thesis are:

- constant learning rate:
$$\dot{\alpha} = 0$$
- linear decay:
$$\dot{\alpha} = c$$
- exponential decay:
$$\dot{\alpha} = -\frac{\alpha}{\tau}$$

### Clipping Continuous Weights to Discrete Values

Since synaptic weights on the hardware are limited to a 4 bit resolution it is desirable to analyze the performance of a given readout depending on a discrete weight clipping. In case of the perceptron this has already been performed by *Rosen-Zvi and Kanter* [2001], whereby the tempotron still needs to be investigated with respect to hardware-specific constraints.

To finally obtain discrete weights different clipping strategies can be applied. The first strategy is to train the weights in a continuously in a software simulation and finally clip them to discrete hardware values. Another strategy, which would keep the weights discrete over the complete training period, could be to incrementing or decrementing the synaptic weights only by factors of the weight resolution. However, *Rosen-Zvi and Kanter* [2001] tuned the weights continuously during the learning period and ultimately clipped them to a discrete set of values for their perceptron analysis. The analysis for the tempotron will focus on the same continuous strategy since the original learning rule is optimized to perform on continuous weights and implements a *gradient descent* (see Section 1.4.3).

The variant of the readout in continuous weight space is referred to as *teacher* while the one in discrete weight space is referred to as *student*.

# 2. Realization of a Multi-Chip Setup and On-Chip Classification

The theoretical and technical prerequisites for the multi-chip operation and the analysis of Liquid State Machines have been outlined in Chapter 1. This chapter presents the conceptional considerations and concrete steps taken to achieve the realization of both the multi-chip system and the environment for the analysis of the LSM. The fundamental concepts for the operation of the multi-chip system in Section 2.1 are presented on a higher level of abstraction, to focus on the more general aspects. The subsequent two section will give a detailed insight into the actual software realization for the hardware system and the LSM.

## 2.1. Concepts for Multi-Chip Operation

The FACETS chip-based system is an evolving platform with a growing number of features. Thanks to the event distribution network, as described in Section 1.1.4, the system has been extended to provide inter-chip routing of neural events beyond chip boundaries. One of the main objectives of this thesis is to establish a framework to enable users to access and exploit this multi-chip functionality from PyNN (Section 1.2.1), while meeting the PyNN philosophy to require minimal knowledge by the user about the simulation or emulation backend.

### 2.1.1. Initial Considerations

The mapping of arbitrary neural network descriptions onto multi-chip setups raises some questions concerning the increasing complexity. These questions arise from new network-related constraints as well as from algorithmic complexity issues.

#### Hardware Inhomogeneities and Limitations

The neural network substrate provides a well defined but finite set of resources. Beyond its limitations it is inhomogeneous for on-chip- compared to off-chip-connections in terms of available connection densities and event transmission delays. Thus, a mapping process needs to distribute the resources intelligently. On the one hand available resources need to be used efficiently to realize as many synapses as possible and on the other hand the distortions due to delays and shared parameters need to be kept as low as possible.

Assuming a sufficiently large hardware system to host all desired biological neurons the effective loss of synapses by the limited mutual connection densities is supposed to stay as low as possible. However the actual connections to be routed in hardware depend on the desired synaptic connections in the neural architecture and the given neuron placement. Hence the routing task itself strongly depends on the placement. To ultimately achieve an optimal mapping in terms of minimal network distortion one would have to neglect the task division, which is explained in Section 1.3.1. This is not desirable with respect to mapping runtime

performance, but at least the routing and placement should be tuned to support each other. For example inhomogeneities in connectivity should already been taken into consideration during the placement step and therefore be represented as an appropriate coast function. The available N-Force-Cluster algorithm described in Section 1.3.1 could be modified to carry out the task.

### Scalability and Performance

The operation of a highly accelerated neuromorphic hardware device may be significantly slowed down by the communication with a host computer, the retranslation and the reprogramming of parameters. This is in particular critical for repetitive experiments to explore large parameter spaces. Therefore, the implementation of the controlling software should perform at least on a similar, but desirably on a faster time scale compared to the neuromorphic computation itself. While the Artificial Neural Network (ANN) does not slow down when scaling up the network size the software does due to the increasing complexity of the task to map architectures of such size. Hence, the algorithms considered for the implementation need to be chosen with respect to their efficiency. Solutions computed by heuristic algorithms might be preferable over purely quality optimized solutions if they can provide a better runtime performance.

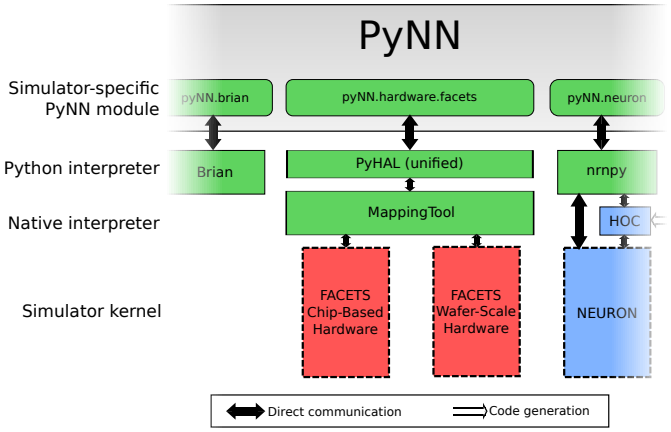


**Figure 2.1.:** The unification of the PyNN backends for the FACETS hardware as implemented and described in this thesis. In comparison the previous software state is illustrated in Figure 1.9. With this new structure less redundant code needs to be maintained and new features are instantaneously available to both systems.

### Modularity and Flexibility

With the GraphModel-based mapping flow (see Section 1.3.1) – developed for the wafer-scale system – a flexible and extensible tool is already at hand. Therefore, the multi-Spikey operation should integrate with the existing framework to demonstrate its generality and more important to minimize code redundancy. This conceptual unification in the mapping layer is illustrated in Figure 2.1 in comparison to the software for the single-chip system depicted in Figure 1.9.

Furthermore, it is desirable that experimentalists can choose from a variety of different algorithms and chain them together in a way suitable for their workflows. On that score generic algorithms should be accessible for both hardware generations. To meet the requirements

of the following step, the subsequent execution of selectable algorithms requires consistent task definitions in terms of specified modifications to the GraphModel structure. Some of the conventions are implicitly enforced by the definition of the specific hardware model itself, but some are more subtle, especially the named edges between nodes that need to be drawn. Changes to the model by preceding algorithms can have a severe impact on the outcome of subsequent operations. Therefore consistent documentation and verification is essential.

### 2.1.2. Specialized Graph Representation

The integration of the mapping flow into the MappingTool as considered in Section 2.1.1 requires a new individual GraphModel container to represent specifically the multi-chip system.

The new derivative of the hardware model container is developed in the style of the hardware container, that is available for the wafer-scale system described in Section 1.3.1. By keeping major concepts similar, established algorithms can be reused more easily. Obviously the graph representation needs to provide structures for: backplanes, communication structures, FPGAs and the Spikey chips. Those are hierarchically ordered in the shape of a tree. Since the configuration is only modular up to a certain level in terms of removing components and extending the system, the tree can be constructed by attaching smaller sub-trees with a predefined structure to higher-level nodes. For example the structure of the chip is determined *a priori* but not the number of chips on one backplane or the number of inter-connected backplanes. An additional hierarchy level is inserted in precaution of a future extensions to the system to operate multiple chips by a single, more modern FPGA. Figure 2.2 shows a sketch of the hardware container structure.

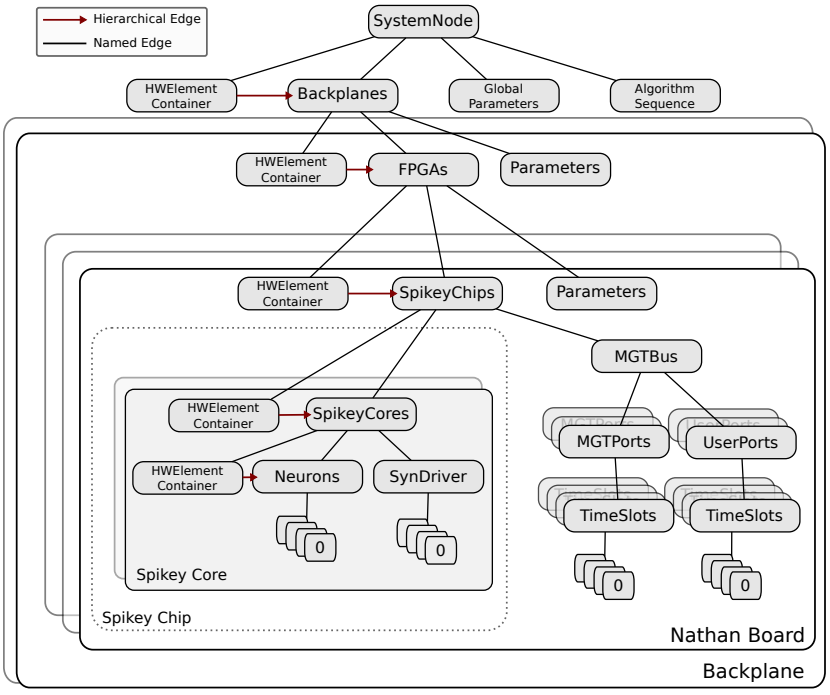


**Figure 2.2.:** A template hardware graph illustrating the basic components of an unfolded `HWModelStage1` definition.

### 2.1.3. Neuron Placement Beyond Chip Boundaries

The placing described in Section 1.3.1 refers to the *abstract* mapping of biological neuron representations onto their hardware counterparts. The available recursive placing requires no specific implementation details. Thus the existing algorithms can be reused for multi-chip operations. Refining or extending the cost functions for the N-Force-Cluster algorithm [*Ehrlich et al.*, 2008] can increase the performance for the multi-chip system in future implementations. The impact of an intelligent placement on the overall mapping distortion is discussed in Section 2.1.1. At the time of submission of this thesis the N-Force-Cluster carries out some wafer-scale system-specific placement decisions which degrades its placement performance in terms of subsequently realizable synaptic connections. The drawback is described in more detail for the performed experiments in Section 3.1.1.

Besides, the fully automated placing of neurons it can be beneficial to provide a more customizable placing. Often neural architectures build upon smaller local structures with higher internal connection densities like for example columns. Consequently, a certain neuron distribution on the hardware is suggested by the connectivity of the original network. Under other conditions it might be necessary to distribute or cluster neurons to achieve a specific timing via the programmable delays. A way to carry out such a manual placing from within PyNN should be offered to modelers. Section 2.2.3 introduces the semi-automatic population placement algorithm, which is controllable via `labels` set for neuron populations in PyNN.

### 2.1.4. Routing of Synaptic Connections

The routing of synaptic connections is less abstract than the placing of neurons, since specific structures of the available communication infrastructure need to be taken into account. All desired synaptic connections between neurons are determined by the neural network description and the preceding placing. The routing decides which connections to realize with respect to hardware and bandwidth limitations. While the individual importance of single synapses within the neural architecture is not known, the main objective is to keep the synapse loss as low as possible and to keep the amount of inter-chip connections within bandwidth limitations. The dropping of arbitrary events within the network due to exceeding bandwidth limitations introduces artificial network distortions, which are difficult to track and analyze.

The realized routing consists of three consecutive steps. Initially the available routes between a given set of Spikey chips need to be discovered. These depend on the actual placement of Nathan modules on the backplanes and may vary from setup to setup. An illustration of this issue can be found in Figure 2.3. By considering the MCGN topology as a common graph structure the discovery can be treated as a shortest-path problem to find routes of minimal length between any two Nathans. In the following algorithmic step it is decided which synaptic connections to realize on each chip and over the network. The connection density is mainly limited by the number of available synapse drivers and the network bandwidth. Therefore, the algorithm needs to allocate synapse drivers appropriately for chip internal and external connections. The third and last routing step distributes the network resources according to the just realized synaptic connections and the previously discovered paths. It tries to fit the routes tightly together to minimize the waste of resources and to maximize the bus saturation. In the following all requirements for the mentioned three steps are successively considered.

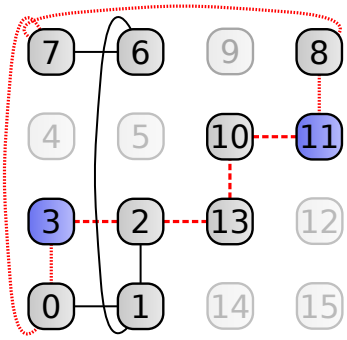


**Figure 2.3.:** A graph illustration of a sparsely occupied backplane, where the faded nodes represent missing Nathan modules and the solid ones represent available Nathans respectively. The edges represent the available links between the present modules. One can see that the paths and number of necessary hops between to Nathans modules depend on the exact configuration of the backplane. In this example the number of network hops to connect module 3 and 11 increases from two to four because the events need to be routed around the missing Nathans 4 and 13. The two shortest paths of length 4 are marked in red

**Network Discovery** The Nathan modules and their MGT links span a graph with edges that could be weighted to prioritize certain connections during the routing process. Transmission delays caused by the physical length of the link are irrelevant, since the isochronous network guarantees the delivery of events after a fixed period of time [*Philipp*, 2008]. Hence the weight of all edges can be considered equal and are set to 1.

For a given setup with a couple of Nathan modules on one backplane the network discovery needs to find the shortest paths between them. The acquired data is prerequisite for both the MCGN resource allocation and the synapse driver allocation, which are both described in more detail below. The former requires the precise topological structure of the routes with minimal length to configure the network accordingly. The latter only needs the distances to prioritize connections that require fewer hops to reach the target, taking into account that shorter routes have a higher effective bandwidth. This is due to the way the MCGN resource allocation distributes the network resources. It fills up empty transmission slots with multiple instances of requested routes until no requested route fits anymore. Since short routes fit more easily, those get assigned more frequently.

The *shortest path* problem is solved by means of a modified Dijkstra algorithm [*Dijkstra*, 1959]. Instead of only returning a shortest path between two Nathan modules, it returns all shortest paths of equal minimal length. By providing the MCGN resource allocation multiple realizations of potential routes with equal source and target, the algorithm can saturate the bus more efficiently. Dijkstra's algorithm provides deterministic runtime performance and is suitable for the desired extension concerning the multiple route realizations. An upper bound for its time complexity can be expressed as $\mathcal{O}(|V|^2 + |E|)$. Where $|V|$ and $|E|$ denote the cardinality of the sets representing the Nathan modules and the MCGN connections, respectively. Due to the fixed network topology it applies $|E| \leq \frac{1}{2} \cdot 4\,|V|$ for experimental setups without additional extern MGT links, while the equality is true for the fully occupied backplane. Consequently, the upper bound for the complexity can be expressed more easily as $\mathcal{O}(|V|^2)$.

**Synapse Driver Allocation** For the distribution of synapse driver resources one could sort the axonal connections and thus the synapse drivers by their number of associated synaptic terminals and then realize as many of them as possible in a descending order. However, considering an optimized placing that clusters neurons with high mutual connection counts onto a single chip, this sorting could lead to a muting of chips to the outside world and therefore induce large artificial distortions. Hence other objectives beside synapse loss need to be taken into account to convey fairness in terms of giving each synapse a certain chance to become realized. Additionally, it is desirable to control the relative importance of synaptic connections depending on whether they are realized chip-internally or via the network. This enables the user to for example emphasize objectives expressed by the preceding neuron placement.

Another ansatz one could think of for distributing the synapse driver resources is using *brute force* for finding the optimal solution with respect to a given set of objectives. To estimate the runtime performance we consider the emulation of a homogeneously connected neural network with the mean connection probability $p$ and a random placing. Then the probability $P$ for each synaptic source not to be connected to at least one of the 192 target neurons per Spikey core is given by: $P = (1-p)^{192}$. For local parts of cortical volume with mean connection probabilities of about $p = 0.1$ [*Thomson and Lamy*, 2007] one can assume $P$ to be 0, which means that any synaptic source wants to be connected to at least one of the neurons on each core. Now, if one wants to try out all possible configurations $N$ one has to check any potential distribution of the 256 available synapse drivers onto all synaptic sources on each core which conforms to:

$$N = n_{\text{cores}} \cdot \binom{n_{\text{sources}}}{256} \quad .$$

Where $n_{\text{cores}}$ and $n_{\text{sources}}$ denote the numbers of cores and synaptic sources in the setup. Since $N$ rapidly increases with $n_{\text{source}}$ the number of brute force possibilities is strongly affected by the network scale. Consequently the brute force approach has been neglected.

Algorithms which employ gradient descents for finding suitable solutions in a configuration space can not be applied, since the configuration space is non-continuous: the decision to turn on a synapse driver for a certain connection is binary. However, the problem can be considered similar to finding the energy minimum for spin-spin interaction in solid-states [*Landau and Binder*, 2005]. To perform the actual multi-objective optimization the so-called simulated annealing [*Kirkpatrick et al.*, 1983], an adaption of the *Metropolis-Hasting* Algorithm [*Hastings*, 1970], has been chosen. The most important reasons for this decision are its applicability for parallel approaches and its successional trajectory through configuration space via neighboring states which allows for the efficient generation of new non-conflicting configurations. The problem of minimizing the synapse loss with additional constraints is translated into an energy minimization problem by expressing the optimization constraints as adequate energy functions. A detailed description of the actual algorithmic realization and energy functions can be found in Section 2.2.4.

**MCGN Resource Allocation** So far available network pathways are identified and the synapse driver resources are distributed among the desired connections. Now the network resources, which provide a limited volume of transmission slots, need to be packed as tightly as possible with routes between terminal Nathan modules. This matches the bin packing or knapsack problem which belongs to the class of NP-hard problems in computational complexity theory [*Garey and Johnson*, 1979]. After a successful packing, remaining space within already

allocated resources can be filled up with multiple representations of requested routes. An alternative to reduce the number of unused transmission slots is the reduction of slots per transmission frame. The latter would increase the relative overhead on the bus caused by the synchronization signal between any two frames and has therefore been neglected.

From several possible strategies to solve the knapsack problem the so-called *first-fit decreasing* is realized in favor over e.g. the *best-fit* [*Yue*, 1990] bin packing algorithm, since it requires only the computational time of $\mathcal{O}(n \log n)$ compared to $\mathcal{O}(n^2)$, where $n$ refers to the number of MCGN routes. At the same time the first-fit decreasing algorithm consumes less than $^{11}/_{9} + 1$ times more of the resources the best-fit algorithm would use [*Yue*, 1990]. Effectively, the disadvantage of less optimal packing turns out to be even smaller as gaps within the network resource allocation are filled up with multiple instances of requested routes. The number of MCGN routes $n$ in the network increases with $\sim 8 \cdot N_{\mathrm{Nathans}}$. However, $n$ affects the runtime performance of both the MCGN resource allocation and the network discovery (described in the text above). The runtime performance of the complete routing process and all other mapping steps is benchmarked in Section 3.1.2.

### 2.1.5. Parameter Translation

In Section 1.3.1 the parameter translation as one step within the mapping from biological networks to the wafer-scale system was described. Problems arising from limited and discrete parameter ranges have been exemplarily discussed for the synaptic weight translation.

In a multi-chip environment the parameters of separate chips can be handled independently, except those concerning the inter-chip communication. Hence the optimization problem mostly can be treated locally and processed in parallel. A combined parameter translation that comprises both the transformation from biological values to idealized hardware parameters and the hardware calibration is desirable in terms of runtime performance and to satisfy the concept of the hardware model holding the final parameters.

The multi-chip system is further used to introduce a new concept for the storage and management of calibration, which will most likely be applied for the wafer-scale system, as well.

#### Application of Translation Data

One advantage of having a highly configurable hardware system is that the deviation of a physical implementation from its ideal due to production process imperfections can be attenuated by using specifically tuned calibrations. The necessary translation data sets need to be managed and require fast any-time access. The data sets necessary for the operation of the wafer-scale system are even larger than for the chip-based system. A calibration data set for the Spikey chip requires 40 kB, while a calibration data set for the wafer system is expected to consume about $20 - 30$MB [1]

Consequently, it was decided to move away from a file oriented calibration data management towards a central database backend. This approach innately allows for concurrent access hence allows for parallel operation and scaling to multiple host PCs as well as multiple database host computers. *MongoDB* has been chosen for the realization as a free and open-source

[1] Rough estimate: 100,000 membrane build blocks per wafer. One membrane building block requires 10 gauging data points consisting each of four 32 bit float values plus approximately half this memory amount for all synapse drivers.

database [*MongoDB*, 2010]. Compared to traditional relational databases which store data internally in fixed table structures and therefore require a certain data formatting, mongoDB can store arbitrary heaps of data in so-called collections. Such databases are often referred to as *scheme-free*. MongoDB is also considered to be used for the FACETS wafer-scale system parameter translation. Thus, The application of the database is useful beyond merely having an archive for multi-chip parameter translation data. Using the database in an every day scenario provides helpful experience for its application in the large-scale system.

Furthermore, mongoDB is inherently designed for horizontal performance scaling: By distributing the load on multiple PCs, so-called shards, the bandwidth of the system can be increased almost linearly. Modified data is kept autonomously in sync on all involved shards with a *copy-on-write* policy. In a FACETS wafer-scale setup one could think of setting up each host computer as a shard to improve the data access latencies and throughput. Access latency can be further improved by making use of the mongoDB's indexing feature for arbitrary fields of data. It enables the system to query and retrieve data more efficiently from the database.

Figure 2.4 shows a schematic of the provided workflow.

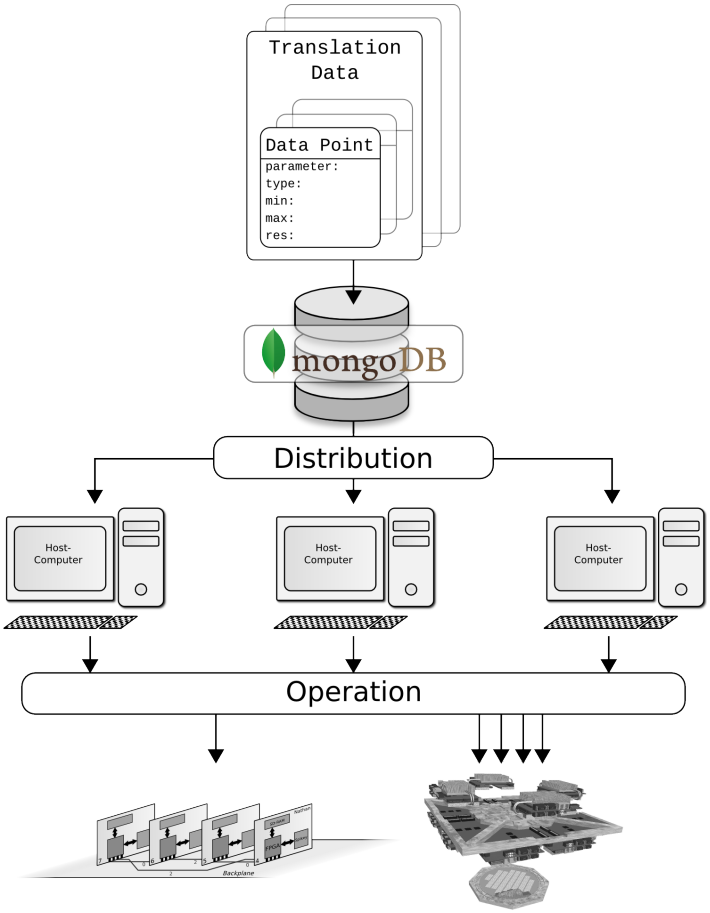


**Figure 2.4.:** Illustration of the parameter translation data distribution scheme. The required data is stored in a central *mongoDB* database, which can be concurrently accessed by multiple host PCs.

A Spikey chip dataset provides consistent gauging values to perform the parameter transla-

tion. The gauged data implicitly accounts for chip-to-chip variations. For more insight into the protocols and methods to obtain the gauging datasets for the qualitative and quantitative behavioral matching with reference simulations consider *Brüderle* [2009].

### 2.1.6. Further Concepts

So far concepts for the neuron placement, synapse routing and parameter translation have been outlined. Although the mapping framework and the operation of the hardware in particular require additional components, their explicit description omitted for reasons of clarity and comprehensibility. Nevertheless, this subsection will briefly introduce some of them in case they require some conceptual ideas or they are designed in some fundamental way which is critical for the mapping flow.

#### PyNN Integration

By design only the initial hardware setup description given in terms of a path language file (see Section 1.3.1) and a biological graph representing the neural architecture are necessary inputs for the MappingTool. Hence a translation from the PyNN description into the hardware independent biological graph is necessary. The corresponding translation is already available for the wafer-scale system and is utilized for operating the *FACETS Executable System Specification* [*Vogginger*, 2010]. It can be reused for the multi-chip system. Thus, unnecessary code redundancy is avoided and the biological graph is implicitly standardized for both systems.

#### Low-Level Software Integration

The Spikey chips as described in Section 1.2.3 are configured by playing back a command sequence to the chip, which is stored in the FPGA memory. For input stimuli, the precise time of the first occurring event depends on the preceding commands in the memory. Such commands may result from the initial configuration or previously performed tasks. Therefore, the synchronous start of the experiment relies not only on a synchronous global time but also on the history of every individual chip. In most occasions this scenario would lead to asynchrony. Hence, in contrast to the single-chip operation, the memory of an FPGA needs to be erased after the chip configuration and before the transmission of the input stimuli to swipe their history. Thereby, after setting all Nathan clocks to a global time and triggering the experiment via the GSS (see Section 1.1.3), coincident events on different chips also physically occur coincidentally.

#### Experiment Controller

After the preparative algorithmic tasks have been successfully performed the experiment needs to be configured on the hardware and is consequently triggered. In a multi-chip setup some more things need to be taken into consideration than for the single-chip operation. After the extraction of parameters from the graph model, the low-level API representations of Spikey chips and spike trains need to be configured appropriately. Furthermore, the network needs to be configured and the clocks need to be synchronized to provide a global time.

Finally the experiment is triggered by the GSS which is emitted by one of the involved Nathans. This way the playback of the input stimuli starts synchronously. Hereby the duration of the experiment is determined by the occurrence of the last input spike. As each Nathan has

its own input an identical dummy end-of-experiment event is inserted into the input sequences. Otherwise some of the chips might disable their output buffers too early and network dynamics driven over the network by events from adjacent chips would be ignored.

After the experiment has been carried out the software needs to collect all output spike trains emitted by each participating chip and insert them back into the GraphModel. This is necessary to be compliant with the concept of holding all experiment related data within the graph representation.

Depending on the setup it should now be possible to either rerun the same experiment, thereby performing a differential remapping and carrying out the modified experiment, or return to the PyNN layer.

## 2.2. Implementing the Multi-Chip Environment

Section 1.1 introduces the state of development in the established software frameworks for single-chip operation and the FACETS wafer-scale system. This section presents the implementation of the multi-chip system based on the fundamental concepts presented in Section 2.1.

### 2.2.1. General Framework

Some of the tools developed for this thesis contribute to the general toolkit used for the operation of both the chip-based and the wafer-scale system. For the sake of completeness they are introduced briefly in the following subsections.

#### Message Logging Interface

Information management in extensive software frameworks is a severe problem that becomes even more critical with concurrent execution. Hence it is desirable to control the information flow in a centralized manner depending on the level of information required for the current operation. Resolving misbehavior in software often requires a more detailed output, which in turn can generate an overwhelming amount of information or even slow down the program execution.

The implemented `C++` logger class enables the user to decide the level of output information granularity for any module within the MappingTool and multi-chip operation. By casting the implementation into a singleton pattern [*Gamma et al.*, 1994], the instantiation is restricted to a single entity to guarantee the unified information filtering. The first initialization determines the logging threshold regulating the subsequent output verbosity. Any further alleged instantiation will just return a reference to the same instance and therefore the same operational properties. Only messages at least matching the threshold are printed to their destination (e.g. file and/or screen).

The logger class provides bindings for the programming languages `C`, `C++` and `Python`. Consequently, it integrates seamlessly into most existing software used in the Electronic Vision(s) group. To simplify the usage in its primary `C++` domain the message logging interface mimics the common Standard Template Library (STL) stream interface and is compatible to any STL compliant stream formatting. The correct operation in concurrent execution is achieved by internal locking mechanisms e.g. the singleton is protected against multiple concurrent initialization. Besides, `boost` thread local storage [*Kempf*, 2001] protects the STL-mimic stream interface, used for sequential message construction, from race conditions.

#### Unified Multi-Class Gigabit Network Configuration Tool

The available software to configure and manage the MCGN is completely written in the programming language `C`. In order to simplify the building process and to integrate the interface in the automated mapping and operation workflow, so that it is suitable for `C++`, an additional wrapping layer is introduced. Furthermore, a unified tool for the stand-alone management of the MCGN has been implemented utilizing the wrapped interface. It can be used to directly access common tasks like the programming of the routing tables and the synchronization of Nathan modules.

### 2.2.2. Multi-Chip Graph Structure

A specialized graph model structure representing the multi-chip system is the foundation which allows all other presented algorithms to perform the arbitrary neural network mapping. It also holds all parameters consistently so that repetitions of former experiments based on a stored graph representation can be performed. The starting point for the creation of the model is a user supplied GMPath language file describing the current setup. It lists the available FPGAs and Spikey chips with the corresponding chip id necessary for the parameter calibration (see Section 2.1.5). Based on the provided data the graph is unfolded by building representations of all configurable hardware entities in the setup. Sub-graphs for the MCGN and chips are attached to each FPGA node. Each chip node consistently supplies parameters e.g. for neurons and synapses. A template representation simplified for clarity is illustrated in Figure 2.2.

### 2.2.3. Multi-Chip Neuron Placement

The placement algorithms of the wafer-scale system can be reused for the multi-chip system as described in Section 2.1.3. From the modeler's point of view operating the device, the quantity of hardware-specific information required for its use should be reduced to a minimum. Therefore the currently available algorithms cluster and place the neuron populations autonomously. For the verification of the system it is crucial to have more manual control over the precise neuron placement. Consequently, to verify the correct operation of multi-chip software framework a more configurable placing algorithm has been implemented.

**Semi-Automated Population Placement** The population clustering placement algorithm enables the user to place populations of neurons via the *PyNN* high level API with matching `labels` on the same physical chip, while the exact placing of each neuron within each population remains random. Any neuron on the chip provides the same configurability and connectivity, thus in theory none of the random chip intern placements is preferable over another. Nevertheless, existing transistor-level variations are averaged out over multiple runs due to the random nature of the placement.

### 2.2.4. Multi-Chip Routing of Synaptic Connections

The routing step determines for every synaptic connection whether it should be realized or rejected. Connections which are desired for a given neural architecture and not yet ultimately realized are referred to as *requested*, while requested connections which have been granted during an intermediate step are referred to as *realized* for that specific step. Different connections

may require different amounts of hardware resources. In case the resources necessary to realize each requested connection exceed the hardware capacity, a suitable subset of connections is granted which maximizes the overall number of realized synapses. Finding this suitable subset is the actual task which has to be performed in the routing step.

For the single chip operation only the assignment of synapse drivers needs to be performed, while the distribution of network resources must be determined for the multi-chip operation. In the realized routing process this is achieved in three steps. Initially a network discovery is performed to determine available paths between any two configured Nathan nodes by means of a modified *Dijkstra* algorithm. Afterwards the synapse driver resources are shared among the requested intra- and inter-chip connections. Finally the acquired information is used to allocate the MCGN resources for the transmission of spikes over the network.

### Network Discovery

In the beginning of the routing step, the network topology for a given set of Nathan modules on one backplane needs to be determined. The topological structure of the MCGN network itself has been outlined in Section 1.1.3. Requisites for the network discovery are considered in Section 2.1.4.

The *shortest path* problem is solved by means of the Dijkstra algorithm, which is described in the following.

**Dijkstra's Algorithm** The implemented Dijkstra algorithm is a prominent representative in the class of algorithms designed to solve the shortest path problem. Thus, it can find the shortest path between any two nodes in a graph with non-negative weighted edges. The implemented variant of the algorithm is modified in the way that it does not only return a single shortest path, but all shortest paths with equal minimal length. Hence a wider range of equidistant routes are available to choose from. As a result the MCGN resource allocation described in Section 2.2.4 can utilize the network resources more efficiently in terms of leaving less unused transmission slots.

The algorithm iteratively visits each node within the graph and assigns the node its distance relative to the starting node and a predecessor node. After termination the minimal distance between any node and the starting node is known.

Initially the algorithm requires a node to start from. In our case every sending Nathan is considered as the starting node once. The algorithm then assigns from the starting node a distance value to any node in the graph: zero for itself and infinite for all other nodes. It further marks all nodes except the starting node as unvisited. Within the process each node can get a predecessor node assigned, which is not defined in the beginning.

After the initialization the starting node is considered as the current node, which refers to the node currently visited during the iterative process.

Now, for the current node the algorithm considers all neighboring nodes that have not been visited so far and calculates their relative distance to the starting node. The distance is given by the sum of the distance from the starting node to the current node and the weight value of the edge that connects considered and current node. In our case, with the weight equal one, this conforms to the number of hops in the network. If the recently calculated weight is less then the weight already assigned to the considered node then its distance value is updated and the current node is set as its predecessor. After having considered all neighboring nodes the current node is marked visited and will never be considered again. Its distance to the starting

node is already minimal. Of all considered neighboring nodes the node with the shortest assigned distance is picked as the next current node. The consideration step continues until all nodes have been visited exactly once.

The shortest path between any node in the graph and the starting node can now be determined by picking the target node and iterating backwards along the predecessors through the graph until one finally reaches the starting node.

In the modified version of the algorithm each node holds a list of equidistant predecessor nodes. Hence the backward iteration runs recursively over all predecessors and returns a list of equidistant shortest paths.

Two intermediate steps during the operation of Dijkstra's algorithm are illustrated in Figure 2.5. Additionally, a pseudocode representation of the realized variant can be found in Appendix B.1.

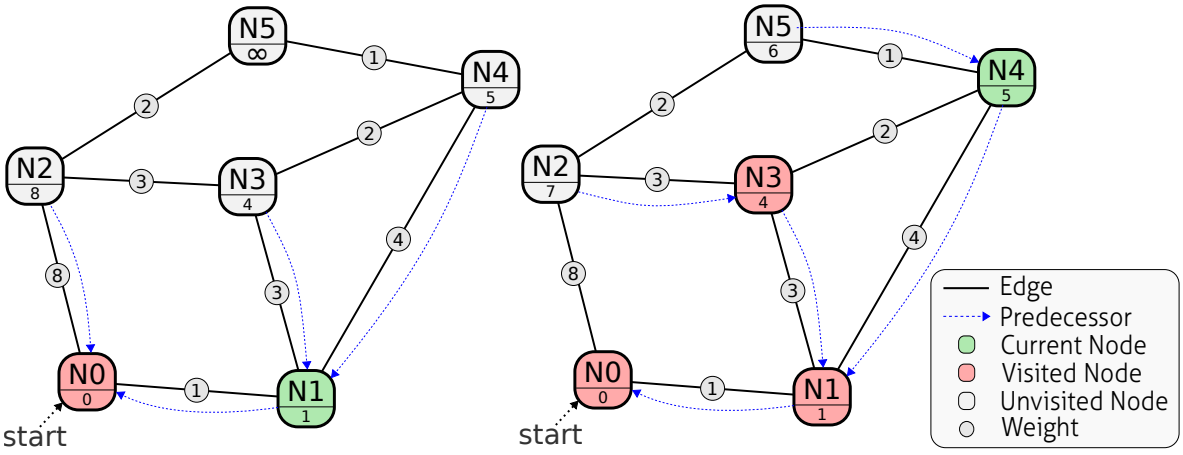


**Figure 2.5.:** Two intermediate states of a graph processed by Dijkstra's algorithm which finds the shortest path between the starting node and any other node. The algorithm processes iteratively all nodes in the graph. The shortest path is indicated by the dashed predecessor edges. For all nodes already marked as visited the indicated path is already minimal. Additionally, each node holds its current distance to the starting node, illustrated in the lower part of each node.

### Synapse Driver Allocation

The synapse driver allocation ultimately decides which connections to realize and how to distribute the synapse drivers among the realized connections. For the final routing only the distribution of realized connections to the network resources is missing. This last task is performed by the MCGN resource allocation as described in the next section.

The graph structure of the data model described in Section 1.3.1 is well suited for the application of recursive algorithms but it is not suitable for the employed multi-objective optimization algorithm. The algorithm requires a data container which allows for efficient random access instead. Therefore data necessary for the process is extracted from the graph and stored in a structure similar to connection matrices. In a common connection matrix the number of rows and columns matches the number of neurons present in a given neural architecture and non-zero entries represent synaptic connections. The realization of synapses terminating an already realized axonal connection or allocated synapse driver respectively is

cheap in terms of hardware resources compared to the new allocation of the synapse driver itself. For an already allocated synapse driver all requested synaptic connections to neurons on the corresponding core can be realized without any extra cost. Therefore we do not need to represent each post-synaptic neuron in our matrix for the optimization step. Each column of the underlying data structure represents a source neuron or external input from the FPGA event memory and each row corresponds to a target Spikey core. Desired connections are represented by the matrix entries. Each entry codes how many connections are requested from the corresponding source to the corresponding target core. An ultimately realized matrix entry corresponds to one allocated synapse driver with the appropriate synapse connections turned on. Hence the synapse driver allocation process needs to ensure that the number of realized entries per row stays below the number of available synapse drivers per Spikey core.

Due to construction the number of entries in the connection matrix is reduced, compared to a common connection matrix, by $\frac{n_{\text{neurons/chip}}}{n_{\text{cores/chip}}}$, which conforms to two orders of magnitude. The runtime performance of the employed optimization algorithm is equally accelerated, since the amount of performed computations per iteration is proportional to the number of matrix entries as described in the text below.

As motivated in Section 2.1.4 the optimization is performed via a simulated annealing process. In a nutshell, the algorithm is initialized with a random configuration of realized connections. In each algorithmic step the state is changed into a neighboring state in configuration space, with a preference for states with lower energy. Ultimately, the configuration state evolves towards lower energies and therefore less synapse loss.

In the beginning, requested connections $c$ are randomly established. The initialization routine already avoids conflicting configurations of connections arising from topology restrictions and bandwidth limitations. A configuration state $s = \{c_0, \cdots, c_n\}$ is represented by a corresponding set of established axonal connections. Two configuration states $s$ and $s'$ are called neighbors if they can be translated into one another by flipping the realization state of two synapse drivers $c_x$ and $c'_x$:

$$s = \{c_0, \cdots, c_x, \cdots, c_n\} \overset{\text{neighbors}}{\Longleftrightarrow} s' = \{c_0, \cdots, c'_x, \cdots, c_n\} \quad .$$

An energy $E(s)$ can be assigned to any state $s$. The energy is the linear combination of energies $E(c)$ for each established connection $c$ corresponding to the current state. Therefore the energy difference between two neighboring states can be expressed as:

$$\Delta E(s, s') = E(c_x) - E(c'_x) \quad .$$

The energy function $E(c)$ of a single established synapse driver can be tuned to match special requirements. In our case the energy function is given by:

$$\begin{aligned} E(c_{ijkl}) = &- A_{\text{Type}}(c_{ijkl}) \cdot \texttt{syn}(c_{ijkl}) \\ &- \sum_{m}^{\text{adjacent cores}} B \cdot \texttt{syn}(c_{ijml}) \\ &+ D \cdot \texttt{mcgn\_hops}(i,j)^2 \cdot \texttt{syn}(c_{ijkl}) \\ &- E \cdot \exp\left(\frac{\texttt{lower\_bound}(i,j) - \texttt{realized}(i,j)}{T_{\text{ext}}}\right) \end{aligned} \quad . \tag{2.1}$$

Where $c_{ijkl}$ denotes the connection from chip $i$ to the synapse driver with index $l$ on chip $j$ and core $k$. $\texttt{syn}(c_{ijkl})$ represents the number of post-synaptic connections realizable via the connection $c_{ijkl}$.

The first term on the right hand side adjusts the energy of a connection depending on the connections type.

$$A_{\text{Type}}(c_{ijkl}) = \begin{cases} A_{\text{intern}} & \text{if } i = j \\ A_{\text{input}} & \text{if } i = \text{memory} \\ A_{\text{extern}} & \text{if } i \neq j \end{cases} \quad .$$

Each connection is either *intern* if it is a connection between neurons within the same or adjacent cores, *input* it is a for stimuli injected from the FPGA memory or *extern* if it is a multi-chip connection over the network. With $A_{\text{Type}}$ different priorities can be assigned to connections of a certain type.

The realization of connections stimulating targets on multiple adjacent cores should be preferred, for external connections in particular. On the one hand it can be beneficial to support the realization of connections between neurons clustered together on the same chip by a preceding optimized placement with respect to its objective. On the other hand and more importantly connections with multiple targets on adjacent cores require less lookup table resources and bandwidth per synapse than the realization of multiple external connections each connected only to a single Spikey core. This issue is taken into consideration by the second term on the right hand side of Equation 2.1.

As mentioned before the MCGN resource allocation assigns short routes a higher effective bandwidth. The function $\texttt{mcgn\_hops}(i, j)$ reflects the necessary number of intermediate hops in the network between any two chips $i, j$. The necessary data has been acquired during the network discovery phase. Internal connections and event memory input obviously have a MCGN hop value equal to zero, which results in an active penalty of external connections with long routes.

The last term represents a soft bound for the minimal number of network connections to avoid the muting of chips, which is motivated in Section 2.1.4. It is further stated that one can expect muting for an optimized placing, since local, more dense connections would be prioritized by the routing to minimize synaptic loss. If during the annealing process the amount of realized connections between two chips $i$ and $j$ – given by $\texttt{realized}(i, j)$ – falls short of the lower bound, the energy for the realization of such connections decreases exponentially. The shape of the lower soft bound can be adjusted by a temperature parameter. Higher temperatures yield a softening of the bound and the muting of single chips becomes more likely. To avoid disproportionately strong bounds on connectivity for sparsely connected chips with counts for requested connection below the lower bound the actual limit needs to be adapted as follows:

$$\texttt{lower\_bound}(i, j) = min\left\{F_{\text{min}}, \texttt{requested}(i, j)\right\} \quad .$$

Where $\texttt{requested}(i, j)$ denotes the number of requested connections between chip $i$ and $j$. $F_{\text{min}}$ is a parameter which can be freely set to adjust the lower soft bound.

During the simulated annealing a configuration state $s^t$ is derived from an initial state $s^0$ by iteratively generating $t$ random neighboring states. To distribute the flipping of synapse drivers $c_x \Rightarrow c'_x$ equally over all originating synapse drivers they are picked subsequently from the connection matrix. One iteration step of the annealing process refers to a complete sweep over the matrix. Any computed neighboring configuration can be either accepted or rejected, with the acceptance probability:

$$P(s^t, s^{t+1}, T(t)) = \min\left\{\exp\left(-\frac{\Delta E(s^t, s^{t+1})}{T(t)}\right), 1\right\} \quad .$$

The time dependent temperature course $T(t)$ controls the acceptance probability for states with an effectively higher total energy. For $T \to -\infty$ only neighboring states with a lower energy are accepted while for $T \to \infty$ any neighboring state is accepted.

Operating with an appropriate temperature is crucial for the optimization quality. By choosing a too low temperature the annealing can get trapped in a local minimum without ever reaching the global one, which would correspond to the optimal solution. For a too high temperature the algorithm ends up in some random state, and never converges to a stable solution. An illustration of the temperature problem can be found in Figure 2.6. To avoid both extremes the temperature is reduced from an initial value $T(0)$ to zero over the complete annealing process. Different cooling strategies can be applied, but experiments (see Section 3.1.1) show that for the given task they have no significant influence. Hence the temperature is reduced linearly, since it requires only two parameters from the user.

$$T(t) = \begin{cases} T(0) \cdot \left(1 - \frac{t}{t_{\max}}\right) & \text{for } t < t_{\max} \\ 0 & \text{else} \end{cases} \quad .$$

As a result the states evolve along a trajectory in configuration state space that tends to configurations with a lower energy and converges ultimately for $t \to t_{\max}$ and therefore $T(t) \to 0$.

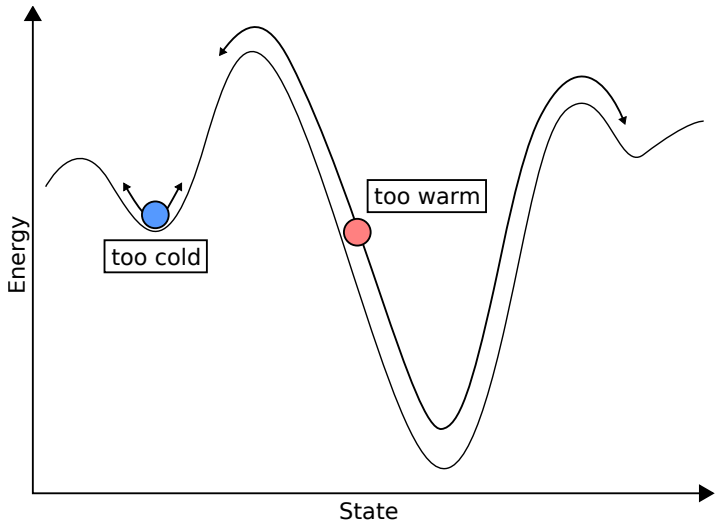


**Figure 2.6.:** Illustration of the problem of finding an optimal solution by means of the simulated annealing multi-objective optimization algorithm described in the text. If the temperature parameter is chosen too low, the optimization process can be trapped in local energy minima, while choosing the temperature to high yields random results.

As subsequent states are derived exclusively from the preceding state ($s^t \Rightarrow s^{t+1}$) the algorithm belongs to the class of *Markov-Chain* Monte Carlo algorithms [*Meyn and Tweedie*, 1993].

A pseudo code representation illustrating the fundamental concept of simulated annealing can be found in Listing Algorithm 1.

**Algorithm 1** Simulated Annealing

$s^t \leftarrow$ random_state( )
$E^t \leftarrow \text{Energy}(s^t)$
$t \leftarrow 0$
**while** $t < t_{\max}$ **and** $E^t > E_{\text{thresh}}$ **do**
  $s^{t+1} \leftarrow \text{neighbor}(s^t)$
  $E^{t+1} \leftarrow \text{Energy}(s^{t+1})$
  ▷ annealing acceptance probability
  **if** $\text{P}(E^t, E^{t+1}, \text{T}(t)) > \text{random}(\,)$ **then**
    $s^t \leftarrow s^{t+1}$
    $E^t \leftarrow E^{t+1}$
  $t \leftarrow t + 1$
**return** $s^t$

#### Multi-Class Gigabit Network Resource Allocation

Network resources need to be allocated for synaptic connections granted in the previous step. The problem is treated as a bin-packing problem and solved by means of a first-fit decreasing algorithm. It initially sorts the calculated routes in a decreasing order. It then tries to fit the routes subsequently into the available resources starting with the longest route. The sorting of available routes is possible as the topology of the network is fixed during the experiment and the resources do not need to be allocated dynamically. Before the algorithm stacks a route on top of already partially occupied resources it tries to fit one of the possible route variants between the other already allocated resources. The insertion step itself requires only $\mathcal{O}(n)$ computational complexity. The overall computational complexity of $\mathcal{O}(n \log n)$ is caused by the initial route sorting.

#### Post-Processing of Routing Data

So far the realization of synaptic connections and the distribution of the network resources have been described. During the post-processing, each realized inter-chip connection needs to be mapped to one of the allocated MCGN resources by choosing the user port and transmission time slot. An optimized mapping would balance the load on the bus. Since the event rates for the afferent neurons are not known *a priori*, the resources can be distributed either equally or linearly. A linear distribution means that the three Spikey output buffers are mapped in a linear manner to the available FPGA user ports. In that case, where the corresponding user port has more than one transmission time slot allocated for a connection bundle, the occurring events are transmitted automatically through any of the slots.

After all routing decisions have been made, the final configuration needs to be written back to the hardware graph, from where the modules responsible for the configuration and operation of the system can read it back as described in Section 2.2.6. The actual information is encoded by means of named edges (see Section 1.3.1) in the way depicted in Figure 2.7. Two different paths of edges provide a detailed mapping to the network, the remote synapse

driver and ultimately the target neuron. This differentiation into two separate paths shortens their effective lengths and speeds up subsequent computation, since the parameter translation requires only the synapse driver path while the MCGN configuration requires both. Besides the detailed description of a specific path running over all involved network port instances, a shortcut connection between the in and out FPGA user ports is provided to reduce the number of dereferencing steps during the network configuration. This reduces the number of required computational expensive GMPath language steps by a factor of approximately 4 (for general performance measurements see Section 3.1.2).

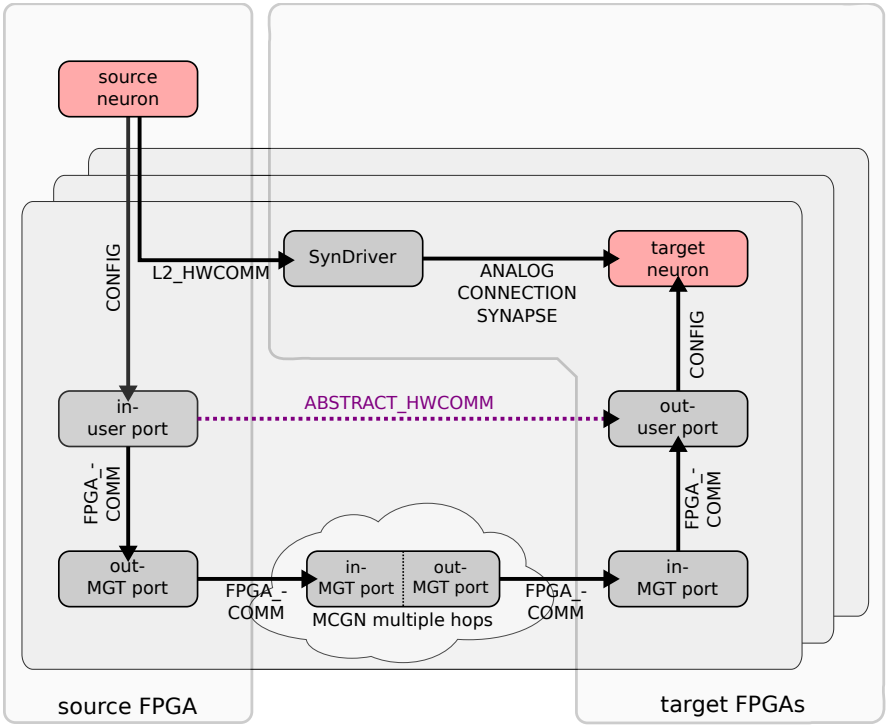


**Figure 2.7.:** Graph representation of a connection from one neuron to several target neurons on different FPGAs and chips. Note that beside the detailed route which runs over almost any involved instances (the mapping of the target synapse driver is handled via another edge), a shortcut connection (dashed) is provided to speed up the graph parsing process.

### 2.2.5. Multi-Chip Parameter Translation

After the successful placing of neurons, the routing of synaptic connections and before the actual experiment execution, the hardware needs to be configured with a suitable set of parameters. Finding an optimal set can be challenging as described in Section 1.3.1 and Section 2.1.5. For each individual chip a custom translation is applied which accounts implicitly for imperfections and transistor-level variations of the underlying substrate. Besides the translations from the hardware into the biological domain a reverse translation is required to translate the output obtained by an experiment back into the biological domain. The corresponding transformations belong to the post-experiment processing and are described in Section 2.2.6. A complete list of chip parameters that need to be configured can be found in Appendix A.

### Database Access

In Section 2.1.5 the change from a file-based system to a central database for the storage and management of calibration data is explained. The database system will be used in the wafer scale system, too. Thus, the realized database access interface is designed to already provide the required flexibility to be used for both hardware systems. It offers thread-safe caching functionality to account for concurrent access by multiple threads to minimize expensive database queries, while parallel access by multiple host computers is provided by the database itself.

MongoDB can store arbitrary sets of data. Nevertheless, the parameter translation expects the translation data to be in a certain format. Each entry provides the necessary data together with a reference to the corresponding translation function. Hence each parameter can be translated by means of an individual translation. The hitherto framework provides functions for constant, linear, polynomial parameters translations as well as interpolation and clipping to neighboring data points. Furthermore, each entry contains the respective parameter limits. The realized framework automatically clips parameters exceeding the hardware specifications. The same applies for discrete parameters. Each entry provides a resolution field so that each non-continuous parameter can be rounded stochastically to one of its neighboring values. The probability of a continuous value to be rounded to the floor or the ceiling is proportional to its distance from the respective discrete neighbor.

### Biology to Hardware-Domain Translation

The translation of parameters is an integral part in the mapping flow. The given large-scale neuromorphic hardware systems provide extensive parameter spaces which need to be configured conscientiously before any experiment.

Initially all necessary data is collected from both the biological and hardware graph. References to the collected data are stored in more suitable, continuous containers. Subsequently the translation of parameters is started. It performs on a per-chip level in parallel, as the multi-chip system derives from a single-chip system, parameters can be set individually for each chip. Hence, the software does not need to account for conflicts arising from the interplay of chips.

In the end the translation finds for every neuron, axon and synapse its correct parameters by the translation provided by the central database. Aside from that the software module provides default parameters for unused analog entities to avoid side effects which could arise from operating parts of the system out of their specification.

Practice has shown that the time consumption of the parameter translation is critical for the runtime performance of the complete mapping process, due to the fact that a significant large amount of GraphModel nodes need to be read, processed and written back. For example the weights of each synapse block consume $49152 \cdot 4\,\text{bit} = 192\,\text{kB}$ of memory and thus fit in the cache of a modern micro processor. Following the GraphModel approach, each synapse would be represented by two nodes in a key-value manner. Storing this amount of data in a non-continuous way would slow down the process tremendously. To circumvent the obstacle the GraphModel has been extended by a template node. The so-called `GMNodeData<>` inherits all basic behaviors from the common nodes but can additionally hold arbitrary sets of data like memory aligned, continuous containers, such that computational intensive tasks can be

processed more efficiently by making use of the new type of nodes. They have already been adopted by the wafer-scale flow in many places.

### 2.2.6. Experiment Control

So far all preparative tasks have been finished. At this point, it is time to collect all necessary data, configure the system and trigger the experiment.

The experiment control performs subsequently the following tasks: the configuration of each chip, the transmission of input stimuli to the event memory, the configuration and synchronization of the network, the synchronous triggering of the experiment start and the retrieval of experiment results.

The experiment control module for the FACETS wafer-scale system and *Executable System Specification* is MappingTool extern, while for the multi-chip system the module is part of the MappingTool itself. It is thus more modular and allows for faster and simpler differential remapping in future implementations. Differential remapping is not yet realized at the time of thesis submission.

**Chip Configuration** The translated parameters are extracted from the GraphModel and transfered to the hardware via the SpikeyHAL. The same applies for the input spike trains, which have already been prepared and stored in a hardware compatible format attached to template nodes in the hardware graph. This allows for efficient parsing and input processing in a *zero-copy* fashion. Additionally a multiplexer, which is responsible for the mapping of the membrane potential onto an output pin, is set appropriately. Likewise the parameter translation the chip configuration is performed individually, making software parallelization possible.

**Network Configuration** The precise route realizations, previously determined by the routing procedure, are extracted from the graph structure. The routing tables for the MCGN are programmed accordingly. The required low level software has been written in `C` [*Philipp*, 2008]. For a seamless integration into the new flow the according software module as well as the module for the Nathan clock synchronization are wrapped with `C++` classes, which can be handled more easily by the linker during the build process.

On top of that the event distribution network is configured for the actual transmission of event data via the priorly programmed MCGN. Thus, sender neuron and receiving synapse driver are identified as well as the sender and receiver user port and the respective time slot via which the network can be accessed. Based on this information connection bundles are tied together and a mfi as well as a lvc are provided for each one (see Section 1.1.4). As mfi identifies a connection bundle on the sender side, an internal lookup table for each sender Nathan and user port combination is provided. If a certain combination is required but not yet in use the next free value for mfi is picked. This applies in a very similar way for lvc, but for combinations of receiver Nathan and receiver user port. The `subnr`, which identifies the connection within its bundle, is looked up in a table that counts the connections realized so far for combinations of sender and receiver Nathan. The valid combinations of sender, receiver and lookup table indices are then submitted to the hardware internal lookup tables, which provide the data necessary for the live event processing described in Section 1.1.4.

Finally after programming both network layers the clock of each individual Nathan is set to a global time. The required functionality is also provided by the MCGN.

#### Experiment Execution

To trigger the experiment run the playback of the input stimuli needs to be started coincidentally on each Nathan. Therefore the Nathan modules are put to listening mode, where they wait for a so-called GSS to appear on the MGT links. One of the involved modules is destined to provide the GSS, while waiting for an external start signal itself. If so far no major problems have occurred the framework instructs the destined Nathan to trigger the experiment execution and the synchronous spike train playback respectively. This happens in a blocking fashion, that means that the execution of the program is paused until the block is released by the hardware signaling the completion of the experimental run.

#### Post-Experiment Processing

The experiment is completed after the playback of the last input spike (see Section 2.2.6). Consequently each Spikey disables its spike output buffer, hence no events are forwarded to the network anymore. By contrast the chip internal dynamics continue unnoticed due to the analog nature of the circuits. This has no impact on the performed experiment but needs to be taken into consideration for subsequent experiments to avoid distortions driven by ongoing activity.

The output spike trains are extracted from each Spikey. Each spike train is translated from the hardware domain back into the respective biological domain in a single step. The spike trains are then filtered for spikes which actually correspond to a configured hardware neuron. This is necessary due to ghost events which irregularly occur as described in Section 3.1.3. The result is attached to the corresponding neuron in the biological graph. This is performed by means of a strict zero-copy policy. This strict zero-copy policy is essential, because unnecessary memory accesses rapidly become the limiting bottleneck for the operation of highly accelerated neuromorphic hardware systems [*FACETS D7-13*, 2010]. Ultimately the output is provided transparently via PyNN and the program execution returns to the python layer.

## 2.3. Liquid State Machines on the FACETS Chip-Based System

This section introduces a Liquid State Machine based on a self-stabilizing neural architecture proposed by *Sussillo et al.* [2007], which will be implemented on the single-chip system and investigated with a focus on the readout and classification part. The fundamental concepts behind the LSM have been outlined in Section 1.4. Furthermore, the task is defined which needs to be solved by the LSM. Finally the realization of a perceptron and an on-chip tempotron classifiers are described.

### 2.3.1. Task Definition

To evaluate the quality of a liquid emulated on the hardware system, a task is being defined and the corresponding input stream presented to the neural network. Its spike activity response is presented to a readout and the classification result is compared against the readout performance without a liquid. In our case, the applied task is to distinguish slices of two shuffled spike trains. The classification is performed by multiple readouts, each trained to classify the templates in a specific time slice in the recent past by considering only the latest slice. A schematic illustrating the generation of the input stimuli can be found in Figure 2.8.

Initially, two $N$ dimensional Poisson process spike sequences vectors [*Downarowicz*, 2008] for $N$ spike sources are generated with a mean rate of 30 Hz and a total length of 1250 ms. Each sequence vector is cut into ordered time slices of 50 ms. Now we have two sets of template slices (A/B) for each of the 25 time slices to choose from. New spike trains are generated by randomly picking one of the two template sets for each time slice. Additionally, a normally distributed time jitter with a standard deviation of $\sigma = 2$ ms is added to each spike time. After that, all picked template sets are concatenated, so that the respective position in time of each slice is conserved.

A set of $N$ target neurons are randomly picked from the liquid architecture. The readily assembled spike train vectors are then projected onto those neurons, one distinct spike train each. The response of the liquid to each input spiketrain is recorded. Each readout is trained to classify the originating templates (A/B) for a specific time slice only by considering the last 50 ms. The further their corresponding time slice is located in the past, the harder the classification task gets. When using a memoryless readout a *memory capacity* of the liquid can be defined by the maximum time for which a classification beyond chance level is observed. Here the term chance level refers the percentage of correct classification one would achieve by random guessing.

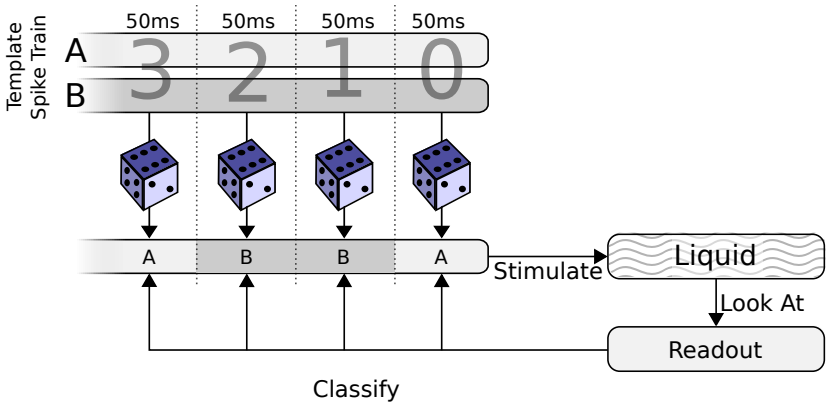


**Figure 2.8.:** An illustration of the applied task. Spike trains are generated by picking 50 ms slices randomly from two template spike sequences (A/B). The generated spike trains are projected into the liquid and the last 50 ms of each response is presented to readouts. Each readout is trained to classify the originating template (A/B) for one specific time slice in the recent past.

### 2.3.2. Liquid Architectures

The main functionality a liquid has to provide is the separation of presented inputs by projecting them into a high dimensional space. A large variety of neural architectures are potential candidates for a liquid, as long as they satisfy the separation property, which is defined in Section 1.4.1. Often neural architectures are tuned to work properly on a certain type and intensity of input. Too strong input for example can lead to chaotic behavior while too weak input might not elicit spiking activity from the architecture.

| Projection | | | STP Mode |
|---|---|---|---|
| E | → | E | depressing |
| E | → | I | facilitating |
| I | → | I | depressing |
| I | → | E | facilitating |

**Table 2.1.:** The Short-Term Plasticity configuration of the two population E (excitatory) and I (inhibitory), as proposed by *Sussillo et al.* [2007] for the self-stabilizing architecture.

### Self-Stabilizing Network

The self-stabilizing architecture, which in the following will serve as a liquid, consists of two neuron populations, one excitatory and one inhibitory, each comprising recurrent connections as well as connections onto each other. The architecture has been originally proposed by *Sussillo et al.* [2007]. An adaption has already been proven to show good-natured properties on the FACETS chip-based hardware system [*Bill*, 2008; *Bill et al.*, 2010].

STP functionality is the key to the self-stabilizing properties of the network. The architecture utilizes synapse dynamics as listed in Table 2.1. On the one hand, strong activity is attenuated by the depressing recurrent connections and the indirect feedback over the inhibitory population via stronger inhibition. Weak activity, on the other hand, is amplified by means of less attenuated recurrent excitation and less feedback from the inhibitory population. As has been shown by *Sussillo et al.* [2007], these self-stabilizing properties enable the architecture to operate within a wide input range.

The original hardware implementation required both cores of the Spikey chip to provide all desired STP functionality. Each synapse driver can only be either inhibitory or excitatory and can only be connected to a specific neuron on each core. Therefore the inhibitory and excitatory neurons needed to separated on the two different cores with non-overlapping neuron ids. In the current revision only one core is available due to a chip design error. It is assumed that the depressing STP functionality for the excitatory recurrent connections is most valuable for the self-stabilizing properties. Thus, from the requested plasticity features, only the depressing excitatory connections are realized in the current hardware implementation. Furthermore, the projections onto the inhibitory population share the plasticity mechanism due to hardware limitations. These limitations can be overcome in a future implementation by making use of a hardware feature that allows to combine synapse drivers such that one driver can provide depressing and the other one facilitating synapse dynamics. Nevertheless, this would cut the network size by half due to the fixed synapse driver routing topology for local neurons.

Figure 2.9 shows the self-stabilizing architecture as currently realized on the chip-based hardware system.

### Column-Based Liquid Architecture

Beside the self-stabilizing liquid architecture presented above, an alternative architecture has been utilized on the hardware system. It is being discussed only briefly, since no explicit investigations have been made on basis of this architecture for this thesis. The liquid is based on a column-shaped pool of neurons, of both inhibitory and excitatory type. The connection probability between two neurons decreases exponentially with their distance. The structure

| | |
|---|---|
| $N_{\text{excitatory}}$ | 144 |
| $N_{\text{inhibitory}}$ | 48 |
| $p_{\text{ee}}/p_{\text{ei}}/p_{\text{ie}}/p_{\text{ii}}$ | 0.05/0.1/0.1/0.2 |
| $g_{\text{ee}}/g_{\text{ei}}/g_{\text{ie}}/g_{\text{ii}}$ [$\mu$S] | 0.002/0.001/0.0015/0.002 |

**Table 2.2.:** Specifications of the self-stabilizing network architecture. Where $N$ represents the number of neurons. $p_{\text{xy}}$ refers to the connection probabilities and $\omega_{\text{xy}}$ to the synaptic efficacies for the connections from population $x$ to population $y$.

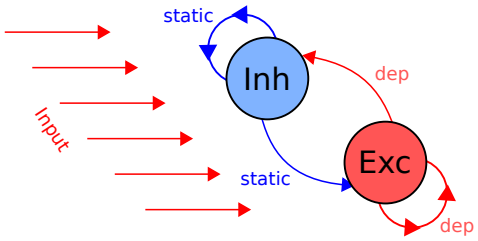


**Figure 2.9.:** A self-stabilizing architecture comprising two populations of neurons, one excitatory and one inhibitory. Each population has local recurrent connections and global connections to the other population. STP synapse dynamics are utilized to stabilize network activity for a wide range of input intensity. Weak activity is amplified, while strong activity is attenuated. It has been originally proposed by *Sussillo et al.* [2007] and realized on the FACETS chip-based hardware system by *Bill et al.* [2010].

has been originally proposed by *Maass et al.* [2002] and has been realized on the FACETS chip-based system by *Albert* [2010].

Since the network does not comprise self-stabilizing functionality, except for a pool of inhibitory neurons within each column, it is more sensitive to hardware and input fluctuations compared to the previously described self-stabilizing architecture. Figure 2.10 illustrates the basic structure of the column-based architecture.

### 2.3.3. Readout Realization

The last component of a LSM when considering the flow of information is the classifier. It is responsible for carrying out the actual task of distinguishing the input patterns. For this purpose two classifiers are implemented and tested for the presented hardware setup, a perceptron and a tempotron. Their fundamental concepts have been described in Section 1.4.3.

#### Perceptron Classifier

The implemented perceptron provides the functionality described in Section 1.4.3 and has been written in Python.

The input as read from the spiking liquid needs to be converted first from a spike-based to a rate-based representation in order to apply it to the perceptron. This can for instance be achieved by performing a convolution of the spike sequence with an exponentially decaying function:

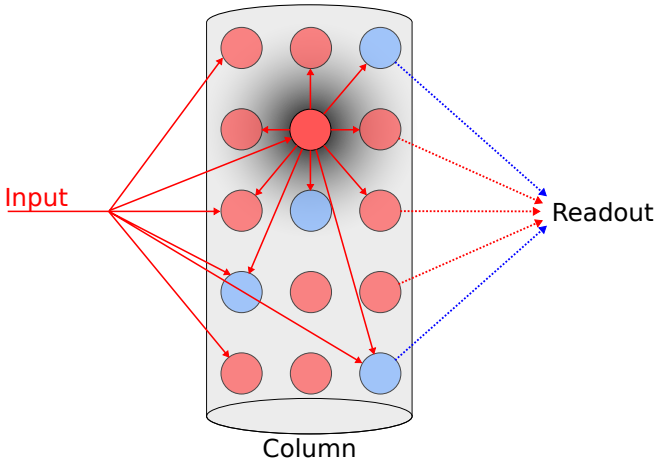


**Figure 2.10.:** A LSM with a column-based liquid proposed by W. Maass [*Maass et al.*, 2002]. Each column comprises both excitatory and inhibitory cells aligned on a three dimensional grid. Connection probabilities between any two cells within each column decrease exponentially with their mutual distance. Input stimuli are projected onto the columns. The responses of the columns is ultimately presented to a classifier that carries out the actual filter task.

$$x_i(t) = \sum_{t_i<t} c \cdot \exp(-\frac{t-t_i}{\tau}) \qquad .$$

Here $x_i(t_0)$ denotes the $i$th input presented to the classifier at time $t_0$. $t_i$ refers to the spikes emitted by the $i$th afferent neuron. $c$ is a free scaling factor and therefore set to one in the following. The time constant $\tau$ characterizes the decay of the contribution per spike to the input signal as time evolves. Note that this adds an exponentially decaying memory with time constant $\tau$ to the input. This can be critical for the memory evaluation of the investigated liquid. $\tau \to \infty$ yields an ever-lasting input memory, while for $\tau \to 0$ the input vector approaches $\vec{0}$ and no information is provided to the perceptron. Hence the experimentalist has to carefully choose a time constant which on the one hand does not masquerade the memory capacity of the liquid and on the other hand yields classifiable input for the perceptron. It is assumed that time constants $\tau$ on a time scale similar to the ones found in the participating neurons is a good point to start further optimizations from.

Even though the perceptron is relatively simple, it has been proven to be a useful tool for the rapid analysis of liquid responses. Because of its low computational complexity, the quality of a certain liquid can be judged already after a few seconds real time.

### Tempotron Classifier

The software tempotron is realized via PyNN, using the simulator backend NEURON [*Hines and Carnevale*, 2006] to carry out the computation. Any supported spiking neuron type can easily be employed as a building block for the tempotron. With respect to a potential hardware realization, LIF neurons with conductance-based synapses are of particular interest. In contrast, the originally proposed tempotron [*Gütig and Sompolinsky*, 2006] is based on LIF neurons with current-based synapses. The deduction of the learning rule – presented in Section 1.4.3 – is in principle only valid for the current-based type. An according deduction for the

voltage course of a LIF neuron with conductance-based synapses is non-trivial. Furthermore, the original deduction required the smooth resetting after the first neural firing within the classification time window, which is not feasible in hardware.

To simplify the transition from a software to a hardware readout the realized tempotron accounts for additional hardware limitations. First of all, the PyNN neuron type specialized for the FACETS hardware is used, which is in principle a standard LIF neuron with conductance-based synapses, but also includes some modification like for example setting $C_m$ to a fixed value of 0.2 nF (compare Equation 1.1).

The major concern for the realization of a classifier on the hardware are the available ranges for synaptic weights. Both their maximum strength and their resolution are limited. Commonly classifier learning rules try to increase the contrast in the readout response between applied input patterns of different classes by tuning the synaptic efficacies accordingly. If it is allowed to, such learning processes may tune synaptic efficacies beyond biologically plausible regimes, and in our case technically even more critical: beyond hardware limitations. The same applies for the tempotron learning process. It continues until a defined maximum of training iterations is reached or a stable state is approached where the classification is perfectly correct.

Tests have further shown that the tempotron classification performs on chance level if the synapses are allowed to flip from excitatory to inhibitory type and vice versa. The reason of this lies in the already mentioned differences between the current-based synapses in the original model and the conductance-based ones on the Spikey chip. The contribution of a single conductance-based synapse to the neuron membrane is the solution of the equation

$$C_m \frac{\mathrm{d}V}{\mathrm{dt}} = -g_l(V - E_l) + I_{\mathrm{syn}}$$

with

$$I_{\mathrm{syn}} = \omega_i g_0 \left(V - E\right) \exp\left(-\frac{t - t_i}{\tau_s}\right) \qquad \text{for } t \geq t_i \qquad ,$$

which can be can be written as:

$$V(t) = E_l \exp\left(-\frac{t - t_i}{\tau_m}\right) + \frac{1}{C_m} \int_{t_i}^{t} \omega_i g_0 \left(V - E\right) \exp\left(-\frac{t' - t_i}{\tau_s}\right) dt' \qquad .$$

Here $V$ is the membrane potential, $E_l$ the leakage potential, $\tau_m$ the membrane time constant, $C_m$ the membrane capacitance, $\omega_i$ the synaptic weight, $g_0$ the base synaptic conductance, $t_i$ the time of the afferent spike which activates the synapse, $E$ the reversal potential (either excitatory or inhibitory, depending on the synapse type) and $\tau_s$ the synaptic time constant. In the case of the current-based synapses, the voltage-dependent term in the integral disappears, making an analytic solution straightforward, as described in Section 1.4.3. Even assuming that the voltage term remains approximately constant during a Post-Synaptic Potential (PSP) (which may be plausible if there are many, relatively weak synapses, but is definitely not true for e.g. the triplet scenarios described in *Gütig and Sompolinsky*, 2006) and can thus be shifted out of the integral, synaptic contributions remain voltage-dependent:

$$V(t) \approx \omega_i \cdot (E - V(t_i)) \cdot K(t - t_i) \qquad ,$$

with the kernel $K$ as described in Equation 1.6.

As this voltage dependence is not taken into account in the learning process, this may significantly alter learning dynamics, especially in cases where the weight update rule switches synapse types from excitatory to inhibitory or vice versa. In such cases the leaning efficacy might non-continuously change between two consecutive learning steps by a factor of $\approx 7$ (for typical values $E_i = -80\,\mathrm{mv}, \overline{V} \approx -70\,\mathrm{mV}$ and $E_e = 0\,\mathrm{mV}$) for a single synapse. This may impose drastic consequences for the convergence of the learning process.

Additionally, synaptic hardware weights have only a resolution of 4 bit each (see Section 1.1). In case the learning rule requires fine granular tuning of weights, this could become a problem. Such a requirement would be inept beyond merely hardware weight resolution, due to the hardware variations and noise.

Consequently, it needs to be shown that a constrained tempotron can classify beyond chance level despite all described limitations. The software tempotron replicates all discussed issues as closely as possible. It accounts for the limited range of weights by setting each weight which exceed hardware limits back to the maximum hardware weight after each learning step. The precise maximum weight in hardware depends on the chosen calibration. The efficacy could be increased in hardware by changing for example the shape of a PSP.

To compensate the flipping of weights, the software tempotron simply checks the value of each weight after each learning step and sets negative weights to zero. Hence, weight flips are banned and the synapse type remains constant. For the dynamic input range it would be beneficial to have both excitatory and inhibitory synapses, but it is not *a priori* clear for which distribution of synapse types the contrast between the input patterns is effectively increased. Thus, all synapses are initialized excitatory. A potentially more sophisticated way to tackle this issue in future implementations is discussed in the outlook of this thesis.

For the last issue – the weight discretization – the tempotron accounts by clipping the weights already in software either after each learning step to discrete values or after the complete training in a student-teacher manner (see Section 1.4.3). The clipping distributes the 16 available weights linearly between zero and the maximum hardware weight.

The simulation results presented in Section 3.2.1 show that despite the constraints the tempotron trained such and afterwards realized in hardware can classify significantly better than chance level.

# 3. Experimental Results

In this chapter a variety of experiments is presented. The first set of experiments shows the correctness of the biology-to-multi-Spikey mapping procedure and provides estimates for the general performance and scalability of the novel multi-chip framework introduced in Section 2.1 and Section 2.2.

The second set of experiments focuses on Liquid State Machines on the FACETS chip-based hardware system and in particular the associated readout. The major aim of these studies is the realization of a spike-based classifier that can be directly run on the hardware system.

## 3.1. Verification and Performance Analysis of the Multi-Chip-System

The experiments presented in this section are carried out to demonstrate the basic functionality of the multi-chip system. First of all, the performance of the routing step (see Section 2.2.4) is evaluated both in terms of routing result quality and algorithm convergence speed. Then the general runtime performance of the complete mapping flow is analyzed to identify bottlenecks and outline possible improvements for future implementations. In the end of the section, a Synfire-Chain experiment (see Section 3.1.4) is performed to verify the correct configuration of the hardware system.

### 3.1.1. Routing Performance Analysis

The goal of this analysis is to investigate the performance of the routing algorithm as described in Section 2.2.4. To consider all objectives of the optimization process comprehensive investigations are necessary. The first experiment evaluates the performance in terms of routing quality. The second experiment analyzes the time necessary to reach a stable routing configuration.

These performance measurements are based on the KTH attractor model by *Lundqvist et al.* [2006]. It is part of the benchmark library, which is used to optimize the neuromorphic mapping workflow for the FACETS wafer-scale system (see Section 1.1.5) and the executable system specification [*Vogginger*, 2010].

**Applied Benchmark Architecture** The maximum number of realizable synapses heavily depends on the utilized neural architecture. Consider an architecture with homogeneous connectivity between all neurons, i.e. each neuron has the same probability to be connected to any other neuron. Without any asymmetries in the original network, an optimizing placement or routing algorithm has usually hardly any possibility to optimize by appropriately arranging the cells or axonal connections. It would hardly matter which neuron is placed on which chip. The same applies for the routing: the synapse loss would remain nearly the same independent of the actually realized synapses. Consequently, the network that is used as a

| Building Block \ Network Size | 384 | 768 | 1536 |
|---|---|---|---|
| Hyper-Columns (HC) | 6 | 8 | 12 |
| Mini-Columns (MC) per HC | 4 | 6 | 8 |
| Pyramidal Cells per MC | 12 | 12 | 12 |
| RSNP Cells per MC | 2 | 3 | 3 |
| Basket Cells per HC | 6 | 6 | 8 |

**Table 3.1.:** Different building block configurations of the employed KTH attractor models in three sizes to test the routing performance.

routing benchmark requires structural inhomogeneities, e.g. clusters of neurons with higher local than global connection densities. The synapse loss after a random initialization can then be compared against the synapse loss after the optimization. Obviously, the achievable minimal synapse loss depends on the placing of neurons. A placing that conserves the architecture intrinsic clustering of neurons with high connection densities can reduce the final synapse loss.

From the benchmark model library the KTH attractor model has been chosen to carry out the analysis. For details on the model characteristics refer to *Lundqvist et al.* [2006]. The network specific dynamics are not of particular interest for the experiments presented here. It merely serves as a benchmark model with neuroscientific relevance. Its implementation as originally proposed might not be necessarily suitable for actual multi-chip experiments, because the typical connection densities between neurons easily exceed the limits of the event network. A future neuroscientifically relevant implementation of the model on the multi-chip system would require a tuning of the model, which is beyond the scope of this work.

Consequently, the size of the KTH attractor model is reduced to fit onto three differently sized setups. Figure 3.1 illustrates the fundamental structure of the KTH model. Three different KTH setups with varying sizes of the KTH building blocks are mapped as listed in Table 3.1.

The routing software can handle arbitrary numbers of Spikey cores per chip. Nevertheless, it is assumed throughout the routing benchmark experiments, that only one core is available per chip. The availability of multiple cores would reduce the overall synapse loss. The assumption has been made with regard to network experiments on the current Spikey revision: For the third revision of the chip only one of the two cores is available, as described in Section 3.1.3. Therefore, results provided by the presented experiments offer a conservative estimate of synapse loss.

#### Routing Quality Evaluation

The purpose of this experiment is to evaluate the routing quality in terms of synapse loss reduction compared to a random routing configuration. Thus, the synapse loss of the optimized configuration state is measured in dependence of the two free temperature parameters $T$ and $T_{\text{ext}}$, which are relevant for the configuration state acceptance of the annealing process (see Section 2.2.4), and the shape of the soft bound, which controls the muting of inter-chip connections.

For the free routing parameters listed in Section 2.2.4 it is assumed that the temperature parameters express more general characteristics than the others. For example the individual

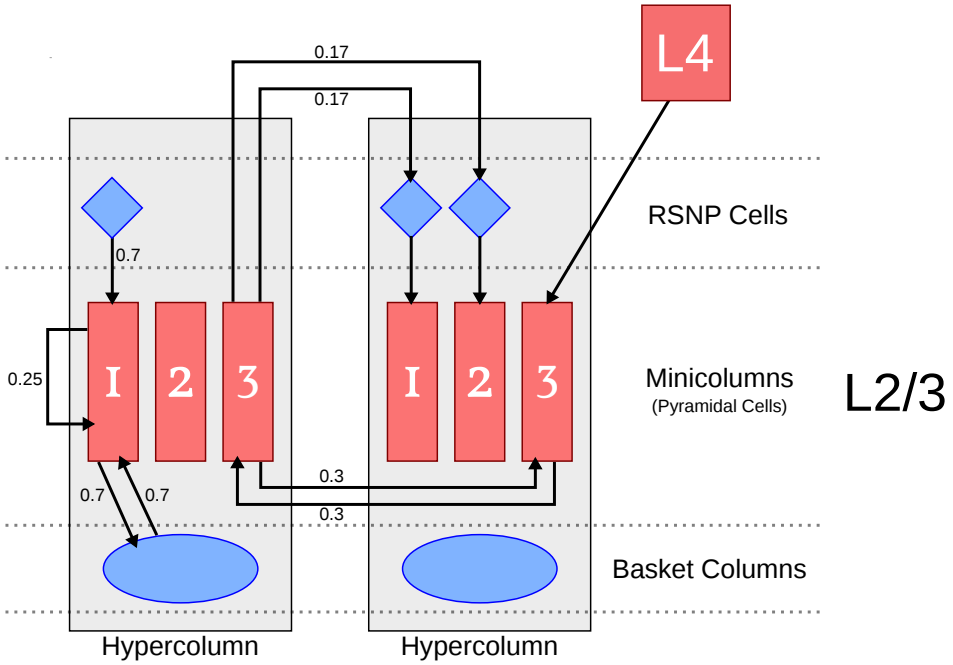


**Figure 3.1.:** The cortically inspired Layer-2/3 KTH attractor model. It combines a soft WTA within each hypercolumn with a long-range strong WTA between the hypercolumns. At any time only one minicolumn per hypercolumn can be active. The model is used to benchmark the routing quality and the runtime performance of the mapping framework. Figure by Mihai Petrovici.

tuning of a certain connection type and the actual lower soft bound heavily rely on the applied neural architecture and the experimentalist's preferences. Consequently they have not been analyzed selectively. In this setup the different types of connections are weighted equally with 1 and the lower bound is set to 10 (compare Section 2.2.4). The same applies for the constants of proportionality in each energy term: They are set to a value of 1.

For every KTH test setup, the random initialization of the synapse driver configuration is performed 100 times, the mean synapse loss and the standard deviation are determined to reduce fluctuations due to the random nature. For the optimization process an initialization with less than $0.1\sigma$ deviation from the mean synapse loss of the random series is picked as a configuration to start from.

The temperatures $T$ and $T_{\text{ext}}$ controlling the state acceptance probability and the adherence of the lower soft bound for external connections are sweeped systematically. Every sampling point corresponds to 10 optimizations with 500 algorithmic iterations each. Considering the analysis further below in the text, 500 iterations per run have been shown to be sufficient to reach a stable configuration state.

**Observables** In a holistic evaluation of the routing quality it is not enough to look only at the synapse loss, but one has to consider all routing optimization objectives (see Section 2.2.4). In the previous considerations the connection type prioritization has been neglected due to its particularity. This has been achieved by setting the respective weights to 1. Consequently, only the synapse loss and the muting of inter-chip connections needed to be considered.

*synapse loss*: The first and most obvious observable is the synapse loss. It reflects the number of synapses that can not be realized for a specific configuration. The loss arises not only from hardware limitations. It can also be caused or amplified by non-optimal configurations. The latter suggests the use of the synapse loss as a primary measure for the mapping quality evaluation.

*muting*: To quantify the degree of inter-chip connectivity distortion (also referred to as *muting*) the observable $m$ is introduced, which reflects the number of realized connections $\texttt{realized}(ij)$ from chip $i$ to chip $j$ that fall short of $\texttt{lower\_bound}(ij)$ as introduced in Section 2.2.4. Thus, the bigger $m$ gets, the more axonal connections between chips are discarded.

$$m = \sum_{\substack{i,j \\ i \neq j}} \max(0, \texttt{lower\_bound}(ij) - \texttt{realized}(ij)) \quad .$$

**Parameters** The impact of both temperature parameters is studied, which are considered to have the most general impact on the routing performance, as previously described in the text.

$T$ The temperature parameter controlling the acceptance probability for states with a lower energy.

$T_{\text{ext}}$ The temperature parameter controlling the shape of the lower soft bound. Smaller values of $T_{\text{ext}}$ enforce a stronger adherence of the bound.

**Results** The color maps in Figure 3.2 show the results of the performed experiment. Figure 3.2a, 3.2e and 3.2e illustrate the mean of the relative synapse loss as a function of $T$ and $T_{\text{ext}}$ after the routing optimization has been performed. The first row corresponds to the KTH system placed via a random mapping, the second row to the system placed via the NFC algorithm, but with *neuron type scaling*[1] on. The last column corresponds to the system placed by means of the NFC algorithm without neuron type scaling. Note that the color code for the relative synapse loss is different in all three figures. As expected, the general synapse loss is high for the default version of the KTH attractor model due to the bandwidth limitations and the limited number of synaptic connections. A complete realization of all synaptic connections would require 99221 specific synapses. The standard deviations for all sampling points have been determined and found to be below one percent, with a trend to higher standard deviations – still below one percent – for higher temperatures, as one would expect for the state propagation becoming more random. These low standard deviations justify the initial assumption that 500 iterations are enough to reach a stable state even for high-temperature runs.

Considering the randomly placed network, which has an initial synapse loss of 74% the optimization achieved a synapse gain of 12.8 to 8 percent. The synapse loss increases continuously for higher temperatures $T$, as one would expect. Similarly, the muting of inter-chip connections increases for larger values of $T_{\text{ext}}$. For values of $T$ above 1700 we can observe a phase transition: the muting vanishes completely even for large values of $T_{\text{ext}}$. In the limit of $T \to \infty$ we expect the system to achieve a synapse loss of about the size of the initial random initialization.

[1] A feature of the NFC algorithm to enforce the placing of inhibitory neurons and excitatory neurons onto different chips.

For the system placed by the N-Force-Cluster algorithm with neuron type scaling enabled the general appearance of the color map indicating the mean synapse loss looks similar. But obviously the synapse loss is significantly higher, many standard deviations apart from the results for the randomly placed system. This can be understood, since the neuron type scaling forces excitatory and inhibitory neurons, which have the highest mutual connection densities in the KTH model, on different chips. This might be beneficial for the wafer-scale system, where higher connection densities between chips are available, but is inept for the chip-based system. Still, the routing optimization is able to improve the overall synapse loss from initially 79.3 to 71 percent. The muting of inter-chip connections for the NFC system with neuron type scaling shows the same phase transition as for the randomly placed system. It appears at approximately the same temperature $T$, but the overall tendency to mute the chips is lower. As previously described in this configuration the NFC places inhibitory and excitatory cells on different chips. Since connection densities for inhibitory connections are higher than for excitatory connections, more synapses can be realized per axonal inter-chip connection. Consequently, their realization is energetically more favorable and the muting becomes less likely.

The last system, the KTH model placed by means of the NFC algorithm with the neuron scale disabled shows the significantly best results. From an initially 0.63% mean synapse loss the system is improved to 0.44%, which corresponds to a gain of almost 20 percent. Beyond that it performs 39 percent better than with the neuron scaling feature enabled. Even though the synapse loss for this system is inevitably still high, the result emphasizes the necessity of an optimized placement to achieve high numbers of synapses. The muting of this system looks a little different. The tendency to mute is weak, because now inhibitory neurons are distributed over the network, some onto each chip. Their high post-synaptic input count makes the realization of such connections energetically favorable and therefore protects the system from strong muting effects. Although the system approaches a similar phase transition to a state where muting become impossible but for significant higher temperatures beyond $T = 4000$.

**Conclusions Drawn from this Experiment Series** This experiment emphasizes the importance of a intelligent placing for a good routing performance. The analysis has been performed for the NFC algorithm with and without neuron type scaling. The synapse loss differs by almost 40% for between both system, which in the presented contest corresponds to an absolute difference of about $40,000$ synapses.

### Configuration State Convergence

The goal of this experiment is to evaluate how long it takes for the simulated annealing process to converge into a stable optimized configuration state. As mentioned in Section 2.2.4, different cooling strategies can be applied. Now, we want to see how these strategies may affect the quality of the final state and the time necessary to reach it.

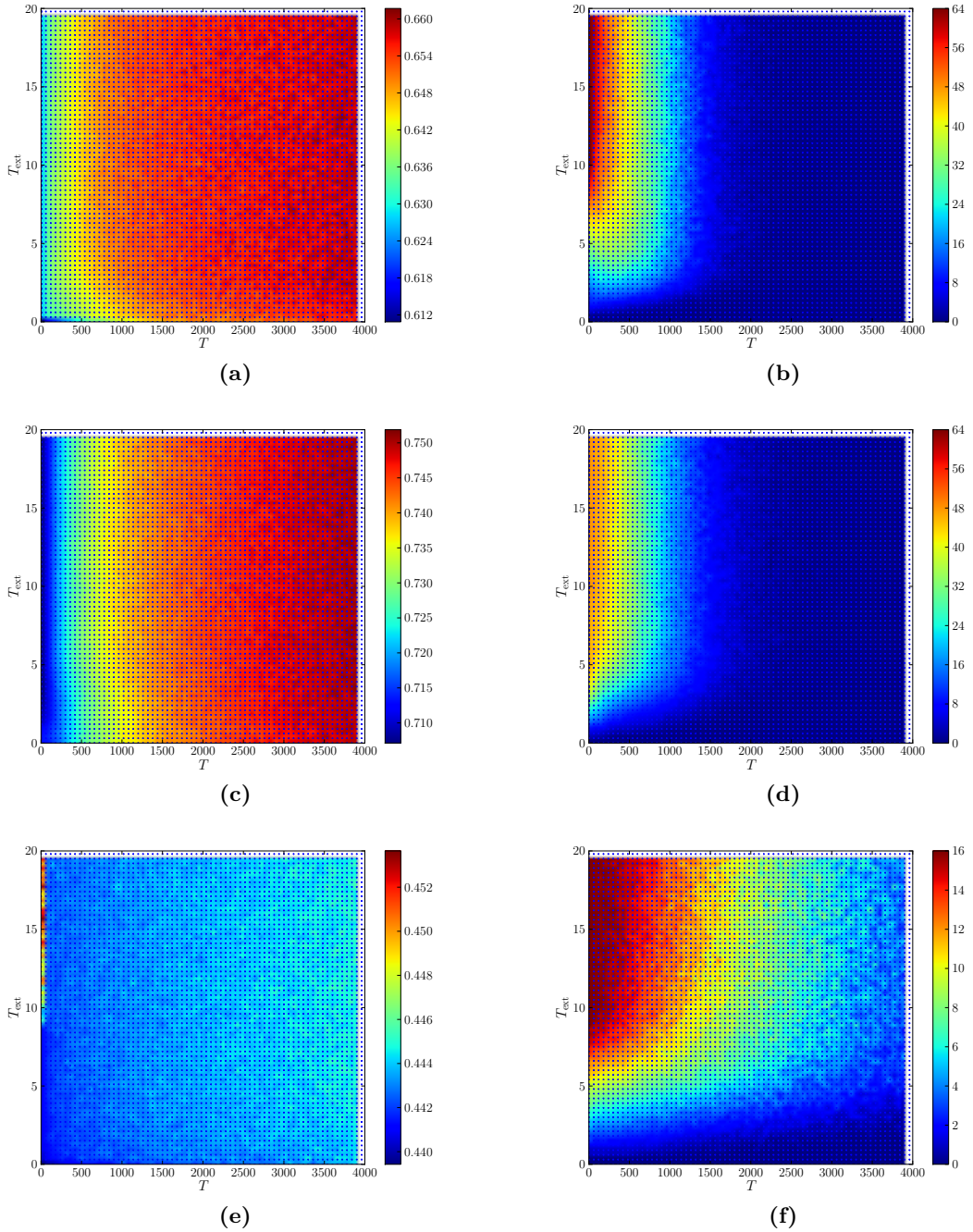


**Figure 3.2.:** The mean synapse loss (left column) and muting (definition see test) quantification (right column) for a 768 neuron KTH attractor model mapped to four chips with one core each. The first row corresponds to the system placed by random, the second row to the system placed by the NFC algorithm and *neuron type scaling* and the third row is the system placed by the NFC algorithm without *neuron type scaling*. Figures **(a)**, **(c)** and **(e)** show the actual synapse loss as a function of the temperature parameters. **(b)**, **(d)** and **(f)** illustrate the *muting* of inter-chip connections. A dedicated discussion of the results can be found in the text.

The following three basic cooling strategies are applied to the optimization process:

$$\text{constant temperature} \quad T(t) = \begin{cases} T(0) & \text{for } t < t_{\max} \\ 0 & \text{else} \end{cases} \tag{3.1}$$

$$\text{linear cooling} \quad T(t) = \begin{cases} T(0) \cdot \left(1 - \frac{t}{t_{\max}}\right) & \text{for } t < t_{\max} \\ 0 & \text{else} \end{cases} \tag{3.2}$$

$$\text{exponential coolling} \quad T(t) = \begin{cases} T(0) \cdot \exp\left(-\frac{t}{\tau}\right) & \text{for } t < t_{\max} \\ 0 & \text{else} \end{cases} \quad . \tag{3.3}$$

This results in the free parameters $T(0)$ for all strategies and $\tau$ for the exponential temperature decay. $t_{\max}$ denotes the number of algorithmic iterations performed for the routing optimization.

In accordance to the previous experiment the initial state is picked from within a range of $0.1\sigma$ away from the mean synapse loss for 100 random initializations. The gradient of the linear cooling strategy depends on the number of overall performed iterations $t_{\max}$, hence the annealing process cannot be measured iteratively. Instead the optimization process starts over again with an updated $t_{\max}$ value for each run, always starting from the same initialization. Each data point corresponds to 10 individual runs.

**Parameters** The state convergence is analyzed as a function of the parameters as follows:

$t_{\max}$ The number of iterations for a complete run of the routing algorithm.

$T(0)$ The initial temperature, which controls the configuration state acceptance for configurations with a higher energy.

$\tau$ The time constant for the exponential cooling strategy.

**Results** Figure 3.3 shows the results for the KTH models scaled to 384, 768 and 1536 neurons respectively. The networks are placed by means of the fully-automated random placement and the initial temperature value is $T(0) = 10$ by default, except of one sweep explicitly marked in Figure 3.3b.

Most obviously, the routing result is robust against parameter variation. Over a significant range, the results match each other within their errors. For the model representation with 768 neurons a wider range has been exemplarily sweeped (see Figure 3.3b). The result is stable until $\tau$ falls below 0.1. In the other limit for $\tau \to \infty$, corresponding to a constant temperature, the result remains stable in terms of achieving the same routing quality each time.

For initial temperature values below $T(0) = 10$, it is expected that the results remain about the same. Otherwise, the results for exponential cooling strategies with a large $\tau$ or the linear strategy for long iterations would further minimize the synapse loss. This behavior is in accordance with the results presented in the previous experiment, where long iterations have been performed over a wide range of temperature parameters. This does not hold for the opposite direction. Very large temperatures worsen the overall synapse loss and increase the error as expected, because the configuration state propagation becomes more and more random.

Most remarkably the number of iterations necessary to reach the stable configuration state increases slowly and is only weakly dependent on the size of the neural architecture. If one assumes that the convergence speed is almost independent of the network size, then the complexity of the optimization problem is proportional to the area of the reduced connection matrix. As described in Section 2.1.4, the respective growth in area is, by a factor of $\frac{n_{\text{neurons/chip}}}{n_{\text{cores/chip}}}$, slower than for standard connection matrices.

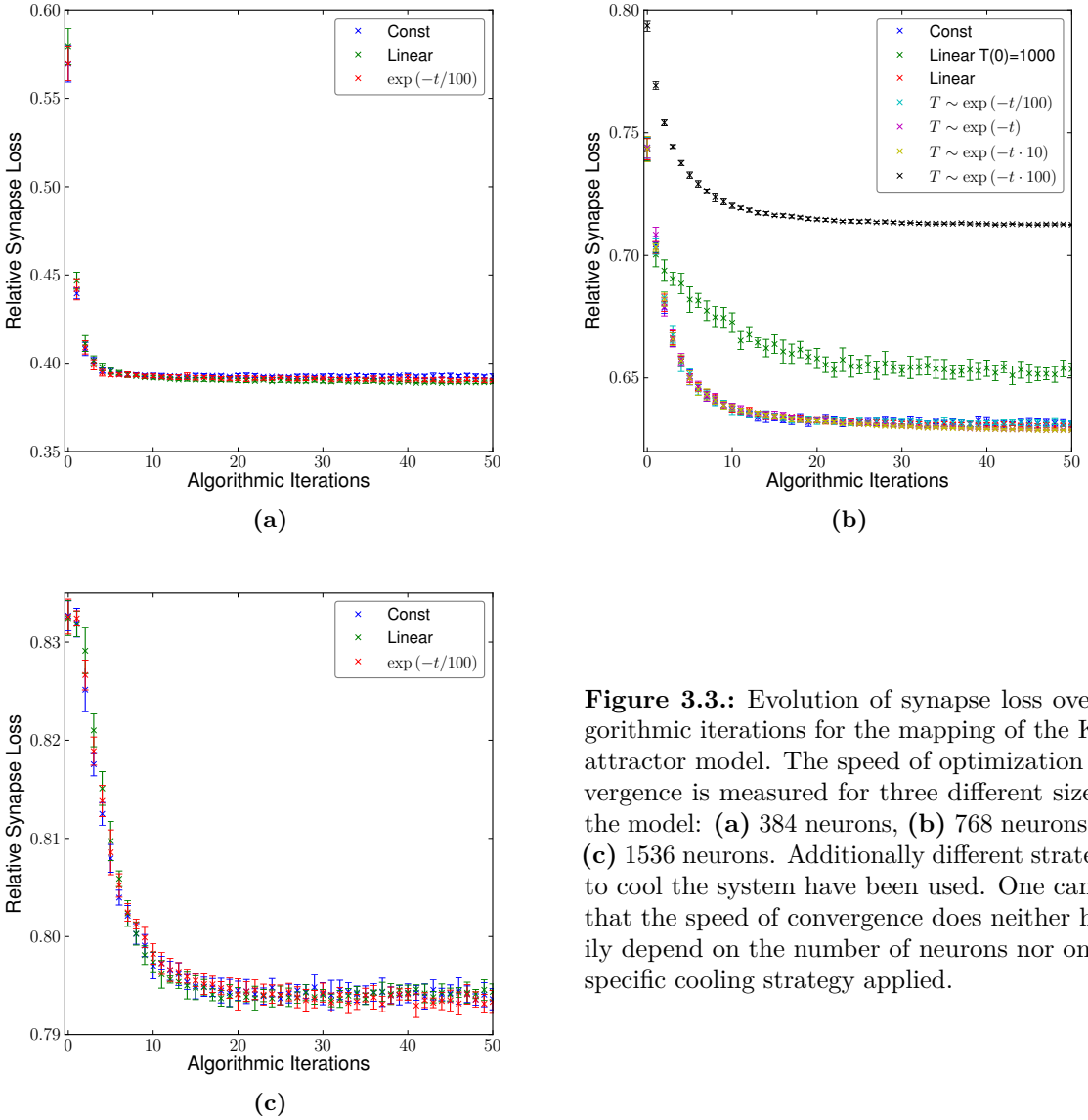


**Figure 3.3.:** Evolution of synapse loss over algorithmic iterations for the mapping of the KTH attractor model. The speed of optimization convergence is measured for three different sizes of the model: **(a)** 384 neurons, **(b)** 768 neurons and **(c)** 1536 neurons. Additionally different strategies to cool the system have been used. One can see that the speed of convergence does neither heavily depend on the number of neurons nor on the specific cooling strategy applied.

**Conclusions Drawn from this Experiment Series** The optimized result of the routing task turns out to be similar for the applied cooling strategies within a wide parameter range. However, the cooling process does make the process more robust against adverse choices of parameters. Therefore, the linear cooling strategy is to be used in the default implementation,

as it requires fewer parameters than the exponential cooling but performs equally in terms of synapse loss.

### 3.1.2. Scalability of the System

The main purpose of interconnecting Spikey chips is to scale up the chip-based system to larger neural network models. While the emulation time remains the same, the algorithmic task of mapping the network becomes more complex. Thus, the software becomes the bottleneck in operating a neuromorphic hardware system. The goal of the following analysis is to find these bottlenecks and show up chances to enhance runtime performance in future implementations.

Simple duration measurement are commonly hardware dependent. Parallel program execution and existing load on the computer can distort the results even more. To achieve more general results the Processor instructions used are counted instead. The counting is performed via *Callgrind*, a software compiler provided by the *Valgrind* software optimization framework [*Nethercote and Seward*, 2007]. In addition to that, the mapping software is compiled without any Processor specific optimizations. Many software compilers can optimize the runtime performance by making use of hardware dependent features. While desirable in normal operation, the optimization undermines the generality of our results.

Note that the results do not reflect performance gain by multiple CPUs. Most of the modules, which have been developed for this thesis, are designed to work in parallel. Thus, many of the measured instructions can be distributed on multiple processors. Another big player for the overall runtime performance is the time spent waiting for input/output operations. Unfortunately, this analysis can not account for such waiting cycles.

The software profiling slows down the mapping process tremendously, hence only a few sample measurements have been performed. Nevertheless the analysis can provide valuable information. For the same input and random seeds we can expect the software to operate deterministically and therefore require the same amount of instructions for each run. The only exception is the experiment control. Its runtime heavily depends on the amount of generated spikes during an experiment.

#### Parameters and Strategies

$N$ The number of neurons in the KTH attractor model

*placing strategy* The analysis is performed for both fully-automated placement algorithms: the random and the NFC placement

**Results** Figure 3.4 and Figure 3.5 illustrate the results of the performance measurement. Note that the figures share the same ranges for their coordinate systems.

The first thing to see is that the increase in complexity for the neuron placement is steeper than for the other components. Initially it is surprising that the runtime of the random placement increases that much. But investigations have shown that the main cause is not the placing itself, but the extraction of subtrees from the biological graph for the recursively operating algorithm. For one chip in the setup no subtree needs to be extracted, instead the algorithm operates directly on the biological graph. This can explain why random placing for the smallest model requires significantly fewer instructions compared to the runs for the larger models. We expect the computational effort for the creation of sub-trees to grow linearly with the number of chips.

For the NFC placement the complexity for the smallest system is already higher. Additionally, the increase in instructions is even steeper than for the random placement. The sampling points are insufficient to extrapolate a general rule for the complexity course. But it is expected that the computational effort grows quadratically with the network size since virtual forces are applied between any two neurons (see Section 1.3.1).

All other modules show an almost linear behavior within the sampling range. This is what one would expect for modules which operate on the configuration of chips which can mostly be treated independently. For example the parameter translation and the experiment control perform their respective tasks on each chip independently.

In Figure 3.5 we can observe an outlier for the experiment control which can be explained, as previously described, by the occurrence of disproportionate large amount of spikes. Potential causes for the increased amount of spikes are: an unfortunate mapping distortion for the system, normal variation for a non-optimized architecture or the hardware issues described in Section 3.1.3.

In a first-order approximation we can expect the system to require for all tasks except the placing, about $7 \cdot 10^{11}$ instructions for 16 chips with one core each. That are approximately the amount of instructions the NFC algorithm requires to place the four chip setup.

Additionally, the consumption of CPU cycles necessary to process the GMPath language invocations are measured (see Section 1.3.1). It has been found that its share of the overall instructions is significant for the overall performance. Note that the path language is used by the benchmarked modules, so that these instructions actually contribute twice. Once to the major modules and once to the path language invocations. This heavy contribution had already been found during an early software profiling. Consequently, path language usage has been minimized throughout the development.

**Conclusions Drawn from this Experiment Series** The results deduced from this experiment can be used to accelerate the mapping in future revisions of the mapping flow. It has been shown that one can expect the most significant performance boost by speeding up the placement. The required creation of subtrees causes the main overhead since it copies the structure each time. One ansatz could be to work on the biological graph directly to minimize copy operations.

Beyond the sheer number of instructions we expect a significant time overhead to be caused by input/output operations. The GraphModel as a non-memory continuous data container does not allow for efficient use of CPU caching features. A way out can be the template notes established throughout this work and previously described in Section 2.2.5. But so far the latter consideration remains a speculation that needs to be tested in further experiments.

### 3.1.3. Hardware Issues

Although the EVNET has been responsibly tested in *Friedmann* [2009], the setup suffers from at least two superposed hardware issues. Both need to be discussed to understand the following experiments.

The first issue concerns the occurrence of so-called ghost events. This phenomenon has already been described in *Brüderle* [2009]. These events appear within the digital spike recordings, although no activity can be observed on the neuron membrane. Commonly only small numbers of such events appear and the ones originating from unused neurons can easily be filtered out in software. Their cause is unknown at the time of thesis submission. The effect

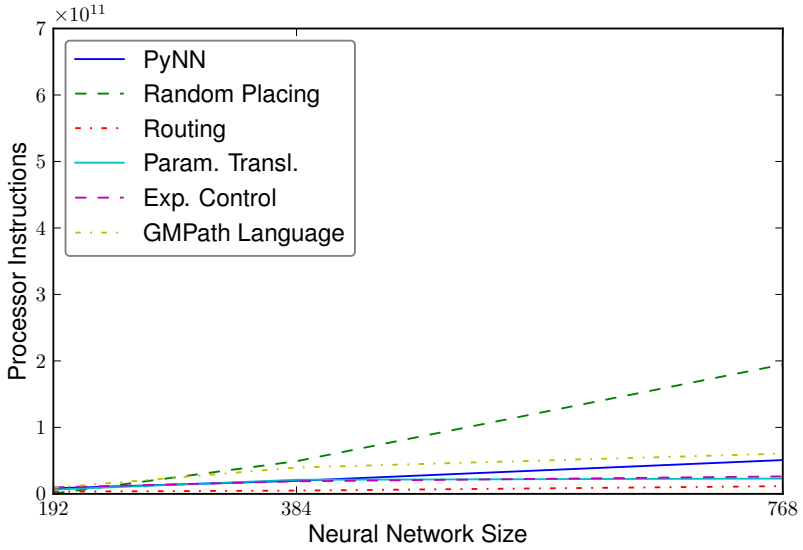


**Figure 3.4.:** The number of CPU instructions consumed for the mapping process broken down to individual steps as a function of the neural network size: One can see the complexity courses for the different mapping steps, including: placing, routing, parameter translation, and experiment control. Furthermore, the instructions caused exclusively by PyNN (see Section 1.2.1) and the GMPath Language (see Section 1.3.1) are listed. Note, that the path language is used by the other modules, therefore the corresponding instructions have counted twice. In this case the fully-automated random placement algorithm provided by the MappingTool has been employed.

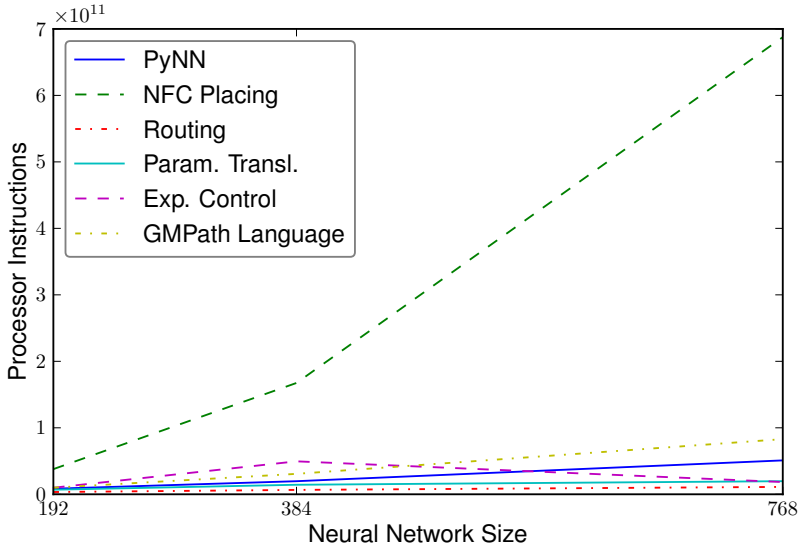


**Figure 3.5.:** The number of CPU instructions for the mapping process broken down to individual steps (for explanation see Figure 3.4) as a function of the neural network size: The placing of neurons has been performed by means of the NFC algorithm (see Section 1.3.1).

was considered to have an insignificant impact on experiments, hence no further investigations have been made on this issue. Unfortunately with the EVNET enabled FPGA configuration the effect is amplified. The intensity varies by several orders of magnitude between experiments and is characteristic for each Spikey-Nathan-combination. The configuration cycle of the chip seems to have an severe impact on that issue. The current workaround is to reconfigure each chip with the same configuration and subsequently measure the emitted spikes without input applied, until the event count stays below a certain limit.

The second phenomenon affects the reliability of event transmission. For the $2nd$ revision of the Spikey chip the EVNET has been proven to be perfectly reliable within bandwidth limitations. This reliability is reduced for the $3rd$ revision. Most likely the issue is related to a synchronization problem between the FPGA event sorting module and the chip clock (the sorting module is described in Section 1.1.4). It is assumed that the problems arise from altered chip timings in the latest chip revision. The reliability could already be improved by a revised FPGA design, but fluctuations remain. Similar to the first issue, the reliability issue is characteristic for each Spikey-Nathan-combination and shows non-deterministic properties.

Additionally, only one block per chip can be used to carry out the experiments due to an design error in the current revision. Thus, the neuron capacity is cut by half. The design error affects the scaling decisions, but does not interfere non-deterministically with actual experiments.

### 3.1.4. Verification of Accurate Hardware Configuration

The goal of the first multi-chip experiment is to check whether the system works as intended. Due to the size of the established software framework and the complexity of the general mapping task, the correct operation needs to be verified. Attention has been payed to the correctness of each individual module throughout the development process. E.g. high-level software tests for the routing process are available which check the integrity of the routing data before and after the operation. To demonstrate the working interplay between the components on a system-level a basic *Synfire-Chain* neural network experiment is utilized.

From now on all experiments are performed from the PyNN-layer. In case any specific modifications are applied to the underlying framework it is indicated in the text. The neuron placement was achieved by means of the deterministic population placer described in Section 2.2.3. The deterministic placement is necessary for the conscientious verification of the system to avoid distortions arising from a corrupt or unfortunate placement.

**Basic Setup** The hardware setup consists of four Nathan modules on one backplane. The placing of the Nathan modules is depicted in Figure 3.6. The general functionality of the MCGN lanes and the network functionality of the Nathans in question have been tested beforehand via low-level communication tests. Note that this offers no protection against the issues described in Section 3.1.3.

Furthermore, the event output buffer depth of the chips is reduced from 128 to 4 events as proposed by *Friedmann* [2009]. Consequently, strong activity can not fill up the transmission buffers, where the events would most likely expire their delivery time and consequently be dropped.

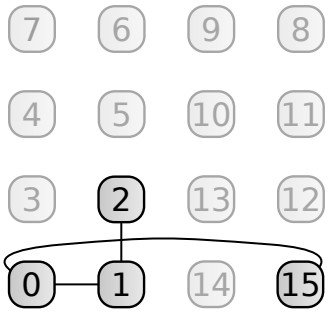


**Figure 3.6.:** The network configuration used for the multi-chip experiments. Four chips in a chain are available. All Spikeys belong to the 3rd revision and therefore provide one analog core, which results in a total count of 768 neurons in the setup.

| Parameter | Value |
|---|---|
| $N_{\text{exc}}/N_{\text{inh}}$ | 20/5 |
| $p_{\text{ee}}^{\text{remote}}/p_{\text{ei}}^{\text{remote}}/p_{\text{ie}}^{\text{local}}$ | 0.6/0.99/0.99 |
| $g_{\text{ee}}^{\text{remote}}/g_{\text{ei}}^{\text{remote}}/g_{\text{ie}}^{\text{local}}$ [$\mu$S] | 0.005/0.003/0.003 |

**Table 3.2.:** Network characteristics for the Synfire-Chain model with feed forward inhibition. Note that the model does not comprise local excitatory connections. A schematic of the model can be found in Figure 3.7

### Synfire-Chain Demonstration

The aim of the experiment is to verify the multi-chip system by issuing a neural architecture, that provides an easily interpretable response. The Synfire-Chain – part of the mapping benchmark library – can offer such a response and beyond that, carries out an actual task.

A Synfire-Chain is basically a feed-forward network with layers of neuron populations. Each population projects excitatory onto the next layer in a ring-like structure. Each population comprises both excitatory and inhibitory neurons. While the excitatory neurons feed their activity only onto neurons of the next population, the inhibitory neurons locally attenuate their excitatory neighbors. Figure 3.7 illustrates the ring-like structure of the Synfire-Chain. Activity in a thoughtful tuned network passes on from population to population and could go on forever in a closed loop. The purpose of the inhibitory neurons is to ideally silence the excitatory neurons after they fired once. Otherwise, high conductances could trigger avalanches of spikes, where the first population emits one spike, the second population already emits two spikes, and so on and so forth.

In our setup, each population resides on a dedicated chip projecting onto a population on the next chip in a closed loop. Thus, activity reaching the last chip in the chain is fed back to the first one. We therefore expect weaves of activity carrying on from chip to chip, each time traversing a chip boundary. The cycle duration of the activity is a direct measure for the delays between the chips in the network.

**Results** The rasterplot in Figure 3.8 clearly shows that activity passes on from chip to chip. The activity is stable for at least 4 s of biological real-time. Although the local inhibition is

not strong enough to limit the spikes to one per excitatory neuron, is is still strong enough to avoid a broadening of the activity waves. The mean cycle time of the activity wave is found to be $191.5 \pm 1.3\,\text{ms}$. The programmable hardware delay is set to $2047 + 127$ fast clock cycles for the global delay plus the local delay, which corresponds to $t_{\text{delay}} = 108.8\,\text{ms}$ in total for a speedup factor of $10^4$. From the cycle duration of the four chip Synfire-Chain a delay of $t'_{\text{delay}} = 47.9 \pm 0.3\,\text{ms}$ is deduced. The measured delay is significant smaller than the programmed delay. This can not be explained by transmission jitter, but rather indicates another problem with the event distribution network in combination with the $3rd$ revision Spikey. Additional measurements have already confirmed this suspicion. The cause is unknown at the time of thesis submissions, but might arise from the same cause as the other hardware issues described in Section 3.1.3. For a properly working system one would expect delays longer than the programmed values, due to the analog nature of the neurons. Incoming spikes contribute to the post-synaptic membrane potential, but it takes a certain amount of time until the threshold potential is finally reached.

Furthermore, multiple runs of the exact same experiment show that the Synfire-Chain non-deterministically dies out from time to time. This can happen even for strong activity in the preceding population. By using the deterministic population placer and keeping the network connections below hardware limits it is ensured that the configuration of the hardware is perfectly the same for each iteration. Thus, this behavior can not be caused by the software itself. Figure 3.9 illustrates such behavior. This is caused most likely by the unreliable spike transmission for the current revision of Spikey chips. The issue is described in Section 3.1.3.

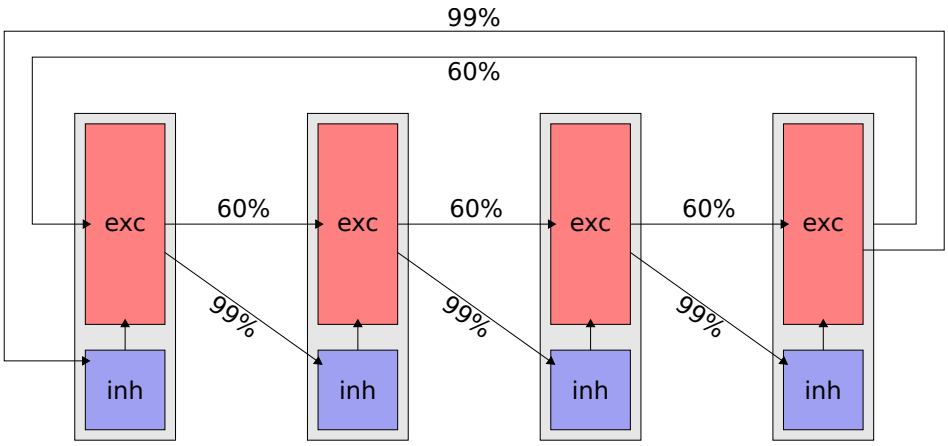


**Figure 3.7.:** The Synfire-Chain with feed-forward inhibition comprises populations in a ring-like structure. Each population – consisting of excitatory and inhibitory neurons – feeds its excitation to the next population. The local inhibition silences its excitatory neighbors after a short delay to avoid a broadening of the activity.

**Conclusions Drawn from this Experiment** The functional correct response of the synfire-chain proves that the established mapping framework is able to configure the system, operate the hardware and deliver the results back to PyNN appropriately. Despite all hardware issues, the foundation is laid to repeat the multi-chip Synfire-Chain experiment in a deterministic and neuroscientifically more relevant setup.

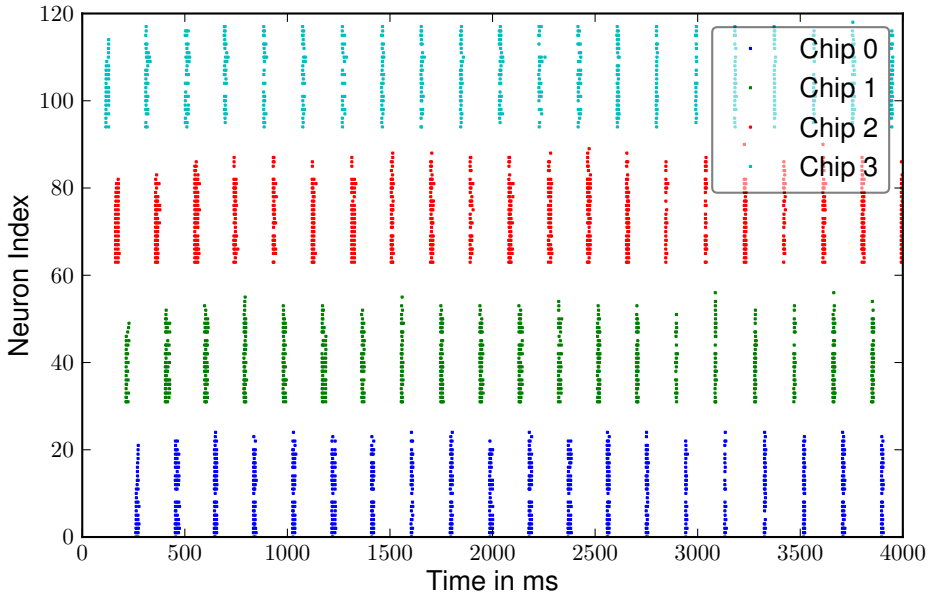


**Figure 3.8.:** Raster plot of an Synfire-Chain experiment on the multi-chip system. After an initial stimulation – in our case the top most population – activity passes on from population to population. Note that the loop is closed so that activity reaching the bottom of the figure continues on the initial population.

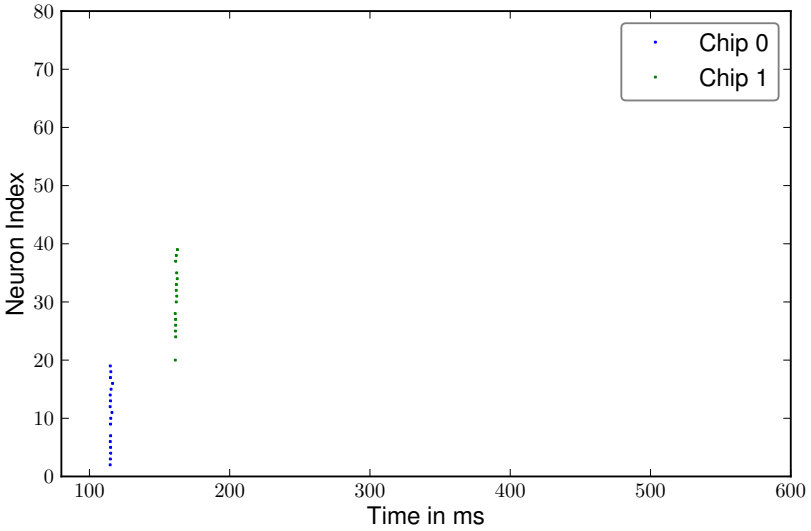


**Figure 3.9.:** The propagating activity of the Synfire-Chain unpredictably dies out due to event droppings caused by the hardware issues described in Section 3.1.3

## 3.2. Neural Network Experiments

This section presents neural network experiments performed on single and multi-chip setups. The purpose of the first experiment is to optimize the constrained tempotron with regard to a future hardware implementation. After that, the memory capacity of the self-stabilizing liquid architecture is analyzed. Finally, initial experiments for a multi-chip self-stabilizing liquid architecture are presented.

### 3.2.1. Single-Chip Liquid Computing

The self-stabilizing architecture as described in Section 2.3.2 has been realized on the FACETS chip-based system. The applied task for the performance evaluation of the LSM is explained in Section 2.3.1. The classification is performed by means of both the perceptron and the tempotron as outlined in Section 2.3.3.

**Basic Setup** The next two experiments which use the self-stabilizing architecture, are carried out on a single-chip setup. Consequently the issue concerning the ghost spikes, as described in Section 3.1.3, is attenuated compared to the multi-chip operation.

#### Constrained Tempotron

The main subject of this analysis is the optimization of a tempotron (see Section 1.4.3) with respect to a hardware realization and the resulting constraints. In addition to that, the presented study offers valuable information about the suitability of the self-stabilizing architecture as a substrate for LSMs. The spike response of such a substrate has been used to train and test the readout. The selective parameter studies described in the following aim at optimizing the classification results.

The tempotron requires significantly more parameters than the perceptron (see Section 1.4.3). Thus, the investigated parameters have been limited to a manageable subset. For example the proposed ratio of $\tau_{\text{membrane}}/\tau_{\text{syn}} = 4$ for the membrane and synaptic time constants has been kept [*Gütig and Sompolinsky*, 2006]. The complete set of parameters that were actually studied is listed below. Each of these has been varied individually in a dedicated experiment series. Throughout these runs, all remaining parameter values have been taken from a common set of default values: Each presented data point implies 1500 preceding training steps. The initial weight values are normally distributed around a mean of $\vec{\omega}^0 = 0.0005\,\mu\text{S}$ with a standard deviation of $0.0002\,\mu\text{S}$. Those studies that employ an exponentially decaying learning rate deploy a time constant of $\tau = 1000$ training iterations. The initial learning rate $\alpha(0)$ was set to 1 for constant learning rates and to 10 for exponentially and linearly decaying rates. The utilized LIF neuron parameter values are listed in Table 3.3.

**Parameters** The following parameters have been selectively studied to optimize the classification performance of the hardware-specifically constrained tempotron.

$E_l$ The resting potential of the neuron

$\vec{\omega}^0$ The mean of the normal distributed initial weight vector

$\alpha(0)$ The initial learning rate

| Parameter | Value |
|---|---|
| $V_{\text{reset}}$ | $-63\,\text{mV}$ |
| $V_{\text{thresh}}$ | $-55\,\text{mV}$ |
| $E^I_{\text{rev}}$ | $-80\,\text{mV}$ |
| $E_l$ | $-58\,\text{mV}$ |
| $g_l$ | $20\,\text{nS}$ |
| $\tau^E_{\text{syn}}/\tau^I_{\text{syn}}$ | $2.5\,\text{mS}$ |

**Table 3.3.:** The set of LIF neuron parameters which are used by default for the selective parameter studies. Note that the studies will show that $E_l$ is already close to optimal. This is no coincidence, the set has been cherry-picked based on early studies and provides a frame for the studies with generally good natured classification characteristics.

$\alpha(n)$ The learning rate decay strategy, which controls the impact of the weight updates over the training period

$\tau_{\text{learn}}$ The time constant for the exponential learning decay

**Results** In the beginning of this analysis description the studies will be presented, which are used to optimize the constrained tempotron. Subsequently a typical learning curve of a optimized software tempotron is presented, followed by a direct, i.e. not further modified mapping of this software-trained tempotron to the actual hardware system. For the selective parameter studies the labels *Frame-N* in each figure refer to the *Frame-N* tempotron classifying the N*th* spike segment in the past. For example the *Frame*-0 tempotron classifies the present spike segment. For a complete task description refer to Section 2.3.1.

The first study investigates the impact of the learning strategy and the choice of the initial learning rate $\alpha(0)$ on the classification performance. Figure 3.10, Figure 3.11 and Figure 3.12 show the classification results for the constant, for the linearly decaying and for the exponentially decaying learning rates, respectively. Each plot shows the classification performance as a function of the initial learning rate.

The figures consistently indicate that after 1500 learning steps the *Frame*-2-tempotron provides no significant correct classification, independent of the employed learning strategy and the initial learning rate. Thus, the *Frame*-2-tempotron has been omitted in the following for reasons of clarity. For the two easier tasks the tempotrons were trained for, namely the *Frame*-0 and the *Frame*-1 classification, the result for those tempotrons using the constant learning rate and the linear decaying learning rate look very alike. However, the tempotrons with the exponentially decaying learning rate show significantly better classification performances for large initial learning rates. The tempotrons with the latter learning strategies decay down to a learning performance close to chance level when the initial learning rate is increased to approximately 8, while the tempotrons with the exponential decay show still correct classification above 80% when starting at the same initial value. Consequently, the classification with a exponential strategy can be considered to be the most robust and is generally preferable.

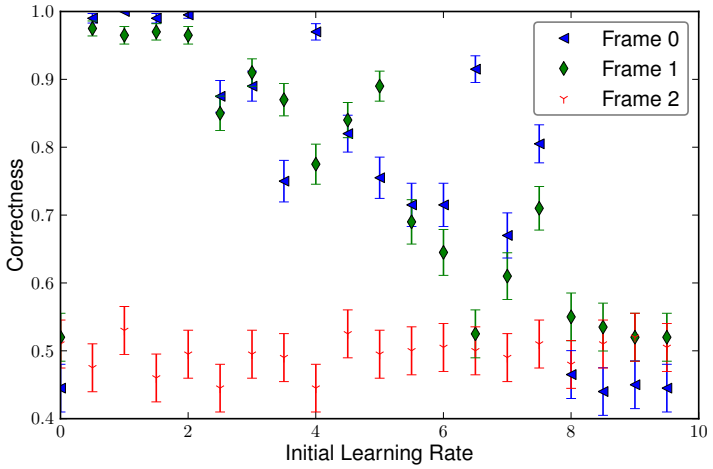


**Figure 3.10.:** The last 3 tempotrons carrying out the task described in Section 2.3.1. The label *Frame N* refers to the classification on the N*th* spike segment in the past. This parameter study focuses on the initial learning rate. Different decaying leaning strategies have been test. In this case the learning rate was constant over the complete 1500 training steps.

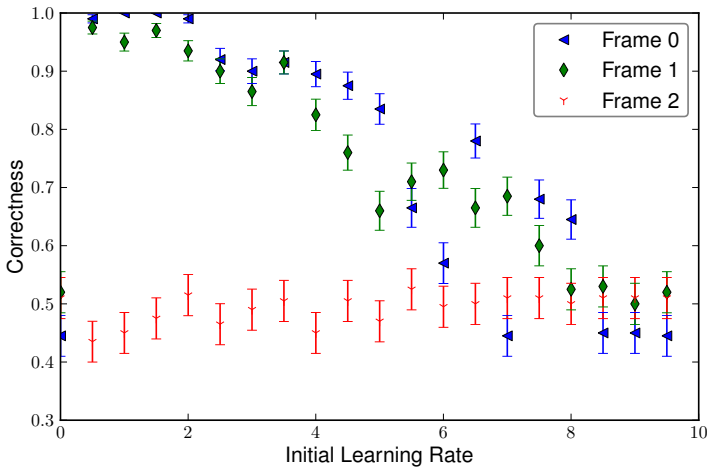


**Figure 3.11.:** Selective parameter study focusing on the initial learning rate in combination with a linearly decaying learning rate (for explanation see Figure 3.10).

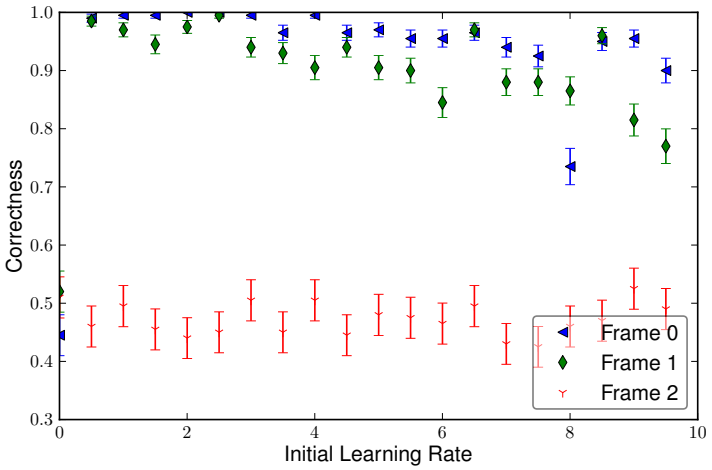


**Figure 3.12.:** Influence of the initial learning rate on the tempotron classification for an exponentially decaying learning rate. (for explanation see Figure 3.10)

The next parameter to be selectively studied is the mean initial weight. The standard deviation of the normally distributed weights is constantly kept at $\sigma = 0.0002\,\mu$S. Figure 3.13 illustrates the results for a large range of initial weights, even exceeding the available hardware range when applying the default chip calibration. Note that the learning rate influences the impact of initial weights on the final classification result. For large learning rates one may assume that the weights can reach basically arbitrary values within a few learning steps, while for small learning rates it can take arbitrarily long to reach a specific target weight. A constant learning rate of 1 has been used because for the chosen default time constant of $\tau = 1000$ and an initial learning rate of 10 the learning rate course remains above 1 over the complete learning period of 1500 steps. Hence, the impact of the initial weights on the final classification performance can be expected to be even less significant for the exponential learning strategy. Aside from that, the constant learning rate of 1 has been proven (see Figure 3.10) to yield good classification results. The results show that neither weak nor strong initial weight values in the available hardware range have a strong impact on the generally good classification performances for the *Frame*-0 and *Frame*-1-tempotron. Thus, the final classification performance is considered to be basically independent of the choice of initial weight values in the accessible hardware range.

Subsequently, the importance of the decay time constant $\tau_{\text{learn}}$ for the exponential strategy is investigated. The results are presented in Figure 3.14. It is important to note that a good choice of $\tau_{\text{learn}}$ for finite learning periods also depends on the maximum number of training steps $t_{\max}$. The reason for this is that both $\tau_{\text{learn}}$ and $t_{\max}$ determine how far $\alpha(t_{\max})$ approaches 0 during the learning phase. For short training periods and long time constants the learning rate is almost constant, while in any case for short time constants the effective learning can stop too early as $\alpha$ decreases rapidly, so that later weight updates in the training have no significant impact on the weights anymore. In the results we can observe that for time constants above $\approx 800$ the classification performance starts to decrease for both the *Frame*-1 and the *Frame*-0 tempotron. This is within the range of what we expect, as for $\tau_{\text{learn}} = 800$ the learning rate drops continuously to $\alpha(1500) = 1.35$, but is above this value most of the time,

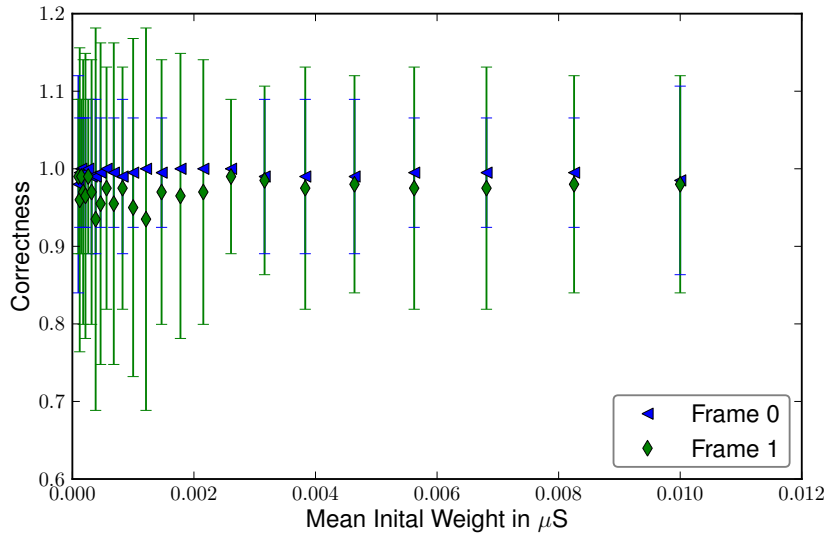


**Figure 3.13.:** Selective parameter study of the impact of the initial weights on the classification performance after 1500 training steps. A constant learning rate has been used for the training.

while the parameter study for the initial learning rate for the constant rate (see Figure 3.10) indicated a performance decrease for 1500 learning steps starting from $\alpha \approx 2$ upwards.

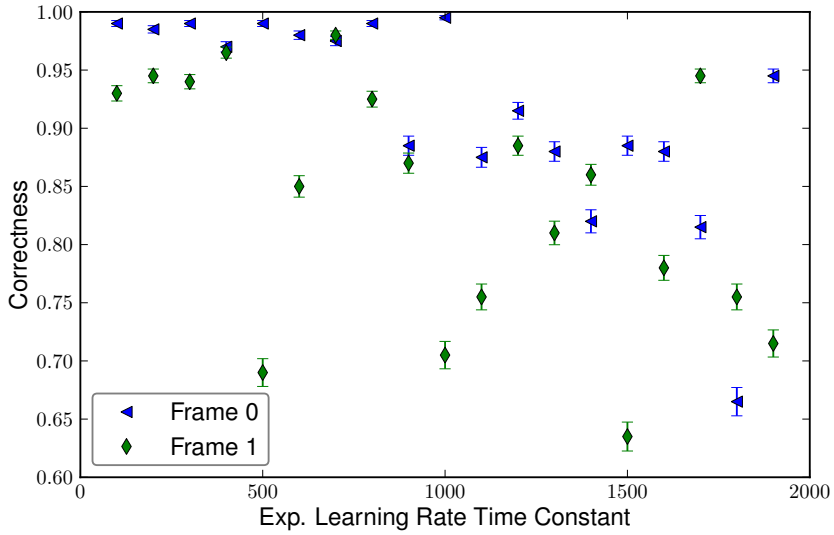


**Figure 3.14.:** Tempotron classification performance for an exponentially decaying learning rate with the time constant $\tau_{\text{learn}}$ after 1500 training iterations. Note that values for $\tau_{\text{learn}}$ which yield a good classification always rely on a sufficient number of training steps, since the smallest applied learning rate is given by $\exp\left(-t_{\max}/\tau_{learn}\right)$.

For a fixed set of $V_{\text{thresh}}$, $V_{\text{reset}}$ and limited weights, the resting potential $E_l$ can have a severe

impact on the ability of the tempotron to emit a spike or not fire at all. A good value for $E_l$ depends on the mean input rate. It is assumed that calibrating the resting potential such that – for the untrained tempotron and the trainings input applied – the chance for firing equals the chance for not firing is a good point to start from. In such a scenario both patterns (+) and (-) have approximately the same contribution to erroneous classifications. Figure 3.15 shows the classification correctness over $E_l$ from a corresponding study with an exponentially decaying learning rate applied. The observed classification correctness is relatively low, but within the range of what can be expected when considering Figure 3.14. In this plot the classification performance for $\tau_{\text{learn}} = 1000$ is already in the strongly varying region. Nevertheless, the study indicates good classification results for $E_l$ values in the range of $-64\,\text{mV}$ to $-62\,\text{mV}$. Considering the symmetry in the course of $E_l$, shifting $E_l$ can in a first order approximation be understood as a constant offset towards the (+) pattern or the (-) pattern respectively.

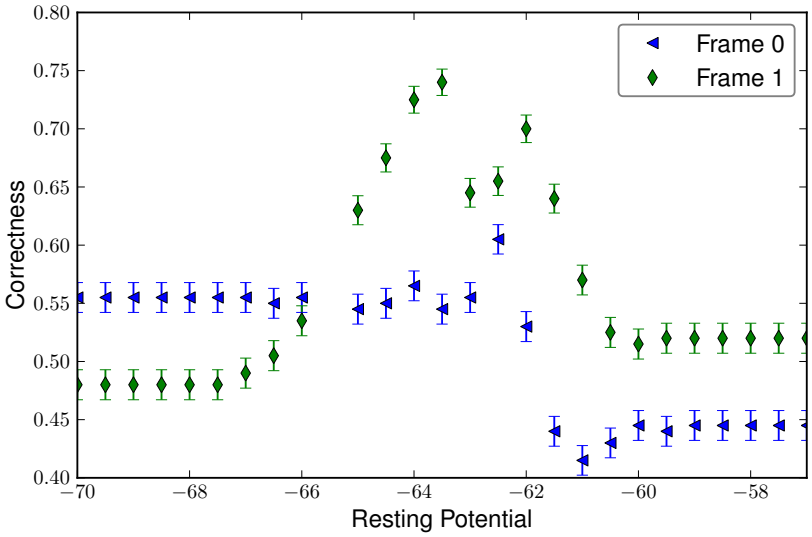


**Figure 3.15.:** Selective parameter study illustrating the impact of the resting potential on the classification performance. For each data point the *Frame*-0 and *Frame*-1-tempotrons have been trained with an individual resting potential over 1500 learning steps with an exponentially decaying learning rate.

**Decisions based on the Preparative Studies** We can now extract a suitable set of parameters from our selective studies. This set is subsequently used to investigate how the classification correctness evolves over the learning period. Figure 3.16 shows the learning curve of the constrained tempotron as simulated in software. One can see that the classification correctness of the *Frame*-0 and *Frame*-1 tempotron quickly approach a close-to-optimal classification result, while the performance of the *Frame*-2 tempotron constantly remains on chance level. The last data point for each tempotron shows the classification results after the weights have been clipped to the 4 bit hardware-like weight values. However, the classification performance remains high, so that no significant difference can be observed. Furthermore, a histogram illustrating the distribution of weights after the training and subsequent clipping of weights for the *Frame*-0 tempotron can be found in Figure 3.17.

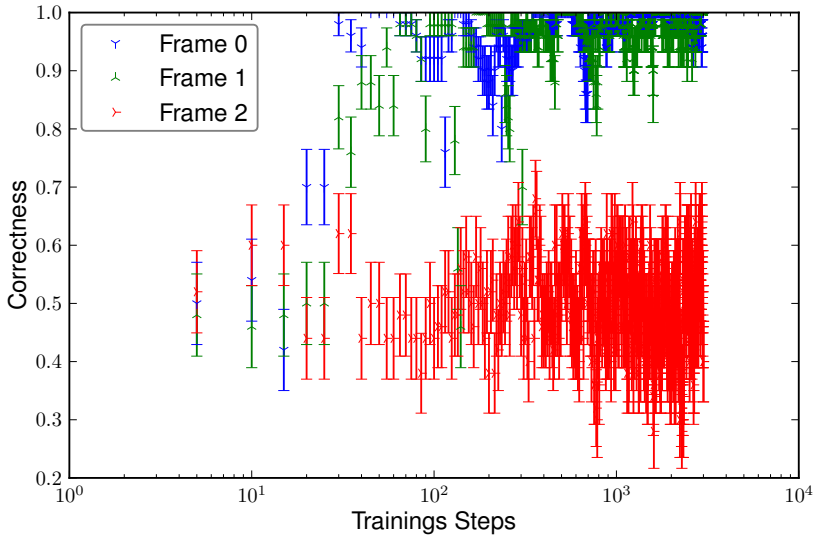


**Figure 3.16.:** Development of the classification performance of constrained tempotrons in software over a training period of 3000 leaning iterations. Note that the training steps are drawn on a logarithmic scale to provide a sufficiently resolved depiction of the early learning phase. Each tempotron is trained to classify one of the last three 50 ms spike sequences, i.e. the *Frame*-0, the *Frame*-1 and the *Frame*-2 tempotrons are plotted. The learning rate rate decays exponentially with a time constant of $\tau = 1000$. As a result, the performance fluctuations are attenuated for the *Frame*-0 and *Frame*-1 tempotrons during late training iterations. The task is described in more detail in Section 2.3.1.

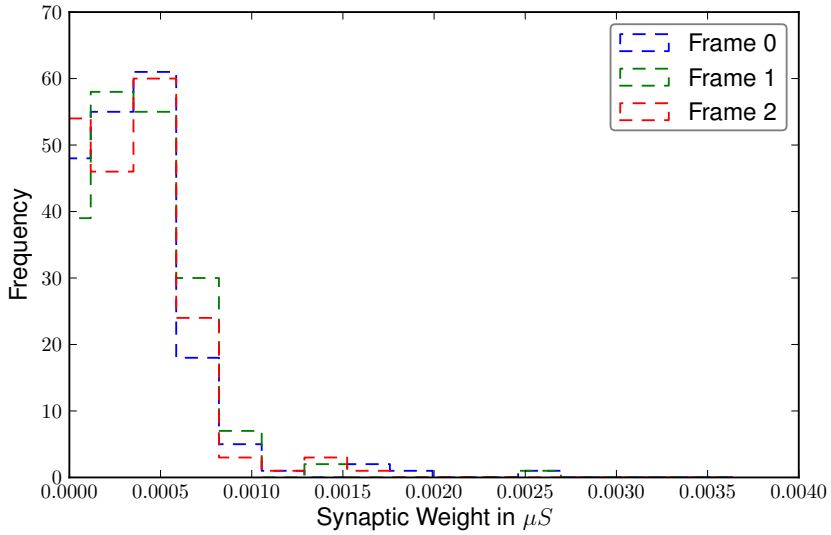


**Figure 3.17.:** Synaptic weight distribution of the hardware-specifically constrained tempotrons after 3000 training steps and the subsequent clipping of continuous to discrete weights. The corresponding on-chip classification performances can be found in Figure 3.18, Figure 3.20 and Figure 3.21. The respective evolution of the in-software classification performance over the training period is illustrated in Figure 3.16.

A direct, i.e. unmodified mapping of a software-trained *Frame*-0 tempotron to the actual chip-based hardware system is shown in Figure 3.18. The weights have been trained in pure software simulations and subsequently been clipped to the discrete hardware weights. The experiment does not yet account for any hardware-specific compensation methods, except of an experimental stretching of weight values $\omega' = a \cdot \omega$. The mapping has been carried out for one neuron on four different chips to test the robustness of classification performance against hardware variations. Obviously, the on-hardware classification is close to optimal for a weight stretching of $a = 1.3$. The characteristic performance course over $a$ is consistent for all utilized neurons and can also be observed for the software tempotron with clipped weights. Most likely the stretching compensates a distortion introduced by the clipping of the continuous weights to discrete hardware values. Otherwise, one would expect the classification to be optimal for $a = 1$ after the learning procedure.

The good quantitative matching between hardware and software emphasizes the potential of the PyNN approach: A readout trained completely in software can be mapped to the FACETS chip-based hardware system without obstacles.

In order to assert the validity of this statement, Figure 3.19 shows the classification performance for a tempotron configured with the same weight values as for the runs shown in Figure 3.18, but now being randomly distributed among the synapses. The results – classification performance is always around chance level – clearly proof the differential and individual training of each synapse, because as obviously applying the same weight *distribution* is not sufficient to reach a significant classification performance.

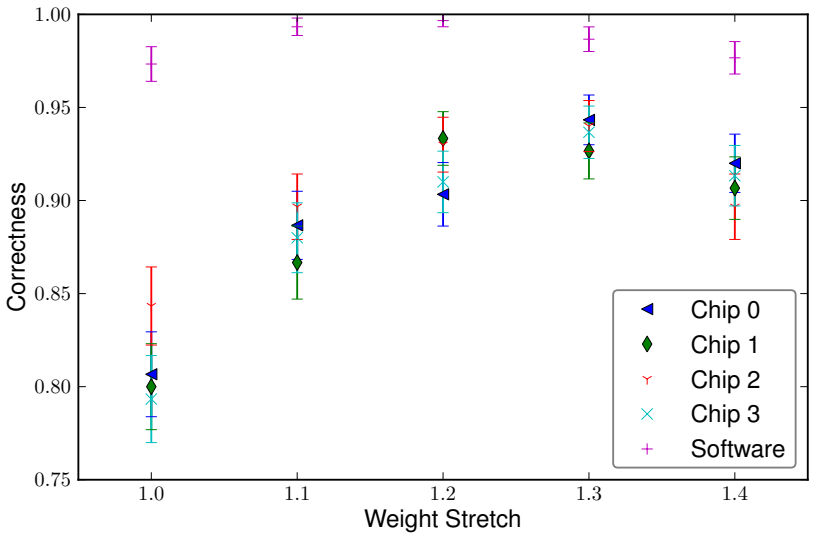


**Figure 3.18.:** A direct mapping of a software-optimized *Frame*-0 tempotron – i.e. classifying the latest spike segment only – onto actual hardware neurons. The illustration shows the readout performance of four neurons located on different Spikey chips. After the weight discretization process, the weight values have additionally been stretched by a factor close to 1, which is depicted on the x-axis. The distortions caused by the discretization are best compensated with a weight stretch of $\approx 1.3$.

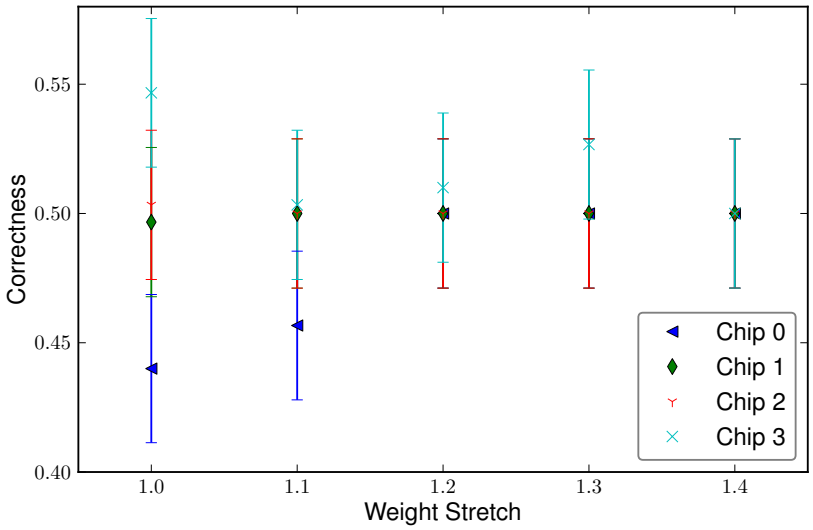


**Figure 3.19.:** Classification performance for the *Frame*-0 tempotron with the same weights as the tempotron presented in Figure 3.18, but randomly shuffled. Obviously the classification remains on chance-level. This illustrates the importance of an individual learning for each weight.

Figure 3.20 and Figure 3.21 show the results for the *Frame*-1 and *Frame*-2 tempotron on hardware. For the 1*st* frame the general course of classification correctness versus weight stretching matches the course in software. Although the hardware-software matching is not within the error ranges, one can clearly see how all hardware classification results and the software result decrease for stronger weight stretching. The classification correctness of the *Frame*-2 tempotron remains on chance level, in accordance with the software prototype.

**Conclusions Drawn from this Experiment Series** The parameters of the tempotron LIF neuron and the parameters responsible for the training have been optimized to solve the liquid computing classification task defined in Section 2.3.1 in a purely spike-based manner and with the goal to port the full setup to hardware. A typical learning curve has been measured, which allows to estimate the minimum effort to be invested for a successful tempotron training.

For the first time a single-cell spike-based in-hardware classifier has been realized on the highly accelerated FACETS system. The mapping of the software-trained tempotrons onto the hardware substrate yield good classification performances close to the respective software prototype. As one important possible application, host-communication bottlenecks can be overcome by performing classification tasks directly on the hardware system and reading back only the processed answer. The fact that a software-trained tempotron can be mapped seamlessly to the FACETS hardware system underlines the potential of the PyNN approach and the importance of a quantitative simulator-hardware matching as proposed e.g. in *Brüderle et al.* [2009]. The results suggest the realization of a complete LSM on hardware including both the liquid and the classifier.

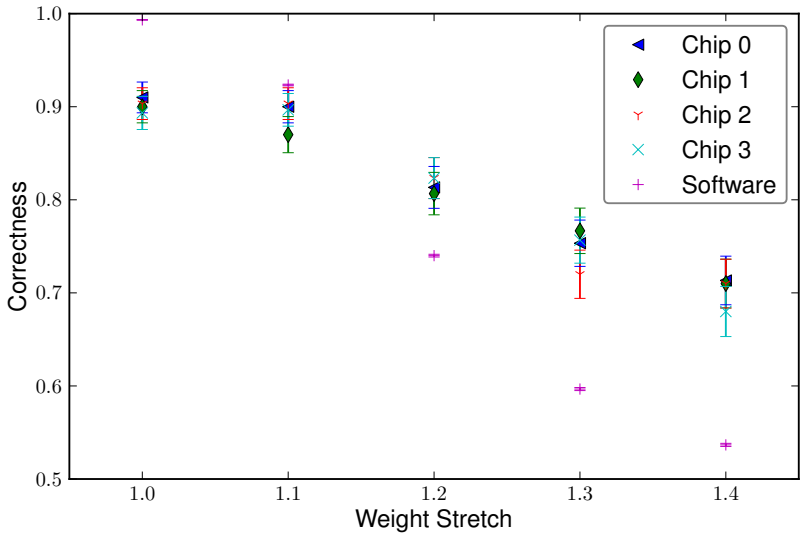


**Figure 3.20.:** A direct mapping of a software-optimized *Frame*-1 tempotron – i.e. classifying one spike segment before the latest one – onto actual hardware neurons. The illustration shows the readout performance for four neurons located on different chips. After the weight discretization process, the weights have additionally been stretched by a factor close to 1, which is depicted on the x-axis. The distortions caused by the discretization can in this case not be fully compensated by the weight stretch.

### Memory Capacity

To estimate the memory capacity inherent to the liquid, the classification performance with liquid is compared to the classification without liquid between input stream and classifier. However, both the perceptron and the tempotron contribute their own memory. For the perceptron the memory is caused by the spike convolution necessary to prepare the input and for the tempotron the memory is inherent to the underlying LIF neuron with its time constant $\tau_{\text{membrane}} = C_{\text{membrane}}/g_{\text{l}}$ (see Section 1.1). Therefore, one needs to isolate the memory provided by the liquid from the memory provided by the readout. In the case of the perceptron the classification can be evaluated for different convolution time constants to get an estimate of the memory provided by the liquid and the one provided by the perceptron. The memory analysis is performed only by means of the perceptron due to the higher computational complexity of the tempotron.

The first spike segment in the past has been used to carry out task, since the previous experiment showed no significant classification success by the tempotron on the second segment in the past.

**Parameters** Subject of the memory capacity analysis is the convolution time constant. A sweep is performed for both a perceptron classifying the response of the liquid and a perceptron classifying the input without a liquid-based pre-processing.

- $\tau$ time constant for the convolution of the spike train with an exponentially decaying function.

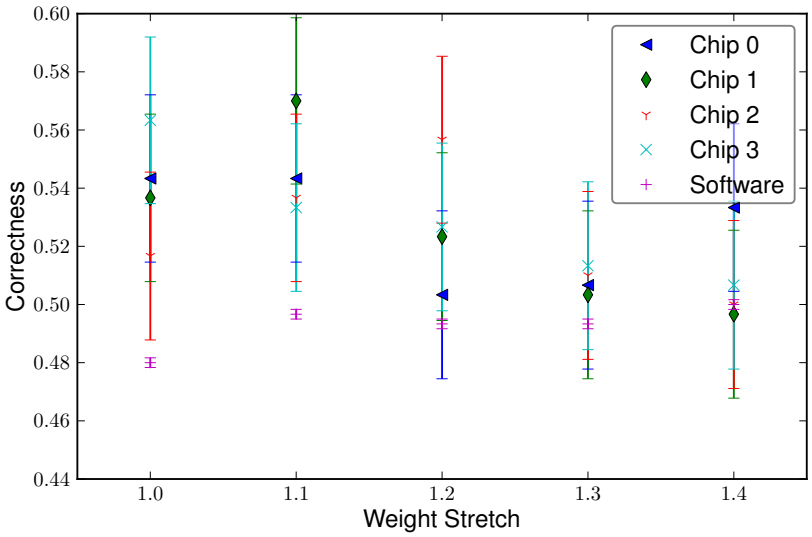


**Figure 3.21.:** A direct mapping of a software-optimized *Frame*-2 tempotron onto actual hardware neurons. The illustration shows the readout performance for four neurons located on different chips. After the weight discretization process, the weights have additionally been stretched by a factor close to 1, which is depicted on the x-axis. Independent of the stretch factor, the classification result remains on chance level.

**Results** Figure 3.22 shows the gathered results. Most obviously one can see that the classification performance for the Perceptron strongly depends on the free parameter $\tau$. For $\tau \to 0$ the classification performance reaches chance level, in accordance to what we expect. For small values of $\tau$ none of the spikes has a significant impact on the actual input presented to the perceptron anymore.

The classification on the liquid outruns the one *without* liquid for small values of $\tau$ and consequently for a small internal memory of the perceptron. For $\tau \sim 12\,\text{ms}$ to $\tau \sim 15\,\text{ms}$ the classification *with* liquid reaches a level below the final plateau where the classification is already above chance level. In this range the classification is strongly dependent on the specific structure of the applied spike sequences. Consider simplified the cast that the perceptron distinguishes the two patterns by the occurrence of their last spikes only. The distance in time between those two spikes be $\Delta t$, then their contribution to the input of the perceptron differs by $exp\left(-\frac{\Delta t}{\tau}\right) - 1$. This is an exponential dependency of the classification correctness on $\Delta t$ for this feature. During the task generation we added a normal distributed jitter with $\sigma = 2\,\text{ms}$ on each spike time. This could explain the 3 ms range of premature classification correctness. This suspicion is supported by the fact, that the same analysis on the column-based liquid shows the same behavior (see Figure 3.23).

The classification performance without any liquid exceeds the one with liquid for large values of $\tau$. The memory contribution of the liquid becomes unimportant for such $\tau$. Without the distortions from the neural architecture and without hardware variations and noise the classification on the unfiltered, liquid-less input is more efficient.

Finally, we want to see how the memory of self-stabilized liquid compares against the column-based liquid analysed by *Albert* [2010]. Figure 3.23 shows the corresponding result for

the same analysis. We can see, that the general appearance of the curves is very similar to the one illustrated in Figure 3.22. However, the classification correctness on the column-based liquid is about 10% better than on the self-stabilizing architecture over the complete range. This is not surprising, since the column-based liquid has been specifically tuned for this task, due to its narrow working point. The self-stabilizing network, in contrast, did not require a time intensive tuning. Although the liquid offers a broader working range, the separation property (see Section 1.4) of the liquid is limited by the self-regulating behavior.

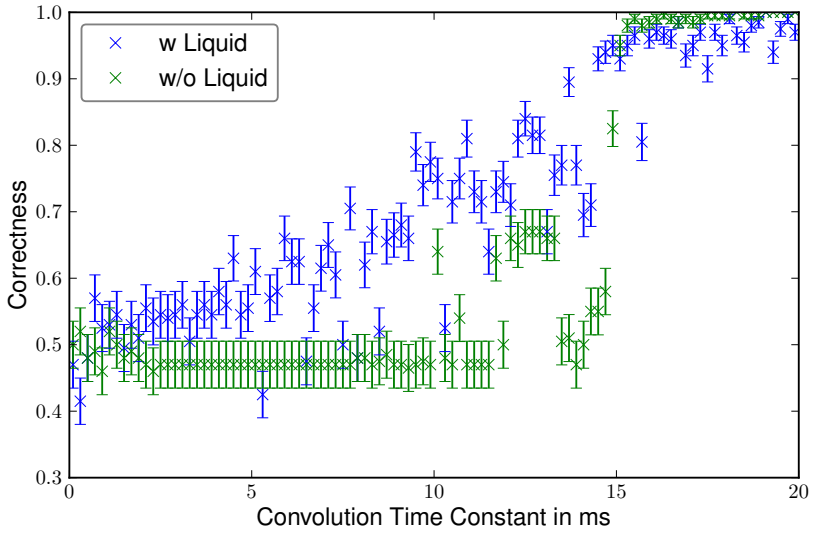


**Figure 3.22.:** Correctness of the perceptron classifying the first spike segment in the past (from $-100\,\text{ms}$ to $-50\,\text{ms}$) in dependence of the convolution time constant $\tau$. For each data point 1000 training steps have been performed and the correctness is measured over 200 test stimuli. The error is given by the standard error of the mean. One can see that the classification beyond chance level sets in significantly earlier for the classification with liquid compared to the classification without liquid.

**Conclusions Drawn from this Experiment** It has been shown, that the self-stabilizing network architecture mapped to the Spikey chip can be utilized as a substrate for LSMs. It enabled the perceptron to classify beyond its own memory capacity. Its memory capacity has been found to be lower than for the column-based architecture [*Albert*, 2010] for the given task and input intensity. However, due to the robustness of the self-stabilizing architecture, it can be used in a more versatile. While the column-based architecture needs to be tuned to a specific task, the self-stabilizing architecture can easily be applied to a multitude of tasks.

### 3.2.2. Feasibility Analysis: A Liquid State Machine on the Multi-Chip-Setup

In Section 3.2.1 the results of the liquid based on the self-stabilizing architecture realized in a single-chip system have been presented. The same architecture has been extended to multiple chips employing the multi-chip mapping framework introduced in this thesis. The architecture-intrinsic ability to amplify weak activity should enable the network to reach its working point already for very few inter-chip connections.

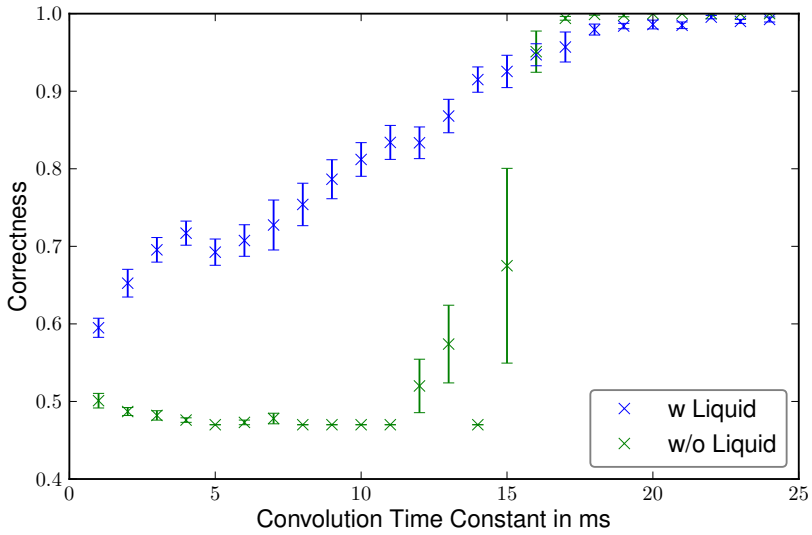


**Figure 3.23.:** Memory capacity analysis performed by M. Albert. The readout performance of a perceptron is measured against its convolution time constant. In accordance to the memory capacity analysis for the self-stabilizing architecture, the first spike segment in the past has been classified. The perceptron has been trained wit 60 samples in 5 iterations. Each data point corresponds to 40 classifications. Data by *Albert* [2010].

The necessary network descriptions have been implemented and a bandwidth analysis has been performed, indicating sufficient throughput via the network. However, a final analysis of the multi-chip liquid shows that the dynamics driven by the hardware issues (see Section 3.1.3) have an impact that is too strong to achieve classification performances beyond chance level.

**Basic Setup** For the feasibility analysis the Nathan setup described in Section 3.1.4 and depicted in Figure 3.6 has been used. The utilized neural architecture is depicted in Figure 3.24. Each chip comprises an individual copy of the single-chip variant of the self-stabilizing network as introduced in Section 2.3.2. Then a number of excitatory remote connections are drawn from one single chip to all other chips. The goal was to initially occupy as little MCGN network resources as possible in order to attenuate distortions induced by event drops as efficiently as possible. The bandwidth analysis below benchmarks the event drops for an increasing number of remote connections from the sending to the receiving Spikey chips.

### Bandwidth Analysis

Initially a single chip, the sending one, is prepared with the self-stabilizing architecture, while the other chips are set to loopback mode[2] to simply receive the input events and output a copy of each so they can be recorded for analysis. Hence, all events sent by the sending chip and transmitted via the network should in principle appear on the receiving side, except for

[2]In loopback mode digital events are forwarded from the input buffers to the output and bypass the analog part of the Spikey Chip, i.e. every received event will appear as a recorded spike in the output buffer of the chip. See [*Grübl*, 2007, Section 4.3.6] for more details.

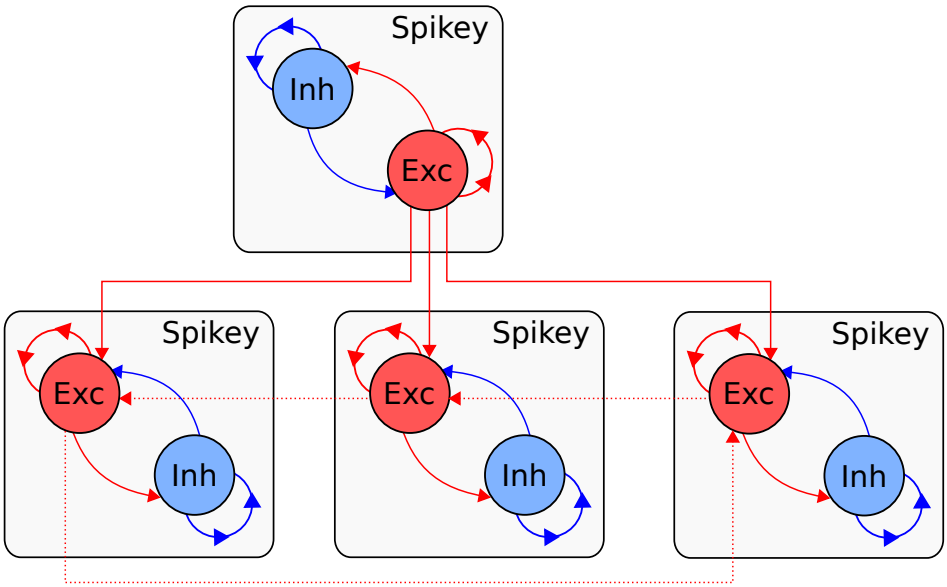


**Figure 3.24.:** A simple proposal for a multi-chip liquid implementation, deduced from the single-chip self-stabilizing architecture described in Section 2.3.2 and depicted in Figure 2.9. The dashed excitatory connections are optional. The goal was do extend the dynamics of a single liquid onto multiple liquids, while using as little MCGN network resources as possible. In case the connections prove to be sufficient one could easily think of extending the proposed architecture to a more versatile one.

events dropped in the network. The difference between the sent and the recorded events on the receiving side should equal the number of dropped events. In *Friedmann* [2009] a special event drop counter had been implemented in the FPGA designs to perform corresponding experiments. This counter was removed later for the final design. These designs diverged from one another by multiple revisions, thereby making a reapplication of this original drop counting feature impossible. The spikes on the sender side need to be filtered for their respective target Nathan, while on the sending side, due to the lack of input stimuli (except of the final end-of-experiment event, which has been described in Section 1.2.2) and the bypassing of the analog part, all occurring spikes are considered to be generated by network neurons. Nevertheless, all events which originate from non-allocated synapse drivers on the target chips are additionally filtered out to attenuate the erroneous counting of ghost events.

In this setup the appearance of ghost events, caused by the hardware issues described in Section 3.1.3, is a critical factor undermining the precision of the result. The measurement is also affected by the fact, that the end-of-experiment event is recorded non-deterministically to the output spike train. However, this distortion is rather small compared to the one induced by the ghost events.

**Parameters** In this bandwidth analysis the network events are driven by the dynamics of the liquid on the sender chip. We consider the parameters of the self-stabilizing architecture as fixed. Hence, the mean network event rate is controlled by the mean liquid activity and the number of remote network connections.

$N_{ext}$ The number of inter-chip connections from the sending Nathan to each receiving Nathan

**Results** Figure 3.25 shows the gathered results from this experiment, plotting the number of measured event drops for different values of $N_{\text{ext}}$ and for different target chips. In addition to that, a histogram illustrating the frequency of measured event drops for different numbers of remote connections can be found in Appendix C.1. The histogram proves that the measured event drops follow a continuous curve and are not randomly distributed due to the hardware issues.

Most obviously, chip 0 and chip 1 respond very similar, while chip 2 is more prone to the ghost event phenomenon. It is remarkable that the number of dropped events is negative for a wide range of remote connections, which means that on the receiving side more events have occurred than events have been sent. Between any two runs any possibly remaining neural network dynamics in the network that might have been kept alive by self-excitation was silenced by setting all hardware weights to 0 for a sufficient period of time. Consequently, remaining activity from previous experiments can be ruled out as a reason for the observed dynamics on the chips. Furthermore, in loopback mode, spikes occurring in the analog parts of the chips are ignored. The mean spike frequency of the self-stabilizing architecture on the sender side has been found to be $\bar{\nu} = 12.2 \pm 9.5\,\text{Hz}$. If the large number of ghost events, especially arriving on chip 2, would be caused by activity from the sender chip one could expect stronger variations in the measured event drops. However, the continuous course of the results and the trend to higher drop rates for an increasing number of remote connections suggest that we nevertheless observe real event drops. Consequently, for the following it is assumed that the ghost events can be considered as an almost constant offset originating from the receiving side.

For an increasing number of remote connections more synapse drivers on the receiving side are allocated. Thus, less ghost events can be filtered out since the events might be actual network input. The number of dropped events decreases in the beginning, because the additional allocated synapse drivers contribute more ghost events than actual network events. This effect appears to be stronger for chip 0 and 1 than for chip 2. A reason for this effect could be that the ghost event rates are characteristic for each Nathan/Spikey combination and each synapse driver. If so, the similar behavior for both chip 0 and chip 1 is suspicious. This could indicate that there is a systematic problem causing the ghost events. However, this remains speculation, since the problem is not understood at the time of thesis submission. Starting from approximately 20 remote connections a positive slope can be observed in the number of event drops. One can speculate that from that point the actual event loss surpasses the gain in ghost events. The linear increase in the event drops is in accordance to what one would expect based on the experiments performed in *Friedmann* [2009]. The experiments showed that for too high transmission rates the delivery rate quickly approached a constant value. Hence, the observed linear increase can be understood as a fully saturated network, which simply delivers a constant rate of events, such that the number of dropped events necessarily increases linearly with the number of remote connections.

For more than 36 network connections the dropped events for chip 0 and chip 1 approach a plateau. This can not be explained by actual network drops, as the total event delivery rates are expected to be constant based on the experiments performed in *Friedmann* [2009]. However, the effect might result from strong additionally allocated synapse drivers, which can be responsible for a huge increase in ghost spikes. Since the effect appears equally on two

chips, which also show a very similar course of dropped events for the whole range of remote connections, this might point at a systematic source for this phenomenon.

Assuming an event loss which is zero for little network connections, we can consider the initial decrease as the range where only very few or even no events are dropped. In this case the decrease is caused by a "background" ghost event rate proportional to the number of allocated synapse drivers. Consequently, the range from 15 to 25 remote connections would indicate the regime within which the first events are getting lost, resulting in the curve exhibiting a trend change towards more dropped events. Hence we conclude that for the chosen network the multi-chip setup is expected to reliably realize up to approximately 20 remote connections.

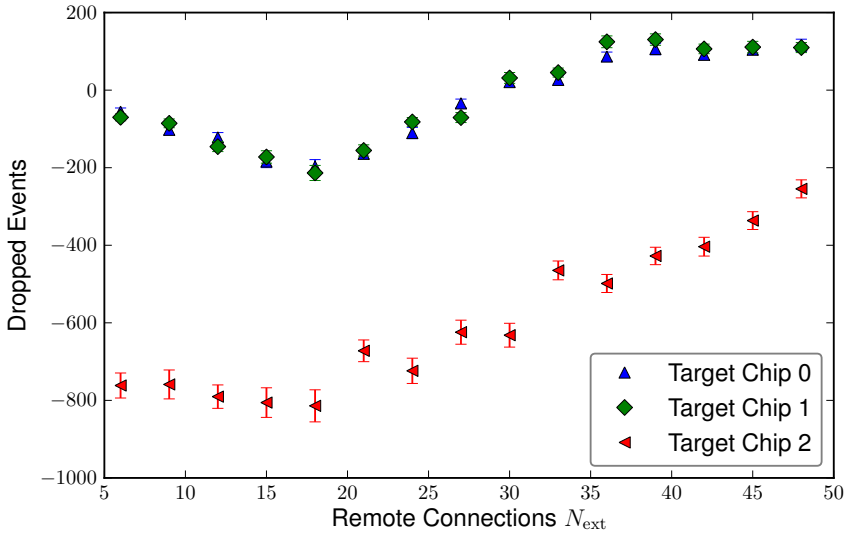


**Figure 3.25.:** Measured event drops caused by the network for different numbers of remote connections. The spikes are provided by a self-stabilizing architecture (see Section 2.3.2) on the sender Nathan. Subsequently, the events from a individual subsets of neurons are transmitted to each of three receiving Nathans. The self-stabilizing network mean spike activity has been found to be $12.2 \pm 9.5\,\text{Hz}$. *Please note*: Negative event drops indicate that more spikes have been measured on the receiving side than spikes have actually been sent. These spikes are caused by the ghost event issue described in Section 3.1.3.

**Conclusions Drawn from this Experiment** Despite the reported hardware issues the presented analysis experiment indicates that the bandwidth of the MCGN is high enough to sustain reliable spike transmission for approximately 15 to 25 remote connections for the given self-stabilizing architecture. Event though, the results are strongly distorted by the ghost event issue, the number of sustainable connections in a working setup can be expected to be higher, because therein no more ghost events need to be transmitted. The method for measuring the network drop rates presented in this section is generic enough to provide valuable information about the expectable network distortions for a large variety of experiments.

### The Multi-Chip Self-Stabilizing Network on Hardware

All prerequisites necessary for the mapping of arbitrary neural networks to the multi-chip system have been developed. Section 2.2 described the actual implementation of the individual algorithms responsible for carrying out the placement of neurons, the routing of intra and inter-chip connections, the parameter translation as well as the operation of the hardware. The correct individual functionality and the correct interplay of those components have been demonstrated via the Synfire-Chain experiment in Section 3.1.4.

Furthermore, very similar to the bandwidth analysis experiment in the multi-chip environment presented in Section 3.2.2, a test experiment based on the self-stabilizing liquid architectures has been performed on the multi-chip hardware system. But in contrast to the bandwidth analysis runs, the loopback mode has been deactivated, i.e. this time a real multi-chip liquid was set up. As suggested by the bandwidth studies, 20 remote connections have been used. However, the network dynamics in this test experiment were fully dominated by the activity induced by the ghost event issues described in Section 3.1.3. Consequently, the classification yielded results on chance level for all spike segments. The presentation of the gathered experimental results has been omitted in this work, because the measured dynamics are driven by technical artefacts and not by the self-stabilizing architecture at all. This points out that due to the hardware issues the multi-chip system is not yet ready for neuroscientific experiments. But as soon as these issues are resolved the experiment can and will be repeated.

# Conclusion and Outlook

The goal of this thesis was to develop and experimentally verify a fully functional framework for the mapping of arbitrary neural architectures onto inter-connected FACETS neuromorphic chips. For this purpose, algorithms to place neurons onto the hardware substrate, to route synaptic connections locally and globally, to translate parameters and support infrastructure to operate the hardware have been implemented. A synfire chain experiment has been performed to verify the correct mapping. Furthermore, a variety of benchmark and analysis results have been presented to study various technical issues and corresponding solutions.

Additionally, a Liquid State Machine based on a self-stabilizing neural architecture and a spike-based classifier have been implemented. This spike-based classifier has further been investigated for an in-hardware realization. With this in mind, a software environment has been realized, in which the readout can be trained under conditions that mimic specific hardware constraints. Porting the software-trained readout to actual hardware - which is possible due to the integration of the FACETS hardware into the PyNN concept - has yielded good classification performances comparable to those of the pure software prototype.

## Achievements

The most important result of this thesis is that a fully functional software framework is now available for the mapping of arbitrary networks onto multiple inter-connected Spikey chips by utilizing the event distribution network. Now, for the first time, non-hardware-experts can utilize the simulator-independent modeling language PyNN to set up and execute their experiments in such a multi-chip environment. For the accomplishment of this goal, various contributions have been made to the FACETS biology-to-hardware MappingTool in general. A new class of template nodes within the versatile graph-based data container of the MappingTool, for example, can improve the runtime performance of the mapping in future software revisions. The novel database-oriented strategy for the organization of parameter translation data serves as a test case for a future adoption into the mapping flow of the wafer-scale system.

For the routing of synaptic signals within the now available multi-chip setup, an optimization process involving simulated annealing was developed. The parameter values controlling this algorithm were studied and optimized on the basis of benchmark models. The routing analysis based on the KTH attractor model showed that the optimization process can have a significant positive impact on the overall synapse loss. The results further indicate a strong dependence of the potential synapse gain on the preceding placing. The NFC algorithm has shown to allow a routing outcome which realizes approximately 20 percent more synapses in the given network compared to a random placement. The precise value depends strongly on the neural network architecture to be mapped. However, one can expect the beneficial impact to increase with the amount of inhomogeneities in mutual connection densities and the size of the network.

Nevertheless, the overall synapse loss will constantly grow for increasing network sizes. This is unavoidable due to the hardware constraints, mainly the limited input count per neuron and the limited network bandwidths. Furthermore, it has been demonstrated that the optimization

rapidly converges to a stable synapse loss. Therefore, modelers can expect robustness against variations in the synapse count from run to run.

Beyond the scope of implementing the system, a Synfire-Chain network model has been realized to demonstrate and verify the functionality of the full PyNN-to-multi-chip system. Thereby, various hardware problems with the event distribution network in combination with Spikey chips of the 3rd revision have been identified. Most of these issues, which are described in more detail in Section 3.1.3, still need to be solved by the developers of the corresponding modules before the system can be beneficially exploited as a neuroscientific modeling tool.

To prepare and study such a neuroscientific application of the system, the foundation for a multi-chip liquid state machine has been laid out. Independent from the utilized neural architecture, a protocol for the general analysis of mean event drops by the network has been developed and tested on the liquid. Due to the mentioned hardware issues, the results are not expressive from a scientific point of view. However, when the hardware issues will be resolved, everything is prepared for the experiment. In another experiment using a single-chip scale variant of the liquid, the general suitability of the self-stabilizing network architecture to be used as a liquid has been shown by investigating its memory capacity. The perceptron showed significantly better classification results for an input previously separated by the network than directly fed-in input without liquid-based pre-processing.

Furthermore, concepts for the realization of a spike-based classifier directly on hardware have been investigated. For this purpose, a tool set for the in-software training of a tempotron with respect to hardware constraints has been developed. It has been demonstrated that the weight update rule can be applied to a LIF neuron with conductance-based synapses with certain restraints, although it has been originally deduced for current-based synapses. Selective parameter studies have been performed in software only to optimize the classification results. An early mapping of an in-software optimized tempotron to actual hardware showed good classification performances close to the results of the corresponding software prototype. The tempotron has been mapped to four different chips to show that the training provides sufficient robustness against variations between chips.

## Discussion and Outlook

This thesis has set the foundation for the Spikey-based neuromorphic exploration of neural architectures beyond single-chip boundaries, which allows for a whole new spectrum of realizable network models for future studies. However, the *a priori* available event distribution network, upon which the presented work builds, has been proven to be still occasionally unreliable and malfunctioning at the time of thesis submission. Therefore, the most critical improvement necessary for future work with the multi-chip system is to locate and solve these issues. The corresponding problem descriptions given in Section 3.1.3 provide a basis for this endeavor.

In the following potential feature extensions for the software framework as well as requirements of the neural network candidates imposed by the multi-chip hardware constraints will be discussed.

**Technical Aspects** The multi-chip framework realization provides, in its current state, access to most hardware features, but still leaves room for improvements. It would be desirable to provide multi-cast connections over the MCGN network. Due to a conceptual design decision to save FPGA resources, neurons can only be forwarded to a single transmit buffer queue (see

Section 1.1.4). To nevertheless enable spikes of a neuron to be transmitted to multiple targets one needs to employ the multi-cast functionality offered by the underlying MCGN network. Thus, intersections in the requested virtual connection sets need to be found using appropriate software methods, and those connections need to be put in separate connection bundles. The bundles could then be transmitted to multiple receivers by programming the MCGN routing tables accordingly. Another possible approach would be to extend the lookup tables, which would require a more modern FPGA, but enable neurons to be part of more than a single connection bundle. Further desirable features are the access to the programmable event transmission delays via PyNN and a differential experiment remapping, i.e. the automated biology-to-hardware translation of only a subset of the full already mapped model due to minor changes in the PyNN description. The latter could be used for selective parameter space explorations with a minimal software overhead. All proposed features can be implemented by extending the existing software, no conceptual problems are to be expected.

**Scalability Considerations** When it comes to scaling up the network to larger sizes, the software performance becomes a major concern. It is inept for the operation of a highly accelerated neuromorphic hardware device to be held back from efficient operation by its software backend. The results of the runtime analysis of the complete mapping flow suggest that for a fully automated workflow, the most significant performance gain is achievable by optimizing the placement step. If no fully-automated placing is required, one can use the semi-automated population placement instead, as has been implemented for and presented in this thesis (see Section 2.2.3). It has not been benchmarked because it requires appropriately sized local network structures for the manual distribution, which the unoptimized KTH model could not offer. From experience, it requires less computational steps than the fully automated version, since a big part of the mapping task is carried out by the user and it has also been optimized particularly for the chip-based system. Future contributions to any part of the MappingTool should consider using the GMPath language cautiously (see Section 1.3.1): It offers great flexibility but may have a significant contribution to the over all runtime when used in frequent loop iterations.

As introduced before, the multi-chip hardware system imposes some requirements on the realizable neural architectures, which are discussed in the following.

**Limited Network Bandwidth and Neuron Input Counts** If a mapping of an architecture onto the multi-chip system is to be poor of distortions it requires the original network to exhibit strong inhomogeneities in the mutual connection densities. Such characteristics allow for an effective clustering of local structures with higher connection densities and global structures with only sparse interconnectivity. In case such local structures exceed the capacity of a single chip, they will become heavily distorted due to the limited network bandwidth and the limited input count per neuron. Apart from the sheer number of connections within the network, the deliverable rates also need to be taken into consideration. For example, if the neurons sending via long distance connections tend to burst, events can get dropped randomly by the network infrastructure and would consequently induce a distortion. This distortion is difficult to analyze or compensate due to its random and highly dynamics-dependent nature.

A theoretical upper limit for an acceptable event rate with an effective event drop below 5% has been found to be at approximately 0.17 events/timestamp in experiments performed by

*Friedmann* [2009]. This corresponds to a chip-to-chip throughput of events with a maximum rate of 3.4 kHz in biological time. The given rate has been extrapolated from measurements from the previous chip generation and could not yet be verified for the current revision due to the hardware issues described in Section 3.1.3. However, the effective rate is expected to be lower if the MCGN topology becomes more complex, since then more local and global network resources need to be shared by multiple participants. Here, local resources refer to the buffers implemented in the sender and receiver logic on each FPGA and global resources to the limited MCGN network resources.

Furthermore, the maximum input count of each neuron remains constant while scaling up the network. The available 256 axonal connections, which can independently project onto the post-synaptic neuron are shared among local neurons and network event sources. Hence, for larger networks, it can be a useful technique to introduce a distance-dependent connection probability into the network architecture. Often, this does not impose any significant restraint from the biological point of view, since biological neural networks commonly exhibit high local connection densities and sparse projections onto distant regions in the nervous system. For example, activity found in thalamus has a significant impact onto the network dynamics in the cortex, despite their sparse connectivity [*Bruno and Sakmann*, 2006].

**Synaptic Delays** Another restraint for the realizable network models is given by the inhomogeneous hardware event delays. While local synaptic delays are unrealistically short, they enter a biological regime for the inter-chip communication. The minimum network delay for the current hardware revision was predicted by *Friedmann* [2009] to be 9.1 ms. During the Synfire-Chain experiment presented in Section 3.1.4, the programmable delays have been found to be non-functional for the current revision of the Spikey chip. In theory, these programmable synaptic delays offer a chance to realize a wider variety of network architectures, when used wisely by the experimentalist (see e.g. *Kremkow et al.*, 2010). If the local neural structures, as discussed above, start to exceed the single chip capacity, the difference in delays can induce distortions with significant impact on the dynamics. For example, the realization of networks with WTA elements could run into problems due to the delay inhomogeneities, since in most spike-based WTA architectures the timing of lateral inhibition is crucial.

**Multi-Chip Realization of Computationally Powerful Architectures** In conclusion, it is architectures with local connection inhomogeneities that exhibit the least distortions in their multi-chip realization. The embedded local network structures with high connectivity should not exceed single-chip capacity. The introduction of distance-dependent connection probabilities based on the topological structure of the MCGN might be useful. If these conditions are met, the multi-chip system can offer access to a large variety of interesting new network models and dynamics. The requirements concerning the network inhomogeneities and delays can be attenuated in future experiments by using higher FPGA clock frequencies and analog calibrations to smaller intrinsic time constants. Both will yield an effectively higher network bandwidth. By utilizing more of the available MGT links, the throughput can further be increased. In the default configuration only four of eight MGT links per Nathan board are connected via the backplane topology. Note that this can only improve the throughput if the limiting bottleneck is not already determined by the hierarchically higher event network.

The KTH attractor model, so far only utilized for the routing analysis, might be a good candidate for a multi-chip implementation. Each hypercolumn works as a WTA network mostly

independent from other hypercolumns. The long range connections between hypercolumns provide additional robustness for the attractor network. One could think of realizing one or more complete hypercolumns per chip, which are connected over the MCGN to locally identical attractor networks on different chips. The long-range connections then induce the additional robustness typical for the KTH attractor model.

A self-stabilizing network architecture based on short-term synaptic plasticity, which has already been realized on the single-chip system, has been translated to a multi-chip network description. Even though this setup could not yet be neuroscientifically investigated due to the hardware issues described in Section 3.1.3, one can expect it to be suitable for a multi-chip realization. The event drop analysis presented in this thesis indicates that the connection bandwidth should be sufficient, despite the occurring ghost events. The experiment is ready to be performed as soon as the event distribution network works reliably. It can be exploited e.g. in liquid state machine experiments.

**On-Chip Classification** The early mapping of a software-trained tempotron to the actual hardware system has shown good classification results. However, the dynamic input range has been artificially constrained to excitatory synapses only (see Section 2.3.3). The algorithmic flipping of synapse types from excitatory to inhibitory or vice versa could otherwise induce irreparable discontinuities in the learning process. One approach to increase the available spectrum of input efficacies could be a random initialization of synapses to either excitatory or inhibitory type. This idea had previously been neglected, since the trend whether a synapse would become more likely excitatory or inhibitory during the training process is *a priori* unknown. A more sophisticated training process could implement an initial training of an unconstrained tempotron with current-based synapses on the same input. The determined weight distribution could subsequently be used to initialize the synapses of the constrained and conductance-based tempotron to either inhibitory or excitatory type. Furthermore, assuming a rather constant mean membrane potential, one could try to compensate the weight update distortions by rescaling synaptic efficacies for excitatory and inhibitory synapses by a constant factor. This approach might be constrained severely by the limited synaptic weights, though.

So far the processing of input by the liquid and the subsequent classification have been treated independently. However, the results from the hardware tempotron suggest a bundling of both the liquid and the classifier on a single chip. Beyond merely overcoming bandwidth limitations, one would get a Liquid State Machine with any-time computing properties realized completely on a neuromorphic-hardware substrate.

# A. Parameter List

## A.1. Shared Chip Parameters

The following parameters are shared within a set of entities. Changing a single parameter can even effect another one considered independend.

**Table A.1.:** Timing Parameters

| Name | default | range | description | annotation |
|---|---|---|---|---|
| `tsense` | 150 | | time until the output of synapse sram bitline reading is valid (in units of external chip clock periods) | does only affect the read back of the synapse array, e.g. for reading the weights after a STDP experiment |
| `tpcsec` | 30 | | pre-charge time for secondary read when processing correlations (given in external chip clock periods) | STDP controller time constant |
| `tpcorperiod` | 360 | | minimum time used for correlation processing on a single row (given in external chip clock periods) | STDP controller time constant |

**Table A.2.:** External Analog Parameters

| Name | default | range | description | annotation |
|---|---|---|---|---|
| `irefdac` | 1.6 | 0.02 - 1.7 | DAC reference current. Determines possible hardware currents | min. prog. current = `irefdac` $\cdot\, 1/(10 \cdot 1024)$<br>max. prog. current = `irefdac` $\cdot\, 1023/10$ |

| | | | | |
|---|---|---|---|---|
| `vcasdac` | 1.6 | 0.02 - 1.7 | cascode DAC voltage (given in V) | Never touch this value |
| `vm` | 0.0 | 0.02 - 0.3 | pre-charge amplitude for STDP correlation measurement. | the larger vm, the smaller the charge stored per measured pre-/post-synaptic correlation |
| `vrest` | 0.0 | 0.0 - 1.7 | end value of falling voltage ramp (given in V) | efficacy is high due to huge impact on integral over EPSP amplitude (shallow curve) |
| `vstart` | 0.25 | 0.02 - 1.7 | start value of rising voltage ramp (given in V) | efficacy is low due to small impact on integral over EPSP amplitude (steep curve) |

**Table A.3.:** Programmable Current

| Name | default | range | description | annotation |
|---|---|---|---|---|
| `outamp[0..7]` | 0.3 | | bias current for 50Ω membrane voltage monitors | should be equal to avoid confusion. Normally only the muxed membrane voltage monitor is soldered. |
| `outamp[8]` | 0.0 | 0.0 | current memory for `ibtest_pin` | should be 0.0 |

**Table A.4.:** Programmable BiasB Currents

| Name | default | range | description | annotation |
|---|---|---|---|---|
| vdtc[0..3] | 0.7 | 0.02 - 2.0 | adjusts the STP time constant (spike history) | higher current yields shorter averaging window time |
| vcb[0..3] | 1.25 | 0.02 - 2.0 | spike driver comparator bias, compares rising ramp against drviout | default value well tested and should be set to a high value to limit overshoot of steep rising voltage ramp |
| vplb[0..3] | 0.15 | 0.02 - 2.0 | spike driver pulse length bias | higher currents yield shorter internal pulses. Important for STP |

| | | | | |
|---|---|---|---|---|
| Ibnoutampba | 0.1 | 0.02 - 2.0 | add to the neuron out amp bias for the membrane monitor | does not influence the experiment |
| Ibnoutampbb | 0.4 | 0.02 - 2.0 | add to the neuron out amp bias for the membrane monitor | does not influence the experiment |
| Ibcorreadb | 0.6 | 0.02 - 2.0 | correlation read out bias | STDP parameter |

Wherever two Voltages e.g.: `Ei`$X$ appear they represent the parameter for an entity with address  mod $2 = X$.

**Table A.5.:** Programmable Vout-Parameters

| Name | default | range | description | annotation |
|---|---|---|---|---|
| `vout[0] = Ei0`<br>`vout[1] = Ei1` | 1.0 | 0.02 - 1.7 | inhibitory reversal potentials | |
| `vout[2] = El0`<br>`vout[3] = El1` | 1.0 | 0.5 - 1.7 | leakage reversal potentials | |
| `vout[4] = Er0`<br>`vout[5] = Er1` | 1.0 | 0.02 - 1.7 | reset potentials | |
| `vout[6] = Ex0`<br>`vout[7] = Ex1` | 1.3 | 0.02 - 1.7 | excitatory reversal potentials | |
| `vout[8] = Vclra` | 1.0 | 0.8 - 1.15 | acausal storage voltage clear bias (capacitor in synapse array) | Higher voltage bias yields less charge stored on capacitor |
| `vout[9] = Vclrc` | 1.0 | 0.02 - 1.7 | causal storage clear voltage bias (capacitor in synapse array) | Higher voltage bias yields less charge stored on capacitor |
| `vout[10] = Vcthigh` | 1.0 | 0.9 - 1.6 ! | STDP correlation threshold high (step up in LUT) | must stay below 1.0V |
| `vout[11] = Vctlow` | 1.0 | 0.5 - 0.8 ! | STDP correlation threshold low (step down in LUT) | must stay below 1.0V |
| `vout[12] = Vfac0`<br>`vout[13] = Vfac1` | 0.02 | 0.02 - 1.7 | STF reference voltage | responsible for facilitation and should be set to a low value |
| `vout[14] = Vstdf0`<br>`vout[15] = Vstdf1` | 1.1 | 0.02 - 1.7 | STP capacitor high potential | |
| `vout[16] = Vt0`<br>`vout[17] = Vt1` | 1.1 | 0.02 - 1.1 | neuron spike threshold voltage | |

| | | | | |
|---|---|---|---|---|
| vout[18] = Vcasneuron | 1.6 | 0.02 - 1.7 | neuron input gate cascode voltage | never touch this value |
| vout[19] = Vresetdll | 1.1 | 0.02 - 1.7 | delay locked loop(dll) reset voltage. Is internally adjusted starting from the reset voltage | If not properly adjusted the 16 time bins doesn't get treated equally and some may stay empty |
| vout[20] = aro_dllvctrl | – | 0.02 - 1.7 | delay locked loop readout control readout voltage | only relevant for spikey v2 & and only bias important |
| vout[21] = aro_pre1b | – | 0.02 - 1.7 | spike input buf 1 presyn | only relevant for spikey v2 & and only bias important |
| vout[22] = aro_selout1hb | – | 0.02 - 1.7 | spike input buf 1 selout | only relevant for spikey v2 & and only bias important |

The vouts are used to generate currents via a Operational Transconductance Amplifier (OTA) and therefore need corresponding bias currents which are listed below.

**Table A.6.:** Programmable Voutbias-Parameters

| Name | default | range | description | annotation |
|---|---|---|---|---|
| voutbias[0] = IbEi0<br>voutbias[1] = IbEi1 | 2.5 | 0.02 - 2.5 | inhibitory reversal potential biases | |
| voutbias[2] = IbEl0<br>voutbias[3] = IbEl1 | 2.5 | 0.02 - 2.5 | leakage reversal potential biases | |
| voutbias[4] = IbEr0<br>voutbias[5] = IbEr1 | 2.5 | 0.02 - 2.5 | reset potential biases | |

| | | | | |
|---|---|---|---|---|
| `voutbias[6] = IbEx0` `voutbias[7] = IbEx1` | 2.5 | 0.02 - 2.5 | excitatory reversal potential biases | |
| `voutbias[8] = IbVclra` | 2.5 | 0.02 - 2.5 | bias of the acausal storage clear voltage bias | |
| `voutbias[9] = IbVclrc` | 2.5 | 0.02 - 2.5 | bias of the causal storage clear voltage bias | |
| `voutbias[10] = IbVcthigh` | 2.5 | 0.02 - 2.5 | STDP correlation threshold high bias | |
| `voutbias[11] = IbVctlow` | 2.5 | 0.02 - 2.5 | STDP correlation threshold low bias | |
| `voutbias[12] = IbVfac0` `voutbias[13] = IbVfac1` | 2.5 | 0.02 - 2.5 | STP reference voltage bias | responsible for facilitation |
| `voutbias[14] = IbVstdf0` `voutbias[15] = IbVstdf1` | 2.5 | 0.02 - 2.5 | STP capacitor high potential bias | |
| `voutbias[16] = IbVt0` `voutbias[17] = IbVt1` | 2.5 | 0.02 - 2.5 | neuron threshold voltage bias | |
| `voutbias[18] = IbVcasneuron` | 2.5 | 0.02 - 2.5 | neuron input cascode gate bias current | |
| `voutbias[19] = IbVresetdll` | 2.5 | 0.02 - 2.5 | dll reset bias | |

| | | | | |
|---|---|---|---|---|
| `voutbias[20] = IbAro_dllvctrl` | – | 0.02 - 2.5 | delay locked loop readout control readout voltage bias | only relevant for spikey v2 |
| `voutbias[21] = IbAro_pre1b` | – | 0.02 - 2.5 | spike input buf 1 presyn bias | only relevant for spikey v2 |
| `voutbias[22] = IbAro_selout1hb` | – | 0.02 - 2.5 | spike input buf 1 selout bias | only relevant for spikey v2 |

## A.2. Unique Parameters

These parameters are unique for every entity on the chip. Apart from that they still might suffer from parasitic effects.

**Table A.7.:** Neuron Parameters

| Name | default | range | description | annotation |
|---|---|---|---|---|
| `ileak` | 0.1 | 0.05-2.00 | membrane leakage conductance | used for membrane time constant calibration |
| `icb` | 0.2 | 0.05-2.00 | threshold comparator bias current | the larger the current gets the faster the operational amplifier responds but the narrower the dynamic range gets. |
| `config[0]` | false | true/false | not implemented | binary configuration flag |
| `config[1]` | false | true/false | record membrane voltage | binary configuration flag |
| `config[2]` | true | true/false | record spikes | binary configuration flag |
| `config[3]` | false | true/false | not implemented | binary configuration flag |

**Table A.8.:** Synapse Driver Parameters

| Name | default | range | description | annotation |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| `drviout` | 1.0 | 0.02 - 2.0 | upper threshold for the voltage ramp | |
| `adjdel` | 0.5 | 0.02 - 2.0 | delay for digital spike signal compared to analog voltage ramp for STDP correlation processing. | |
| `drvifall` | 1.0 | 0.02 - 2.0 | current to control falling ramp gradient | |
| `drvirise` | 1.0 | 0.02 - 2.0 | current to control rising ramp gradient | |
| `config[0..1]` | 0 | [0,1,2,3] | interpreation:<br>0 = source is playback memory<br>1 = input from adjacent block<br>2 = input equals input from previous row (feature to increase dynamic range for weights)<br>3 = input from same block | source configuration of synapse driver |
| `config[2]` | true | true/false | synapse driver is excitatory[1] | binary configuration flag |
| `config[3]` | false | true/false | synapse driver is inhibitory[1] | binary configuration flag |
| `config[4]` | false | true/false | STP enable | binary configuration flag |
| `config[5]` | false | true/false | facilitating synapse(false)/depressing synapse(true) | binary configuration flag. The STP implementation is exclusive facilitating or depressing for every syn driver |
| `config[6]` | false | true/false | use C2. Conductance parallel to active partition base conductance | binary configuration flag. Increases STP impact |
| `config[7]` | false | true/false | use C4. Conductance parallel to active partition base conductance | binary configuration flag. Increases STP impact |

[1] if either inhibitory and excitatory flag are set, then the inhibitory effect outbalances. If neither is set, then the syn driver is switched off

# B. Program Code Listings

## B.1. Dijkstra's Algorithm Listing

**Algorithm 2** Modified *Dijkstra's Algorithm*

```
function DIJKSTRA(Graph, source)
    for v in Graph do                                              ▷ Initialization
        dist[v] ← ∞
        previous[v] ← ∅
    dist[source] ← 0                                   ▷ Distance from source to source
    Q ← Graph                                          ▷ set Q of not yet optimized nodes
    while Q do                                                     ▷ The main loop
        u ← min(Qcapdist[])
        if dist[u] = ∞ then          ▷ all remaining vertices are inaccessible from source
            break
        Q ← Q \ {u}
        for v in neighbor(u, Q) do                          ▷ neighbor of node u in set Q
            alt ← dist[u] + 1
            if alt < dist[v] then
                dist[v] ← alt; previous[v] ← u
            if alt = dist[v] then
                previous[v] ← previous[v] + [u]
    routes[][]
    for targetinGraph do
        GetRoute(previous[], target, source, routes[][], ∅)        ▷ recursive path extraction
    return routes[][]

function GETROUTE(previous[], u, source, route[], routes[][])
    if route[0] ≠ source then
        while previous[u] ≠ ∅ do
            new_route[] ← [u] + route[]
            GetRoute(previous[], previous[u], source, new_route[], routes[][])
    else
        routes[u][] ← routes[u][] + route[]
```

# C. Figures and Tables

## C.1. Distribution of Dropped Network Events

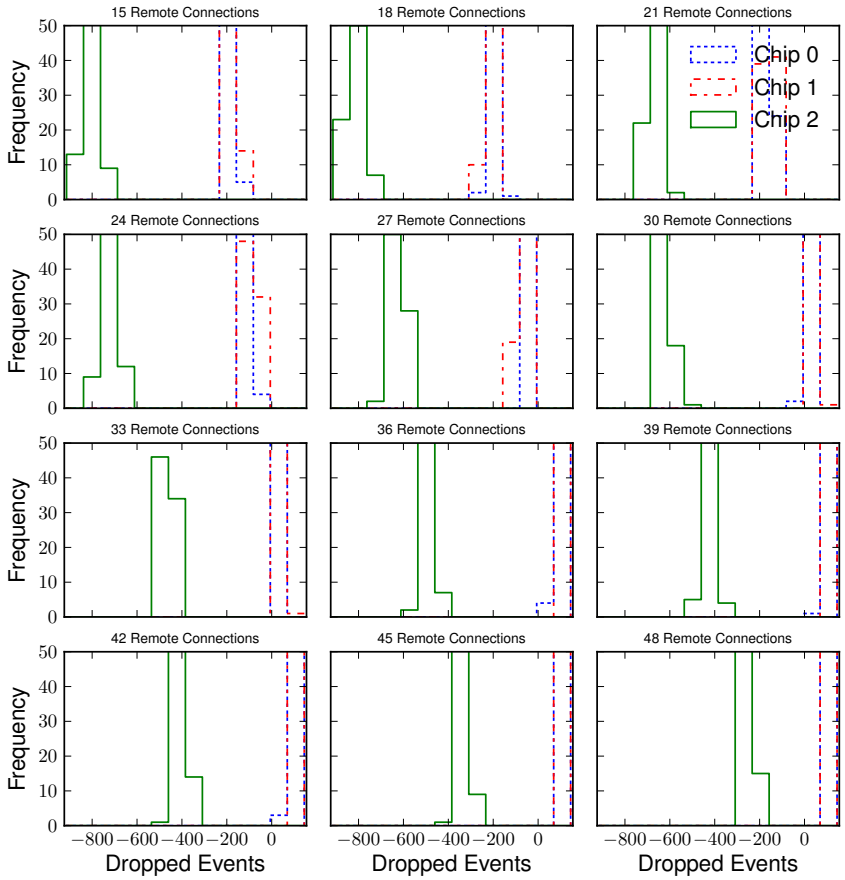


**Figure C.1.:** The distribution of event drops caused by the network for different numbers of remote connections. The spikes are provided by a self-stabilizing architecture (see Section 2.3.2) on the sender Nathan. Subsequently, the events from a individual subsets of neurons are transmitted to each of three receiving Nathans. The self-stabilizing network mean spike activity has been found to be $12.2 \pm 9.5\,\mathrm{Hz}$. The negative event rates indicate, that more spikes have been received than have been actually sent The extra events are cause by the ghost event issue described in Section 3.1.3.

# D. Acronyms

**AdEx** Adaptive Exponential Integrate-and-Fire . . . . . 11

**ANN** Artificial Neural Network . . . . . 24

**API** Application Programming Interface . . . . . 12

**ARQ** Automatic Repeat reQuest . . . . . 6

**ASIC** Application-Specific Integrated Circuit . . . . . 3

**CMOS** Complementary Metal-Oxide-Semiconductor . . . . . 3

**DNC** Digital Network Chip . . . . . 11

**EVNET** Event Network . . . . . 8

**FACETS** Fast Analog Computing with Emergent Transient States

**FIFO** First In, First Out . . . . . 9

**FPGA** Field Programmable Gate Array . . . . . 5

**FSM** Finite State Machine . . . . . 17

**GSS** Global Start Signal . . . . . 7

**HAL** Hardware Abstraction Layer

**HICANN** High Input Count Analog Neuronal Network . . . . . 11

**LIF** Leaky-Integrate-and-Fire . . . 3

**LSM** Liquid State Machine . . . 2

**LUT** LookUp Table . . . 8

**lvc** Logical Virtual Connection . . . 9

**LVDS** Low-Voltage Differential Signaling . . . 7

**MCGN** Multi-Class Gigabit Network . . . 5

**mfi** multi FIFO index . . . 9

**MGT** Multi-Gigabit Transceiver . . . 6

**NFC** N-Force-Cluster . . . 16

**OTA** Operational Transconductance Amplifier . . . 91

**PCB** Printed Circuit Board . . . 5

**PCI** Peripheral Component Interconnect . . . 6

**PSP** Post-Synaptic Potential . . . 48

**QoS** Quality of Service . . . 6

**RSNP** Regular Spiking Non Pyramidal

**SC** SlowControl . . . 5

**SDRAM** Synchronous Dynamic Random Access Memory . . . 13

**STDP** Spike-Timing-Dependent Plasticity . . . 4

**STL** Standard Template Library . . . . . 32

**STP** Short-Term Plasticity . . . . . 4

**WTA** Winner Take All

**XML** Extensible Markup Language . . . . . 13

# Bibliography


Albert, M., Liquid Computing mit Neuromorpher Hardware, Bachelor thesis (German), Ruprecht-Karls-Universität, Heidelberg, HD-KIP-10-43, `http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=2042`, 2010.

Alpaydin, E., 2004.

Bill, J., Self-stabilizing network architectures on a neuromorphic hardware system, Diploma thesis (English), Ruprecht-Karls-Universität, Heidelberg, HD-KIP-08-44, `http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=1893`, 2008.

Bill, J., K. Schuch, D. Brüderle, J. Schemmel, W. Maass, and K. Meier, Compensating inhomogeneities of neuromorphic VLSI devices via short-term synaptic plasticity, *Front. Comp. Neurosci.*, *4*, 2010.

Braitenberg, V., and A. Schüz, *Anatomy of the Cortex: Statistics and Geometry*, Springer Verlag, Berlin, Heidelberg, New York, 1991.

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

Brüderle, D., Neuroscientific modeling with a mixed-signal VLSI hardware system, Ph.D. thesis, Ruprecht-Karls-Universität, Heidelberg, 2009, document no. HD-KIP-09-30.

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*, 2009.

Bruno, R. M., and B. Sakmann, Cortex Is Driven by Weak but Synchronously Active Thalamocortical Synapses, *Science*, *312*, 1622–1627, 2006.

Cortes, C., and V. Vapnik, Support-vector networks, *Machine Learning*, *20*, 273–297, 1995.

Davison, A., E. Muller, D. Brüderle, and J. Kremkow, A common language for neuronal networks in software and hardware, *The Neuromorphic Engineer*, 2010.

Dijkstra, E. W., A note on two problems in connexion with graphs, *Numerische Mathematik*, *1*, 269–271, 1959.

Downarowicz, T., Law of series/poisson process, *Scholarpedia*, *3*, 3922, 2008.

Ehrlich, M., K. Wendt, and R. Schüffny, Parallel mapping algorithms for a novel mapping & configuration software for the FACETS project, in *CEA'08: Proceedings of the 2nd WSEAS International Conference on Computer Engineering and Applications*, pp. 152–157, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2008.

Ehrlich, M., K. Wendt, L. Zühl, R. Schüffny, D. Brüderle, E. Müller, and B. Vogginger, A software framework for mapping neural networks to a wafer-scale neuromorphic hardware system, in *Proceedings of the Artificial Neural Networks and Intelligent Information Processing Conference (ANNIIP) 2010*, pp. 43–52, 2010.

FACETS D7-13, Demonstrate the operation of the FACETS stage 2 software system, FACETS Deliverable D7-13, 2010, UHEI and TUD.

FACETS M7-5, Verify that the layer-2 communication reaches the bandwidth requirements for a multi-wafer system, including the host communication via GBit-Ethernet, FACETS Deliverable M7-5, 2010, UHEI and TUD.

Fieres, J., J. Schemmel, and K. Meier, Realizing biological spiking network models in a configurable wafer-scale hardware system, in *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN'08)*, IEEE Press, 2008.

Friedmann, S., Extending a hardware neural network beyond chip boundaries, Diploma thesis (English), Ruprecht-Karls-Universität, Heidelberg, HD-KIP-09-41, `http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=1938`, 2009.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman, 1994.

Garey, M., and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

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

Gütig, R., and H. Sompolinsky, The tempotron: a neuron that learns spike timing-based decisions, *Nat Neurosci*, *9*, 420–428, 2006.

Hastings, W. K., Monte carlo sampling methods using markov chains and their applications, *Biometrika*, *57*, 1970.

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

Hopcroft, J. E., and J. D. Ullman, *Formal languages and their relation to automata*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.

Jaeger, H., The "echo state" approach to analysing and training recurrent neural networks, *Tech. Rep. GMD Report 148*, German National Research Center for Information Technology, 2001.

Kanungo, T., D. Mount, N. Netanyahu, R. S. Christine Piatko, and A. Wu, An efficient k-means clustering algorithm: Analysis and implementation, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *24*, 881, 2002.

Kempf, B., The boost.threads library, *Dr. Dobb's*, 2001.

Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi, Optimization by simulated annealing, *Science*, *220*, 671–680, 1983.

Kremkow, J., L. Perrinet, G. Masson, and A. Aertsen, Functional consequences of correlated excitatory and inhibitory conductances in cortical networks., *J Comput Neurosci*, *28*, 579–594, 2010.

Landau, D. P., and K. Binder, *A guide to Monte Carlo simulations in statistical physics*, 2005.

Lundqvist, M., M. Rehn, M. Djurfeldt, and A. Lansner, Attractor dynamics in a modular network model of neocortex, *Computation in Neural Systems*, 2006.

Maass, W., T. Natschläger, and H. Markram, Real-time computing without stable states: A new framework for neural computation based on perturbations, *Neural Computation*, *14*, 2531–2560, 2002.

Maass, W., T. Natschläger, and H. Markram, *Computational models for generic cortical microcircuits*, chap. 18, pp. 575–605, J. Feng, Boca Raton, 2004.

Meyer, D., F. Leisch, and K. Hornik, The support vector machine under test, *Neurocomputing*, *55*, 169 – 186, 2003.

Meyn, S., and R. Tweedie, *Markov chains and stochastic stability*, Springer Verlag, Berlin, Heidelberg, New York, 1993.

Millner, S., Andreas, Grubl, K. Meier, J. Schemmel, and M.-O. Schwartz, A VLSI implementation of the adaptive exponential integrate-and-fire neuron model, in *Advances in Neural Information Processing Systems (NIPS), accepted*, 2010.

MongoDB, A scalable, high-performance, open source, document-oriented database written in C++, `http://www.mongodb.org/`, 2010.

Morrison, A., M. Diesmann, and W. Gerstner, Phenomenological models of synaptic plasticity based on spike timing, *Biological Cybernetics*, *98*, 459–478, 2008.

Nethercote, N., and J. Seward, Valgrind: A framework for heavyweight dynamic binary instrumentation, in *ACM SIGPLAN*, 2007.

Novikoff, A. B., On convergence proofs for perceptrons, in *Proceedings of the Symposium on the Mathematical Theory of Automata*, vol. 12, pp. 615–622, 1963.

Pacheco, P. S., *Parallel programming with MPI*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

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

Rosen-Zvi, M., and I. Kanter, Training a perceptron in a discrete weight space, *Physical Review E*, *64*, 2001.

Rosenblatt, F., The perceptron: a probabilistic model for information storage and organization in the brain, *Psychological Review*, *65*, 386–408, 1958.

Savage, J. E., and M. Zubair, Evaluating multicore algorithms on the unified memory model, *Scientific Programming*, 2009.

Schemmel, J., S. Hohmann, K. Meier, and F. Schürmann, A mixed-mode analog neural network using current-steering synapses, *Analog Integrated Circuits and Signal Processing*, *38*, 233–244, 2004.

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., D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner, A wafer-scale neuromorphic hardware system for large-scale neural modeling, in *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10)*, IEEE Press, 2010.

Schilling, M., A highly efficient transport layer for the connection of neuromorphic hardware systems, Diploma thesis, Ruprecht-Karls-Universität, Heidelberg, HD-KIP-10-09, `http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=2000`, 2010.

Schmuker, M., M. Weidert, and R. Menzel, A network model for learing-induced changes in odor representation in the antennal lobe, in *Proceedings of the second french conference on Computational Neuroscience*, edited by L. U. Perrinet and E. Daucé, Marseille, 2008.

Sussillo, D., T. Toyoizumi, and W. Maass, Self-Tuning of Neural Circuits Through Short-Term Synaptic Plasticity, *J Neurophysiol*, *97*, 4079–4095, 2007.

Thomson, A. M., and C. Lamy, Functional maps of neocortical local circuitry, *Frontiers in neuroscience*, *1*, 19–42, 2007.

Tsodyks, M., and H. Markram, The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability, *Proceedings of the national academy of science USA*, *94*, 719–723, 1997.

Vogginger, B., Testing the operation workflow of a neuromorphic hardware system with a functionally accurate model, Diploma thesis, Ruprecht-Karls-Universität, Heidelberg, HD-KIP-10-12, `http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=2003`, 2010.

Wendt, K., M. Ehrlich, and R. Schüffny, Gmpath - a path language for navigation, information query and modification of data graphs, in *Proceedings of the Artificial Neural Networks and Intelligent Information Processing Conference (ANNIIP) 2010*, pp. 31–42, 2010.

Yue, M., A simple proof of the inequality for the ffd bin-packing algorithm, *Acta Mathematicae Applicatae Sinica*, *Volume 7, Number 4*, 321–331, 1990.

# Acknowledgments (Danksagungen)

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

Prof. Sylvie Renaud for being referee for this thesis.

Daniel for being Daniel. Die nächste Diplomarbeit auf jeden fall wieder bei ihm.

Bernie, Eric, Mihai, Paul und Tom für die Unterstützung und das tägliche Kickerzeremoniell.

Simon und Andi für das amüsante Fehlersuchen.

Allen anderen Visionären für die super Atmosphere.

Bärbel für das Bewältigen des temporären Freizeitzuwachses und lecker Kuchen.

My family.

## Statement of Originality (Erklärung):

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Ich versichere, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, June 20, 2011

.....
(signature)