Dissertation

submitted to the

Combined Faculties for Natural Sciences and Mathematics

of the Ruperto Carola University of Heidelberg, Germany

for the degree of

Doctor of Natural Sciences

Presented by

Dipl.-Phys. Sebastian Jeltsch

born in Ludwigsburg, Germany

Date of oral examination: July 23, 2014

A Scalable Workflow for a Configurable Neuromorphic Platform

Referees:	Prof. Dr. Karlheinz Meier
	Prof. Dr. Bernd Jähne

A Scalable Workflow for a Configurable Neuromorphic Platform

This thesis establishes a scalable multi-user workflow for the operation of a highly configurable, large-scale neuromorphic hardware platform. The resulting software framework provides unified low-level as well as parallel high-level access. The latter is realized by an efficient abstract neural network description library, an automated translation of networks into hardware specific configurations and an experiment server infrastructure responsible for scheduling and executing experiments. Scalability, manual guidance and a broad support for handling hardware imperfections render the model translation process suitable for large networks as well as large-scale neuromorphic systems. Networks with local connectivity, random networks and cortical column models are explored to study the topological aptitude of the neuromorphic platform and to benchmark the workflow. Depending on the model, performance improvements of more than two orders of magnitude have been achieved over a previous implementation. Additionally, an automated defect assessment for hardware synapses is introduced, indicating that most synapses are available for model emulation.

In a second study, a tempotron-based hardware liquid state machine has been developed and applied to different tasks, including a memory challenge and digit recognition. The trained tempotron inherently compensates for fixed pattern variations making the setup suitable for analog neuromorphic hardware. The achieved performance is comparable to reference software simulations.

Ein skalierbarer Workflow für eine konfiguierbare neuromorphe Plattform

Die vorliegende Arbeit stellt einen skalierbaren Mehrbenutzer-Workflow für den Betrieb einer sehr flexiblen, großskaligen neuromorphen Hardwareplattform bereit. Die entwickelte Software erlaubt einheitlichen, hardwarenahen sowie parallelen, abstrahierten Zugriff. Letzteres ist realisiert mittels einer effizienten Softwarebibliothek für die Beschreibung neuronaler Netze, einer automatisierten Übersetzung von Netzen in hardwarespezifische Konfigurationen und einer verteilten Infrastruktur für die koordinierte Ausführung von Experimenten. Skalierbarkeit, manuelle Kontrolle und eine umfassende Behandlung von Hardwaredefekten ermöglichen das Übersetzen großer neuronaler Netze für großskalige neuromorphe Systeme. Netze mit lokaler Struktur, mit zufälliger Struktur sowie Modelle von kortikalen Säulen wurden verwendet um die Grenzen des neuromorphen Systems zu untersuchen und die Leistungsfähigkeit der Software zu vermessen. Im Vergleich zu einer bestehen Implementierung wurden, abhängig von den Eigenschaften des Netzes, Leistungsverbesserungen von bis zu zwei Größenordnungen erzielt. Weiterhin wurde mittels einer automatisierten Untersuchung der Synapsenschaltungen gezeigt, dass die Mehrzahl der Synapsen funktioniert.

Im zweiten Teil wurde eine tempotron-basierte Liquid State Machine in Hardware realisiert. Unter anderem wurden das Erinnerungsvermögen, sowie die Fähigkeit Zeichen zu erkennen, getestet. Fertigungsbedingte Variationen werden durch das Training kompensiert. Die erzielten Ergebnisse sind vergleichbar mit den Resultaten entsprechender Softwaresimulationen.

Contents

Introduction

Everyday we are surrounded by powerful computation devices: notebooks, mobile phones and the internet are virtually everywhere. At the same time, computer clusters are used to study our consumer habits and preferences. Despite the different architectures and physical extents, all these systems require high single instruction throughput for efficient computation. However, performance limits of this kind of architectures are approaching. Clock frequencies are limited by power density (Kim et al., 2003) and modern process technologies have reduced transistor gates to merely a few atom layers (Schulz, 1999). Over the last few years, the focus has shifted to concurrent architectures, both with shared and distributed memory. The latter is mostly implemented in terms of computer clusters, which allows for extensive horizontal scaling. Their extent is only limited by cost, heat dissipation and sensitivity to single component failure (Hennessy and Patterson, 2012). Developing efficient algorithms for conventional distributed systems is challenging and problems that cannot easily be partitioned into individual subproblems are often solved more efficiently on a single computer.

Nature, on the other hand, came up with its own distributed computational architecture: the brain. The human brain, in particular, contains approximately 10^{11} neurons, which are interconnected via 10^{15} synapses forming complex networks to carry out massively parallel computation (Azevedo et al., 2009; Drachman, 2005). Information is exchanged via action potentials or spikes, a sparse spatio-temporal code (Attwell and Laughlin, 2001). The resulting design, which evolved over millions of years, is extremely efficient in terms of energy. The human brain consumes only about 20 W of power (Clarke and Sokoloff, 1999). Moreover, the design is extremely robust. It can cope with disturbance by directing attention (Corbetta and Shulman, 2002), adapt to changing environments (Cohen et al., 2007) and in some cases even recover from severe lesions (Robertson and Murre, 1999).

Tasks that can easily be captured in algorithmic descriptions are typically solved more efficiently by computers, like arithmetic and numerics. However, the human brain outperforms state of the art technical solutions in areas like pattern completion, complex body movement and higher order abstractions. Tasks that become more and more important, e.g., for robotics.

Understanding the human brain is one of the great challenges of the 21st century (White House, 2014). An endeavor that is ambitiously pursued among others by the BrainScaleS project (2014) and the Human Brain Project (Markram, 2012b). These projects bring together scientists from different fields, like biology, medicine, psychology, mathematics and computer science to build comprehensive brain maps and to develop a comprehensive theory linking models of limited scope across different spatial and temporal scales. Another long-term goal is the establishment of a novel brain-inspired computing paradigm and novel computing architectures to tackle the above mentioned shortcomings of traditional computers.

Detailed biological measurements have built the foundations for the knowledge we have today about the inner workings of the brain. Unfortunately, these measurements always provide access to only a limited number of variables in the system (Stevenson and Kording, 2011) Computational neuroscience has therefore become a second invaluable approach to study neural networks from the bottom up on a very detailed level, reproducing high-level functionality and exploring the frontiers of neural circuitry. However, mapping large neural architectures to conventional computers rapidly becomes infeasible due to the fundamental computational differences limiting the size and complexity of studies. Large computer clusters are necessary to conduct detailed, spike-based simulations for small fractions of the brain over the course of only a few seconds (Markram, 2006). Simulations on more abstract levels can capture larger areas, but cannot account for emergent properties on a microscopic single-cell scale. These limitations are overcome by highly-parallel *neuromorphic* hardware systems that physically emulate neural building blocks rather than simulating them. Due to the inherent parallelism, these systems can be scaled towards large neural network emulations without slowing down.

The Hybrid Multi-Scale Facility (HMF) developed within the BrainScaleS project is build upon a highly configurable, large-scale neuromorphic hardware platform. Neural networks are realized energy efficiently in a mixed-signal fashion, where spike communication is implemented by a high bandwidth, asynchronous, digital interconnection fabric, while neurons are implemented as analog circuitry. Due to shorter intrinsic time constants, these neurons operate around 10^3 to 10^5 times faster than biological ones. Therefore, long-term experiments can be conducted in a short period of time. Notably, the energy required per synaptic transmission is about 6 to 10 orders of magnitude less than what is needed for similar simulations on traditional computer clusters (Markram and Meier, 2012). The HMF will provide access to emulations with up to 10^6 neurons until the end of the BrainScaleS project. A second installment to be realized within the first 18 month of the Human Brain Project is going to provide up to 10^9 neurons (HBP SP9, 2014).

State of the art neuromorphic computing shows many similarities to the advent of both, computers and Field Programmable Gate Arrays (FPGAs). Initially, those systems have been mere subjects to research. The scope was limited by the availability of basic tools for their configuration and operation. Over time, operating systems, compilers and improved user interfaces revolutionized the way people interact with computers. Similarly, advanced design tools have been essential for FPGAs to become the de facto standard for low volume hardware and rapid prototype development.

Setting up large networks on the HMF on a low level is a challenging and error prone task even for hardware experts due to its complexity and size. Following the example of computers and FPGAs, it is important to establish an automated mapping of abstract neural network descriptions to hardware specific configurations that enables novices and experts alike to efficiently operate the system. Moreover, project partners and external users who want to exploit the unique properties of the HMF require remote access. This thesis introduces a novel software infrastructure providing both. After outlining the software design principles and implementation, several benchmarks are presented that measure workflow and mapping performance characteristics including early hardware results. Finally, a spike-based liquid state machine (Maass et al., 2002; Jaeger, 2001) implementation is demonstrated on a smaller chip-based system.

Structure of the Thesis

This thesis is structured into nine chapters followed by a comprehensive discussion and outlook. The nine chapters are briefly outlined below.

1. **The Neuromorphic Platform:** In this chapter, all major HMF components are introduced, including the HICANN microchip. A special focus is put on the topology and the large configuration space, which build the foundation for subsequent chapters.
2. **A Scalable Workflow for Neuromorphic Computing:** The accelerated, large-scale HMF can no longer be considered a single-user system due to its complexity and vast computational resources. To increase utilization, its resources have to be shared among many users, much like computer clusters. Here, the design and a prototype implementation of such a multi-user setup are presented.
3. **Neural Network Description:** Starting with the introduction of PyNN, the HMF high-level user interface is presented. PyHMF is a PyNN-compatible, high performance and topology preserving `C++` implementation with a thin auto-generated `Python` wrapper on top.
4. **System-Level Software:** This chapter outlines a redesign of the low-level software, which provides unified access to the HMF.
5. **Mapping Neural Networks to Hardware Specific Configurations:** This chapter introduces a modular feedforward mapping strategy, which enables an efficient translation of abstract neural networks into hardware configurations.
6. **A Scalable Implementation of a Feedforward Wafer Mapping:** An actual implementation of the above mentioned feedforward mapping strategy is presented. Scalability towards large networks is achieved by means of efficient algorithms and shared-memory parallelization.
7. **Handling Hardware Defects:** The HMF is explicitly designed to cope with fabrication defects and analog variations. A dedicated defect framework manages defects, which can then be used to implement workarounds during the mapping step. Moreover, the measurement of synapse defects for a HICANN are presented.

8. **Workflow and Mapping Benchmarks:** This chapter explores the suitability of the HMF and the new mapping for a wide range of network topologies, which includes an early attempt of mapping a chain-model to hardware. A comprehensive study of homogeneous random networks is used to compare the novel workflow with its predecessor. The chapter closes with a comprehensive discussion on neural network topologies suitable for the HMF.

9. **Spike-Based Classification with Accelerated Neuromorphic Hardware:** A tempotron-based liquid state machine has been implemented on an accelerated neuromorphic hardware system. Therefore, a hardware adaptation of the tempotron readout is derived, suitable for conductance-based synapse models. The setup is then trained upon three different tasks including handwritten digit recognition.

1 The Neuromorphic Platform

The Hybrid Multi-Scale Facility (HMF) is a platform for large-scale neural network simulations that combines the flexibility of conventional cluster computing with the efficiency of neuromorphic hardware systems. The wafer-scale system is the accelerated neuromorphic core and consists of uncut wafers of 384 HICANN chips with 512 neuron circuits each.

This chapter introduces all major components of the HMF on an abstract level. Initially, a brief overview of the wafer-scale system is given, then the individual components are explained following a bottom-up approach, starting with the HICANN microchip and moving towards the inter-chip and host communication networks. A special focus is put on the functional aspects and configurability of the system.

1.1 The BrainScaleS Project

The BrainScaleS project (BSP) funded by the European Union started in 2011 and is running over 4 years (BrainScaleS, 2014). Researchers from 19 groups across ten different European countries set out to deepen our understanding of the brain at different spatial and temporal scales. Scales range from in-vivo single-cell measurements on a sub-millisecond level to large-scale network simulations with millions of neurons and long-term learning (Kunkel et al., 2012; Nessler et al., 2013; Pozzorini et al., 2013). The resources necessary to computationally study brain structure are provided by computer clusters (Kunkel et al., 2012; Djurfeldt et al., 2008), many core systems (Furber et al., 2012) and mixed-signal neuromorphic hardware (HBP SP9, 2014). The latter is the central issue of this dissertation.

1.2 The Hybrid Multi-Scale Facility

The HMF combines the flexibility provided by conventional cluster computing with the efficiency and performance of accelerated, mixed-signal neuromorphic hardware systems (HBP SP9, 2014). This hybrid approach allows scientists to study networks on the most suitable substrate. A long-term goal is to connect different aspects of a model running on different substrates in a closed loop, e.g., an analog emulation controlling a virtual robot that is simulated on the cluster. A first prototype installation is shown in Figure 1.1.

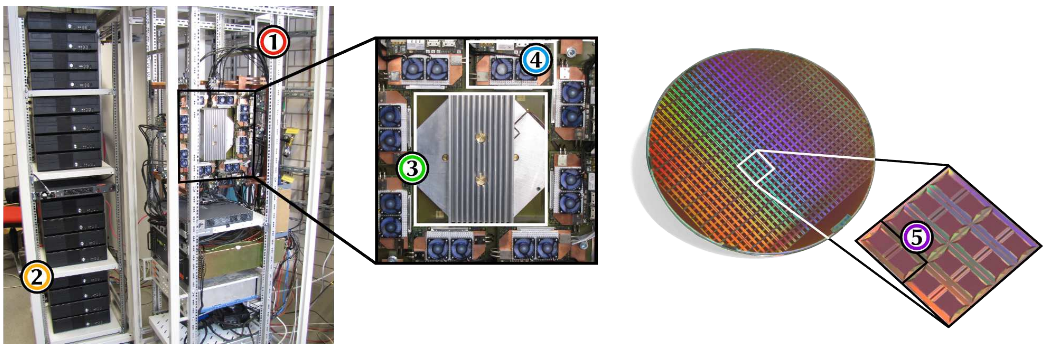


Figure 1.1: The first HMF prototype installation at the Kirchoff-Institute for Physics in Heidelberg. The rack with the wafer system is in the front **(1)** and a 16 node commodity cluster in the back **(2)**. The actual wafer is installed under the aluminum heat sink **(3)**. Off-wafer communication is established via 12 FPGA communication modules **(4)**. A wafer consists of 384 interconnected HICANN chips **(5)**.

1.3 The Wafer-Scale System

The wafer-scale system is the neuromorphic core of the HMF: an accelerated, large-scale, configurable hardware system developed at the Kirchoff-Institute in Heidelberg and at the TU Dresden. The system is composed of multiple wafer modules. Every wafer implements 384 HICANN chips with up to 512 neurons and 114 688 synapses each. After fabrication, wafers are left intact instead of being cut for the production of single chips. This so-called wafer-scale integration is a unique property of the system, where wafer-wide connectivity is established by means of post-processing, as explained in see Section 1.6. This allows high inter-chip communication bandwidth at low cost and with low power consumption.

1.4 The HICANN Microchip

The mixed-signal High Input Count Analog Neural Network (HICANN) microchip is the basic building block of the wafer system. It has a die size of $5 \times 10\,\mathrm{mm}^2$ and is fabricated in 180 nm CMOS technology. Figure 1.2 shows a micro photograph of the chip. The most prominent features are the two large synapse arrays, covering most of the surface area. They are separated by digital communication structures crossing the chip horizontally in the center and vertically on both sides.

This chapter focuses on the functional aspects of the chip, more detailed technical descriptions are provided by Schemmel et al. (2008, 2010).

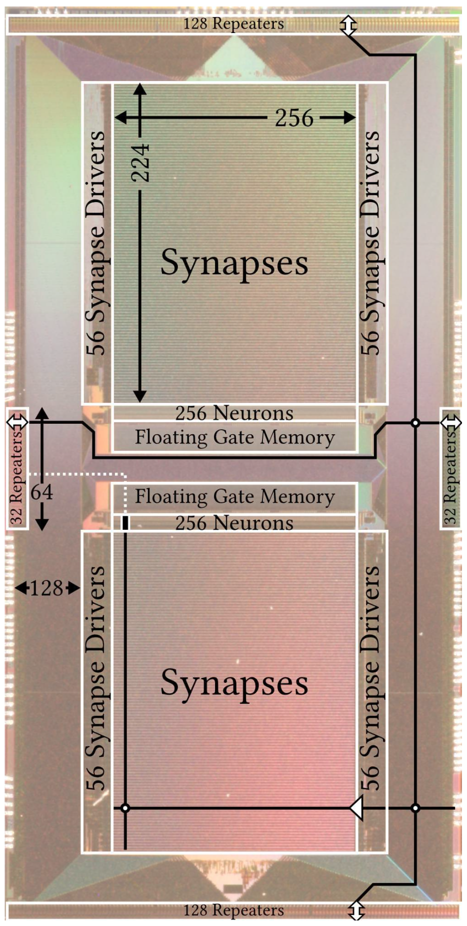


Figure 1.2: A micro photograph of the $5 \times 10\,\mathrm{mm}^2$ HICANN die. Prominently, the synapse arrays cover most of the area. 512 neuron circuits are aligned along the edges of the arrays towards the center of the chip. Layer 1 buses, indicated by black lines, run horizontally through the center and vertically along both sides of the chip. They implement the Layer 1 on-wafer event network for spike communication described in Section 1.5. Repeaters at the chip boundaries restore signal quality, thus enabling long-range connectivity across multiple chips. Events can be relayed from vertical buses to neurons via synapse drivers and synapses, where they finally generate postsynaptic potentials (PSPs) on the analog membrane of the neuron.

1.4.1 Analog Neurons

Every HICANN implements 512 analog neuron circuits (Millner, 2012). Half of which are located on either edge of the synapse arrays towards the center of the chip, as shown in Figure 1.2.

The dynamics of the analog neuron circuits evolve in continuous time and are faster than biological real time due to shorter intrinsic time constants. The speedup can be configured within a range of 10^3 to 10^5, i.e., a speedup of 10^4 means that the emulation of a 1 s experiment takes only 10^{-4} s. This speedup renders the system suitable for learning experiments that would otherwise take long periods of time. The behavior of neurons is characterized by the adaptive exponential integrate-and-fire (AdEx) model (Brette and Gerstner, 2005) with conductance based synapses according to

$$-C_\mathrm{m}\frac{\mathrm{d}\,V_\mathrm{m}}{\mathrm{dt}} = g_\mathrm{l}(V_\mathrm{m}-E_\mathrm{l}) - g_\mathrm{l}\Delta_\mathrm{t}\exp\left(\frac{V_\mathrm{m}-V_\mathrm{t}}{\Delta_\mathrm{t}}\right) + g_\mathrm{e}(t)(V_\mathrm{m}-E_\mathrm{e}) + g_\mathrm{i}(t)(V_\mathrm{m}-E_\mathrm{i}) + \omega \tag{1.1}$$

$$-\tau_w\frac{\mathrm{d}\,w}{\mathrm{dt}} = w - a(V_\mathrm{m}-E_\mathrm{l}) \qquad . \tag{1.2}$$

Where V_m and C_m denote the membrane voltage and capacitance of the neuron, respectively. The latter can be set to either 2.16 pF or 0.16 pF in hardware. E_l, E_e and E_i are reversal potentials for leakage, excitation and inhibition, while g_l, g_e and g_i are the corresponding conductances. The spike initiation potential V_t and the slope factor Δ_t control the non-linear dynamics of the exponential term. The right most term ω in Equation (1.1) contributes an adaption current. The dynamics of the adaptation are described by Equation (1.2). Here, a is an adaption variable with the units of a conductance and τ_w is the adaption time constant. The spiking behavior of the AdEx model is described by

$$\left.\begin{aligned} V_\mathrm{m} &\to V_\mathrm{reset} \\ \omega &\to \omega + b \end{aligned}\right\} \quad \text{if } V_\mathrm{m} \geq \Theta \qquad . \tag{1.3}$$

Whenever the membrane voltage V_m crosses the threshold potential Θ a spike is detected, V_m is set back to V_reset and ω is increased by the constant b. The latter is known as spike-triggered adaption. Given a stimulation with a constant current and for positive b, the excitability is reduced each time the neuron spikes. Actual hardware recordings of characteristic AdEx spike patterns (Naud et al., 2008) are presented in Figure 8.1.

On hardware, neuron dynamics are controlled by 22 parameters, which also include technical parameters to adjust the operational regime (HBP SP9, 2014). These parameters are stored in analog floating-gate cells, as described by Srowig et al. (2007), and can be set individually for each circuit instance, with exception of V_reset. During experiments, the floating gate controller can be repurposed to stimulate a single neuron per chip with a configurable current course.

Neighboring neuron circuits within blocks of 64, 32 from the top and the bottom array, can be combined in order to build neurons with higher input counts. Every connected neuron circuit adds 224 synapses, allowing to built neurons with up to $64 \times 224 = 14\,336$ individual inputs. Technically, neuron membrane capacitors are connected in parallel, therefore adding the capacitances. The interconnection topology is shown in Figure 4.2 item 5. The spike detection mechanism of connected circuits has to be deactivated for all but one. A configurable routing allows to relay the corresponding spike signal to the other circuits to trigger their local voltage reset mechanisms whenever a threshold crossing of the membrane voltage is detected. Setting up the routing correctly is critical because erroneously routed spike signals can trigger a reset cascade, which may damage the chip (Millner, 2011). Future revisions of the chip are planned to implement configurable conductances between membranes to build neurons with multi-compartmental dendrites, as explained by Millner et al. (2012).

1.4.2 Plasticity

The HICANN chip provides two types of plasticity to dynamically modulate synaptic efficacies: short term plasticity (STP) and spike timing dependent plasticity (STDP).

STP models mobility and limited availability of neurotransmitters in biological synapses according to Tsodyks and Markram (1997). The mechanism and its technical implementation is inherited from the Spikey predecessor chip and is described in more detail by Schemmel et al. (2007) and Brüderle (2009).

STDP, on the other hand, enables long-term learning based on spike correlation. Again, the technical implementation is based on Spikey and is described by Schemmel et al. (2006) and Pfeil et al. (2012). More information on the current development and future implementation is provided by Friedmann (2013).

1.5 On-Wafer Event Network

This section introduces the statically switched Layer 1 on-wafer network fabric that implements asynchronous, high-bandwidth communication between chips and has to be configured prior to the experiment. On every HICANN, 64 bus segments run horizontally through the center and 128 vertically on both sides. The bus structure is clearly visible as a distinct H-shape in Figure 1.2. Operated at a clock frequency of 100 MHz, buses provide a bandwidth of 125 MEvents/s resulting in a total throughput of 40 GEvents/s per chip.

Before introducing the network components, a short overview of the event routing is given from source to targets. The individual components and routing options are explained in more detail in the subsequent sections. Whenever the membrane voltage V_m of a neuron crosses the threshold Θ, a digital 6 bit address event is generated. The source address can be set individually for each neuron. Apart from neurons, other spike sources exist, including external FPGAs and on-chip background generators. These events can be combined using the so-called merger tree to increase network utilization, see Figure 1.3. The bottom most mergers relay these events into the Layer 1 network at special synchronous parallel Layer 1 (SPL1) repeaters operating specific horizontal bus segments. Configurable sparse crossbar switches connect horizontal and vertical buses in order to route events across the wafer. Routes always start at a single SPL1 repeater and branch out to target any number of chips. At a target, vertical buses can be connected to synapse drivers via select switches to relay events into the synapse array. A two-stage address decoding scheme selectively activates synapses in order to connect drivers to neurons, where incoming events finally generate PSPs.

Layer 1 events encode spike signals as source addresses (Deiss et al., 1998), thus spike times are implicitly defined by the instant they arrive at the receiver. This also means,

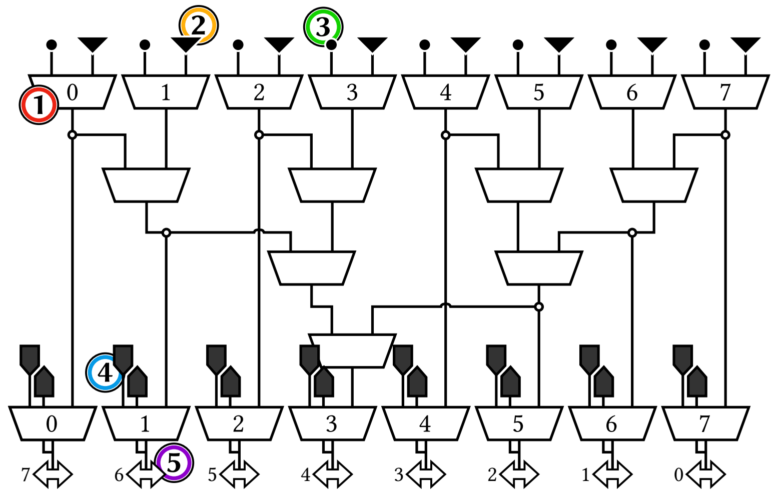


Figure 1.3: An illustration of the HICANN merger tree topology. Events flow from the top to the bottom of the structure. Each merger **(1)** can be configured to forward either events from one of its inputs or merge both with static priority. In the top row, events from up to 64 neurons **(2)** and a background generator **(3)** can be merged. Events from the external FPGA enter the tree at the bottom row of DNC mergers **(4)**. The output of the Digital Network Chip (DNC) mergers interfaces the Layer 1 network via special SPL1 repeaters **(5)** connected to every 8th horizontal bus.

there are no programmable on-wafer delays. A short, but fixed one is given by the physical transmission delay. It takes in the order of a few milliseconds biological time for a spike to round trip from neuron to neuron on a single HICANN and a speedup of 10^4 (Kleider, 2014). In the future, configurable delays can be implemented using the Layer 2 off-wafer network described further below.

1.5.1 Merger Tree

The merger tree, as shown in Figure 1.3, can merge events from different sources into fewer connections to use the Layer 1 routing resources more efficiently. The 6 bit address space provided by the Layer 1 fabric allows to transmit events from up to 64 sources in a single shared connection. In order to distinguish them on the receiving side, sources must have disjoint addresses. The number of spike sources is reduced, for example, whenever neighboring circuits are combined to build larger neurons, i.e., if always 4 circuits are combined only 16 sources exist per neuron block, see Section 1.4.1. Additionally events from the on-chip background generators and external FPGAs can be merged.

The 512 neuron circuits on HICANN are grouped in 8 blocks of 64 neurons, with 32 from the top and bottom synapse array each. In the top most tier, every merger receives events from one block of neurons and one background generator. Background

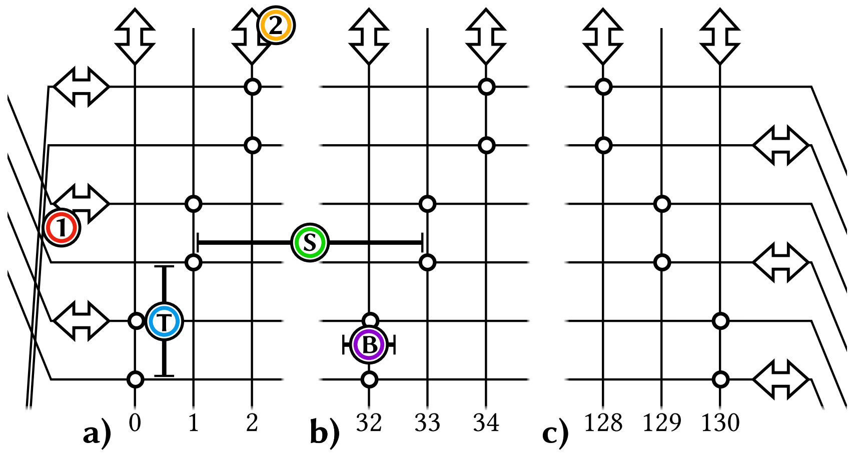


Figure 1.4: Three cutouts of the HICANN crossbar responsible for connecting horizontal and vertical Layer 1 buses. Cutouts **a)** and **b)** illustrate the topology found on the left side of chip and **c)** on the right. The switch layout is mirrored along a vertical axis through the center of the chip. Horizontal repeaters **(1)** refresh signals for every odd bus address on the left and every even address on the right. Similarly, vertical repeaters **(2)** refresh signals on every other bus on either the top or bottom of the chip. The sparse matrix can be characterized by three parameters: sparseness **(S)**, offset **(T)** and block width **(B)**. The respective parameters for the HICANN crossbar are $S = 32, T = \pm 2, B = 1$, with $+$ and $-$ for the left and right side, respectively.

generators can be configured to generate either random Poisson spikes at a given rate or elicit periodic events. They are typically used to lock repeater delay-locked loops (DLLs) and are not available for model emulation as explained in Section 1.5.5. Then, 23 mergers are connected in five subsequent tiers according to Figure 1.3. Any merger instance can be configured to forward events from either its left or right input or merge both. In the latter case, the left input is statically prioritized. At the bottom most tier, so-called DNC mergers interface external DNCs (Scholze et al., 2011a,b) that are in turn connected to FPGAs. DNC mergers can therefore combine chip-local events with events from external FPGA spike sources. Furthermore, events can be routed off-wafer to DNCs and FPGAs, where they are timestamped and recorded, respectively. Finally, DNC mergers relay events into the Layer 1 network via special SPL1 repeaters located on every eighth horizontal bus, which corresponds to the bus indices 6, 14, 22, 30, 38, 46, 54, 62.

1.5.2 Buses, Crossbars and Repeaters

Connectivity between HICANNs is established via an orthogonal grid of 64 horizontal and 256 vertical buses per chip. This bus topology is clearly visible on the micro photograph in Figure 1.2. Horizontal buses run along the center of the chip, while vertical buses run along both sides.

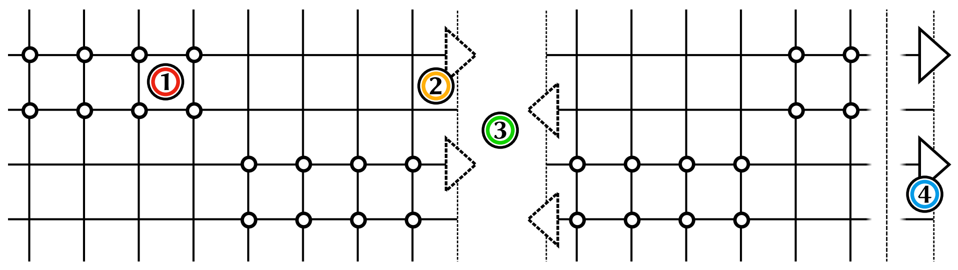


Figure 1.5: An illustration of the HICANN select switches **(1)**, which connect vertical Layer 1 buses and synapse drivers **(2)**. Synapse drivers can relay events into the synapse array **(3)** from both sides. Switches in every other row connect to synapse drivers on adjacent HICANN chips **(4)**. The select switch topology is characterized by $S = 16$, $T = \mp 2$ and $B = 4$, as defined in Figure 1.4.

Long range communication requires repeaters to refresh events at least once every chip boundary crossing (Schemmel et al., 2008). Repeaters exist for both, vertical and horizontal buses. They are located in alternating order at the top or bottom and left or right edge of the chip, respectively, according to Figure 1.4.

Every 8th horizontal repeater is an SPL1 repeater. These special repeaters can also relay input from the chip-local merger tree as explained in Section 1.5.1. Moreover, horizontal and vertical bus indices are shifted by two at the chip boundaries, which is referred to as bus swap. Thus, SPL1 repeaters on neighboring chips are not directly connected. Adjacent chips can therefore insert events into the Layer 1 network without interfering with each other, making the event routing more flexible.

Crossbars on the left and right side of each chip establish connectivity between horizontal and vertical bus segments. Their switch configuration is sparse to limit capacitive load on the one hand and use the remaining chip area for the interleaving of digital logic on the other (Grübl, 2014a). Consequently, not any two orthogonal buses can be connected. Figure 1.4 depicts the sparse matrix pattern of the HICANN crossbar, which is characterized by three parameters: sparseness S, offset T and block width B. A switch exists for vertical bus x and horizontal bus y if

$$E(x, y) = \left(x + B \left\lfloor \frac{y}{T} \right\rfloor\right) \bmod S < B \tag{1.4}$$

evaluates as true.

1.5.3 Select Switches and Synapse Drivers

Whenever a route reaches a chip containing target neurons, events have to be relayed into the synapse arrays. For this purpose, vertical buses can be connected horizontally to synapse drivers via select switches as shown in Figure 1.5. The topology allows to connect each vertical bus to 14 drivers on the same and 14 drivers on the adjacent chip.

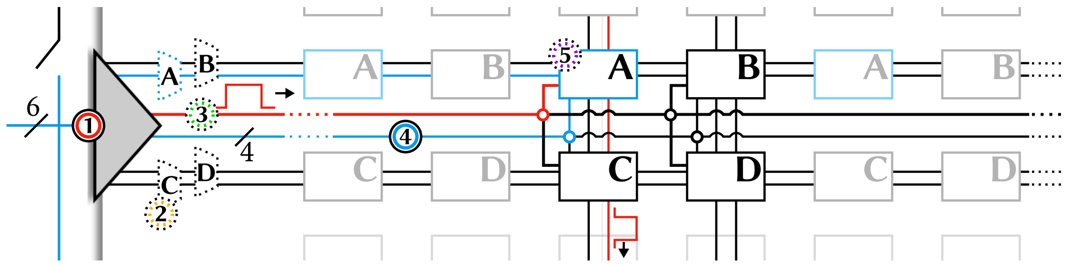


Figure 1.6: The two-stage driver and synapse address decoding scheme. Active signal and address buses are colored in red and blue, respectively. Layer 1 events (6 bit) enter the synapse array via synapse drivers **(1)**. They can receive input either directly from a vertical bus or an adjacent driver. Local 2 bit decoders **(2)** control strobe lines to selectively activate synapses according to the A, B, C, D pattern. Additionally, drivers generate a rectangular voltage pulse **(3)** that is sent horizontally into the synapse array together with the remaining 4 LSB of the address **(4)**. Active synapses **(5)** connect horizontal and vertical lines whenever the address and a local 4 bit decoder value match.

Half of the drivers are located in the top synapse array, the other half in the bottom. Beyond direct Layer 1 input, adjacent drivers can be connected to share common input. This feature is explained in more detail in the subsequent section.

1.5.4 Synapse Arrays

On HICANN, 224 drivers, 56 on each side of each synapse array (see Figure 1.2), can relay events from vertical Layer 1 buses into the synapse arrays. Every driver operates two rows of 256 synapses each and receives input either directly from a vertical bus or shares a common input with adjacent drivers. The latter yields less capacitive load compared to relaying events directly from vertical buses to several individual drivers. High load deteriorates signal quality, as explained in Section 1.5.5. Thus, sharing inputs allows to connect more synapse drivers and therefore map a larger address space according to the address decoding scheme explained in the following.

In order to route the 6 bit events from the up to 64 sources, synapse drivers and synapses implement a two-stage address decoding scheme, which is illustrated in Figure 1.6. This scheme provides a trade-off between flexibility and area covered by the synapse in order to implement a larger number of synapses per chip.

Whenever an event arrives at the synapse driver, the two most significant bits (MSBs) of the 6 bit address are compared by four local decoders. They control strobe signals to selectively activate synapses in the top and bottom row as well as odd and even neuron columns, resulting in the A, B, C, D pattern in Figure 1.6. Only active synapses are vertically connected to neuron circuits and can therefore relay spike signals. If any of the decoder values matches, the synapse driver generates a rectangular voltage pulse and sends it horizontally into the synapse array alongside the remaining four address bits. The integral of the voltage pulse determines the synaptic base efficacy. The pulse amplitude is a fixed parameter controlled per driver, whereas the pulse length is modulated dynamically to implement STP, see Section 1.4.2.

Active synapses decode the remaining least significant bits (LSBs) of the address and compare them against a configurable local value. If the address matches, row and column are connected and a current pulse is generated, which is sent vertically towards the neuron circuit in the corresponding column via one of two collector lines. The amplitude of the current pulse is controlled by two factors: the integral of the incoming voltage pulse and a local 4 bit digital synapse weight. Note that the collector line connectivity is configured per synapse row and therefore shared by synapses A and B as well as C and D.

Every neuron has two synaptic input circuits, one per collector line. They translate incoming current pulses from any of the 224 synapses within their column into exponential conductance courses that connect the neuron to a configurable reversal potential (HBP SP9, 2014). Depending on the input configuration, the generated PSPs can either be excitatory or inhibitory.

1.5.5 Transmission Reliability

The Layer 1 network is merely a link layer according to the OSI layer model (ISO/IEC 7498-1:1994, 1994). The unidirectional communication is inherently unreliable because transmissions cannot be acknowledged by the receiver. Therefore, no transport layer protects events from being lost. Instead, the network topology and parameters have to be chosen carefully in order to optimize transmission reliability.

Firstly, repeaters and synapse drivers require periodic address 0 events, in order to lock local DLLs. Hence, background generators are set up to provide the necessary activity. These events carry no information and are usually ignored on the receiving side by setting the decoders accordingly. Empirically, rates around 3 Hz in the biological time domain (BTD) are sufficient to retain an established lock (Koke, 2014), however, optimal strategies to acquire the initial lock are under investigation, as of May 2014.

Furthermore, a set of analog parameters (V_{ccas}, V_{OH} and V_{OL}) determines transmission properties like power consumption, reliability and delay. Optimal values are subject to calibration. However, within the scope of this thesis it is assumed that the default values work for most experiments. A listing of all corresponding parameters and their default values can be found in Appendix A.2.

Finally, the capacitive load acceptable between repeaters is limited. Connecting components increases the wire capacitance and therefore RC time constants, which deteriorates the digital signal quality. For every Layer 1 bus segment no more than one crossbar and select switch per chip must be set. Similarly, only a limited number of synapse drivers can be connected to vertical buses. This concerns both, primary drivers, which are connected directly to a vertical bus, and drivers connected via neighbors. However, the latter produces less load. Early measurements by Kononov (2011) indicated that for Layer 1 clock frequencies of 100 MHz to 150 MHz up to

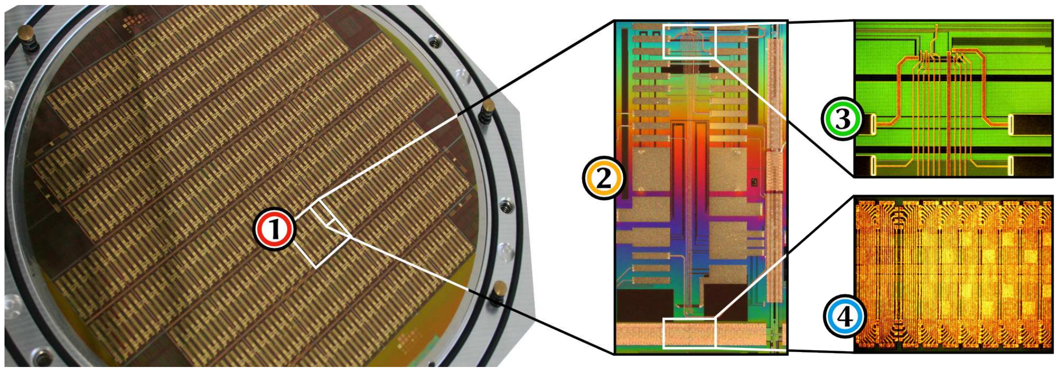


Figure 1.7: Illustration of the post-processing metal layer deposited as part of the wafer-scale integration. An uncut 20 cm wafer contains 48 reticles with 8 HICANN chips each **(1)**. Every post-processed chip **(2)** provides stripe connector pads **(3)** to implement off-wafer connections to the main printed circuit board (PCB) and FPGAs. Deposited vertical **(4)** and horizontal interconnects realize Layer 1 on-wafer connectivity across reticle boundaries.

4 connected neighboring synapse drivers worked reliably in more than 97 % of the measurements conducted using the first version of the HICANN chip.

1.6 Wafer-Scale Integration

One of the major challenges for implementing accelerated, large-scale neural networks is to provide the necessary connectivity. In biology, neurons typically receive input from 10^4 to 10^5 other neurons (Drachman, 2005). Assuming a speedup factor of 10^4 and an average biological firing rate of 10 Hz this corresponds to an input bandwidth requirement of 1 to 10 GEvents/s per neuron. The cumulative input bandwidth per chip is even higher because different neurons likely receive different input. Wafer-scale integration is the key technique, used for the HMF, to implement the high-bandwidth Layer 1 as well as the off-wafer Layer 2 network described in Section 1.5 and Section 1.7, respectively. Photos of the technical realization are shown in Figure 1.7.

After fabrication, the wafer containing the 384 HICANNs is left intact. A subsequent post-processing deposits metal inter-connectors on the surface, see Figure 1.7 item 4. Therefore, connectivity can be established beyond reticle boundaries, which denotes the connected area captured by a single lithography mask. The high integration density reduces wire impedance leading to more energy efficient systems. Furthermore, off-wafer communication can be established vertically over the whole wafer area by means of deposited connector pads, see Figure 1.7 item 3. These pads are connected to the main PCB of the wafer system via stripe connectors (Zoglauer, 2009).

On the downside, wafer-scale integration causes random defects, resulting from impurities in the manufacturing process, to remain in the system (Stapper et al., 1983). In normal production, a wafer is cut into individual chips, their quality assessed and unsuitable ones dismissed. However, this not an option for wafer-scale systems, instead they need to be tolerant against such defects. The HMF allows to workaround most localized defects via appropriate configurations. For example, Layer 1 connections that would normally pass through a defect region can be routed differently. As a measure of last resort, whole reticles can be cut from power to avert e.g., shorts that otherwise impede the operation of the remaining reticles. Efficiently finding configurations that workaround defects and handle imperfections is important for experiment reproducibility and one of the central issues of this thesis.

1.7 Off-Wafer Communication

Beyond recurrent connectivity, neural network experiments require external spike stimulation, the ability to record spike responses and host communication to operate the system. The Layer 2 off-wafer network provides all necessary means. The infrastructure is provided by a hierarchy of host computers connected to FPGAs over ethernet, which are in turn connected to custom DNCs. Every wafer module hosts 12 FPGAs, with four DNCs each. A single DNC is responsible for managing the communication for eight HICANN chips. The Layer 1 and Layer 2 networks interface at the bottom layer of the merger tree as explained in Section 1.5.1. For clock frequencies of 100 MHz, the bandwidth between a DNC and a single HICANN is 25 MEvents/s.

1.7.1 Real-Time Spike Handling

External stimulation and spike recording require high-bandwidth communication to cope with the 10^3 to 10^5 speedup factor of the analog neurons. Therefore, spike input trains are prepared in DRAM on an FPGA board prior to the experiment to achieve high throughput and avoid timing jitter caused by packet delivery over ethernet or process scheduling on the host. Each FPGA can store up to 2.5×10^8 events in memory (Grübl, 2014b). Real-time host communication is possible, but with lower throughput and no deterministic timing. Similarly, spike output is recorded to DRAM on the FPGA board and accessed by host computers after the experiment.

Future FPGA firmware versions are going to provide programmable spike delays via the Layer 2 network. Spikes are therefore routed off wafer, buffered in the FPGA and reinserted into the Layer 1 network after a configurable delay has been elapsed.

1.7.2 Host Communication

Communication between wafer system and the conventional computer cluster is necessary for the initial hardware setup, the subsequent readout of results and interactive experiments. Wafers are therefore connected to an top-of-rack ethernet switch via 12 FPGAs with 2×1 Gbit uplinks each. A 10 Gbit ethernet backbone switch aggregates connections from the host computers and the wafer switches. This simple but efficient fat tree topology (Leiserson, 1985) allows communication between any host and FPGA with low latency and high bandwidth.

An automatic-repeat-request transport layer protocol (Philipp, 2008; Schilling, 2010) similar to PGM (Speakman et al., 2001) provides reliable communication between FPGAs, DNCs and HICANNs on top of the link layer. Packets lost in transfer are resent after a configurable timeout.

Notably, close-loop experiments and system operation have different performance requirements. Experiment setup and result collection demand high bandwidth, whereas closed-loop experiments require low-latency communication to deliver spike responses within short periods of time. For example, 1 ms of biological time corresponds to 0.1 µs in the hardware time domain (HTD) for a speedup of 10^4. To realize biologically relevant delays of 10 ms to 100 ms, the system has to respond in less than 10 µs, which is challenging in gigabit ethernet terms (Loeser and Haertig, 2004).

1.8 Future Deployments

Within the BSP a system of six wafer modules is built until the end of 2014. Beyond that, another 20 wafer system is built during the first 18 month of the Human Brain Project (HBP) (HBP SP9, 2014). This system is going to provide access to almost 4×10^6 accelerated neurons in a system that has an expected power consumption of 30 kW.

An updated 65 nm HICANN successor, ready for production use, is planned for month 30 of the HBP. The new system is going to further improve power efficiency, provide higher bandwidths and implement advanced plasticity mechanisms including a general purpose plasticity processor (Friedmann, 2013).

2 A Scalable Workflow for Neuromorphic Computing

The HMF is a powerful but complex computational platform. Development and operation mandate constant monitoring by hardware experts. In order to make its resources available for a broader range of users, a scalable, multi-user workflow is required. Furthermore, batch processing enables the system to work more economically. This chapter proposes a workflow design for the platform and presents a prototype implementation thereof.

Firstly, the existing HMF workflow is introduced, followed by the workflow redesign including brief discussions on all major components. Subsequently, an experiment broker prototype is presented, which is responsible for accounting, dispatching jobs and tying together the other workflow components. The chapter closes with a measurement of experiment throughput for the prototype implementation.

The workflow and prototype implementation presented in this chapter have been developed by Eric Müller and the author.

2.1 Existing Workflow

The previously existing HMF workflow has been developed as part of the Fast Analog Computing with Emergent Transient States (FACETS) project. It started as a fork of the Spikey workflow, which has been the first neuromorphic hardware system to feature an abstract PyNN interface (see Section 3.1; Davison et al., 2008; Brüderle et al., 2009). Though, the underlying hardware is different and the translation of neural networks into hardware configurations had to be redesigned in order to cope with the different topology and increased complexity of the system. However, the simple single-user workflow design as sketched in Figure 2.1 remained unchanged. A more detailed introduction is given by Brüderle et al. (2011).

The existing workflow mainly consists of three steps. Firstly, an abstract representation of the PyNN neural network is set up. Secondly, a corresponding hardware configuration is derived from the abstract representation in a process referred to as *mapping* throughout this thesis. The mapping has to find configurations that closely resemble the intended network models within the constraints of the hardware system. Finally, the hardware is configured, the experiment conducted and the results are retuned to the user in PyNN format. The workflow runs in a single process on a single computer and requires privileged, direct access to the wafer system. Although this

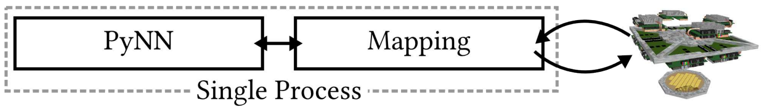


Figure 2.1: The existing HMF single user, single process workflow. PyNN directly controls the mapping process, which is responsible for both, translation of the abstract neural network into a corresponding hardware configuration and experiment execution. The workflow requires immediate and privileged access to the hardware platform.

mode of operation is also supported by the redesign, a more elaborate workflow has to be established to utilize available resources to full capacity and allow multiple users to access the HMF in parallel.

2.2 Workflow Redesign

The HMF is a complex platform that requires constant monitoring by hardware experts, similar to conventional computer clusters. This mandatory overhead suggests the use of batch processing to provide simple access for many users and increase resource utilization. In fact, multi-user remote access for project partners is requested by the HBP. The original workflow has never been designed for this use case. Consequently, a redesign became necessary to deal with the up-to-date requirements. The redesigned workflow is presented in Figure 2.2.

Users only need the PyHMF client installed on their local computers to run PyNN scripts. The client transparently handles model construction, serialization and communication with an experiment broker at the HMF site. There, the model is translated into a hardware configuration, which in-turn is scheduled for execution on either the ESS (see Section 2.2.5) or the wafer-scale system. After completing the run, the experiment server collects the results from the host computers and translates them into the biological model domain. Finally, the results are relayed via the experiment broker back to the user. Note that the new approach is PyNN-compatible and allows users to easily switch between different simulation backends, the only difference is that experiments are conducted remotely in the HMF case.

2.2.1 Batch Processing

Non-interactive batch processing is used for conventional computer clusters to increase resource utilization (Kubo, 1999). Analogously, batch processing allows to used conventional and neuromorphic resources more economically. Figure 2.3 shows how multiple mapping jobs from different users or from parameter sweeps can be processes by the HMF cluster in parallel using available batch systems. For example, generating hardware configurations is typically slow compared to the actual hardware experiment

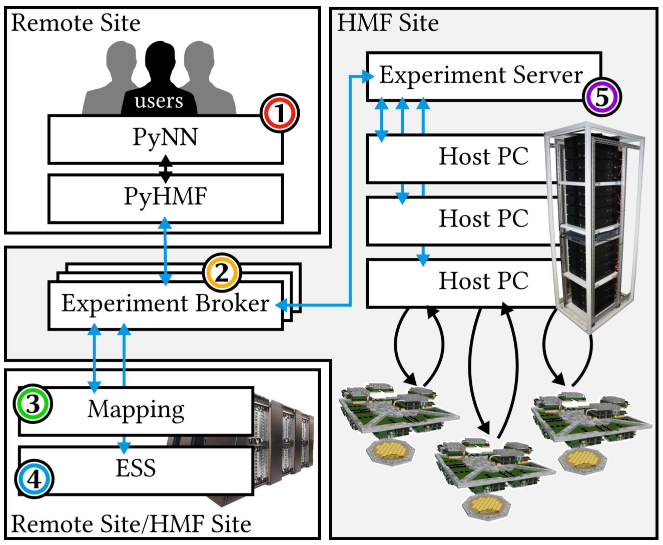


Figure 2.2: The redesigned HMF workflow. Users **(1)** execute PyNN experiments using the PyHMF client, which serializes the network model and sends it to an experiment broker at the HMF site **(2)**. A mapping of the network model to a hardware configuration is subsequently scheduled on the cluster **(3)**. When the broker receives the mapped configuration and the requested resources are available, the experiment is either simulated on the ESS **(4)** or emulated on the wafer system **(5)**. Afterwards, the results are translated into the biological model domain by the experiment server and sent back to the user. Note that inter process communication (IPC) between components is indicated by blue arrows.

due to the high analog speedup factor. Thus, scheduling available configurations from concurrent mappings in a second batch queue increases hardware utilization.

2.2.2 High-Level User Interface

The computational resources of the HMF can be accessed by authorized users via the PyHMF client. It provides a PyNN-compatible (PyNN, 2014) application programming interface (API) to construct neural network representations that can be serialized and sent to the remote experiment broker. Chapter 3 discusses both, PyNN and PyHMF, in more detail.

The redesign uses native PyNN API to dispatch jobs rather than additional interfaces or remote shell access. This design may enable future PyHMF versions to capture parameter sweeps and therefore map this series of experiments more efficiently than unrelated experiments dispatched as individual jobs. Nonetheless, remote shell access can be provided because PyHMF is equally well suited for local access.

The client-server architecture in the redesign further simplifies the distribution and maintenance of software for users and developers alike. Compared to the previous approach, users need only a small fraction of the software stack, namely the hard-

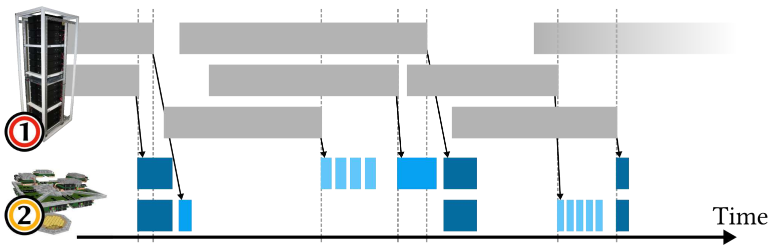


Figure 2.3: Experiments on the accelerated neuromorphic system typically take little time due to the high speedup factor. Preparing configurations for multiple experiments and users in parallel helps to utilize the system to its capacity. Both, conventional **(1)** and neuromorphic **(2)** resources are scheduled following a batch approach. Parameter sweeps, single and multi-wafer experiments are shown in light, mid and dark blue, respectively.

ware independent PyHMF client. Server-side software and hardware deployments are maintained independently by the developers and hardware experts. Moreover, the same client can be used to transparently access different hardware backends or future hardware revisions. Currently, the client supports access to the wafer system and the ESS.

2.2.3 Experiment Broker

The experiment broker is the central communication hub, which is responsible for connecting all endpoints and progressing experiments.

When the broker receives a PyHMF experiment from an authorized user, it creates a local job, spawns a mapping process and relays the network description. The broker keeps track of pending requests. Experiments are expected to take in the order of minutes to hours depending on the load of the HMF. It is therefore necessary to cope with repeated client disconnects and reconnects. After the mapping is completed, the configuration is forwarded to the requested backend i.e., an HMF experiment server or the Executable System Specification (ESS). The broker finally relays experiment results back to the user.

Several brokers can operate in parallel to distribute the work, either at different public addresses or behind load balancers (Bowman-Amuah, 2003). Distributing the load onto several brokers also improves availability by avoiding single points of failure (Piedad and Hawkins, 2001).

2.2.4 The Mapping

The mapping is responsible for translating abstract neural network descriptions into corresponding hardware configurations. The new workflow reassigns many of the tasks previously part of the mapping in order to modularize the system. Flow control, in particular, is now a responsibility of the experiment broker and experiment server.

Still, the mapping task is algorithmically the most complex problem and key for high hardware utilization. The mapping step is a central topic of this thesis and discussed in Chapters 5 to 8.

2.2.5 Executable System Specification

The ESS is a high-level simulation of the wafer-system that has been developed by Vogginger (2010). Mapped configurations can be simulated including bandwidth limitations caused by the Layer 1 network in order to prepare network models for the emulation on the HMF or to study the influence of mapping distortions on the network dynamics (Brüderle et al., 2011). As of May 2014, the ESS has been in the process of being integrated into to the new workflow (Pape, 2013) via the low-level interfaces presented in Chapter 4. On completion, the experiment broker can redirect mapped configurations transparently to either actual hardware or the ESS. The underlying mechanism and interfaces are explained in Section 4.2.4. Apart from preparing network models, the ESS is a valuable tool for workflow verification and continuous integration (Duvall et al., 2007).

2.2.6 Experiment Server

The experiment server is responsible for coordinating the host computers and the hardware access as well as conducting the transformation of results from hardware into biological model domain.

Configurations are mapped for particular hardware instances to account for their individual characteristics, like analog variations. Thus, the broker has to relay configurations and the information necessary for the transformation of results to the appropriate experiment server. Next, the experiment server schedules the experiment for execution on the target hardware instance. As soon as a window for execution opens, the configuration is distributed to the responsible host computers, which are subsequently instructed to carry out the experiment. Afterwards, spike train and membrane recordings are collected from the wafer system. In a final step, the results are translated into the biological model domain. This means, recording times are scaled according to the hardware speedup factor and voltages are translated according to calibration data.

At the time of writing, the experiment server has not yet been implemented. Instead, experiments are triggered either by the broker or mapping process. However, the necessary software for translating the results is available and has been used, e.g., for the Hellfire chain experiment presented in Section 8.3.

2.3 Experiment Broker Prototype

A prototype implementation of the experiment broker, as described in Section 2.2.3 has been developed as part of this thesis. The so-called Ester broker is a multi-threaded

$$\begin{tikzcd}
\text{null} \arrow[r] & \text{new} \arrow[r] \arrow[dr] & \text{running} \arrow[r] \arrow[d] & \text{finished} \arrow[r] \arrow[dl] & \text{completed} \\
 & & \text{error} & &
\end{tikzcd}$$

Figure 2.4: Transition diagram of the simple job state machine in Ester. New jobs are initialized as *null* and transition from left to right. After successful initialization the state is changed to *new*. Jobs in the *running* state are currently executed, *finished* jobs are ready to be sent back to the user and *completed* jobs can be deleted. The job is trapped in an *error* state whenever a failure occurs. Future Ester versions might also consider additional states to support the cancellation and suspension of jobs.

server application written in `C++`. It listens for incoming PyHMF client requests containing network experiments and dispatches mapping jobs accordingly. As soon as a job is completed, the experiment results are sent back to the user. The software presented in this section, has been developed in cooperation with Eric Müller.

2.3.1 Implementation and Design

The client-server communication between PyHMF and the broker is implemented by means of the `RCF` framework for remote procedure calls (Delta V, 2013), which also provides strong SSL-based authentication facilities (Freier et al., 2011). The necessary IPC infrastructure for spawning the mapping process and forwarding the network description to the mapping is implemented on top of MPI (Graham et al., 2006). Payload for the client-broker as well as the broker-mapping communication is marshalled via `boost::serialization` (Ramey, 2004; Husmann, 2012). A simple state machine processes incoming PyHMF jobs following the transition diagram shown in Figure 2.4. Multiple worker threads process queued jobs, advance their states and finally collect completed jobs in order to free up memory.

2.3.2 Experiment Throughput

Ester is designed to receive experiments from multiple users in parallel. The following performance measurement of Ester explores the suitability of `RCF` in combination with `boost::serialization` for the designated use case (Husmann, 2011). Here, the ability to the experiment broker to handle concurrent PyHMF requests is measured neglecting both, mapping and hardware interaction. The computational and software setup described in Appendix A.1 have been used to conduct the measurement.

Experiment throughput is measured as function of the number of concurrent clients requests and parallel server threads for different experiment sizes. Homogeneous random networks, as presented in Section 8.4, are used as client reference experiments that are sent to the broker. The memory consumption, for this kind of network, depends quadratically on the number of neurons N independent of the connection probability because PyHMF stores synapses in connection matrices, see Section 3.2.2. The connection probability has arbitrarily been set to 5 %. Networks instances with 10, 100, 500, 1000, 2000 and 5000 neurons have been measured, resulting in serialized experiment descriptions of 0.003, 0.10, 2.0, 7.8, 31 and 191 MB, respectively.

For the measurements presented, the clients and the broker both run on the same computer, communicating over the Linux loopback network device (Biro et al., 1993). Thus, the effective network bandwidth is very high, in fact, the throughput is only limited by the ability of `RCF` and `boost::serialization` to serialize, handle and deserialize experiments. Note that the broker currently deserializes the complete experiment. Future versions are going to split experiments into an Ester header and the representation of the actual neuron network. The latter can be forwarded efficiently to the mapping process without causing any serialization overhead on the broker.

The measured client request performance is presented in Figure 2.5. Up to 50 PyHMF clients transfer homogeneous random networks of different sizes to the Ester broker, which provides up to 20 threads. The different network sizes have been color coded. The average throughput for different numbers of server threads, but a fixed number of concurrent client connections and vice versa are indicated by solid lines. Note that for larger network sizes and many clients some measurements could not be conducted due to memory limitations.

The throughput for small network sizes of 10, 100 and also 500 neurons is significantly lower compared to the larger networks because they are bound by the serialization and communication overhead. For networks of 10 and 100 neurons, parallelization overhead can be observed on the server-side as a further throughput reduction in the bottom plot in Figure 2.5 towards higher concurrency. The effect saturates for about 4 threads. For network sizes beyond 500 neurons, a throughput of up to 1.6 GB/s is reached. This means, the current implementation is efficient enough to fully utilize the 10 Gbit/s ethernet links used in the HMF cluster.

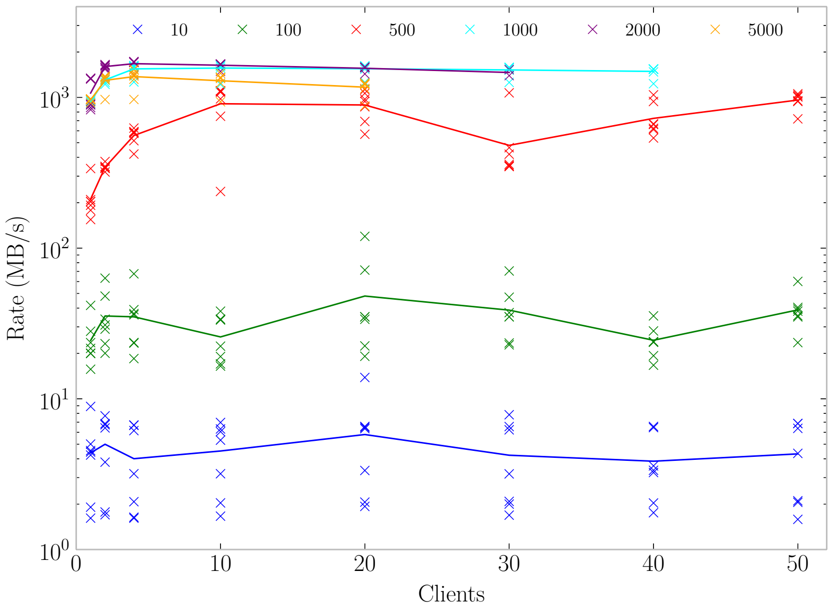
Moreover, the results show an increased throughput for large network instances when using multiple server threads compared to the single-threaded version. This effect already saturates for two server threads because two clients concurrently communicating with the broker via the fast loopback interface fully utilize the quad-core processor of the PC used for the experiment. In production use, the gain per thread can expected to be higher for many users with slow uplinks. In this case, the processor resources are sufficient to handle these lower-bandwidth connections in parallel. The same argument holds for concurrent client requests, increased throughput can be observed for two clients and large networks in the upper plot in Figure 2.5.

Finally, the experiment throughput is reduced for large networks and many concurrent client connections because the broker has to swap memory to disk. This effect is most prominent for the largest network instance with 5000 neurons. Therefore, brokers should monitor their state and reject further incoming jobs whenever they run short of resources to prevent performance regressions in production use and more importantly prevent denial of service (Handley et al., 2006).

2.4 Summary

A novel, scalable multi-user workflow has been presented that provides remote access to the HMF, increases resource utilization and supports highly available deployments. The separation into client and server-side does not only improve scalability, but also simplifies software maintenance for users and developers alike.

Finally, an experiment broker prototype implementation has been presented providing authentication and experiment transmission over IPC. Although still in development, it has been shown to handle concurrent client requests and provide high experiment throughput of up to 1.6 GB/s, which is sufficient to saturate the 10 Gbit/s ethernet links of HMF cluster nodes. Future Ester versions are planned to increase the throughput even further by splitting experiments into an Ester header and the actual network representation, therefore avoiding unnecessary serialization overhead. Missing features and prospective development are discussed in the thesis outlook.



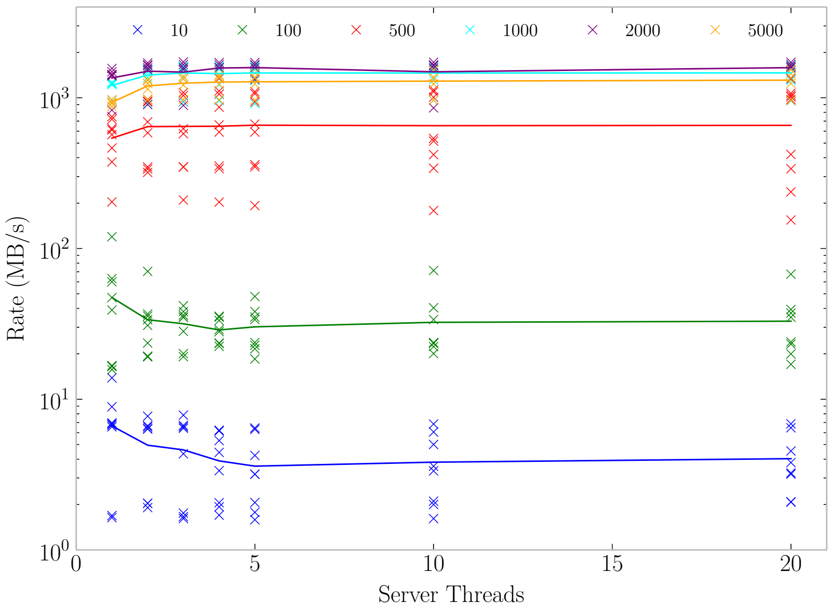


Figure 2.5: PyHMF and Ester throughput measurement for homogeneous random networks. On top, the throughput is shown as a function of clients for different network sizes. Data points of equal color and within the same column have been measured for different numbers of server threads. The throughput typically increases with the number of threads. Below, the same data points are plotted as a function of server threads rather than concurrent clients. Thus, data points of equal color and within the same column correspond to measurements with different client counts. Note that each data point represents the average throughput over many transmissions. Solid lines indicate the average over data points belonging to the same network size and number of clients (top) or server threads (bottom).

3 Neural Network Description

The abstract neural network description language PyNN provides unified access to different simulation backends, including the HMF. This chapter briefly introduces PyNN, explains its benefits and its shortcomings that ultimately led to the development of PyHMF. PyHMF is an efficient and hierarchy preserving implementation of PyNN implemented in `C++`. Code generation provides the necessary means to create PyNN-compliant `Python` bindings. The results in Chapter 8 indicate a performance gain of more than two orders of magnitude compared to standard PyNN, depending on the network topology.

The work presented in this chapter is a shared development effort by Sebastian Billaudelle, Christoph Koke, Eric Müller and the author. The reduction of the PyNN hypergraph into a simpler graph problem has been provided by the author.

3.1 The PyNN Description Language

PyNN is a simulator independent abstract description language for spiking neural networks implemented in `Python` (PyNN, 2014). It provides transparent unified access to different simulation backends (Davison et al., 2008). Experiments described in PyNN can be executed on different supported simulators depending on performance and precision requirements or for cross-simulator verification. A complete API specification of the current stable version 0.7 is given by Davison (2011b). In this chapter PyNN API elements are capitalized to familiarize the reader with their names and to emphasize their connection to specific software implementations rather than their abstract meaning. However, subsequent chapters fall back to the correct English spelling.

Spikey has been the first neuromorphic hardware system to adopt PyNN as its primary high-level interface (Brüderle et al., 2009). Today, PyNN backends also exist for the HMF (Brüderle et al., 2011) and the SpiNNaker system (Galluppi et al., 2010).

3.1.1 Model Hierarchy

Early versions of PyNN provided mainly two high-level building blocks to set up neural networks: *Populations* and *Projections*. Populations are groups of neurons. They can have different parameters, but share a common neuron model. Projections connect Populations with other Populations or themselves. A Connector argument specifies Projection properties, e.g., stochastic or user-defined connectivity.

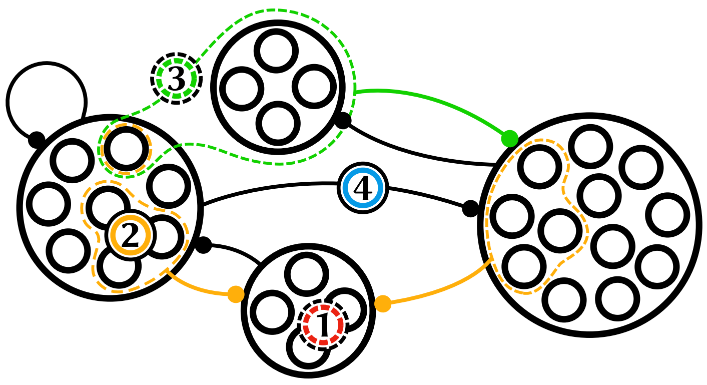


Figure 3.1: The hierarchical interface abstractions provided by PyNN 0.7 and newer. Neurons (small circles) are organized in Populations **(1)**. Each neuron only belongs to a single Population. Additionally, neurons can be organized in PopulationViews **(2)** and Assemblies **(3)**. PopulationViews contain subsets of Populations, whereas Assemblies contain any combination of Populations, PopulationViews and Assemblies. Any aggregation of neurons can be connected via Projections **(4)**, which are internally expanded into connections between individual neurons.

In PyNN, Populations and Projections are merely API abstractions. The frontend instantly expands Projections into individual connections between neurons that are represented by an adjacency list. From a users point of view, this makes little difference. However, the hierarchical information is no longer available for simulation backends. This structural information is valuable for e.g., partitioning and distributing the network to nodes in a computer cluster or chips in a neuromorphic hardware system. An alternative, coarser approach is to represent the network topology as a graph of Populations connected by Projections. In this picture, edges carry the Projection properties and individual connections between neurons can be lazily expanded whenever necessary (Watt and Findlay, 2004).

PyNN 0.7 introduced two new interface concepts: *PopulationViews* and *Assemblies*. PopulationViews are subsets of Populations. Assemblies on the other hand, are supersets of Populations, PopulationViews and Assemblies. The new interface provides users with more flexibility and convenience in terms of network construction. These additions did not require any changes to the internal structure of PyNN, which is still based on adjacency lists for individual neurons. All hierarchical API abstractions provided since version 0.7 are visualized in Figure 3.1.

The introduction of PopulationViews and Assemblies, however, complicates the picture of a coarse Population graph as described earlier. From a topological point of view, the graph becomes a hypergraph because edges involving Assemblies may connect sets of vertices rather than single vertices. Whereas edges involving PopulationViews may connect only fractions of Populations and therefore fractions of vertices.

Turning a graph problem into a hypergraph problem has several downsides. Firstly,

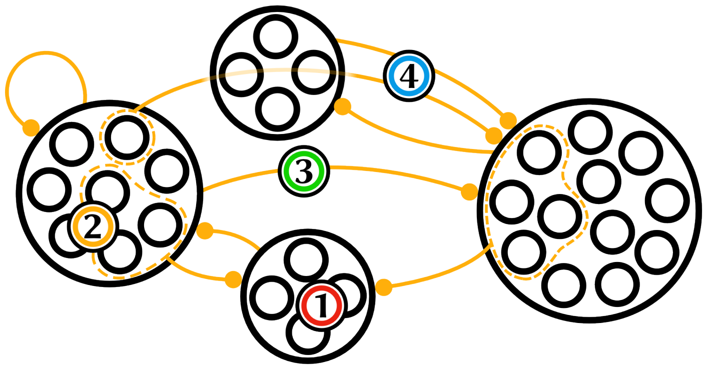


Figure 3.2: The PyHMF reduction of the PyNN hypergraph shown in Figure 3.1. Connectivity between Populations **(1)** and subsets thereof **(2)** is provided uniformly by ProjectionViews **(3)**. They can arbitrarily reference subsets of neurons in the source and target Populations. Note that the green Projection in Figure 3.1 has been split into two distinct ProjectionViews **(4)**.

there are only a few hypergraph software libraries available, which are inherently more complex to use. Secondly, the massive algorithmic toolbox that exists to efficiently handle graph problems is no longer accessible. The following section thus presents a PopulationView and Assembly-compatible strategy for PyHMF to reduce the hypergraph to a simpler graph problem.

3.2 A Hierarchical Network Representation

During the workflow redesign, presented in Chapter 2, the existing PyNN backend had to be reconsidered. The new workflow requires components to communicate via IPC, which mandates a neural network representation that can be serialized. Instead of implementing another backend that translates PyNN connectivity into a native, serializable format, PyHMF has been developed as a more radical approach.

PyHMF is an efficient hierarchy-preserving `C++` library for the description and representation of abstract neural networks. The library provides the necessary IPC facilities for workflow integration and the native `C++` interfaces can seamlessly be used by other workflow components. Having unified, native access to connectivity saves memory by avoiding redundant copies of synapses. This section presents a reduction of PyNN hypergraphs to plain graphs that enables PyHMF to provide a PyNN-compatible interface while representing networks internally as simpler, more efficient graph hierarchies. Code generation is used to automatically wrap these interfaces to PyNN-compliant `Python` API.

3.2.1 Hypergraph Reduction

PopulationViews and Assemblies turn the Population graph into a hypergraph as discussed in Section 3.1.1. This might be more complex than necessary and is algorithmically less accessible than plain graphs. Here, a reduction of the hypergraph to a simpler graph problem is presented.

Considering PopulationViews first, an obvious problem is that edges in the earlier picture may connect subsets of Populations. One approach is to cut the Population into smaller Populations such that each PopulationView has an corresponding isomorphic Population. Two different but overlapping views cut a Population in up to three pieces, one Population for both disjoint sets and one for the intersection of neurons. Furthermore, Projections from and to these PopulationViews have to be divided into multiple sub-Projections. Excessive use of PopulationViews ultimately fragments Populations into individual neurons rendering this approach inefficient.

A second approach is to store the membership of neurons in Projections rather than PopulationViews, thus introducing the concept of *ProjectionViews*. Then, connections between Populations and PopulationViews can be expressed by ProjectionViews alike.

Concerning Assemblies, we reconsider both previous approaches. Here, splitting Projections between Assemblies into smaller ProjectionViews is simpler because no Populations have to be divided. Moving Assembly membership information into connections, on the other hand, can be expensive for large Assemblies. Consequently, Projections involving Assemblies are divided into multiple ProjectionViews, each addressing a subset of connections in the original Projection.

In conclusion, ProjectionViews and the splitting of Projection into multiple ProjectionViews for Assemblies reduces the PyNN hypergraph into a simpler Population-based graph. Figure 3.2 shows the corresponding reduction for the network previously shown in Figure 3.1.

3.2.2 Implementation and Design

PyHMF is implemented as a modular `C++` library with a little over 10.000 lines of code. This is achieved by means of code generation to create the PyNN-compatible `Python` bindings. Furthermore, implementing the network setup natively in `C++` avoids dynamic language overhead (Barany, 2014) and frequent foreign language calls from `Python` into `C` for random number generation (Behnel et al., 2011), which provides a significant speedup over PyNN. The library itself is split into a frontend and a backend layer. The backend layer implements all facilities to build the network representations following the graph hierarchy described earlier. The frontend part has been kept lightweight using opaque pointers (Sutter, 1998) to minimize header dependencies and make them more accessible for code generation. This separation enables the backend part to remain stable, while the frontend evolves alongside future API versions of PyNN. Whenever the frontend layer changes, the `Python` bindings are updated automatically. The native interfaces further allow to directly use the neuron parameters in the hardware calibration libraries (see Section 6.7.1) to provide consistent translations from PyNN parameters into the hardware parameter domain. The parameters have therefore been moved into a separate small library to reduce the software footprint. Communication between PyHMF and Ester is established by means of `RCF` and `boost::serialization`, as explained in Section 2.3.

A rigorous set of tests continuously verifies all components of PyHMF. For example, PyNN and PyHMF Connectors are compared against one another on the individual synapse level. In consequence, several issues in the connectivity setup of the original PyNN package have been discovered, which have been reported and consequently been resolved.

Synapse Representation

PyNN is merely an interface description, it does not specify how connections are represented internally. In practice, the representation of synapses depends on the Connector type. This leads to inconsistent behavior, e.g., weight matrices can be extracted from internal adjacency lists, which causes multiple connections between pairs of neurons to be silently lost. PyHMF on the other hand, consistently stores synapses in an individual connection matrix per Projection. This simplifies serialization, reduces software complexity and provides constant time lookups, however, may cost extra memory. This extra cost has to be payed whenever sparse connectivity is set up between large Populations, e.g., a Projection between two Populations with N and M neurons consumes $\sim N \times M$ memory (Even and Even, 2011). The design decision has been made under the premise that networks for the HMF should be structured and emphasize local connectivity to allow faithful hardware representations thereof, see Chapter 5. Still, worst-case, full-wafer homogeneous random networks with more than 4.5×10^4 neurons have been mapped, as shown in Section 8.4. In fact, mapping homogeneous random networks with connection probabilities of $p \geq 10\,\%$ requires less memory using the new workflow compared to the old one due to implementation and adjacency list overhead. The new workflow is consistently more memory efficient for more structured neural networks, see Figure 8.17.

The communication overhead introduced by sparse matrices is of lesser concern because they can be compressed efficiently (Galli et al., 1998). Future versions are planned to realize synapses lazily to delay the expansion of individual synapses, which can then be realized as part of the mapping process. This is possible whenever user do not access individual synapses prior to the experiment submission.

Batch Flow Integration

The HMF workflow uses native PyNN API to dispatch jobs to the experiment broker, see Section 2.2.2. Handling multiple experiments within a single description may enable future mappings to handle parameter sweeps more efficiently such that e.g., only analog parameters are reset instead of rebuilding the whole connectivity. If, on the other hand, parameter sweeps are dispatched as independent jobs, the workflow cannot capture the relationship between these subsequent experiments. However, PyNN does not yet offer the necessary explicit interfaces to model the required behavior. This section briefly discusses the pitfalls involved in using the existing interface to implement a PyNN-compatible job control.

PyNN provides `run()`, `reset()` and `end()` commands as dedicated commands for flow control, while only `run()` is actively used by most backends. Thus, commands between subsequent invocations of `run()` can be interpreted as incremental changes to consecutive experiments. Furthermore, `reset()` can be used to render later commands unrelated to earlier ones. After a final call to `end()`, the actual execution is triggered and the network representation is send to the broker.

However, no means currently exist to access result instances produced by different experiment instances. This has been reported and future versions of PyNN may support native parameter sweeps (Davison, 2011a). Alternatively, a non-blocking, promise-oriented interface may be used to retrieve the results of a specific experiment instance, which requires to only change the return types of PyNN interface functions.

3.2.3 Wrapper Generation

Multiple components of the redesigned HMF software stack, presented in Chapter 2, provide auto-generated `Python` bindings, including PyHMF. Here, the process is exemplarily introduced for PyHMF, but applies identically to the other components.

The wrapper generation is built upon `pygccxml` and `py++` (Yakovenko and Baas, 2013). The former is responsible for parsing the `C++` header files and constructing a corresponding abstract syntax tree. Subsequently, `py++` translates the syntax tree into `boost::python` (Abrahams and Grosse-Kunstleve, 2003) wrapper code, which can be compiled into a `CPython` accessible shared library.

The `pygccxml` parser is based on `g++-4.4` and is therefore limited to `C++03` language features. Custom extensions provide limited support for `C++11` containers, such as arrays and hash maps. Moreover, the code generation flow has been tightly integrated into the WAF (Nagy, 2014) build system by Christoph Koke. This means, the `Python` bindings are updated on-the-fly whenever the underlying native interfaces change.

3.2.4 Interfacing Workflow Components

Another challenge that arises in the context of the redesigned workflow is to forward data to other workflow components because the user's python interpreter and backend implementations no longer share the same process space. Therefore, PyHMF needs to provide a PyNN-compatible mechanism in order to pass payload along the tool chain. Typical use cases include user authentication, requesting particular hardware instances or guidance of the mapping process.

PyHMF provides an abstract `MetaData` base class. Workflow components can implement their own `MetaData` objects, which are registered as arguments to the PyNN `setup()` call. An example listing thereof is given in Section 6.8.

Registered instances are then serialized and sent alongside the network description to the experiment broker. The broker is responsible for forwarding the payload to the designated target components. Implementations exist for Ester, HALBe and marocco described in Sections 2.3 and 4.2 and Chapter 6, respectively.

3.3 Summary

PyHMF is a complete and efficient `C++` library for the description and representation of abstract neural networks. It offers a significant performance gain over PyNN by using connection matrices providing constant time lookups for synapses and implementing the network setup in native `C++`. The latter eliminates frequent foreign language calls from `Python` into `C` for random number generation, which is one of the main performance bottlenecks of standard PyNN. The native interfaces seamlessly integrate into other workflow components developed in `C++`. Additionally, a fully PyNN-compliant `Python` API is provided by means of code generation, tracking changes to the underlying implementations and updating bindings automatically.

Furthermore, a method has been derived to reduce the hypergraph expressed by PyNN API to an algorithmically more accessible plain graph, which enables PyHMF to preserve hierarchies in the actual network description. This hierarchical information can subsequently be accessed by backends to e.g., partition the network for efficient distributed simulations or an optimized placement of neurons to a neuromorphic hardware system. PyHMF might therefore be an interesting frontend for other simulation backends as well.

Network setup times for PyHMF are reported in Chapter 8. The actual performance gain depends on the network size and connection topology. For all studied networks, PyHMF has been more than an order of magnitude faster than the predecessor HMF and current NEST (Gewaltig and Diesmann, 2007) backends. For many instances a speedup of more than two orders of magnitude has been achieved.

4 System-Level Software

In the process of restructuring the workflow, the low-level API has been redesigned as well to provide unified access to all components of the wafer-scale system. There are two layers of low-level interfaces. A lower, stateless layer that reflects the actual hardware access granularity and a stateful layer on top, which captures the complete configuration space and provides extra convenience. The type-rich interfaces and coordinate implementations capture many common user errors during compile time by means of static code analysis. Furthermore, coordinates implement implicit range checks rendering out-of-bound access obsolete.

Several people in Heidelberg and Dresden contributed to the work presented in this chapter. The major contributors in alphabetical order are: Christoph Koke, Alexander Kononov, Eric Müller, Bernhard Vogginger and the author. The author, in particular, has been responsible for the HALBe interface design and the type-rich coordinate system.

A preliminary version of Section 4.2.3 has been published by the author in the neuromorphic platform specification document (HBP SP9, 2014).

4.1 Design Overview

The mixed-signal wafer system implements neural networks by connecting a diverse set of physical circuitry to set up neurons, establish synaptic wiring and more. Typically, inter-component connectivity is sparse and realized as either digital communication or direct analog coupling. To cope with the inherent diversity and complexity of the system, a lot of effort has been put into building solid software foundations. Two low-level APIs, HALBe and StHAL, provide access to the system on different levels of abstraction. Their design is sketched in Figure 4.1.

HALBe closely resembles the access granularity of the hardware system. It is designed for production use and low-level hardware tests alike. It provides a reentrant, free function oriented interface and defines a coordinate system that coherently addresses all components of the wafer system. This coordinate system has become the standard for referencing hardware entities throughout the workflow.

StHAL builds upon HALBe and provides higher level, stateful API. The lack of state in HALBe complicates tasks across single component boundaries. Combining individual

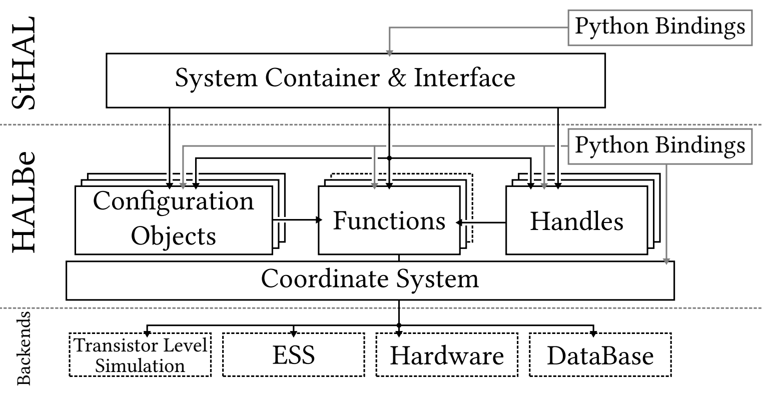


Figure 4.1: Overview of the low-level software hierarchy. Access to different simulation backends is provided on two different levels. HALBe provides fine-grained low-level access, whereas StHAL implements more abstract interfaces. The coordinate system plays a central role and is used throughout the higher software layers (not visible). Each layer provides code-generated `Python` bindings to enable scripting and low-level interactive access. The design further allows higher-level software to access different backends transparently.

neuron circuits, for example, involves multiple HALBe calls. Additionally, the system has to be configured in a sensible order, e.g., floating gates have to be set early to supply currents and voltages for other components. StHAL integrates all individual HALBe configuration objects into a single coherent configuration space describing the whole experimental setup. Thus, StHAL itself can dispatch all necessary HALBe calls in the appropriate order.

Both, HALBe and StHAL provide `Python` bindings for scriptable access at the lowest level, interactive experiments and system exploration. The bindings are automatically generated at compile time by `pygccxml`, `py++` and custom extensions, as explained in Section 3.2.3. Therefore, bindings are always up-to-date and closely follow the development of the native API.

4.2 Stateless Hardware Abstraction Layer

The lowest-level HALBe interface provides unified, fine-grained access to all features of HICANNs, DNCs and FPGAs. It replaces development and testing interfaces mostly implemented by the original hardware designers. The redesign lowers the overall complexity of the system, makes it more accessible for other developers and hardens it for production use. The interface is divided into four major components: access handles, a type-rich coordinate system, configuration containers and a stateless function backend. Typically, one of each is required for configuring a single functional hardware unit. For example, the complete configuration space of the HICANN is captured by 1 handle, 28 configuration containers, 37 functions and 59 different coordinates.

4.2.1 Design Concept

HALBe is designed to provide unified access to all hardware components at the lowest level. Well parametrized, free functions only access local data in order to minimize shared state as a common source of data races. Data dependencies are fully defined by the function signatures and mandatory shared resources, like the communication channel, which is encapsulated in handle instances and passed by mutable reference. Within a handle, conventional process synchronization means can be used at a single well-defined location to avoid data races. Functions are therefore reentrant and can be called concurrently to parallelize access to the hardware system. Furthermore, polymorphic handles provide unified access to different backends like the wafer system, ESS, transistor level simulation or database storage.

The actual configuration space is defined by a set of small containers, which reflect the hardware read-write atomicity. For example, a row of synaptic weights is represented by a single container as these weights have to be written to and read from hardware at once. Containers additionally implement simple domain specific interfaces to set up configurations more conveniently.

4.2.2 Type-Rich Interfaces

As mentioned earlier, most HALBe functionality is provided by free functions. With little exception, pairs exist for writing data to the system and reading it back. In a novel approach, the C++ type system is used to render the interfaces safer, more expressive and implementations shorter. This builds the foundations for the HALBe coordinate system presented afterwards. Firstly, the approach is introduced by example, followed by a short discussion of the advantages and its general applicability.

A typical pair of functions is shown in the listing below. In this case, they configure a line of crossbar switches and reads back the current configuration.

```cpp
void set_crossbar_switch_row(
    Handle::HICANN& h,
    Coordinate::HLineOnHICANN const& y,
    Coordinate::SideHorizontal const& s,
    CrossbarRow const& switches);

CrossbarRow get_crossbar_switch_row(
    Handle::HICANN& h,
    Coordinate::HLineOnHICANN const& y,
    Coordinate::SideHorizontal const& s);
```

Here, handle `h` designates a communication channel to a specific HICANN chip, `y` references one of 64 horizontal Layer 1 buses, `s` references either left or right and the actual switch configuration is given by `switches`. Note that the handle `h` is the only captured, mutable state. The novel interface as well as a corresponding more conventional C-style interface are compared in the following.

```
// HALBe style
auto row = get_crossbar_switch_row(h, HLineOnHICANN(4), left);

// equivalent C-style snippet
CrossbarRow row;
get_crossbar_switch_row(h, 4, 0, &row);
```

Both functions read back the switch configuration in row 4 on the left side of HICANN `h`. Note that returning the data by value is not a performance issue, since any modern C++ compiler supports return value optimization (Meyers, 1995).

Clearly, the first listing is simpler to reason about without reading the API documentation. The type-rich inteffice, for example, states that the address value 4 refers to a horizontal crossbar switch row and 0 addresses the left side of the HICANN chip. Expressive code is important, but it is even more important that the new interface enforces safer code. Usually, the function `set_crossbar_switch_row` is responsible for sanitizing all its arguments. Generally, every interface function has to sanitize its arguments because no assumption can be made about the user's intend. This is a common source of errors in its own regard. Especially when writing data to hardware, arguments have to be checked consistently beforehand. Otherwise, failing writes due to e.g., out-of-bound access may leave the system in an inconsistent or unrecoverable state. The type-rich interface moves consistency checks out of the function scope into the arguments, which become responsible for sanitizing themselves. In the previous example, an out-of-bounds instantiation of `HLineOnHICANN` would trigger an exception before `set_crossbar_switch_row` would even been entered. As a welcome side effect, function implementations become shorter and more concise by widely skipping input sanitization. Finally, function calls with unsuitable argument types never go unnoticed. These errors are subject to static code analysis and detected at compile time.

In general, type-rich interfaces with intrinsic range sanitization have proven themselves a valuable approach to harden implementations and improve code quality wherever complex addressing is necessary. They are therefore used in the HALBe coordinate system for the wafer system, but have also been adopted for other prototype systems.

4.2.3 Coordinate System

Conventional computers access memory and IO devices by mapping them into a single linear address space. Neuromorphic hardware devices, on the other hand, implement small special purpose circuits with local memory. Using the immediate address space of the digital logic is complicated by the fact that the mapping of memory location to function varies across instances of the same analog circuit. Without shared random access, memory components have to communicate either via digital buses or physical coupling. The necessary connectivity is often defined by their relative orientation and position to one another. Thus, unlike conventional computers, programming a neuromorphic hardware device requires explicit knowledge about the topology. In fact, getting the connectivity right is one of the major obstacles when setting up experiments manually on a low-level. HALBe consistently arranges components in a Cartesian coordinate system that reflects the inter-component connectivity more naturally. Exemplarily, some of the coordinates are outlined in Figure 4.2. Here, the term coordinate system is used rather than address space because the latter commonly refers to purely sequential addressing.

All indices are consistently counted from left to right and top to bottom, according to the HICANN orientation shown in Figures 1.2 and 4.2. Grid coordinates also provide support for linear addressing by enumerating all instances in a row-first fashion. Moreover, sparse grids are supported for hardware components that are either sparse or irregularly assigned to grid points, e.g., synapse drivers only exist every other switch row. Addressing non-existing components with an invalid x, y coordinate or enumeration yields a language exception.

Finally, coordinates implement conversions to derive coordinates of connected components, for example, the coordinate of a synapse driver can be derived from a corresponding select switch row coordinate.

Implementation

The implementation of coordinates is based on a generic template library for ranged integers developed by the author. Any ranged integer instance carries its valid value range as part of the type signature. The range of an instance is sanitized during its construction and after any compound assignment operation, like `+=`, `&=`, etc. All other integer operations do not mutate the bound instance, but rather return a new one, which is in turn sanitized during construction. Therefore, no range violation goes unnoticed. Optionally at compile time, ranged integers can be collapsed into their corresponding built-in type to diminish any runtime overhead for unsafer, but higher performance production builds. The following listing exemplifies the use of ranged integers.

```
typedef integral_range<unsigned, 64, 4> type; // type in [4,64]
type a = 0;  // raises exception
type b = 4;
b =>> 1;     // raises exception (a=2)

void magic_fun(integral_range<uint8_t, 7, 0> const& v);
```

Here, instances of `type` can represent integer values in $[4, 64]$. Instantiating `a` with zero raises an exception, whereas `b=4` is valid. Subsequently, shifting `b` to the right yields two, thus raising another exception. Notably, the function declaration of `magic_fun` clearly states its expected parameter range, no extra documentation is necessary.

HALBe coordinates are implemented on top of the ranged integer library as individual types. `C++` inheritance simplifies the implementation of new coordinates. Sparse grids are implemented efficiently via CRTP (Coplien, 1995) callback functions, which eliminate virtual function overhead and enable inline code. In the following, the conciseness of this approach is exemplified for an implementation of a fully sanitized 2×2 neuron coordinate.

```
struct NeuronOnQuad :
    public GridCoordinate<NeuronOnQuad, XRanged<1, 0>, YRanged<1, 0> >
{
    NeuronOnQuad() = default;
    NeuronOnQuad(x_type const& x, y_type const& y) :
        self_type(x, y) {}
};
```

`x_type`, `y_type` and `self_type` are defined as part of the base class, which also provides the common grid, enumeration, serialization and `C++11` hash map interfaces. The new coordinate can subsequently be used according to:

```
NeuronOnQuad a(X(1), Y(1));
NeuronOnQuad b(Enum(3)); // references same neuron via enum
NeuronOnQuad c = NeuronOnHICANN(Enum(0)).quad();
```

The first two examples, `a` and `b`, reference the same neuron in Cartesian and enumeration coordinates, respectively. The third example briefly outlines a coordinate conversion.

4.2.4 Handles

Every HALBe interface function expects a hardware handle as its first argument. Handles reference target hardware components and aggregate shared state, including communication channels and `lockfile` (van Smoorenburg, 2012) handles for low-level hardware access control.

Polymorphic handles further allow for transparent switching between different available backends, like single-HICANN, wafer system, transistor level simulation, ESS or mongoDB database access (MongoDB Inc., 2013). The latter two are briefly introduced in the following. Using a single consistent code path to access all backends reduces the software complexity and simplifies verification. Moreover, arbitrary handles can be combined to carry out the same experiment on multiple backends.

mongoDB Database handles have been implemented for all hardware devices. Changing the handle instantiation is sufficient to serialize and persistently store any configuration in a database, rather than writing it to hardware. Likewise, experiments can be read from the database and conducted on e.g., a hardware instance later on. This functionality is provided by a generic `boost::serialization` archive for mongoDB implemented by the author.

A web-based visualizer has been developed by Björn Kindler. It access configurations stored in the database and therefore visualize configurations of any HALBe-based workflow including marocco, StHAL and HALBe itself.

ESS The ESS is a high-level simulator of the wafer system, see Section 2.2.5. As of May 2014, the ESS (Vogginger, 2010) was being integrated into the HALBe workflow by means of the polymorphic handle mechanism (Pape, 2013). This integration is a significant improvement making the ESS behave more like actual hardware. The ESS therefore becomes a valuable tool for workflow verification and continuous integration (Duvall et al., 2007).

4.2.5 Stateless Function Interface

Access to backends is implemented via stateless functions. They typically expect a handle, coordinates and a configuration container as their arguments. Writing data to hardware involves three steps. First, the configuration is converted into a hardware binary format. Then, coordinates are translated into memory locations. Finally, the data is written using the communication channel referenced by the handle. To read any configuration or result back from the hardware, the steps have to be carried out in reverse order, from binary to HALBe format.

HALBe functions have no shared state and are therefore reentrant. However, handles are not yet synchronized regarding the access to shared resources. Future versions of HALBe and StHAL (see below) are planned to speed up experiment setup times by parallelizing the configuration process across multiple FPGAs.

4.3 Stateful Hardware Abstraction Layer

Hardware access at a very detailed level is important for efficient parallelization and low-level tests. However, it requires extensive knowledge about component interdependencies. StHAL is a shallow layer on top of HALBe that integrates the different configuration objects into a coherent experiment representation spanning multiple wafers. Any configuration can be archived, loaded or programmed to any of the supported backends in a single step. Furthermore, explicit knowledge about hardware configuration order is no longer necessary. StHAL dispatches HALBe calls in an efficient manner. Many tasks requiring multiple configuration steps in HALBe, like recording a neuron membrane trace, can be accomplished by a single call in StHAL.

The software presented in this chapter has been developed by Christoph Koke, Eric Müller and the author.

4.3.1 Connection Database

The StHAL connection database helps to keep track of the connectivity between pluggable components to establish communication with the appropriate devices automatically. For example, setting up an analog membrane recording requires configuration and readout of a HICANN, an FPGA and a USB analog digital converter (ADC) (HBP SP9, 2014), which otherwise would have to be provided by the user. The database is also used to load calibration data and correct analog recordings automatically.

4.3.2 Python Bindings

StHAL captures the complete configuration space provided by HALBe and adds interfaces to conveniently access functionality involving several components. Furthermore, the underlying containers can consistently be accessed using the type-rich HALBe coordinate system via `C++` array overloads. Notably, this enables `Python` bindings that closely resemble the native StHAL interface. Frequently, when one language is wrapped to another, mandatory interface changes have to be made in order to be syntactically compliant with the target language. For example, `Python` functions typically return values by reference. However, these references require an *lvalue* assignment before being accessed, except for array operators. Using array overloads renders the following a valid `C++` and `Python` expression.

```
hicann.neurons[NeuronOnHICANN(X(0), Y(42))] = Neuron();
```

Here, the 42nd neuron in the upper synapse array on `hicann` is reset to its default values.

4.4 Summary

In this chapter, two modern low-level interfaces have been presented. They built a solid foundation for unified access to a complex, inhomogeneous hardware systems. The separate layers allow fine grained, race free access on the one hand and a convenient, configuration-order-invariant interface on the other. Future versions of StHAL are planned to speedup configuration times by parallelizing the access to multiple FPGAs

The novel, type-rich API makes the low-level interfaces more expressive, concise and robust. Most importantly, they provide more surface for static code analysis. While coordinates rigorously detect any out-of-bound access during runtime. On these grounds, the HALBe interface design has been adopted for other prototype systems as well.

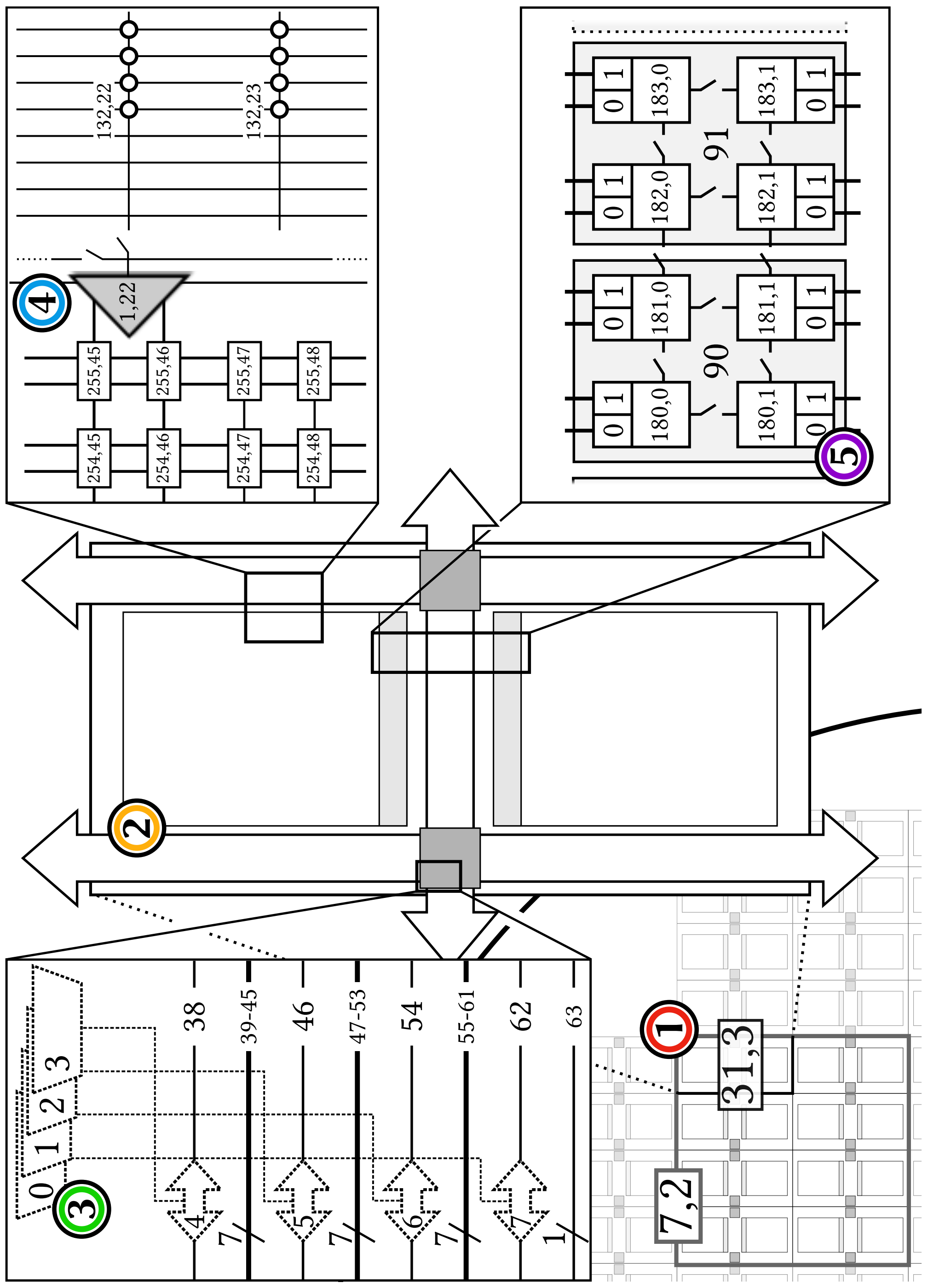


Figure 4.2: An illustration of HALBe coordinates for HICANN (31,3) **(2)** on reticle (7,2) **(1)**. Coordinates address components either as enumerations or grid coordinates, denoted as plain indices and combinations of x, y, respectively. Item **(3)** shows the mapping of DNC mergers to SPL1 repeaters. The cutout at item **(3)** exemplifies the usefulness of Cartesian coordinates for the addressing of synapses, synapse drivers and select switches. Lastly, the neuron inter-connection topology across and within blocks of 2×2 neurons is illustrated **(5)**. This figure has been published by the author in the neuromorphic platform specification document (HBP SP9, 2014).

5 Mapping Neural Networks to Hardware Specific Configurations

Manually implementing neural networks at chip-scale using the low-level interface is a challenging task due to the complexity of the wafer system. Implementing larger networks becomes rapidly infeasible. An automated translation of abstract neural network descriptions to hardware specific configurations is therefore required for HMF-scale experiments. Furthermore, it enables users without detailed hardware knowledge and hardware experts alike to utilize the system.

This chapter outlines the general requirements for such an automated translation. A strategy is introduced to derive configurations for the wafer-scale system in a modular feedforward fashion. The subsequent chapter presents implementations thereof, closely following this strategy.

In the following, the term *mapping* is used to refer to the translation of abstract neural networks to hardware specific configurations.

5.1 Motivation

Any neural network experiment supposed to run on the HMF has first to be translated into a *corresponding* hardware configuration. However, no canonic way exists to derive such configurations. In fact, it is not even clear what corresponding refers to. Generally speaking, corresponding configurations should closely resemblance the intended network dynamics within the constraints of a particular simulation substrate. Software simulators, for example, distort the dynamics due to limited machine precision and numerical artifacts. On analog hardware, the distortions are less subtle, including noise, fixed pattern variation, as well as limited parameter ranges and communication bandwidth.

How close a configuration resembles the model has to be quantified by means of a measure. An appropriate measure helps to preserve certain properties or aspects of the model during the mapping by quantifying the aberration. However, appropriate measures are model dependent, i.e., some networks are rather sensitive to analog variations, while others require the realization of particular synapses.

For simple experiments many configurations may exist that resemble the intended experiment equally well. For example, neurons can be placed to alternative locations or connections can be routed differently. The task of finding configurations that closely resemble the network model becomes increasingly complex with increasing network size.

At some point, solutions might no longer exist without accepting distortions like the loss of synapses. The acceptable degree is model specific.

Generally, any configuration can be established manually via the interfaces presented in Chapter 4. However, it requires detailed knowledge of the system, takes a lot of time and is prone to user errors. Often it is not even clear, what is the most promising strategy to set up a given network with as little distortions as possible. Additionally, configurations are hardware instance specific due to individual variations and defects and cannot easily be swapped between different setups. Moreover, exploiting the accelerated nature of the system for extensive parameter sweeps demands configuration updates at high rates.

Ultimately, realizing networks with up to 10^6 neurons BSP (BrainScaleS, 2014) and up to 10^9 neurons in the HBP (Markram, 2012b) on specialized hardware like the HMF mandates an automated tool chain for model compilation. A first version of such a tool has been developed in cooperation between Dresden and Heidelberg as part of the FACETS project (Brüderle et al., 2011). A faster, more modular successor has been developed as part of this thesis. This so-called marocco framework is explained in more detail in Chapter 6.

5.2 Requirements

This section discusses the key requirements that need to be addressed by any suitable HMF mapping flow and briefly outlines how marocco approaches them. These requirements are not fixed and may change over time as the system evolves and new use cases are discovered.

5.2.1 Modularity and Extensibility

The HMF is a complex, highly configurable system for large-scale neural network emulations. However, it is also an explorative platform for studying novel computing paradigms and seeking future applications for accelerated neuromorphic hardware systems. Any mapping has to inevitably reflect the inherent complexity of the hardware system to provide access to all its features, but at the same time needs to be flexible enough to support hardware changes as well as evolving requirements.

Marocco therefore uses a feedforward approach, described in Section 5.3, to divide the mapping task into a sequence of smaller mapping steps. This modularizes the translation process, reduces the overall complexity and simplifies development. Notably, feedforward approaches are frequently used to solve similar problems, see Section 5.3.2. Marocco further provides a suite of unit tests and uses continuous integration to verify changes to implementations and monitor progress.

Moreover, many former mapping responsibilities, like calibration and the translation of experiment results into the biological model domain, have been moved into modular software libraries or independent workflow components. Well-defined IPC interfaces between workflow components structure the mapping further, as explained in Chapter 2.

5.2.2 Objectives and Guidance

The mapping has to establish configurations that closely resemble the intended neural network from a topological as well as a functional point of view. A proximity measure is required to quantify the model distortion, as discussed earlier. Generally, finding configurations that minimize the distortions is a multi-objective optimization problem (Hwang and Masud, 1979). However, finding the most suitable measure is a problem in its own regard and strongly depends on the network. A reasonable, general optimization objective that allows to optimize generic networks was found to be the number of lost synapses because connectivity is frequently the limiting resource. Analog neuron properties, on the other hand, can be configured individually, see Section 1.4.1. Thus, maximizing the number of realized synapses is a reasonable first-order approximation to minimize network distortions.

The mapping task becomes more challenging as the network models grow larger and resource limits are approached. At some point, it might no longer be possible to implement the model without accepting e.g., synaptic loss. However, a few synapses might be functionally more important than many others. Manual guidance is therefore necessary to handle a wider range of possible network models and applications with a single tool. Marocco provides users with the ability to add mapping constraints in order to optimize configurations for their personal intend. For example, neurons can be placed manually to preserve the neighborhood relationship of model neurons, which typically reduces synaptic loss, see Section 8.5. Furthermore, synaptic connectivity can be prioritized to guide potential loss either towards synapses that are functionally less important or assert particular connections. A list of implemented mechanisms is given in the final thesis discussion.

5.2.3 Defect Handling

Wafer-scale integration is the key technique that provides energy efficient, high bandwidth communication for the wafer system. However, random manufacturing defects inevitably remain in the system, as explained in Section 1.6. The wafer-system allows to workaround most localized defects by setting up the configuration appropriately, e.g., defect neuron circuits can be omitted or Layer 1 connectivity routed differently. The mapping has to respect defects and find workarounds, wherever possible and appropriate, to efficiently realize reproducible network experiments at wafer scale.

The implementations presented in Chapter 6 can algorithmically handle defect neuron circuits, mergers, Layer 1 network components, synapse drivers and synapses. The management, blacklisting of components and integration of defect maps into the mapping flow is discussed in Chapter 7.

5.2.4 Performance

Finding suitable configurations as fast as possible is important to exploit the accelerated system for extensive parameter sweeps on the one hand and to increase hardware utilization for multi-user operation on the other, as described in Section 2.2.1. A very slow network translation, on the other hand, might even render very large neural networks computationally inaccessible.

Clearly, the algorithmic complexity and therefore the mapping time inevitably depends on the size of the neural network model. In fact, any implementation has to scale at least linear with the number of model synapses because every synapse needs to be handled individually. Notably, the number of synapses in networks of N neurons and with predominantly local connectivity is N, opposed to N^2 for random network topologies.

A short introduction to algorithmic complexity follows to characterize the performance of implementations in Chapter 6. Afterwards, distributed and shared-memory parallelization are briefly discussed (Shan and Singh, 2001), regarding their suitability for the mapping task.

Algorithmic Complexity

The BSP and HBP target network sizes of 10^6 and 10^9, respectively. This mandates the use of efficient algorithms and data structures to derive suitable configurations in reasonable time.

The *Big O Notation* is used in the following to characterize the algorithmic complexity as a function of the input size N. Here, $f(N) = O(g(N))$ denotes that $c \cdot g(N)$ provides an upper bound for $f(n)$. Thus, a constant c exists such that $f(n) \leq c \cdot g(n)$ for all $n > n_0$ and a sufficiently large n_0.

The Big O Notation provides machine independent comparability by characterizing runtime in terms of lower and upper bounds rather than specific time measurements. For example, a simple linear algorithm might run on a given input for 100 s, but runs twice as long on a machine only half as fast. This methodology is explained in more detail by Skiena (2008).

Parallelization

Apart from more efficient algorithms, parallelization can be used to speed up program execution. In this case, the actual problem in question remains at least as complex as in the sequential case, but is solved by multiple processors in parallel. Not every problem can be parallelized efficiently. Two major approaches exist, distributed and shared-memory parallelization, which mainly differ in terms of data locality and communication overhead. Their suitability regarding the translation of neural networks to hardware configurations is discussed in the following.

Distributed parallelization (Graham et al., 2006) is beneficial in situations where the problem can be divided into smaller subproblems and only little synchronization is required between the processes solving the individual parts. This approach is typically employed on computer clusters build around a fast, low-latency network. The main advantage of this approach is the scalability beyond single-machine boundaries, i.e., computers can be added to cope with larger problems. For problems that require a lot of communication, the network latency ultimately becomes the dominating bottleneck. In fact, many problems can be solved faster on a single machine due to lower latency. Furthermore, designing and implementing efficient distributed algorithms is typically more complex compared to sequential versions.

Shared-Memory parallelization (Robbins and Robbins, 2003) provides cheaper means of communication and process synchronization, which renders more problems susceptible to concurrent execution. Multiple *threads* of execution operate on a shared memory. The scalability of this approach is limited by the number of processing units in the system with access to this memory. Nevertheless, the trend in processor technology is towards many core systems. Off-the-shelf servers with tens of cores are easily available. One step further, graphics cards provide thousands of shader units. However, suitable problems need to be massively parallel and consist of simple subproblems. Matrix multiplication is a popular example, but even then optimizing performance is a challenging and platform dependent task (Baskaran and Bordawekar, 2008; Yang et al., 2011).

Suitability Mapping networks at the wafer scope frequently involves the assignment of shared resources like neurons or Layer 1 buses. Any such assignment requires synchronization in order to avoid data races or resource overallocation and to make sure all processes work on up-to-date resource maps. These assignments cannot efficiently be solved by distributed computation. However, marocco uses shared-memory parallelization to speed up susceptible mapping tasks that predominantly assign local resources. The respective benefits and shortcomings are individually discussed for each implementation in Chapter 6.

5.3 A Wafer Mapping

Mapping arbitrary neural network to hardware is generally a complex multi-objective optimization problem. Firstly, the quality of accessible solutions is discussed. Then, a partitioning of the mapping task into a sequence of smaller subproblems is presented. Subdividing the problem complicates finding globally optimal configurations, however it improves both, scalability as well as modularity, and is therefore the method of choice.

5.3.1 Optimal Solutions

Finding suitable hardware configurations can algorithmically be seen as a subgraph isomorphism problem, which is known to be NP-complete (Cook, 1971). Thus, no algorithms are known to yield optimal solutions in polynomial time. Networks of up to 10^6 neurons are targeted within the BSP, urging algorithms with polynomial complexity and small exponents. Consequently, only good but not necessarily optimal solutions can be found in reasonable time.

Dividing the problem into smaller subproblems reduces the individual complexity and therefore makes the overall problem more accessible. Solutions to the individual subproblems are then used to construct the final, coherent solution to the mapping problem.

5.3.2 Similar Problems

Similar problems have to be solved by FPGA tools and software compilers to translate abstract descriptions into designs and machine code, respectively. Without going into detail, these complex problems are typically solved by complex tools that are either proprietary or developed by large open source communities over years GCC (2014). Even though problems and tools vary, they mostly use linear, feedforward approached to divide the overall problem and solve the respective subproblems. For example, FPGA designs are derived in successive steps such as net partitioning, chip planning, placement, clock tree synthesis, signal routing and timing closure (Kahng et al., 2011). The modular flow further allows to use third party tools for individual steps. These tools may provide access to specific features, not part of the primary tool chain (Krupnova and Saucier, 2000).

5.3.3 Feedforward Mapping Approach

Originally, the mapping had been coarsely divided into a placement and routing step, inspired by existing FPGA design tools (Kahng et al., 2011). This approach has been described by Fieres et al. (2008). Using a finer-grained partitioning helps to improve the modularity of the workflow. However, partitioning the problem into many small subproblems, complicates finding globally optimized solutions. Reasonably, the mapping steps resemble the natural steps involved in manually setting up hardware configurations using the low-level interfaces described in Chapter 4. Figure 5.1 illustrates the individual functional wafer components that have to be configured in order to set up an experiment.

Moreover, the feedforward mapping should establish connectivity consistently from either source to target or the other way round to simplify the process of combining the individual solutions. Establishing connectivity following the flow of information from source to target is the more natural order, as Layer 1 routes start at a single SPL1 repeater and subsequently branch out to reach multiple terminals. Whereas guiding routes in backward direction from multiple targets towards the source repeater and merging the bus allocations along the way requires more effort.

The mapping task has ultimately been divided into a sequence of seven individual steps, which are briefly outlined in the following. Detailed algorithmic descriptions for implementations of these steps are given in Chapter 6.

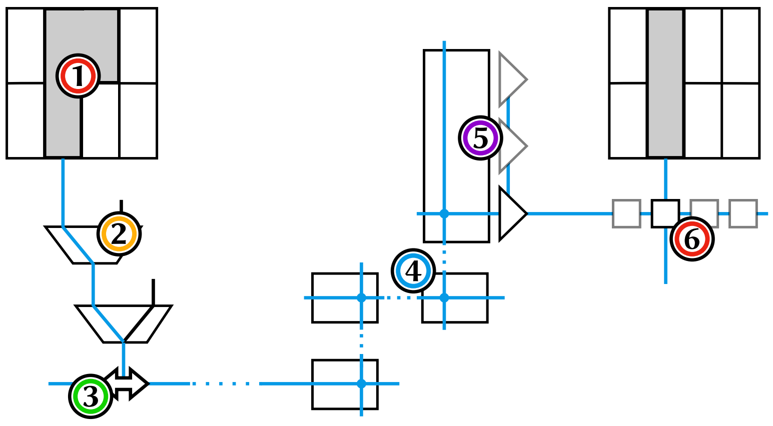


Figure 5.1: An overview of the connection topology of the wafer-scale system. The modular feedforward mapping derives configurations for all configurable components following the flow of information. Initially, neurons **(1)** have to be placed and neuron circuits combined depending on the input requirements. Next, spikes from multiple neurons and external sources are merged **(2)** and inserted via SPL1 repeaters **(3)** into the Layer 1 network. Crossbars **(4)** and repeaters are configured to establish long range connectivity. Whenever a route reaches a target chip, events are relayed from vertical buses into the synapse array via synapse drivers **(5)**. Finally, synapses **(6)** forward events vertically to target neurons.

Neuron Placement

The neuron placement is responsible for assigning model neurons to hardware neurons. This is necessarily the first step, subsequent steps require the hardware locations of source and target neurons to establish connectivity and set parameters. The neuron placement has a significant impact on the final loss of model synapses, e.g., placing densely connected neurons far apart leads to an increased routing resource utilization.

The neuron placement is also responsible for the size and shape of hardware neurons. Model neurons that receive many inputs benefit from a larger hardware representation, since every circuits adds another column of synapses. Whereas neurons receiving only a few inputs can be placed closer together in order to save neurons and use the routing resources more efficiently.

Currently, simple heuristics are used to guide the placement process. Future implementations could, for example, use edge-cut minimizing graph partitioning algorithms (Karypis et al., 1998).

Analog properties of hardware neurons can mostly be neglected during the placement because all neuron parameters except V_{reset} can be set and calibrated individually. However, blacklisted neuron circuits have to be omitted in the process.

Merger Routing

The merger tree, as explained in Section 1.5.1, routes events from neurons, background generators and external sources to SPL1 repeaters connected to specific horizontal Layer 1 buses. The merger routing however, is only concerned with neurons and background generators. External sources can be assigned more freely on the lowest merger tier and are mapped in the subsequent step.

Theoretically, 6 bit Layer 1 events from 64 sources can be merged in order to save routing resources. However, some addresses have a special purpose, i.e., address 0 is used for repeater locking (see Section 1.5.5) and four more addresses are reserved to implement disabled synapses. The latter is a consequence of the two stage address decoding scheme explained in Section 1.5.4. Horizontal synapse rows and neuron collector lines are connected whenever the lower 4 bit of an address match, even if the digital weight is set to zero. Thus, a 4 bit word is dedicated to avoid leakage conductance for disabled synapses by setting the decoder to an unused value (Vogginer, 2013; Schemmel, 2014a). In consequence, only 59 of 64 possible spike sources can be used.

In general, configurations should minimize the local SPL1 requirements in order to leave more Layer 1 resources for other connections and more repeater for external spike input. An advanced merger routing might also consider bandwidth requirements of individual neurons. However, no generic method exists to predict firing rates prior to experiment execution. A modeler could provide estimates to guide the process in future implementations.

Finally, blacklisted mergers have to be omitted. Neuron circuits belonging to defect mergers in the topmost tier should also be blacklisted because no means exist to route events from these neurons.

Spike Input Placement

The spike input placement assigns external FPGA spike sources to SPL1 repeaters. In accordance with the previous merger routing, up to 59 spike sources can be mapped arbitrarily to available SPL1 repeaters. Unlike neurons, the bandwidth requirements for spike sources can be estimated beforehand. An advanced input placement should minimize Layer 1 resource requirements on one hand and limit the accumulated bandwidth to avoid event dropping on the other. Additionally, inputs can be duplicated and injected at multiple locations to avoid long connections spanning the whole wafer.

Wafer Routing

The wafer routing derives crossbar switch and repeater configurations to connect as many mapped SPL1 repeaters as possible to vertical buses on chips with efferent neurons.

The complexity of the task depends heavily on the actual neural network topology and the ability of the prior neuron and spike input placement to preserve locality. Layer 1 routes start on a single horizontal bus defined by the SPL1 repeater assignment as part of the merger tree routing and spike input placement. Afterwards, they branch out to reach efferent neurons on multiple target chips. Events from different chips cannot be merged into a single connection. Hence, sources from any chip compete with sources from other chips for exclusive access to the same shared Layer 1 resources. Each route should therefore be as short as possible in order to reduce the amount of bus allocations and leave more resources to other connections. In a first order approximation, the problem is equivalent to the minimum rectilinear Steiner tree problem, which is known to be NP-hard (Karp, 1972; Garey and Johnson, 1979). The actual problem is even more complex because there are e.g., fewer horizontal bus segments than vertical ones making the former more valuable.

Finally, implementations need to establish routes around blacklisted buses, switches and repeaters.

Synapse Driver Routing

In this routing step, inbound Layer 1 connections or more specifically vertical Layer 1 buses are assigned to drivers in order to relay events into synapse arrays of chips hosting efferent neurons. The assignment is mainly constrained by the limited amount of drivers and the sparse select switch topology. Multiple adjacent drivers can be combined to map larger source address spaces within the capacitive Layer 1 limits. The actual number of synapse drivers necessary to realize all synapses depends on the afferent address space distribution, STP parameter sets, shape of efferent neurons and availability of synapses.

Typically, more synapse drivers should be assigned to inbound connections implementing more synapses, while connection with any local target should still receive at least a single driver. Their corresponding Layer 1 resources might otherwise be allocated in vain. However, this cannot be guaranteed because of up to 16 vertical buses compete for 14 shared drivers due to select switch sparseness and the relatively low number of drivers in relation to the number of vertical buses. Again, implementations have to deal with blacklisted components, in this case select switches and synapse drivers.

Synapse Array Routing

The synapse array routing does two things: firstly assigning 2 MSB values to the synapse driver decoders and secondly assigning model synapses to hardware instances.

The first step maps the afferent address space to sets of synapses according to the A, B, C, D pattern illustrated in Figure 1.6. Spike sources with the same 4 LSB and synapse type can be realized as part of the same set. The address space should therefore be mapped such that the set sizes represent the relative number of synapses in order to minimize the amount of synapses that are lost during the subsequent synapse assignment.

Then, model synapses are assigned to available hardware synapses in the order they appear in the model description. Future implementations could also cluster synapses with similar efficacies to maximize the dynamic range for weights in the same row, see Section 6.7.3. All model synapses can be realized if a sufficient number of drivers has been assigned in the previous synapse driver routing, not too many synapses are blacklisted and the afferent address space is mapped appropriately. Otherwise synapses can get lost.

Parameter Transformation

The last mapping step towards a complete hardware configuration is concerned with technical parameters to establish a operational regime, spike input generation and analog neuron parameters.

Firstly, the operational regime is established, which is typically a model independent task. Next, neuron parameters are translated from the biological model domain into the hardware parameter domain such that the hardware dynamics closely resemble their model paragon. However, the accessible parameter ranges for time, voltage, currents and capacities differ between biology and hardware. For example, biological voltages are typically in the mV range, whereas CMOS transistors operate in the volts domain. Parameters have therefore to be scaled and subsequently be corrected for individual circuit variations according to calibration data. Scaling and calibration are both described in more detail by Schwartz (2012) and in Section 6.7. Finally, input spike sequences have to be translated into the HTD and stochastic spike sources have to be expanded.

6 A Scalable Implementation of a Feedforward Wafer Mapping

The previous chapter introduced a decomposition of the mapping task into a series of simpler mapping steps in order to increase modularity and lower the overall problem complexity. This chapter presents implementations of these mapping steps provided by the marocco mapping framework, which has been established as part of this thesis. During the design and development a special focus was put on their scalability towards large neural networks and large-scale neuromorphic hardware systems.

Inside knowledge about the inner workings is neither necessary to map neural networks nor to run hardware experiments. However, it may help to express PyNN experiments in a form that is more accessible for automated translation or to manually guide the mapping process towards less distorted configurations.

The algorithms are introduced following the course of the feedforward mapping approach introduced in Chapter 5. For each step, first a short description of the input and expected output is given. These inputs and outputs internally correspond to well-defined `C++` data structures, simplifying the development of new or domain-specific implementations. Hardware defect maps are another input, which is typically provided by a database, see Chapter 7. They are listed separately to distinguish them from the user input defined by the neural network description. From an operational point of view, handling user input and defects is equally important, e.g., model neuron placed to defect neuron circuits are of little use at best or may even interfere. In addition to the explicit inputs, all mapping steps have read-only access to the PyHMF network description, see Section 3.2, and write access to the StHAL configuration container, see Section 4.3. At the end of each section, the algorithmic complexity of the individual implementations is briefly discussed. Benchmark results for marocco are given in Chapter 8. The chapter closes with the RoQt routing visualizer, which provides quick access to Layer 1 and synapse driver connectivity, helping users to identify resource bottlenecks in their configurations.

Preliminary versions of Sections 6.1 to 6.7 have been published by the author in the neuromorphic platform specification document (HBP SP9, 2014).

6.1 Neuron Placement

Input:	A list of available HICANN chips in the HMF; Optionally, a partial or complete user defined population placement.
Output:	An assignment of every model neuron to a set of hardware neuron circuits.
Defects:	Blacklist of neurons.

The redesigned mapping process provides a simple but fast neuron placement implementation. In PyNN, network connectivity is typically established between populations. Thus, populations tend to have common sources and targets. Heuristically placing neurons within a population to the same chip saves routing resources because synapses can be realized by shared Layer 1 connections. This works particularly well for feedforward neural network architectures. However, population views and assemblies simplify the realization of other connection patterns (see Chapter 3), rendering the heuristic less effective. Users may improve the mapping quality by partitioning their networks such that populations aggregate neurons that are intended to end up closely together on the wafer.

Additionally, users can flexibly place populations to individually sized hardware neurons, providing fine-grained control over input counts. This is a major advantage over the older predecessor mapping, where all model neurons had to have the same hardware neuron size. The current implementation only allows block shaped neurons, which can receive events from both, top and bottom, synapse arrays to simplify synapse array routing later on. Thus, neurons have to consist of an even number of circuits. Consequently, events can be relayed to neurons from the top and bottom synapse arrays. In practice, the restriction to block shapes is only a minor limitation, because input counts of medium sized networks easily exceed the number of synapses provided by a single neuron circuit. In fact, a reasonable default size has found to be eight interconnected circuits, see Chapter 8.

Guidance

The implementation allows users to guide the neuron placement by explicitly specifying target chips for populations. The modeler can also control the shape of neurons individually for each population to tune the input counts, such that neurons in populations receiving lots of input are realized as larger hardware neurons. Whereas neurons receiving little input can be realized by smaller hardware neurons to save resources. The following code listing demonstrates how the neuron placement can be controlled from a PyNN script.

```
marocco = Pymarocco()
marocco.placement.setDefaultNeuronSize(4)
setup(marocco=marocco)

# ...
p = Population(113, IF_cond_exp)
```

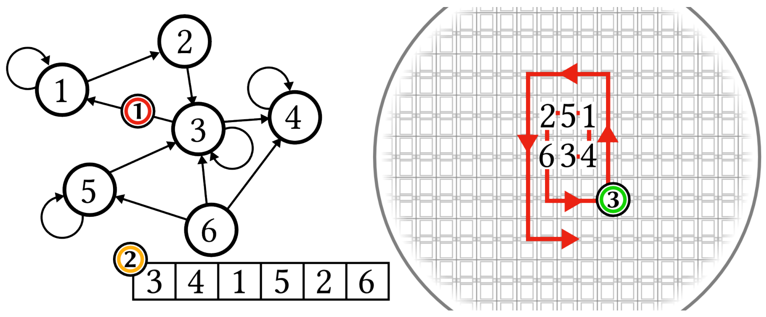


Figure 6.1: The PyHMF population graph **(1)** is sorted by in-degree in descending order **(2)**. Populations from the list are placed iteratively to hardware neurons on chips spiraling outwards from the center of the wafer **(3)**. This simple heuristic keeps neurons within a population close together on the wafer. Moreover, populations with lots of input are likely to end up closer to the center, which reduces the routing resource requirements, assuming that sources are uniformly distributed on the wafer.

```
targets = [HICANNGlobal(X(14), Y(20)), HICANNGlobal(X(15), Y(20))]
marocco.placement.add(population=p, targets=targets, size=8)
```

Initially, a PyMarocco instance is generated and the default neuron size set to four. Then, a population of 113 neurons is placed to the HICANN chips with Cartesian coordinates $(X, Y) = (14, 20)$ and $(15, 20)$. Neurons in this population will be represented by hardware neurons consisting of eight circuits, despite the default. The PyMarocco object is the PyHMF interface for the mapping framework and is described in Section 6.8.

Algorithm

The algorithm keeps track of available neuron circuits to make sure they are only assigned once or not at all in case of blacklisted circuits. First, manually placed populations are placed to their respective hardware locations. Whenever the manual placement runs out of resources because either not enough chips have been specified for a population or too many neurons have been blacklisted, the assignment fails and the user is informed.

In the next step, all remaining populations have to be placed. The process is illustrated in Figure 6.1. Firstly, populations are sorted in descending order based on their in-degree in the population graph, i.e., populations receiving more input are placed earlier. Secondly, all available chips are sorted in ascending order based on their location (X, Y). For any two chips at (X_0, Y_0) and (X_1, Y_1), chip 0 appears prior to chip 1 if it is closer to the center of the wafer. In case they are equally far away, chip 0 appears before chip 1 if α_0 is smaller than α_1 for $\alpha_i = \text{atan2}\,(X_i, Y_i)$. The resulting ordering lists chips along an imaginary spiral, starting in the wafer center and growing towards the wafer boundaries. Starting with chips in the center helps to avoid boundary effects like reduced Layer 1 resource densities for the outermost HICANNs.

Furthermore, the resulting placement has a convex shape, which can be beneficial for some wafer routing implementations, for example, the *iterative horizontal growth* wafer routing introduced in Section 6.4.4.

Now, populations and chips are iteratively taken from the beginning of both lists. Each time, as many as possible model neurons are assigned to available hardware neurons from left to right. Not yet assigned model neurons are reinserted at the beginning of the population list in order to be assigned in the next iteration. How many hardware neurons are required depends on the number of model neurons, the defect distribution and the hardware neuron size. The latter is given by a configurable default size. If a chip provides more neurons than necessary, it is reinserted at the beginning of the chip list and used for the next population. Even though large populations may end up being fragmented across multiple chips, they are guaranteed to be placed nearby. This reduces the routing resources necessary for connections targeting this population.

By default, the number of available hardware neuron circuits per chip is artificially constrained for hardware neuron sizes of 4 and 8 to increase the Layer 1 address utilization, as explained in Section 5.3.3. Fewer SPL1 repeaters are required in the subsequent merger routing stage to route events from all neurons off chip, if the number of placed model neurons is reduced to 118 and 59 for 4 and 8-circuit configuration, respectively. Experiments have shown that this is generally beneficial to improve the synaptic loss. However, can be turned off to increase the number of available neuron circuits.

Runtime

Initially, two lists are sorted. Sorting is done in $O(n) = n \lg n$, where n denotes the number of elements in the list. Then, model neurons are iteratively assigned to hardware neurons. In the worst case, only a single neuron is assigned per iteration. This contributes a worst-case limit of $O(N) = N$ for N neurons. The contributions sum up to

$$O(n, m) = m \lg m + n \lg n + N \qquad , \tag{6.1}$$

where n is the number of populations and m the number of chips. Which of both sorting terms dominates the runtime ultimately depends on the input size. In any case, the implementation is efficient enough to place a large number of neurons to large deployments of the HMF.

6.2 Merger Routing

Input: Neuron placement.
Output: A merger tree configuration and Layer 1 address assignment.
Defects: Blacklist of mergers and SPL1 repeaters.

After the model neurons have been placed, the next step towards connecting neurons is to map their outputs to SPL1 repeaters. As explained in Section 5.3, only 59 of

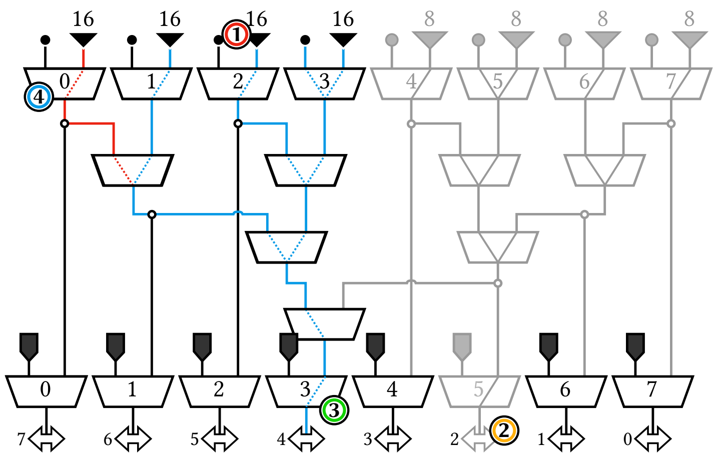


Figure 6.2: The merger routing algorithm connects neuron blocks **(1)** to SPL1 repeaters **(2)**. The number of placed neurons is indicated above each block. Moreover, already established merger configurations are indicated by solid lines within mergers, while currently considered ones are shown as dashed lines, see text. The neuron blocks greyed out on the right have been merged via DNC merger 5 in a previous iteration. The remaining blocks from 0 to 3 are currently considered to be merged via DNC merger 3. However, by adding block 0, the total number of neurons would exceed 59, thus only blue connections are established. Block 0 will be routed via SPL1 repeater 7 in a later iteration.

the theoretically 64 available addresses can be used. Address 0 is reserved repeater locking and all four addresses of format `XX0001` are reserved to reduce leakage for unused hardware synapses, with $\texttt{X} \in \{0, 1\}$. Thus, events from up to 59 neurons in different neuron blocks can be merged in order to save Layer 1 resources. To establish the event routing for the merger tree, every merger is set to either forward its left, its right or both of its inputs. Figure 6.3 recaptures the merger tree topology introduced in Section 1.5.1 and illustrates the iterative merger routing implementation.

Unlike its name, the merger tree is actually not a tree. It is an overlay of 8 trees with the SPL1 repeaters as their respective root nodes. The complete structure can be modeled more conveniently as a directed acyclic graph. Edges connect mergers from lower tiers to mergers in higher tiers. Using a graph, rather than implementing the fixed topology as part of the algorithm, allows to keep the algorithm generic and to cope with defects efficiently, i.e., blacklisted mergers are simply skipped during graph construction.

Looking at the merger tree, one of the first things to notice is that mergers are not equally capable of multiplexing events from different blocks. For example, the bottom most merger connected to DNC merger 3 is the only merger capable of collecting events from all neuron blocks. DNC merger 5, on the other hand, can only capture events from blocks 4 to 7. The algorithm considers mergers in a fixed order from more to less capable in order to find configurations that use as few as possible SPL1 outputs.

In a first trial, a breadth-first search (Skiena, 2008) is started from DNC merger 3 to discover reachable neuron blocks. If less than 59 model neurons are placed on the chip and no defects are present, merging all blocks yields a complete, efficient and valid configuration. Therefore, DNC merger 3 is set to forward its right input, all intermediate mergers are set to forward both inputs and all other top mergers except merger 3 are set to right only. Additionally, the top merger 3 has to merge address 0 events from the background generator to lock repeater DLLs. If all blocks have been captured, no further trials are necessary, otherwise the current trial is discarded.

In a second trial DNC mergers are considered for merging in the following fixed order 5, 3, 1, 6, 4, 2, 7, 0. Unlike the first trial, assignments are established iteratively and partial configurations are kept even if not all reachable blocks can be captured because the total number of neurons would exceed 59.

In each iteration, the accessibility of yet unassigned blocks is determined by means of a breadth-first search originating from the current DNC merger i. Then, starting from the top-tier merger i, as many as possible accessible, neighboring blocks to the left and right of merger i are merged until the total number of neurons exceeds 59. Neurons within these blocks are therefore assigned to the corresponding SPL1 repeater and require no further treatment. The actual merger configuration is found by following the path established during the graph search to each of the captured blocks. Moreover, the top-tier merger i is configured to forward both its inputs in order to insert address 0 background events and the bottom DNC merger i is set right only. Afterwards, utilized mergers are removed from the graph and are therefore no longer considered in subsequent iterations. The iteration terminates as soon as there are no more pending blocks or all DNC mergers have been processed. DNC mergers that have not yet been assigned are available for spike input placement in the subsequent mapping step.

The initial trial with merger 3 is important in order to find optimal configurations when all neurons can be mapped to a single SPL1 repeater. However, starting the second trial with merger 3 can yield an unresolvable edge case where neurons from blocks 3 and 4 can be merged but not from block 5.

Finally, 6 bit Layer 1 addresses are assigned to neurons. The assignment is in principle arbitrary, however, using consecutive addresses to minimize the number of different MSB decoder values improves the synapse driver assignment later on. Furthermore, addresses are assigned from the highest to the lowest address because the lowest address block $[0, 15]$ contains 2 reserved addresses.

On the bottom line, the strategy produces efficient configurations for arbitrary neuron placements in terms of Layer 1 bus utilization and can also handle merger defects efficiently.

Runtime

In the worst case, all 8 mergers have to be considered in the second trial, which requires constant time per chip independent of the actual neural network extent. Thus, a local merger routing for N chips is established in linear time $O(N) = N$.

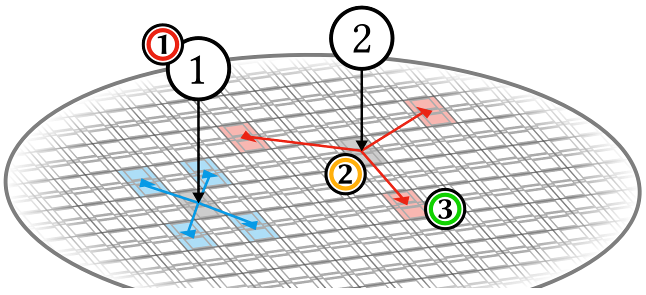


Figure 6.3: The spike input placement is responsible for assigning spike sources to SPL1 repeaters. Spike inputs **(1)** are inserted at SPL1 repeaters on chips at or near the geometric center **(2)** of their corresponding target chips **(3)**. Sources targeting many chips are assigned in an earlier iteration and can therefore be assigned more freely.

Furthermore, merger tree configurations for multiple chips are derived efficiently in parallel. No means of synchronization are necessary because only chip-local resources are assigned.

6.3 Spike Input Placement

Input: Neuron placement and a list of remaining SPL1 repeaters.
Output: Mapping of spike inputs to SPL1 repeaters and Layer 1 address assignment.
Defects: Blacklist for SPL1 repeaters.

The spike input placement is responsible for mapping external spike sources to SPL1 repeaters as injection points into the Layer 1 network. The actual spike trains are prepared as part of the parameter transformation, described in Section 5.3.3, because spike times depend on the speedup of the system, which is subject to analog neuron parameters. Note that the coordinates of input FPGAs and DNCs are implicitly defined by the HICANN coordinate and the FPGA interconnection topology. In fact, the StHAL interface provides HICANN-based access to input spike trains.

The algorithm minimizes the required Layer 1 resources, later required in the wafer routing, by injecting spike inputs close to the geometric center of their corresponding target chips, as shown in Figure 6.3. Key to a fast implementation is to find suitable insertion points efficiently. Therefore, available SPL1 repeaters are organized in a KD-tree data structure (Bentley, 1975). The `nanoflann C++` library (Blanco, 2013) provides the necessary nearest neighbor algorithms.

Initially, spike input populations are collected in a list and sorted in descending order by the number of target chips. Afterwards, the ideal insertion point is determined for each entry as the closest available SPL1 repeater to the geometric center of the target chips. All SPL1 repeaters which are not blacklisted and have not yet been designated for relaying events from neurons are available for external inputs.

Iteratively, input populations are taken from the front of the source list. The SPL1 repeater closest to the ideal insertion point is then queried in the KD-tree. Again, up to 59 sources can be mapped to a single SPL1 repeater. If the number of sources exceeds the remaining capacity of the SPL1 repeater, as many as possible sources are assigned and the remaining ones are reinserted at the front of the source list. In case the repeater has been fully assigned, it is removed from the tree. The algorithm terminates when either all inputs have been assigned or the system runs out of available SPL1 repeaters. In the latter case, the user is informed.

Runtime

The worst-case runtime for M inputs and N chips is given by

$$O(N, M) = N^2 + M \cdot N \qquad . \tag{6.2}$$

The first N^2 term is contributed by the initial KD-tree construction, while the second $M \cdot N$ term denotes the repeated queries and deletions from the tree for all M inputs. In the average case, the KD-tree access complexity is reduced to $N \lg N$, resulting in

$$O(N, M) = (N + M) \cdot \lg N \qquad . \tag{6.3}$$

Thus, the implementation can be used up to large hardware and neural network sizes.

6.4 Wafer Routing

Input: Neuron Placement; Assignment of neurons and spike sources to SPL1 repeaters.

Output: A complete configuration of crossbar switches and repeaters; A mapping of afferent neurons to vertical buses on chips with efferent neurons.

Defects: Blacklists for buses, repeaters and crossbar switches.

The wafer routing configures repeaters and crossbar switches to establish long-range connectivity via the Layer 1 network. Here, two implementations are presented, a simple one using Dijkstra's algorithm to discover shortest paths iteratively and an optimized Layer 1 graph search that minimizes horizontal bus allocations. Both algorithms efficiently realize detours to cope with non-rectangular routing grids resulting from wafer boundaries, defect elements or local congestion. In this context, a detour is simply an alternative route that leads around an obstacle, like blacklisted Layer 1 buses.

6.4.1 Graph-Based Wafer Network Representation

Graphs provide a natural representation of the Layer 1 network. Topology, defects and current resource utilization can be represented alike. The major advantage of this approach is that algorithms can be developed more generically. They recursively discover possible routes by searching the graph. Whenever a route is established,

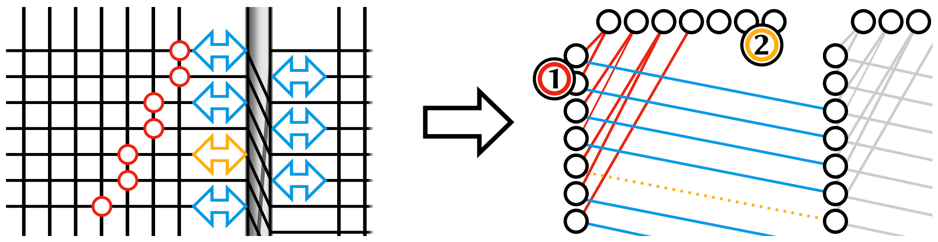


Figure 6.4: The Layer 1 routing topology is implemented as an undirected graph. Horizontal **(1)** and vertical **(2)** buses are represented as vertices, while repeaters and switches connecting buses are realized as edges. Defect hardware components, like the repeater highlighted in yellow, are left out during graph construction and therefore omitted during the wafer routing.

participating components are removed from the graph to keep an up-to-date map of currently available resources. Decoupling the algorithm from the network topology renders the implementations more flexible to topology changes and can be used to test future hardware designs, as presented in Section 8.4.4.

The Layer 1 topology is modeled as an undirected graph because buses have no preferred direction, which means events can be transmitted from left to right and right to left alike. Buses are modeled as vertices. Crossbar switches as well as repeaters connect buses, hence they are modeled as edges. Figure 6.4 depicts the mapping between the Layer 1 fabric and graph components.

The corresponding dual graph, where buses are modeled by edges and vertices represent switches and repeaters, is unsuitable. This representation *splits* bus lanes, such that routes can occupy a fraction of a bus which is physically impossible.

The wafer routing implementations use the `boost::graph` library (Siek et al., 2001), a popular and efficient `C++` template library that ships with many algorithms.

6.4.2 Common Tasks

Both wafer-routing implementations realize routes iteratively, but use different strategies to discover candidate routes in the graph. Their commonalities are discussed in the following to focus on their differences later on.

Route Representation

A Layer 1 route is defined by its starting point at a single source SPL1 repeater, a list of intermediate buses and the vertical target buses on chips with efferent neurons. Note that the injection point on the sending side is well-defined down to the single bus. On the receiving end, any vertical bus running alongside a target chip can in principle be used to relay events into the synapse array. Pending routes, which have not yet been routed, are defined as a tuple of the source SPL1 repeater and a list of target chips.

Pending routes can be constructed for every source chip by following all projections originating from local neurons to the efferent populations. Neurons therein are translated into target chips by looking up their placement.

Importantly, all projections from all neurons mapped to the same SPL1 repeater have to be routed at once because the necessary resources are removed from the graph after the route has been established.

Route Priority

Routes are established iteratively. Pending connections that are routed earlier have access to more resources, making their realization more likely. Thus, routing priorities are implemented by scheduling the allocation order. By default all pending routes have the same priority. Pending routes of equal priority are sorted based on the distance of their source SPL1 repeater from the wafer center. Connections originating closer to the center are routed first. On the other hand, realizing secluded routes with distant targets first may congest central areas early. This typically increases synaptic loss, especially since neurons with high in-degree have been placed towards the center.

Users can guide the wafer routing by specifying custom priorities for PyNN projections. However, there is no direct correspondence between projections and Layer 1 routes. Routes can represent multiple projections from multiple populations. The effective route priority is defined as the maximum priority of any represented projection.

Parallelization

Multiple pending routes compete for the same shared Layer 1 resources. The wafer routing therefore lacks a natural partitioning into independent subproblems, which makes it notoriously hard to parallelize. Even if the access to individual buses is managed such that only a single thread can allocate at any time, competing threads can still steal buses from other thread, potentially trapping them in unresolvable situations. Real world approaches exist for parallel routings, however, with limited concurrency and high synchronization costs (Gort and Anderson, 2010). For reasons of simplicity, the wafer routing is carried out purely sequential.

Adjacent Synapse Drivers

Events can be relayed from vertical buses to synapse drivers on the corresponding as well as the adjacent HICANN, see Figure 1.5. In order to simplify the parallelization of the synapse driver routing in the subsequent wafer routing step, both implementations avoid configurations where a single vertical bus is used to inject events into both HICANNs. Otherwise, synchronization is required to ensure that the cumulative capacitive load on the bus does not exceed the acceptable limit. Dropping this confinement yields topologically valid configurations, however, capacitive limits are no longer enforced. In the worst case, twice as many drivers are connected.

Greedy Iterative versus Global Optimization

Finding globally optimal solutions that minimize the overall synaptic loss are unlikely to be found with greedy iterative approaches (Black, 2005). Monte Carlo (Motwani and Raghavan, 1995) multi-objective optimization approaches, on the other hand, strive for globally optimized solution. Popular examples are simulated annealing (Kirkpatrick et al., 1983), genetic algorithms (Goldberg, 1989; Mitchell, 1996) or particle swarm optimization (Parsopoulos and Vrahatis, 2002). The performance of these approaches generally depends on the efficient generation of candidate solutions and evaluation thereof. Both conditions cannot easily be met for the wafer routing. For example, evaluating the synaptic loss as the primary optimization target requires to carry trough the subsequent two mapping steps for every candidate solution, which is infeasible for an efficient routing.

The iterative algorithms implement corrections to attenuate their greediness and leave important resources in foresight to connections routed later in the process, e.g., horizontal buses with mapped SPL1 repeaters are avoided. Allocating these buses makes it impossible to route the corresponding sources. Additional corrections are outlined alongside the individual wafer routing implementations.

6.4.3 Iterative Shortest Path Routing

After modeling the Layer 1 network as a graph, the routing problem can intuitively be approached using available shortest path algorithms. A good starting point is Dijkstra's algorithm (Dijkstra, 1959). Modern implementations find the shortest path between a single bus and any other bus in $O(E, V) = |E| + |V| \log |V|$ (Fredman and Tarjan, 1987), where E is the set of switches and repeaters and V the set of buses. Here, the stock implementation provided by the `boost::graph` library is used. Note that the problem size decreases over time, as resources are taken out of the graph from iteration to iteration.

Graph searches are invoked iteratively for all pending routes, see Section 6.4.2. In each iteration, Dijkstra's algorithm is invoked to discover the shortest path from the current source SPL1 repeater to any other bus in the graph. The distance between two buses is given by the sum over all weighted edges along the path, thus crossbars and repeaters. This can be used to flexibly model hardware characteristics as explained further below. Shorter paths are discovered earlier in the process due to the greediness of the algorithm. Consequently, the first reachable vertical bus on a target chip yields the shortest possible connection to this chip.

Whenever a connection to one of the pending targets has been found, the actual path is checked to make sure it meets the Layer 1 capacitive constraints, see Section 1.5.5. Possible violations are multiple switches set per bus on the same HICANN to establish chip-local detours, for example, from one horizontal bus to a different horizontal bus on the same chip. The capacitive limit cannot be modeled by weighted edges. Instead, backtracking is used to detect and omit these configurations. The candidate path is

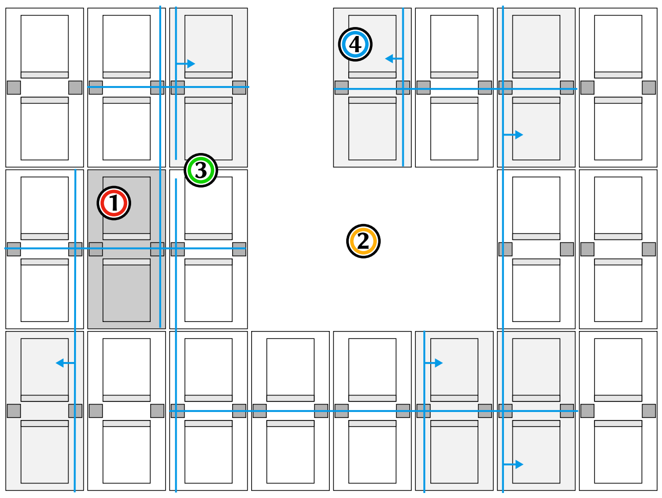


Figure 6.5: A routing problem solved via the iterative shortest path algorithm. The source chip **(1)** of the route is highlighted in dark and its targets in light grey, while defect chips **(2)** are left out. Dijkstra's algorithm generically searches the shortest path from the source SPL1 repeater to any other bus, including vertical buses on target chips. Depending on edge weights and the current resource allocation nearby chips might be discovered on different paths first, resulting in non-optimal solutions **(3)** that occupy more buses than necessary. However, targets are reached whenever possible, including the chip at **(4)**, which is missed by the horizontal growth algorithm shown in Figure 6.6.

traced from the target to the source bus. Whenever a crossbar switch is encountered, it is checked whether other switches in this crossbar have been used by this path already. If so, the candidate is discarded and the current graph search is continued. When the source is successfully reached, all utilized crossbar switches and buses are memorized and allocated, respectively. Furthermore, the target chip is removed from the list of pending targets and Dijkstra's algorithm is continued for remaining target chips.

The search ends either if all target chips have been successfully routed, no further buses can be reached or the current path exceeds a configurable length limit. The latter avoids routes of last resort whenever detours across the whole wafer are the only possible option. Realizing these routes takes large amounts of Layer 1 resources, which might better be left to other connections. The maximum length can be specified as multiples of the L_1 distance between source and the furthest target chip.

The algorithm ultimately terminates when all pending routes have been processed.

The iterative shortest path routing is exemplified for the routing task depicted in Figure 6.5. Notably, the algorithm can route concave areas and therefore reach the chip at item (4).

Modeling Hardware Characteristics

The length of any path established by Dijkstra's algorithm is measured as the sum over all weighted edges along the way, thus the distance measure can be controlled to model hardware characteristics.

These weights can be set by the user to customize the routing. Currently, three different types of edges are distinguished for connections to vertical buses, to horizontal buses and to buses with mapped SPL1 repeaters. By default, the former two are equally set to one. The latter are expensive and therefore set to 10^4. Dijkstra's algorithm avoids these buses as long as other options exist. Using them becomes a matter of last resort, such routes are skipped completely if the effective length exceeds the maximum route length. Weights are calculated on-the-fly to emulate directed edges on the undirected graph, save memory and implement congestion control as explained below.

Congestion Control

Always picking the shortest possible path to construct routes yields a greedy approach. In fact, Dijkstra's algorithm itself is greedy, it always continues with the shortest, yet unvisited, edge and ultimately produces optimal results. This is different for resource allocations, where instances are exclusive and iteratively taken away. An optimal routing would have to consider all connections at once, before allocating any resource, to produce configurations that minimize the global synaptic loss.

Here, greediness is attenuated by scaling edge weights according to the current bus utilization. Thus, paths through congested areas appear effectively longer. The total weight $w_{\text{total}}(h, i)$ for any edge connecting to a horizontal bus ($i = 0$) or vertical bus ($i = 1$) on chip h is given by

$$w_{\text{total}}(h, i) = w_{\text{static}}(i) + \alpha \cdot \nu(h, i) \qquad . \tag{6.4}$$

Where $w_{\text{static}}(i)$ models the static hardware characteristics, introduced above, and $\nu(h, i)$ denotes the number of allocated buses for chip h and orientation i. The constant α is a scaling factor to control the relative amplitude of the static and dynamic weights and is $\alpha = 1/40$ by default.

Shortcomings

The iterative shortest path approach tends to allocate more resources than necessary. Constructing minimum rectilinear Steiner trees (MRSTs) rather than combining shortest paths would minimize the number of allocated resources individually for every iteration. However, MRSTs construction is an NP-complete problem (Karp, 1972; Garey and Johnson, 1979). Most efficient MRST heuristics require fixed vertex sets (Kahng and Robins, 1992), however pending routes only specify target chips rather than specific target buses. Early experimental results have shown that the wafer routing is mostly limited by horizontal bus occupation. A second algorithm has specifically been design to allocate as few as possible horizontal buses in every routing iteration. The implementation is subsequently presented in Section 6.4.4.

Runtime

In the worst case, the whole graph has to be searched for every source to find all targets. Similar behavior is expected for random network models, where any two HICANN chips have to be connected, see Section 8.4. The number of source SPL1 repeaters is proportional to the number of model neurons N. The algorithmic complexity is therefore bound by the N invocations of Dijkstra's algorithm, which has a runtime complexity of $O(E, V) = |V| \log |V| + |E|$ (Siek, 2001). Where V is the set of buses and E the set of switches and repeaters. The backtracking is typically fast and uses internally constant time lookups. It contributes a multiplicative worst-case complexity of $|V|$. This leads to the following runtime complexity

$$O(N, E, V) = N \cdot |V| \cdot (|V| \log |V| + |E|) \qquad . \tag{6.5}$$

At first glance, the complexity scales linearly in the number of neurons N. However, the hardware extent denoted by E and V has to be chosen sufficiently large to accommodate all neurons, which can then considered proportional to N. This is only a conservative worst-case assessment, sparse network models are typically routed much faster and also full-wafer random networks have successfully been mapped, see Section 8.4.1. Furthermore, V and E are constantly reduced in size, as resources are allocated and therefore removed from the graph.

6.4.4 Iterative Horizontal Growth Routing

The iterative horizontal growth algorithm is the second wafer-routing implementation. It is optimal in terms of horizontal bus utilization. Similar to the *backbone* algorithm described by Fieres et al. (2008) only $N + 1$ horizontal buses are required if the leftmost and rightmost HICANNs are horizontally N chips apart. However, the following algorithm does not depend on a single horizontal *backbone*, which renders the approach suitable for defects, congested areas and non-rectangular routing grids.

Routes grow horizontally until either the outer most chip has been reached or the horizontal continuation of the path is blocked. In the latter case, a vertical detour is established to continue the horizontal growth in another row of HICANNs. Whenever a chip is reached that is in the same column as any of the target chips, a vertical connection is established. Figure 6.6 shows a routing problem solved by horizontal growth. Note that the algorithm does not change directions as the connection grows, therefore concave neuron placements cannot be routed.

Like the previous shortest path algorithm, the horizontal growth algorithm is invoked iteratively for all pending routes according to Section 6.4.2. Whereas the actual horizontal growth for every iteration is implemented recursively. For every pending route, starting at the horizontal bus of the source SPL1 repeater, the route grows left

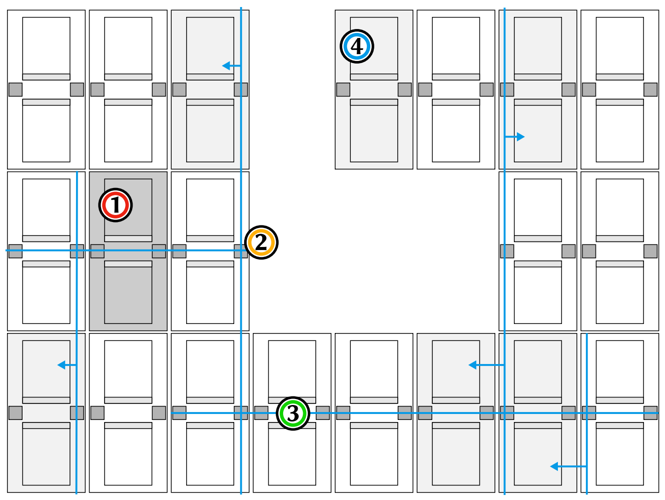


Figure 6.6: The same routing problem as shown in Figure 6.5 solved by the iterative horizontal growth algorithm. The source chip **(1)** of the route is depicted in dark, while requested targets are colored in light grey. Defects **(2)** are vertically detoured to continue growth in another HICANN row **(3)**. The target chip at **(4)** is unreachable because the algorithm keeps growing in a single direction.

and right until the columns containing the leftmost and rightmost target chips have been reached or no more suitable detours can be found.

In each recursion, outgoing edges from the current vertex are examined to find the subsequent horizontal bus on the adjacent chip towards the direction of growth. If the corresponding horizontal bus is found and it is not required for inserting events for other connections, the recursion continues, otherwise a vertical detour needs to be established. Several reasons can lead to the bus not being found, for example, prior allocations, blacklisting or it simply does not exist because the wafer boundary has been reached.

In case of a detour, all vertically reachable buses on the current chip are considered. The best possible option is determined by walking vertically for each option until the wafer boundaries at the top and bottom are reached or the path is blocked. Then, the number of horizontally reachable target chips in the direction of growth is individually counted for every crossbar switch along the way. Consistently, buses with mapped SPL1 repeaters are skipped. The evaluation does also not consider any further detours. The option that yields the maximum number of reachable targets, if any, is chosen to establish the detour accordingly. Subsequently, the growth continues in the HICANN row accessed by the detour.

Whenever a chip is reached that is in the same column as any of the targets, a vertical connection is established. Again, all vertical buses reachable via local crossbar switches are considered. The best option is picked based on a score. For each option, the score is initially set to zero and a vertical walk is started, both upwards and downwards.

When a target chip is encountered the score is increment by two, if less than 12 other routes are already competing for the same set of synapse drivers, otherwise by one. The scoring is useful to reduce the number of competing routes because up to 16 vertical buses share 14 synapse drivers, see Section 1.5.3.

The horizontal growth terminates when either all targets have been reached or no more viable detours can be found. The wafer routing ultimately terminates after all pending routes have been processed.

Runtime

In the worst case, connections have to be established from any chip to any other and detours have to be found for every horizontal recursion. The number of iterations is proportional to the number of SPL1 sources which in turn is proportional to the number of neurons N in the model. Clearly, the horizontal growth itself is bound by the number of buses in the system $|V|$. Detouring considers a limited number of vertical options and subsequently counts the horizontally accessible targets. During the evaluation, none of the buses is considered twice, thus limiting its complexity by another factor of $|V|$. The total complexity is therefore bound by

$$O(N, V) = N \cdot |V|^2 \qquad . \tag{6.6}$$

V is a hardware property, whereas N is defined by the network model. However, the number of chips and therefore V has to be chosen large enough to host all neurons. This means, V is typically proportional to N resulting in a cubic worst-case bound on runtime complexity. Nonetheless, this is only a conservative worst-case bound and networks with local connectivity are typically routed much faster. In any case, the implementation is sufficiently fast to route worst-case networks in reasonable time e.g., full-wafer homogeneous random networks have been mapped in approximately 100 s, see Section 8.4.1.

6.4.5 Comparison

Both implementations have been benchmarked in Section 8.4.1. In most cases, the *iterative horizontal growth* can establish more long-range connectivity, while also being faster. It has specifically been designed to cope with the shortcomings of the *shortest path* approach by reducing the horizontal bus allocations.

However, the shortest path approach is extremely generic and might be useful for e.g., inter-wafer routing or situations where discovery towards a single direction is not enough, like concave routing areas.

6.5 Synapse Driver Routing

Input:	An assignment of spike sources to vertical Layer 1 buses for each target chip.
Output:	An assignment of vertical Layer 1 buses, representing inbound connections, to synapse drivers.
Defects:	Blacklist of synapse drivers.

The synapse driver routing is concerned with the insertion of spike events into the synapse arrays, individually for each HICANN. The prior wafer routing has established Layer 1 connectivity such that inbound connections are assigned to vertical buses on target chips. Next, these vertical buses have to be connected to a primary synapse drivers to relay events into the synapse arrays. Furthermore, adjacent secondary synapse drivers can be connected sharing the input in order to map large address spaces or realize different STP parameters.

For experiments with modest input requirements per HICANN, multiple configurations may exist that equally well insert events from all inbound connections. For larger networks, however, synapse drivers are a limited resource. Every synapse driver can receive input from inbound connections either via one of 16 reachable vertical buses (see Section 1.5.3) or via an adjacent driver to the top or bottom. All these sources equally compete for exclusive access to the respective driver, making this a complex optimization problem. Additionally, inbound connections often require more than one driver in order to map all multiplexed spike sources or realize different STP parameter sets. At the same time, the maximum capacitive load allowed per Layer 1 bus segment constrains the number of connectible drivers, see Section 1.5.5.

Finding optimized assignments is important to avoid inhomogeneous synaptic loss due to locally rejected connections, which cause all synapses from all corresponding sources to be lost. Furthermore, the respective Layer 1 resources have been occupied in vain. The optimization is simplified by the fact that drivers on different sides of the synapse array can be accessed by a non-overlapping set of connections. Both sides can therefore be optimized independently.

A simpler version of the problem, where vertical buses are connectible to arbitrary drivers, can algorithmically be considered a Knapsack optimization problem, which is known to be NP-hard (Garey and Johnson, 1979). However, the problem at hand is even more complex. For example, vertical buses have access only to a subset of synapse drivers due to the select switch sparseness (see Figure 1.5). Additionally individual drivers might be blacklisted.

This chapter presents two synapse driver routing implementations, following two different approaches. Firstly, an efficient bin-packing heuristic (Dósa, 2007) and, secondly, a simulated annealing optimization is described.

6.5.1 Requested Synapse Drivers

As a starting point for the synapse driver routing optimization, the theoretical number of drivers required to realize all synapses is calculated per inbound connection. This number depends on the mapping of afferent neurons to the 6 bit address space, model projection properties and the actual connectivity between pre and postsynaptic neurons. The address space can be mapped according to the two stage decoding scheme explained in Section 1.5.4, while different STP parameters are more expensive to realize and require dedicated synapse drivers.

The access granularity of hardware synapses suggests to initially count the number of necessary half synapse rows. A half synapse row is defined as all synapses within one row receiving the same strobe signal, which corresponds to the A, B, C, D pattern in Figure 1.6. For any inbound Layer 1 connection r, model synapses from spikes sources within the same bin $b = (t_{\mathrm{e/i}}, 2\,\mathrm{MSB})$ can be implemented by hardware synapses in the same half synapse row. Where b bins spike sources with the same 2 most significant address bits and the same synapse type $t_{\mathrm{e/i}}$, which can either be excitatory or inhibitory. The number of necessary half synapse rows L for bin b, connection r, efferent neuron j and STP parameter set P_{STP} is given by

$$L_r(j, b, P_{\mathrm{STP}}) = \left\lceil \frac{1}{w_{\mathrm{j}}/2} \sum_{i \in b} n_r(i, j, P_{\mathrm{STP}}) \right\rceil \quad . \tag{6.7}$$

On the right-hand side, $n_r(i, j, P_{\mathrm{STP}})$ denotes the number of model synapses connecting afferent neuron i to efferent neuron j using the same set of STP parameters. In other words, the number of model synapses that can be realized as part of the same half synapse row is given by the sum over all sources within b. Dividing this number by half the physical width w_{j} of the target hardware neuron j, which determines the number of accessible synapse columns, and rounding the result to the ceiling yields the number of necessary half synapse rows $L_r(j, b, P_{\mathrm{STP}})$.

The effective number of necessary half synapse rows per bin over all efferent neurons is determined by the efferent neuron which receives the most input. Thus, to get the number of necessary synapse drivers, firstly, the maximum L_r of all efferent neurons j is determined and the results summed up for all bins b. However, every synapse driver has access to four half synapse rows, which reduces the effective number of necessary drivers. Furthermore, the result has to be rounded to the ceiling to respect the access granularity. So far, the STP parameter sets have been ignored. Every parameter set requires an independent set of drivers, therefore the total number of requested drivers D_r can be calculated by simply summing over all requested STP parameter sets:

$$D_r = \sum_{P_{\mathrm{STP}}} \left\lceil \frac{1}{4} \sum_{b} \max_{j} L_r(j, b, P_{\mathrm{STP}}) \right\rceil \quad . \tag{6.8}$$

If D_r drivers are allocated for r, theoretically all model synapses can be realized. However, the calculation cannot account for blacklisted hardware synapses. The synapses drivers have not yet been assigned and it is therefore not yet clear which hardware synapses can be accessed. Therefore, some model synapses can still be lost during the subsequent synapse assignment. To circumvent the issue for low input counts, additional synapse drivers can be requested. When D_r becomes larger than the capacitive limit allows, model synapses are inevitably lost.

Note that using a wide variety of different STP parameters has a strong influence on the number of necessary drivers. Thus, merging similar STP parameter sets in PyNN for projections with shared neuron targets is advisable.

Finally, the total number of synapses, represented by the inbound connection r, is determined in order to prioritize inbound connections that contribute many synapses. The synapses are counted according to

$$N_r = \sum_{j,b,P_{\mathrm{STP}}} \sum_{i \in b} n_r(i, j, P_{\mathrm{STP}}) \qquad . \tag{6.9}$$

6.5.2 Iterative Best Fit Driver Assignment

The iterative best fit algorithm is similar to the synapse driver routing presented by Fieres et al. (2008). The drivers are iteratively assigned to inbound connections in a first-fit decreasing fashion. This heuristic is known to yield close to optimal results for bin packing problems (Dósa, 2007). In case the overall amount of requested drivers per chip exceeds the number of available drivers, the problem can no longer be approximated by bin packing. Thus, the driver requirements for all inbound connections need to be normalized to the number of available drivers. Otherwise, allocating drivers for the first few connections in a first-fit fashion might use up all drivers leaving none for the remaining connections. Furthermore, assignments can not necessarily be allocated side by side due to the select switch sparseness, which fragments the synapse driver assignments and therefore the *bins* in bin packing. Consequently, some inbound connections may not be assignable to drivers and therefore all represented synapses are lost. In a second trial, inbound connections ruled out during the initial normalization stage are assigned to synapse drivers that are left from the first trial.

Normalization of Required Synapse Drivers

In case the total number of required synapse drivers exceeds the number of available drivers, the requested drivers have to be normalized to the number of available ones in order to further approximate the problem by bin packing.

The normalization is carried out independently for synapse drivers on the left and right side of the chip. Initially, the number of requested drivers D_r is collected for all inbound connections r on all 256 connectible vertical buses, half of which are on the neighboring HICANN. If the sum $D = \sum_r \min(D_r, D_{\mathrm{max}})$, where D_{max} is the maximum number drivers with shared input, does not exceed the number of available drivers, the drivers can be assigned without prior normalization. The number of available drivers

A is typically 112 minus the number of blacklisted drivers. Otherwise, the following normalization is applied.

$$D'_r = \begin{cases} 0 & \text{if } D_r = 0 \\ \max(1, \min(D_r, D_{\max}, \left\lfloor \frac{A \cdot N_r}{N_{\text{total}}} \right\rfloor)) & \text{else} \end{cases}, \tag{6.10}$$

with the number of model synapses N_r represented by the inbound connection r and the total number of model synapses N_{total} targeting neurons on the local chip.

After the normalization, it is possible that $K = \sum_r D'_r$ is larger than A because of the maximum function in Equation (6.10). In this case, the different D'_r are iteratively decremented in descending order of $\Delta_r = D'_r - A \cdot {}^{N_r}/_{N_{\text{total}}}$, if $\Delta_r > 0$, until $K = A$. Connections r that contribute only few synapses might end up with $D'_r = 0$.

It can also happen that K is smaller than A after the normalization, mainly for two reasons. Either if $D_{\max}$ times the number of inbound connections is less than A or the inbound connections represent only few synapses, but require many drivers because of e.g., different STP parameter sets. In the latter case, the assignments are increased in ascending order of Δ_r, if $\Delta_r < 0$, until either $K = A$ or $D'_r = \min(D_r, D_{\max})$ for all connections.

Driver Assignment

Initially, all inbound connections are ordered in a list according to their normalized driver requirements D'_r from many to few. Two vectors with 56 entries each are initialized, representing the synapse driver banks in the top and bottom synapse arrays. Where entry j of the first and second vector corresponds to the jth driver in the top and bottom synapse array, respectively. Entries keep also track of the availability of their respective driver. Initially, all except blacklisted drivers are available.

Next, the actual iterative assignment begins. In each iteration, the frontmost connection r is taken from the list. The algorithm considers all possible assignment options in order to find the best insertion point. For each inbound connection and its corresponding vertical bus up to 14 reachable drivers exist. Furthermore, an assignment of n drivers can be shifted around the insertion point p, as long as p is $\in [x, x+n)$ the drivers can be assigned.

First, all 14 options are checked whether they provide D'_r available adjacent synapse drivers. If more than one candidate exists, the first option in the smallest gap of available drivers is chosen in order to minimize fragmentation. An assignment for multiple drivers is then shifted such that the distance between this assignment and the closest other assignment is minimized to further reduce fragmentation. If no option with a sufficient amount of drivers exists, the option with the largest number of remaining drivers is chosen. Assigned drivers are marked as no longer available for subsequent assignments. If none of the options has any drivers left, the connections cannot be realized and the corresponding synapses are lost. The iterations terminate after all inbound connections in the list have been processed.

In a second assignment step, inbound connections that have been dropped during the normalization are scheduled for insertion. These connections are sorted by their original driver requirement D_r in descending order. Then, the same algorithm is used to assign the connections to the remaining synapse drivers. The algorithm terminates after the second assignment step finishes.

6.5.3 Simulated Annealing Driver Assignment

The second synapse driver routing implementation uses simulated annealing (Kirkpatrick et al., 1983) to optimize the local assignment of synapse drivers. Routing constraints are modeled by a cost function $E(s)$ that measures the quality of synapse driver assignments s. Constraints can easily be modified by changing the cost function.

The optimization starts with an arbitrary initial assignment s. Then, neighboring assignments s' of s are explored in each iteration. These neighbors are derived by a so-called propagator $K(s) = s'$. In case K is ergodic and neighboring assignments are explored indefinitely, the global optimum will eventually be reached.

Successor assignments s' with costs $E(s')$ lower than $E(s)$ are instantly accepted, whereas more expensive assignments are accepted with a reduced probability of

$$p(s, s') \sim \exp\left(\frac{\Delta E}{T}\right) \quad , \tag{6.11}$$

where $\Delta E = E(s') - E(s)$ denotes the cost difference between the candidate assignment s' and the current assignment s. T is the temperature of the system, which decreases over time, therefore accepting more expensive assignments becomes less likely. At some point the system *freezes* in its current local cost minimum. Accepting assignments with higher costs is generally important to avoid being trapped in local minima when only considering neighboring assignments.

Representation

Every HICANN has four banks of 56 synapse drivers, one located on each side of the synapse arrays in the top and bottom half of the chip. The synapse drivers on each side can be reached from 256 vertical buses, half of which are on the local and the other half on the adjacent chip. The banks on either side are non-overlapping resources and can thus be optimized independently.

The algorithm models the synapse driver banks on either side as an interval of drivers. Synapse drivers in the top and bottom cannot share inputs, they are therefore represented as two independent intervals $[0, 56)$ and $[56, 112)$ for the top and bottom bank, respectively. An assignment of an inbound connection r to a set of drivers can be expressed as an assignment of r to an subinterval of length $\min(D_r, D_{\max})$ in either $[0, 56)$ or $[56, 112)$. $D_{\max}$ is the maximum number of drivers with shared input and D_r is the number of requested drivers for r according to Equation (6.8).

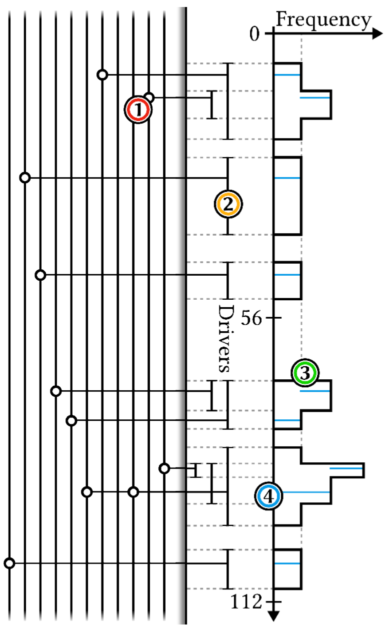


Figure 6.7: The simulated annealing synapse driver routing models the two banks of 56 drivers as intervals. Vertical buses **(1)**, featuring inbound connections, are assigned to drivers by assigning them to a subinterval within $[0, 56)$ or $[56, 112)$ **(2)**. The optimization tries to minimize the amount of overlapping driver assignments **(3)**. Ideally, every inbound connection r receives its necessary number of drivers D_r, according to Equation (6.8), without over-assigning any driver. Primary drivers directly receiving input from vertical buses are highlighted in blue. Configurations with overlapping primary drivers **(4)** are particularly expensive because only one of the connections can ultimately be realized, the others are lost.

To support efficient access to existing synapse driver assignments and modifications thereof during the optimization, a `boost::ICL` interval tree data structure is used (Boost, 2010). Figure 6.7 illustrates the mapping from vertical buses, featuring inbound connections, to driver intervals. Every interval has an up-to-date count of overlapping assignments. A count larger than one indicates more than one connections are assigned to the same driver. The simulated annealing tries to minimize the overlap by reassigning overlapping assignments. Blacklisted drivers can be handled as fixed assignments with a high overlap value. During the optimization process, other assignments are moved away from these drivers to resolve these expensive configurations. In a final post-processing step, overlapping assignments are resolved and assignments to blacklisted drivers are dropped in order to produce valid hardware configurations.

Propagation

The optimization performance strongly depends on the quality of candidate assignments. The propagator therefore guides the exploration of candidates by randomly reassigning inbound connections for highly overlapping drivers rather than reassigning connections completely at random.

In each annealing step, the propagator choses randomly either the top or bottom driver interval and identifies the subinterval with maximum overlap. One of the inbound connections assigned to this subinterval is chosen at random and is randomly reassigned to a different accessible subinterval. Subintervals are accessible if a select switch exists such that the corresponding synapse driver range can be accessed. Successive assignments are neighbors in the sense that most connection assignments remain unaltered. In other words, the propagator tries to improve the current assignment by iteratively reducing the overlap for the most overlapping drivers.

Optimization

Initially, all inbound connections are randomly assigned to accessible subintervals. The length of the subinterval for the connection r is given by $\min(D_r, D_{\max})$. D_r and $D_{\max}$ denote the number of necessary synapse drivers for r and the maximum number of connected synapse drivers, respectively. Iteratively, the connections are reassigned by the propagator described above, for a configurable number of iterations, set to 5000 by default, or until all overlaps are resolved. For chips with a small number of inputs, the optimization typically stops after a few iterations. The temperature decreases over time and the system freezes in a locally optimal assignment of minimal cost. The cost function is defined as

$$E(s) = O(s) + c \cdot O_{\text{primary}}(s) \qquad . \tag{6.12}$$

$E(s)$ denotes the cost of the assignment s and $O(s)$ is the corresponding driver overlap according to Figure 6.7. The second term on the right-hand side $O_{\text{primary}}(s)$ accounts for overlapping primary synapse drivers. Overlapping primary drivers receive input directly from more than one vertical Layer 1 bus. Such assignments are particularly expensive because all except one of the corresponding connections have to be dropped during the post-processing to produce valid hardware configurations. The constant c allows to scale the relative cost of both contributions and is set to 20 by default.

During the course of optimization, the best assignment so far is always remembered. Exemplarily, the evolution of cost during an driver assignment optimization is shown in Figure 6.8. After optimization, the best solution is picked for the final post-processing to generate a valid synapse driver configuration.

Post-Processing

The final post-processing has to produce a valid synapse driver configuration from the simulated annealing interval assignment, which may contain over-assigned drivers. Such assignments are often unavoidable for high input counts, despite the propagator constantly trying to resolve these assignments. The final post-processing ensures that synapse drivers are assigned only once.

Firstly, for overlapping primary drivers, all except the inbound connection representing the largest number of synapses are rejected. Assignments to blacklisted drivers are rejected as well. Next, the connectivity between primary and secondary drivers has to be established. Currently, only a simple approach is implemented. Therefore, assigned inbound connections are sorted in descending order by their respective number of synapses. Successively, connections are taken from the front of the list. For each connection, connectivity to still reachable secondary drivers is established according to their respective annealing intervals. After processing all entries, a valid synapse driver

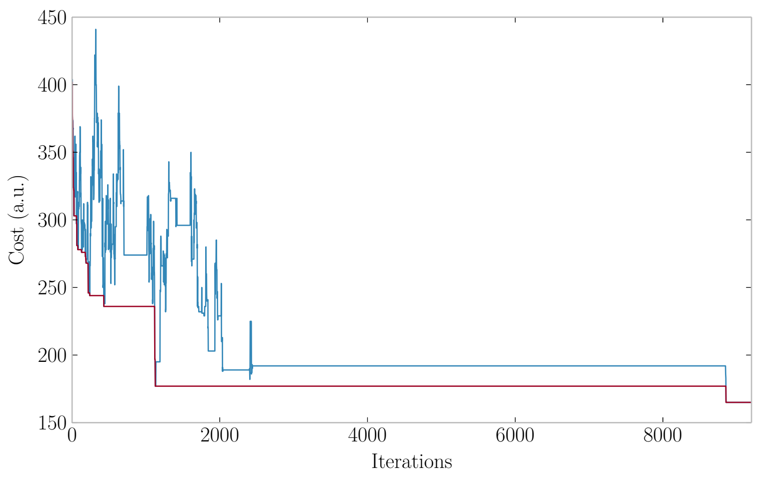


Figure 6.8: Synapse driver routing optimization via simulated annealing. The optimization starts with a random initial assignment, better configurations are iteratively explored. The cost for the current and best solution are shown in blue and red, respectively. Over time the temperature decreases and the system freezes, until after about 8800 steps the best solution is found. After the optimization finishes, the best solution is used to establish the synapse driver configuration.

routing has been established. This simple, greedy approach currently limits the quality of configurations because connections towards the end of the list end up with little or no secondary drivers. A prior normalization, similar to the one described in Section 6.5.2, is expected to improve the final configuration, however, it is not yet implemented.

Runtime

The runtime for both implementations is bound by a fixed maximum problem size, since only chip-local resources are assigned. Furthermore, assignments for drivers on either side of the synapse arrays can be handled independently because they share no common inputs. In the worst case, assignments for 2×128 inbound connections have to be optimized. Therefore, the runtime complexity of the iterative best-fit is bound by a constant and depends linearly on the number of annealing steps N for the simulated annealing. Empirically, both implementations are fast, however, the simulated annealing over 5000 iterations typically takes 20 % longer Section 8.4.1. .

Moreover, the implementations carry out the optimization for multiple chips is in parallel to reduce the overall time spent on the synapse driver routing. However, capacitive bus limits introduce a dependency between adjacent chips. The limits are currently only enforced if the prior wafer routing has set up exclusive vertical buses for adjacent chips receiving shared input, which is usually only a minor limitation because a sufficient number of vertical buses exist.

6.6 Synapse Array Routing

Input: Synapse driver routing for each chip.

Output: A mapping of the spike source address space to synapse rows and an assignment of model to hardware synapses.

Defects: Blacklist of synapses.

The synapse array routing completes the connectivity setup. Firstly, the Layer 1 address space of spike sources is mapped to half synapse rows according to the two-stage decoding scheme (see Section 1.5.4) and, secondly, model synapses are assigned to hardware synapses.

6.6.1 Synapse Row Assignment

In the first step, 2 bit synapse driver decoder configurations have to be established in order to assign inbound spike sources to half synapse rows. The half synapse row address decoding scheme is indicated by the A, B, C, D pattern shown in Figure 1.6. Synapses within the same bin $b = (t_{\mathrm{e/i}}, 2\,\mathrm{MSB})$, according to Section 6.5, can efficiently be realized as part of the same half synapse row. Whereas sources in different bins require different synapse driver decoder configurations.

The implementation assigns half synapse rows to the binned inbound connections according to the relative number of model synapses they represent, in order to minimize the synaptic loss during the subsequent synapse assignment. With this method, all model synapses for inbound connections that received their requested number of drivers in the prior synapse driver routing can theoretically be realized if no synapses are blacklisted.

Notably, the number of accessible half synapse rows depends on the size of the target hardware neuron. For small hardware neurons, without horizontally interconnected circuits, only every other half synapse row can relay events to it. Furthermore, if the horizontal neuron extent is odd, the number of hardware synapses connected to strobe signal A and B as well as C and D differs and depends on the horizontal neuron offset. It is therefore advisable to use hardware neuron sizes that are multiples of 4, which results in even horizontal neuron extents. Otherwise, synapses might be lost during the following assignment.

6.6.2 Synapse Assignment

In the second step, model synapses are assigned to hardware paragons in the order they appear in the model description. Meaning that synapses to efferent neurons earlier in the model description have a higher static priority and are more likely to be realized. The order only influences which model synapses are realized, the absolute amount of realizable synapses is not influenced.

Initially, all synapse weights and decoders are set to zero and `0001`, respectively, as explained in Section 6.2, to minimize the leakage conductance onto neuron membranes for unused synapses. Subsequently, all projections that are part of routed inbound connections with local targets are iterated in the order they appear in the model description. For every projection, synapses are mapped in a target-first manner, which means that synapses to an efferent neuron i are realized before synapses to an efferent neuron j if $i < j$. Consequently, all synapses in bin b and to the given target neuron can efficiently be marked as lost whenever no more hardware synapses are left for this bin-target combination.

For every model synapse, firstly, its corresponding bin b is determined and used to lookup the next, yet unassigned, hardware synapse for the corresponding target neuron in a table. Every table lookup takes only constant time. In case the hardware synapse has not been blacklisted, the model synapse is assigned and the algorithm continues with the next model synapse. Whenever a hardware synapse is blacklisted, the next one is looked up until either a usable one is found or no more accessible synapses are left. In order to get to the next available hardware synapse, the table entry is updated after each lookup. Therefore, the column index of the table entry is incremented by two. Afterwards, if the table entry points to a column that belongs to a neighboring neuron, the row index is set to the next, yet unassigned, half synapse row for b and the column index is reset to the leftmost synapse column of the target neuron. If no more accessible half synapses rows exist, the remaining synapses in b are marked as lost and the iteration continues with synapses belonging to other bins.

Note that the translation of model conductances into digital weights is described in Section 6.7.

Runtime

The runtime of the implementation scales linearly with the number of model synapses because all synapses have to be processed regardless of their realization. The weight lookup in constant time ensures that each hardware synapse is handled only once and remaining hardware synapses are accessed efficiently. Moreover, the implementation conducts the synapse array routing for multiple HICANNs in parallel. On the bottom line, the synapse array routing is typically fast, the overall mapping time is dominated by the wafer routing, see Section 8.4.1. This leads to mapping runtimes that are almost independent from the connection probability of the random network instance, see Figure 8.8.

6.7 Parameter Transformation

Input: Neuron placement, synapse assignment; Parameter transformations including scaling, calibration data and limited ranges.
Output: Sets of digital and analog parameters for the chips, neurons and synapses.
Defects: None

The final step towards setting up the hardware is to specify all remaining analog and digital parameters. Some are set to defaults to establish an operating regime for the circuitry. Others have to be chosen to resemble the model as closely as possible. The actual parameter transformation is not part of the mapping framework, but has been moved into the Calibtic parameter framework. Thus, other workflow components can use the model transformations and calibration data alike, e.g., StHAL (see Section 4.3) uses Calibtic to automatically correct recordings of analog voltage traces.

Parameters are transformed in two steps. Firstly, they are scaled from the biological model into the hardware domain. Afterwards, a circuit specific correction is applied, which is referred to as calibration. The scaling from model into hardware domain as well as the following correction are both implemented within the Calibtic framework. Here, the basic scalings of voltages, times, conductances and currents are presented for the sake of completeness. They have mostly been described by Schwartz (2012) alongside the generation of calibration data, which is outside the scope of this thesis.

6.7.1 Parameter Framework

The Calibtic parameter framework is an extensible `C++` library that provides storage agnostic access to calibration data. Calibtic has been established by the author as part of this thesis. At the time of writing, it has been used to implement the analog readout calibration as well as a redesigned calibration for HICANN analog parameters.

Notably, Calibtic stores the transformation type together with the actual data, describing the complete translation process in a context insensitive manner. Therefore, transformations can be changed or updated as needed over time, while old data sets remain consistent and applicable without changing the implementations of the transformation routines. This is a major improvement over former approaches where changing e.g., the data acquisition side could invalidate existing transformations or render application thereof inconsistent.

Moreover, Calibtic can be used for both, the acquisition and application of calibration data. Data acquired in low-level measurements via HALBe and StHAL can be stored using the `Python` interfaces of Calibtic. On the application side, PyHMF is well integrated into Calibtic such that analog model parameters can directly be translated into hardware floating-gate values.

A dynamic plug-in system adds flexible storage options. As of May 2014, storage backends exist for `XML`, `JSON` and mongoDB. The latter two are realized by means of a generic mongoDB `boost::serialization` (Ramey, 2004) archive written by the author.

6.7.2 From Biology to Hardware

The first step of the parameter transformation maps biological model parameters to hardware parameters. For example, typical biological voltages are in the order of a few millivolts, whereas integrated Complementary Metal–Oxide–Semiconductor (CMOS) transistors operate in the range of a few volts. Furthermore, due to smaller capacitances

C and higher conductances g in the hardware system, the intrinsic time constants, given by $\tau = {}^{C}\!/_{g}$, are scaled by a factor of 10^3 to 10^5 compared to biological real time.

After the parameters have been scaled to the appropriate hardware range, a calibration is applied in a second step to account for individual circuit variations. The relative parameter change is typically small compared to the initial scaling. Detailed descriptions of the actual calibrations are given by Schwartz (2012).

Time and Time Constants

Times and time constants are linearly scaled by the speedup factor α_{acc} according to

$$\tau_{\text{scaled}} = \frac{\tau_{\text{model}}}{\alpha_{\text{acc}}} \qquad . \tag{6.13}$$

Note that the effective speedup can be controlled and is subject to calibration. As mentioned before, the system is designed to allow acceleration factors from 10^3 to 10^5. By default, calibrations target a factor of 10^4.

Voltage Ranges

Voltages are transformed linearly from millivolts in the model to volts in the hardware domain, according to

$$v_{\text{scaled}} = \alpha_v v_{\text{model}} + v_{\text{shift}} \qquad . \tag{6.14}$$

Here, α_v is a linear scaling factor and v_{shift} an offset that can be used to shift voltage ranges relative to one another. Most model dynamics depend on voltage differences, thus the common shift takes no direct effect. However, voltage parameters can be shifted relatively to the shared V_{reset} parameter. The parameters α_v and v_{shift} can be chosen freely to map a given dynamic range. By default, α_v and v_{shift} are set to 10 and 1200 mV, respectively, to map a biological range of $(-120, 0)$ mV to $(0, 1.2)$ V in hardware.

The voltage transformation applies for all voltages including the reversal potentials E_{l}, E_{e}, E_{i}, the reset potential V_{reset}, the spike initiation potential V_{thresh} and the spike detection threshold Θ. The only exception is the AdEx slope factor Δ_{t}, which is scaled without being shifted.

Membrane Capacitance

To support a wider range of possible speedup factors in hardware, the membrane capacitance can be configured to either 2.16 pF or 0.16 pF. Typically, the larger capacitance is chosen except for high speedup factors of around 10^5. The total membrane capacitance for interconnected neurons is the sum over all interconnected circuit capacitances, following

$$C_{\text{m}} = \sum_{i}^{N} C_{\text{m,i}} \qquad , \tag{6.15}$$

where N is the number of connected neuron circuits and $C_{\mathrm{m,i}}$ their individual capacitances. In principle, any combination of capacitances is possible, however, as of May 2014, no transformations existed for interconnected neurons.

Conductances

The time constant τ_{m} is responsible for the leaky dynamics of a neuron and is controlled by the leakage conductance g_{leak} according to

$$\tau_{\mathrm{m}} = \frac{C_{\mathrm{m}}}{g_{\mathrm{leak}}} \quad . \tag{6.16}$$

The effective membrane capacitance C_{m} has been introduced above. Using Equation (6.13), the previous equation can be rephrased as

$$\begin{aligned} g_{\mathrm{leak,scaled}} &= \frac{C_{\mathrm{m,scaled}}}{\tau_{\mathrm{m,model}}} \alpha_{\mathrm{acc}} \\ &= \frac{C_{\mathrm{m,scaled}}}{C_{\mathrm{m,model}}} \alpha_{\mathrm{acc}}\, g_{\mathrm{leak,model}} \quad . \end{aligned} \tag{6.17}$$

Where α_{acc} is the acceleration factor, $\tau_{\mathrm{m,model}}$ is the model membrane time constant and $g_{\mathrm{leak,model}}$ is the model leakage conductance. Further, $C_{\mathrm{m,scaled}}$ and $C_{\mathrm{m,model}}$ are the total hardware and model membrane capacitance, respectively.

This g_{leak} transformation can identically be applied to other conductances, like synaptic efficacies and the adaption coupling parameter a, which leaves us with the generic expression

$$g_{\mathrm{scaled}} = \frac{C_{\mathrm{scaled}}}{C_{\mathrm{model}}} \alpha_{\mathrm{acc}}\, g_{\mathrm{model}} \quad . \tag{6.18}$$

Currents

Currents are transformed according to Ohm's law using the previous transformations for voltages in Equation (6.14) and conductances in Equation (6.18)

$$I_{\mathrm{m}} = g \cdot U = \frac{C_{\mathrm{scaled}}}{C_{\mathrm{model}}} \alpha_{\mathrm{acc}}\, \alpha_{v} \quad . \tag{6.19}$$

Here, I_{m} denotes a model current, g a conductance and U a voltage. This transformation is used for the AdEx adaption current b and the external current stimuli.

6.7.3 Shared Parameters

This section briefly outlines the transformations for parameters that are shared by multiple circuit instances and require therefore special considerations.

Shared Neuron Parameter V_{reset}

Multiple neurons share a common V_{reset}, which is in fact the only shared AdEx model parameter. All other analog neuron parameters can be set and calibrated individually for every neuron. Therefore, v_{shift} in Equation (6.14) can be tuned such that all voltage differences are set correctly relative to V_{reset}, while the model dynamics remain the same.

Theoretically, there are 4 different shared floating-gate voltages for V_{reset}. They are assigned to neurons with odd and even index in the top and bottom synapse array. However, only a single value can effectively be used because larger hardware neurons typically span neuron circuits connected to all four instances of V_{reset}.

Row-Wise Synaptic Efficacies

The wafer system implements conductance-based synapses, which have to be scaled according to Equation (6.18), where the ratio of capacitances $C_{\text{scaled}}/C_{\text{model}}$ depends on the hardware neuron size. The effective synaptic conductance g_{syn} is the product of a driver-wise base-conductance $g_{\text{max}}(j)/g_{\text{div}}(j)$ and an individual 4 bit digital weight g_{digital}, thus

$$g_{\text{syn}}(i,j) = \frac{g_{\text{max}}(j)}{g_{\text{div}}(j)}\, g_{\text{digital}}(i,j) \qquad . \tag{6.20}$$

Where i and j denote the ith synapse in the jth column. The row-wise shared parameter g_{max} can be chosen individually from a set of four configurable floating-gate values and g_{div} is a 4 bit divider that can be set to a value in the interval $[1, 16]$.

Transformations should set g_{max} and g_{div} such that the maximum digital weight corresponds to the strongest model synapse in that row in order to maximize the dynamic range. Experiments involving STDP might choose a higher g_{max} in order to make a digital weight of eight coincide with the maximum scaled model conductance. Therefore, digital weight can be potentiated upon learning. However, g_{max} is currently set to a small default value because no calibration exist yet and the synaptic input circuits saturates already for relatively small values (see Millner, 2012, Section 3.6.4).

After choosing g_{max}, the digital weights are set such that the resulting conductance resembles the scaled model conductance as closely as possible. Weights have a 4 bit resolution, thus 16 discrete values are accessible. In order to minimize average distortion, weights are clipped stochastically. This means, for a target conductance g_{scaled}, the closest two accessible conductances g_+ above and g_- below are picked from a list of the 16 possible values via branch and bound (Land and Doig, 1960). Finally, g_- is picked with a probability of $p = \frac{(g_{\text{scaled}} - g_+)}{(g_- - g_+)}$ and g_+ otherwise.

6.7.4 Spike Sources

PyNN *SpikeSourceArray* and *SpikeSourcePoisson* spike sources are implemented via FPGA spike playback over the Layer 2 off-wafer network, see Section 1.7. Layer 1 source addresses have been established during the spike input placement, as explained in Section 6.3. Spike trains can therefore be generated by using the respective address and translating spike times according to Equation (6.13). For spike arrays, the spike times are simply provided by the model description, whereas spike times for Poisson sources of rate ν have to be constructed. According to Heeger (2000), time is discretized into small intervals of $\delta t = 1\,\text{ms}$. For each interval, a uniformly distributed real random number $x \in [0, 1]$ is generated, when $x < \nu \cdot \delta t$ a spike is set to occur in the corresponding interval.

Furthermore, a small, fixed offset is added to every spike to postpone the actual experiment onset. This offset is currently set to 20 µs (HTD) to allow repeater DLLs to lock reliably.

Finally, Layer 1 spike transmissions have to be augmented with address 0 events, as explained in Section 1.5.5. For recurrent neuron connections, events from the chip-local background generators can be used. The necessary merger configuration has been established during the merger routing, described in Section 6.2. It is therefore sufficient to simply enable all background generators. External spike source on the other hand require dedicated address 0 events as part of the FPGA playback stream. In both cases, spikes with a fixed inter spike interval (ISI) of 30 µs in the HTD are used according to Koke (2014). However, an FPGA timing issue prevents reliable locking for external spike inputs with the current firmware. This issue is discussed in more detail in Section 8.3.3.

6.7.5 Current Sources

A periodic step current can be applied to a single neuron per HICANN by using the floating-gate controller to replay a current course of 129 values. The duration of one period is $T = 129 \times \delta t$, where $\delta t = {}^{4}\!/\!{}_{f}$ and typically $f = 100\,\text{MHz}$ (HTD). PyNN current sources are translated according to Equation (6.19). In cases where more than one current source is requested per chip, only the first can be realized.

Runtime

The parameter transformation is straight forward and does not involve any complex algorithms. Ultimately, the runtime depends on the implementations in Calibtic. Most HICANN parameters are determined in constant time, rendering the implementation suitable for large networks. Furthermore, marocco performs the parameter transformation for multiple chips in parallel.

6.8 High-Level Mapping Interface

The PyMarocco interface provides user-friendly access to the marocco framework to e.g., guide the mapping or control parameters of the presented algorithms. The `C++` interfaces are automatically wrapped to `Python` using the code-generation flow described in Section 3.2.3. PyHMF integration and IPC functionality is implemented using the `MetaData` mechanism described in Section 3.2.4.

A PyMarocco object can be passed to the PyNN `setup()` call to register the `MetaData` instance. When the experiment is triggered, the object is automatically forwarded to the marocco process and after completion an updated PyMarocco instance is sent back to the user, which provide access to mapping outcome. The following listing briefly illustrates the use of the PyMarocco interface.

```
marocco = PyMarocco()
marocco.placement.setDefaultNeuronSize(8)
marocco.defects.chip = [ HICANNGlobal(Enum(42)) ]
# ...
pynn.setup(marocco=marocco)
# ...
```

Exemplarily, the default hardware neuron size is set to 8 circuits and HICANN 42 is marked as defect to be omitted during the mapping.

Accessing Mapped Synapses

PyMarocco provides convenient access to the mapped realization of synapses. This mapping reflects synaptic loss and distorted efficacies imposed by limited parameter ranges, precision and digital discretization. This feature has been used to set up the follow up software simulation presented in Section 8.6. The following small code listing demonstrates the interface.

```
p = pynn.Projection(src, trg, AllToAllConnector())
# ...
pynn.end()
weights = marocco.stats.getWeights(p) # get synapse mapping
```

Here, `weights` is a connection matrix containing the synapse realizations for the PyNN projection `p`.

6.9 Routing Visualization

RoQt is an interactive `Python QT` application (Nokia, 2009) that visualizes the Layer 1 routing as well as the synapse driver assignment. Visualizations can also be exported as pixel and vector graphics, an example is shown in Figure 6.9.

Layer 1 routes are highlighted in bright colors, such that connections can be traced from an SPL1 source to the corresponding targets. Used repeaters, crossbar and select switches are highlighted accordingly. Furthermore, the target-side synapse driver routing is visualized.

The visualization enables users to quickly access the hardware realization of their network model. Therefore, mapping bottlenecks causing distortions, like synaptic loss,

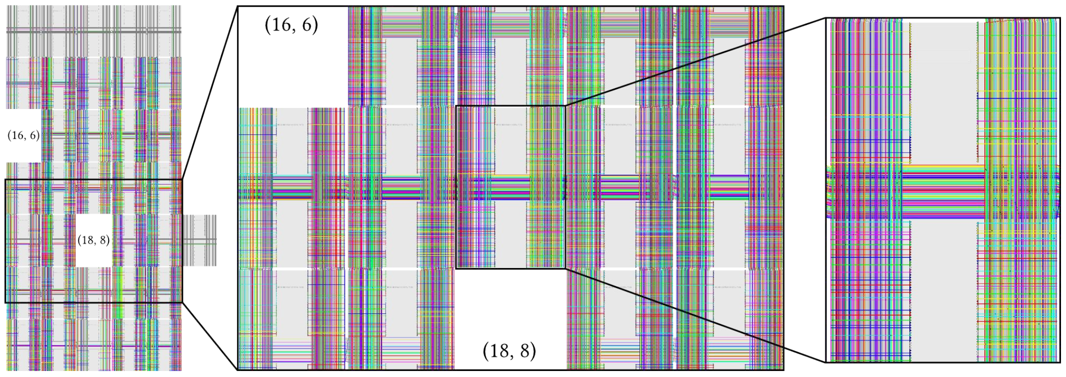


Figure 6.9: RoQt visualization of the layer 2/3 attractor model, introduced in Section 8.5, with 10 hypercolumns of 10 minicolumns each. The default neuron placement has been used to distribute the neurons. HICANNs $(X, Y) = (16, 6)$ and $(18, 8)$ have been blacklisted and are therefore omitted. Layer 1 connectivity and the synapse driver routing are indicated by colored buses and drivers, respectively. Enabled crossbar and select switches are shown as black circles (not visible). The interactive application allows users to zoom in and manually track individual Layer 1 connections. Here, three cutouts at different zoom levels are combined for demonstration purposes.

can be identified quickly. Losses are typically caused by either Layer 1 or local driver congestion. The former might be improved by using a more efficient manual placement strategy, whereas the latter can be attenuated by using larger hardware neurons for either all or specific target populations.

6.10 Summary

In this chapter, the marocco framework has been presented. It provides a complete and working set of implementations following the feedforward mapping flow introduced in Section 5.3. A special focus during development has been set on efficient and scalable algorithms. Wherever appropriate, shared-memory parallelization is used to further speed up the individual mapping steps. In terms of algorithmic complexity, the mapping flow is dominated by the wafer routing, e.g., the *Iterative Horizontal Growth* approach scales cubically in the number of neurons as long as the placement minimizes the active wafer area. However, this is only a worst-case bound by a smaller amortized complexity. For example, the time necessary to route networks with preserved local connectivity grows linearly in the number of neurons because routing individual sources is done in constant time. Nonetheless, mapping rates beyond 10^5 synapses per second have been achieved for worst-case, full-wafer homogeneous random networks, see Chapter 8.

A comprehensive discussion on mapping performance, features and future development is given at the end of this thesis.

7 Handling Hardware Defects

Wafer-scale integration is the key technique for realizing the high bandwidth, on-wafer Layer 1 network necessary for large-scale neural network implementations. However, random defects caused by manufacturing impurities inevitably remain in the system. Handling such defects is of utmost importance for the reproducibility of experiments on the HMF. The mapping implementations, presented in Chapter 6, have therefore been designed to automatically work around defects for a wide variety of components on different levels. This also means, large neural networks can be efficiently implemented on arbitrary wafer instances.

This chapter introduces the ReDMan framework, which handles the creation and management of defect maps. This provides an interface to integrate the results of low-level hardware measurements into the mapping framework and therefore control the mapping of neural networks. As part of the ReDMan framework, an automated classification for the detection of unreliable synaptic transmissions has been developed. This implicitly verifies the availability and interoperability of many HICANN components. However, the classification is currently complicated by the overly sensitive synaptic input circuits, as described by Millner (2012). The chapter closes with an conservative estimate of the actual synapse defect rate, showing that most synapses and most other components are working as intended, thus available for hardware experiments.

The software presented in this chapter has been developed by Johann Klähn and the author. The measurements have been conducted and published by Klähn (2013) under the supervision of the author.

7.1 Defect Management

The ReDMan framework provides the necessary means to persistently store and manage defect maps for a wide range of hardware imperfections. While, implementing suitable workarounds for known defects lies within the responsibilities of the mapping. Thus, ReDMan defines a data-oriented interface to integrate low-level measurements into the mapping process and therefore control the realization of neural networks for individual wafer instances.

Resources can be organized either individually or in hierarchies. In the latter case, high defect rates on a lower level can be represented more efficiently as a single defect on a higher level. Further, defects can be stored in either whitelists or blacklists.

Which representation is more suitable depends on the expected defect rate. A plug-in system, modeled after the Calibtic calibration framework, see Section 6.7.1, provides dynamically loadable and flexible storage options. Implementations exist for `XML` and mongoDB.

7.1.1 Interface

The ReDMan framework is a small, extensible `C++` template library. New resources can be added conveniently by means of inheritance.

Even though calibration and defect management have different use cases and maintain different kinds of data, it is reasonable to assume that data for both is acquired in the same process. Calibration routines are mostly developed in `Python`, raising the requirement for ReDMan to provide `Python` bindings as well. Auto-generated bindings (see Section 3.2.3) are provided to make defects accessible for both, the acquisition of calibration data as well as low-level interactive experiments. The following code listing briefly outlines the ReDMan interface.

```
# load mongoDB storage plug-in
db = redman.loadBackend(redman.loadLibrary('libredman_mongo.so'))
db.init() # connect to DB

wafer = redman.Wafer(db, Coordinate::Wafer(5)) # load wafer 5

hicanns = wafer.hicanns()
hicanns.disable(Coordinate::HICANNOnWafer(X(16), Y(10)))

h = hicanns.find(Coordinate::HICANNOnWafer(X(20), Y(7)))
nrns = h.neurons()
nrns.disable(Coordinate::NeuronOnHICANN(Enum(8)))

wafer.commit() # update DB entry
```

Here, the mongoDB database plug-in is loaded and a connection is established to the default database instance. Then, HICANN $(X, Y) = (16, 10)$ and neuron 8 on HICANN $(20, 7)$ are marked as defect and, finally, the updated defect map is written back to the database.

7.1.2 Defect Granularity

In order to handle defects efficiently, logical components have to be identified that can be worked around by an appropriate configuration. Looking at spatial scales, the transistor level is too detailed for handling defects efficiently. The wafer level on the other end is too coarse, e.g., a limited number of defect Layer 1 buses can be tolerated and simply avoided during the routing process. A list of logical components that can algorithmically be handled by the mapping process is given in Section 5.2.3.

Organizing defects in hierarchies helps to establish workarounds more efficiently during the mapping, e.g., a defect chip can either be handled as a lengthy list of individual components or as single defect chip. Moreover, omitting working components located in areas of high defect rates can yield better overall configurations. For example, a chip where only few synapse drivers are available might still have many working neurons. However, placing model neurons to this chip inevitably results in high synaptic loss.

7.1.3 Defect Description

Building a defect map for all logical components in the HMF raises the question of how to characterize and represent defects. In some cases, a component might work reliably 90 % of the time. Depending on the actual experiment it might be acceptable to tolerate rare glitches. Using these extra resources might even benefit the model performance. However, a general-purpose mapping should avoid such components by default and focus on experiment reproducibility.

The handling of defects is complicated by the fact that the reliability of many components depends on their respective configuration. Therefore, measuring component reliability for a single set of parameters and skipping components below a certain threshold during the mapping is insufficient. Even if for every component an operational range can be found, the mapping might still not be able to establish configurations using all components because many hardware components share parameters. Whenever the intersection of operational range show parameters is empty. For the sake of simplicity, defects are captured as binary maps alongside the parameters that have been used during the data acquisition. This simple but efficient representation allows a coherent interpretation of the availability of components. The mapping then uses the defect map that is most suitable for the intended target set of parameters.

7.2 Synapse Measurement

This section introduces an automated classification of spike transmission reliability in order to efficiently generate comprehensive defect maps for a large number of hardware synapses. Synapses have been chosen as the starting point for the automatic integration of defect data into ReDMan framework because synapses are fundamental for any neural network implementation. Furthermore, demonstrating the correct operation of synapses implicitly verifies the functionality of many other components. In fact, most components are involved in the process of spike transmission, including neurons as analog spike detectors. As a matter of fact, the results are mainly limited by the oversensitivity of the synaptic input circuits regarding the synaptic time constant V_{syntc}. The results presented in this chapter should therefore be considered a lower bound on the actual availability of hardware synapses.

7.2.1 Signal Pathways

The HICANN chip provides several options to connect chip-local background generators to neurons and therefore to implement synaptic signal pathways. Firstly, neurons can receive input via two independent synaptic input circuits from any synapse driver in the corresponding synapse array. Secondly, background generator events can be routed to every synapse driver on the local chip via four different vertical buses, resulting in a total of eight possible signal pathways. The synapse measurement uses all eight possible pathways in order be more robust against single component failure and to help identifying failing components. For example, if only one or two of the pathways fail to transmit spike signals, the problem is likely caused by one of the Layer 1 buses rather than the synapse itself.

Moreover, using only chip-local resources allows to conduct the measurement for several chips in parallel to speed up the synapse evaluation for a full wafer setup. However, the presented implementation has not yet been parallelized.

7.2.2 Analog Parameters

The synapse measurement identifies successful spike transmissions as PSPs in analog neuron recordings. Therefore, the evaluation depends on the analog properties of the neurons. However, no neuron calibration existed that accounts for synaptic input circuits. Consequently, the default HALBe (see Section 4.2) parameters have been used for all neurons. A complete list of parameters is provided in Appendix A.2. Most of the default parameters are sensible, however, the synaptic input circuits are overly sensitive regarding the synaptic time constant V_{syntc}, which severely limits the spike detection capabilities. This issue has previously been reported by Millner (2012). The influence of V_{syntc} on the classification performance has been studied in a follow up measurement presented in Section 7.2.6.

7.2.3 Automatic Classification

The classification is carried out on membrane recordings from neurons that are used as analog spike detectors. Measuring membrane traces for all 1.15×10^5 synapses per HICANN for all eight synaptic pathways results in over 9×10^5 individual membrane recordings per chip. An automated classification is therefore necessary in order to deal with the amount of recordings and in particular to generate comprehensive defect maps for the HMF

The implemented classification had to overcome two issues. Firstly, it needed to be robust against varying PSP shapes, which are caused by the uncalibrated synaptic input circuits. Secondly, no reliable trigger mechanism has been available causing any membrane recording to be arbitrarily shifted in time.

Using periodic spike signals both issues can be resolved by a classification in the frequency rather than the time domain. Furthermore, the spectrum can be reduced to a narrow range, which corresponds to the expected background generator rate, to simplify the classification.

Background Generators

The background generators are set to elicit periodic spikes every 500 clock cycles. Given a hardware clock frequency of $f = 100\,\text{MHz}$ a rate of $f_{\text{PSP}} = 0.2\,\text{MHz}$ is expected. The resulting ISI of 5 µs and 50 ms biologically is sufficient to clearly distinguish consecutive spikes because the membrane time constant is typically shorter such that the neuron can fully recover in between spikes.

Membrane Recordings

A preparatory study has shown that a recording of 5 PSPs is a reasonable compromise between a clear frequency contribution to the spectrum on the one hand and fast analog voltage recordings on the other. The recording time is therefore set to 25 µs, which corresponds to 2400 samples of the used ADC (HBP SP9, 2014). A single Fourier transformation for this number of samples takes on average 135 µs on an Intel Core i3 running at 3.07 GHz and is thus sufficiently fast to be performed on a large number of recordings.

Classification Criteria

Within a frequency range of 0.1 MHz to 1.25 MHz amplitudes above the 0.85 quantile are used for classification. Whenever these frequencies meet the following criteria, the recording is classified as a successful spike signal transmission.

1. The lowest amplitude matches (±40 kHz) the frequency of the background generator.
2. All amplitudes are harmonics of the background generator frequency.
3. The mean of the difference between successive amplitudes is negative.

Figure 7.1 exemplifies the classification process for a failing and a working signal transmission.

Error Estimate

A test set of 1200 reference membrane recordings has been manually classified to evaluate the classification error of the automated procedure. The algorithm produced 2 false-positives and 26 false-negatives on the test set. This is a total error rate of 2.3 %. False-negatives lead to more synapses being blacklisted and are of lesser concern. Whereas, false-positives threaten experiment reliability. They have found to be below 0.17 % for the test data. Finally, all membrane recordings are archived for future reevaluation using e.g., updated classification criteria.

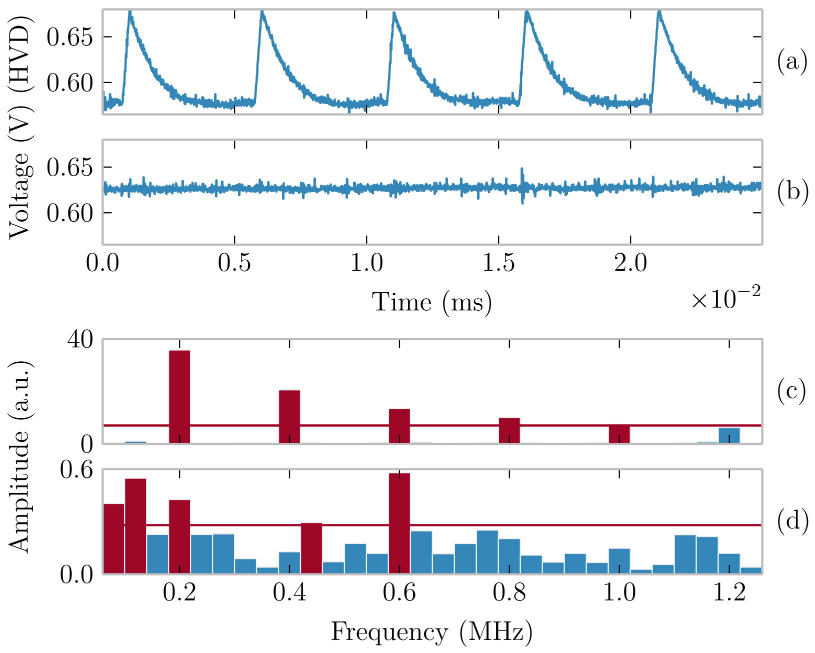


Figure 7.1: Membrane recordings of two neurons stimulated with a periodic spike input over a working **(a)** and a defect **(b)** synaptic pathway. The corresponding spectra for the working **(c)** and defect **(d)** synaptic pathways are shown in the bottom plot. The frequency range is reduced to the expected stimulus frequency and harmonics thereof. Amplitudes above the 0.85 quantile (red line) are highlighted. These dominant frequencies are used for the presented classification of defects. The criteria are explained in the text. Figure is based on data from Klähn (2013).

7.2.4 Measurement Protocol

The order in which synapses are measured and the relative configuration frequency of components has a significant impact on the total measurement time. In particular, reconfiguring floating gates is expensive. Depending on the actual values, it can take tens of seconds to reset analog parameters. Given the approximately 1.15×10^5 synapses per chip, it is important to reset analog parameters as rarely as possible. The following protocol has been specifically developed to minimize the measurement time.

Subsequent headings describe iterations over nested loops. The nesting depth is indicated by the number of prefixed >. For example, there are eight iterations over all possible Layer 1 pathways for each of the 224 synapse drivers. Within each section, the necessary HALBe interface calls are listed together with a short description.

> For each Synapse Driver (224x)

The outer most iterations run over all synapse drivers on the chip. Consequently, synapses are measured in a row-first fashion.

`reset, init` Resets on the wafer are shared by all chips. To avoid interference between different experiments running on the same wafer a cooperative `lockfile`-based synchronization scheme has been introduced. The `reset` call blocks until all active users have called the function, then the reset is triggered and all invocations return. Without a regular reset, other users on the system are blocked for long periods of time. Furthermore, resetting and reinitializing the hardware can help in case of sporadic transmission errors that have been present at the time of measurement.

`set_fg_values` Floating gate parameters have to be reprogrammed occasionally, the values otherwise drift over time. This effect has been quantified by Kononov (2011). However, configuring floating gates in the outer most loop is sufficient to ensure reliable parameter values on the one hand, while keeping the performance penalty at an acceptable level on the other.

`set_dnc_merger, set_background_generator, set_repeater,set_neuron_config, set_merger_tree` After each reset, the mergers, background generators, repeaters and neuron configurations have to be reprogrammed. Background generators are configured first to allow all repeaters and synapse drivers to lock their internal DLLs onto the address 0 events.

>> For each Layer 1 Bus and Synaptic Input (8x per driver)

This nested loop runs over all 8 synaptic pathways, see Section 7.2.1. Accordingly, the four vertical bus options and the two synaptic input options are iterated.

`set_crossbar_switch_row, set_syndriver_switch_row` One crossbar and one select switch have to be set to connect the corresponding SPL1 repeater with the current synapse driver.

`set_synapse_driver` The synapse driver configuration is responsible for selecting the synaptic input. Furthermore, the local driver reset is triggered to make sure that the synapse driver DLL is locked.

>>> For each Synapse (265x per Synapse Row)

An analog membrane trace has to be recorded for every synapse.

`set_decoder_double_row, set_weights_row` In principle, all decoders and weights in the current row can be set to forward events to neurons. However, to avoid cross talk only a single column is connected at a time. Columns are disconnected from the driver row by setting synapse decoders to unused Layer 1 addresses.

`set_denmem_quad`, `set_analog` The analog readout lines of odd and even neurons are shared. Therefore, neurons from previous iterations have to be disconnected. Then, the neuron in the active column has to be configured to output its analog membrane voltage to the readout line. Finally, the multiplexer of the analog readout is set to output the corresponding readout line.

`ADC::config`, `ADC::trigger_now`, `ADC::get_trace` The actual membrane recording is set up, triggered and read back from an attached ADC after each measurement run (HBP SP9, 2014).

The measurement for a single chip takes in total approximately 26 h using a single analog readout. Normalized to the number of synapses, 0.8 s are spent per synapse including the four possible Layer 1 connections and both synaptic inputs. The total time spent on reconfiguring the floating gates is around 1.5 h. The measurement time can be cut by about half if both chip-local analog readouts are used to measure two synapse columns in parallel (HBP SP9, 2014).

7.2.5 Distribution of Defects

The defect distribution has been measured for HICANN $(X, Y) = (16, 10)$ on the first prototype wafer system. The lack of a neuron calibration for V_{syntc}, as explained in Section 7.2.2, causes many synapses to be classified as defect. The results presented in this section should only be considered a lower bound on the number of actual defects. Subsequently, the influence of V_{syntc} on the classification performance is studied in Section 7.2.6.

The measured defect map is shown in Figure 7.2. Every pixel represents a single synapse measured over the four possible Layer 1 connections per synaptic input. The color codes for the number of defect classifications. Dark blue synapses have passed all tests. The maps are presented independently for both synaptic input circuits of the neurons. They have been found to be responsible for most defect classifications because of their individual V_{syntc} mismatch. Therefore, some of the red columns in one plot correspond to a blue column in the other. Note that synapse row indices smaller than 224 refer to synapses located in the upper synapse array and to the lower otherwise. They therefore connected to different synaptic inputs, which belong to neuron circuits in either the top or bottom half. The characteristic line patterns are discussed in the following.

Horizontal Lines in the result indicate defects spanning a complete synapse row. They can be caused by defects of one of the following components: SPL1 repeaters, background generators, Layer 1 buses or synapse drivers. The first three would cause periodic defects in multiple rows.

In Figure 7.2 horizontal lines can be observed for the synapse row indices: $(48, 49)$, $(142, 143)$, $(160, 161)$. The adjacent synapse rows are operated by the same synapse

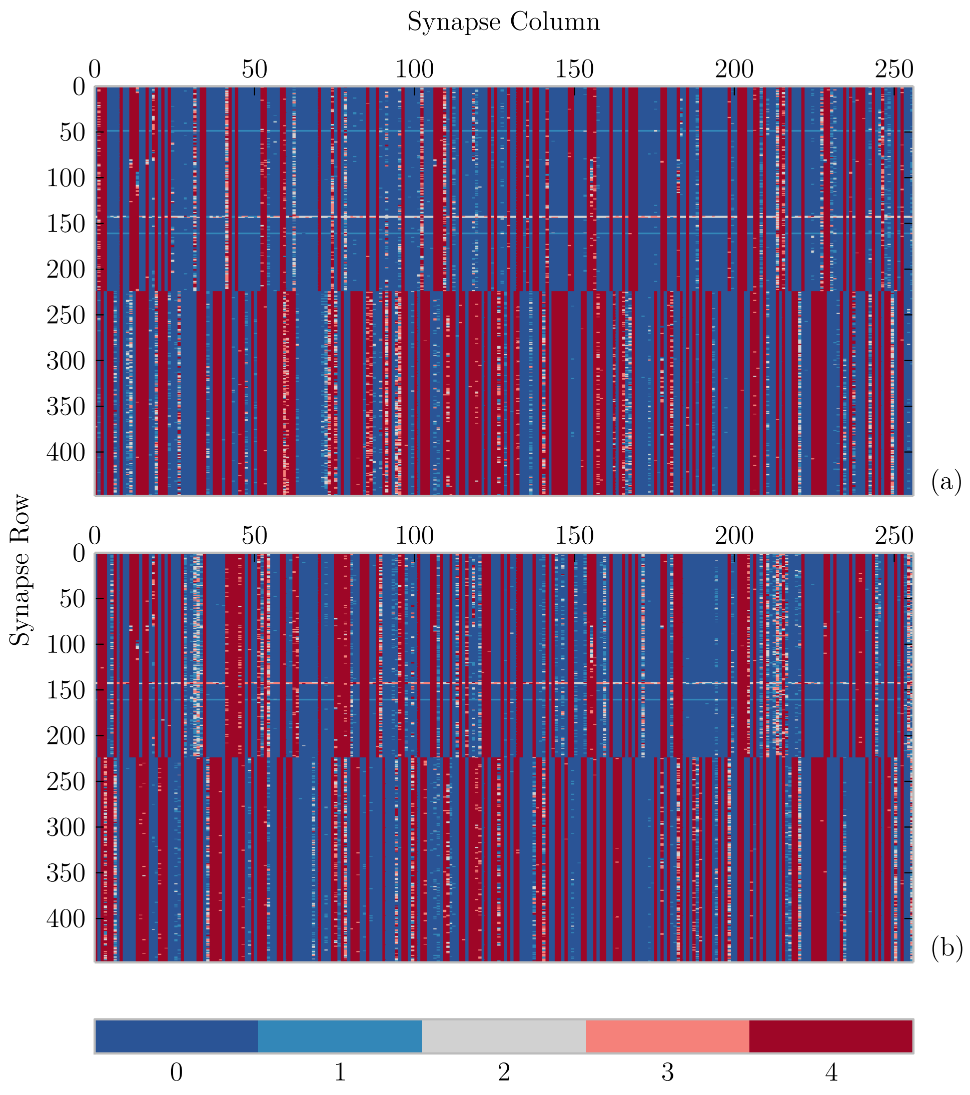


Figure 7.2: Synapse defect maps for a HICANN, visualized individually for each synaptic input circuit **(a)** and **(b)**. Synapse rows from 0 to 223 are located in the top synapse array and in the bottom synapse array otherwise. Each pixel corresponds to a synapse. The color encodes the number signal pathways that have been classified as defect. Every synapse can be reached via four Layer 1 connections per synaptic input. The column-wise synapse defects (red) are likely caused by an uncalibrated V_{syntc} and are studied in Section 7.2.6. Plot taken from Klähn (2013).

drivers, however, share no other common component. Thus, the pattern is most certainly produced by individual synapse driver glitches. Additionally, these rows are colorized irregularly, indicating sporadic failure. Possibly, the synapse drivers could not properly lock their internal DLLs, causing the unreliable behavior. Furthermore, the horizontal lines for synapse row 48 and 49 are only visible for one of the synaptic inputs. Synapse drivers need to relock their internal DLLs, when their configuration is changed to operate the second synaptic input after all 256 synapses per row have been measured. Again, unreliable locking could cause the observed behavior, thus supporting the previous DLL locking hypothesis. This does not necessarily mean that the corresponding drivers are inherently unreliable or even defect. This behavior could be the result of non-optimal parameter settings and calibrations might therefore be able to improve reliability. However, this has to be investigated by future studies.

Vertical Lines are the most dominant pattern in Figure 7.2. All synapses in the same synapse array and within the same column share a common neuron and the two synaptic input circuits. Defects of other components in the signal pathway cannot cause this kind of pattern because the synapses in different row receive input via different synapse drivers and different Layer 1 connections. Notably, red vertical columns in the upper plot in Figure 7.2 do not necessarily correspond to red columns in the lower plot. The respective measurements have been conducted for different synaptic input circuits. It is therefore reasonable to assume that the pattern is caused by the input circuits rather than the neuron itself. The following section identifies the lack of a V_{syntc} calibration as the likeliest cause for the high column-wise defect rate.

7.2.6 Synaptic Time Constant

The results presented in Figure 7.2 show that frequently all synapses within the same column are classified as defect. This particular pattern suggests that either the synaptic input circuit or the neuron itself causes the issue. In both cases, it can be considered a measurement artifact because an uncalibrated neuron does not influence the functionality of a synapse itself. Unfortunately, any synapse measurement requires neurons as intermediate PSPs detectors.

Millner (2012) found that the synaptic input circuits work only within a narrow range of V_{syntc}, varying independently for every input. Normally, these inputs translate incoming spikes that are represented by current pulses into an exponentially decaying conductance course, which in turn generates the PSP on the neuron membrane. The influence of V_{syntc} on the PSP shape is exemplified for one synapse in Figure 7.3.

If it is possible to show that the classification of the synapses fails due to the lack of neuron calibration rather than actual defects, it is reasonable to mask these measurements to derive an estimate on the actual number of defect synapses. Therefore, the classification result has been studied as a function of V_{syntc} for a few synapses. The result for 40 synapses of neuron circuit 300 is shown in Figure 7.4.

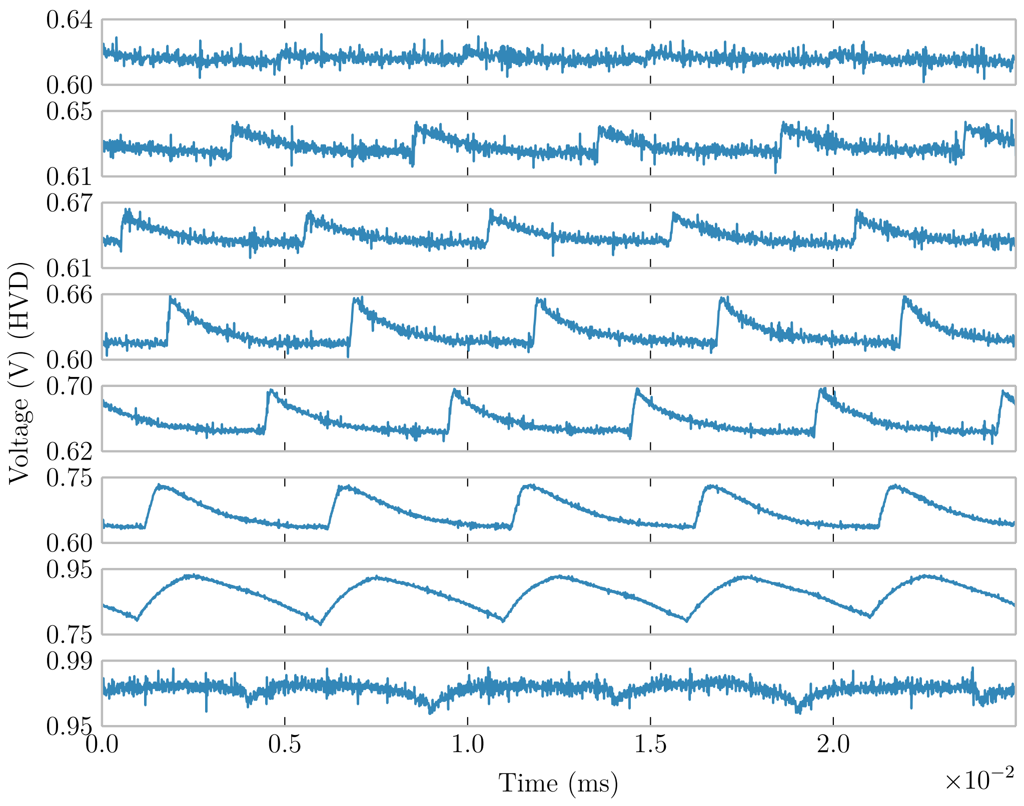


Figure 7.3: Membrane recordings of a single neuron periodically stimulated over a single synapse for different values for V_{syntc}. The corresponding V_{syntc} values are indicated on the right side of each panel and are given in digital analog converter (DAC) units. Their impact on the PSP shape is clearly visible. The respective DACs have a 10 bit resolution and can therefore be set to values between 0 and 1024. However, synaptic signal transmission provides clear PSP contributions only within a narrow range from approximately 700 to 820. Figure based on data from Klähn (2013).

Again, the color encodes the number of pathways classified as defect. Clearly, V_{syntc} influences the outcome. As expected, the area where most classifications are positive is independent for both synaptic inputs. Consequently, the parameter overlap is smaller compared to a single input. Fortunately, this is only an issue if fixed default values are used. A calibration can compensate for the variations of both inputs individually.

Note that even some of the classifications within the acceptable range yield defects. This can either be a hardware effect or false-negative classifications by the algorithm, which have previously been estimated to be around 2.2 %. The sharp transition between blue and red over multiple synapses indicates that the usable parameter range for V_{syntc} is mostly independent of the synapse driver, which simplifies a calibration of V_{syntc}. The results narrow down reasonable DAC values for V_{syntc} to range from 700 to 820.

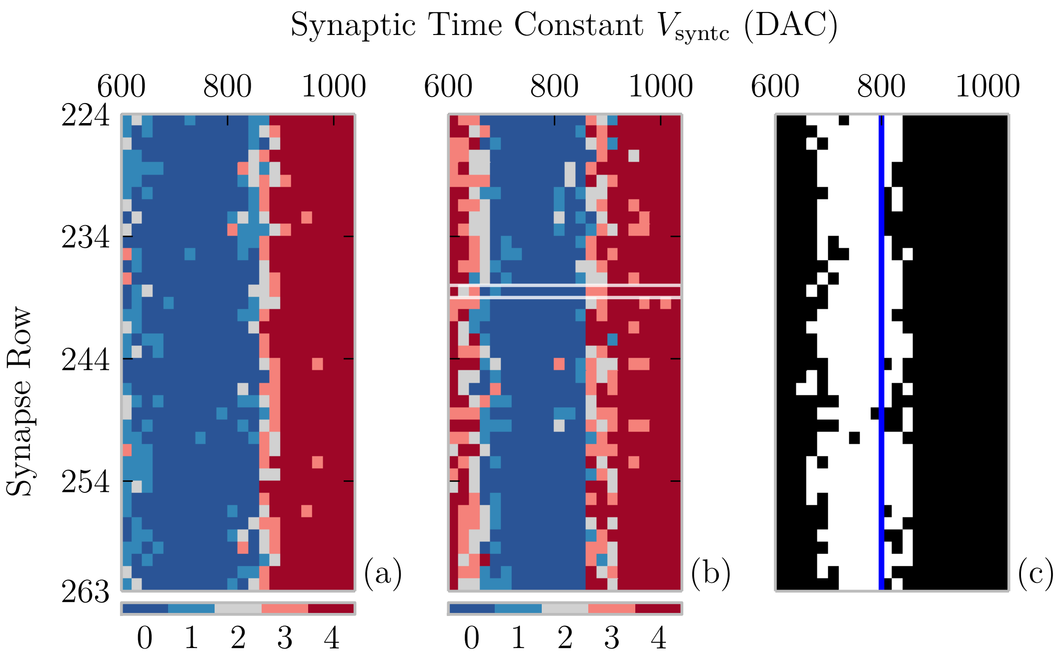


Figure 7.4: A sweep over the synaptic time constant V_{syntc} for the topmost 40 synapses of neuron 300 (bottom half). The color codes for the number of error classifications via the 4 possible signal pathways. The results for the first and second synaptic input are shown in **(a)** and **(b)**, respectively. The effective classification result is given by the intersection of both inputs by means of a logical `or`, as shown in **(c)**, where white means non of the trials has been classified as defect. Clearly, the working range is narrower for the intersection than for the individual inputs. Consequently, successful classifications are less likely for shared parameter values, even though they can, in principle, be set individually for each input. Here, a DAC setting of 800 has been used for V_{syntc}, indicated by the blue line. The synapse 239, circled in white, has been used to record the membrane traces illustrated in Figure 7.3. Figure taken from Klähn (2013).

A thorough calibration, however, should also evaluate the actual PSP shapes and not only rely on signal periodicity.

7.3 Estimation of Synapse Availability

The results presented in Section 7.2.6 showed that most defect classifications visible in Figure 7.2 are resolvable by means of calibration. This means, measurements of failing transmissions due to an unsuitable value of V_{syntc} should be masked to derive a representative estimate on the actual number of defect synapses.

Neurons in columns with high synapse defect rates above 40 % are considered unsuitably configured and therefore masked. This results in 285 out of the 512 masked neurons, slightly more in the lower synapse array with 152 versus 133 in the upper.

Masking synapses connected to masked neurons as well as unreliable synapse drivers results in 50 110 unmasked synapses whereof only 2924 synapses have been classified as defect. This yields a lower bound on the number of working synapses of 94 %. Notably, the cut-off threshold of 40 % has been chosen conservatively.

The resulting masked map of synapse defects is shown in Figure 7.5. Most defect classifications are still clustered within columns, suggesting that the corresponding synaptic input worked unreliably. Such behavior is expected for V_{syntc} values in the range of transition from working to non-working parameter values, as shown in Figure 7.4. Thus, most of the remaining defect classification are expected to be either false-negatives (2 %) or still caused by the synaptic inputs.

On the bottom line, most synapses on the tested HICANN work as intended and are therefore available for network experiments. The results are expected to further improve for calibrated synaptic inputs and future HICANN revisions, where the dynamic range of V_{syntc} will be extended to fix the issue.

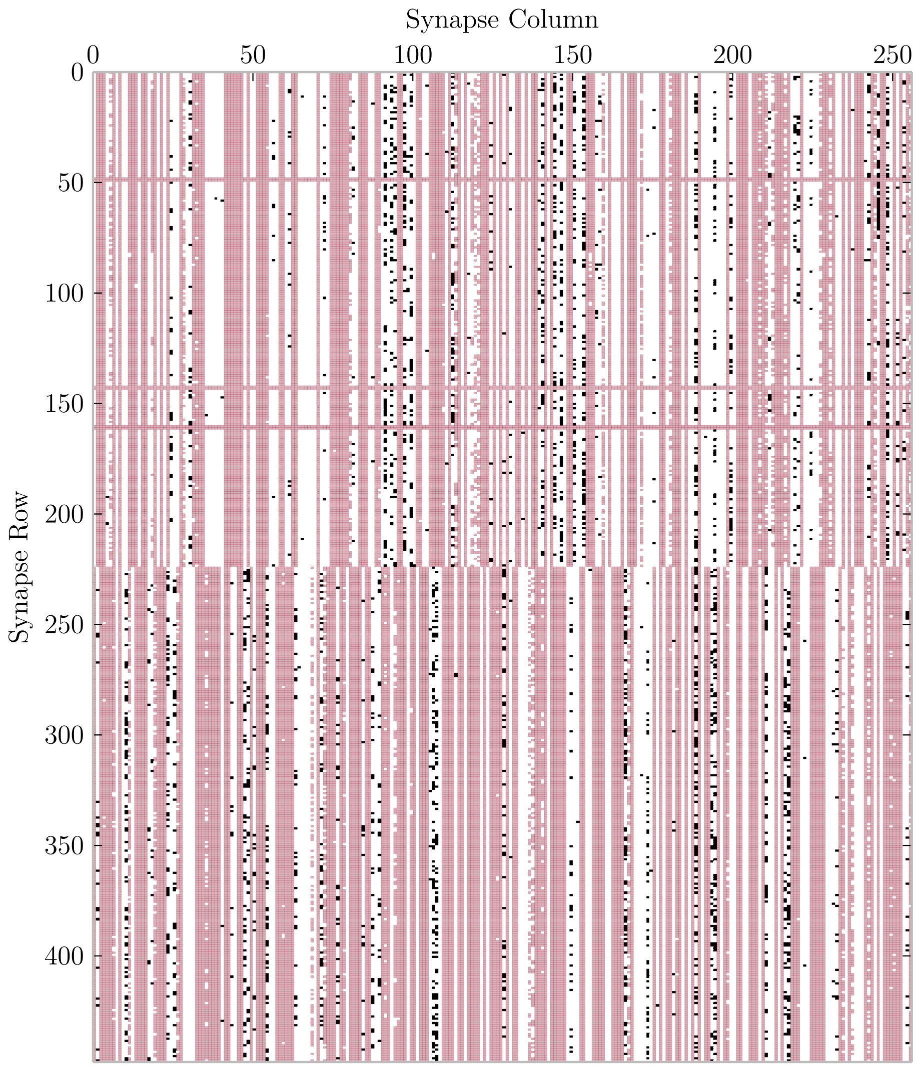


Figure 7.5: The final defect map for synapses. The results from both synaptic inputs in Figure 7.2 have been combined. Synapses are marked as defect (black) if any of the eight different signal pathways has been tested negative. Synapses colored red have been masked. They belong either to unreliable synapse drivers or neurons with unsuitable synaptic time constants. The masking threshold for column-wise defects has conservatively chosen to be 40 %. However, the remaining defects are still clustered in columns. These defects are expected to be resolved by a future calibration of synaptic time constants. Figure is taken from Klähn (2013).

8 Workflow and Mapping Benchmarks

This chapter benchmarks the workflow redesign and explores the suitability of the HMF for a variety of neural network topologies from artificially planar to complex cortical column models. Firstly, single neuron experiments are mapped to actual hardware to verify simple configurations. Then, a simulation of a multi-HICANN Hellfire chain is conducted using a preliminary version of the ESS workflow integration to demonstrate the correctness of more complex configurations. An early Hellfire chain implementation on the wafer system could not achieve the intended model dynamics. However, provided vital information to guide the development of calibrations and helped to identify an DLL locking issue for external spike inputs. Afterwards, a comprehensive study of homogeneous random networks outlines the characteristics of individual mapping steps and identifies Layer 1 bottlenecks for network implementations requiring dense long-range connectivity. Moreover, the predecessor workflow and the new workflow are compared in terms of synaptic loss, runtime performance and memory footprint. Subsequently, three different cortical column architectures, an AI state model and large planar networks are mapped. The chapter closes with a summary of all previous mapping results and discusses topology implications regarding suitable neural network models for the HMF.

8.1 Benchmark Environment

Wherever possible, the mapping results of the redesigned (marocco) and older predecessor (MappingTool) workflow are put into perspective. The old workflow is briefly introduced in Section 2.1. To keep the results as comparable as possible, all runs have been carried out on the same computational setup and both applications have been compiled using the same software optimizations listed in Appendix A.1. Furthermore, both implementations map configurations for a full, defect-free wafer instance.

The synaptic loss is expected to be similar for both workflows due to the algorithmic similarities in many approaches and the common hardware constraints, in particular. Note that the force-based cluster algorithm for neuron placement, which is described by Brüderle et al. (2011), is no longer supported by the MappingTool (Vogginer, 2014b). Therefore, the default approach has been used, which places neurons in the order they appear in the model description from the top left corner of the wafer to the bottom right in a row-first fashion.

From a runtime point of view, the new workflow turns out to be significantly faster than the old one for any studied network instance in this chapter. The effective

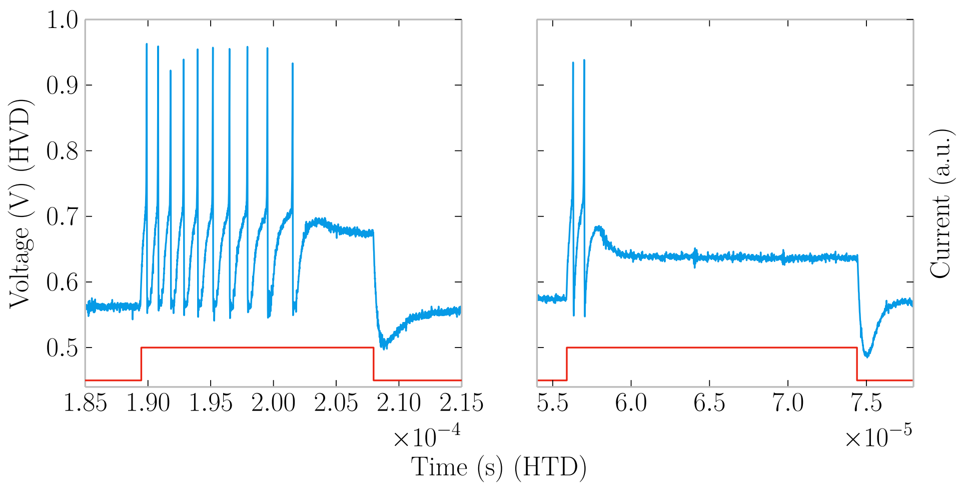


Figure 8.1: Voltage recordings (blue) of a 4-circuit hardware neuron stimulated by a rectangular current pulse (red). The complete experiment has been set up, executed and recorded via PyNN, using the new workflow. Note that the current pulse is only outlined by its principal shape because it cannot be measured. The neuron shows spike frequency adaption behavior on the left and phasic bursting behavior on the right. Both, the exponential and adaption term of the AdEx model are clearly visible. The former causes the steep rise of voltages above 0.7 V, whereas the latter is responsible for the frequency reduction and voltage under-shoot after the stimulation is turned off. Voltages and times are given in the hardware voltage domain (HVD) and hardware time domain (HTD), respectively.

performance gain depends on the topology. For most experiments, the runtime for setting up the abstract network representation and the subsequent mapping are benchmarked separately. Networks are set up via PyNN and PyHMF for the old and new workflow, respectively, while the MappingTool and marocco are responsible for mapping.

Statistical errors in all runtime measurements and distortion studies are typically small, including networks with random connectivity according to the law of large numbers and have therefore been omitted wherever appropriate.

8.2 Current Stimulus for Multi-Circuit Neurons

In a first simple experiment, a multi-circuit neuron is stimulated with a rectangular current pulse. Similar experiments for single-circuit neurons have been presented by Schwartz (2012) and Tran (2013). In this study, however, the PyNN interface is used, rather than the low-level HALBe interface, to combine multiple-circuits into larger hardware neurons, set up the current stimulation via the floating-gate controller and record the experiment. Verifying the multi-circuit configuration is particularly important because erroneously routed spike signals can damage the chip, see Section 1.4.1. The dynamics of the AdEx neuron model and its characteristic spike patterns are explained in more detail by Naud et al. (2008).

At the time of writing, no calibrations for multi-circuit neurons configurations existed. Therefore, parameters had to be manually tuned to reproduce some of the patterns presented by Tran (2013). The final parameters are listed in Appendix A.2.

Exemplarily, two spike patterns for a 4-circuit neuron configuration are presented in Figure 8.1, including frequency adaptation and phasic bursting (Naud et al., 2008). The sketched current pulse indicates its principal shape, however, cannot be directly measured on hardware.

Clearly, for membrane voltages V_m around the spike initiation potential $V_\mathrm{thresh} = 0.7\,\mathrm{V}$ the exponential term of the AdEx neuron model starts to pull up V_m until the spike detection threshold Θ is reached. There, the membrane is set back to V_reset. In the right-hand plot, V_reset is slightly below E_l. Furthermore, spike triggered neuron adaptation increases the ISIs until no spike is elicited anymore. The adaption is stronger in the phasic bursting case, such that spiking stops after a short initial burst. After the stimulus is turned off, the adaption causes a hyperpolarization of the membrane potential, which then recovers to E_l. The adaption current decays exponentially according to Equation (1.2).

This brief study demonstrates the correct configuration of multi-circuit neurons, the correct set up of the critical spike routing and the accessibility of current stimuli via PyNN. Notably, this simple experiment also exemplifies the operation and interoperation of all the novel software components from the most abstract PyNN to the lowest HALBe level.

8.3 Hellfire Chain Network

The Hellfire model is a chain-like network topology developed by Daniel Brüderle and Tom Clayton in 2010 for the neuromorphic Spikey chip (Brüderle et al., 2010). The network consists of two parallel chains, a primary carrier chain for signal propagation and a secondary control chain to preserve the signal shape. A simplified illustration of the topology is given in Figure 8.2. Whenever a chain link X in the carrier chain is active, the corresponding control link becomes active, which in turn inhibits the carrier links $X-1$ and $X+2$ to limit the activity to a narrow window of two links only. As the activity moves forward to the next link via the excitatory connections in the carrier chain, the activity window shifts to the right and the current link is inhibited by the next control link. This safe-guarding mechanism renders the Hellfire chain robust against effects like signal dispersion (Brüderle et al., 2010). In the following, the chain is typically closed to form a ring by connecting the last and first links. Therefore, activity in the ring sustains itself and the signal can theoretically propagate for an indefinite period of time.

8.3.1 Guided Placement

Chain-like topologies predominantly implement local connectivity, which is limited to a few neighboring chain links. Such networks can be routed efficiently, as long as the

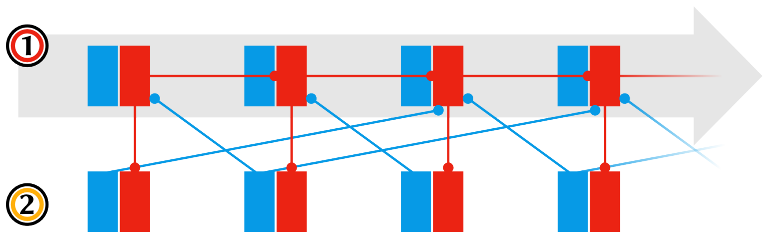


Figure 8.2: A schematic of the Hellfire chain model, which consists of two parallel chains. Each chain link consists of an excitatory and an inhibitory population, which are shown in red and blue, respectively. Information is conveyed via the carrier chain **(1)** from left to right. Activity in the secondary control chain **(2)** is synchronized to the carrier chain and safe-guards the extent of activity by inhibiting carrier links right before and the one after the next. A detailed listing of connectivity can be found in Appendix A.2.

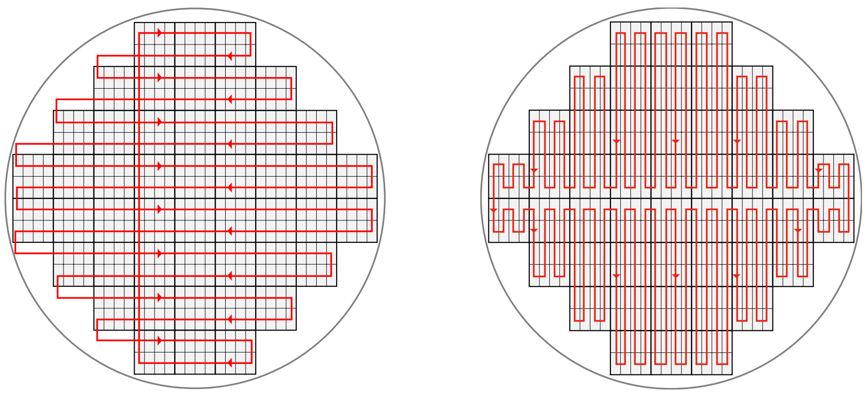


Figure 8.3: Two strategies for a resource efficient neuron placement of chain-like topologies to wafers. Note that the arrows only indicate the placement order and not the physical Layer 1 connections. Laying out the chain in the zigzag fashion, shown left, results in long-range connections when closing the chain to a ring-like topology. Zigzag from top to bottom is worse, requiring more scarce horizontal buses. The meander strategy, shown right, maps ring topologies efficiently without imposing any long-distance horizontal connections. This strategy has therefore been used to place the Hellfire and Synfire chain networks presented in this chapter.

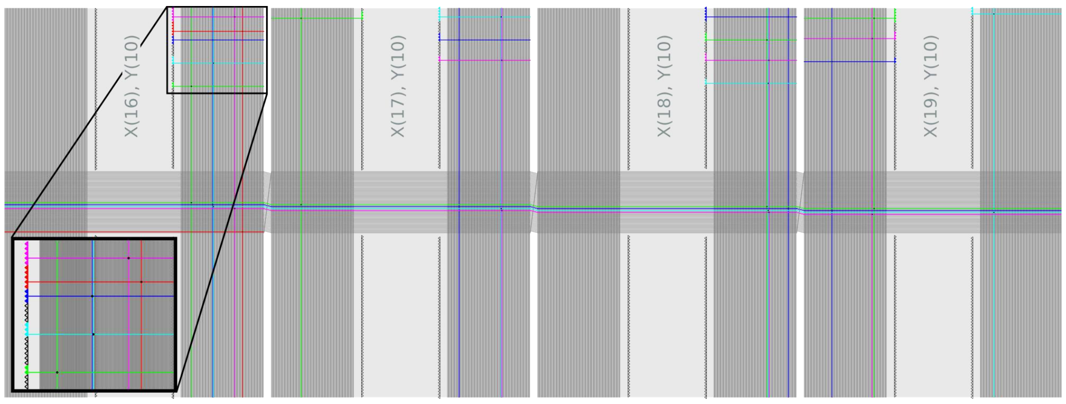


Figure 8.4: Routing visualization of a 4-link Hellfire chain mapped to 4 adjacent HICANNs, each implementing one carrier and one control chain link. Every chip hosts 28 neurons built from 4 circuits each. A single outgoing Layer 1 connection per HICANN relays events to the other links. The connections are shown in 4 different colors running horizontally near the center. The initial spike stimulus is injected via the red horizontal bus on the left chip. This configuration has also been mapped to hardware, the results are presented in Section 8.3.3.

neighborhood relationship between links is preserved during the placement. This means that any connected pair of neurons is placed closely together. In fact, wafer-size Hellfire chain instances can be realized without running short of routing resources.

The guided placement provides control over the layout of the ring topology on the wafer. An intuitive approach is to e.g., start with a chip in a corner and then fill up the wafer in either a row-first or column-first zigzag fashion as shown in Figure 8.3. Both options lead to long-range connectivity when closing the chain, which may become an issue depending on the actual model topology and the number of neurons per chip. The second meander-like strategy in Figure 8.3 solves the problem elegantly and is therefore used to realize both, the Hellfire and Synfire chain, see Section 8.9.1. Generally, long-range Layer 1 connections should be avoided because they can block other connections from their targets. Horizontal buses are particularly expensive due to the fact that only one fifth of all buses are horizontal.

In this study, variably sized Hellfire chain instances are mapped such that the largest network instances utilize the wafer to its full extent. Up to four chain links of 4×7 neurons each are placed per HICANN and four circuits are used per neuron. The number of model synapses S linearly grows in the number of links L due to the purely local connection structure and has found to be $S = 616 \times E$. For four links per chip, 112 of 118 available Layer 1 source addresses and 87.5 % of all neuron circuits are used, which means that neuron as well as Layer 1 resources are used efficiently. Up to two chain links per chip can be routed using a single SPL1 output. In cases where two SPL1 outputs per chip have to be routed, the importance of efficient placement strategies is stressed, e.g., up to $4 \times 4 \times 2 = 32$ cross-wafer connections have to be routed for the already efficient zigzag strategy shown in Figure 8.3.

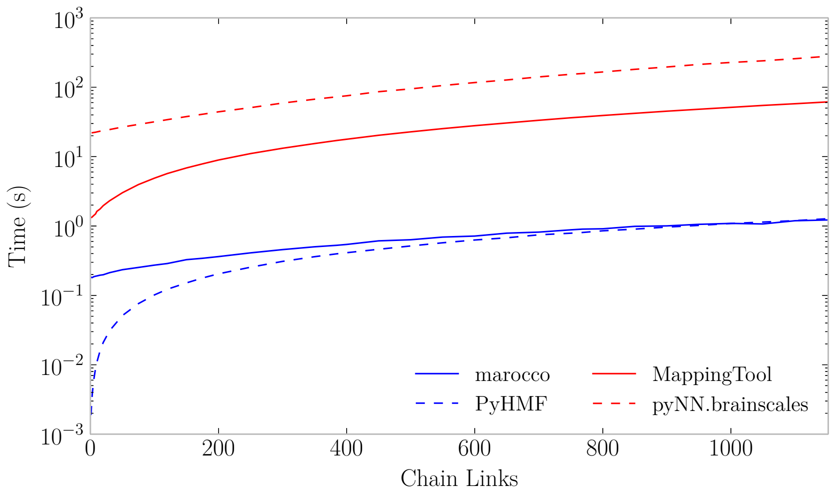


Figure 8.5: Network setup and mapping times for the Hellfire chain model indicated by dashed and solid lines, respectively, for the old (red) and new (blue) workflow. The maximum instance size is bound by the wafer extent. Marocco is typically more than an order of magnitude faster, while the network setup via PyHMF is even more than two orders of magnitude faster. The average speedup for both components combined has found to be 112 and increases towards larger network instances.

Exemplarily, a mapped Hellfire chain of length 4 with one link per chip is visualized in Figure 8.4. The connectivity matches the topology shown in Figure 8.2. Each chip connects to any other chip to excite the subsequent link to the right and inhibit the others. The left most chip also receives external spike inputs to initiate signal propagation at the start of the experiment.

8.3.2 Workflow Performance Comparison

The Hellfire chain model is the first setup to compare the runtime performance of both workflows. Network setup times and mapping performance have been studied separately and are shown in Figure 8.5.

Clearly, the new workflow is faster. The mapping step is typically more than one order of magnitude faster, except for very small network instances. Setup times are even more than two orders of magnitude shorter for all studied chain lengths. Remarkably, the performance gain of the new workflow increases towards larger network instances and therefore pays off even more. The average speedup over all mapped network instances is 112. While mapping the largest full-wafer network instance with marocco has found to be even 137 times faster. It only takes 2.5 s versus 343 s for the MappingTool.

Note that for technical reasons the MappingTool uses its default linear placement, which produces results similar to the first zigzag strategy in Figure 8.3, whereas marocco uses the meander strategy. Using the default placement for the MappingTool causes lost synapses for some of the network instances depending on where the first and last chain links are located on the wafer. However, the occurrence of synaptic loss does not affect the runtime measurements as shown by the smooth curve progression.

The performance improvements have been achieved by more efficient implementations and parallelization alike. Nonetheless, parallelization only applies to the mapping and can only account for a factor of approximately 3 to 4 on the computational setup, which is described in Appendix A.1.

For the old workflow, the network setup takes longer than the actual mapping, which is surprising regarding the relative complexities of the tasks. This can be explained by the fact that most functionality of the old PyNN backend is implemented in `Python` on the one hand and that mapping chain topologies is particularly simple and therefore fast on the other. For the new workflow, network setup via PyHMF is typically faster than the subsequent mapping via marocco, as one would expect. Towards large network instances the setup and mapping times of the new workflow approach each other because the relative mapping overhead per synapse becomes smaller. Furthermore, marocco carries out the synapse assignment of multiple HICANNs in parallel, whereas PyHMF creates model synapses purely sequential.

8.3.3 Towards a Hardware Implementation

The Hellfire chain model has been chosen as the first model to be implemented on hardware due to its prior realization on the neuromorphic Spikey chip (Brüderle et al., 2010). However, an early implementation of the model on the wafer system failed to achieve a propagation of spiking activity along the Hellfire chain links. The implementation is currently limited by the lack of comprehensive multi-circuit neuron calibrations. Furthermore, a DLL locking issue for external spike inputs has been identified. Marocco has been updated to automatically work around this problem. Future FPGA firmware or hardware revisions are expected to provide a more comprehensive solution.

In this section, firstly, the analog regime and excitability of neurons is studied. Next, the correctness of mapped configurations is verified by an ESS simulation using a preliminary version of the HALBe integration. Finally, observations and current limitations are discussed regarding the early Hellfire chain implementation on hardware.

Preliminary Study of Neuron Excitability

In this preparatory study, the default neuron parameter transformations for single-circuit neurons and preliminary calibrations are used to configure a multi-circuit hardware neuron using the model parameters of the Hellfire chain network. A corresponding configuration is applied to the wafer setup and a neuron tested regarding its excitability and therefore its ability The resulting configuration is applied to the wafer setup and the neuron tested regarding its excitability. Suitable neurons for a Hellfire chain

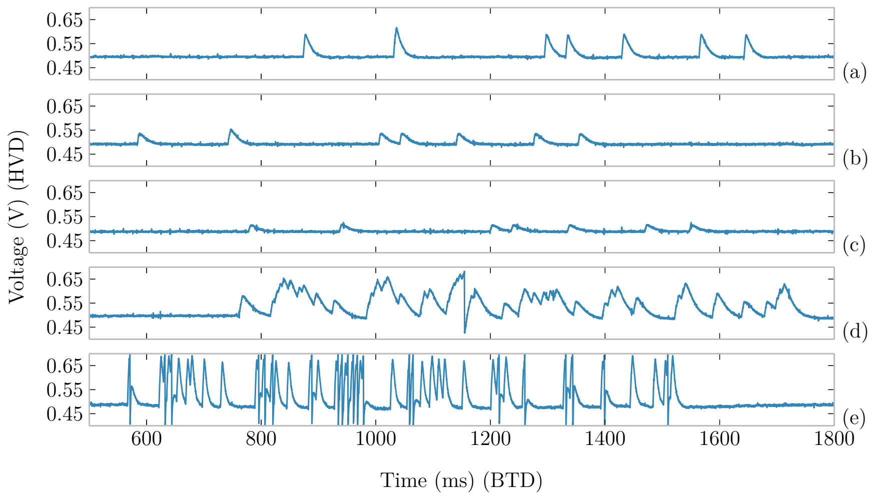


Figure 8.6: Neurons built from 1, 2 and 4 circuits in **(a)**, **(b)** and **(c)**, respectively, are stimulated with 10 Hz Poisson spike input. In **(d)**, the two circuit neuron is stimulated using a higher input rate of 40 Hz, which is sufficient to elicit a single spike. The bottom plot **(e)** shows the same setup with reduced membrane capacitance from 2×2.16 pF to 2×0.16 pF. Consequently, the neuron is more sensitive to input and spikes several times. Times are given in the biological time domain (BTD), while voltages are in the hardware voltage domain (HVD). Note that the recording trigger has been working unreliably causing random time offset.

implementation have to be sensitive enough to pick up spiking activity to propagate the signal further.

Simple voltage and membrane time constant calibrations exist for single circuit neurons. However, multiple neuron circuits have to be connected by the mapping for the implemented synapse driver routing and to achieve high input counts, while keeping the number of connected synapse drivers small to ensure reliable Layer 1 communication, see Section 1.5.5. Ideally, the membrane capacitance of a neuron is the sum over the individual circuit capacitances. Achieving PSPs of equal amplitude on a larger membrane requires stronger synaptic currents. The goal is therefore to establish a suitable analog regime for 4-circuit neurons that could in principle allow signal propagation. A thorough calibration would go beyond the scope of this thesis.

Voltage traces for differently sized hardware neurons stimulated with Poisson spike input are shown in Figure 8.6. The corresponding hardware parameters, which have been produced by the default parameter transformations at that time using the Hellfire parameters, are listed in Appendix A.2. These transformations are currently being reworked, thus, results are expected to change in the future.

From **(a)** to **(c)** in Figure 8.6 the neuron is built from 1, 2 and 4 circuits, respectively, and receives Poisson input with a biological rate of 10 Hz via a single synapse. This synapse is set to the maximum digital weight, uses the default synapse driver base-efficacy $g_{\max}$ and minimal conductance divider g_{div}, see Section 6.7.3. The latter maximizes the synaptic efficacy for fixed values of $g_{\max}$. The input spikes should in principle coincide for the first three recordings because the same spike trains have been used. However, one can see that the experiment trigger for the analog recorder has not been working reliably at the time of measurement resulting in arbitrary time offsets.

Clearly, the circuit capacitances add up. Every time the neuron size is doubled, the PSP amplitude is approximately halved. As explained earlier, getting larger neurons to fire requires stronger stimulation. In this case, the provided input is not strong enough to trigger any spike, even for the single-circuit neuron. Thus, the biological input rate is increased from 10 Hz to 40 Hz in order to get the neuron to fire using the 2-circuit neuron configuration for **(d)** and **(e)**. In **(d)** a single spike has been triggered at around $t = 1160\,\text{ms}$ (BTD). Note that the synaptic efficacy is already at the hardware limit and can therefore not be strengthened any further. The excitability of the neuron could have been increased by lowering the threshold voltage. However, without a suitable neuron voltage calibration this can be done only to a certain extent without risking that some of the neurons start spiking randomly by themselves. In consequence, the circuits are switched to the alternative, smaller capacitance of 0.16 pF rather than the default of 2.16 pF. This also reduces the membrane time constant and has originally been designed to support higher analog speedup factors. Here, the small capacitance is used to make the neuron more sensitive to spike input, hence strengthening the synaptic efficacy. The respective response is shown in **(e)**. Significantly more spikes have been triggered for the same stimulus in comparison to **(d)**. It is therefore possible to elicit spikes with a single strong synapse.

Other parameters, like g_{l}, could have been studied to correct the membrane time constant with respect to the reduced capacitances according to $\tau_{\text{m}} = {}^{C}\!/_{g_{\text{l}}}$. However, calibrating the system is beyond the scope of the thesis.

Hardware Simulation and Mapping Verification

To verify the configurations generated by marocco a preliminary HALBe integration of the ESS (Pape, 2013) is used to conduct a simulation of the Hellfire chain. The most significant difference between prior ESS studies using the MappingTool and the new workflow is that everything is set up using only HALBe interfaces. This means that unlike before, the ESS has no access to the original network model, which is an important improvement making the ESS behave more like the actual hardware system. Thus, neuron parameters have to be reconstructed from the hardware DAC values supplied via the HALBe interfaces. However, the parameter transformation and especially the reverse transformation of the ESS are currently under active development. Yet, the ESS captures the topological properties of hardware correctly, including the merger tree, the Layer 1 network and synapse array connectivity. For this study, the reverse transformation of neuron parameters has been tuned such that the results reproduce the output of an ideal NEST-based (Diesmann and Gewaltig, 2002; Eppler

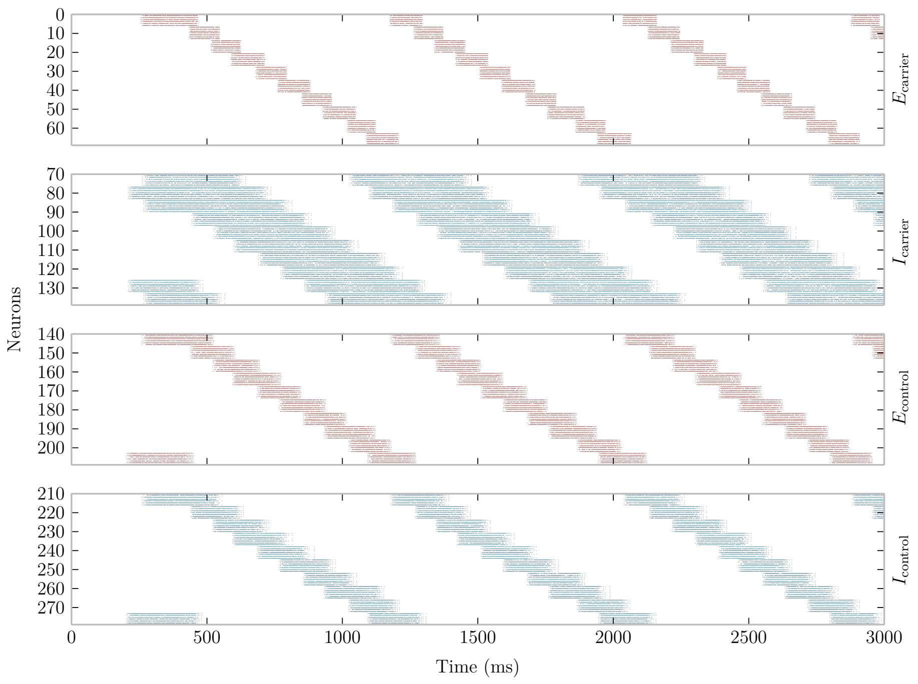


Figure 8.7: Hellfire chain of 10-links set up by marocco and simulated on the ESS. A preliminary version of the HALBe integration has been used to configure the simulator, see Section 2.2.5. Activity is conveyed by the excitatory carrier chain populations E_{carrier}. The narrow excitatory activity is safe-guarded by broader inhibitory activity in I_{carrier} that inhibits activity in prior and subsequent links, Thus preventing signal dispersion. The successful simulation and behavior in accordance with a reference simulation demonstrates that all topological aspects of the Hellfire chain have been mapped correctly. Data provided by Paul Müller, who is responsible for the ESS workflow integration.

et al., 2008) reference simulation. In fact, the reference simulation has been omitted because the raster plots of both simulations look alike.

The spike responses produced by the ESS are presented in Figure 8.7. The activity is separately shown for excitatory and inhibitory populations as well as the carrier and control chain in accordance with Figure 8.2. The signal is conveyed by excitatory carrier chain populations E_{carrier}. Note that the activity of the inhibitory populations in the carrier chain I_{carrier} lasts about 4 times as long. It starts to receive stimulation as soon as the chain link two in advance becomes active and keeps receiving stimulation until the activity has passed on to the link after the next. The activity in the control chain populations E_{control} and I_{control} is simply driven by the activity of the excitatory carrier population and is therefore synchronized. Their activity lasts a little longer because they are not actively inhibited by other populations, unlike the excitatory carrier populations.

The successful signal propagation and network dynamics in close match with the reference simulation demonstrate that the wafer routing and all topological aspects of the hardware configuration are mapped correctly.

Hardware Implementation

Finally, a 4-link 4-HICANN Hellfire chain configuration has been mapped and set up on the wafer system. The mapping is visualized in Figure 8.4. Every chain link consists of 4×7 neurons, each built from 4 neuron circuits. The neuron parameters have been set according to the single neuron excitability study, described earlier. A listing of these parameters and more informations about the actual hardware instance is given in Appendix A.2. Furthermore, the initial excitatory input, which starts the signal propagation, has been increased by setting all weights to maximum and doubling the rate. On hardware, spiking activity has been observed in the first chain link. However, only few neurons did fire and the ones that did, erratically fired in short bursts.

The first issue that only a few neurons received sufficient input to fire, is likely caused by the lack of a V_{syntc} calibration, which is explained in Section 7.2.6. Therefore, many of the used synaptic input circuits either worked unreliably or distorted incoming PSPs. Furthermore, comprehensive multi-circuit neuron calibrations are required to reduce neuron to neuron variability and make the setup more robust.

The observed bursting behavior of active neurons might be caused by adaptation of the AdEx neurons on the one hand or a locking issue for external spike inputs on the other. Firstly, the adaptation and exponential dynamics of the hardware AdEx neuron are currently examined in more detail via transistor level simulations, improved control thereof will then be provided by updated calibrations. Secondly, a repeater locking issue for external input has been discovered as part of this study and is currently under investigation. According to Section 1.5.5, repeaters and synapse drivers require periodic address 0 events in order to lock internal DLLs. The on-chip background generators provide these events for recurrent on-wafer connections. Once activated, the background generators provide constant stimulation even if no experiment is in progress, thus locking the internal DLLs of repeaters and synapse drivers at any time. The background generators have also been used in all low-level measurements so far to augment external spike inputs to ensure DLL locking. However, using background generators for external input locking in more complex network setups complicates the event routing for neurons via the merger tree and reduces Layer 1 utilization. Therefore, reduces the amount of implementable long-range connectivity. Instead, a simpler workaround has been integrated into marocco, which dedicates merger 7 in the top merger tier for background events. This reduces the number of accessible neuron circuits per HICANN by 64, however is only a minor limitation in a full-wafer setup for small to medium size networks. This workaround has also been used in the prior excitability study. Note that address 0 events could in principle also be supplied by the FPGAs. In fact, this has been the intended approach, however, the time between repeater configuration and event onset is too long for the DLLs to lock reliably. Worst-case timings and optimal strategies to provide events earlier in the programming process are currently investigated. The issue is expected to be fixed by an FPGA firmware update providing explicit control over configuration and spike onset times. In the long run, changing the merger tree topology to insert background generator events on the lowest merger tier could also resolve the issue and render the use of background generators more flexible.

Apart from activity on the first chip, PSPs have been recorded on the second HICANN containing the second Hellfire chain link. This might indicate successful event routing of events from the first to the second chip. However, spike and voltage recording are not yet synchronized. Thus, verifying that the PSPs are actually caused by activity on the first chip has not been possible. Moreover, the stimulation was neither strong nor frequent enough to cause spiking behavior on the second HICANN and therefore propagate the Hellfire signal.

Even though no signal propagation has yet been achieved for the Hellfire chain model, this study helped to identify an DLL locking issue for external spike input, which is now under thorough investigation on a lower level. Meanwhile, marocco has been updated to automatically work around this issue by dedicating a part of the merger tree for external spike input. Therefore, the necessary address 0 events can be supplied by on-chip background generators. Ultimately, comprehensive calibrations for multi-circuit neurons will resolve the synaptic input issue and provide a suitable analog regime for neurons. It is expected that this will ultimately enable working Hellfire chain implementations on the HMF.

8.4 Homogeneous Random Networks

Homogeneous random networks are simple models where any neuron is connected to any other neuron with a fixed probability of p. The number of model synapses N_{syn} in a network of N neurons grows therefore with $p \times N^2$. This model has previously been studied by Fieres et al. (2008) using a prototype mapping and a preliminary hardware topology. In accordance with the publication, networks with connection probabilities of $p = \{1\,\%, 5\,\%, 10\,\%\}$ are studied. This provides a reference frame for the interpretation of the results in this section. However, they only mapped network instances with up to 16 383 neurons, whereas full-wafer instances with more than 4.5×10^4 neurons are mapped in the thesis at hand.

The structural isotropy of homogeneous random networks makes them particularly suitable for routing benchmarks. In fact, the neuron placement has little impact on the final configuration. To understand this property, we first look at the probability of a single neuron not to receive input from another neuron located on HICANN x

$$p_x = (1 - p)^n \qquad , \tag{8.1}$$

where n is the number of neurons placed per chip which can either be $n_8 = 59$ or

$n_4 = 118$ for 8 and 4 circuits per neuron, respectively. The probability for a whole HICANN not to receive input from HICANN x is then

$$P_x = p_x^n = (1-p)^{n^2} \quad . \tag{8.2}$$

P_x approaches zero for only a few neurons per chip even for low connection probabilities p, e.g., $P_x = 6.4 \times 10^{-16}$ for the lowest connection probability $p = 1\,\%$ and $n = n_8 = 59$. This means, every HICANN likely receives input from every other HICANN, leading to a worst-case all-to-all inter-chip connectivity. At the same time, it is very unlikely that a neuron placements exists that simplifies the wafer routing. Another consequence is that the wafer routing task becomes independent from the connection probability and only depends on the number of neurons. This also means, inter-chip connectivity is independent from the actual random processes.

Different network instances have been studied regarding their utilization of hardware synapses, network setup time, mapping time and most importantly their model fidelity after mapping. The fidelity ν_fidelity has been defined by Fieres et al. (2008) as the fraction of synapses that are successfully realized on hardware, according to

$$\nu_\text{fidelity} = \frac{N_\text{realized}}{N_\text{syn}} = 1 - \nu_\text{loss} \quad , \tag{8.3}$$

with the number of realized synapses N_realized, the number of model synapses N_syn and the relative number of lost synapses ν_loss.

The results are presented in Figure 8.8. Different colors and line styles denote different quantities and connection probabilities $p \in \{1\,\%, 5\,\%, 10\,\%\}$, respectively. Note that marocco provides two implementations for both, wafer routing and synapse driver routing. Here, only the results for the *iterative horizontal growth* wafer routing (see Section 6.4.4) and the *iterative best fit* synapse driver routing (see Section 6.5.2) algorithms are presented because this combination produced the best results on average. The *iterative shortest path* wafer routing (see Section 6.4.3) produced 6.5 % higher synaptic loss on average due to higher Layer 1 bus utilization. The *simulated annealing* synapse driver routing (see Section 6.5.3) produced the best results for some homogeneous random network instances. However, it requires parameter tuning, is typically 20 % slower and leads on average to 4 % less synapses using the default parameters compared to the *iterative best fit* synapse driver routing.

Homogeneous random network instances ranging from 5000 to 45 312 neurons have been mapped, which is the practical limit for 4-circuit neurons in combination with efficient Layer 1 network utilization, as explained in Section 6.1. The largest network instances use 92 % of all available neuron circuits on the wafer. For every network instance, configurations of 4 and 8 circuits per neuron have been chosen such that all neurons still fit on the wafer and at the same time the synaptic loss is minimized. In

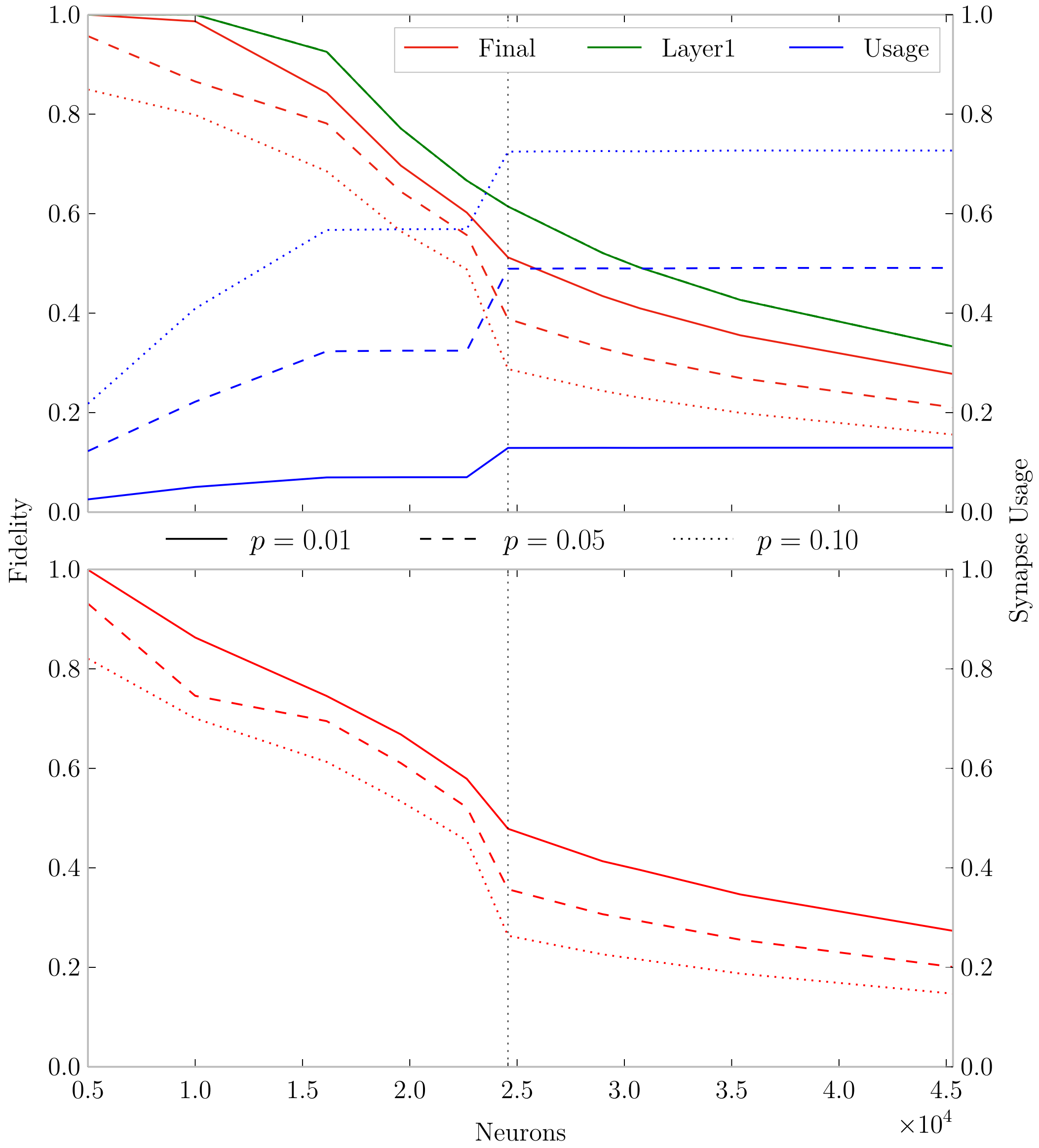


Figure 8.8: Mapping results for marocco and the MappingTool on homogeneous random networks in the upper and lower plots, respectively. The final fidelity (red), an intermediate fidelity after the Layer 1 wafer routing (green) and the synapse usage (blue) are shown. The latter two are only available for marocco. Network instances with connection probabilities p of 1 %, 5 % and 10 % are denoted by different line styles. Notably, the fidelity after the wafer routing does not depend on p and the synapse usage saturates for larger networks, see text. The vertical grid lines at 24 576 neurons indicate the transition from 8 to 4 circuits per neuron. At this point, the model fidelity regresses because fewer inputs are available per neuron, whereas the synapse usage increases, see text.

practice, the 8 circuit configuration is used for network instances as large possible, since neurons receive input from approximately

$$N_{\mathrm{inp}} = p \cdot N \tag{8.4}$$

other neurons. This number easily exceed the 4×224 synapses of smaller 4-circuit configurations. The transition from 8 to 4 circuits is indicated by vertical grid lines at $N = 24\,576$ in all Figures from Figures 8.8 to 8.12. Using 12 or more circuits per neuron increased the synaptic loss because the Layer 1 resources are exhausted earlier due to the larger number of active chips. Such configurations can still be beneficial for higher connection probabilities requiring more inputs per neuron.

Figure 8.8 shows the model fidelity and the usage of hardware synapses, which is defined as the fraction of mapped hardware synapses on chips that have neurons placed to them. The usage is given for marocco only, since the MappingTool produced unreasonably small values. Furthermore, the green line in the upper plot outlines the fidelity of the model after the Layer 1 wafer routing, before the actual synapses have been assigned. This intermediate fidelity accounts for all model synapses that have not yet been explicitly lost as a consequence of unreachable chips. Moreover, only a single green line is visible. The wafer routing is independent of the connection probability because all probabilities alike lead to chip-wise all-to-all connectivity, as explained earlier.

The distance between the green line and the upper edge of the plot denotes the fraction of synapses that are lost during the wafer routing due Layer 1 connections that could not be established. Similarly, the area between the final synaptic loss, shown in red, and the green line denote synapses that are lost locally either during synapse driver or synapse array routing.

Both mapping implementations produced similar results in terms of model fidelity. Generally, marocco produced slightly less distorted representations. This is most apparent for smaller networks, e.g., the $N = 10^4$ and $p = 1\,\%$ network instance is mapped almost loss-free by marocco, whereas the MappingTool produces $\sim 10\,\%$ loss.

Reducing the number of circuits per neuron from 8 to 4 at $N = 24\,576$ leads to a sharp decline in model fidelity, as the number of inputs per neurons is suddenly halved. The penalty is emphasized towards higher connection probabilities because of an increased input requirement per neuron according to Equation (8.4). Since this is an immediate result of topological constraints, the effect is comparable for both mapping implementations. The Layer 1 fidelity is barely influenced by the circuit reduction and declines steadily.

The usage of hardware synapses is expected to increase linearly with the number of neurons N and fixed connection probabilities p according to Equation (8.4). In fact, this behavior can be observed up to $N = 16\,128$ neurons, where the utilization saturates, which coincides with a steeper decline in fidelity. This effect is caused by Layer 1 bus and synapse driver exhaustion, rendering the number of inbound connections per HICANN constant. Therefore, any neuron can receive input from only a fixed effective number of afferent neurons N_{eff} limiting the number of realizable synapses per neuron to $N_{\text{eff}} \cdot p$. When the number of circuits per neuron is reduced from 8 to 4, twice as many neurons are placed per chip, thus increasing N_{eff}. Gaps of unused hardware synapses can now be used for additional connections, which in turn increases the synapse utilization. After the circuit reduction, Layer 1 resources are still exhausted rendering the number of inbound connections constant. However, the number of realizable synapses per neuron is increases because N_{eff} has been increased. For network instances with $p = 9\,\%$ and 24 576 neurons or more, the synapse usage settles at around 73 %. Theoretical, up to 92 % of the synapses could be used in accordance with the 92 % of mapped neuron circuits (8×59 and 4×118). The discrepancy of 19 %, between 92 % and 73 %, is caused by routing granularity, e.g., multiple synapses have to be assigned even though only a single one is needed due to the two-stage decoding scheme.

Fieres et al. (2008) observed a similar convergence at around 80 %, though the results are difficult to compare because the hardware topology has significantly changed since then. For example, the number of synapse drivers has been reduced by approximately 14 %. A dedicated mapping run with marocco for $N = 24\,576$ and $p = 100\,\%$ showed a synapse usage of 91.9 % ($\sim 92\,\%$) verifying that the implementation can successfully assign all available hardware synapses. The remaining 0.1 % can be explained by unassigned synapse drivers due to select switch sparseness.

The isotropy of homogeneous random networks allows to model the network fidelity ν_{fidelity} as a function of neurons, the synapse usage u and the connection probability p as follows

$$\begin{aligned}\nu_{\text{fidelity}}(N, u, p) &= \frac{N_{\text{realized}}}{N_{\text{syn}}} \\ &= \frac{u \cdot N_{\text{chip}} \cdot N_{\text{syn/chip}}}{N^2 \cdot p} \\ &= \frac{u}{N} \frac{N_{\text{syn/chip}}}{n \cdot p} \qquad .\end{aligned} \tag{8.5}$$

Where $N_{\text{chip}} = {}^{N}\!/_{n}$ denotes the number of actively used chips to place N neurons and the number of neurons per chip n as in Equation (8.1). The number of hardware synapses per chip $N_{\text{syn/chip}}$ is fixed, therefore $u \cdot N_{\text{syn/chip}}$ is the number of used synapses per chip. When u saturates, the fidelity for a given connection probability p and fixed number of neurons per chip n decreases according to

$$\nu_{\text{fidelity}}(N) = \frac{C}{N} \qquad , \tag{8.6}$$

with the constant factor $C = \frac{u \cdot N_{\text{syn/chip}}}{n \cdot p}$. This $\sim {}^{1}\!/_{N}$ decline can be observed in the upper plot in Figure 8.8 for $N > 16\,128$, where u becomes constant. For example, $C \approx 7.1 \times 10^3$ for $p = 10\,\%$ and the measured synapse usage of $u = 73\,\%$, which leads to $\nu_{\text{fidelity}}(4.5 \times 10^4) \approx 0.16$ in accordance with the independently measured results.

8.4.1 Workflow Performance Comparison

In this section, the performance of the network setup and mapping implementations is discusses and compared for both workflows. The measured runtime is shown in Figure 8.9.

The setup times of PyHMF and the old PyNN backend both scale proportional with the number of model synapses, which is indicated by the green lines following $p \cdot N^2$. This is expected behavior, at least N^2 random numbers have to be generated in order to set up the random connectivity, which dominates the instantiation of N neurons.

In contrast, the mapping runtime is more complex. For marocco, the time is almost independent from p, whereas higher connection probabilities slow down the MappingTool. This is a consequence of the underlying data structures that are used to represent synapses. PyHMF instantiates connection matrices, thus marocco can look up connections fast in constant time. In fact, the relative amount of time spent on the local synapse routing is small such that the overall runtime is dominated by the wafer routing, which is independent from p for homogeneous random networks. Instead, the runtime scales cubically with the active wafer area explaining the super linear dependency of marocco on the number of model synapses.

The old PyNN backend on the other hand uses adjacency lists to store synapses. The MappingTool translates this representation into an equivalent incidence list graph (Even and Even, 2011), leading to redundant copies of connectivity. Nonetheless, looking up specific connections in either representation takes longer for higher p, more specifically, the necessary time increases according to Equation (8.4). For larger network instances, the runtime of the MappingTool follows the course of model synapses more closely because significantly more time is spent on the actual routing of synapses. This explanation is supported by the observation that the course deviates stronger for the smallest connection probability of $p = 1\,\%$.

At approximately $N = 2.4 \times 10^4$ neurons, the number of circuits per neuron jumps from 8 to 4 and the number of used chips therefore decreases, which in turn simplifies the wafer routing. The resulting speedup is more accentuated for marocco because a larger fraction of its total runtime is spent on the routing of Layer 1 connections.

Runtime Synapse Normalization

Normalizing the runtime to the number of model synapses allows for a simple performance comparison of both workflows and enables runtime estimates for other random network models given their number of synapses. The results are presented in Figure 8.10.

Firstly, the network setup times scale directly proportional with the number of model synapses, as mentioned earlier. This results in the constant ratio for both, PyHMF and the old PyNN backend. Overhead causes a minor aberration, which can only be seen for the old workflow, small network instances and a connection probability of $p = 1\,\%$.

The non-constant ratio for the mapping implementations indicates a more complex scaling behavior. For both implementations, the assignment of hardware synapses is

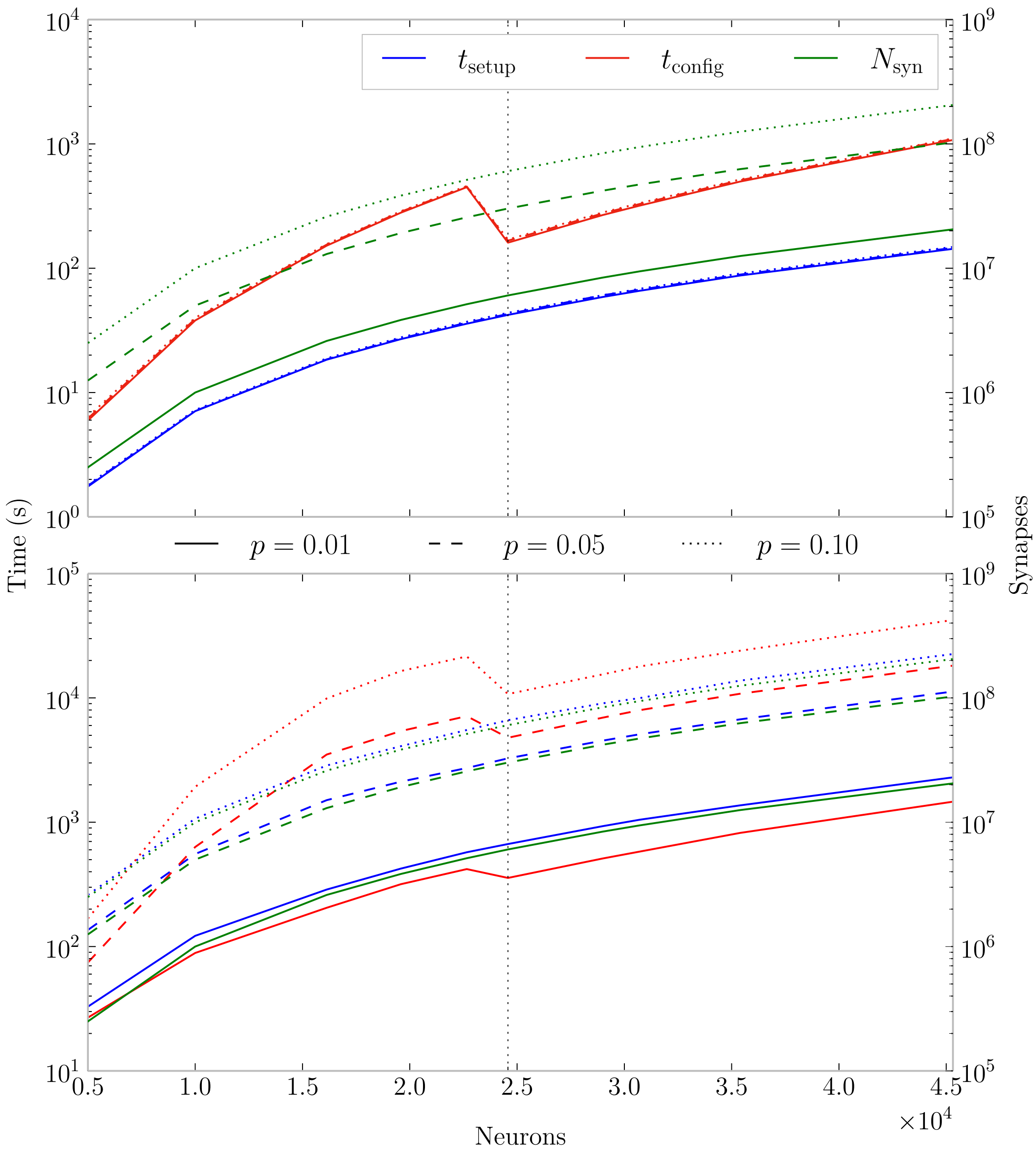


Figure 8.9: Runtime measurement for homogeneous random networks conducted individually for the network setup (blue) and mapping (red). The new and old workflows are shown in the lower and upper plots, respectively. Line styles and grid lines are explained in Figure 8.8. The number of model synapses is shown in green and grows according to $p \cdot N^2$. The setup times for both workflows follow the number of synapses. Furthermore, the runtime of the new workflow is mostly independent of the connection probability p, which is a consequence of the underlying data structures, see text. The reduction in neuron size at $N = 24\,576$ (grid line) reduces the active wafer area and therefore speeds up the wafer routing.

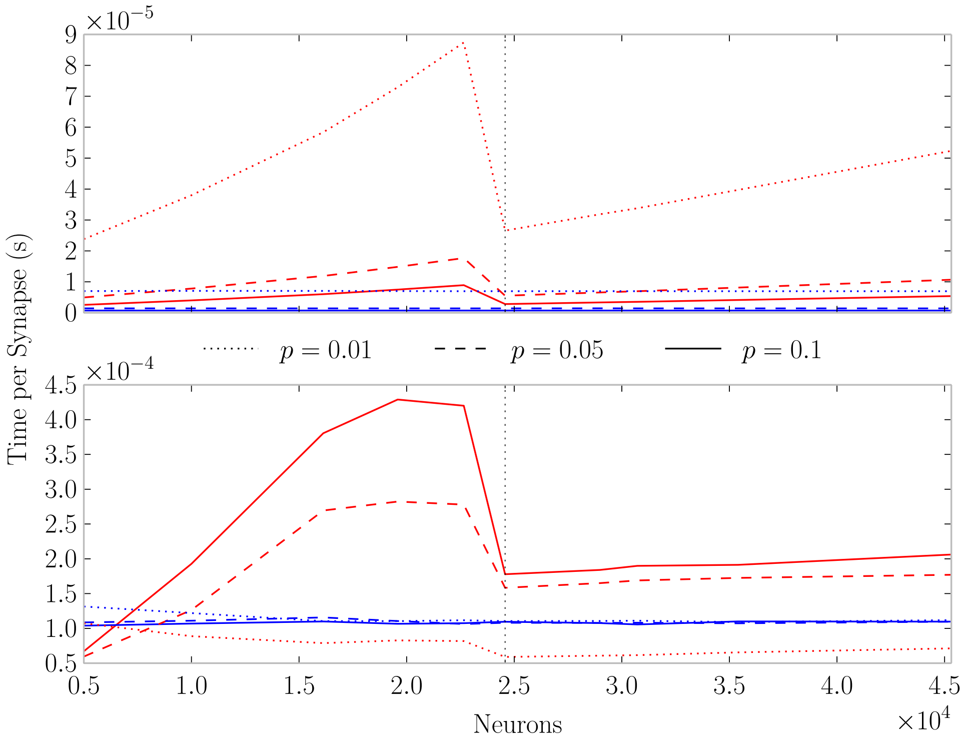


Figure 8.10: The setup (blue) and mapping (red) times from Figure 8.9 normalized to the number of synapses. Results for the new and old workflow are shown in the upper and lower plot, respectively. Notably, marocco maps single synapses faster for higher connection probabilities, which is an immediate consequence of the different synapse data structures, see text. Mapping synapses via the MappingTool takes only twice as long as setting up model synapses via PyNN, which is surprising given the higher complexity of the mapping task. Note that the ordinate ranges differ, the new workflow is consistently faster.

assumed to scale linearly with the number of synapses. Thus, the functional dependency is imposed by other mapping steps, mainly the Layer 1 wafer routing. The overhead is attenuated for high connection probabilities, as relatively more time is spent on the assignment of hardware synapses. Similarly, the influence of the wafer routing is weaker for large network instances. Where Layer 1 network congestions causes wafer routing iterations to terminate earlier, resulting in a more gradual slope of the mapping time. The contribution of the wafer routing is suddenly reduced as the number of circuits per neuron decreases from 8 to 4 at $N = 24\,576$. There, the active wafer area is reduced in size, which simplifies the wafer routing and effectively reduces the mapping time per synapse.

Notably, the mapping time per synapse is shorter for higher connection probabilities using marocco. This indicates that the synapse array routing is carried out efficiently. The slightly increased runtimes are divided by a larger number of model synapses. For the MappingTool it is the other way round, the time per synapse is longer for high connection probabilities, as a result of the incidence list representation of synapses. Thus, marocco is more scalable towards networks with high connection densities.

Moreover, a plateau is reached at $\approx 1.6 \times 10^4$ neurons that coincides with the hardware usage plateau observed earlier for marocco. As no further synapses can be realized per HICANN, the time spent on the synapse routing per chip becomes constant. This shows that the MappingTool spends a larger fraction of the mapping time for the synapse array routing compared to marocco, where no regression can be observed. Instead the runtime is dominated by the wafer routing.

In a first order approximation, the runtime per synapse for the network setup and the mapping is considered constant for large network instances and $p = 10\,\%$. Averaging all measurements with $N \geq 24\,576$ for the old workflow leads to

$$\begin{aligned} {}^{T_{\mathrm{PyNN}}}\!/\!_{N_{\mathrm{syn}}} &= (1.09 \pm 0.02) \times 10^{-4}\,\mathrm{s} \\ {}^{T_{\mathrm{MappingTool}}}\!/\!_{N_{\mathrm{syn}}} &= (1.90 \pm 0.01) \times 10^{-4}\,\mathrm{s} \qquad . \end{aligned}$$

The error is given by the standard deviation and is small for both components, ratifying the normalization. Note that the relative amount of time spent on setting up a model synapse compared to mapping one is approximately $^1\!/\!_2$. Given the relative complexity of both tasks, this indicates that the PyNN backend sets up the network rather inefficiently.

Accordingly, the normalization is applied to the new workflow resulting in

$$\begin{aligned} {}^{T_{\mathrm{PyHMF}}}\!/\!_{N_{\mathrm{syn}}} &= (7.23 \pm 0.01) \times 10^{-7}\,\mathrm{s} \\ {}^{T_{\mathrm{marocco}}}\!/\!_{N_{\mathrm{syn}}} &= (3.84 \pm 0.88) \times 10^{-6}\,\mathrm{s} \qquad . \end{aligned}$$

Again, the error on the normalized network setup times is small, as expected for linear scaling with the number of synapses. Whereas runtimes of marocco deviates stronger due to the more complex runtime dependency. Still, the error is sufficiently small to justify the normalization. In case of the new workflow, the ratio of the time needed to set up a synapse versus mapping one is about $^1\!/\!_5$. Given the higher complexity of the mapping task, this is more reasonable compared to the old workflow.

In conclusion, PyHMF is about 150 times faster than the old PyNN backend and marocco can map homogeneous random networks with $p = 10\,\%$ approximately 50 times faster than the MappingTool.

Memory Performance

The different synapse data structures do not only affect the runtime, but also have a significant impact on the memory consumption, which is shown in Figure 8.11. A graph in matrix representation with $|V|$ vertices requires $\sim |V|^2$ memory, independent of the actual number of connection (Even and Even, 2011). The memory consumption for marocco is therefore independent of p and scales with N^2. Thus, pathological networks with many neurons but sparse connectivity can be constructed such that the N^2 memory dependency becomes a scalability bottleneck. However, this is only

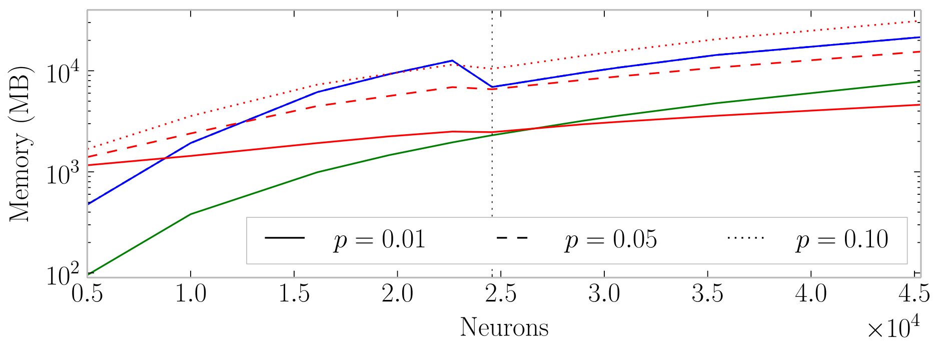


Figure 8.11: Memory consumption for the mapping of homogeneous random networks as a function of the number of neurons N. Results for the MappingTool and marocco are shown in red and blue, respectively. The size of a correspondingly sized connection matrix is shown in green ($\sim N^2$). For larger networks and low connection probabilities p, the MappingTool consumes less memory because connections can be stored more economically in lists than matrices, as used by marocco. However, the MappingTool outruns marocco, regarding memory consumption, towards higher connection probabilities due to a higher overhead per synapse. Random networks are the worst-case scenario. The penalty for connection matrices is significantly less for structured models. For the scenario presented in Figure 8.17, marocco is consistently more efficient.

relevant for unstructured, fully connected networks, which are also pathologic regarding an HMF implementation. Alternatively, such networks can be represented in PyHMF more economically using population views, effectively rendering the representation an incident list, see Section 3.2.1. A memory comparison for both workflows and structured networks is given in Figure 8.17, showing a clear memory advantage for the new workflow.

Despite the N^2 memory dependency, full-wafer worst-case random networks with more than 4.5×10^4 neurons have been mapped, which required approximately 21 GB of memory. A reasonable amount, given that the connection matrix itself consumes 7.6 GB of RAM and has to be stored twice for the model synapses and the tracking of realized synapses. Theoretically, list representations store sparse graphs more memory efficiently. However, the memory footprint of the old workflow for $p = 5\,\%$ is already similar to the new workflow and exceeds it for $p = 10\,\%$. The largest corresponding network instance, for example, results in a 45 % higher memory consumption for the old workflow.

On the bottom line, storing synapses in a list representation gives no clear memory advantage, but generally slows down access times. Ultimately, using connection matrices in PyHMF has been an advantageous decision.

8.4.2 Persistent Storage of Wafer Configurations

The ability to store wafer configurations persistently on disk is required to reevaluate or reproduce HMF experiments at a later time. The StHAL configuration containers described in Section 4.3 provide the necessary serialization functionality by means of `boost::serialization` (Ramey, 2004).

Storing any full-wafer configuration in binary format requires 205 MB, which is more than the 44 MB stated by Brüderle et al. (2011). At that time, StHAL and HALBe did not exist, their estimate was based purely on the hardware configuration space, rather than a corresponding software representation. For software, word access granularity, memory alignment and implementation overhead increase the representation size. However, during serialization StHAL omits HICANNs that have not been configured. Thus, efficiently producing smaller configurations for experiments using only part of the system.

Moreover, wafer configurations typically have a low entropy due to similarly configured components and can therefore be efficiently compressed using e.g., bzip2 (Seward, 2014). For example, a full-wafer default configuration can be reduced in size from 205 MB to 31 kB. To acquire a representative estimate on the necessary storage space for configurations of higher entropy, full-wafer homogeneous random networks have been mapped and serialized. The resulting configurations could be compressed from initially 205 MB to approximately 12 MB, a 17 fold reduction in size. Thus, around 8.7×10^4 full-wafer configurations of real network experiments can be stored on a single 1 TB disk. Note that this assessment only captures the topological configuration space, whereas spike inputs require additional memory depending on the spike count.

8.4.3 Mapping Parallelization

The improved performance for marocco has been achieved by shared-memory parallelization in addition to the utilization of more efficient algorithms. The fraction of time spent in concurrent code regions has been measured to study the effectiveness of the current parallelization. Measurements and the ideal speedup are shown in Figure 8.12. The latter is defined by Amdahl's law (Amdahl, 1967), which parametrizes the speedup S as a function of the parallel threads of execution n according to

$$S(n) = \left(B + \frac{1}{n}(1 - B)\right)^{-1} \quad . \tag{8.7}$$

Where $B \in [0, 1]$ is the relative amount of time spent in serial execution. The ideal speedup is the maximum speedup that can be achieved for an infinite number of threads and is given by $\lim_{n\to\infty} S(n) = {}^{1}\!/\!{}_{B}$.

The time spent in parallel regions declines towards larger network sizes, since more time is spent on the wafer routing. Speedup factors of 3 and more can be achieved for random networks of up to 1.2×10^4 neurons. Similarly, higher speedups can be expected for structured networks, where less long-range connections have to be routed.

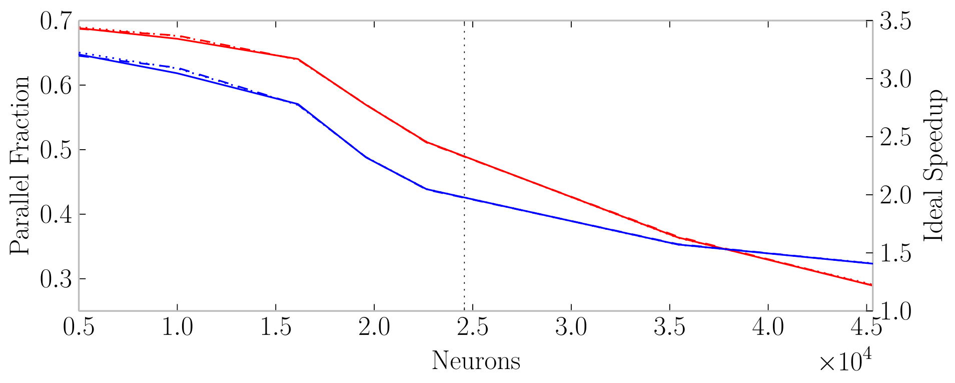


Figure 8.12: The relative amount of time spent by marocco in concurrent code regions and the corresponding ideal speedup, according to Amdahl's law, are shown in red and blue, respectively. For larger network instances the sequential wafer routing takes more time, which results in a speedup regression. Generally, the efficiency of parallelization depends on the network topology, with random networks being the worst-case. The mapping of networks with simpler inter-chip connectivity can be parallelized more efficiently.

According to Equation (8.7), the effective speedup also depends on the number of parallel threads n. The mapping parallelizes on an individual HICANN level, thus the number of parallel threads is bound by the number HICANNs that are used for a given neural network instance. However, even for the smallest random network instance with 5000 neurons, 85 HICANNs are used, leaving enough room for parallelization on modern multi-core machines.

Starting from $N = 16\,128$, the ideal speedup declines more rapidly. This coincides with the saturation of synapse usage shown in Figure 8.8, which causes the time spent on the synapse driver and synapse array routing per chip to become constant, whereas the time for wafer routing further increases. Consequently, the relative amount of time spent in concurrent code regions is reduced, therefore reducing the effectiveness of the parallelization. Notably, this is only worst-case behavior and characteristic for the all-to-all inter-chip connectivity required for random networks.

8.4.4 Layer 1 Network Optimization

Finally, homogeneous random networks are used to explore topology improvements for future versions of the HMF. In an initial study, the swap of Layer 1 buses between adjacent HICANNs is optimized. Afterwards, alternative crossbar layouts are tested regarding their ability to implement random networks.

Inter-HICANN Bus Swap

Horizontal and vertical Layer 1 buses are swapped by two across HICANN boundaries, as explained in Section 1.5.2. This enables adjacent chips to output events on the same

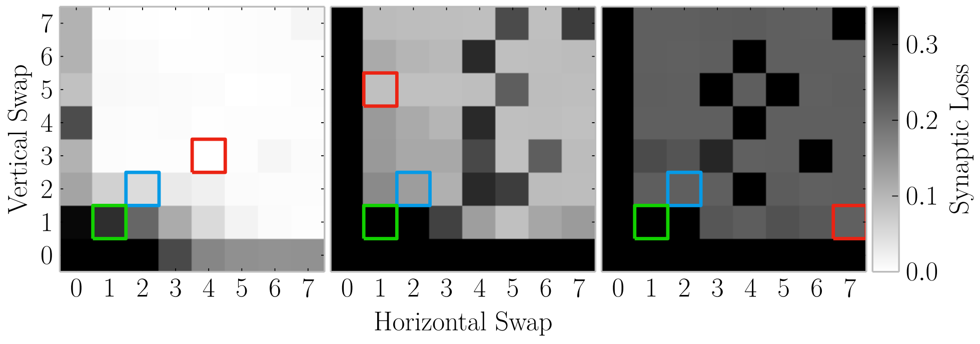
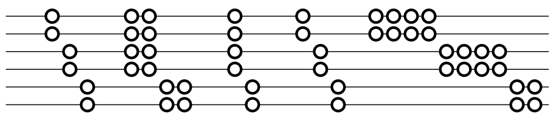


Figure 8.13: Synaptic loss as a function of the horizontal and vertical Layer 1 bus swaps. The results have been measured for homogeneous random networks with a connection probability of $p = 5\,\%$ and 5000 (left), 10 000 (center), and 16 128 (right) neurons. The topologies of the first and second HICANN version are marked by the green and blue boxes, respectively. The second version performs significantly better. Furthermore, the minimal losses are indicated by red boxes, however, no clear winner exists and several combinations produce similar results. Note that synaptic loss is cut at 35 % to increase contrast, only configurations with low synaptic loss are of interest.

local bus segments without interfering with one another. Furthermore, vertical bus swaps make the routing more flexible with respect to the sparse crossbar and select switch matrices. Historically, the bus swap has already been changed. The first version of HICANN implemented a bus swap of one versus two for the second HICANN version. The single-bus swap had been tested via routing simulations carried out by Fieres et al. (2008). However, a different crossbar and select switch layout ultimately had to be implemented due to chip design constraints (Grübl, 2014a). However, changing the switch layout without changing the bus swap caused a routing regression for the first HICANN version. This study verifies the improved routing capabilities of the second version of HICANN and provides guidance for future hardware topologies. Notably, marocco can easily be adapted to arbitrary bus swaps by changing the Layer 1 graph representation introduced in Section 6.4.1.

The synaptic loss for different bus swaps and homogeneous random networks with 5000, 10 000 as well as 16 128 neurons, a fixed connection probability of $p = 5\,\%$ and 8-circuits per neuron is shown in Figure 8.13. Bus swaps for horizontal and vertical buses have been studied separately, they are denoted along the abscissa and ordinate. The bus swaps for HICANN version 1, version 2 and the local optimum are marked by the green, blue and red boxes, respectively. Clearly, the topology change improved the routing performance. Even slightly better configurations exist, though there is no clear overall winner. The optimum depends on the particular network instance and multiple configurations lead to similar results. The best average result over all network instances has been produced by a horizontal bus swap of 7 and a vertical bus swap of 4. However, implementing wider bus swaps might not be desirable as more chip design resources are required (Grübl, 2014a).



	D	**A1**	**A2**	**A3**	**A4**
S	32	32	16	16	64
T	±2	±4	±4	±2	±2
B	1	2	1	1	4

Figure 8.14: Cutouts of the current crossbar layout **D** and alternative layouts **A1** to **A4** alongside their respective sparse switch parameters according to Figure 1.4. The + and − refer to the left and right half of the crossbar, respectively. These layouts have been studied regarding their ability to improve the realization of dense, long-range connectivity in homogeneous random networks.

Layer 1 Crossbar Switch Layout

In a second study, alternative crossbar layouts are explored regarding their suitability for future chip revisions. The next revision is planned to be fabricated in a 65 nm technology, which is expected to at least double the number of switches per crossbar that can be implemented (Schemmel, 2014a).

The four alternative crossbar layouts AX illustrated in Figure 8.14 have been used. Each layout implements 1024 switches, twice as many as the current crossbar layout D. The alternatives are chosen such that the arrangement of switches on the grid of 64 horizontal and 256 vertical buses shows a high degree of symmetry in accordance with the current layout.

The synaptic loss for the inhomogeneous random network instance with 16 128 neurons, a connection probability of $p = 5\,\%$ and the different crossbar layouts is shown in Figure 8.15. The routing performance is measured as a function of the defect rate f for random Layer 1 bus defects, i.e., $f = 20\,\%$ means that stochastically every fifth bus is unavailable. The bus swap of the second HICANN version is used.

The results are similar for all layouts, indicating that bus swaps and crossbar layout are well tuned and the detouring mechanism of the wafer routing works efficiently. For the current crossbar switch layout, sparseness is only a minor routing limitation. Even a fully occupied crossbar layout did not significantly improve the synaptic loss (results not shown). However, slowed down the mapping because more detours and vertical bus options had to be considered during the wafer routing. The measurement has been repeated for other bus swap configurations. There, changing the crossbar layout helped in some cases to improve the routing performance. For example, for a horizontal and vertical bus swap of 3 the alternative layouts could reduce the synaptic loss by approximately 8 %. On the bottom line, sparseness of crossbar switches is a minor limitation as long as the topology is well tuned. Any future topological changes should be accompanied by appropriate routing studies. The Layer 1 representation within marocco can easily be updated and therefore provides the necessary means to conduct these studies.

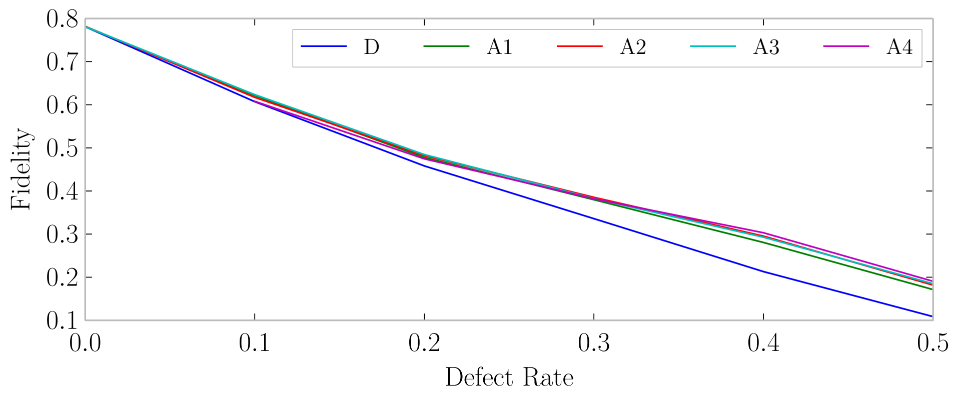


Figure 8.15: Model fidelity for homogeneous random networks with 16 128 neurons and a connection probability of $p = 5\,\%$ for alternative crossbar layouts as a function of the Layer 1 bus defect rate. The next chip revision is expected to support more switches, therefore, layouts with twice as many switches as the current layout **D** have been studied. These layouts **A1** to **A4** are illustrated in Figure 8.14. The current layout is well tuned. The alternative layouts with twice as many switches provide only minor improvements for defect rates below 20 %, however, routing around defects is simplified for high defect rates.

An analysis of the individual causes for lost Layer 1 connections revealed that horizontal bus bandwidth is currently the dominating resource bottleneck. Nonetheless, extra switches can provide additional detouring options, which result in slightly improved results for defect rates of $f = 20\,\%$ and higher. The current wafer system supports random network implementations with synaptic loss below 20 % of up to 8×10^4 to 1.5×10^4 neurons depending on the connection densities, see Sections 8.4, 8.6 and 8.7. In case larger random networks or more long-range connectivity are requested for future hardware revisions, the horizontal Layer 1 network bandwidth needs to be increased.

8.5 Layer 2/3 Attractor Model

The layer 2/3 network is a cortically inspired attractor model developed by project partners at the KTH in Stockholm (Lundqvist et al., 2006, 2010). The hypercolumn and minicolumn structure of the network is explained in Figure 8.16. The two different winner-take-all (WTA) topologies ensure that whenever a minicolumn is active, it locally inhibits its competitors by exciting corresponding and inhibiting orthogonal minicolumns in other hypercolumns. The dense local connectivity within hypercolumns and the sparse long-range connectivity in between makes the network particularly susceptible for a mapping to the wafer system. Its structure enables hardware realizations with significantly fewer distortions than e.g., homogeneous random networks. Furthermore, the network is part of the FACETS demonstrator benchmark library and has previously been studied by Brüderle et al. (2011).

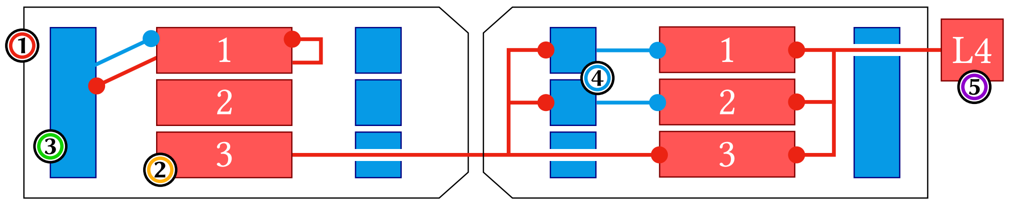


Figure 8.16: Schematic of the layer 2/3 attractor network model. Excitatory and inhibitory populations are shown in red and blue, respectively. Note that every kind of connection is outlined once only. The model consists of hypercolumns **(1)**, which are internally composed of multiple minicolumns consisting of pyramidal neurons **(2)**. Minicolumns are recurrently connected and to a corresponding minicolumn in every other hypercolumn. Furthermore, minicolumns are connected to a local pool of inhibitory basket cells **(3)** in a soft WTA fashion. Activity in one minicolumn therefore suppresses activity in the other minicolumns. Additionally, minicolumns inhibit activity of orthogonal minicolumns in other hypercolumns via intermediate remote RSNP cells **(4)** realizing a second, strong WTA topology. Lastly, the attractor network dynamics are driven by external cortex layer 4 spike input **(5)**.

8.5.1 Workflow Performance Comparison

In a first scaling experiment, configurations generated by the MappingTool and marocco are compared for different instances of the layer 2/3 attractor model. The network instances range from 891 neurons with 1.6×10^5 synapses to 10 692 neurons with approximately 2.4×10^6 synapses. The corresponding hyper and minicolumn dimensions are listed in Appendix A.2. The results for neuron configurations of 8, 12 and 16 circuits are shown in Figure 8.17. Note that the MappingTool only supports neuron configurations that are powers of 2, therefore the 12 circuit trial has been skipped.

The measured synaptic loss varies across the range and does not monotonously increase towards larger networks because the topology changes for different hyper and minicolumn proportions.

For small network instances, marocco generates less synaptic loss and configurations with 12 circuits per neuron perform best. From around 1.5×10^6 synapses onwards, marocco produces higher synaptic loss than the MappingTool for 16-circuit configurations. Still, the 8 circuit configurations remain slightly more efficient until approximately 2.3×10^6 synapses. Interestingly, the synaptic loss generated by the MappingTool remains almost constant between 1.5×10^6 and 2.5×10^6 synapses, whereas the loss for marocco keeps going up. Understanding this effect requires detailed knowledge about the implementation and has therefore been discussed with one of the MappingTool authors: Vogginer (2014a). Still, the nature of the effect remains unclear, but might possibly be an artefact caused by the different placement strategies. For example, placing populations side by side rather than circular around the wafer center might benefit the inter-chip connectivity in this particular case.

The runtime in Figure 8.17 (b) is given as the combined runtime spent on both, the network setup and the mapping. The new workflow is about an oder of magnitude faster, even though the performance gain is smaller compared to the Hellfire chain

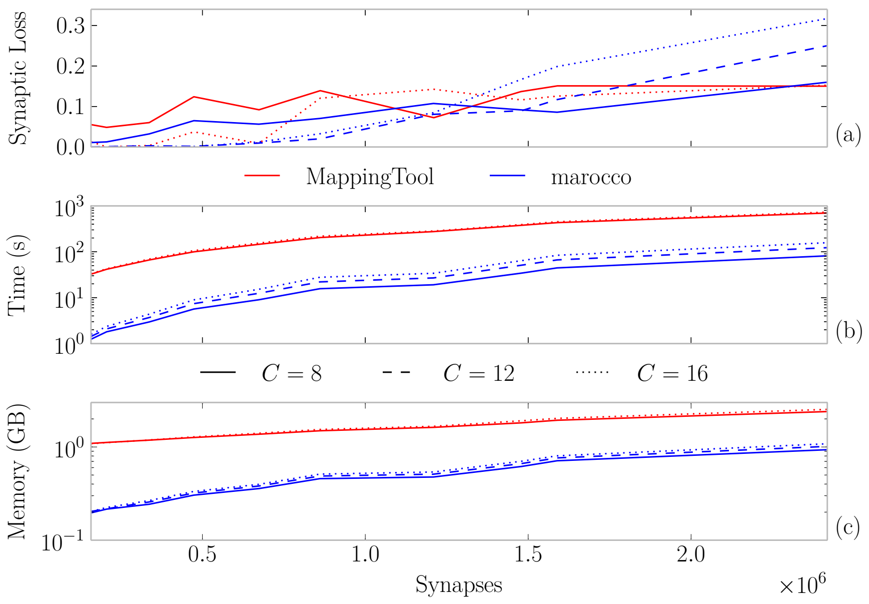


Figure 8.17: A scaling study for the old workflow (red) and new workflow (blue) on the layer 2/3 attractor model. The different line styles refer to neurons built from 8, 12 (marocco only) and 16 circuits. The synaptic loss for marocco and the MappingTool is compared in **(a)**. The minimal loss has been produced by marocco for 70 % of the network instances. Setting up and mapping the networks is generally faster with the new workflow **(b)**. The new workflow is also more memory efficient **(c)**, despite using connection matrices to represent synapses.

and homogeneous random networks in Sections 8.3.2 and 8.4. On close inspection, one can see that the speedup slightly regresses at the point where the synaptic loss of the MappingTool stagnates, while the loss for marocco keeps going up.

The results for the memory consumption of both workflows show a behavior comparable to the runtime results. Initially, the new workflow consumes about an order of magnitude less memory than the old one. This advantage becomes smaller as the network instances grow larger and slightly regresses at around 1.5×10^6 synapses similar to what has been observed for the mapping runtimes. Even though the new workflow stores synapses in matrices rather than in lists, it consumes on average 4 times less memory. The advantages and shortcomings of both synapse representations are discussed in more detail for the homogeneous random networks in Section 8.4.

8.5.2 Guided Placement

The layer 2/3 attractor network implements dense local and sparse long-range connectivity, as mentioned earlier. Here, the possibility is explored to guide the mapping towards a hardware representation that exploits the local structure to minimize distortions.

Different placement strategies have been tested to lay out the 3D column structure

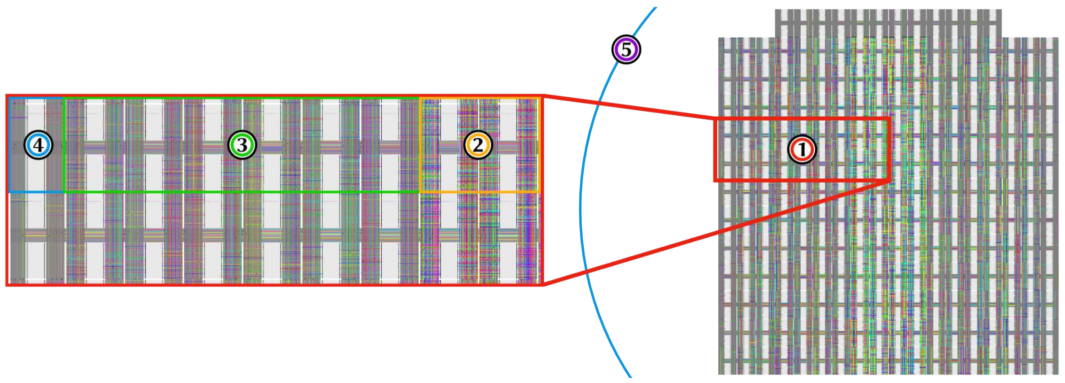


Figure 8.18: Visualization for a wafer mapping of the layer 2/3 attractor model with 24 minicolumns per hypercolumn and 24 hypercolumns. Every hypercolumn **(1)** consists of regular spiking non-pyramidal (RSNP) **(2)**, pyramidal **(3)** and basket **(4)** cells placed to individual chips. Active Layer 1 buses are colorized. The Layer 1 connection densities are emphasized towards the center where the RSNP cells have been placed. They receive input from any other hypercolumn. Moreover, the wafer size is outlined in blue **(5)**. The figure has been published by the author internally in UHEI and TUD (2014).

on the 2D wafer substrate, but only the most successful one is presented in Figure 8.18. Individual hypercolumns are laid out horizontally to 9 adjacent chips. Two hypercolumns facing each other in the center are realized per horizontal line of HICANNs. Vertically, 12 of these double columns can be stacked on top of each other, resulting in a total of 24 hypercolumns. Every hypercolumn contains 24 minicolumns, each consisting of 29 pyramidal, 2 RSNP and 1 basket cell, thus, 18 432 neurons in total. The resulting network implements approximately 4.5×10^6 synapses. The number of circuits per neuron is set individually for each cell type to 4 per pyramidal, 8 per basket and 16 per RSNP cell according to their relative input requirements. Furthermore, pyramidal and RSNP cells, in particular, have been placed towards the center of the wafer. Basket cells on the other hand have local connections only and are placed to the outer most HICANNs. This placement strategy helps to reduce the average Layer 1 route length, as long-range connectivity, implemented by these cells, is limited to the wafer center. The increased connectivity can be observed in Figure 8.18, Layer 1 bus allocation rates are higher in the area where RSNP cells have been paced to. The results for the guided mapping and a default mapping are given in Table 8.1, demonstrating the effectiveness of this placement strategy. The synaptic loss is cut by approximately half, from initially 34.8 % to 17.1 %.

So far, possibilities to optimize the mapping of an existing network model have been discussed. However, network models may also be tuned to enable less distorted implementations on the HMF. Therefore, the layer 4 spike input provided by a global pool of 5000 sources has been replaced by 24 smaller local input pools with 500 sources each, which result in 1.2×10^4 independent sources. Note that this only changes the total number of spike source, while the total number of synapses in the network remains

Placement	default		guided	
Input	global	local	global	local
Loss Layer 1 (%)	29.4	19.5	4.3	4.2
Loss (%)	34.8	25.8	17.1	8.8
t_{PyHMF} (s)	28.1 ± 0.1	21.9 ± 0.1	28.1 ± 0.1	21.9 ± 0.1
t_{marocco} (s)	257.0 ± 0.5	143.4 ± 0.5	79.5 ± 0.1	24.1 ± 0.2

Table 8.1: Mapping results for a large layer 2/3 attractor model instance with 18 432 neurons, 4.5×10^6 synapses and either global or local spike input pools, see text. The network has been mapped with the default placement as well as the guided custom placement illustrated in Figure 8.18. The errors are small and given by the standard deviation. The error on synaptic loss for different random seeds and varying stochastic connectivity is negligible and has therefore been omitted.

unaltered. In case of the global pool, the spike input placement algorithm inserts all sources close to the wafer center, where congestion is already high due to the long-range RSNP connectivity. From a model point of view, the global pool has already been a compromise introduced as part of the FACETS demonstrator adaptation. Ideally, each neuron receives non-overlapping, uncorrelated Poisson input. Therefore, implementing more independent Poisson sources in local pools is closer to the original model. Using the local input pools reduces the synaptic loss from 17.1 % to 8.8 %, see Table 8.1.

It has not been possible to compare the results for both mappings because the large 24×24 network instance resulted in a memory error for the MappingTool. Nevertheless, the network setup times have been measured. Setting up the network with the old PyNN backend takes 1291 s and 2500 s for the global and local input pools, respectively. The corresponding setup and mapping times for marocco are given in Table 8.1. Thus, PyHMF constructs the network representation approximately 46 times faster for the global pool and 114 times faster for the local ones.

8.6 June Cortical Column

The June cortical column is a model provided by project partners at the *Forschungszentrum Jülich* (Potjans and Diesmann, 2011). It models cortex layers 2/3, 4, 5 and 6 (Shipp, 2007) and is designed to mimic the respective firing behaviors observed *in vivo*. The original model has been downscaled and translated into a corresponding PyNN model description by Sacha van Albada. This lead to the here studied model, which comprises 7713 neurons and approximately 5.4×10^6 synapses.

The connection structure across individual layers is visualized in Figure 8.19. Most layers are connected, leading to Layer 1 all-to-all connectivity similar to the worst-case connectivity for homogeneous random networks, see Section 8.4. However, in this case, connection densities are higher towards the main diagonal. This means that recurrent connectivity within layers is emphasized, thus weak locality exposed. Still, the global

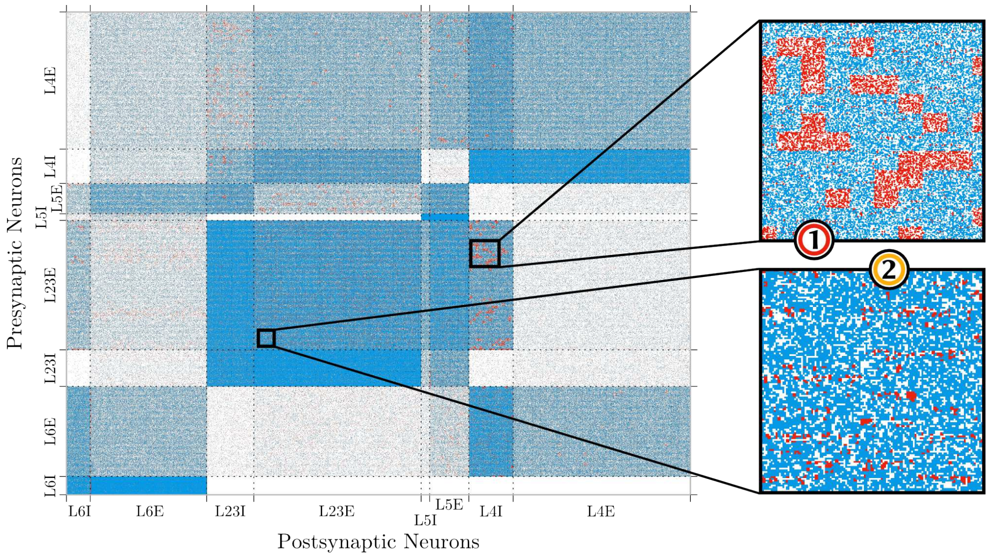


Figure 8.19: Cortex layer 2/3, 4, 5, 6 connectivity for the 7713 neuron June cortical column network. Each pixel represents a synapse. The model has been mapped with 12 circuits per neuron. Synapses colored blue have been realized, whereas red ones are lost. Connections that are lost during the Layer 1 routing lead to small blocks of lost synapses **(1)**. Local resource exhaustion, on the other hand, generates sparse loss **(2)**, which is typical for densely connected areas leading to high input requirements.

connection structure impedes efficient guided placement strategies to maximize model fidelity. The default strategy is used to place neurons in the order they appear in the model description spiraling outwards from the center of the wafer, as explained in Section 6.1. Nonetheless, different population orderings have been explored to minimize the loss. Ultimately, the ordering indicated in Figure 8.19 has been used, starting with layer 6 in the center and placing layer 4 last.

The network has been mapped for hardware neurons consisting of 4, 8, 12 and 16 circuits, yielding network fidelities of $71\,\%, 92\,\%, 96\,\%$ and $94\,\%$, respectively. For small hardware neuron configurations, more synapses are lost locally because of reduced input counts, whereas larger configurations render the Layer 1 routing more challenging by increasing the active area on the wafer. The corresponding network setup and mapping times for the 12-circuit neuron configuration have found to be 12 s and 19 s, respectively. This corresponds to a mapping time of 3.5×10^{-6} s per synapse, which is approximately 10 % less compare to homogeneous random networks, see Section 8.4.1.

Influence of Synaptic Loss on the Network Dynamics

In this section, the influence of mapping induced inhomogeneous synaptic loss on the June network dynamics is briefly studied by means of software simulations. Software studies have shown to be a valuable tool to prepare networks for the HMF (Brüderle et al., 2011). In principle, the ESS is designed to simulate configurations of models that have been mapped, however, the HALBe integration does not yet correctly account for analog neuron parameters (see Sections 2.2.5 and 8.3.3). Consequently, the model synapses that have been mapped successfully are used in NEST-based (Diesmann and Gewaltig, 2002; Eppler et al., 2008) follow-up simulations. The mapping of synapses is automatically tracked by marocco and can conveniently be accessed by means of the PyMarocco interface, see Section 6.8.

Notably, setting up the network via PyHMF takes only (22.4 ± 0.1) s compared to (311 ± 2) s in PyNN using the original NEST backend. This means, PyHMF is approximately 14 times faster, although the fast fixed probability connector provided by the NEST backend is used, which has specifically been designed to speed up the creation of synapses.

The simulations have been conducted using the synapses that have been mapped for the least distorted 12-circuit trial. This means, only 4 % of the synapses are missing. However, the loss is not homogeneous, e.g., sets of synapses are lost during the wafer routing whenever Layer 1 connections cannot be established. Such inhomogeneities can be a problem and affect the network dynamics. Thus, reference simulations have been carried out for homogeneous loss to compare the results and study the impact.

The combined average firing across all layers has found to be (5.55 ± 0.16) Hz, (5.54 ± 0.21) Hz and (5.83 ± 0.26) Hz for the original network, with mapping distortions and 5 % homogeneous loss, respectively. This means, all average firing rates are compatible within uncertainties. Neither the loss nor the inhomogeneities changed the response significantly, which is a promising result regarding a future hardware implementation.

Further results for individual cortical layers and simulations for a wider spectrum of homogeneous synaptic loss are presented in Appendix A.3. However, a comprehensive analysis of the model dynamics is beyond the scope of this thesis.

8.7 EPFL Cortical Column

The 10^4 neuron EPFL cortical column (Markram, 2006) models cortex layers similar to the June network in the previous section. The network is implemented as a fully expanded net list derived from biological findings. This net list also contains multiple connections between pairs of neurons, e.g., to realize complex stimulation patterns using multiple synapses with different delays. However, neither does the HMF currently support programmable delays nor can a single projection in PyHMF represent multiple synapses between a pair of neurons natively due to the underlying connection matrices. Both shortcomings can be handled in the future by FPGA firmware and PyHMF revisions. Even with PyHMF support in place, the network would require compensation because realizing all synapses with an equal, fixed transmission delay may result in overly strong stimulation. In a first order approximation, multiple synapses have been

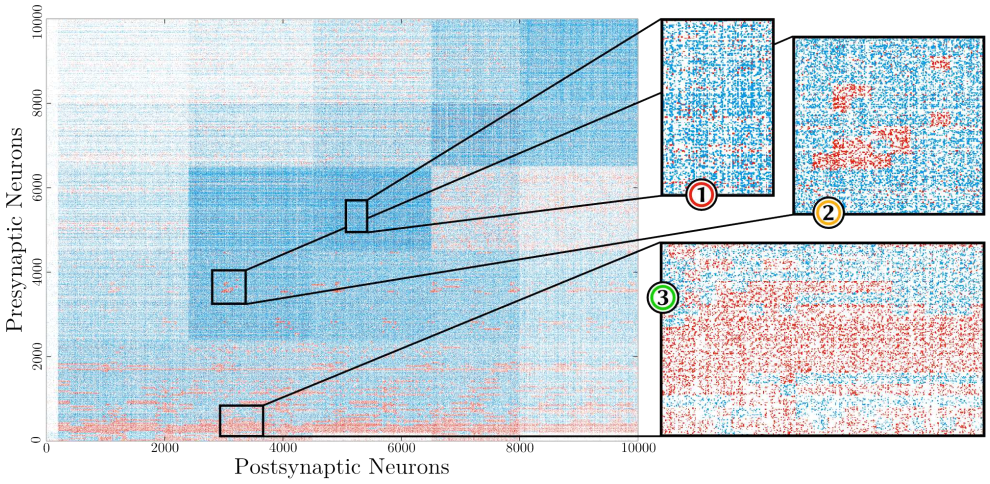


Figure 8.20: Connectivity of the 10^4 neuron EPFL cortical column model. Multiple synapses between pairs of neurons have been collapsed, resulting in approximately 3.6×10^6 synapses. Realized and lost synapses are shown in blue and red, respectively. Synapses that are lost during the local routing are typically sparse **(1)**, whereas rejected Layer 1 connections lead to small blocks of lost synapses **(2)**. More synapses are lost towards the bottom **(3)** because neurons with lower indices have been placed further away from the wafer center. They are therefore routed in later wafer routing iterations when Layer 1 congestion is already high.

collapsed into one by picking only the strongest efficacy. This approach significantly reduces the number of synapses from 8×10^6 to 3.6×10^6.

The collapsed network has been mapped for different hardware neuron sizes. The corresponding connectivity and the final mapping outcome is visualized in Figure 8.20. Using 12 circuits per neuron yielded the highest network fidelity of 84 %, whereby 5.5 % of the synapses are lost during the Layer 1 wafer routing. Figure 8.20 reveals that many of the synapses lost during the wafer routing originate from neurons that are placed farther away from the wafer center. These connections are routed in later iterations when the Layer 1 congestion is already high such that their realization is less likely. Setting up the network with PyHMF and the subsequent mapping take (19.7 ± 0.5) s and (170.1 ± 0.5) s, respectively. Thus, the average time for mapping a single synapse is approximately 4.7×10^{-5} s.

8.8 Asynchronous Irregular States

Asynchronous irregular (AI) activity states are a phenomenon observed in the cerebral cortex of awake animals (Destexhe et al., 2003a). It has been shown that networks of randomly connected AdEx neurons can produce similar activity self-sustained over long periods of time, even down to small network sizes (Destexhe, 2009).

An AI state network implementation by Muller and Davison (2009) has been mapped for different network sizes with up to 4.5×10^5 neurons utilizing the whole wafer.

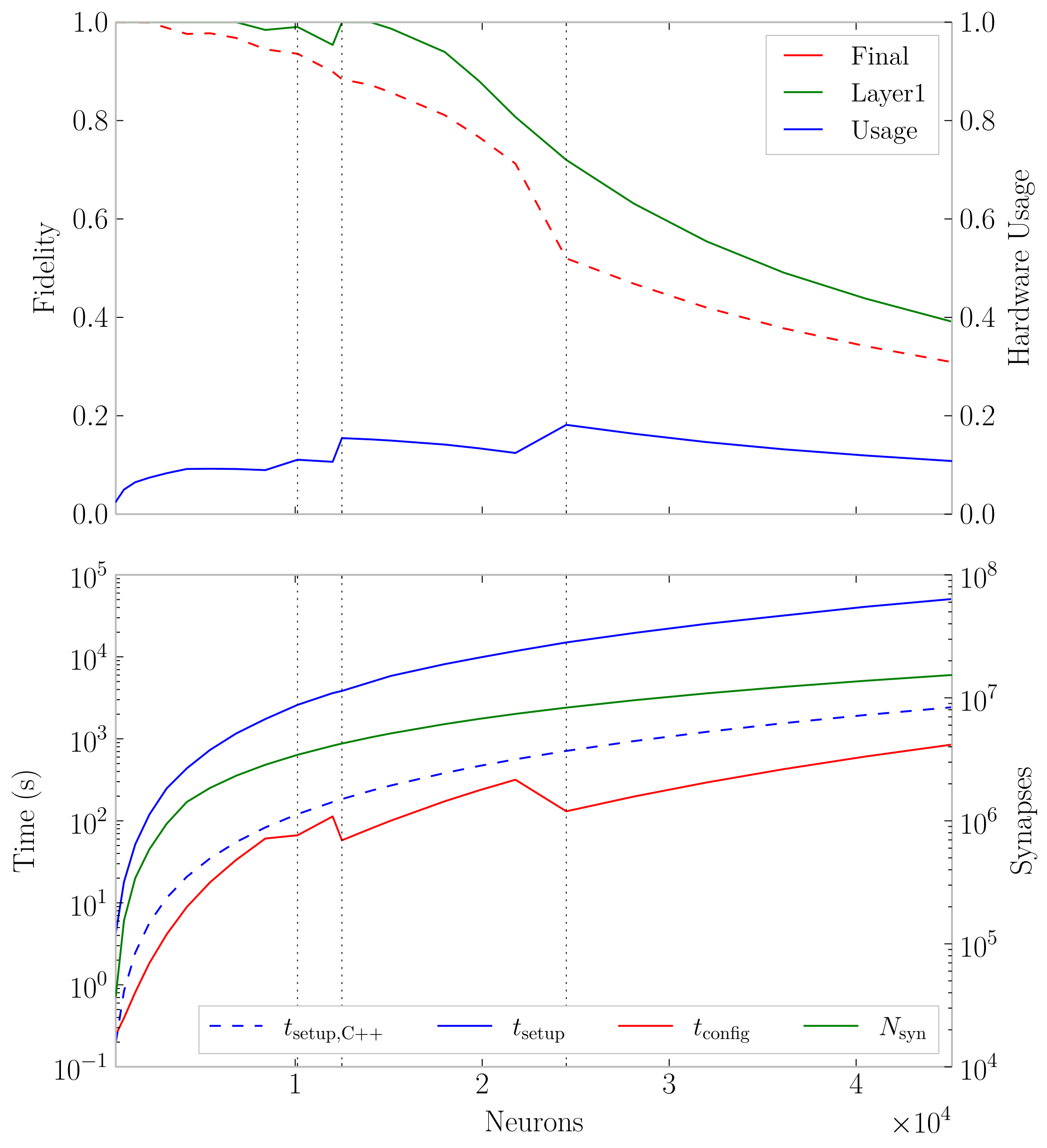


Figure 8.21: Mapping results for the AI state model. The vertical grid lines indicate the number of circuits per neuron going down from 16 over 12 and 8 to finally 4. Each time, the active area on the wafer is reduced, thus simplifying the Layer 1 wafer routing. This can be seen in the Layer 1 fidelity as well as in the mapping times. The final fidelity, on the other hand, regresses because more synapses are lost locally. This is seen most prominently for the final transition from 8 to 4 circuits. Notably, the mapping time (red) is dominated by the time required for setting up the network representations (blue). The slow distance dependent probability connector has to call into `Python` for every synapse to evaluate the distance expression. The dashed blue line shows the performance improvement for native expression evaluation in `C++`, providing a 21 fold speedup. However, the network setup is still slower than the mapping.

Neurons in this model are aligned in a 3D torus-like grid, where pairs of neurons are randomly connected with a probability of $p(d) \sim \exp(-d^2)$, where d denotes the distance between the neurons. For reasons of simplicity, the default neuron placement has been used.

The results are presented in Figure 8.21. For each network instance, the number of circuits per neuron has been chosen such that the fidelity is maximized. Vertical grid lines indicate where the number of circuits has been reduced from initially 16 over 12 and 8 to ultimately 4. For every jump in neuron size, the active wafer area is reduced, which in turn speeds up the wafer routing and the mapping accordingly. The jump from 8 to 4 circuits per neuron significantly reduces the fidelity. At this point, the number of assignable synapses per incoming Layer 1 connection becomes insufficient. For small network instances, on the other hand, the reduction improves Layer 1 fidelity because the Layer 1 bus congestion is reduced, thus more neurons can be routed via shared connections. The synapse usage is comparable to the homogeneous random network instances with a connection probability of $p = 1\,\%$, see Figure 8.8.

In this study, only the default placement has been used for reasons of simplicity. However, an optimized placement of the 3D neuron grid can likely improve the mapping by exploiting the locality of connections. Even then, the torus-like connectivity is expected to cause high inhomogeneous distortions due to the long-range Layer 1 connections wrapping from one end of the wafer to the other.

Notably, the setup times are more than two orders of magnitude longer compared to homogeneous random networks. This is due to the slow distance dependent probability connector (DDPC), which repeatedly has to call into `Python` to evaluate the distance dependency. The network setup can be accelerated by evaluating the distance expression natively in `C` or `C++`. A first version has been implemented using libmatheval (Free Software Foundation, 2014). The results are shown by the dashed blue line in Figure 8.21. Native evaluation reduces the setup times by a factor of 21.1 ± 0.3, but the DDPC remains more than an order of magnitude slower than the simpler fixed probability connector. To speed up the repeated evaluation further, other options like muparser (Berg, 2014) or mathpresso (Kobalicek, 2010) can be explored. They use just in time compilation to translate math expressions on-the-fly into native machine instructions.

Expression evaluation in `Python` captures scoped variables and symbols, which is an issue for any kind of native version. Not capturing the local scope may break compatibility for some PyNN scripts, but might still be desirable given the ability to lazily evaluate DDPCs on the server, see Section 3.2.2. Alternatively, implementations could fall back to `Python` evaluation on the user side, if the compilation of the expression string fails.

8.9 Planar Network Topologies

Before all prior mappings are summarized in the subsequent section, two simple planar topologies are briefly introduced and their key characteristics pointed out. Their respective mapping results are subsequently discussed in Section 8.10.

8.9.1 Synfire Chain Network

The Synfire chain model has originally been proposed by Baker et al. (2001) and Prut et al. (1998) to explain high precision firing patterns as observed in biological networks. The implementation at hand is based on the network described by Kremkow et al. (2010), which has been shown to have increased selectivity on the initial stimulus. The Synfire chain is a feedforward network, similar to the Hellfire chain described in Section 8.3. Groups of neurons are connected in a chain-like topology, where neurons in one chain link excite neurons in the succeeding link. Each link consists of 16 excitatory neurons and a small local pool of 4 inhibitory neurons. Neurons in the inhibitory pool are randomly connected to 15 excitatory neurons within the same link. Every excitatory neuron projects onto 9 excitatory and 3 inhibitory neurons in the subsequent link.

For this study, the network size is scaled from 1 link to full wafer size with $384 \times 4 = 1536$ links in steps of 50. The largest network instance consists of 30 720 neurons with 3.9×10^5 synapses. Similar to the Hellfire chain, neurons have been placed such that up to 4 chain links are realized per chip. The mapping results are presented as part of the summary in Section 8.10.

8.9.2 Grid Network Topology

The grid network model is a planer 2D network. One can think of it as small groups of neurons laid out in a rectangular grid with connections only between neighboring groups. Similar to the previous linear chain models (see Sections 8.3 and 8.9.1), the grid can be mapped efficiently to the 2D surface of the wafer such that connections retain their local scope. After the placement, neurons on every HICANN only connect to neurons on adjacent chips in all four cardinal directions. Neurons placed to chips at the wafer boundary have less neighbors and therefore realize less connections.

Moreover, the network geometry and connection probabilities have specifically been outfitted for the wafer system. Thus, large numbers of neurons and connections can be realized, which neither exhaust Layer 1 nor local synapse resources. Two instances of the network have been mapped to the full wafer extent with $N_8 = 384 \times 59$ and $N_4 = 384 \times 118$ neurons for configurations of 8 and 4 circuits per hardware neuron, respectively. For connection probabilities of $p = 15\,\%$ between adjacent chips, no synapses are lost. This results in approximately $S_8 = 7.6 \times 10^5$ and $S_4 = 3.1 \times 10^6$ synapses overall. The mapping results are discussed in the following summary.

8.10 Topology Exploration

The loss of synapses for all previously mapped network instances has been summarized as function of the instance size in terms of neurons as well as synapses to study the suitability of the HMF for different topologies. Generally, networks with fewer loss are considered to be more faithful representations. Still, this is not the only possible measure to judge distortions. It is considered particularly useful as it allows us to compare the vastly different networks. Other, more complex measures might include e.g., delays or analog parameters. Ultimately, the best approach depends on the model. The section closes with a brief overview of mapping runtimes normalized to the number of model synapses.

Synaptic Loss

The results for synaptic loss are illustrated in Figure 8.22. Vertical grid lines in the top neuron plot at around 2.5×10^4 and 4.5×10^4 outline the maximum number of model neurons that can be realized efficiently for 8 and 4 circuits per hardware neuron, respectively. The losses increase significantly for both, homogeneous random networks and the AI state model, as the neurons are reduced in size from 8 to 4 circuits and the number of possible inputs per neuron is therefore halved. Accordingly, network instances with higher connection probabilities are affected stronger. For example, homogeneous random network instances are displayed as blue squares in light, mid and dark blue for connection probabilities of $1\,\%, 5\,\%$ and $10\,\%$, respectively. Their increase in synaptic loss is emphasized towards higher connection probabilities.

The single vertical grid line in the bottom synapse plot at around 4.4×10^7 synapses indicates the total number of hardware synapses on the wafer. Any model that requires more synapses encounters inevitable loss.

Apparently, there are network topologies like the grid network, the Synfire chain and the Hellfire chain that are particular suitable for the system. No synapses are lost during the mapping process, even for large network instances. The former two models are both planar topologies, which means their network graphs can be embedded in a 2D plane such that no edges intersect. Planar networks can therefore be placed to the 2D wafer substrate particularly easy. However, planarity is not a necessary property for loss-free realizations, as shown by the Hellfire chain model, which is sufficiently local but not planar.

Random networks, on the other hand, cause worst-case all-to-all connectivity between chips leading to bus congestion already for small network instances. The synaptic losses for random and local networks, at both ends of the spectrum, stake out an effective reference frame wherein the synaptic loss of other topologies is embedded.

Regarding this reference frame, the layer 2/3 attractor network is particularly interesting. Note that for each network instance up to 10 692 neurons, triplets of data points exist, which correspond to hardware neuron sizes of 8, 12 and 16 circuits. These instances have all been placed using the default placement approach (see Section 6.1)

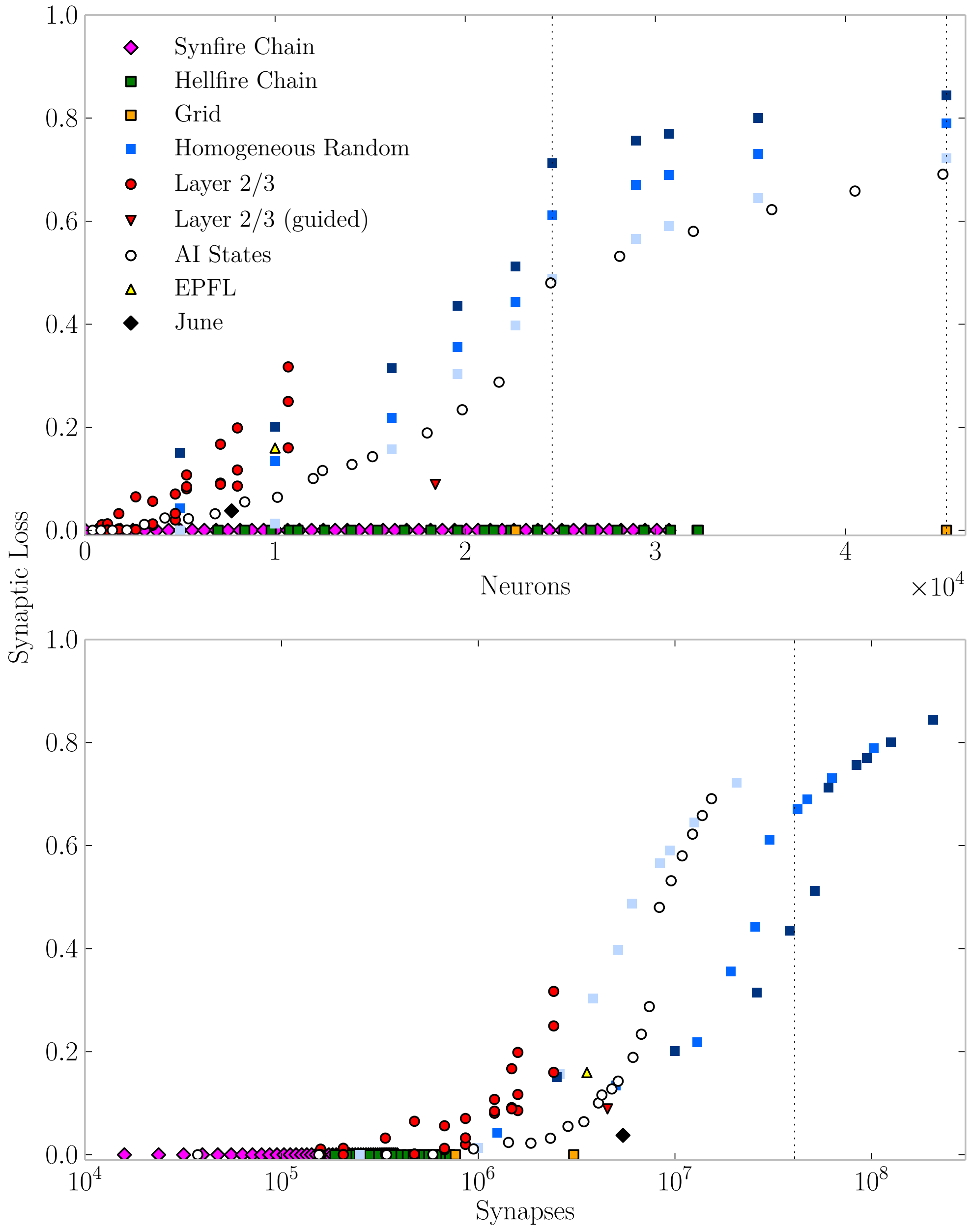


Figure 8.22: Exploration of synaptic loss for a wide range of different network topologies as a function of model neurons and synapses in the top and bottom plot, respectively. The vertical grid lines in the top plot represent the maximum number of neurons per wafer for configurations of 8 and 4 circuits per hardware neuron. The bottom grid line indicates the total number of synapses on a wafer. Models requiring more synapses than available suffer inevitable loss. Clearly, models with sufficiently local connectivity, like the Synfire and Hellfire chain as well as the grid network, are particularly suited for the HMF. Notably, the cortically inspired layer 2/3 model can be mapped with only little distortions up to large network sizes using manual guidance (red triangle). A preliminary version of this figure has been published by the author in the internal UHEI and TUD (2014) report.

leading to distortions similar to those for homogeneous random networks or even higher due to dense local connectivity within hypercolumns. Nevertheless, the prominent red triangle at around 1.9×10^4 neurons clearly shows that the model can be implemented more efficiently than homogeneous random networks due to its local structure. Manual guidance is used to place neurons such that the neighborhood relation of neurons within hypercolumns is preserved and scale the size of the different neuron types according to their relative input requirements. The resulting synaptic loss is significantly below the loss found for e.g., the second largest sibling, despite having almost twice as many neurons. Notably, this sample point lies right within the frame staked out by random networks and the local topologies.

The June network is another interesting example that outlines the importance of locality for models mapped to the HMF. It is basically a random network in the sense that any two HICANNs have to be connected, however, it can be realized more efficiently than e.g., the EFPL cortical column or homogeneous random networks. Firstly, the June model has about 20 % less neurons than the EPFL model, which is just enough to avoid the Layer 1 resource exhaustion that can be observed in Figure 8.20 for the outermost neurons. Secondly, comparing both connection structures (see Figures 8.19 and 8.20) the June network has a greater emphasis on local connectivity than the EPFL network. Thus, the penalty is less for the June network, even though long range connections are lost in both cases.

So far, mostly long range connectivity has been discussed with regard to Layer 1 constraints. For larger network instances the number of inputs per neuron is another limited resource, which depends on the hardware neuron size. As mentioned earlier, more synapses are lost for homogeneous random networks and the AI state model when the number of circuits per neuron is reduced from 8 to 4 at $N = 2.5 \times 10^4$. The gain is emphasized towards higher connection probabilities because more synapses are required for the given network size. Another example is the layer 2/3 attractor model where increasing the hardware neuron size for RSNP cells to 16 has been an important optimization to realize the high input counts.

Looking at the ratio of lost synapses over synapses in the bottom plot of Figure 8.22 for homogeneous random networks indicates that the loss is emphasized towards lower connection probabilities shown in light blue. For lower connection probabilities, a larger fraction of the synapses is lost during the wafer routing, while many hardware synapses on each chip remain unused. The latter is shown by the low synapse usage in Figure 8.8. Moreover, as the number model synapses for networks with a connection probability of $p = 5\,\%$ and $p = 10\,\%$ approaches the total number of hardware synapses in the system, the losses rapidly increase. Soon, these extra losses dominate any prior loss such that the relative synaptic loss over the number of synapses becomes almost independent of p. On close inspection of network instances with similar synapse counts, the loss is slightly less for $p = 10\,\%$ compared to $p = 5\,\%$ because more hardware synapses are used per chip for roughly the same number of model synapses. Whereas the synaptic loss in the upper neuron plot simply follows $1 - C(p)/N$ according to Equation (8.6), where N denotes the number of neurons and $C(p)$ is a constant depending on p. In Section 8.4.4, the tuning of the Layer 1 network has been studied in order to lower $C(p)$ and therefore improve the ability of the HMF to implement dense long-range connectivity. The results have shown that the bus swap as well as the crossbar layout are already well tuned in the second version of HICANN. For the current system, the network fidelity of large random networks is mainly limited by the bandwidth of horizontal buses.

Neuron Sizes

Throughout the studied networks it has been found that for larger, non-local topologies the model fidelity is typically maximized for neuron sizes of 8 circuits or more. There are mainly two reasons for the improved fidelity. Firstly, less SPL1 outputs have to be routed, thus reducing the horizontal Layer 1 congestion, secondly, the number of inputs per neuron is increased. Additionally, fewer synapse drivers have to be connected to realize the same number of synapses per Layer 1 connection, therefore increasing the effective number of inputs per chip, while reducing the capacitive load on the Layer 1 connection, see Section 1.5.5. However, using larger hardware neurons reduces the number of individually available neurons. For example, placing 118 or 59 neurons per HICANN yields a high utilization of the Layer 1 address space, but only 23 % and 12 % of the theoretically available neurons are individually used. Combining neurons is an important feature that allows to scale the input count dynamically, thus rendering the HMF suitable for a wide range of different network topologies, as shown by this study. Using more individual neurons without reducing the model fidelity is only possible if the ratio of synaptic inputs or synapse drivers per neuron is increased. The next hardware revision is further going to implement configurable conductances between neuron circuits to support compartmental dendrite models (Millner, 2012) making this feature even more valuable beyond input scaling.

Runtime Performance

Finally, an overview of the mapping runtime normalized to the number of model synapses is shown in Figure 8.23. For any network, at least 10^4 synapses and up to 10^6 synapses per second could be mapped. The determining factor is the complexity of inter-chip connectivity and therefore the runtime imposed by the wafer routing. Sparse random networks can be considered the worst case, since relatively few synapses are implemented per Layer 1 connection, while at the same time every chip requires connections to any other chip, thus saturating the network. Networks with preserved local connectivity, on the other hand, have simpler inter-chip connectivity and can therefore be mapped more efficiently. Prominent examples are the Synfire and the Hellfire chain as well as the grid network. Note that for small instances of the Synfire chain network performance regresses due to a fixed mapping overhead, which is divided by a small number of model synapses. The impact of the wafer routing on the runtime can further be observed for the triplets of red data points of the layer 2/3 scaling study, where instances of equal network size are mapped faster for smaller hardware

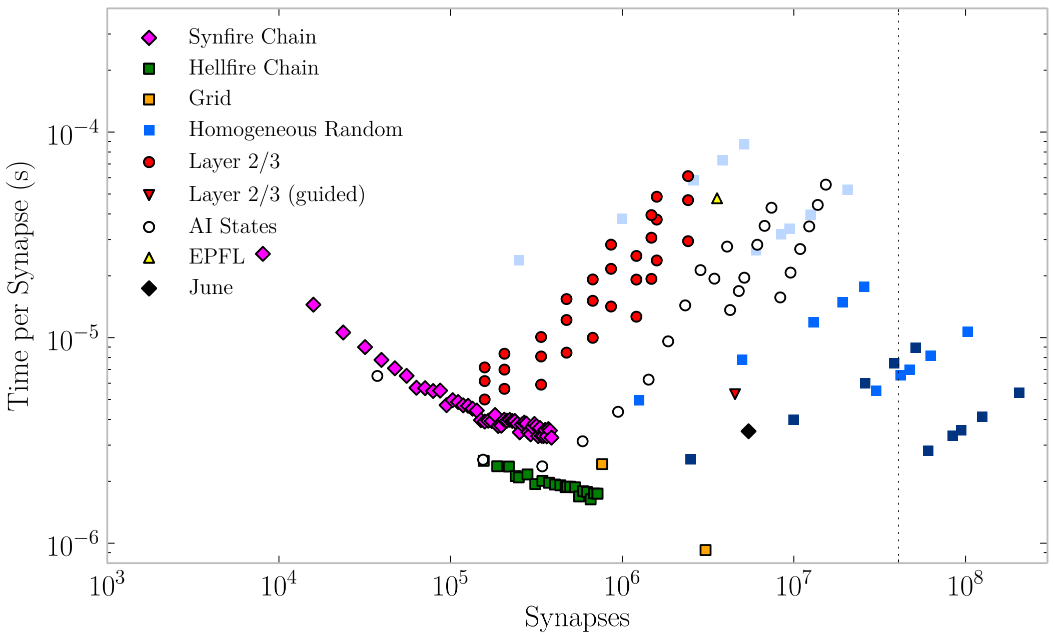


Figure 8.23: A normalization of the mapping runtime on the number of model synapses for all previous network models. The performance differences mostly depend on the necessary inter-chip connectivity, dense long-range connectivity imposes longer runtimes on the wafer routing. Furthermore, small model instances are stronger influenced by a fixed mapping runtime overhead, which can be observed for e.g., chain models requiring less time per synapse towards larger networks. Thus, large models with local inter-chip connectivity can be mapped most efficiently. The vertical grid line indicates the total number of synapses on a wafer in accordance with Figure 8.22.

neurons because of the smaller active wafer area. Similarly, for homogeneous random networks and the AI state model a speedup towards larger network sizes is observed whenever the number of circuits per neuron is reduced, simplifying the wafer routing in accordance with Figures 8.9 and 8.21. Despite them being worst-case networks, up to 2.6×10^5 synapses can be mapped per second in accordance with Section 8.4.1, which is approximately 50 times faster than the MappingTool.

9 Spike-Based Classification with Accelerated Neuromorphic Hardware

Liquid state machines (LSMs) as proposed by Maass et al. (2002) and Jaeger (2001) provide a theoretical framework for generic computation on time continuous input. They consist of a recurrent network, the so-called liquid, that projects inputs into a higher-dimensional space. A subsequent readout continuously classifies the liquid state. The idea is that only the classifier has to be trained for classification, while the liquid remains untouched and helps the readout to accomplish its task. In this chapter, a complete implementation suitable for accelerated neuromorphic hardware is presented. The liquid is based on the self-balancing network architecture by Bill et al. (2010b) and connected to a tempotron-like binary classifier (Gütig and Sompolinsky, 2006). Both parts of the LSM are realized the neuromorphic Spikey chip, allowing to efficiently discriminate inputs at high rates based on the bandwidth friendly, sparse tempotron response.

The setup has first been introduced by Jeltsch (2010) and the integrated setup, realizing liquid and readout together, has been published by the author in Pfeil et al. (2013). Limitations of the setup have been further explored by Probst (2011) and Holford (2011). Specifically, the results for the demanding task in Section 9.4.2 and the digit recognition in Section 9.4.3 have been produced in cooperation with Dimitri Probst. The traceless learning introduced in Section 9.3.3 has been studied by Nathan Holford under the supervision of the author.

9.1 Hardware Platform

The following experiments have been conducted on Spikey version 4, a neuromorphic mixed-signal chip that has been developed prior to HICANN. The chips have a lot in common, for example, both operate about 10^4 times faster than biological real time, implement analog neurons with conductance-based synapses and 4 bit digital weights. Most notably, Spikey provides 192 simpler leaky-integrate and fire (LIF) neurons, compared to the AdEx neurons on HICANN. Further differences are explained in the text wherever appropriate. Figure 9.1 shows the new USB Spikey system, however, most experiments have been conducted on an ethernet-based predecessor platform due to unreliable host communication of the new system. First experiments on the new USB system are promising, indicating a 17 times speedup for the LSM over the predecessor, which reduces the time per learning iteration to approximately 0.2 s.

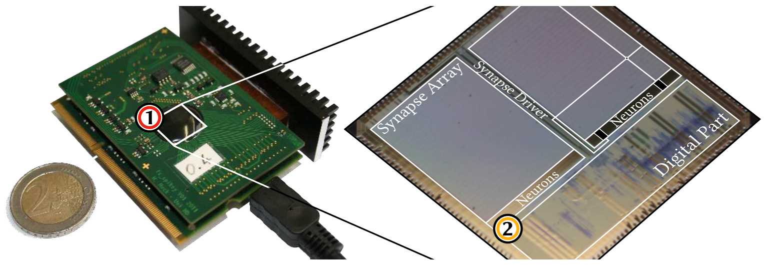


Figure 9.1: A photo of the USB Spikey system **(1)**. The PCB is approximately the size of a credit card. Spikey **(2)** itself is hidden underneath the black cap. The neuron connectivity is similar to the one found on HICANN, including row-wise synapse drivers and 4 bit digital weights.

Schemmel et al. (2006) and Pfeil et al. (2013) provide further details on the hardware platform.

9.2 Liquid State Machine

LSMs as proposed by Maass et al. (2002) and Jaeger (2001) consist of a liquid substrate, which projects input into a higher dimensional space, and a subsequent readout that continuously classifies the liquid state. This kind of input preprocessing is similar to the kernel tick in machine learning (Aizerman et al., 1964) and conceptually simplifies the task of the readout.

Any recurrent network that maps different inputs to different responses meets the sufficient separation property and can therefore be used as a liquid. Here, a network topology similar to the one described by Bill et al. (2010a) has been used. Bootstrapping this network on hardware is simplified by its self-stabilizing property, which causes rich dynamics over a wide range of inputs. The network consists of two populations, an excitatory and an inhibitory one, with a neuron ratio of 80:20. The network size is scaled to utilize all remaining neurons after the readouts have been placed, which typically results in 147 to 191 neurons. The connection probabilities of recurrent and feedforward connections are illustrated in Figure 9.4. Each neuron in the liquid receives 4 inputs from a pool of 32 excitatory and 32 inhibitory sources. Note that synapses within the liquid have fixed efficacies. Only the feedforward connections from the liquid to the readout are trained upon learning. A full list of analog neuron parameters is given in Appendix A.2.

9.2.1 Tempotron

A tempotron (Gütig and Sompolinsky, 2006) single neuron binary classifier is used as LSM readout, thus leaving most of the available hardware neurons to the liquid. The original weight update rule implements learning with gradient descent dynamics for LIF neurons with current-based synapses but without spike triggered voltage reset. Upon training, the tempotron learns to distinguish between two input classes and responds by either emitting a single or no spike within a given classification time window. Whenever the tempotron spikes, all subsequent inputs are shunted artificially to ensure that the tempotron elicits no further spikes.

This section firstly introduces the original learning rule. Subsequently, a model adaptation is presented that copes with voltage resets, conductance-based synapses and hardware limitations.

The PSP kernel K of a LIF neuron with current-based exponential synapses is given by

$$K(t-t_k) = V_0 \left[\exp\left(-\frac{t-t_k}{\tau_\mathrm{m}}\right) - \exp\left(-\frac{t-t_k}{\tau_\mathrm{s}}\right)\right] \cdot \Theta(t-t_k) \qquad . \tag{9.1}$$

Where τ_m and τ_s are the membrane and synaptic time constants, V_0 is a constant PSP scaling factor, t_k is the time of the kth afferent spike and $\Theta(t)$ the Heaviside step function. The membrane voltage $V_\mathrm{m}(t)$ of a neuron without spiking can then be written as

$$V_\mathrm{m}(t) = \sum_j \omega_j \sum_k K(t-t_k) + E_\mathrm{l} \qquad , \tag{9.2}$$

with the leakage potential E_l and synaptic weights ω_j.

A gradient descent update rule guides the learning of afferent weights for erroneous classifications. It can be derived from the following cost function

$$E_\pm = \pm(V_\mathrm{thresh} - V_\mathrm{m}(t_\mathrm{max})) \cdot \Theta\left(\pm(V_\mathrm{thresh} - V_\mathrm{m}(t_\mathrm{max}))\right) \qquad . \tag{9.3}$$

Here, $E_\pm$ denotes the involved cost and t_max is the time that maximizes $V_\mathrm{m}(t_\mathrm{max})$ within the classification window. The Heaviside step function ensures that only erroneous classifications contribute a cost. The $(+)$ case applies for erroneous spikes and the $(-)$ case for missing spikes.

The cost function measures to what extent the correct response has been missed as the distance between the membrane voltage $V_\mathrm{m}(t_\mathrm{max})$ and the threshold potential V_thresh. The negative derivative of the cost function with respect to the afferent weights $(-\,{}^{\mathrm{d}\,E_\pm}/_{\mathrm{d}\omega_\mathrm{j}})$ leads to the gradient descent update rule

$$\Delta w_j^n = \begin{cases} 0 & \text{correct} \\ \mp\alpha(n) \sum_{t_{j,k}<t_\mathrm{max}} K(t_\mathrm{max} - t_{j,k}) & \text{erroneous} \end{cases} \qquad . \tag{9.4}$$

Where $\Delta\omega_j^n$ denotes the weight update of the jth afferent neuron after n learning iterations. The update depends on the PSP amplitude caused by a spikes at $t_{j,k}$, which is given by $K(t_\mathrm{max} - t_{j,k})$ at time $t_\mathrm{max}t$. $\alpha(n)$ is an optional learning rate, which may decrease over time to modulate the amplitude and ensure that the weights eventually converge. All experiments in this chapter use a learning rate of the form $\alpha(n) = \alpha_0 \exp\left({}^{n}/_{\tau_\alpha}\right)$.

In other words, the excitatory afferents that contributed causally to an erroneous spike are weakened, while inhibitory ones are strengthened by Δw_j^n in (+) trials. In (−) trials, the weights are updated in the opposite manner.

Typically, all synapses are initialized weak. Upon learning, a few synapses are strengthened quickly until a correct input-output mapping has been established such that no more weight updates are necessary according to Equation (9.4). This means that the other synapses remain weak, which leads to a bimodal-like weight distribution.

9.3 Hardware Adaptation

Implementing the readout on hardware requires a few considerations in order to achieve decent classification performance, as Spikey is a close but not perfect match for the original tempotron.

The original tempotron weight update rule implements analytically precise gradient descent dynamics. However, some of the constraints necessary to derive the rule in the first place cannot be met by Spikey and therefore have to be compensated for. Consequently, the weight update rule is merely an approximated gradient descent. Firstly, shunting inputs after the first tempotron spike is neither possible on Spikey nor biological. This constraint has simply been dropped as already proposed by Gütig and Sompolinsky (2006). Consequently, the tempotron signals its decision by either emitting none or any number of spikes. Secondly, the spike-triggered membrane voltage reset is imposed by the hardware implementation. This means, in the (+) case, t_max is approximated by the spike time rather than the time where the voltage of a corresponding free membrane had been maximized.

9.3.1 Conductance-Based Synapses

The most significant model deviation imposed by hardware are the conductance-based synapses. Early experiments have shown poor chance-level classification performance without any compensation. The original update rule requires that changes to the afferent efficacies ω_j affect the PSP strength proportionally. This requirement holds for current-based synapses according to Equation (9.2). However, for conductance-based synapses the current contribution $I_{j,k}$ per spike k via synapse j depends not only on the synaptic weight ω_j, but also on the momentary membrane voltage $V_\mathrm{m}(t)$, according to

$$C_\mathrm{m}\frac{\mathrm{d}\,V_\mathrm{m}}{\mathrm{dt}} = -g_\mathrm{l}(V_\mathrm{m} - E_\mathrm{l}) + \omega_j I_{j,k} \tag{9.5}$$

with

$$I_{j,k}(t) = g_j\left(V_\mathrm{m}(t) - E_j\right)\exp\left(-\frac{t - t_k}{\tau_\mathrm{s}}\right)\Theta(t - t_k) \qquad . \tag{9.6}$$

Where, E_l and g_l denote the leakage potential and conductance, C_m the membrane capacitance, ω_j the synaptic weight, g_j the synaptic base conductance as well as τ_m and τ_s the membrane and synaptic time constants, respectively. On Spikey, two reversal potentials E_e and E_i for excitatory and inhibitory synapses exist that are shared by all neurons.

In a first order approximation, the voltage difference $(V_\mathrm{m}(t) - E_j)$ in Equation (9.6) is considered constant over the course of a PSP. For weak synapses and short time constants or a neuron in the high-conductance state (Destexhe et al., 2003b) this is a reasonable approximation. For a single spike and a resting neuron, the PSP kernel K' for synapse j can be written as

$$K'(t - t_k) \sim (V_\mathrm{m}(t_k) - E_j) \cdot K(t - t_k) \qquad , \tag{9.7}$$

with the PSP kernel $K(t)$ for a LIF neuron with current-based synapses according to Equation (9.1). In order for this to work, the hardware time constants τ_m and τ_s used by $K(t)$ have to be calibrated, which has been done by Brüderle (2009).

Furthermore, the dynamic range of V_m, which is typically bound by V_rest) and V_thresh except for strong inhibition, can be set such that $(V_\mathrm{thresh} - V_\mathrm{rest})$ is small compared to $(V_\mathrm{m}(t) - E_j)$ for both reversal potentials $E_j \in \{E_\mathrm{e}, E_\mathrm{i}\}$. The strength of a PSP can then assumed to be independent of $V_\mathrm{m}(t) = \overline{V_\mathrm{m}}$, which means that the synapses effectively behave like current-based synapses. The PSP strength depends on the constant scaling factor $(\overline{V_\mathrm{m}} - E_j)$ that is different for both synapse types.

Consequently, as long as the PSPs are small and the dynamic range of V_m is far from both reversal potentials, the original tempotron weight update rule can be used. Notably, the iterative training adapts the tempotron to the inputs as well as the underlying substrate, hence inherently compensating for fixed-pattern variations of the system. However, the updates have to be scaled for different synapses types to compensate for asymmetry according to

$$\Delta\omega_\mathrm{exc} = \frac{\overline{V_\mathrm{m}} - E_\mathrm{i}}{\overline{V_\mathrm{m}} - E_\mathrm{e}} \cdot \Delta\omega_\mathrm{inh} \qquad . \tag{9.8}$$

A scaling is unnecessary if E_e and E_i are set symmetrically around $\overline{V_\mathrm{m}}$, but this is outside the accessible hardware ranges. For hardware the scaling factor is always smaller than one. The analog neuron parameters used in the following experiments are listed in Appendix A.2.

The original tempotron allows synapses to change arbitrarily from excitatory to inhibitory and vice versa. For experiments realizing liquid and readout on a single chip the synapse type is predetermined by the liquid topology because synapses for both, liquid and readout are operated by the same synapse driver, which determines their type. Changing synapse types upon learning would also change the synapse types for neurons in the liquid and thus violate the concept of a static liquid for LSMs. Hence, whenever synapse j develops a negative weight ω_j, it is clipped to zero. For tempotron only experiments, where e.g., recorded liquid responses are played back, the synapse type could in principle change. However, the remaining discontinuity for efficacies due to different reversal potentials and limited weight resolution complicate

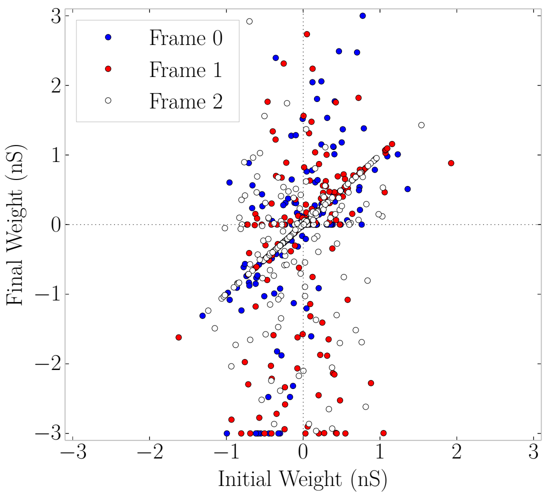


Figure 9.2: Weight development for the last three classification windows of the memory task, see Section 9.4.1. The normally distributed initial weights are plotted against the final weights after a software-based training. Synapse types could change freely from inhibitory to excitatory and vice versa. Weights within the top left and bottom right quadrant did change their type and would otherwise be locked to zero. These additional synapses increase the classification performance for *Frame 1* compared to fixed synapse types in Figure 9.6. Figure taken from Probst (2011).

learning. Therefore, changing synapse types is not allowed in the following experiments. This restriction limits the dynamic range of the tempotron because synapses that would normally change from excitatory to inhibitory or vice versa are locked to zero and cannot contribute to the classification as shown in Figure 9.2. The following two strategies for training have been explored by Probst (2011) to study this effect.

Preserved Synapse Type (PST)

This strategy simply preserves synapse types, thus keeping them either excitatory or inhibitory during the training. Their type is determined by the liquid topology or randomly at start. The synaptic weights are initially set to 0.1 nS. Iteratively, the weights evolve according to Equations (9.4) and (9.8). Whenever a synaptic weight is turned negative to indicate a type change, the weight is clipped to zero.

Prior Evaluation in Software (PEiS)

Using the approximation from Equation (9.7), conductance-based synapses behave similarly under certain conditions. Thus, synapses of an ideal tempotron that actively contribute to a classification are likely to do so for a hardware readout with conductance-based synapses. Specifically, their preferred type is expected to be the same. Hence, a preparatory software study with current-based synapses can determine the preferred type for each synapse and initialize them afterwards on hardware accordingly. This strategy can help to maximize the classification capabilities of the tempotron. By comparing both strategies, the influence of fixed types on the classification performance can be evaluated. However, this strategy can only employed for experiments where the synapse types can be chosen freely, e.g., a sole tempotron that classifies recorded liquid responses. In cases where the synapse type is predetermined by the liquid this strategy cannot be used.

After the initial software evaluation the weights are further trained on hardware following the previous strategy to optimize the classification for conductance-based synapses as well as compensate for fixed-pattern variations on hardware.

9.3.2 Limited Weight Resolution

Synaptic efficacies on Spikey have both, a limited parameter range and a 4 bit resolution. Firstly, the limited efficacies can be compensated by scaling the distance of E_l and V_thresh such that for a given input a few strong synapses suffice to elicit a spike. Secondly, the weight discretization is tolerable, as tempotron readouts themselves typically establish a discrete, bimodal weight distribution upon training. However, the learning rate $\alpha(n)$ may impede further training if it becomes smaller than the weight resolution provided by a single LSB.

The tempotron implementation in software limits all weights to a maximum of 3 nS in order to keep them within hardware mappable range and, hence, simplify the transition to hardware for the learning strategy with prior synapse type evaluation.

9.3.3 Traceless Learning

Learning requires access to the membrane recordings of the tempotron to derive t_max in the $(-)$ cases, which slows down execution because significantly more data has to be transfered. This thwarts the benefits of accelerated systems on the one hand and limits the number of readouts that can be trained in parallel on the other, since only four traces can be accessed at a time. Here, a spike-based approach is presented that can be used to train a tempotron for suitable tasks without membrane recordings.

As mentioned earlier, t_max is set to the spike time in the $(+)$ case rather than the time of maximum membrane voltage of a corresponding neuron with free membrane dynamics. Experiments have shown that the tempotron is even more robust regarding imprecise measures of t_max. For example, t_max can be chosen from a normal distribution centered around the classification window at random in the $(-)$ case, which means no voltage course is required anymore and the learning is purely spike-based. The consequent learning progress is shown in Figure 9.3 over 2000 training iterations for the memory task presented later in Section 9.4.1. Interestingly, the tempotron still achieves close to perfect classification performance. Even though the readout is no longer learning correctly in the $(-)$ case, it can be sufficient to learn in the $(+)$ case only. Therefore, some kind of normalization has to prevent excitatory efficacies from converging to zero and limit inhibitory ones because the $(+)$-update only weakens excitatory weights and strengthens inhibitory ones. The $(-)$-update can be employed as such normalization, by arbitrarily strengthening and weakening excitatory and inhibitory weights, respectively.

The argument is supported by the right plot in Figure 9.3, which shows the classification performance after 600 training iterations for different fixed values of t_max. The normalization works more efficiently towards the end of the classification window because more afferents can contribute to the membrane state until t_{max} and are thus subject to learning. This also means that the performance of the spike-based learning approach depends on the spatio-temporal structure of the input and may not work for arbitrary tasks. Nevertheless, this approach allows to train multiple hardware tempotron readouts efficiently in parallel.

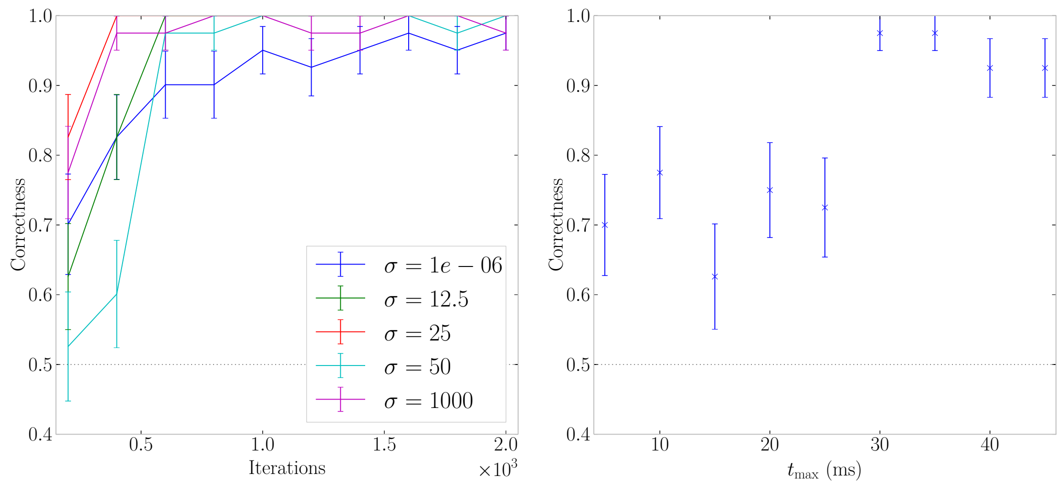


Figure 9.3: Traceless learning results for random t_{max} in the $(-)$ case. In the left-hand plot, the learning is studied for normally distributed t_{max} centered around $\mu = 25\,\mathrm{ms}$ for different σ in the $(-)$ case. All classifiers reach a correctness beyond 90 %, though performance regresses for $\sigma = 10^{-6}$. The plot on the right shows the final classification performance after 600 training iterations for fixed values of t_{max} in the $(-)$ case. The tempotron performs significantly better for values of t_{max} in the second half of the window ($t_{\mathrm{max}} > 25\,\mathrm{ms}$) because more synapses actively contribute to the membrane state until t_{max} and are therefore subject to normalization. Figures based on data from Holford (2011).

9.3.4 Short Term Plasticity

The liquid network uses STP (Brüderle, 2009) to stabilize its dynamics and meet the separation property for a wider range of inputs. However, STP on Spikey is configured at the synapse driver for a complete row of synapses, including both, recurrent as well as feedforward connections to the readout. Therefore, either synapses for both, liquid and tempotron, are modulated by STP or none. Preliminary hardware experiments have shown that strong STP severely affects the classification performance of the tempotron because the learning rule cannot account for short-term changes in synaptic efficacies.

Consequently, STP is either turned off completely or kept weak, even though disabling plasticity makes the network more sensitive to varying spike input. This means, for strong recurrent excitation the network may no longer suffice the necessary separation property, rendering the network an unsuitable liquid substrate. In experiments without STP, V_{thresh} has therefore been tuned for the specific task and hardware instance.

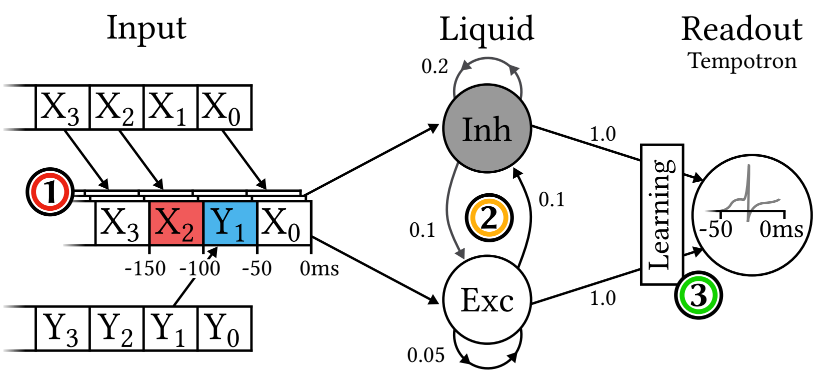


Figure 9.4: Schematic of the LSM and the memory task, see Section 9.4.1. Input spike trains are composed from 50 ms segments that are randomly picked from either of the two template spike trains X and Y **(1)**. These spike trains are subsequently streamed into the liquid **(2)** using 64 inputs. Each neuron receives spikes from four of the inputs. For the memory task, the network consists of 191 neurons, leaving one neuron to the tempotron. Connectivity is illustrated as connection probabilities next to the arrows. The tempotron **(3)** is trained iteratively to distinguish segments from X and Y by looking only at the last 50 ms of the liquid response. A similar figure has been published by the author in Pfeil et al. (2013).

9.4 Applications

The LSM presented above has been applied to three different tasks. The first *memory task* setup has been introduced by Jeltsch (2010). The second *demanding task* is designed to challenge the tempotron and therefore support the general usefulness of a liquid for pre-processing the input. In the final task, handwritten digits from the MNIST database (LeCun and Cortes, 1998) have to be recognized. The latter two tasks have first been presented by Probst (2011).

9.4.1 Memory Task

The binary memory task has originally been proposed by Maass et al. (2002) for a different LSM setup. A history of spike train segments has to be reconstructed from a continuous input stream. The input is generated by cutting two template spike trains, denoted as X and Y, into small segments for time windows of 50 ms each. Then, input spike sequences are composed by randomly picking a segment from either X or Y for every time window. Afterwards, normally distributed jitter with a standard deviation of $\sigma = 1$ ms is added to every spike time in order to make the task more challenging. Figure 9.4 illustrates the protocol.

Then, the spike trains are iteratively fed into the liquid and subsequently read out by the classifier. In individual experiments, a tempotron is trained to distinguish the templates for one of the 50 ms time windows by always looking at the last 50 ms of the liquid response only. Therefore, the readout has to base its decision on the echo stored in the liquid. Earlier time windows are more challenging due to the fading memory of

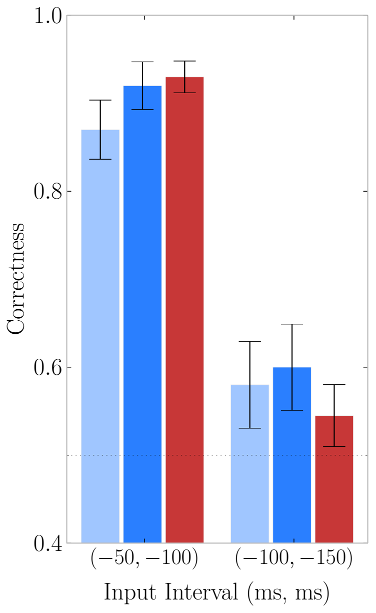
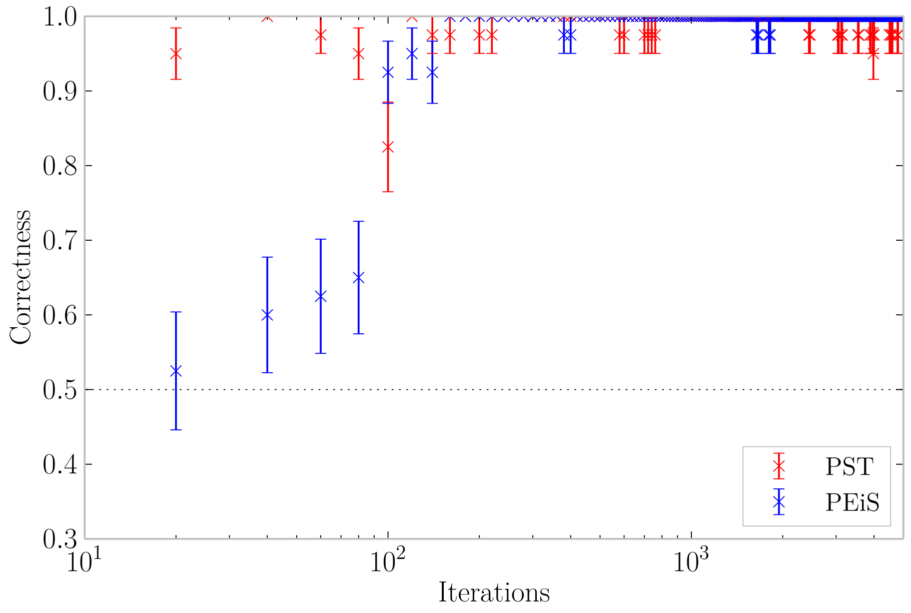


Figure 9.5: Accuracy of the LSM solving the memory task for the different classification windows *Frame 1* and *Frame 2*. The setups have been trained over 1000 training iterations and their final responses averaged over 200 samples. The software results are shown in red, while results for ethernet and USB Spikey are shown in light and dark blue, respectively. Hardware and software perform similar, indicating the suitability of the substrate for the LSM setup. A similar figure has been published in Pfeil et al. (2013).

the liquid. A readout that classifies a segment n, which is n windows in the past, is denoted as *Frame* n.

This experiment has been carried out with software and hardware tempotron implementations. In the former case, the response of the hardware liquid is recorded but classified in software. The hardware implementation, on the other hand, realizes both, liquid and readout together on the Spikey chip. The firing threshold of the liquid has been tuned independently to optimize memory capacity for software and hardware implementations independently. Note that tuning V_{thresh} for the hardware tempotron also changes the liquid behavior because the parameter is shared by every other neuron. Finally, individual learning curves $\alpha(n)$ have been applied to optimize performance on the one hand and account for the limited weight resolution on hardware on the other.

The results are presented in Figure 9.5. Both, the hardware and the software LSM correctly classify about 90 % of the spike train segments between 50 ms to 100 ms in the past. For earlier inputs, all implementations drop to chance level, which is 50 % for binary tasks. Furthermore, the two different learning strategies have been explored. Figure 9.6 presents the learning curves for hardware in the time windows from 0 ms to 50 ms and 50 ms to 100 ms as well as both training strategies (see Section 9.3.1). Most notably, the strategy with prior software evaluation shows a steeper learning progress. There are more weights with preferred synapse type that can be tuned towards the correct response. Though, for the final outcome the strategy makes little difference because the tempotron typically bases its decisions on a few strong synapses, while the others remain weak. This means, as long as the readout is trained long enough and there are sufficient weights of suitable type to trigger the correct response for both input classes the strategy has little impact on the final performance.



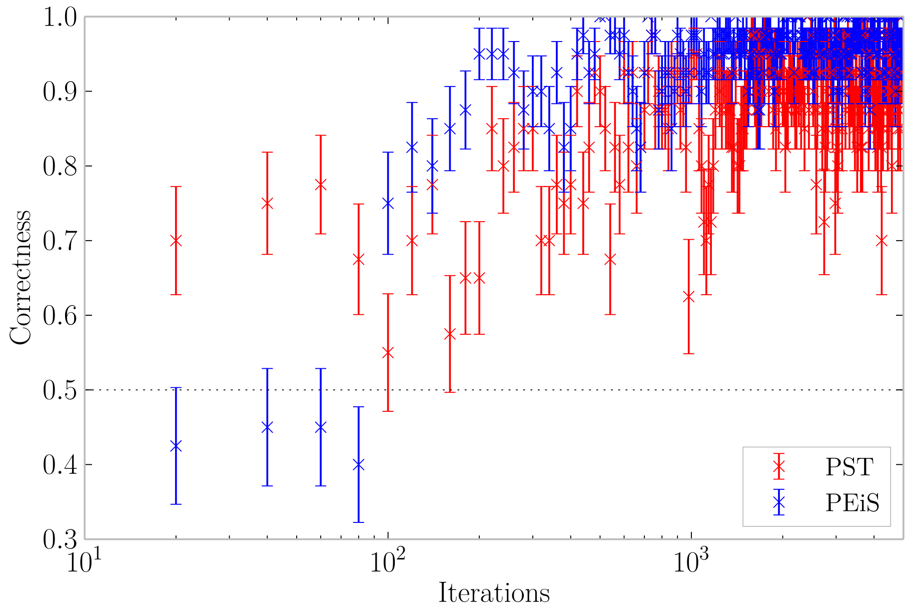


Figure 9.6: Progress over 5000 learning iterations for the *Frame 0* (top) and *Frame 1* (bottom) hardware tempotron. The performances for the PST and PEiS training strategies are shown in red and blue, respectively. Interestingly, for *Frame 0* the tempotron learns faster with the simple PST strategy. For the PEiS strategy, on the other hand, unsuitably strong weights from the prior software training need to be weakened. Ultimately, both reach almost perfect accuracy. However, classifying *Frame 1* is more challenging, rendering the PEiS strategy slightly more effective.

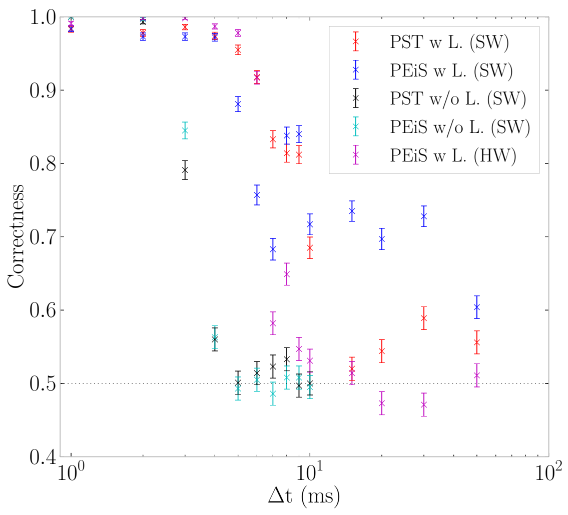


Figure 9.7: Classification performance for the demanding task described in Section 9.4.2. The different training strategies and substrates have been explored as function of Δt. For small Δt the performance is almost perfect for all setups. Performance drops to chance level when Δt becomes larger. Clearly, the liquid extends the accessible range. The limit depends on both, the liquid implementation as well as the readout training strategy. The software liquid with prior weight evaluation performs best and classifies beyond chance level event up to $\Delta t = 30\,\text{ms}$. Figure based on data from Probst (2011).

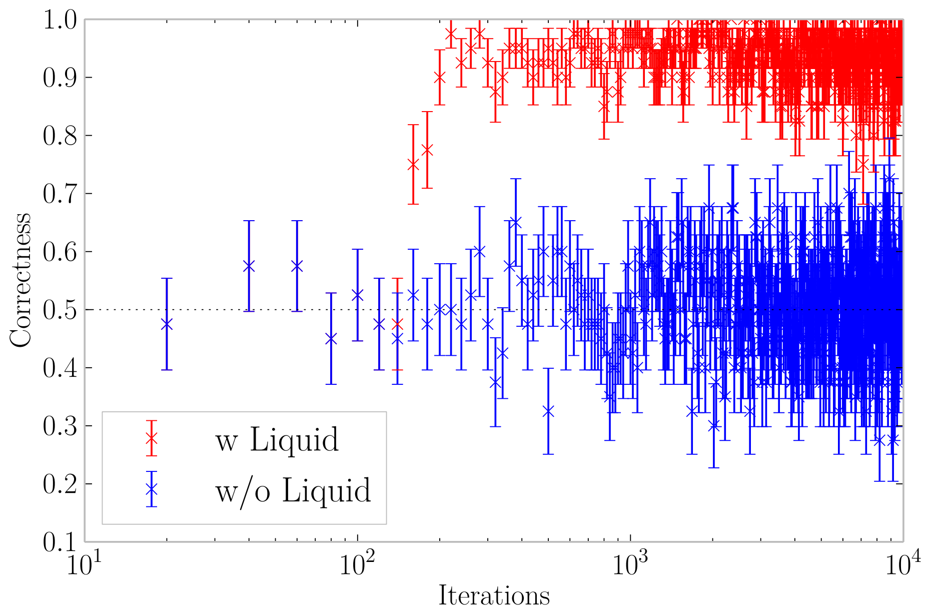
9.4.2 Demanding Task

The tempotron by itself is a powerful, nonlinear classifier. A special task has been designed to specifically challenge the tempotron and demonstrate that a liquid can improve readout performance.

For this task, two similar 100 ms spike patterns A and B have to be discriminated. Both patterns consist of 12 excitatory and 12 inhibitory inputs. All inputs fire once at $t_\text{A} = 10\,\text{ms}$ for pattern A and $t_\text{B} = 100\,\text{ms} - \Delta t$ for pattern B. Where, Δt is a free parameter defined by the task instance. Thus, pattern A and B are spatially identical, but shifted relative to one another in the classification window. Additionally, a narrow, normal distributed jitter of $\sigma = 0.7\,\text{ms}$ is applied. The liquid has been tuned to the relatively low input rate by lowering V_thresh to $-56\,\text{mV}$.

Then, the tempotron is trained to fire for input A and remain silent for input B. The classifier cannot simply develop a few strong weights to elicit the correct behavior because the spatial structure of both input patterns is identical. In fact, the only option is to delay the response for a sufficiently long time, such that a spike for input B misses the classification window. A liquid can provide additional delay and therefore enable the LSM to work for larger Δt than the sole tempotron.

The results for different setups and values of Δt are shown in Figure 9.7. Note that only the tempotron is implemented in software. The actual classification is conducted on hardware spike recordings of the liquid response. Until about $\Delta t = 3\,\text{ms}$ the task is simple enough for all setups to delay the spike for pattern B. For larger Δt, the liquid becomes necessary to achieve good classification performance. The hardware LSM drops to chance level at around $\Delta t = 6\,\text{ms}$, closely followed by the PST software LSM at $\Delta t = 10\,\text{ms}$. Interestingly, the software implementation with optimized synapse types classifies beyond chance level until $\Delta t = 30\,\text{ms}$. Figure 9.8 presents the learning curve and final weight distribution for tempotron readouts with and without liquid



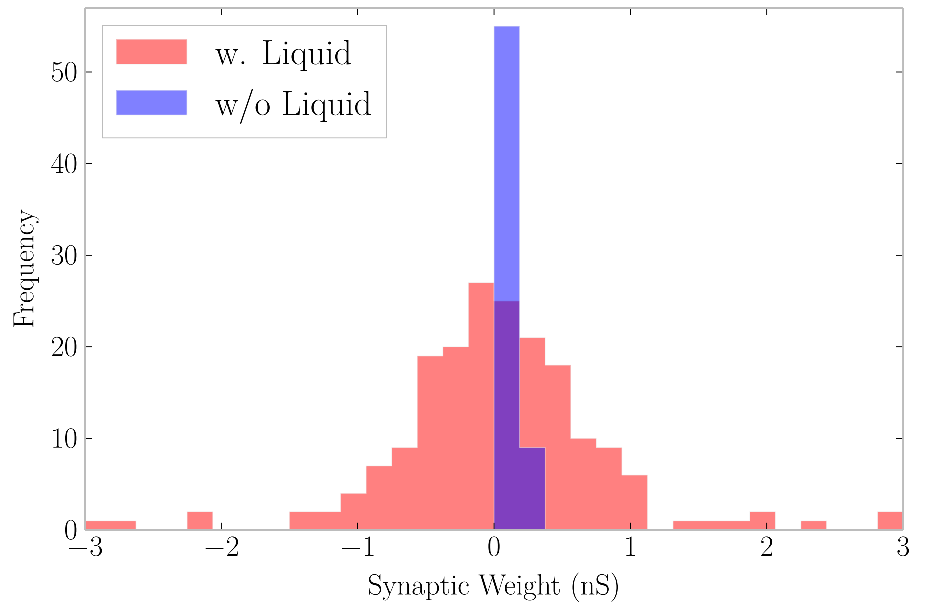


Figure 9.8: On top, development of classification performance for the demanding task with $\Delta t = 5\,\mathrm{ms}$. The PEiS learning strategy with preceding software evaluation has been used in order to increase the effective number of synapses that contribute to the decision. Clearly, the LSM outperforms the sole tempotron. Below, the corresponding weight distribution is shown after the complete 10^4 training iterations. The tempotron without liquid cannot distinguish between the patterns, which causes weak weights around zero and leads to chance-level performance. However with liquid the input is sufficiently delayed, such that a few strong weights can evolve to prime classification. Figures based on data from Probst (2011).

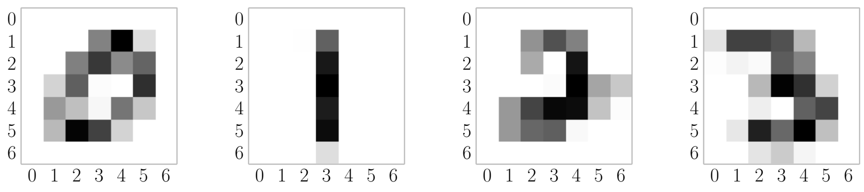


Figure 9.9: Handwriting examples from the MNIST database(LeCun and Cortes, 1998) for the digits from 0 to 3. The images have been down sampled from 28×28 pixels to 7×7 pixels. Each of the 49 pixels is mapped to a Poisson input with its pixel brightness linearly mapped to the input rate. Inputs are excitatory and inhibitory in alternating order. Note, the shape of the digit *1* is clearly distinct from all others, simplifying its pairwise separation.

pre-processing at $\Delta t = 5\,\text{ms}$. Clearly, the liquid improves the performance of the isolated tempotron.

Notably, the learned weight distribution is rather unusual for a tempotron. Typically, tempotron readouts evolve a bimodal weight distribution with some strong synapses, as explained earlier. However, for this task it is beneficial to evolve a broad distribution of rather weak synapses to delay the spike response as long as possible beyond the classification window for pattern B.

9.4.3 Handwritten Digit Recognition

The last task uses a hardware LSM to recognize handwritten digits from the MNIST database (LeCun and Cortes, 1998). Both the liquid and classifiers are realized on-chip. This task does not specifically challenge any property of the LSM, however, demonstrates its universality. Moreover, the extensibility of multiple binary tempotron readouts towards multi-class decisions is tested.

The multi digit task has been mapped to a set of binary classifications by conducting pairwise separation between all combinations of the n digits, therefore $\frac{n(n-1)}{2}$ readouts are required to cover all possibilities. Thus, implementing readouts for the pairwise discrimination of 10 digits results in 45 tempotron neurons. The liquid populations are scaled accordingly to occupy the remaining 147 neurons on Spikey (see Table 9.1) leaving synapse drivers for 64 external spike inputs. Early experiments for 10 digits showed that the setup cannot reliably classify the digits. The task has therefore been simplified to only discriminate between the digits from 0 to 3.

Inputs are constructed from the images by assigning pixels to excitatory and inhibitory inputs in alternating oder. However, the MNIST images measure 28×28 pixels, which exceeds the available 64 inputs. Hence, the images are down-sampled by collapsing four pixels into one, resulting in 7×7 images. The pixel brightness is then translated linearly into Poisson spike trains with up to $80\,\text{Hz}$ and a duration of $100\,\text{ms}$. Example digits are illustrated in Figure 9.9. The overall input rate is given by the average image brightness and varies depending on the actual digit, e.g., threes typically cover more

N_e	107
N_i	40
$p_\mathrm{ee}/p_\mathrm{ei}/p_\mathrm{ie}/p_\mathrm{ii}$	0.05/0.1/0.1/0.2
$g_\mathrm{ee}/g_\mathrm{ei}/g_\mathrm{ie}/g_\mathrm{ii}$ (nS)	2/3/1.5/2
STP	weak

Table 9.1: Parameters of the self-stabilizing LSM used for the handwritten digit recognition task. With the number of excitatory and inhibitory neurons N_e and N_i, the connection probabilities p_xy and synaptic weights g_xy between populations x and y. Weak STP is enabled to make the liquid more robust against the varying input for different digits.

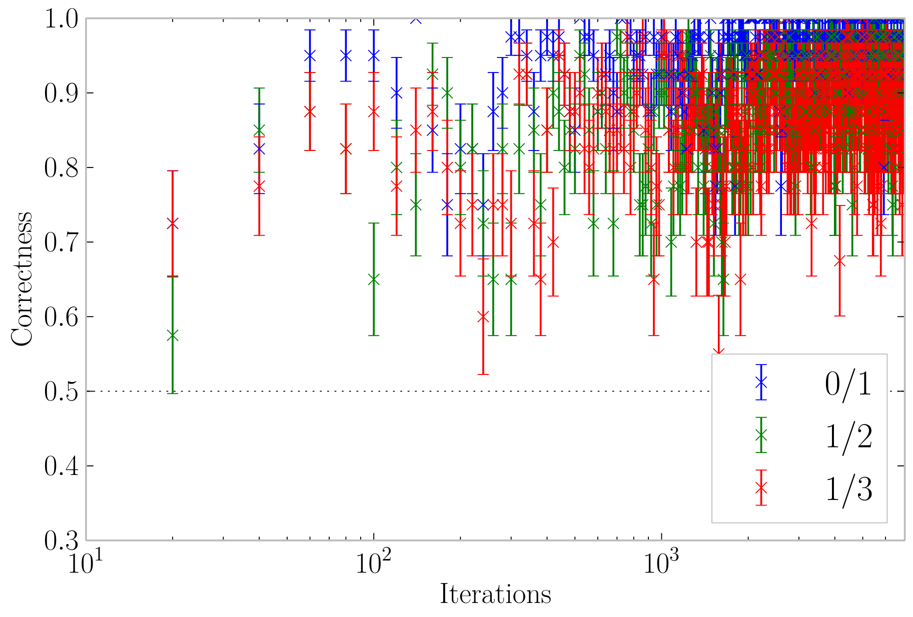
pixels than ones and therefore translate into higher input rates. Consequently, weak STP is enabled to stabilize the liquid against these input variations.

The results of the pairwise discriminations for the software and hardware readout are illustrated in Table 9.2 and Figure 9.10. Again, only the readout is implemented in software, the classification is carried out based on spike recordings of the hardware liquid.

For most digits, the software outperforms the hardware realization. On possible reason is the lack of STP for the software tempotron, which renders the weight update rule more effective as explained in Section 9.3.4. Furthermore, some digits are easier to distinguish than others, e.g., *1* has a distinct shape simplifying its pairwise discrimination against other digits.

Classifying the actual input digits requires to combine answers from the quorum of pairwise classifiers into a coherent answer. For an input digit x the y/z classifier still contributes to the overall decision. Moreover, the similarities of some digits lead to correlated decisions. For example, a classifier that separates between 1 and 8, but sees a 3 is more likely to answer 8 than 1. However a simple majority vote is used as first order approximation. The digit that receives the most votes is considered the original input. If two or more digits are elected with an equal number of votes, the input is chosen randomly from the rivals. In a first trial over 4000 samples, the hardware LSM achieved a correctness of $(50.3 \pm 0.8)\,\%$ for the four digit task, which is clearly above the chance level of 25 %. Currently, the performance is bound by the poor discrimination accuracies for classifications between 0/2, 0/3 and 2/3. Presumably, STP, low resolution input images and the small liquid size mainly limit the low performance for this more challenging digits.

The results show that the LSM setup can yet be used to discriminate real world inputs, like handwritten digits, but at the same time point out current limitations. Future experiments are expected to further improve upon multi-class decision capabilities, however, other approaches beyond sets of independent binary classifiers exist, like neural sampling (Buesing et al., 2011; Petrovici et al., 2013).



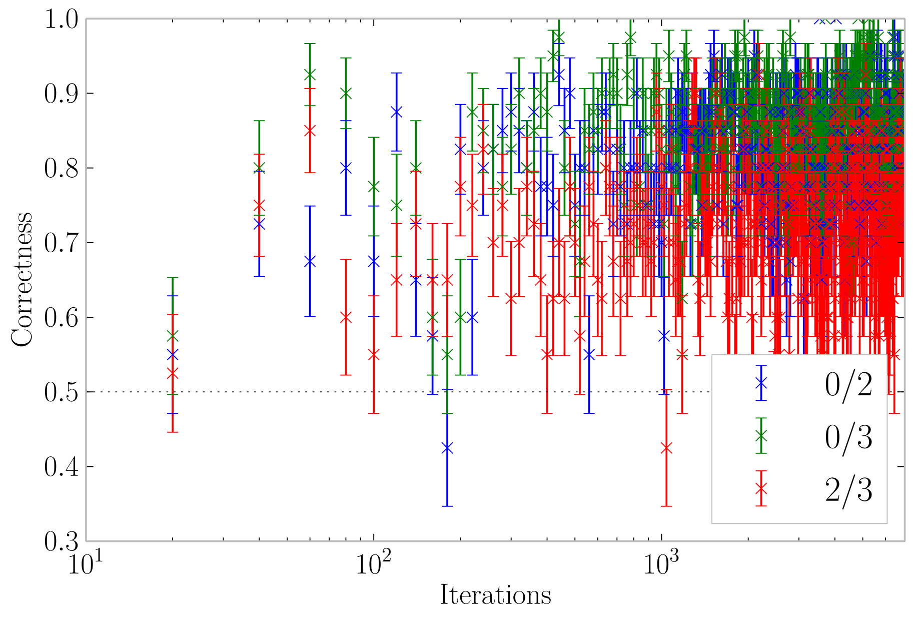


Figure 9.10: Learning progress over 7000 training iterations for the software tempotron using the simple training strategy. The easier pairwise discriminations 0/1, 1/3 and 1/3 are shown in the upper plot and the more challenging ones 0/2, 0/3 and 2/3 in the bottom. In both cases the time until the learning converges is similar. However, a higher final accuracy is achieved for the simpler discriminations. Figures based on data from Probst (2011).

Digit	0/1	0/2	0/3	1/2	1/3	2/3
SW (%)	96.2 ± 0.6	81.5 ± 1.2	87.2 ± 1.1	90.1 ± 1.0	88.7 ± 1.0	75.6 ± 1.4
HW (%)	94.0 ± 0.8	68.0 ± 0.5	85.8 ± 1.1	83.3 ± 1.2	74.7 ± 1.4	69.4 ± 1.5

Table 9.2: Pairwise digit classification rates for the software and hardware LSM. Certain pairs are easier to classify than others, depending on how similar the low-resolution representations of the digits are. Results taken from Probst (2011).

9.5 Summary

The early experimental setup presented by Jeltsch (2010) has been further developed to an on-chip LSM and has been published as part of Pfeil et al. (2013). Two different learning strategies have been explored and a method devised to train a tempotron purely based on its spike response. This allows to train more tempotron readouts in parallel on hardware, where the access to membrane traces is limited. The universality of the LSM setup has been demonstrated for three different tasks. The demanding task even showed that a liquid can improve classification performance beyond the scope of the single readout. Lastly, the setup and a future implementation on HICANN are discussed in the final discussion at the end of this thesis.

Discussion and Outlook

As part of this thesis, a fast, scalable multi-user workflow for the HMF has been designed and implemented. The resulting software framework provides access to the system from the lowest hardware to the most abstract PyNN level. The latter is achieved via a redesigned mapping of neural network descriptions into hardware specific configurations. This mapping is algorithmically more scalable than its predecessor and speeds up the process by making use of shared-memory parallelization allowing to map larger networks than before. Modular libraries have been developed to handle hardware imperfections and defects. A new high performance PyNN-compatible user interface seamlessly integrates with native `C++` backends and preserves the hierarchy of the network descriptions as such. The new workflow, the mapping framework and the new PyNN interface have been comprehensively benchmarked and verified. The findings for all major workflow components as well as the hardware liquid state machine setup are discussed below.

Embedded Classification with Liquid State Machines

An implementation of a liquid state machine, that is suitable for accelerated neuromorphic hardware systems, has been presented. The results have been published as part of Pfeil et al. (2013). The universality of the setup has been exemplified for different tasks, including a memory challenge and handwritten digit recognition. Furthermore, a method has been devised for a purely spike-based training of tempotron readouts that works without voltage recordings. It can be used to train many instances in parallel as access to voltage traces is limited on hardware.

First experiments on the new USB platform promise a 17 fold speedup and the next Spikey revision, which has recently arrived, provides extra neurons. It will also support implementations using strong STP and no STP for the self-stabilizing liquid and tempotron, respectively.

The next step is to port the LSM setup to HICANN. As a starting point, the more complex AdEx neurons can be simplified to LIF ones, closely mimicking the Spikey neurons. Subsequently, more advanced learning strategies may be able to exploit their richer neuron dynamics. Moreover, the spike mechanism on HICANN can be disabled, thus resembling the original weight update rule in the $(-)$ case more faithfully. However, a prior calibration of synaptic inputs is mandatory because they have found to saturate early and a roughly linear dependency between digital weight and PSP amplitude is necessary.

The tempotron has shown to be a suitable neuromorphic classifier for accelerated neuromorphic hardware systems and can easily be embedded into other networks due to its modest single-neuron requirement. Therefore, network responses can be collapsed into bandwidth friendly single spikes to speed up result aggregation and to evaluate experiments efficiently in the accelerated hardware domain. The LSM and tempotron, in particular, can be considered a valuable first contribution to a future neuromorphic toolbox that enables users to perform on-system network analysis.

A Hierarchical Neural Network Representation

PyHMF, a PyNN-compatible high performance library for the description and representation of spike-based abstract neural networks, has been developed. It has shown to be typically more than two orders of magnitude faster than other PyNN backends. This performance leap has been achieved by storing synapses in matrices rather than adjacency lists on the one hand and a more efficient native `C++` implementation on the other. The latter avoids the overhead of frequent foreign language calls from `Python` into `C` for random number generation. The compliant PyNN API is automatically wrapped from `C++`, thus updated on-the-fly whenever the underlying interface changes.

An important advantage of PyHMF over PyNN is its ability to preserve hierarchical information that is expressed in the actual model description. PyNN on the other hand, expands populations, assemblies and population views instantly into flat lists of neurons. A reduction of the PyNN hypergraph to a plain graph has been derived to capture the hierarchy and keep the topology algorithmically more tractable. The hierarchy can be used by backends e.g., to partition the network on an abstract level for bandwidth optimized simulations on a computer cluster or neuromorphic hardware systems alike.

Notably, all available networks in the group's model repositories have successfully been set up using PyHMF. However, PyHMF does not yet handle multiple synapses between a pair of neurons as part of a single projection. This is an immediate consequence of using connection matrices rather than adjacency lists. This shortcoming can currently be worked around by expressing an isomorphic network using multiple projections. Future versions may also detect congruent synapses and automatically split them into multiple projections. Nevertheless, the EPFL cortical column model in Section 8.7 is the only studied model that implements such connectivity. Furthermore, native evaluation of math expression has been presented in Section 8.8, which significantly speeds up the distance dependent probability connector.

In conclusion, PyHMF is a stand-alone PyNN-compatible, high-level and high performance interface. Other simulator backends could also benefit from its performance advantage, seamless native code integration and preserved model hierarchy. In fact, PyHMF might be an interesting candidate for a shared development effort within the HBP (Markram, 2012b). For example, the lazy evaluation of synapses could provide efficient model set up for simulations on large-scale computer clusters.

System-Level Software and Workflow Redesign

As part of this thesis, the low-level hardware interfaces and the workflow have been redesigned completely. The former provides unified access to all hardware components, while the latter has been split into a broker-based design to make the workflow more scalable and to support multi-user remote operation.

Firstly, the new low-level software foundation unifies the access to the HMF and therefore reduces the overall software complexity hardening the setup for production use. The comprehensive coordinate system and type-rich interfaces render code more expressive, capture misuse at compile time and out-of-bounds errors rigorously at runtime. The type-rich design has proven itself very useful and is now used for other prototype systems as well. Future versions of StHAL are planned to speed up wafer configuration by parallelizing the access across multiple FPGAs.

For the new multi-user workflow, all necessary components have been established for setting up the network representation, transferring it to the experiment broker, managing the experiment execution and submitting the results back. PyHMF hides the more complex workflow from the user, everything happens in a PyNN-compatible fashion transparently in the background.

A first version of the Ester experiment broker is benchmarked in Section 2.3.2. It achieves experiment transmission rates of up to 1.6 GB/s, which is sufficient to handle many user requests in parallel and to saturate the 10 Gbit ethernet links of the HMF cluster nodes. The throughput can further be increased by separating experiments into a header and payload section to avoid unnecessary experiment deserialization on the Ester broker. Deployments can additionally be scaled beyond single broker boundaries by means of load balancers. Future Ester versions should, firstly, allow client reconnects to render access more user friendly and, secondly, support self-monitoring to reject incoming requests if running short of resources to avoid denial of service.

The workflow redesign is a major step towards opening up the HMF as a novel computational platform for the neuroscience community in accordance with the open-access policy of the HBP.

Mapping of Neural Networks to Hardware

The marocco mapping framework has been established as part of this thesis. Scalable algorithms and the use of shared-memory parallelization allow to map networks more efficiently than with the previous implementation. The new implementation is robust and able to map all networks part of the work group's model repositories. The correctness of mapped configurations has been demonstrated by means of simple hardware experiments and an ESS Hellfire chain simulation using the prototype HALBe integration. During the design, a special focus has been put on extensibility and modularity to facilitate future development. Therefore, the mapping task has been decomposed into a well-defined feedforward flow. Furthermore, defect handling as well as calibrations have been split into independent libraries that are valuable for other workflow components alike. The new mapping extends upon the features of the prior implementation, is generally faster and the code base is only about one fifth the size of the MappingTool.

New means of manual guidance provide flexible control over the mapping, which is necessary to cope with the wide range of possible neural network topologies and to optimize their realization on the HMF:

- No more configuration files are necessary, the mapping is interfaced seamlessly from within the PyNN script by means of PyMarocco.
- A visualizer provides immediate access to the routing.
- The size of hardware neurons can be chosen individually for each population.
- Projections can be prioritized to render their realization more likely or to assert functionally important connections during the wafer routing.
- The most important improvement over the MappingTool is that now excitatory and inhibitory neurons can be placed to the same HICANN and neurons no longer need to obey Dale's principle (Strata and Harvey, 1999). This enables arbitrary single HICANN experiments as well as more flexible inter-neuron connectivity.

Defect Handling and Synapse Measurement

Wafer-scale integration is the key technique for energy efficiency and for the high connection densities of the HMF. However, defect chips that would otherwise be dismissed remain in the system. The wafer system has therefore been designed from the ground up to allow a wide range of workarounds that have to be implemented dynamically by the mapping. The ReDMan library has been introduced to manage defect maps for arbitrary components and to interface marocco.

Moreover, an automated classification of available synapses has been developed. The results presented in Section 7.2 currently suffer from uncalibrated, overly sensitive synaptic time constants that lead to many false defect classifications. Nonetheless, the rate of defect synapses has conservatively been estimated to be below 6 %, which is already an upper bound. Calibrations are expected to further improve the results. This exemplifies that most hardware synapses will be available for actual network experiments.

Performance Comparison

The performance of the old and new mapping flows as well as the two PyNN implementations have been benchmarked and compared against one another whenever possible including synaptic loss, runtime and memory footprint.

The synaptic loss is mostly similar for both workflows due to partly similar algorithmic approaches and common hardware constraints. Still, marocco produced better configurations for all instances of the worst-case homogeneous random networks.

With respect to runtime, PyHMF and marocco are consistently faster, the mapping e.g., uses more efficient algorithms and shared-memory parallelization. The actual speedup, however, depends on the size and topology of the model. For example, a 50 fold speedup has been achieved for wafer-scale homogeneous random networks with a connection probability of $p = 10\,\%$. The speedup is expected to be even higher for larger networks or higher connection probabilities due to the more efficient scaling. In conclusion, the goal of developing a significantly faster mapping has been achieved.

For large network models memory is the most expensive computational resource, ultimately limiting the maximum network extent. The memory footprint for worst-case and structured networks has been studied for homogeneous random networks and the layer 2/3 attractor model, respectively. Mapping homogeneous random networks at wafer-scale with marocco for any connection probability p requires approximately 21 GB of memory. Whereas the memory footprint of the MappingTool depends on p and has found to be roughly on par for $p = 5\,\%$ and to even be $45\,\%$ higher for $p = 10\,\%$ at full wafer-scale, despite using sparse network friendly adjacency lists. For marocco, most of the memory is actually consumed by the synapse matrices themselves, which shows that little overhead is imposed and memory is handled efficiently. For structured networks, like the layer 2/3 attractor model, marocco required on average 4 times less memory than the MappingTool. Overall, consistently using matrix representations for synapses offers clear performance benefits without any significant memory disadvantage.

HMF Topology Study

The suitability of the HMF for different networks has been studied. Random networks that require worst-case all-to-all connectivity between HICANNs can only be implemented with low distortions for up to about 10^4 neurons, before the Layer 1 network starts to congest. Planar networks and networks with an emphasis on local connectivity can be implemented up to much larger sizes with only few or no synaptic loss. Moreover, it has been found that most networks are ideally mapped using large hardware neurons consisting of 8 or 12 circuits to reduce local synaptic loss by increasing the synaptic input count per neuron. Note that this, however, reduces the effective number of mappable neurons. The next HICANN revision is planned to provide configurable conductances between circuits allowing to build complex compartmental dendrite models, making this feature compelling beyond dynamic input scaling.

The flexibility provided by marocco has been used to explore different Layer 1 network topologies. This study has shown that changing the bus swap from the first to the second HICANN version clearly improved the ability to route long-range connectivity. Moreover, the current layout of crossbar switches and bus swaps is well tuned, e.g., adding more switches or changing the bus swap can not significantly improve the routing performance. In order to realize more long-range connectivity, the

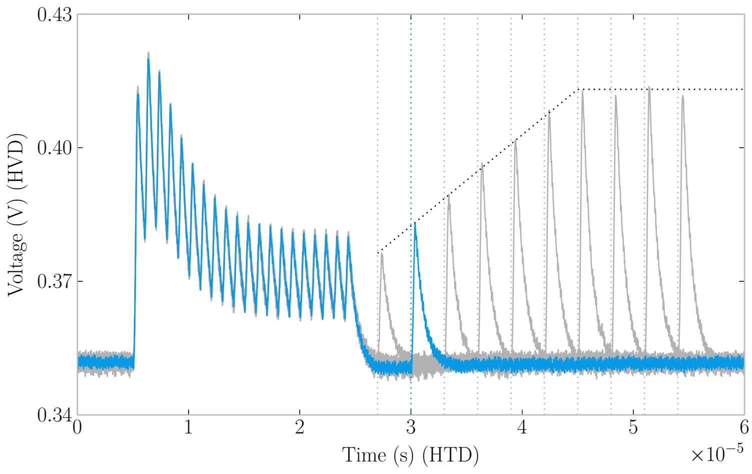


Figure 9.11: Measurement of the recovery time for the STP depression mechanism. A short initial burst saturates the inactive partition, a spike probe is then used to test the recovery of the PSP amplitude in consecutive experiments. Every trace is averaged over 50 repetitions. Interestingly, the course of recovery is linear rather than exponential as one would expect for the Tsodyks and Markram (1997) model. The storage capacitors are discharged by a current mirror rather than a resistive element according to Schemmel et al. (2007 Figure 4). The data has been provided by Sebastian Billaudelle.

bandwidth of horizontal buses has to be increased. The routing of local connectivity, on the other hand, can be improved by increasing the sustainable capacitive load of Layer 1 buses. This allows for larger numbers of interconnected synapse drivers (Kononov, 2011), which is necessary to map larger afferent address spaces, different STP parameters or share connections efficiently between adjacent chips.

Hardware Experiments and Future Mapping Development

All necessary tools to map and deploy networks for the HMF are in place. The Hellfire chain network has been mapped to hardware as a first more complex model spanning multiple HICANNs. An ESS simulation has verified that the network is mapped correctly. However, signal propagation along the chain has not yet been achieved on hardware. Nonetheless, this early attempt provided vital information to guide the development of calibrations and to identify a heretofore unknown repeater locking issue. As an interim solution for the latter, a workaround has been added to marocco, which dedicates a part of the merger tree to provide address 0 events to lock repeaters. However, this reduces the number of accessible neuron circuits per chip. The simplest thorough solution for future versions of the hardware is to connect the background generators on the lowest merger tier. In fact, the next revision is planned to substitute the merger tree with a more flexible merger crossbar (Schemmel, 2014b). Ultimately, comprehensive calibrations for multi-circuit neurons are expected to increase analog precision and therefore enable working Hellfire chain implementations.

The future software development has to address three major topics. Firstly, the integration of the ESS into the HALBe workflow, which is an important step towards testing the complete workflow with remote users. The ESS also simplifies continuous integration necessary to control the complexity of the growing software stack and to guide software quality (Duvall et al., 2007). Secondly, providing access to the hardware STP and STDP plasticity features. Support for STP is currently in development, an early measurement is shown in Figure 9.11. Interestingly, the recovery is linear rather than exponential, as expected for the Tsodyks and Markram (1997) model, because STP storage capacitors are discharged by a current mirror rather than a resistive element. Note that future ESS versions should account for this model deviation. Finally, a multi-wafer routing needs to be implemented in order to map models to future multi-wafer HMF deployments, however, the necessary Layer 2 event network and software interfaces are not yet available. The basic design of such a routing is discussed in the following.

Towards a 10^4 Columns Cortical Model

A large 10^4 wafer system is planned as part of the HBP. This system will provide the necessary resources to conduct accelerated spike-based emulations of at least 10^4 cortical columns, which roughly corresponds to 10 % of rat cortex (Markram, 2012a). Mapping a single EPFL cortical column currently takes about 3.2 minutes with the new workflow, which means 10^4 individual columns can be mapped in about 22 days on a single off-the-shelf computer.

Scaling the mapping towards thousands of cortical columns and wafers requires a hierarchical approach in order to distribute neurons efficiently such that connectivity between wafers is minimized, to conduct the mapping with limited memory resources and to distribute the computation. This has in fact been one of the main rationals behind the development of the hierarchy preserving PyHMF. The lazy evaluation of synapses will allow to easily build a coarse 10^4 column representation on a single machine to perform an initial partitioning. Multiple wafers can then be placed and routed mostly independent and thus in parallel. Connections between the individual wafers have to be routed afterwards e.g., by modeling the Layer 2 network as a graph and using an iterative search algorithm like the *Iterative Shortest Path Routing* described in Section 6.4.3. Given the assumption that columns are mostly connected to neighboring columns, such a routing is expected to be even faster than routing the dense on-wafer connectivity. A complete 10^4 column network can then be set up in just a few minutes to hours depending on how efficiently and to how many cluster nodes the mapping of the individual columns is distributed. In cases where mapping time is critical and computational resources are limited it might also be an option to replicate a single column onto multiple wafer instances. However, these configurations cannot account for individual wafer variations and defects. Similarly, the inter-wafer routing can be set up symmetrically, if the connectivity between columns is reasonable generic and nicely matches a 2D toroidal wafer grid network.

In the near future, the HMF will open up exciting opportunities to conduct detailed long-term learning experiments of large cortex areas. While detailed simulations of a single column on the BlueBrain cluster are about 300 times slower than biological real time (Markram, 2012a), emulating one year of development of the 10^4 columns network takes less than an hour on the HMF with a speedup of 10^4.

Appendix

A.1 Test Setup

Computer Setup

All measurements have been carried out on one of the HMF cluster nodes, which is running Debian Wheezy with the 3.10.111 bpo70+1 (2013-09-24) kernel. The node is equipped as follows:

- Intel® Core™ i7-2600 CPU @ 3.40GHz
- 32GB memory
- OCZ Vertex 2 SSD 128GB

Software Setup

The software has been compiled using `g++-4.7.2` (GCC, 2014) with `-Os` optimizations. No special link time optimization have been applied. The `Python` interpreter is `CPython 2.7.3` (Python, 2014).

A.2 Parameter Sets

The following sections summarize the default HALBe parameters used in experiments.

Default Hardware Parameters

Four floating-gate arrays per HICANN store analog parameters in 129×24 cells each. Both, current and voltage cells exist, which cover parameter ranges of 0 µA to 2.5 µA and 0 V to 1.8 V respectively. These cells are programmed via an ADC, which linearly maps 10 bit digital values to the corresponding parameter ranges.

Shared Floating Gate Parameters One column of 24 cells per array is devoted to shared parameters, which are used by multiple functional circuit instances. Note that the assignment of parameters to functional elements may vary depending on the parameter and the actual floating-gate array.

Parameter	Value	Component	Description
$I_{\text{op\_bias}}$	1023	-	Internal OP bias
V_{dllres}	200	-	DLL control voltage of receivers is pulled to this voltage during reset of PLL
V_{ccas}	800	Layer 1	Biasing of Layer 1 input amplifier and V_{cbias}
V_{reset}	300	Neuron	Voltage, membrane is pulled to during reset pulse (left:even, right:odd neuron circuits)
V_{bout}	1023	Neuron	Global biasing of neuron readout
V_{bexp}	1023	Neuron	Lower exp voltage driver bias
I_{breset}	1023	Neuron	Current used to pull down membrane to reset potential(left:even, right:odd neuron circuits)
I_{bstim}	1023	Neuron	Bias for neuron stimulation circuit
V_{gmax0}	1000	Synapse Driver	Synaptic base-conductance
V_{gmax1}	1000	Synapse Driver	Synaptic base-conductance
V_{gmax2}	1000	Synapse Driver	Synaptic base-conductance
V_{gmax3}	1000	Synapse Driver	Synaptic base-conductance
V_{fac}	0	STP	STP voltage V_{fac} used for facilitation mode
V_{dep}	0	STP	STP voltage V_{dep} used for depression mode
V_{stdf}	0	STP	STP reset voltage for facilitation
V_{bstdf}	0	STP	STP bias
V_{dtc}	0	STP	STP dtc bias
V_{thigh}	0	STDP	STDP readout compare voltageB
V_{tlow}	0	STDP	STDP readout compare voltage
V_{clra}	0	STDP	STDP clr voltage (acausal)
V_{clrc}	0	STDP	STDP clr voltage (causal)
V_{m}	0	STDP	Start load voltage of causal STDP capacitor (ground for acausal)
V_{br}	0	STDP	Bias for STDP readout circuit

Neuron Floating Gate Parameters Individual parameters for 128 neurons are stored per array in 128 columns of 24 rows. The parameters are intuitively mapped to neuron circuits, e.g., the top-right array provides voltages and currents for the 128 top-right neuron instances.

Parameter	Value	Description
E_{l}	300	Leakage reversal potential
E_{syni}	100	Synapse inhibitory reversal potential
E_{synx}	570	Synapse excitatory reversal potential
I_{bexp}	1023	Biasing for operational amplifier of exponential circuit (technical)
I_{convi}	1023	Controls maximum inhibitory synaptic conductance/current
I_{convx}	1023	Controls maximum excitatory synaptic conductance/current
I_{fire}	511	Current flow onto the adaptation capacitance per spike; Integrated for spike pulse length
I_{gl}	200	Proportional to leakage conductance for small signals and maximum leakage current
I_{gladapt}	100	Adaptation conductance modeling parameter a in AdEx equation

I_{intbbi}	511	Bias for operational amplifier in inhibitory synapse input circuits (technical)
I_{intbbx}	511	Bias for operational amplifier in excitatory synapse input circuits (technical)
I_{pl}	511	Current for adjustment of refractory period (integrated)
I_{radapt}	511	Conductance for adjustment of adaptation time constant
I_{rexp}	511	Exponential slope current; controls ΔT
I_{spikeamp}	1023	Bias current of spike threshold comparator (technical)
V_{exp}	400	Exponential reference potential
V_{syni}	511	Synapse excitatory reversal potential
V_{syntci}	800	Control voltage for the time constant of inhibitory synaptic pulses
V_{syntcx}	800	Control voltage for the time constant of excitatory synaptic pulses
V_{synx}	511	Zero voltage of collecting line from synapse array (technical)
V_{t}	500	Membrane Voltage needed to detect a spike

External Voltages Apart from the floating gate voltage and current parameters the repeater voltages V_{OL} and V_{OH} are provided externally. They control the baseline power consumption of the pre-amplifiers and therefore the Layer 1 transmission reliability. By default hey are set to $V_{\text{OL}} = 0.7\,\text{V}$ and $V_{\text{OH}} = 0.9\,\text{V}$.

Layer 2/3 Attractor Model Parameter

The studied network instances part of the performance comparison in Section 8.5 had the following hyper and minicolumn geometries:

(HC, MC)	(9, 3)	(18, 2)	(9, 6)	(27, 3)	(18, 6)
Neurons	891	1188	1782	2673	3564
Synapses	157934	205929	336860	473627	673231

(HC, MC)	(36,4)	(9,18)	(18,12)	(27,9)	(17,18)
Neurons	4752	5346	7128	8019	10692
Synapses	860702	1209031	1479046	1588852	2418186

Liquid Parameter

The measurements on the backplane and the new USB platform have been carried out on Spikey no. 436 and 603, respectively. For the USB platform, mainly the V_{thresh} needed to be increased. However, both platforms use different calibrations, thus parameters are not immediately comparable.

Parameter	Value
V_{thresh}	$-53\,\mathrm{mV}$
V_{reset}	$-63\,\mathrm{mV}$
E_{l}	$-58\,\mathrm{mV}$
E_{i}	$-80\,\mathrm{mV}$
g_{l}	$20\,\mathrm{nS}$
τ_{s}	$2.5\,\mathrm{ms}$

Table A.5: Backplane Spikey

Parameter	Value
V_{thresh}	$-43\,\mathrm{mV}$
V_{reset}	$-63\,\mathrm{mV}$
E_{l}	$-59\,\mathrm{mV}$
E_{i}	$-80\,\mathrm{mV}$
g_{l}	$20\,\mathrm{nS}$
τ_{s}	$2.5\,\mathrm{ms}$

Table A.6: USB Spikey

Current Input

Membrane recordings for different behaviors of the AdEx neuron model are presented in Section 8.2. Neurons have been built from 4 individual circuits. In the following, only the non-default parameters are outlined in 10 bit DAC units.

Parameter	Phasic Spiking	Spike Frequency Adaptation
I_{fire}	20	20
I_{gladapt}	1000	100
I_{radapt}	500	300
V_{exp}	250	250
V_{t}	550	550
V_{reset}	290	300

Hellfire Hardware Study

The following parameters have been used to implement the Hellfire chain model on hardware. The parameters are the result of translating the ideal model parameters using the available default neuron transformation. Note that the model parameter have been chosen such that the threshold and leakage potential V_{t} and E_{l} end up close. They are less than 40 DAC apart. Putting them even closer to increase the excitability of neurons, however, can lead to spontaneous spiking due to individual circuit variations and noise. Furthermore, the small neuron capacitances of 0.16 pF have been used.

The measurement has been carried out on the HICANNs from $(X, Y) = (16, 10)$ to $(19, 10)$ on the first prototype system.

Parameter	Value
E_l	281
E_syni	227
E_synx	691
I_bexp	1023
I_convi	1023
I_convx	1023
I_fire	22
I_gl	133
I_gladapt	34
I_intbbi	1023
I_intbbx	1023

Parameter	Value
I_pl	511
I_radapt	128
I_rexp	1023
I_spikeamp	1023
V_exp	204
V_syni	511
V_syntci	795
V_syntcx	795
V_synx	511
V_t	320

Recurrent Hellfire connectivity from the Nth chain link to itself, N-1th, N+1th and N+2th chain links. Projections are set up using an `AllToAllConnector` as follows:

Pre	Post	Weight (nS)
E^N_carrier	E^N_carrier	3.3
E^N_carrier	I^N_carrier	1.0
I^N_carrier	E^N_carrier	1.0
E^N_control	E^N_control	2.0
I^N_control	E^N_control	15.3
E^N_control	I^N_control	10.0
E^N_carrier	E^N_control	1.0
E^N_carrier	I^N_control	1.0

Pre	Post	Weight (nS)
E^N_carrier	E^{N+1}_carrier	3.3
I^N_control	E^{N+2}_carrier	15.0
E^N_control	I^{N+2}_carrier	5.0
I^N_control	E^{N-1}_carrier	15.0
E^N_control	I^{N-1}_carrier	5.0

LSM Setup

The following list of parameters has been used for the hardware and software LSM implementations in Chapter 9. Note that, the parameters are implicitly used by both, liquid and tempotron because odd and even neurons on Spikey share common parameters. V_thresh has individually been tuned for different Spikey chips and different task.

Parameter	Value
V_reset	$-63.0\,\text{mV}$
V_rest	$-58.0\,\text{mV}$
E_i	$-80.0\,\text{mV}$
g_l	$20.0\,\text{nS}$
τ_s	$2.5\,\text{ms}$

A.3 Further Measurements

June Simulation

The simulation results for the June network with different kinds of synaptic loss are shown in Figure A.12. They have been included on request of Mihai Petrovici and reasons of completeness. However, a comprehensive discussion of the June network dynamics is beyond the scope of this thesis.

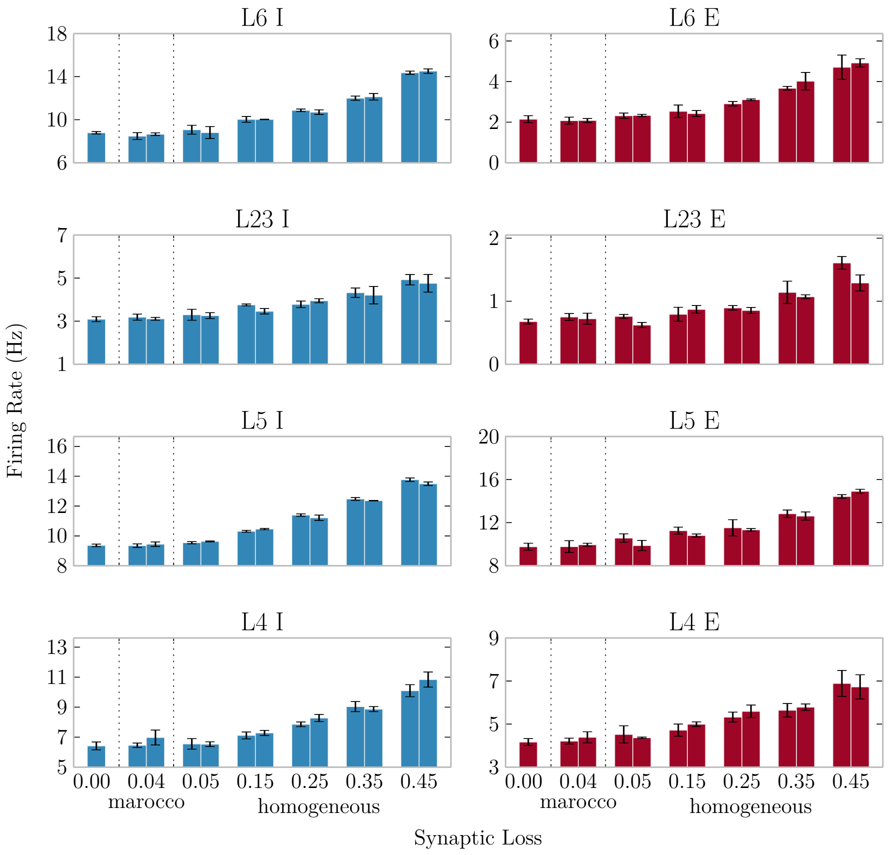


Figure A.12: Average firing rates for the individual cortical layers of the June network simulated with different kinds of synaptic loss. The left-hand bar in each plot corresponds to undistorted reference simulations. The following bars denote the results for the 4 % inhomogeneous loss generated by marocco as well as 5 %, 15 %, 25 %, 35 % and 45 % of homogeneous loss. Two bars are shown for simulations including loss, where the left and right bar refer to simulations without and with compensation, respectively. The compensation simply scales synaptic weights according to $1 - \nu_{\text{loss}}$ and has been proposed by Mihai Petrovici. However in this case, the method can not compensate for the increase in firing rates towards higher synaptic loss.

Acronyms

ADC Analog Digital Converter 44, 95, 98, 173

AdEx Adaptive Exponential Integrate-and-Fire 8, 84–86, 106, 107, 115, 137, 147, 165, 174, 176

AI Asynchronous Irregular 105, 137, 138, 141, 143, 145

API Application Programming Interface 21, 29–33, 35, 37, 38, 40, 44, 166

BSP BrainScaleS Project 2, 5, 17, 48, 50, 51

BTD Biological Time Domain 14, 112, 113

CMOS Complementary Metal–Oxide–Semiconductor 56, 83

DAC Digital Analog Converter 101, 102, 113, 176

DDPC DistanceDependentProbabilityConnector 138, 139, 166

DLL Delay-Locked Loop 11, 14, 62, 87, 97, 100, 105, 111, 115, 116

DNC Digital Network Chip 10, 11, 16, 17, 38, 45, 61–63

ESS Executable System Specification 20–23, 39, 42, 43, 105, 111, 113, 114, 136, 167, 170, 171

FACETS Fast Analog Computing with Emergent Transient States 19, 48, 130, 134

FPGA Field Programmable Gate Array 2, 6, 9–11, 15–17, 38, 43, 44, 52, 54, 63, 86, 87, 111, 115, 116, 136, 167

HAL Hardware Abstraction Layer 35, 37–45, 57, 63, 83, 94, 96, 106, 107, 111, 113, 114, 126, 136, 167, 171, 173

HBP Human Brain Project 1, 2, 17, 20, 48, 50, 166, 167, 171

HICANN High Input Count Analog Neural Network 3, 5–7, 9–13, 15–17, 38–42, 44, 45, 58, 59, 63, 66, 67, 70, 71, 73, 75, 77, 82, 83, 87–89, 91, 92, 94, 98, 99, 103, 105, 109, 111, 115–117, 119, 124, 126–129, 133, 140, 143, 144, 147, 148, 163, 165, 168–170, 173, 176

HMF Hybrid Multi-Scale Facility 2–6, 15, 16, 19–23, 25, 26, 29, 33–35, 47, 48, 58, 60, 91, 93, 94, 105, 116, 125–127, 133, 135, 136, 141–144, 165, 167–173

HTD Hardware Time Domain 17, 56, 87, 106, 170

HVD Hardware Voltage Domain 96, 101, 106, 112, 170

IPC Inter Process Communication 21, 24, 26, 31, 48, 88

ISI Inter Spike Interval 87, 95, 107

LIF Leaky-Integrate and Fire 147–149, 151, 165

LSB Least Significant Bit 13, 14, 55, 153

LSM Liquid State Machine 3, 4, 147, 148, 151, 155, 156, 158–161, 163, 165, 166, 177, I

MRST Minimum Rectilinear Steiner Tree 55, 69

MSB Most Significant Bit 13, 55, 62, 74, 81

NP Nondeterministic Polynomial time 51, 55, 69, 73

PCB Printed Circuit Board 15, 148

PEiS Prior Evaluation in Software 157, 159

PSP Post-Synaptic Potential 7, 9, 14, 94, 95, 100–102, 112, 113, 115, 116, 149–151, 165, 170

PST Preserved Synapse Type 157, 158

PyNN Python Neural Networks 3, 19–21, 29–35, 57, 58, 66, 75, 86–88, 106, 107, 111, 121, 123, 124, 134, 136, 139, 165–168

RSNP Regular Spiking Non-Pyramidal 131, 133, 134, 143

SPL1 Synchronous Parallel Layer 1 9–12, 45, 52–55, 60–72, 88, 97, 98, 109, 144

STDP Spike Timing Dependent Plasticity 9, 86, 171, 174

STP Short Term Plasticity 9, 14, 55, 73–76, 154, 161, 165, 170, 171, 174

WTA Winner-Take-All 130, 131

Bibliography

D. Abrahams and R. Grosse-Kunstleve. Building hybrid systems with Boost.Python, 2003. URL `http://www.boostpro.com/writing/bpl.pdf`.

M. A. Aizerman, E. A. Braverman, and L. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. In *Automation and Remote Control,*, number 25 in Automation and Remote Control,, pages 821–837, 1964.

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560.

D. Attwell and S. B. Laughlin. An energy budget for signaling in the grey matter of the brain. *Journal of Cerebral Blood Flow & Metabolism*, 21(10):1133–1145, 2001.

F. A. Azevedo, L. R. Carvalho, L. T. Grinberg, J. M. Farfel, R. E. Ferretti, R. E. Leite, R. Lent, S. Herculano-Houzel, et al. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, 513(5):532–541, 2009.

S. Baker, R. Spinks, A. Jackson, and R. Lemon. Synchronization in monkey motor cortex during a precision grip task. i. task-dependent modulation in single-unit synchrony. *Journal of Neurophysiology*, 85(2):869–885, 2001.

G. Barany. Python interpreter performance deconstructed. In *Proceedings on "Dyla'14 at PLDI, Edinburgh, UK"*, June 2014.

M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies. *IBM Reserach Report, RC24704 (W0812-047)*, 2008.

S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science and Engineering*, 13.2:31–39, 2011.

J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975. ISSN 0001-0782. doi: 10.1145/361002.361007.

I. Berg. *muparser Version 2.2.3 Website*, 2014. URL `http://muparser.beltoforion.de/`.

J. Bill, K. Schuch, D. Brüderle, J. Schemmel, W. Maass, and K. Meier. Compensating inhomogeneities of neuromorphic VLSI devices via short-term synaptic plasticity. 4 (129), 2010a.

J. Bill, 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(129), 2010b.

R. Biro, F. N. van Kempen, and D. Becker. *Linux Pseudo-driver for the loopback interface implementation*, 1993. URL `http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/drivers/net/loopback.c`.

P. E. Black. Greedy algorithm. *Dictionary of Algorithms and Data Structures*, 2005. URL `http://xlinux.nist.gov/dads/`.

J. L. Blanco. *nanoflann Version 1.1.7 Website*, 2013. URL `http://code.google.com/p/nanoflann/`.

Boost. *Boost Interval Container Library Version 1.49.0 Website*, 2010. URL `http://www.boost.org/doc/libs/1_49_0/libs/icl/doc/html/index.html`.

M. K. Bowman-Amuah. Load balancer in environment services patterns, June 2003. US Patent 6,578,068.

BrainScaleS. Brain-inspired multiscale computation in neuromorphic hybrid systems (project number 269921), 2014. URL `http://www.brainscales.eu`.

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

D. Brüderle. *Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System*. PhD thesis, Universität Heidelberg, 2009.

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

D. Brüderle, M. Petrovici, B. Vogginger, M. Ehrlich, T. Pfeil, S. Millner, A. Grübl, K. Wendt, E. Müller, M.-O. Schwartz, D. de Oliveira, S. Jeltsch, J. Fieres, M. Schilling, P. Müller, O. Breitwieser, V. Petkov, L. Muller, A. Davison, P. Krishnamurthy, J. Kremkow, M. Lundqvist, E. Muller, J. Partzsch, S. Scholze, L. Zühl, C. Mayr, A. Destexhe, M. Diesmann, T. Potjans, A. Lansner, R. Schüffny, J. Schemmel, and K. Meier. A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems. *Biological Cybernetics*, 104:263–296, 2011. ISSN 0340-1200. doi: 10.1007/s00422-011-0435-9.

D. Brüderle et al. Simulator-like exploration of network architectures with the facets hardware systems and pynn, 2010. URL `https://capocaccia.ethz.ch/capo/wiki/2010/facetshw10`.

L. Buesing, J. Bill, B. Nessler, and W. Maass. Neural dynamics as sampling: A model for stochastic computation in recurrent networks of spiking neurons. *PLoS Computational Biology*, 7(11):e1002211, 2011.

D. D. Clarke and L. Sokoloff. *Basic Neurochemistry: Molecular, Cellular and Medical Aspects*, chapter Circulation and Energy Metabolism of the Brain, page 637–670. Lippincott Williams and Wilkins, Philadelphia: Lippincott-Raven, 6 edition, 1999.

J. D. Cohen, S. M. McClure, and J. Y. Angela. Should i stay or should i go? how the human brain manages the trade-off between exploitation and exploration. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 362(1481):933–942, 2007.

S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi: 10.1145/800157.805047.

J. O. Coplien. Curiously recurring template patterns. 1000 East Warrenville Rd., Naperville, IL 60566 USA, 1995. AT&T Bell Laboratories.

M. Corbetta and G. L. Shulman. Control of goal-directed and stimulus-driven attention in the brain. *Nature reviews neuroscience*, 3(3):201–215, 2002.

A. Davison. Personal communication, 2011a.

A. Davison. *PyNN 0.7 API Documentation*, september 2011b. URL `http://neuralensemble.org/trac/PyNN/wiki/API-0.7`.

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

S. R. Deiss, R. J. Douglas, A. M. Whatley, and A. M. A pulse-coded communications infrastructure for neuromorphic systems, 1998.

Delta V. *Remote Call Framework Website*, 2013. URL `http://www.deltavsoft.com`.

A. Destexhe. Self-sustained asynchronous irregular states and Up/Down states in thalamic, cortical and thalamocortical networks of nonlinear integrate-and-fire neurons. *Journal of Computational Neuroscience*, 3:493 – 506, 2009.

A. Destexhe, M. Rudolph, and D. Pare. The high-conductance state of neocortical neurons in vivo. *Nature Reviews Neuroscience*, 4:739–751, 2003a.

A. Destexhe, M. Rudolph, and D. Paré. The high-conductance state of neocortical neurons in vivo. *Nature reviews neuroscience*, 4(9):739–751, 2003b.

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

E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0029-599X. doi: 10.1007/BF01386390.

M. Djurfeldt, M. Lundqvist, C. Johansson, M. Rehn, O. Ekeberg, and A. Lansner. Brain-scale simulation of the neocortex on the ibm blue gene/l supercomputer. *IBM Journal of Research and Development*, 52(1.2):31–41, January 2008.

D. A. Drachman. Do we have brain to spare? *Neurology*, 64(12):2004–2005, June 2005. ISSN 0028-3878. doi: 10.1212/01.wnl.0000166914.38327.bb.

P. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Signature Series. Pearson Education, 2007. ISBN 9780321630148. URL `http://books.google.de/books?id=PV9qfEdv9L0C`.

G. Dósa. The tight bound of first fit decreasing bin-packing algorithm is ffd(i)11/9opt(i)+6/9. In B. Chen, M. Paterson, and G. Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, volume 4614 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74449-8. doi: 10.1007/978-3-540-74450-4_1.

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

S. Even and G. Even. *Graph Algorithms*. Cambridge University Press, 2011. ISBN 9781139504157.

FACETS. Fast analog computing with emergent transient states, 2010. URL `http://www.facets-project.org`.

J. Fieres, 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)*, 2008.

M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987. ISSN 0004-5411. doi: 10.1145/28869.28874.

I. Free Software Foundation. *libmatheval Version 1.1.8 Website*, 2014. URL `http://www.gnu.org/software/libmatheval/`.

GCC. *The GNU Compiler Collection Website.* Free Software Foundation Inc., 59 Temple Place Boston MA, USA, 2014. URL `http://gcc.gnu.org/`.

A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), Aug. 2011. URL `http://www.ietf.org/rfc/rfc6101.txt`.

S. Friedmann. *A new approach to learning in neuromorphic hardware.* PhD thesis, Universität Heidelberg, 2013.

S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown. Overview of the SpiNNaker system architecture. *IEEE Transactions on Computers*, 99(PrePrints), 2012. ISSN 0018-9340. doi: 10.1109/TC.2012.142.

N. Galli, B. Seybold, and K. Simon. Compression of sparse matrices: Achieving almost minimal table size. In *Proceedings of Conference on "Algorithms and Experiments "(ALEX98) Trento, Italy*, pages 27–33, Feb 1998.

F. Galluppi, A. Rast, S. Davies, and S. Furber. A general-purpose model translation system for a universal neural chip. In K. Wong, B. Mendis, and A. Bouzerdoum, editors, *Neural Information Processing. Theory and Algorithms*, volume 6443 of *Lecture Notes in Computer Science*, pages 58–65. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-17536-7. doi: 10.1007/978-3-642-17537-4_8.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1st edition, 1979. ISBN 978-0716710455.

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

D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN 0201157675.

M. Gort and J. H. Anderson. Deterministic multi-core parallel routing for fpgas. pages 78–86. IEEE, 2010. ISBN 978-1-4244-8981-7. doi: 10.1109/FPT.2010.5681758.

R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

A. Grübl. Personal communication, January 2014a.

A. Grübl. Personal communication, March 2014b.

R. Gütig and H. Sompolinsky. The tempotron: a neuron that learns spike timing-based decisions. 9(3):420–428, Mar 2006.

M. Handley, E. Rescorla, and IAB. Internet Denial-of-Service Considerations. RFC 4732 (Informational), Dec. 2006. URL `http://www.ietf.org/rfc/rfc4732.txt`.

HBP SP9. *Neuromorphic Platform Specification.* Human Brain Project, Mar. 2014.

D. Heeger. Poisson model of spike generation. Technical report, 2000. URL `http://www.cns.nyu.edu/~david/handouts/poisson.pdf`.

J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2012.

N. Holford. Internship – spike-based classification on accelerated neuromorphic hardware, June 2011. URL `http://www.kip.uni-heidelberg.de/cms/fileadmin/groups/vision/Downloads/Internship_Reports/report_nholford.pdf`.

K. Husmann. Internship – HMF transmitter, Nov. 2011. URL `http://www.kip.uni-heidelberg.de/cms/fileadmin/groups/vision/Downloads/Internship_Reports/report_khusmann.pdf`.

K.-H. Husmann. Handling spike data in an accelerated neuromorphic system. Bachelor thesis, Universität Heidelberg, 2012.

C. Hwang and A. Masud. *Multiple objective decision making, methods and applications: a state-of-the-art survey.* Lecture notes in economics and mathematical systems. Springer-Verlag, 1979. ISBN 9780387091112.

ISO/IEC 7498-1:1994. Information Technology — Open Systems Interconnection — Basic Reference Model: The Basic Model. ISO/IEC 7498-1:1994, ISO, Geneva, Switzerland, Nov. 1994. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=20269`.

H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. Technical Report GMD Report 148, German National Research Center for Information Technology, St. Augustin, Germany, 2001.

S. Jeltsch. Computing with transient states on a neuromorphic multi-chip environment. Diploma thesis, Universität Heidelberg, 2010. HD-KIP 10-54.

A. B. Kahng and G. Robins. A new class of iterative steiner tree heuristics with good performance. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(7):893–902, 1992. doi: 10.1109/43.144853.

A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu. *VLSI Physical Design: From Graph Partitioning to Timing Closure.* Springer Verlag, 1st edition, 2011. ISBN 978-9048195909.

R. M. Karp. Reducibility among combinatorial problems. In R. Miller, J. Thatcher, and J. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972. ISBN 978-1-4684-2003-6. doi: 10.1007/978-1-4684-2001-2_9.

G. Karypis, V. Kumar, and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

M. Kleider. Personal communication, February 2014.

J. Klähn. Untersuchung und Management von Synapsendefektverteilungen in einem großskaligen neuromorphen Hardwaresystem. Bachelor thesis (german), Universität Heidelberg, 2013. HD-KIP 13-36.

P. Kobalicek. *mathpresso - Mathematical Expression Evaluator And Jit Compiler for C++ Language Project Page*, 2010. URL `http://code.google.com/p/mathpresso/`.

C. Koke. Personal communication, April 2014.

A. Kononov. Testing of an analog neuromorphic network chip. Diploma thesis, Universität Heidelberg, 2011. HD-KIP-11-83.

J. Kremkow, L. U. Perrinet, G. S. Masson, and A. Aertsen. Functional consequences of correlated excitatory and inhibitory conductances in cortical networks. 28(3):579–594, 2010.

H. Krupnova and G. Saucier. Fpga technology snapshot: Current devices and design tools. In *IEEE International Workshop on Rapid System Prototyping*, page 200, 2000.

H. Kubo. Method of scheduling a job in a clustered computer system and device therefor, March 1999. US Patent 5,881,284.

S. Kunkel, T. C. Potjans, J. M. Eppler, H. E. E. Plesser, A. Morrison, and M. Diesmann. Meeting the memory challenges of brain-scale network simulation. *Frontiers in neuroinformatics*, 5:35, 2012.

A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

Y. LeCun and C. Cortes. The mnist database of handwritten digits, 1998.

C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, Oct. 1985. ISSN 0018-9340.

J. Loeser and H. Haertig. Low-latency hard real-time communication over switched ethernet. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 13–22, June 2004. doi: 10.1109/EMRTS.2004.1310992.

M. Lundqvist, M. Rehn, M. Djurfeldt, and A. Lansner. Attractor dynamics in a modular network of neocortex. *Network: Computation in Neural Systems*, 17:3:253–276, 2006.

M. Lundqvist, A. Compte, and A. Lansner. Bistable, irregular firing and population oscillations in a modular attractor memory network. *PLoS Comput Biol*, 6(6), 06 2010.

W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: a new framwork for neural compuation based on perturbation. 14(11):2531–2560, 2002.

H. Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

H. Markram. Blue brain project, August 2012a. URL `http://www.artificialbrains.com/blue-brain-project`.

H. Markram. The human brain project. *Scientific American*, 306(6):50–55, 2012b.

H. Markram and K. Meier. The human brain project - preparatory study, 2012. URL `http://cordis.europa.eu/fp7/ict/programme/fet/flagship/doc/6pilots-hbp-publicreport_en.pdf`. A Report to the European Commission.

S. Meyers. *More Effective C++*. Addison Wesley Longman, Amsterdam, December 1995. ISBN 978-0201633719.

S. Millner. Personal communication, 2011.

S. Millner. *Development of a Multi-Compartment Neuron Model Emulation*. PhD thesis, Universität Heidelberg, November 2012.

S. Millner, A. Hartel, J. Schemmel, and K. Meier. Towards biologically realistic multi-compartment neuron model emulation in analog VLSI. In *Proceedings ESANN 2012*, 2012.

M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-13316-4.

MongoDB Website. MongoDB Inc., 2013. URL `http://www.mongodb.org/`.

R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995. ISBN 0-521-47465-5, 9780521474658.

L. Muller and A. Davison. Asynchronous irregular and up/down states in excitatory and inhibitory neural networks. 2009. URL `https://senselab.med.yale.edu/modeldb/ShowModel.asp?model=126466`.

T. Nagy. *WAF - The Meta Build System Website*, 2014. URL `https://code.google.com/p/waf/`.

R. Naud, N. Marcille, C. Clopath, and W. Gerstner. Firing patterns in the adaptive exponential integrate-and-fire model. *Biological Cybernetics*, 99(4):335–347, Nov 2008. doi: 10.1007/s00422-008-0264-7.

B. Nessler, M. Pfeiffer, L. Buesing, and W. Maass. Bayesian computation emerges in generic cortical microcircuits through spike-timing-dependent plasticity. *PLoS computational biology*, 9(4):e1003037, 2013.

PyNN. *A Python package for simulator-independent specification of neuronal network models Website*. The NeuralEnsemble Initiative, 2014. URL `http://www.neuralensemble.org/PyNN`.

Nokia. Qt cross-platform application framework. `http://www.qtsoftware.com/`, 2009.

C. Pape. Vergleich der Executable System Specification mit neuromorpher Hardware über eine gemeinsame Bedienungsschnittstelle. Bachelor thesis (german), Universität Heidelberg, 2013. HD-KIP 13-93-55.

K. E. Parsopoulos and M. N. Vrahatis. Recent approaches to global optimization problems through particle swarm optimization. *Natural Computing*, 1(2-3):235–306, 2002. ISSN 1567-7818. doi: 10.1023/A:1016568309421.

M. A. Petrovici, J. Bill, I. Bytschok, J. Schemmel, and K. Meier. Stochastic inference with deterministic spiking neurons. *arXiv preprint arXiv:1311.3211*, 2013.

T. Pfeil, T. C. Potjans, S. Schrader, W. Potjans, J. Schemmel, M. Diesmann, and K. Meier. Is a 4-bit synaptic weight resolution enough? - constraints on enabling spike-timing dependent plasticity in neuromorphic hardware. *Frontiers in Neuroscience*, 6 (90), 2012. ISSN 1662-453X. doi: 10.3389/fnins.2012.00090.

T. Pfeil, A. Grübl, S. Jeltsch, E. Müller, P. Müller, M. A. Petrovici, M. Schmuker, D. Brüderle, J. Schemmel, and K. Meier. Six networks on a universal neuromorphic computing substrate. *Frontiers in Neuroscience*, 7:11, 2013. ISSN 1662-453X. doi: 10.3389/fnins.2013.00011.

S. Philipp. *Design and Implementation of a Multi-Class Network Architecture for Hardware Neural Networks*. PhD thesis, Universität Heidelberg, 2008.

F. Piedad and M. Hawkins. *High Availability: Design, Techniques, and Processes*. Enterprise computing series. Prentice Hall PTR, 2001. ISBN 9780130962881.

T. C. Potjans and M. Diesmann. The cell-type specific connectivity of the local cortical network explains prominent features of neuronal activity. *arXiv preprint arXiv:1106.5678*, 2011.

C. Pozzorini, R. Naud, S. Mensi, and W. Gerstner. Temporal whitening by power-law adaptation in neocortical neurons. *Nature neuroscience*, 16(7):942–948, 2013.

D. Probst. Analysis of the liquid computing paradigm on a neuromorphic hardware system. Bachelor thesis (English), University of Heidelberg, HD-KIP 11-47, 2011.

Y. Prut, E. Vaadia, H. Bergman, I. Haalman, H. Slovin, and M. Abeles. Spatiotemporal structure of cortical activity: Properties and behavioral relevance. *J. Neurophysiol*, 79:2857–2874, 1998.

Python. *The Python Programming Language*. Python Software Foundation Website, 2014. URL `http://www.python.org`.

R. Ramey. *Boost Serialization Version 1.49.0 Website*, 2004. URL `http://www.boost.org/doc/libs/1_49_0/libs/serialization/doc/`.

K. Robbins and S. Robbins. *UNIX Systems Programming: Communication, Concurrency, and Threads*. Prentice Hall PTR, 2003. ISBN 9780130424112.

I. H. Robertson and J. M. J. Murre. Rehabilitation of brain damage: Brain plasticity and principles of guided recovery. *Psychological Bulletin*, 125(5):544–575, Sep 1999. doi: 10.1037/0033-2909.125.5.544.

J. Schemmel. Personal communication, January 2014a.

J. Schemmel. Personal communication, April 2014b.

J. Schemmel, 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)*. IEEE Press, 2006.

J. Schemmel, 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)*, pages 3367–3370. IEEE Press, 2007.

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

J. Schemmel, 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)*, pages 1947–1950, 2010.

M. Schilling. A highly efficient transport layer for the connection of neuromorphic hardware systems. Diploma thesis, Universität Heidelberg, 2010. HD-KIP-10-09.

S. Scholze, H. Eisenreich, S. Höppner, G. Ellguth, S. Henker, M. Ander, S. Hänzsche, J. Partzsch, C. Mayr, and R. Schüffny. A 32 GBit/s communication SoC for a waferscale neuromorphic system. *Integration, the VLSI Journal*, 2011a. doi: 10.1016/j.vlsi.2011.05.003. in press.

S. Scholze, S. Schiefer, J. Partzsch, S. Hartmann, C. G. Mayr, S. Höppner, H. Eisenreich, S. Henker, B. Vogginger, and R. Schüffny. VLSI implementation of a 2.8GEvent/s packet based AER interface with routing and event sorting functionality. *Frontiers in Neuromorphic Engineering*, 5(117):1–13, 2011b.

M. Schulz. The end of the road for silicon? *Nature*, 399(6738):729–730, 1999.

M.-O. Schwartz. *PhD thesis*, University of Heidelberg, in preparation, 2012.

J. Seward. *bzip2 Version 1.0.6 Website*, 2014. URL `http://www.bzip.org`.

H. Shan and J. Singh. A comparison of mpi, shmem and cache-coherent shared address space programming models on a tightly-coupled multiprocessors. *International Journal of Parallel Programming*, 29(3):283–318, 2001. ISSN 0885-7458. doi: 10.1023/A:1011120120698.

S. Shipp. Structure and function of the cerebral cortex. *Current Biology*, 17(12):R443–9, 2007. ISSN 0960-9822. URL `http://www.cell.com/current-biology/retrieve/pii/S0960982207011487`.

J. Siek. *Boost Graph Dijkstra's Algorithm Implementation 1.49.0 Website*, 2001. URL `http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/dijkstra_shortest_paths.html`.

J. Siek, L.-Q. Lee, and A. Lumsdaine. *Boost Graph Library Version 1.49.0 Website*, 2001. URL `http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/index.html`.

S. Skiena. *The Algorithm Design Manual.* Springer Verlag, 2nd edition, 2008. ISBN 978-1848000698.

T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, and L. Vicisano. Pgm reliable transport protocol specification. RFC 3208 (Experimental), Dec. 2001. URL `http://www.ietf.org/rfc/rfc3208.txt`.

A. Srowig, J.-P. Loock, K. Meier, J. Schemmel, H. Eisenreich, G. Ellguth, and R. Schüffny. Analog floating gate memory in a 0.18 μm single-poly CMOS process. *FACETS internal documentation*, 2007.

C. H. Stapper, F. M. Armstrong, and K. Saji. Integrated circuit yield statistics. *Proceedings of the IEEE*, 71(4):453–470, 1983.

I. H. Stevenson and K. P. Kording. How advances in neural recording affect data analysis. *Nature neuroscience*, 14(2):139–142, 2011.

P. Strata and R. Harvey. Dale's principle. *Brain Research Bulletin*, 50(5–6):349–350, 1999. ISSN 0361-9230. doi: 10.1016/S0361-9230(99)00100-8.

H. Sutter. The joy of pimpls, 1998. URL `http://www.gotw.ca/publications/mill05.htm`.

B. Tran. Demonstrationsexperimente auf neuromorpher Hardware. Bachelor thesis (german), Universität Heidelberg, 2013. HD-KIP 13-55.

M. Tsodyks 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, Jan. 1997.

UHEI and TUD. Brainscales year 3 annual EU report. BrainScaleS Annual Report EU Report, 2014.

M. van Smoorenburg. *liblockfile: NFS-safe locking library Website*, 2012. URL `http://www.t2-project.org/packages/liblockfile.html`.

B. Vogginer. Personal communication, October 2013.

B. Vogginer. Personal communication, March 2014a.

B. Vogginer. Personal communication, Februrary 2014b.

B. Vogginger. Testing the operation workflow of a neuromorphic hardware system with a functionally accurate model. Diploma thesis, Universität Heidelberg, 2010. HD-KIP-10-12.

D. A. Watt and W. Findlay. *Programming language design concepts.* John Wiley and Sons, 2004. ISBN 978-0-470-85320-7.

White House. 21st century grand challenges, 2014. URL `http://www.whitehouse.gov/administration/eop/ostp/grand-challenges`.

R. Yakovenko and M. Baas. *pygccxml and pyplusplus sourceforge project site*, 2013. URL `http://sourceforge.net/projects/pygccxml/`.

X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. *Proc. VLDB Endow.*, 4(4):231–242, Jan. 2011. ISSN 2150-8097.

H. Zoglauer. Entwicklung und Testergebnisse eines Prototypensystems für die Wafer-Scale-Integration. Diploma thesis (german), Universität Heidelberg, 2009. HD-KIP 09-28.

Acknowledgments

I want to thank all persons who supported this work, especially:

Prof. Dr. Karlheinz Meier and Dr. Johannes Schemmel for their mentoring and the opportunity to carry out this thesis.

Prof. Dr. Bernd Jähne for assessing this thesis.

My deepest gratitude to my beloved bachelors and internees: Johann, Sebastian B., Nathan and Dimitri.

Thanks to Christoph, Tom, Paul, Simon, Matthias, Johann, Sebastian B and Eric for proof reading as well as their valuable input. Special thanks to Sebastian S., who fought his way through most chapters.

My office colleagues, Christoph and Tom for being most awesome.

My fellow fighter Matthias (awesome++).

Mihai for his valuable JUNE insights.

All other Visionaries for being supportive, great colleagues and plain awesome.

Bärbel for everything.

My family.