Dissertation

submitted to the Combined Faculty of

Natural Sciences and Mathematics

of Heidelberg University, Germany

for the degree of

Doctor of Natural Sciences

Put forward by

Christian Mauch

born in: Stuttgart

Oral examination: 14.12.2021

Operating Accelerated Neuromorphic Hardware
– A Scalable and Sustainable Approach

Referees:

Dr. habil. Johannes Schemmel (Heidelberg University)

Prof. Dr. Hans-Christian Schultz-Coulon (Heidelberg University)

Abstract

Accelerated mixed-signal neuromorphic hardware presents a promising approach to overcome run time and scalability issues of software-based neural network simulations. It accomplishes this by physical emulation of the neuronal dynamics via specialized analog circuitry instead of numerical calculations. However, facilitating the advantage of such highly custom hardware with a similar convenience as conventional simulators poses various challenges. This thesis addresses these in two ways:
First a multi-layered software architecture developed for the second-generation BrainScaleS neuromorphic systems is presented. Well-defined interfaces allow utilization of the hardware in different stages of development with the appropriate level of abstraction. The upper layers provide an interface to efficiently describe neuroscience experiments and handle automated translation of population-based spiking neural network graphs to valid hardware configurations and experiment flow programs. Suitable run time performance and scalability of the software are verified by extensive measurements while usability is demonstrated via an SNN-based Sudoku solver. The second part covers the challenges of supplying novel compute hardware as a research platform to the neuroscience community. A convenient and robust multi-user access is facilitated via customization of the prevalent SLURM resource scheduler to the requirements of neuromorphic experiment workflows. Finally, a monitoring infrastructure vital for system commissioning and experiment reproducibility is established.

Zusammenfassung

Beschleunigte neuromorphe digital-analoge Hardware ist ein vielversprechender Ansatz zur Überwindung der Laufzeit- und Skalierbarkeitsproblemen von softwarebasierten Simulationen neuronaler Netze. Erreicht wird dies durch die physikalische Emulation der neuronalen Dynamik mittels spezialisierter analoger Schaltungen anstelle von numerischen Berechnungen. Es ist jedoch eine Herausforderung, die Vorteile solcher hochgradig spezialisierter Hardware mit einem ähnlichen Komfort wie bei herkömmlichen Simulatoren zu verbinden. In dieser Arbeit werden diese auf zwei Arten angegangen:
Im ersten Teil wird eine mehrschichtige Softwarearchitektur für die zweite Generation der BrainScaleS neuromorphen Systeme vorgestellt. Gut definierte Schnittstellen ermöglichen die Nutzung der Hardware in verschiedenen Entwicklungsstadien mit der entsprechenden Abstraktionsebene. Die oberen Schichten bieten eine Schnittstelle zur effizienten Beschreibung neurowissenschaftlicher Experimente und zur automatischen Übersetzung populationsbasierter Graphen pulsbasierter neuronaler Netze in valide Hardwarekonfigurationen und Experimentablaufprogramme. Die geeignete Laufzeitleistung und Skalierbarkeit der Software wird durch umfangreiche Messungen verifiziert, während die Benutzerfreundlichkeit anhand eines Sudoku-Lösers auf Basis pulsbasierter Netze demonstriert wird. Der zweite Teil befasst sich mit den Herausforderungen, die mit der Bereitstellung dieser neuartigen Hardware als Forschungsplattform für die neurowissenschaftliche Gemeinschaft verbunden sind. Ein handlicher und robuster Mehrbenutzerzugang wird durch die Anpassung des weit verbreiteten SLURM-Ressourcenmanagers an die Anforderungen neuromorpher Experimentabläufe ermöglicht. Abschließend wird eine Überwachungsinfrastruktur eingerichtet, die für die Inbetriebnahme des Systems und die Reproduzierbarkeit der Experimente unerlässlich ist.

Contents

Chapter 1

Introduction

The nervous system, and in particular the brain, is the evolutionary result for the need of animals to interact with their environment. It comprises a myriad of interconnected nerve cells also called neurons, from which astonishing cognitive capabilities emerge. The field of computational neuroscience strives to develop models that explain and replicate the functionality of these systems. Spiking Neural Networks (SNNs) are one of the basic models to describe the brains structure and investigate its computational capabilities. The complexity of such models and systems often necessitates numerical simulations. Traditionally these simulations are performed in software on conventional compute architectures [Diesmann et al. 2002; Hines et al. 2003; EPFL et al. 2008]. However, they become compute and time intensive, especially with scaled up network sizes. For example, a human-scale brain simulation of the cerebellum comprising $6.8 \cdot 10^{10}$ neurons and $5.4 \cdot 10^{12}$ synapses was conducted utilizing all 82 944 compute nodes of the K supercomputer [Yamazaki et al. 2021]. One minute of non-functional network activity took over 10 hours of wall-clock simulation time, a slowdown by a factor of about 600. Moreover, the K supercomputer has an average power consumption of about 12 MW [Yamamoto et al. 2014]. Reasons for these huge compute requirements are, on the one hand, that solving the differential equations describing the neuron dynamics is numerically expensive. On the other hand, distribution of the spike events between the hundreds of compute nodes introduces a significant communication overhead [Zenke et al. 2014].

These challenges were met with different approaches, from efficiency improvements in traditional numerics to optimization in event distribution [Pronold et al. 2021a]. Another approach is the development of dedicated hardware to physically emulate neural networks, which is called neuromorphic computing and was first coined by Caver Mead [Mead 1989; Mead 1990]. The term has been applied more broadly in recent years, to any specialized hardware that mimics structure and behavior of neural systems. Various neuromorphic computing devices were

developed over the years, from FPGA-based accelerators to full-custom digital or mixed-signal chips [Indiveri et al. 2011; Furber et al. 2012; Hu et al. 2014; Davies et al. 2018].

One such approach, the BrainScaleS (BSS) architecture [Schemmel et al. 2010; Schemmel et al. 2020], is the focus of this thesis. It implements a physical representation of the neuron dynamics in the form of analog circuits, i.e., no numerical computations are performed. However, the distribution of spike events throughout the network is performed via a digital bus system, thereby making it a mixed-signal system. The chosen circuit dimensions allow the neuromorphic substrate to operate with speed-up of 10^3 to 10^5 compared to real-time. This high acceleration factor enables the investigation of long term neural development experiments previously not possible as biological years could be emulated in mere hours wall-clock time. These advantages however come with trade-offs to generality and precision. Regarding generality, the neuron model and its parameter range are fixed to the chosen design whereas in software simulation these can be chosen arbitrarily. Limited precision is caused by variations in the analog substrate as well as constraints in parameter resolution, mostly only a few bits, compared to floating-point precision in software. Furthermore, the physical emulation of the neuronal dynamics entails a continuous operation, meaning it cannot be halted and continued later on. Therefore, one has to cope with these trade-offs during operation.

Neuromorphic hardware not only provides means to expedite neuroscience research but could usher in new computing paradigms. The ever-increasing thirst of human society for compute power could be quenched due to Moore's law [Moore 1965]. It is an empirical observation that "predicts" a doubling of integration density in microcircuits, or effectively in compute power, every two years. However, over the last decade this evolution has encountered a number of roadblocks [Theis et al. 2017] which lead to the pursue of novel non-von-Neumann architectures [Neumann 1945]. The renascence of Artificial Neuronal Networks (ANNs) in Machine Learning (ML) at the beginning of the 2010s through deep learning [Krizhevsky et al. 2012; LeCun et al. 2015; Tan et al. 2018] is, at least algorithmically, a step in the direction of such a paradigm shift. Various tasks formerly thought of as requiring "human level intelligence" have been solved with this approach [Silver et al. 2016; OpenAI et al. 2019]. Their breakthrough was made possible by performant specialized algorithms for matrix multiplication utilizing Graphics Processing Units (GPUs). However, deep learning with ANNs still has a huge computational cost leading to development to even more specialized hardware like Google's Tensor Processing Unit (TPU) [Jouppi et al. 2017].

Another big role for the success of ANNs in ML was played by various software frameworks like TensorFlow [Abadi et al. 2015] or PyTorch [Paszke et al. 2019a]. They allow researchers to formulate experiments in high-level interfaces which

abstract away the underlying hardware operations. Thus, one can take advantage of GPUs or TPUs without requiring a deep knowledge of the technical background. Likewise, the facilitation of specialized neuromorphic hardware also requires specialized software to fully exploit its advantages [Rhodes et al. 2018; Lin et al. 2018].

The first main goal of this thesis is the design and development of a sophisticated software architecture for the BrainScaleS Generation 2 (BSS-2) neuromorphic hardware system. It aims to support a similar high-level workflow that abstracts away as many system specifics as possible so that researchers can focus on their experts and not on hardware particularities. To this end, several aspects of system operation need to be covered such as handling of communication, abstraction of hardware components, managing runtime control and defining high-level experiment descriptions. All these aspects need to cope with the particular characteristics of the analog hardware. For example, the high acceleration factor and continuous network emulation set high demands on throughput as well as latency. Similarly, scalability and sustainability of the software are important to handle changes in hardware makeup and evolution to large-scale systems. This is especially true in an academic environment where the stay of researchers is shorter than the lifetime of the hard- and software.

The main purpose of the BSS neuromorphic hardware is to facilitate computational neuroscience experiments. To this end, the Electronic Visions(s)[1] group as part of the Human Brain Project [Markram 2012] strives to provide access to theses systems as computing platforms for the scientific community. The second major goal of this thesis is to ensure that conducting science with this platform is robust, convenient and, most notably, reproducible. Typically large scale neuroscience simulations, like most other large scale simulations, are run in super-computing centers also called High Performance Computings (HPCs). There multi-user access to the compute nodes is managed via resource schedulers. Likewise, access to the neuromorphic hardware systems is supported by a resource scheduler customized to the particular characteristics of the hardware. Furthermore, reproducibility and robustness of experiments are facilitated by an extensive monitoring infrastructure.

Thesis Outline

The thesis is structured as follows:

Chapter 2 provides an introductory insight into the most relevant topics of this thesis. First, an introduction to biological spiking neural networks is given in section 2.1. Then, the concept of neuromorphic computing, especially in

[1] https://www.kip.uni-heidelberg.de/vision/

the context of computational neurosciences, is discussed further in section 2.2. Subsequently, sections 2.2.3 and 2.2.4 give an overview of the neuromorphic systems covered in this thesis, namely BrainScaleS Generation 1 (BSS-1) and BSS-2. Furthermore, general concepts of software engineering and their relevance to science, in particular the collaborative workflow practiced in the Electronic Vision(s) group, are discussed in section 2.3.

Building on this, chapter 3 presents the developed software framework facilitating utilization of the BSS-2 hardware. It discusses the particular requirements that accelerated neuromorphic hardware and neuroscience experiments have on software design and the resulting decisions regarding the chosen software architecture. A short overview of the architecture is given as an orientation which is subsequently followed up with more detailed explanations in sections 3.2 to 3.5. Chapter 3 is then concluded by demonstrating the viability and usability of the software by extensive performance measurements (section 3.6) and an exemplary neural network experiment (section 3.7).

Chapter 4 discusses the efforts regarding platform operation. First, the customizations to the Simple Linux Utility for Resource Management (Slurm) resource scheduler are described in section 4.1. This includes an analysis of the hardware utilization via scheduler usage statistics. Secondly the extensive monitoring infrastructure is described in section 4.2, which facilitates robust and reproducible platform operations.

Chapter 5 summarizes and discusses the achievements and limitations of the conducted thesis. It is concluded by an outlook regarding future of the systems, in particular their scale-up.

Chapter 2

Background

2.1 Spiking Neural Networks

Spiking Neural Networks (SNNs) are used to describe and investigate the dynamics of the neurons and their connections found in the brain. Figure 2.1 illustrates the principle of spiking neurons and their synaptic interconnection, going from biological neurons to a simple abstract model. Action potentials, also called spikes, travel along the axon of the pre-synaptic neurons and induce a change in membrane potential of the post-synaptic neuron. If the change was positive the connection or synapse is called excitatory and in case of negative change it is called inhibitory. The post-synaptic neuron typically has hundreds to thousands of pre-synaptic connections which it accumulates. With enough input events in quick succession it can emit a spike itself.

This of course is a simplification of all the mechanisms present in real biological neurons. There are several models which describe these mechanisms with varying detail, with the Hodgkin–Huxley model [Hodgkin et al. 1952] being one of the most influential. It is relatively complex computational wise as it incorporates the dynamics of sodium and potassium ion-channels to describe the neuron behaviour. However, the full detail of these dynamics are not always necessary when investigating the behaviour of large neural networks. Therefore, in many simulations of large networks simpler neuron models are utilized, two of which are also implemented in the BSS hardware and explained in the following.

2.1.1 The Leaky Integrate-and-Fire Model

The LIF neuron model is a simple yet prevalent model as its dynamics are often sufficient to describe functional networks [Stein 1967]. Its dynamics are illustrated

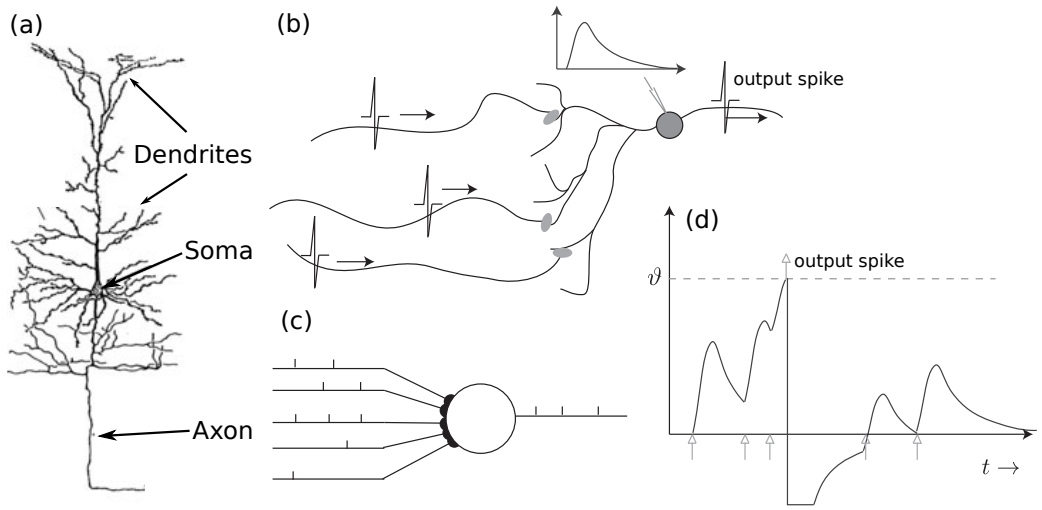


Figure 2.1: Abstraction of the functionality of spiking neurons. (a) Illustration of a biological neuron. (b) Functionality of spike input accumulation. Axons of pre-synaptic neurons are connected to dendrites of the accumulating post-synaptic neuron via synapses (gray ellipses). An incoming spike event leads to rises of the neuron membrane potential, which then decays back to the resting potential (see inlet). If enough excitatory input spikes arrive in quick succession the accumulating neuron fires, i.e., emits a spike itself. (c) Abstract representation of a neuron receiving multiple incoming spike sequences which result in an outgoing spike sequence, thereby performing non-linear information processing. (d) Neuron membrane dynamics triggered by spike input according to the LIF neuron model. With enough input spikes the membrane potential reaches the threshold ϑ. This results in quick reset of the membrane potential and emission of an outgoing spike. Modified from Grüning et al. 2014.

in fig. 2.1(d) and described by the following equation.

$$C_{\mathrm{m}} \frac{\mathrm{d}V_{\mathrm{m}}}{\mathrm{d}t} = -g_{\mathrm{leak}}(V_{\mathrm{m}} - V_{\mathrm{leak}}) + I \tag{2.1}$$

Without any synaptic input I the membrane potential V_{m} rests at the leakage potential V_{leak}. An incoming spike event elicits an input current I which charges the membrane capacitance C_{m}. The resulting potential then decays back to V_{leak} with the time constant $\tau_{\mathrm{m}} = C_{\mathrm{m}}/g_{\mathrm{leak}}$, where g_{leak} is the leakage conductance. Enough excitatory input events in short succession bring V_{m} over the threshold ϑ, which triggers an outgoing spike. Simultaneously, the membrane potential is brought to the reset potential V_{reset} for a duration defined by a refractory time constant τ_{ref}.

2.1.2 The Adaptive Exponential Integrate-and-Fire Model

The Adaptive Exponential Integrate-and-Fire (AdEx) neuron model [Brette et al. 2005] is an extension to the LIF model that remedies some of its limitations [Markram et al. 2004]. On the one hand, an adaptive term is added that provides the neuron with some capability to remember its past states prior to a reset. On the other hand, an exponential term is added that promotes spiking the closer the membrane potential is already to the threshold. These additional terms allow recreation of complex fire patterns found in biology that the LIF model is not capable of producing. Extending Equation (2.1) with these terms results in:

$$C_{\mathrm{m}}\frac{\mathrm{d}V_{\mathrm{m}}}{\mathrm{d}t} = -g_{\mathrm{leak}}(V_{\mathrm{m}} - V_{\mathrm{leak}}) + g_{\mathrm{leak}}\Delta_{\mathrm{T}} \cdot \exp(\frac{V_{\mathrm{m}} - V_{\mathrm{T}}}{\Delta_{\mathrm{T}}}) - w + I \tag{2.2}$$

$$\tau_w\frac{\mathrm{d}w}{\mathrm{d}t} = a(V_{\mathrm{m}} - V_{\mathrm{leak}}) - w \tag{2.3}$$

with w being the adaptation current which is parametrized by a time constant τ_w and an adaptation constant a. Once a spike is emitted w is increased by an amount b. Regarding the exponential term, Δ_{T} defines the scaling factor and V_{T} the exponential onset. V_{T} is typically set closely below the firing threshold ϑ. Setting w and Δ_{T} to 0 results again in the LIF equation (eq. (2.1)).

2.1.3 Multi-Compartment Neuron Models

Until now neurons had no spacial structure, i.e, were regarded as points. However, as seen in fig. 2.1a, real biological neurons have a spacial extension that also affects their behaviour. To model such behaviour the concept of compartmental models was introduced [Gerstner et al. 2002] in which multiple point neurons, i.e., the compartments, are interconnected to form larger structures. When using the LIF model to describe the individual compartments, the dynamics of one compartment i are given by the equation:

$$C_i\frac{\mathrm{d}V_i}{\mathrm{d}t} = -g_{\mathrm{mem},i}V_i + \sum_k g_{\mathrm{spine},ik}(V_k - V_i) + I_i \tag{2.4}$$

where $g_{\mathrm{mem},i}$ describes the conductance to the membrane of the individual compartment and $g_{\mathrm{spine},ik}$ the conductance between connecting compartments.

2.1.4 Synaptic Input and Plasticity

In the previous eqs. (2.1) and (2.2) the synaptic input was abstracted as an arbitrary input current. Shape, amplitude and duration of this input current vary depending

on the synapse type and its parameters. Two prominent synapse types are Current Based (CUBA) synapses and Conductance Based (COBA) synapses.

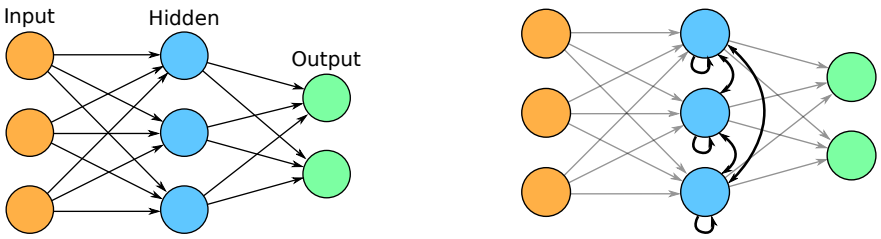
The primary parameter that defines the strength of the interconnectivity between two neurons is the synaptic weight. The dynamics and therefore functionality of neural networks greatly depend on these weights. One of the major features of neural networks is their ability to learn which is among others facilitated by modification of this weight. The capability to modify the synaptic input strength is called synaptic plasticity and can occur on differing time scales depending on the type of plasticity model or rule. For example Short-term Plasticity (STP) as the name suggest happens on short time scales, i.e., milliseconds to seconds [Tsodyks et al. 1997]. It reduces or increases the strength of the synaptic input depending on how active the neuron is. This for example can represent the depletion of neurotransmitters in biological neurons. One of the first concepts in synaptic learning, i.e, long-lasting changes to synaptic weights, is the Hebbian theory [Hebb 1949] which is colloquially summarized as "Cells that fire together wire together". More explicitly the synaptic weight is strengthened or weakened depending on the temporal correlation between arrival of pre-synaptic spikes and emission of a post-synaptic spikes. Spike Timing Dependent Plasticity (STDP) [Markram et al. 1997] is a widespread plasticity rules that applies the Hebbian theory.

2.1.5 Network Topologies

The structure of neural networks, i.e., the interconnection between neurons, is crucial to their functionality as much as or even more so than the dynamics of the individual neurons. Neural networks are typically illustrated as graphs where the vertices represent neurons and edges the synaptic connections. The most common example of neural network structures is the feed forward network, illustrated in fig. 2.2a. It is a simple example ANN structure to describe deep learning in ML, with a typical application being image classification tasks. The input signals are propagated through one or more hidden layers resulting in a desired activity of the output layer, this typically being only one active neuron. If the network graph is cyclic, i.e., there are connections that feed back activity, it becomes a recurrent network (see fig. 2.2b). This can provide the network with a quasi memory which is for example utilized for time sequence data such as speech.

2.2 Neuromorphic Computing Platforms

Modern neuroscience research that aims to decipher the brain is highly interdisciplinary and cost intensive leading to endeavors like the BRAIN Initiative [Insel



(a) Feed-Forward Network (b) Recurrent Network

Figure 2.2: Common neural network structures. (a) Separate layers are connected in one direction, consequently signals are also propagated and processed only in one direction, hence the name feed-forward. (b) Additional recurrent connections can function as quasi memory, e.g., for time series data such as speech.

et al. 2013] or the Human Brain Project [Markram 2012]. They aim to consolidate efforts by sharing access to large-scale experiment platforms as is similar for large-scale physics projects in astronomy or high energy physics. The following section will give a short overview of neuromorphic computing platforms, in particular the BSS systems. First however, the typical workflow for large-scale neural network research is discussed. Especially the neuroscience software tools like simulator frameworks are introduced as they serve as connection points to the neuromorphic platforms.

2.2.1 Computational Neuroscience Workflow

Neuroscience Software

In computational neuroscience, like in most other computation centric research fields, shared software tools emerge to consolidate efforts. One category of such tools are dedicated spiking neural network simulators, of which a few are shortly presented highlighting their different main applications. The *NEST* simulator [Diesmann et al. 2002] is specialized for large-scale point neuron simulations with focus on scaling on HPC clusters through intrinsic Message Passing Interface (MPI) support. It tries in particular to address the bandwidth and memory bottlenecks of event distribution over compute nodes [Pronold et al. 2021a; Pronold et al. 2021b]. The *NEURON* [Hines et al. 2003] and *Arbor* [Akar et al. 2019] simulators focus on multi compartment models. *Brian2* [Stimberg et al. 2019] focuses flexibility, e.g., with convenient description of custom neuron models. They are several approaches to accelerate these CPU-based simulators with GPU-based back-ends like *GeNN* [Yavuz et al. 2016] or *CARLsim*[Chou et al. 2018].

Kulkarni et al. 2021 benchmarks several of those simulators and reinforces that there is no one-fits-all simulator but most have their particular use case. However,

it is not always clear which simulator is best applicable in a given situation. Common back-end agnostic interfaces were developed to ease switching between simulator frameworks and facilitating comparability. The most prominent being *PyNN* [Davison et al. 2009a] and *Nengo* [Bekolay et al. 2014].

In recent years many ML learning inspired approaches where adopted for SNN, for example, the back-propagation trough time algorithm [Werbos 1990] or LSTM cells [Bellec et al. 2018]. Consequently, ML frameworks like *TensorFlow* [Abadi et al. 2015] or *PyTorch* [Paszke et al. 2019a] found their way into the neuroscience community, with the latter being more prominent [He 2019]. Vice versa, the sparsity in information transport of SNN and therefore potential energy efficiency, peaked interest in the ML community. Resulting from this are SNN extensions to *pytorch* like *BindsNET* which utilize similar population based interfaces of SNN simulators.

High Performance Computing

Large-scale SNN neuroscience experiments typically demand high computation capabilities, not only for the simulation of the networks themselves but also for pre- and post-processing, e.g., of large spike data. As high computation capabilities are needed in a plethora of research fields, collaboratively funded High Performance Computing (HPC) clusters like the Jülich Supercomputing Centre are operated. Challenges for construction and operations of super-compute clusters are for example setup of a suitable connection topology of the compute nodes to provide balanced data exchange. To facilitate the distributed compute resources inter process communication is needed, for example handled via MPI. Furthermore, sharing of compute resources for multiple users needs to be ensured, which is solved by resource schedulers.

In typical HPC environments simulations take several hours up to weeks, which consequently leads to long waiting times. Therefore, researchers first prototype their experiment scripts on a small scale allowing for relatively fast feedback-loops. Only then they submit long running jobs to the cluster.

In a similar vein, neuromorphic computing platforms also need to manage shared access for researchers. Chapter 4 presents how this is facilitated for the BSS platform.

2.2.2 Neuromorphic Approaches

The field of neuromorphic computing emerged from the desire to overcome the limitations of conventional simulations of SNNs and the constraints of the von-Neumann architecture. It saw a strong growth in the 2010s, similar to the rise of ANNs in ML, resulting in different technological approaches [Schuman et al.

2017]. They range from FPGAs-based emulations [Wang et al. 2018] to custom digital and mixed-signal ASICs [Indiveri et al. 2011; Benjamin et al. 2014; Merolla et al. 2014] or even new materials like memristors [Hu et al. 2014; Li et al. 2018] and photonics [Feldmann et al. 2019].

As this thesis focuses on the facilitation of neuromorphic computing platforms an overview is given of systems which are already available to the community, including the necessary software support. For a broader overview of neuromorphic devices please refer to Schuman et al. 2017; Thakur et al. 2018; Li et al. 2018; Young et al. 2019.

SpiNNaker

The SpiNNaker neuromorphic platform developed at the University of Manchester mainly tackles the issue of spike distribution between computation cores of conventional numerical simulators [Furber et al. 2012]. It does this by interconnecting a multitude of general purpose Advanced RISC Machines (ARM) compute cores with a routing mesh to efficiently handle spike transfer between the cores. Each core is connected to its top, right and diagonally top right neighbor resulting in a two-dimensional torus mesh for the complete network topology. A single SpiNNaker chip utilizes 17 ARM cores where 48 of those chips are combined on large circuit boards which again are interconnected to build up the full system. It thus incorporates over 10^6 of these ARM processors which are able to simulate over 10^9 neurons with 10 000 synapses each. As the performed computation is general purpose, arbitrary neuron models and plasticity rules are supported. Real-time simulation speed is possible however slowdown occurs depending on the simulated model. Regarding software support, SpiNNaker provides a *PyNN* back-end as a experiment interface [Rhodes et al. 2018].

Loihi

The Loihi neuromorphic chip developed by Intel implements specialized fully digital circuits which calculate the neuron dynamics in discrete time steps [Davies et al. 2018]. One Loihi chip contains 128 neuromorphic cores which implement up to 130 000 LIF neurons with 1000 CUBA synapses each. These neurons can be combined to form larger logical neurons. Synaptic plasticity is facilitated by a learning framework which allows to define custom update rules from a set of given functions and observables to operate on. The cores are connected to a 2D bus system allowing arbitrary connections within a chip. However, fan-in and fan-out, i.e., maximum number of connections to other neurons, of these cores is limited, thus neuron placement is constrained and needs to be handled. Each core runs asynchronously within a time step however the spike events between

cores need to be synchronised effectively constraining the overall performance to the slowest core. 768 of those chips are combined in a rack-mount which thus has the capacity to simulate over 10^8 neurons [Intel 2020]. Experiment description for the Loihi system is supported via multiple interfaces such as a custom population based Application Programming Interface (API) [Lin et al. 2018] similar to *PyNN*, a *Nengo*[1] back-end and a *Tensorflow* like API [Rueckauer et al. 2021]. Unfortunately the work behind Loihi and conducted research on it is shrouded by non disclosure agreements.

BrainScaleS

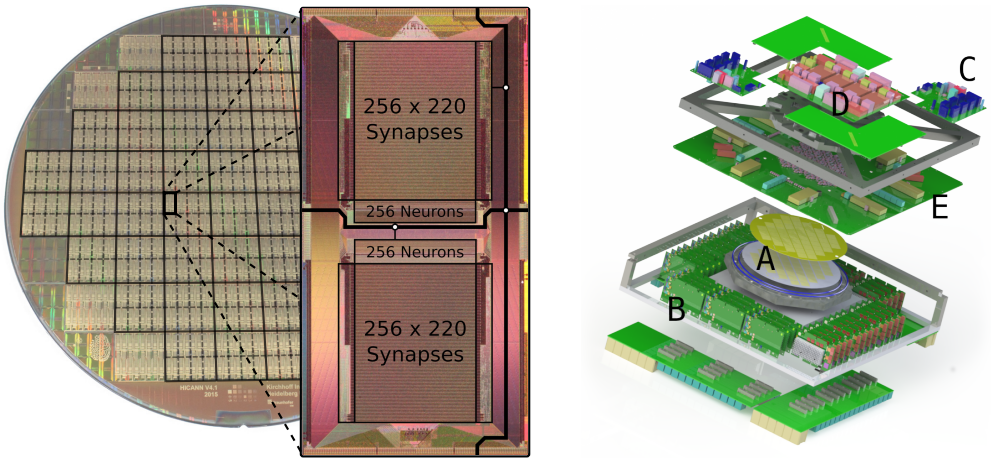
The BrainScaleS (BSS) neuromorphic hardware developed by the Electronic Vision(s) group at the University of Heidelberg [Schemmel et al. 2010; Schemmel et al. 2020] does not simulate neuron dynamics numerically but emulate them in physical representations of the underling neuron model. Multiple different chip generations where developed of which a short overview is given. The predecessor chip *Spikey* marks the first chip generation developed by the group [Schemmel et al. 2006]. Its main application was as portable single chip system however also a multi chip system was designed [Philipp et al. 2007]. Large-scale networks were then more specifically targeted with the BSS systems. As facilitation of BSS-1 and BSS-2 are central elements of this thesis they are described in more detail in section 2.2.3 and section 2.2.4 respectively.

2.2.3 BrainScaleS-1

The BSS-1 hardware is the first generation of systems developed in the Electronic Vision(s) group with the explicit goal to provide large-scale accelerated SNN emulation. It utilizes wafer-scale integration to implement up to 196 608 AdEx neurons (section 2.1.2) and over 40 million COBA synapses (section 2.1.4) on a single wafer module, illustrated in fig. 2.3a. Each wafer consist of 8×48=384 so-called High-Input Count Analog Neuronal Network (HICANN) chips that implement 512 neurons each. In contrast to conventional chip manufacturing, single chip dies are not cut out but kept intact to be interconnected via an additional post-processing layer. This post-processing also provides connection pads for, e.g. supply voltages, analog readout or communication. Multiple neuron circuits can be combined, in groups of up to 64, by directly connecting their membranes to form larger neurons with increases synaptic fan-in. It is to note that this functionally does not constitute multi compartment capabilities as described in section 2.1.3. The presented overview of BSS-1 is constrained to information

[1] https://www.nengo.ai/nengo-loihi/ 2021-09-16

which is primarily relevant for chapter 4 and as background for BSS-2. A more detailed description of the system is given in Schemmel et al. 2010.



(a) Silicon Wafer (b) Wafer Module

Figure 2.3: BSS-1 neuromorphic wafer module. (a): Post-processed silicon wafer with zoom-in to a single HICANN chip. One wafer comprises 8×48=384 of these chips, where groups of eight HICANNs are called a *reticle* (dashed smaller grid on wafer). Each chip implements 512 neuron circuits with 220 synapses each, where multiple circuities can be combined to form larger neurons. At maximum granularity, 196 608 neurons with 220 synapses each are realized or at minimum granularity 3072 neurons with 14 080 synapses each. Spike transmission is handled via a digital bus system that spans horizontally and vertically over the wafer, with additional connections on- and off-wafer. (b): Exploded view of a wafer module with its various components (dimensions: 50 cm×50 cm×15 cm). A: silicon neuromorphic wafer B: 48 communication FPGAs C: analog readout module D: supply voltage distribution E: main PCB interconnecting all components. Modified from Müller et al. 2020b.

Neuron dynamics

The AdEx neuron dynamics are physically time-continuously emulated by analog circuitry implemented in 180 nm Complementary Metal-Oxide-Semiconductor (CMOS) technology. It operates at a speedup of 10^3 to 10^5 compared to biological real-time, which is a consequence of the circuit dimensions deliberately chosen by the hardware designers. All parameters of the implemented AdEx model are configurable, to a certain degree, resulting in the range for the speedup. These configurable parameters need to be provided in analog, i.e., as voltages or currents.

To this end, an analog neuron parameter storage is implemented by floating gates cells, which take a digital value as input. One of their downsides is a high trial-to-trial variation, i.e., writing the same digital value multiple times results in varying analog values. This can be somewhat circumvented utilizing the inherent longevity of the written values for consecutive experiments.

Spike Transmission

Spike events are transmitted via a digital bus system that stretches over the wafer in a mesh like structure, thus making the BSS-1 architecture mixed-signal. When a neuron spikes it emits a digital packet containing its address. This packet is either directly sent off chip or propagated over the wafer to all target neurons via a pre-determined configurable routing. Similarly, external events can be inserted onto the chip. Events are duplicated at mesh nodes if necessary. When a packet passes a target chip, the synapse array then checks the source address of events and forwards it to the neuron if the addresses match. FPGA-to-chip data transfer is implemented via two communication technologies. On the one hand an Low-Voltage Differential Signaling (LVDS) high-speed link, hereinafter simply called high-speed link. And on the other hand a much slower but robust communication channel via Joint Test Action Group (JTAG). The latter is used to initialize the links or as a fallback when physical link connection is faulty.

Auxiliary Hardware

To utilize the neuromorphic wafers a multitude of auxiliary hardware is needed. Figure 2.3b shows an exploded view of the various modular components constituting a BSS-1 wafer module. 48 FPGA boards provide a communication interface for configuration and spike data. Further boards provide readout for analog data, i.e., neuron membrane voltages, and generation of various supply voltages. All components are interconnected via a central PCB.

Platform Setup

The BSS-1 compute platform constitutes several wafer modules, conventional computes nodes for experiment control and analysis as well as the necessary communication infrastructure. Figure 2.4 shows a photograph of the fully assembled platform.

Fixed Pattern Noise

One of the major challenges when working with analog circuitry are the inherent parameter variations due to the underlying CMOS manufacturing process. This

Figure 2.4: BSS-1 neuromorphic computing platform. Up to 4 wafer modules are mounted in a rack. Each wafer is connected via $48 \times 1\,{}^{\text{Gbit}}/_{\text{s}}$ ports. Conventional compute nodes are mounted in middle rack and connected to the setups via $10\,{}^{\text{Gbit}}/_{\text{s}}$ switches.

means, setting model parameters for multiple neurons to the same value results in different behaviour. To circumvent this, calibration is applied which finds particular parameter settings so that neurons show similar behaviour. The concept is illustrated in Figure 2.5. This calibration however cannot completely hide all variations due to the limited parameter resolution on hardware. Furthermore, the available parameter range might not be sufficiently large that all neurons can be calibrated for a desired parameter set.

Software Support

The BSS-1 computing platform provides support for the *PyNN* API via a custom back-end. The underlying software stack manages translation of the abstract experiment description to a valid hardware configuration and executes it. Müller et al. 2020b presents a more detailed view of the BSS-1 software and system operation, which includes work covered in chapter 4. Several of these concepts are also presented in chapter 3 which addresses the design of the BSS-2 software architecture.

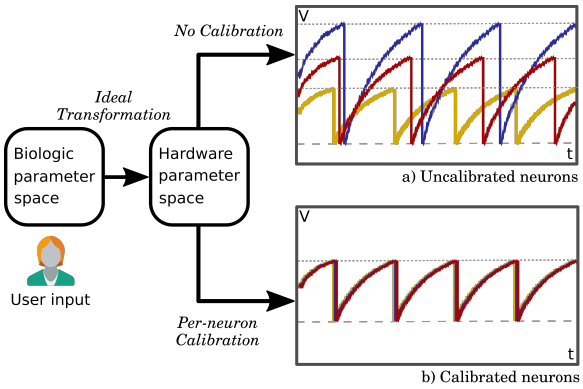


Figure 2.5: Principle of neuron parameter calibration. User specifies a parameter sets. Fixed pattern noise inherent to the substrate leads to variations in behaviour without calibration (a). With applied calibration (b), parameters are slightly modified to compensate variations, resulting in similar neuron behaviour. Taken from Müller et al. 2020b.

2.2.4 BrainScaleS-2

The BSS-2 chip generation represents the shift from 180 nm to 65 nm CMOS technology. A change to the higher resolution manufacturing process enabled several improvements and new features which are explained in the following. Nevertheless, the core principles stay the same, neuron dynamics are physically emulated as analog circuits in a mixed-signal system. As described in section 2.2.2 there were two prototype generations for BSS-2, the HICANN-DLS and the HICANN-X chips. Conceptually both are similar with HICANN-X for the most part being a scaled up version of HICANN-DLS, going from 32 neurons with 32 synapses each to 512 neurons with 256 synapses. Therefore only the most current iteration, the HICANN-X v2 chip, is described, especially its differences to BSS-1. Hereinafter BSS-2 refers to this chip version.

Chip Overview

The BSS-2 chip implements 512 AdEx neurons with 256 CUBA synapses each. Parameter ranges of the analog circuits were chosen to provide an acceleration of about 10^3 compared to biological real-time. This speedup is one order of magnitude smaller than on BSS-1 which was decided to relax timing constraints.

A simplified schematic of the chip functionality and relevant components is illustrated in fig. 2.6 and is explained in the subsequent sections. It is a stark abstraction of the actual highly symmetrical and repetitive chip layout which can be seen in fig. 2.7 on the left. For more in-depth descriptions refer to Schemmel et al. 2020.

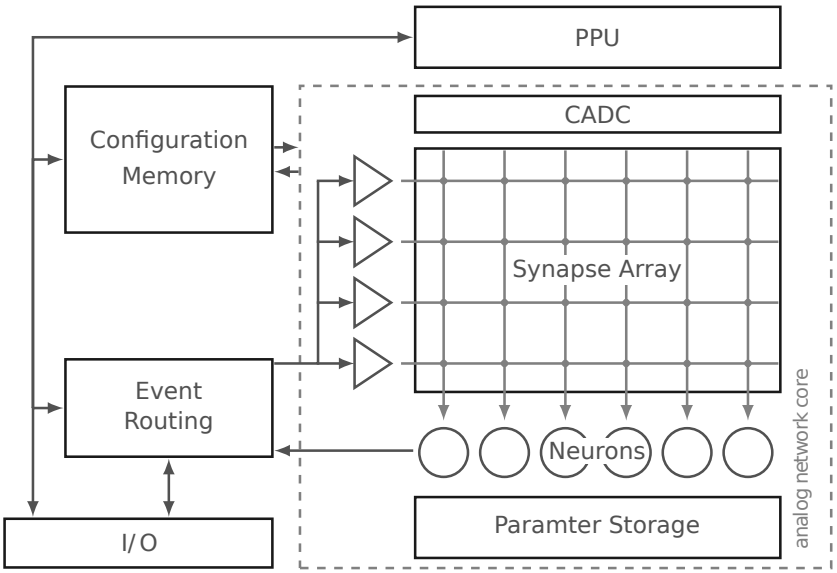


Figure 2.6: Abstract schematic of the BSS-2 chip-layout and functionality. The dotted box on the right annotates the analog network core, i.e., neurons and their synapses. It is surrounded by data-flow and control logic which are connected to the off-chip I/O. Taken from Billaudelle et al. 2020.

I/O

Data in- and output of the chip is handled via a communication FPGA. It handles timed release of configuration data and spike events onto the chip as well as data readout. Multiple spike events can be packed into single communication packets to further utilize the limited bandwidth. This results in spikes taking up 4 bit in compressed and 8 bit in uncompressed case [Karasenko 2020].

Configuration Memory

Configuration memory represents access to the configuration space of the various on chip components. This configuration covers for example control parameters for the operation points of components but also digital parameters of the network components like synaptic weights.

Analog Parameter Storage

Storage of the analog neuron circuit parameters was changed from floating gates on BSS-1 to a capacitive memory [Hock 2014]. One of the main advantages of this approach are significantly lower trial-to-trial variations.

Event Routing

Event routing handles interconnection of neuron spike in- and output as well as on- and off-chip spike sources. Such a routing is needed as full static connectivity becomes infeasible rather quickly. Off-chip sources correspond to either host computer generated events or events from other chips in future multi-chip experiments. On-chip exist a configurable spike generator which generates either regular or Poisson distributed spike sequences. The digital packets representing spike events contain the address of the source neuron. An event packet is then filtered by the components involved in routing according to this address. The challenging part then becomes to find a valid routing under the given constraints (see section 3.4.2).

On-Chip Analog Readout

Another improvement over BSS-1 are extensive on-chip analog readout capabilities. On the one hand, there is a Analog-to-Digital Converter (ADC) with high temporal resolution which can readout analog parameters of one neuron at a time. As its primary application is the readout of membrane voltages it is called MADC. On the other hand, there are lower resolution ADCs for each neuron/synapse column, hence called CADC. There, a trade-off can be made between temporal resolution and number of concurrent read out columns. They are primarily used to read out correlation capacitors, which are in turn used to facilitated STDP like learning rules (section 2.1.4) via the Plasticity Processing Unit (PPU). Alternatively they can be utilized to readout membrane voltages, albeit with a significantly lower resolution than the MADC.

Plasticity Processing Unit

The so-called Plasticity Processing Unit (PPU) is a general purpose 32 bit Single Instruction Multiple Data (SIMD) processor integrated on the same substrate with the primary intent to implement arbitrary plasticity rules, hence the name [Friedmann 2013]. It is based on the PowerPC architecture with a custom-built vector unit extension. It provides parallel computation for plasticity relevant observables, namely the synapse array and results from the correlation sensors. Most other observable can be accessed in a serial fashion. Available C and C++ compiler support for the PowerPC instruction set was extended for the vector unit allowing to write efficient code conveniently [Heimbrecht 2017]. This code can then be loaded into the PPU memory for later execution. Due to its general purpose nature it is furthermore utilized, for example, as an experiment controller or a virtual environment for accelerated robotics applications [Wunderlich et al. 2019; Schreiber 2021].

Multi Compartment Capabilities

Furthermore, the chip provides multi compartment capabilities as described in section 2.1.3. However, these capabilities are not explicitly covered in this thesis. Kaiser et al. 2021 provides further insight into their application.

Analog Multiply-Accumulate

BSS-2 provides in addition to the SNN operation also the capabilities to perform analog matrix multiplications. This is implemented with the same circuitry but with a different configuration set, that transforms the synapse matrix and neuron circuit into multiply accumulate units. This operation mode can be utilized to perform ANN-based ML task for example demonstrated in Stradmann et al. 2021. However, these capabilities are not applied in this thesis as the main focus lies on spiking operation. Nevertheless, the software architecture described in chapter 3 fully covers this use case.

Platform Setup

The BSS-2 chips are not yet ready for wafer-scale integration, nevertheless they are already utilized in several studies with smaller scale networks [Billaudelle et al. 2020; Göltz et al. 2021; Czischek et al. 2021]. Figure 2.7 shows photographs of a bonded BSS-2 chip and several single chip setups constitution the current state of the BSS-2 computing platform. They are connected to several compute nodes for experiment control and analysis.

2.3 Software Development Concepts

A significant portion of the work conducted in this thesis covers the design and development of software to facilitate neuromorphic research. As the lifetime of the neuromorphic experiment platform, including its software, typically exceeds the stay of individual group members it needs to be sustainable. This necessitates a sufficient level of software quality. However, the impact of software quality and sustainability in science is an open issue [Hatton 2007; Merali 2010], down to the ambiguity of its definition [Venters et al. 2014].

A plain example of bad software quality is wrong code, often simply called bugs. One prominent case was the discovery of bugs in functional Magnetic Resonance Imaging (fMRI) analysis tools which question the validity of a multitude of studies [Eklund et al. 2016]. Another more tragic example were the Therac-25 accidents, where software bugs and insufficient quality-control culminated in administration of radiation overdoses leading to at least 3 fatalities and 3 seriously

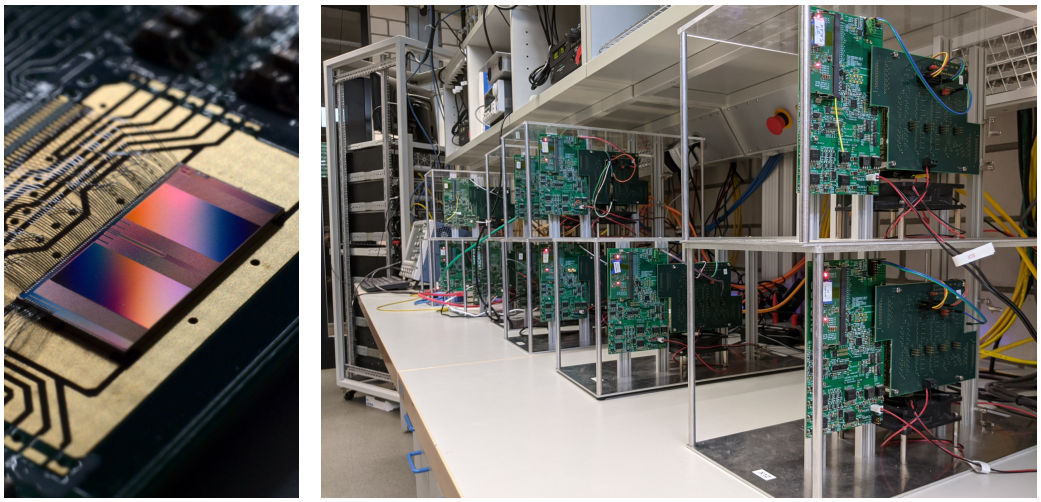


Figure 2.7: BSS-2 neuromorphic computing platform. Left: Single bonded HICANN-X chip. Image taken from Müller et al. 2020a. Right: Lab setups each carrying a single neuromorphic chip which is covered by the white plastic cap (top left). Each setup is connected via 1 $^{\text{Gbit}}/_{\text{s}}$ Ethernet which is controlled by a communication FPGA (not visible, on the back). Conventional compute nodes are mounted in a rack and connected to the setups via 10 $^{\text{Gbit}}/_{\text{s}}$ switches (left background).

injured [Leveson et al. 1993]. Neuromorphic systems are not yet in a state to endanger human lives, nevertheless, correctness needs to be a primary goal.

Further aspects to software quality, that are maybe not directly apparent, are, for example, code readability or type safety. To provide better context short introductions to software engineering concepts, tools and best practices is given which are applied in this thesis.

Testing Virtually all code contains errors and therefore needs to be tested extensively. There are several approaches to verify desired code behaviour, one being so-called unit testing. Its principle is to test individual contained parts of the code base as it is easier to manage testing for small modules. Conversely, integration testing describes verifying the correct operation of large sections or the entire code base. They are necessary as unit testing does not cover the interactions between units.

Code Readability If the task or logic code is easy to grasp it has good readability. This can be facilitated by consistent formatting, e.g., indents, or comments describing the intent of the code. But also avoiding convoluted nesting of condi-

tionals and loops helps readability. So-called code linter tools can be utilized to automate consistent formatting, reducing load on the developer.

Code Duplication Duplicate code, i.e., multiple code segments that fulfill the same task, is undesirable. It introduces unnecessary maintenance-overhead and can make code less readable. Duplicate code segments should therefore be refactored to a single source of truth that is easier to maintain.

Technical Dept Technical dept is used to describe the pitfall of choosing a quick but insufficient solution instead of a sustainable but more time-involving solution. The former leads to accumulation of development costs in the future, i.e. dept. Examples are, skipping implementation of tests or insufficient documentation.

Version Control Software source code is often a living document which is frequently modified. Version control tools provide a means to track and uniquely identify these changes to code. This is especially valuable when multiple persons work in the same code base. *Git*[2] is a prominent version control tool and utilized for the software development conducted in the Electronic Vision(s) group.

Code Review Code Review describes the practice that changes to code, typically done via version control, are reviewed other developers. It is ubiquitous in open-source and industrial software development as a means to increase software quality [Bird et al. 2013]. The Electronic Vision(s) group practices code review not only for quality assurance but also as a learning method for new group members which, as physics students, typically do not have a pronounced software background. To this end the *gerrit*[3] framework is utilized by the group.

Continuous Integration Continuous Integration describes the practice of constantly verifying all changes done to the code base. By ensuring that all changes do not break any tests the software continuously remains in a usable state.

[2]https://git-scm.com/ 2021-09-16
[3]https://www.gerritcodereview.com/ 2021-09-16

Chapter 3

Neuromorphic Software Architecture

The previous chapter introduced the BSS neuromorphic hardware platform in the context of computational neuroscience. Furthermore, some concepts of software engineering were introduced that are relevant for the work in this chapter. The following presents the software framework that enables utilization of the BSS-2 neuromorphic hardware.

First, different challenges and requirements to the software, like the high acceleration factor, are highlighted in section 3.1. The resulting design decisions for the software architecture are discussed. To this end a broad overview of the various layers is given, including a discussion regarding the collaborative nature of the performed software work. A more detailed descriptions of these layers follows in the subsequent sections. Afterwards, performance measurements utilizing all layers are conducted to validate their proper implementation, in section 3.6. In particular, scalability towards multi-chip experiments is investigated. The chapter is concluded by a 4×4 Sudoku solver demonstrating the capabilities of hard- and software.

The concepts presented in the following chapter are explained in the context of the latest BSS-2 chip generations. However, most concepts are applicable not only to the specific substrate but to any hardware system with similar application and constraints. They are the culmination of the design and development efforts of many previous chip generations.

3.1 Architecture

3.1.1 Goals and Requirements

Designing and implementing a software framework that enables convenient utilization of novel neuromorphic devices is no small feat. On one side of the spectrum, custom designed hardware needs to be abstracted and controlled. And on the other side high-level experiment frameworks need to be supported which automatically handle the hardware specifics to provide efficient workflows.

To provide better context the different goals and requirements are discussed in the following. First, the various potential users with diverse scientific background are characterized. Then, relevant modes of operation of a neuromorphic system are explained. Finally, the resulting requirements to performance are highlighted.

Target Users

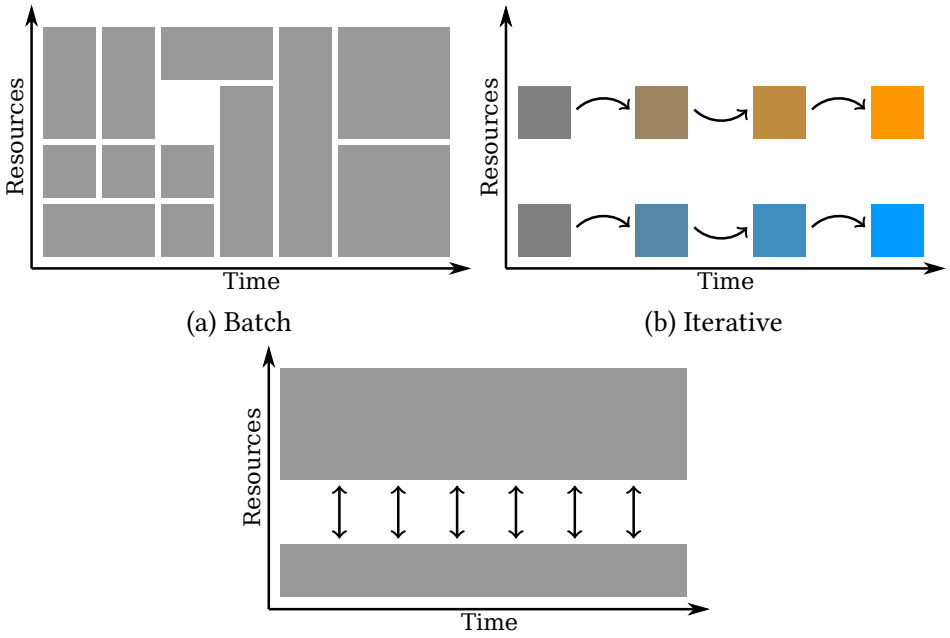
Computational neuroscience is a highly interdisciplinary research field. As a consequence potential users of a neuromorphic device have varying areas of expertise and expectations regarding its usage. Furthermore, typical workflows during development and commissioning of the systems differs from typical neuroscience experiment-workflow. A hardware developer may require low-level access for fast prototyping and testing of individual components. On the other hand, for high-level neuroscience experiments as much hardware specifics as possible, e.g. topological constraints or fixed pattern noise, should be automatically taken care of. Thus, usage of the systems needs to be supported on different levels of abstraction.

Experiment Usage Modes

To provide a suitable software framework that utilizes the neuromorphic hardware efficiently one first needs to consider potential modes of operation. We differentiate between three main experiment usage modes, shown in fig. 3.1.

Batch In batch mode each experiment can be independently executed. Each individual run fully describes neural network configuration and spike input. Such experiments can be parallelized by consecutive execution on multiple hardware setups to increase throughput. Example experiments of this category are long-running learning experiments utilizing on-chip plasticity capabilities.

Iterative (in-the-loop) The iterative or also called in-the-loop usage describes experiments with multiple consecutive hardware runs that depend on the previ-



(a) Batch (b) Iterative

(c) Closed Loop

Figure 3.1: Sketch of different usage modes of neuromorphic hardware. Blocks represent execution instances on the hardware.

ous execution. Compared to batch experiments individual hardware executions are typical shorter lived. On first iteration a neural network with some initial conditions is executed on hardware. Response data, for example spike trains are read back and analyzed. For the following run weights or spike input are modified according to some learning update rule. This is repeated until convergence of some optimization criteria is reached. Examples are learning methods inspired by machine learning approaches like the back propagation algorithm [Rumelhart et al. 1986; Bellec et al. 2019].

Closed Loop Closed loop experiments differ compared to batch or iterative insofar as that the experiment controller is time-coupled to the network emulation. This demands tight constraints to communication latency and performance. Typical examples are sensor-motor loops where an agent navigates a virtual environment.

The rest of this chapter will focus on the first two usage modes. An exemplary work that focuses on closed loop operation utilizing the on-chip processor (PPU) can be found in Schreiber 2021.

Performance

The aforementioned use cases and the nature of analog neuromorphic hardware create certain requirements to a software stack. Especially the high acceleration factor of 10^3 and time-continuous emulation necessitate performant software. As the network emulation cannot be paused data transfer and processing throughput need to be high. Simultaneously low latency is relevant in particular for iterative use cases due to their sequential nature.

3.1.2 Software Stack Overview

Taking the requirements explained in the previous section into account an overview of the developed software stack is given. First, the utilized programming languages are and considerations to sustainability are discussed. Afterwards the architecture of different layers is shortly presented. It aims to provide an introduction of the overarching idea with the individual layers being explained in more detail in subsequent sections.

Utilized Programming Languages

The two programming languages dominantly utilized in the software stack are C++ and Python. The core of most layers is written in C++ due to is high performance and efficiency. Its multi-paradigm support allows for different approaches to the various tasks required to utilize neuromorphic hardware. Due to its strongly-typed nature many issues can already be tackled at compile time further increasing performance and correctness. A further reason for its use is legacy - most already existing code which can be reused is written in C++.

Python is prevalent in the natural sciences due to its flexibility, ease of use and thus relative low learning curve. There are a myriad of Python libraries improving efficiency for scientific workflows, e.g., *numpy*[1], *pandas*[2] or *matplotlib*[3]. Supporting a Python interface gives access to all these valuable tools. Therefore, virtually all frameworks in neuroscience and machine learning provide a Python API with the BSS-2 software stack being no exception [Abadi et al. 2015; Rhodes et al. 2018; Lin et al. 2018; Paszke et al. 2019a]. To this end the library *genpybind* [Klähn 2020] is employed which provides automated wrapping of C++ via code annotations. Yet, providing support for both languages leads to design constraints for the APIs connecting the various layers.

[1] http://numpy.scipy.org 2021-07-20
[2] https://pandas.pydata.org/ 2021-07-20
[3] https://matplotlib.org/ 2021-07-20

Sustainability

Development of neuromorphic chips is a long-lasting effort over several years, involving changes to various parts of system design. The software therefore needs to be sustainable, e.g., easily adaptable to changes of hardware components. This is achieved by separating the software in multiple independent layers, i.e., splitting it up in modules. Layering is a very common architecture pattern that allows, if executed properly, changes to parts of a code base, e.g. communication, with only minor changes to rest of the code. This requires well-defined APIs for the individual layers which has the additional benefit of providing access to experimenters on different levels of abstraction.

The single-chip prototype setups are planned with multi-chip or even wafer scale operation in mind. This demands bearing parallel execution on multiple chip instance in mind when designing the software architecture. Furthermore, the ability to unit test sections of code should also be taken into account to increase stability and sustainability.

Developed Layers

As explained earlier structuring software into well-defined layers is vital in keeping it maintainable and extendable. Figure 3.2 shows a schematic of the developed software architecture and its various applications on different levels.

The stack will be described from a bottom up view as this is the typical development priority when implementing a new chip. The presented schematic is not exhaustive and only shows layers and corresponding repositories directly relevant for experiment control and abstraction of the BSS-2 neuromorphic hardware. All mentioned repositories are open source under the *GNU Lesser General Public License v2* and available at `https://github.com/electronicvisions`.

Communication The first step to utilize hardware systems, from the software point of view, is the ability to send and receive data. With proper abstraction the underlying transport protocol and technology should be interchangeable. Communication is therefore structured into a common connection interface *hxcomm* that supports various back-ends (cf. section 3.2).

Hardware Abstraction The next higher layer category is the *hardware abstraction*, presented in section 3.3. Responsibility of this layer can be compared to device drivers. It provides an abstract structure of the various hardware and chip components and their control flow.

Within this category the lowest layer is the abstraction of control FPGA instructions. Combined with communication this is already sufficient to provide

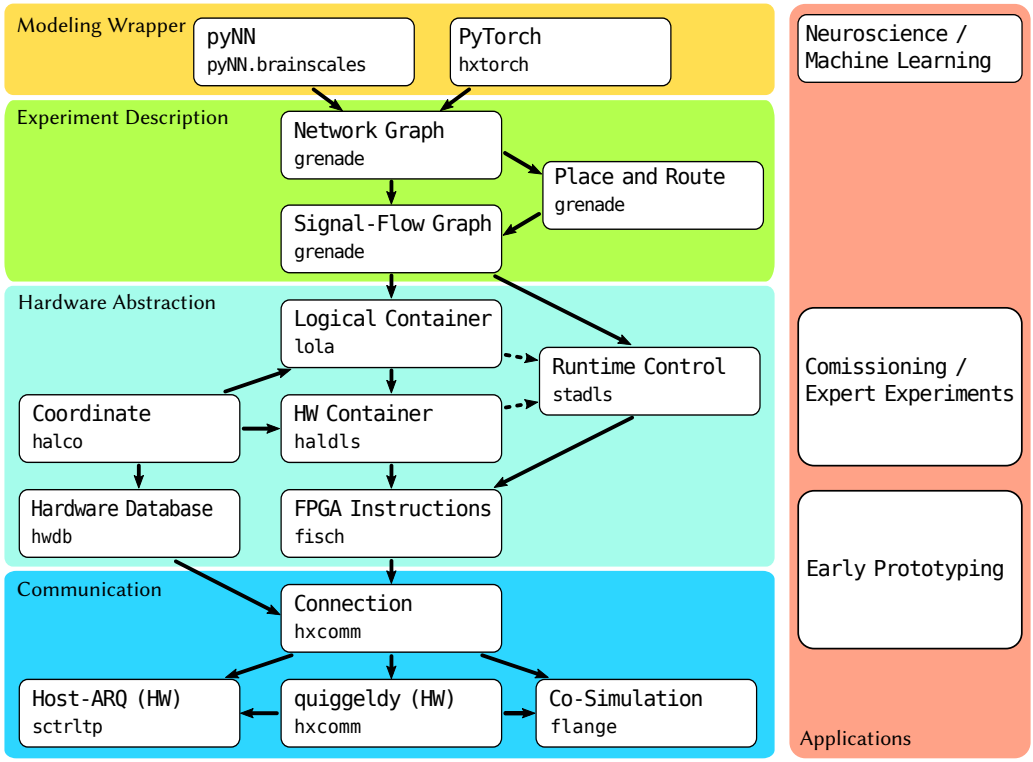


Figure 3.2: Overview of the developed BSS-2 software architecture and its applications. Left side: Colored boxes in background represent categories of different levels of abstraction. White boxes represent individual layers with their specific repositories names and their dependencies. Right side: Various applications on their corresponding abstraction levels, i.e., which APIs they utilize. See text for detailed description of the individual components.

an interface for fast prototyping in early stages of system development, i.e., manually setting addresses and bits.

An intuitive structure of the convoluted address space is provided by the *coordinate* layer. It represents each hardware component via custom ranged types that can be converted to other corresponding types.

A structured representation of the configuration space of hardware components is implemented in the *container* layer. These containers also define de- and encoding of their abstract representation to the on-hardware data formatting. A pair of coordinate and container objects then represent the state of a uniquely identifiable hardware component.

The *runtime control* layer provides an interface to describe timed sequenced of read and write instructions of such coordinate-container pairs as well es spike events. These timed sequences, also called playback programs, can then be loaded

to and executed on the hardware which sends back record response data. On this level of abstraction most of commissioning and early expert experiments are conducted (section 3.3.6).

Experiment Description To effectively utilize the hardware especially for larger experiments an abstract description of neural network experiments is needed. First a signal-flow graph representing the hardware configuration is defined. It contains a hardware abstraction graph and an experiment flow description interface that utilizes these graphs.

Building on-top of the signal-flow graph description a high-level abstract representation of neural network topology is developed. An automated translation from this high level abstraction to a valid hardware configuration is handled by a place and route interface. Detailed explanation can be found in section 3.4.

Modeling Wrapper Various back-end agnostic modeling languages emerged to provide access to various simulators or neuromorphic hardware systems to a wide range of researchers. The BSS-2 software stack comprises wrappers to two of such modeling frameworks, namely *PyNN* [Davison et al. 2009b] and *PyTorch* [Paszke et al. 2019b]. Their goal is to hide as many hardware specifics as possible which is not always expedient to fully exploit all capabilities of specif hardware back-ends. The necessary trade-offs and modifications to the APIs are described in section 3.5.

3.1.3 Prior Work

The Electronic Vision(s) group accumulated valuable experience regarding hardware as well as software design over the past two decades. Many aspects in this thesis profit from and built upon the results acquired during development of previous chip generations, especially BSS-1 [Brüderle 2009; Müller 2014; Jeltsch 2014]. A short overview of the lessons learned during their development and operation is given. Likewise, it is discussed how they influenced the earlier explained architecture of the BSS-2 software stack.

Most requirements for the BSS-2 system previously explained in section 3.1.1 also apply to BSS-1. Thus, several principles like layering, coordinates or abstraction of hardware components into container were already utilized. One aspect lacking was a clear separation of communication and hardware abstraction, in particular data de- and encoding. The encoding and decoding functionalities directly performed corresponding read and write command to the hardware which hamper serialization and abstraction of different communication back-ends Therefore a deliberate effort is made to separate the different layers as well as possible.

Another inadequate aspect was that hardware specifics where abstracted away too early in too low levels of the software stack. This complicated debugging in higher levels of the software stack as not the full set of possible hardware features and observables was accessible.

Nevertheless, many concepts and feature proved valuable. For example the Ethernet based custom ARQ-protocol connecting FPGA and hostiles could directly be reused. Also, the coordinate framework could be utilized as is explained later in section 3.3.1.

3.1.4 Collaborative Work

Development, commissioning and operation of novel neuromorphic devices is a long-lasting effort of dozens of people. Especially software development is highly collaborative manifesting for example in the code review based workflow practiced by the group (section 2.3). Since the introduction of the first BSS hardware architecture E. Müller leads the software development efforts and provides architectural guidance. Still, almost all detailed software design decisions are discussed and decided as a group. For clarity, an overview of the contributions from the author as well as other main contributors to the software architecture are stated. Major contributions are additionally stated again at the beginning of each section. E. Müller is a common contributor and thus not mentioned below.

Section 3.2 Communication
: The connection interface was developed by Y. Stradmann, O. Breitwieser and P. Spilger in collaboration with FPGA developers. The author integrated the HostARQ communication back-end, added various stability and enhancement changes including a protocol versioning scheme. The Co-simulation back-end was mainly developed by P. Spilger. The quiggeldy back-end was developed by O. Breitwieser.

Section 3.3 Hardware Abstraction
: Initial conceptualization and implementation of coordinates, container and runtime control were performed by the author, J. Klähn and D. Stöckel for the HICANN-DLS chip. The initial implementation for this chip was finalized by P. Spilger. Additional extensions and further improvements for the full-size HICANN-X chip were performed by the author and P. Spilger.

Section 3.4 Experiment Description
: Conceptualization and implementation was done by P. Spilger during his master thesis [Spilger 2021] which the author co-supervised. The

author contributed by validation work and performance analysis presented in this thesis.

Section 3.5 Modeling Wrapper
The PyNN back-end was established by Milena Czierlinski during her bachelor thesis [Czierlinski 2020] which the author co-supervised. The author subsequently refactored, improved the implementation and added various features including calibration injection, manual neuron placement or quiggeldy micro-scheduler support. P. Spilger and the author integrated the experiment description layer which was validated in this thesis.

3.2 Communication

Communication describes the ability to exchange data between a host machine and an instance of neuromorphic hardware be it real or simulated. It represents the lowest level of the software stack and specifics of the implemented communication protocol should be irrelevant to upper layers. Communication layer is therefore split into a connection API and the various abstracted away back ends.

3.2.1 Connection Interface

The basic instructions a communication interface needs to provide are sending and receiving of data. Execution and receiving of data to and from the chip is managed by the controlling FPGA in a chunk like fashion, which is explained in more detail in section 3.3.3. Hence, the interface is designed to take one or multiple messages, i.e. a vector, to be sent and returns a vector of response data and potential run time statistics like connection stability and errors. Furthermore, the interface is typed to support differentiation of various FPGA-to-hardware transport protocols like JTAG [IEEE 2001], Serial Peripheral Interface (SPI) or *omnibus* [Friedmann 2015] which are represented by so called UT-messages [Karasenko 2020]. Multiple on chip transport protocols are supported as they have vastly different trade-offs between throughput and stability. These protocols should not be confused with the different possible host-to-FPGA communication protocols implemented by the back-ends explained in following sections.

Another useful feature of this layer is the ability to choose a desired communication back-end at runtime either explicitly in code or through environment variables. This allows fast and easy switching between different back-ends, e.g., during testing on different setups without changing or recompiling code. Further-

more, extension to other potential communication back-ends, e.g., PCI-Express, is properly encapsulated.

3.2.2 Back-Ends

Host-ARQ

Host-ARQ is an Ethernet-based light-weight implementation of a sliding window transport protocol similar to Transmission Control Protocol (TCP) between conventional host and control FPGA. It is the communication back-end used for the BSS-2-HICANN-X hardware setups. The accelerated nature of the neuromorphic system demands high throughput so that data transfer does not become a bottleneck. Host-ARQ is therefore highly optimized. It was primarily developed in Müller 2014 for BSS-1 and further enhanced in Mauch 2016 and this thesis. For example error handling is improved and more descriptive. A versioning scheme is implemented to support automated detection of host and FPGA functionality mismatch. This improves productivity of experimenters as time investigating cryptic error messages is reduced.

Measurements of the raw Host-ARQ implementation were previously conducted for BSS-1 systems with near maximum performance in Müller 2014; Mauch 2016. The theoretical maximum throughput for Host-ARQ with maximum packet size via a $1\,{}^{\text{Gbit}}/_{\text{s}}$ Ethernet connection is 117 MB/s. To ensure proper migration of FPGA and software implementation measurements where repeated for BSS-2. Figure 3.3 shows the throughput for increasing batches of loop-backed data over 5 repetitions. For smaller amounts of data throughput cannot be fully utilized until a saturation is reached. This is expected as the round trip time dominates execution time for small payload sizes. With $115\,{}^{\text{MB}}/_{\text{s}}$ this saturation is close to the theoretical maximum. Later, Section 3.6 investigates this performance under scaling for the full BSS-2 stack.

Co-Simulation

In the past unit tests for hardware simulations needed to be implemented in a separate software environment compatible with the simulation environment. This lead to additional overhead such as duplicate implementations of the same test but also prohibited full integration tests resulting in limited test coverage. The back-end provides a connection to hardware simulations of the FPGA and current as well as future chip revisions. Thereby enabling hardware/software co-design, i.e., testing and development with the full software stack prior to expensive hardware tape-outs. It is implemented by a Remote Call Framework (RCF) based connection to a running simulation server.

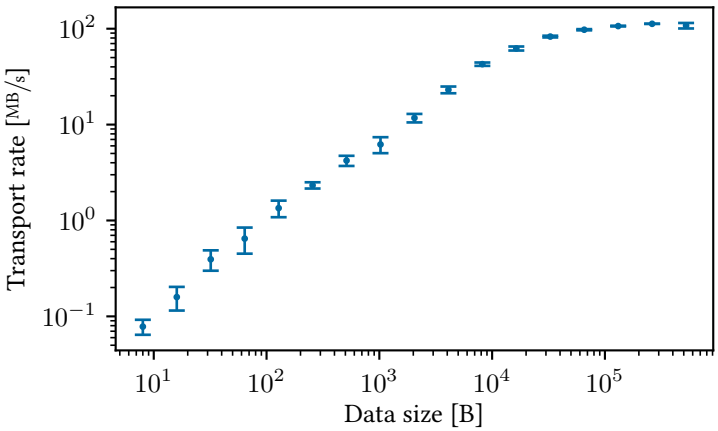


Figure 3.3: Rate of looped-back transport of data between host computer and FPGA via 1 $^{\text{Gbit}}/_{\text{s}}$ Ethernet. The measurement is repeated 5 times for a data size between 8 B and 512 kB. For large-enough data to be transported, the rate approaches the expectation. Maximum rate is 115 $^{\text{MB}}/_{\text{s}}$. Modified from Spilger 2021.

Micro-Scheduler

Quiggeldy is a fast experiment micro scheduler with the goal of increasing hardware utilization. As it is closely related to platform operation, particularly cluster scheduling, its overall functionality is described in section 4.1.8. This scheduler is abstracted to a communication back-end that uses RCF to serialize the data stream to server instances. These server instances themselves implement either a Host-ARQ or and co-simulator back-end.

3.3 Hardware Abstraction

The plethora of configurable components of the neuromorphic system like neuron circuits, synapse array, analog memory, ADCs or PPUs leads to a large configuration space. Access, i.e. reading or writing, to this configuration is implemented on the hardware side via a register-like access. Registers are identified via an address and provide read and/or write access to 32 bit of data. Depending on the corresponding component this data can represent an integer, a boolean or an arbitrary bit formatting. The bit formatting of components needs to be abstracted in an intuitive structured way to make them manageable. Furthermore, the control flow of those units needs to be handled, e.g., order of writing the configuration is relevant due to interdependencies. The following sections present the software layers tackling this abstraction.

First, a hierarchical structure for the address space of the components as a kind of *coordinate system* is described in section 3.3.1. Then an abstract representation of the configuration of individual components, i.e., actual bits on hardware, is developed as so called *containers*. Section 3.3.2 explains the design decisions of these *containers* and how a pair of *coordinate* and *container* uniquely represent the configuration of a single component on a system. Utilizing those unique representations a runtime control framework is designed (cf. section 3.3.3) allowing to define temporal sequences of write and read commands. Finally, a representation of the state of hardware setups in a hardware database is explained in section 3.3.4.

3.3.1 Coordinates

To access an individual hardware component one needs its corresponding register address. There are hundreds of such registers which need to be represented in software. A naive approach would be to represent these by builtin numerical types, i.e., integers. Instead, custom ranged types are used, which provides type safety and other features like automated range checking.

Hardware design patterns result in repeating structures and therefore high symmetry in chip layout, which naturally leads to abstraction on different scales. Figure 3.4 shows the layout schematic of the latest BSS-2 chip with annotations for different component regions. The high symmetry is immediately evident with top and bottom chip halves being mirrored horizontally and therefore called *hemispheres*. Each hemisphere again is vertically mirrored resulting in *quadrants*. This symmetry is reflected in a hierarchical structure of the chip's coordinate system.

Repeating components can be addresses relative to the different regions. Neuron circuits are taken as an example. Depending on the hierarchical level, a neuron can be addressed via `NeuronOnQuadrant`, `NeuronOnHemisphere` or `NeuronOnChip`. All three have different allowed ranges of 128, 256 and 512 respectively.

It is possible to intuitively transform between these representations. Coordinates of a higher region can be cast down to a lower level for example `NeuronOnChip.toNeuronOnQuadrant()`. Vice versa, lower views can be combined to create higher-level coordinates, e.g., `NeuronOnChip(NeuronOnHemisphere, HemisphereOnChip)`. Coordinates of different components that relate to each other can also be transformed. A synapse is always related to a Neuron for example resulting in the possible translation of `SynapseOnQuadrant.toNeuronOnQuadrant()`. Translation from a neuron to synapse on the other hand will return a vector of all related synapses.

Furthermore, coordinates can be combined to form two-dimensional grids with synapses being an example. `SynapseOnQuadrant` is composed of `SynapseRowOn-`

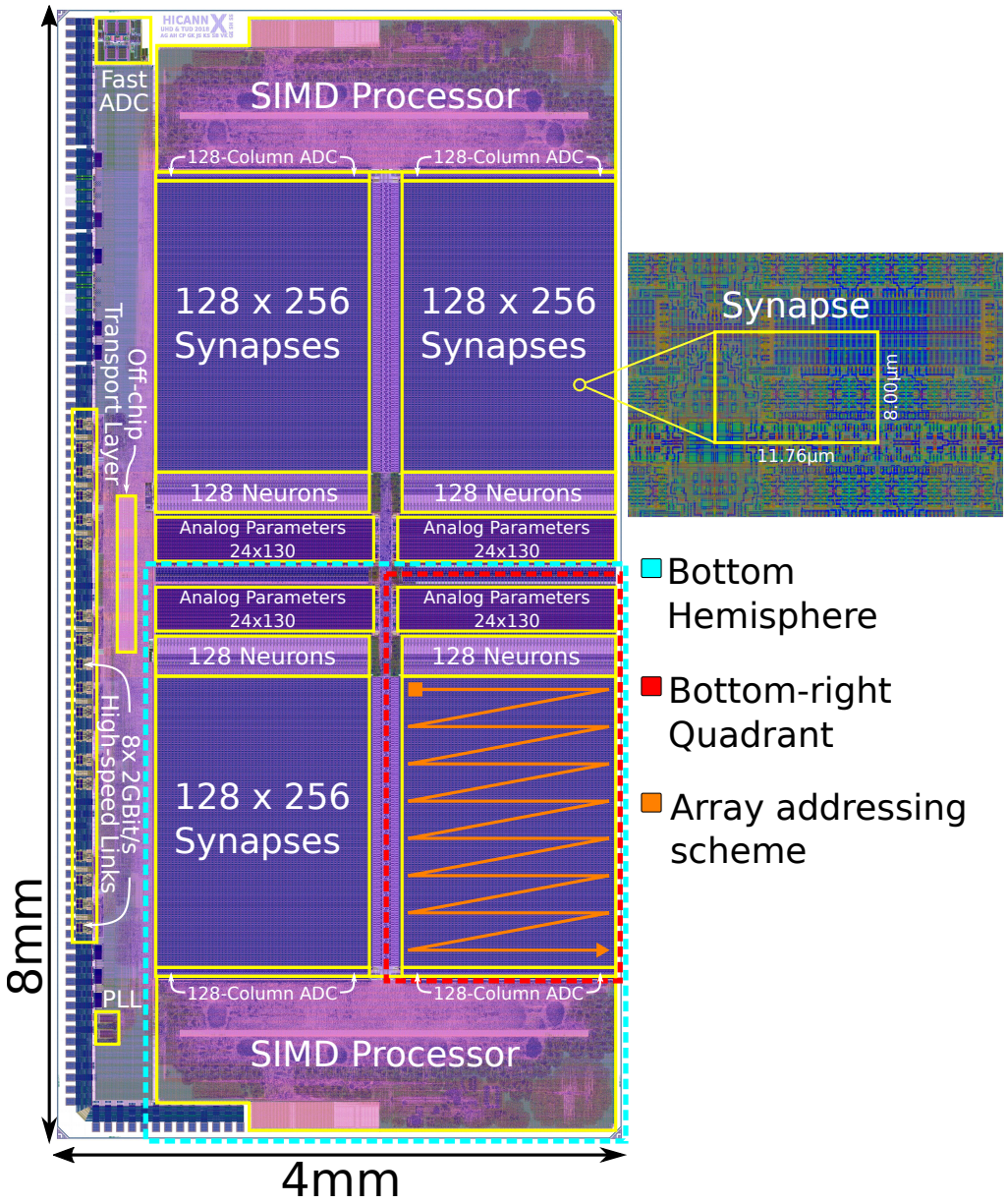


Figure 3.4: Schematic of the highly symmetrical layout of the latest BSS-2 chip. Various component regions are framed in yellow. Dashed lines indicate logically separable regions of the chip. Synapses and neurons are partitioned into four quadrants. Two embedded SIMD processors (PPU) as well as columnar ADCs are located in the upper and lower chip hemisphere. The row-major ordering scheme of two-dimensional coordinates is shown in orange. Taken from Müller et al. 2020c.

`Quadrant` and `SynapseColumnOnQuadrant` correlating to the number of synapses connected to a neuron (255) and the number of neurons (128) on a quadrant respectively. Such coordinates can then be constructed by X and Y components or by an enumerated number which is defined by row major order.

Additionally, coordinates can be utilized as iterators, e.g., in `for` loops. This facilitates, for instance, the creation of arrays with typed indexes as shown in listing 3.1.

Listing 3.1: Example usage of custom coordinate type

```
for(auto synapse : iter_all<SynapseOnChip>()) {
    my_synapse_matrix[synapse].weight =
        SynapseWeight(42);
}
```

Overall this typed coordinate system has several advantages over plain integers. First of all type safety and automated range checks enforce correctness. The ability of writing code that is relative to the respective regions and prevents code duplication and increases readability as the intent which coordinate ought to be used is explicit. Conversion tied to the individual coordinates further increases correctness, convenience and readability. Due to its lightweight design it comes with little to no downside in terms of run time performance.

The explained concept and implementation of a coordinate system were originally developed for the BSS-1 systems. The main changes accomplished during this thesis regarding the coordinates are explained in the following. First, the previous implementation was separated and transformed to a shared common repository for both BSS-1 and BSS-2. This allows both platforms to profit from subsequent improvements to the code base. But more importantly is serves as a basis for the shared resource management explained in section 4.1. A shift on a conceptual level compared to BSS-1 is that every component on chip must be represented by at least by one coordinate. This ensures a complete coverage and therefore addressability of the chip systems.

3.3.2 Container

As coordinates represent the address of hardware entities containers represent their configuration. The configuration of a hardware component is managed by one or multiple registers which depending on the component can be read from and/or be written to. A container object represents and stores a possible state of these registers. The hardware properties are abstracted on different software layers (cf. fig. 3.2).

FPGA-Instructions

On the lowest level a heterogeneous set of communication clients, e.g., SPI, JTAG or *omnibus* are abstracted away to a uniform register-access like interface. For example writing a value via SPI requires multiple commands which are passed to the underlying communication layer as aforementioned UT-messages. Yet, this operation is disguised to the upper facing layer as a single register word.

Hardware Containers

Building upon this encapsulation into different register word types, the next layer abstracts the configuration of the various hardware entities into containers. Figure 3.5 shows the concepts of this abstraction.

The configuration of individual hardware entities is encapsulated in one or more register words. Each word consists of a certain amount of data bits depending on the underlying on-chip protocol, for example *omnibus* has 32 bit. The meaning of these bits are abstracted in named member of the corresponding container. These members depending on the abstracted entity are appropriately typed. For example the switch for the leak term of the LIF neuron model named `enable_leak` is of boolean type. The corresponding value of the `refractory_time` is a ranged integer with the same implementation as coordinates (cf. section 3.3.1). This intuitive structure and naming leads to quasi automated in-code documentation.

Individual containers adhere to the smallest possible read/write granularity to not constraint higher layer abstraction as well as preventing of sending and receiving unnecessary data. Each container implements a conversion from its state to a vector of payload words called `encoding`. The reverse conversion, i.e., taking raw payload bits and setting its state, is called `decoding` respectively. Additionally, a translation of corresponding coordinate to the actual bit address is provided. These functions are employed by the runtime control described later in section 3.3.3. Implementation wise bit-fields are utilized for translation between named members and plain bits, see listing 3.2 for an example. Compared to plain manual bit shifts theses provide clear, readable and therefore maintainable code. Furthermore, containers are composable to allow forming of lager of units of repetitive circuits, for example grouping multiple synapses into a matrix.

Logical Containers

The composition feature is utilized in the next higher layer (Logical Containers) to group different types of containers to form logical units. For example all containers related to a neuron are composed to define its state, e.g. its analog parameters like leak voltage as well as its digital parameters. When designing the structure of these containers typical usage as well as performance needs to be

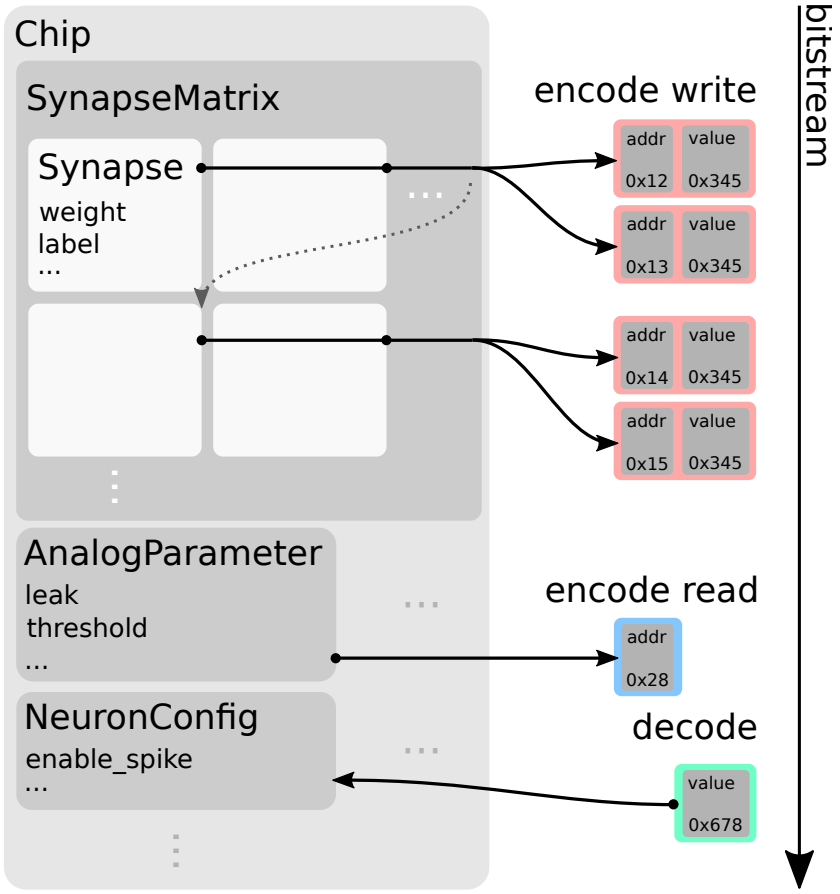


Figure 3.5: Configurable hardware entities are modeled by nested data structures encapsulating named data elements. An algorithm visits recursively the nested data structures and generates a hardware configuration bitstream. Taken from Müller et al. 2020c.

considered. A common use case are iterative experiments where only synaptic weights are updated. It is therefore imperative that the synapse weight matrix remains its own container. The logical layer furthermore provides an abstraction for compositions of individual neuron circuits to multi-compartment neurons (cf. section 2.1.3).

The composition feature is implemented in a tree like fashion where only leaf nodes actually store data. Encoding and decoding is implemented in a visitor pattern which recursively travels this tree to generate address and corresponding payload words from the leaf nodes. This design provides a clear and extendable interface for data composition abstraction.

Regarding the various communication protocols described for the lowest ab-

Listing 3.2: Example bit-field utilized for bit formatting of spike background generator container.

```
struct __attribute__((packed)) {
                                // bits ; word
  uint32_t enable         :  1; // 0     ; 0
  uint32_t enable_random  :  1; // 1     ; 0
  uint32_t /* unused */   : 14; // 2-15  ; 0
  uint32_t period         : 16; // 16-31 ; 0
  uint32_t seed           : 32; // 0-31  ; 1
  uint32_t mask           :  8; // 0-7   ; 2
  uint32_t rate           :  8; // 8-15  ; 2
  uint32_t neuron_label   : 14; // 15-29 ; 2
  uint32_t /* unused */   :  2; // 29-31 ; 2
};
```

straction layer there can exist a 1⟶N relation between a container and multiple register word types. This abstracts the fact that a specific entity might be accessible via different on-chip communication protocols, e.g., JTAG or high speed links. The desired protocol can transparently be selected upon invocation of the aforementioned visitor. This provides an easy shift between debugging and normal operation, where prior utilizes the near fail-safe but slow JTAG protocol.

An additional feature of the hardware abstraction layer is the support for multiple chip versions via separated C++ name-spaces. As only a small subset of features changes in-between chip iterations most code can therefore be shared features between the chip versions. This is especially valuable in the transition period when a new chip is commissioned while the old chip is still used by the majority of researchers.

Storing and loading of container instances is facilitated by a serialization method. With this a specific hardware configuration can be stored for example to a file and loaded by another process at a later time or on a different machine. This is for example utilized by the current calibration tools.

Overall container provide a unified abstract interface of the neuromorphic hardware configuration for upper facing layers hiding implementation details like conversion to and from register values.

3.3.3 Runtime Control

In section 3.1.1 three generalized usage modes *batch*, *iterative* and *closed loop* are outlined. The following presents how these usage modes are covered by the runtime control.

Due to the dynamic nature of the emulated neuron model timing of input stimulus, output data, access to observables as well as controllables is essential. Additionally, configuration of the system might also require time-controlled execution for technical or experiment control reasons. To this end the control FPGA implements a runtime control framework. The corresponding software representation was developed during this thesis.

First, the software interface abstracting runtime control is explained. Then, runtime control is illustrated based on a hypothetical experiment running on BSS-2.

Runtime control is described by a temporally ordered sequence of commands, hereafter called playback program. Generation of such a sequence is comprised of three functions `write`, `read` and `block_until`. The first two functions represent handling of data be it sending of input spikes, setting or reading out configuration. They utilize the coordinates and containers explained in previous sections. The `write` function takes a container instance as data payload and a corresponding coordinate as destination. `read` only takes a coordinate as parameter because each hardware component is uniquely identifiable by a specific coordinate instance. An `std::future`-like[4] object is returned representing the to be read data which will become valid after experiment execution. The `block_until` function enables timing of data commands by halting release of subsequent commands until a specific condition. The most commonly utilized condition is reaching a certain time tracked by a timer on the FPGA. Said timer can be reset to enable relative timing instead of keeping track of one monolithic timing. Additional conditions are checking emptiness of communication channels or comparison against the value of a specific register. The latter for example is used to synchronize playback program execution with PPU-programs. Listing 3.3 provides a simple code example of playback generation. Generation of playback programs is handled via the widespread builder design pattern, leading to better separation of construction and representation of complex timed sequences.

Next, flow of a playback program execution is described with the aid of an exemplary hypothetical experiment. The experiment encompasses external spike stimulus with concurrent on chip PPU interaction, e.g., implementing synaptic plasticity. Figure 3.6 illustrates and describes the runtime flow of the experiment. Only parts involving the actual neuromorphic hardware are shown but not for example neural network setup or analysis of recorded data.

In the following some of the considerations that went into the design of the runtime control are further discussed. Especially scaling of neuromorphic experiments both horizontally and vertically, i.e., multi-chip and long experiment runtime respectively, was kept in mind. The `std::future` like interface for read

[4]https://en.cppreference.com/w/cpp/thread/future 2021-08-02

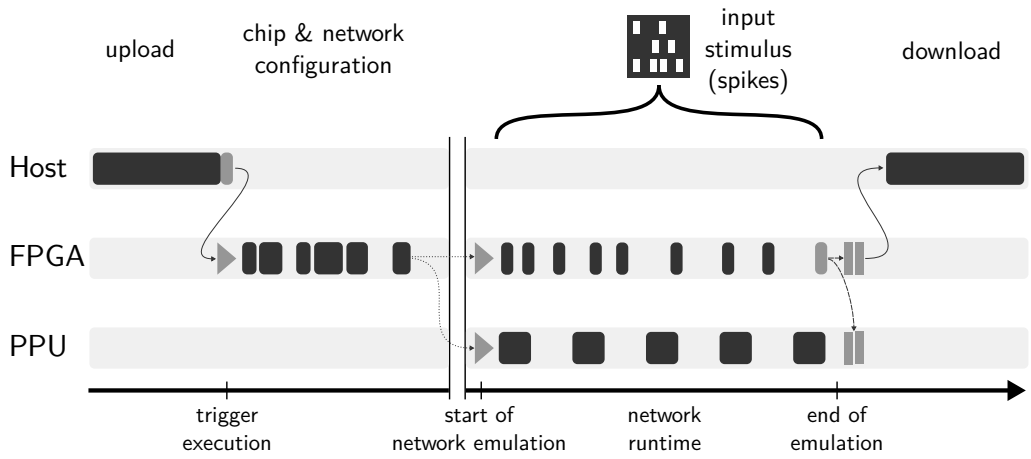


Figure 3.6: Schematic showing control flow of a playback program running concurrently with code on the embedded processor (e.g. plasticity rule). Black boxes denote activity. First, the host sets up communication with the control FPGA and then loads the constructed playback program onto it. Each playback program is appended with a special instruction denoting the end of the program (gray box). After the FPGA loads this instruction execution is automatically started. Initially the chip is fully configured including bringing components in a working state and the configuration of the neural network. As analog circuits may need time to settle a wait is inserted before experiment start (cut in time bar). Now PPU execution is started concurrently with release of spike input. Depending on the experiment FPGA and PPU execution need to be synchronized (not shown). During execution all responses like spikes or analog readout traces are recorded with time annotation by the FPGA. Experiment execution is finished once the special end of program instruction (gray box) is reached, also triggering upload of response data to the host machine. Modified from Müller et al. 2020a.

Listing 3.3: Example usage of playback builder pattern

```
PlaybackProgramBuilder builder

builder.write(NeuronConfigOnDLS(42), my_neuron_config)
builder.block_until(TimerOnDLS(0), Timer.Value(1000))
ticket = builder.read(SpikeCounterOnDLS(3))

program = builder.done()
run(connection, program)

read_count = ticket.get()
```

back data was chosen as it is suitable for large experiments where data cannot be stored fully on the FPGA or even host machines. It for example would support patterns like double buffering of playback programs to support quasi indefinitely running experiments.

The builder pattern approach for playback program generations allows for the definition of template builders. For example the most frequently used template encapsulates the base initial configuration of a chip. This framework also lends itself well to define templates for various categories of neural network experiments. Iterative experiments are prime candidates due to their repetitive nature. This reduces code duplication, improves readability of code as intend is clearly stated and most of al ensures proper initialization. Multiple of such template as well as experiment specific program builder can be concatenated.

Next it is reasoned to which extent this runtime control facilitate the previously described experimental usage modes. The definitions of three usage cases becomes vague regarding PPU utilization. For example the described experiment could be a plain batch job when one regards the PPU usage just as "on-chip" learning. But one could also categorize it as closed loop operation as the modification of network settings is done in a time coupled fashion. Nevertheless, all use cases are sufficiently covered with the exception of closed loop experiments where the host acts as sole experiment controller.

3.3.4 Hardware Database

Analog neuromorphic hardware setups are a conglomerate of components with varying properties and configuration. For reproducibility, it is necessary to track this composition and have them uniquely identifiably. To this end a hardware database was developed for BSS-1 functioning as the single source of truth for the state of all hardware setups [Koke 2017]. This database was separated out of the

BSS-1 software stack and extended for the various BSS-2 prototype setups over the course of this thesis. Exemplary information that are stored are IPs addresses, component serial numbers and chip versions. The underlying on-disk data format is *yaml* as it is human-readable and has already available implementations for C, C++ and Python. Some features are present by code examples in Listing 3.4. It shows a database entry of a BSS-2 HICANN-X cube setup(section 2.2.4). Cube setups always have at least two FPGAs present but not necessarily have a chip equipped. As components can change in time the combination of setup ID, FPGA number, chip ID are used to generate a unique identifier. In this example the identifier is `hxcube13fpga3chip31` and is used, e.g., for file names to store calibration data.

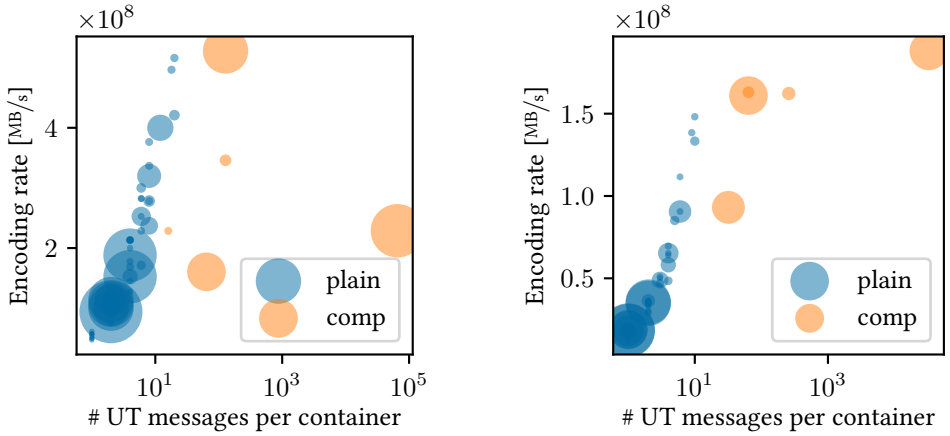
Listing 3.4: Example hardware database entry for a BSS-2 HICANN-X cube setup. Only one of the two FPGAs is equipped with a chip.

```
---
hxcube_id: 13
fpgas:
  - fpga: 0
    ip: 192.168.73.1
  - fpga: 3
    ip: 192.168.73.4
    handwritten_chip_serial: 31
    ldo_version: 2
    chip_revision: 2
usb_host: 'AMTHost13'
usb_serial: 'AFE25B471E002000'
```

3.3.5 Performance

In the following the performance of the hardware abstraction is measured to investigate its impact on overall system efficiency. Neuromorphic experiments are rather data heavy, especially the synapse matrix makes up a large portion of the configuration space. A typical use case are iterative jobs where only synaptic weight is adapted between network emulations. Therefore, low performance on writing and reading of data would be detrimental.

First, the encoding rate for read and write commands is measured. Encoding in this context describes more than `encode` function described in section 3.3.2. It encapsulates the entire process from creation of response tickets in case of reads to the generation of UT-messaged sequences. Figure 3.7 shows the encoding rate of all plain and composite (logical) containers with randomly filled data, where



(a) Write encoding speed: 304 MB/s (composite) 128 MB/s (plain)

(b) Read encoding speed: 158 MB/s (composite) 24 MB/s (plain)

Figure 3.7: Encoding rate of read and write operations on randomly filled plain and composite container. Measurements are taken by averaging the time for 10^2 random valid container values per type. Marker size represents a container type's portion of the complete system configuration in UT-message count multiplied by all coordinate values, i.e., instances of the container on the hardware. Average rates are weighted with container aforementioned container portion. Measurements conducted on Epyc compute nodes (appendix B.2). Modified from Spilger 2021.

the random data is constrained to be valid. For plain container both write and read encoding rate shows proportionality to number of encoded UT-messages, i.e., container size. This is expected as encoding many small containers has more overhead to few larger containers. This proportionality does not hold true for composite containers. Nevertheless, overall encoding speed is higher for composite containers again due to mitigating of overhead.

To provide comparable numbers the weighted average encoding rate is determined, where the proportion of a container type to overall configuration space is taken as weight. This result in 304 MB/s for composite and 128 MB/s for plain in case of write and 158 MB/s and 24 MB/s for read respectively. Comparing these values shows that read encoding is unexpectedly slower than write encoding. This is caused by the overhead of creating the placeholder data entries for the to be read back data. Next the time it takes to encode write and read commands for the whole chip is estimated. The number of messages which need to be encoded for a whole chip is 539 212 in case of write and 301 905 in case of read, where each message has size of 8 B. Write messages contain address and payload whereas

reads only contain the address. The number of readable entities on chip is larger as there are some components that are read-only, predominantly the correlation sensors of the synapse matrix. Taking these numbers then yields an encoding time of 14 ms for write and 15 ms for read respectively.

Now decoding performance is investigated with a similar measurement setup. Valid response UT-messages are generated with random payload and decoded resulting in fig. 3.8.

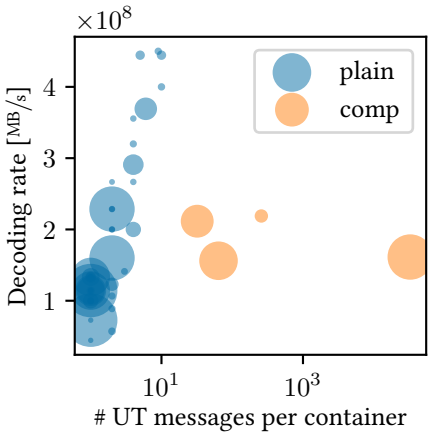


Figure 3.8: Decode rate of plain and composite container from random data. Measurements are taken by averaging the time for 10^2 random container values per type. Marker size represents a container type's portion of the complete system configuration in UT-message count multiplied by all coordinate values, i.e., instances of the container on the hardware. Weighted average decoding rate is 167 $^{\text{MB}}/_{\text{s}}$ for composite container and 129 $^{\text{MB}}/_{\text{s}}$ for plain container. Measurements conducted on Epyc compute nodes (appendix B.2). Modified from Spilger 2021.

Again decode rate is proportional to container size for plain container and not for composite container. Weighted average rates are 167 $^{\text{MB}}/_{\text{s}}$ for composite and 129 $^{\text{MB}}/_{\text{s}}$ for plain container. Estimation of decode duration for a whole chip results in about 14 ms.

Thus, encoding and decoding do not pose a bottleneck to sustained operation when compared against the bandwidth of 115 $^{\text{MB}}/_{\text{s}}$ of the FPGA-to-host connection (fig. 3.3). The communication speed only passed barely, therefore, if connection speed would be upgraded to 10 $^{\text{Gbit}}/_{\text{s}}$ the software could not keep up. However, the available bandwidth will be quickly consumed in multi-chip experiment anyhow, therefore not being an immediate issue which needs addressing.

Overall time spent for encoding and decoding of a whole chip configuration however can be an overhead for short iterative experiments in millisecond range.

Finally, performance of the hardware database is assessed. Measurement results are taken from section 4.1.5. Loading the database file is by far the most expensive operation taking 65 ± 5 ms. Execution time of querying of database entries however is below milliseconds and therefore negligible. Retrieving hardware configuration information is performed only once per experiment, thus overall impact on performance is limited.

3.3.6 Example Studies

The developed abstraction of the hardware components and its operation are already sufficient for hardware experts to conduct research on SNN learning strategies. Two studies that built on top of the hardware abstraction layer are shortly outlined.

Cramer et al. 2021 employs surrogate gradient training methods [Neftci et al. 2019] for deep as well as recurrent SNNs. To this end they integrate the hardware abstraction layer into the ML learning framework *PyTorch*. This enables training of the SNN topologies based on loss functions that incorporate the analog membrane potentials of the neuron circuits as well as their output spikes. Based on this training framework, they reached accuracy levels equivalent to software implementations of the same SNNs on various datasets and set new benchmarks for energy consumption, classification latency, and throughput.

Göltz et al. 2021 derived closed-form equations for the spike-time gradients in multi-layer SNNs. They utilize the same training framework as the previous study. Based on these equations, they optimized multi-layer networks employing a time-to-first-spike coding scheme, both in software simulations as well as on BrainScaleS-2.

3.4 Experiment Description

The previous section introduced the abstraction of the hardware as a flat collection of configuration and runtime control instructions. That level of abstraction is sufficient to formulate single-chip experiments by expert users as the configuration of the chip is still manageable (cf. section 3.3.6). However, in the light of multi-chip or even wafer-scale experiments a high-level abstract description of neural networks is pivotal. The user-facing API should be as general as possible permitting formulation of a wide range of network topologies. Goal of this layer is to provide such a high level experiment description and its automated translation to a valid hardware configuration.

First, a signal-flow graph-based approach providing structured description of hardware configuration and experiment flow is presented in section 3.4.1.

Building on top of this interface a high level description for network topology and its in and output is defined in section 3.4.2. The necessary complex process of placing and routing such an abstract network to a valid hardware configuration is explained in section 3.4.2.

The work described in this section was predominantly performed by Philipp Spilger during his master thesis [Spilger 2021] which was co-supervised by the author. The author contributed by validation and performance analysis of the implementation and helped by the integration into *PyNN* described in section 3.5.1. An in depth explanation of the different graph APIs and their implementation is therefore given in Spilger 2021. It also covers the facilitation of the non-spiking analog inference operation (cf. section 2.2.4) which is not subject of this thesis.

3.4.1 Signal-Flow Graph Description

The topology of neural networks is predominantly described in the form of graphs [Davison et al. 2009b; Abadi et al. 2015; Paszke et al. 2019b]. As neuromorphic devices aim to emulate neural networks their configuration and experiment flow naturally lend themselves to a graph-based representation. To this end a signal-flow graph description was developed for BSS-2 systems. On its lowest abstraction layer it provides a description for hardware network configuration, in particular the digital event routing. An experiment flow abstracting the lower graphs as single instructions is implemented on the next higher layer.

Hardware Abstraction Graph

First, a short introduction to signal-flow graphs and the utilized nomenclature is given. A signal-flow graph is a directed graph where the vertices represent units that process incoming signals in some way and propagate them according to the connected edges. Figure 3.9 shows the concept of a signal-flow graph by example of abstract mathematical operations.

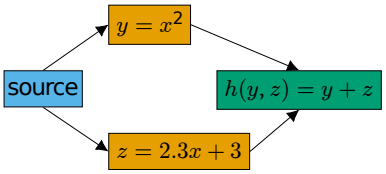
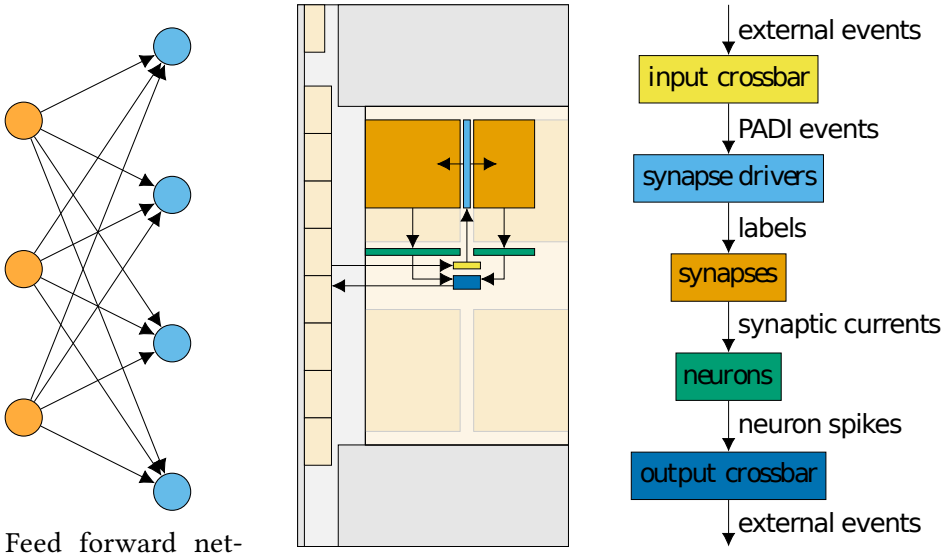


Figure 3.9: Signal-flow graph example. Following a source vertex, data is transformed by two parallel operations (y and z) at the top and bottom vertex. Their results are transformed in the right vertex (h). Modified from Spilger 2021.

In case of hardware abstraction vertices on the one hand represent individual chip components, e.g., neurons, synapses or routing crossbar. Edges on the other hand represent the interconnection thereof. For each vertex the allowed and expected neighboring vertex types and connectivity types are defined. This allows

to check validity of a constructed graph, e.g., unconnected or to many inputs or outputs from vertices. Such a graph then provides an explicit description of the signal-flow for neural network topologies on hardware. Figure 3.10 exemplary shows a graph representation of a feed-forward network realized on a BSS-2 chip. Besides ensuring correct network topology a graph-based representation



(a) Feed forward network

(b) Hardware Graph

Figure 3.10: Signal-flow graph representation of a feed-forward neural network on a BSS-2 chip. (a) shows the topology of a simple feed-forward network where neurons of the first layer are external events and the second layer are neurons to be realized on hardware. (b) visualization of chip components realizing the feed-forward network and its corresponding graph representation. Modified from Spilger 2021.

additionally provides access to graph optimization algorithms. It furthermore facilitates a direct visualization of the network topology.

It is important to note that the real-time evolution, e.g., insertion of spike trains or readout of membrane voltage, is not explicitly described by such a graph but is abstracted as properties of input and output vertices. This also applies to PPU operation concurrent to the real-time evolution, where it is merely represented as the content of its data and instruction memory.

Not all hardware configuration is encapsulated by this graph description as it is not directly relevant for the signal-flow representation. For example the configuration that controls the operation points of components like recording frequency of ADCs. This part of configuration is provided to and handled by the experiment flow described in the following section.

The implementation of this signal-flow graph description heavily utilizes the `boost::graph` library[5]. It defines an intuitive API for graph construction and provides a multitude off functionality like optimization algorithms or visualization of the graph.

Experiment Flow Description

Building upon the hardware signal-flow graph an abstract description for experiment flow is developed. The full composition of the hardware signal-flow graph is abstracted away to an individual run on hardware hereinafter called execution instance. Such an execution instance in turn is abstracted to a single vertex in the now explained experiment flow graph.

Main goal of this experiment flow description is providing a formulation for experiments that exceed resources of a single-chip instance. Such a description is imperative for multi-chip experiments but can likewise be utilized to time slice an experiment onto a single-chip instance. A feed forward network with three layers is taken as an example visualized in fig. 3.11.

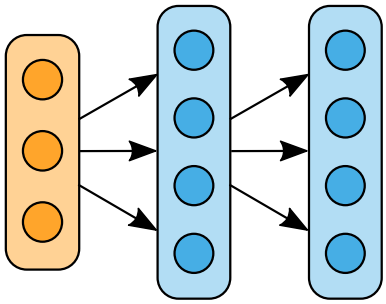


Figure 3.11: Partitioning of feed forward network. First layer (orange) is input from host machine. Second and third layer (cyan) are realized on neuromorphic hardware. As signal flows only in one direction, i.e., no recurrence, experiment can be sliced.

Assuming 512 neurons in each layer such a network needs resources of two chip instances as a single BSS-2 chip implements only 512 neuron circuits. Such a network could either be realized by executing concurrently on two interconnected chips or time sliced on a single-chip. This is possible for feed-forward networks where only the relative timing of events into and out from the layer is relevant. This does not hold true for example for recurrent networks. There the timing between layers needs to be precise for network functionality due to the continuous nature of the neuromorphic substrate. In such a case partitioning over multi-chip is only possible with sufficient low latency interconnection, i.e., latency in the same order of magnitude as neuron time constants.

Next generation and execution of hardware instruction sequences from the high level experiment flow description is presented. Figure 3.12 shows compilation and execution of an exemplary experiment flow graph on a single-chip instance.

[5]https://www.boost.org/doc/libs/1_75_0/libs/graph/doc/ 2021-08-06

The experiment graph defines the possible ordering of execution instances as each vertex can only be processed once its input vertices are executed. A valid ordering is guaranteed as the graph is forced to be acyclic. Processing of an execution instance is separated into three steps. First, it is converted to an instruction sequence (cf. section 3.3.3), then it is executed on hardware and finally its resulting data is analyzed and possibly provided to the next vertex. Pre- and post-processing are handled by conventional compute nodes whereas execution happens on neuromorphic hardware instances. Therefore, pre- and post-processing can be parallelized and performed concurrently to neuromorphic hardware execution.

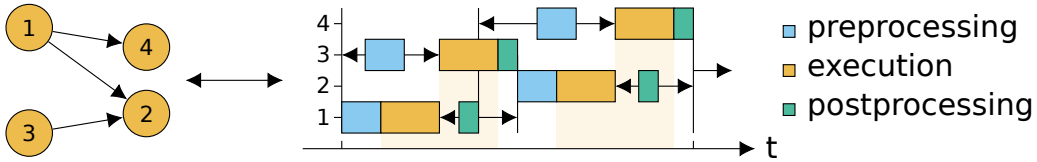


Figure 3.12: Compilation and execution of an experiment flow graph (left) into time sliced execution instances on a single neuromorphic chip (right). Preprocessing encapsulates generation of hardware instruction sequence which is followed by execution of said sequence and finalized by post-processing of read back data for the next execution instance. Execution happens on the same physical chip and is therefore executed sequentially, whereas pre- and post-processing can be executed concurrently. Taken from Spilger 2021.

Figure 3.13 demonstrates the impact of multi chip utilization. The same exemplary experiment graph as fig. 3.12 is executed however an additional chip instance is utilized. As vertex 1 and 3 are independent execution can be trivially parallelized.

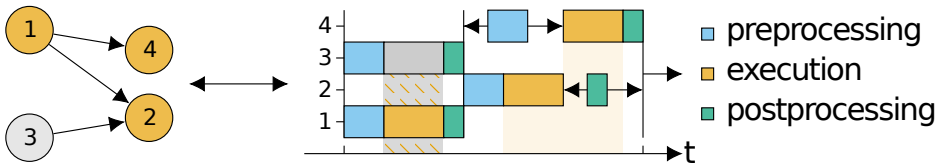


Figure 3.13: Compilation and execution of the same experiment flow graph as in fig. 3.12 however utilizing an additional neuromorphic chip. Execution instances 1 and 3 can now be executed concurrently resulting in shorter overall run time. Modified from Spilger 2021.

For the implementation of the experiment flow graph the `tbb::flow` library [TBB 2021] is utilized. It provides for example automated concurrent execution of the experiment graph.

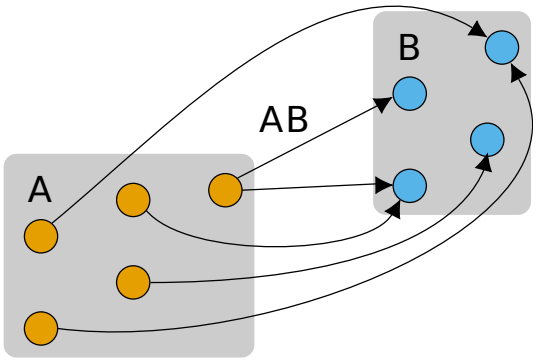


Figure 3.14: Abstract neural network graph. Grey boxes A and B represent populations, i.e., collections of neurons. The collection of connections between these populations is encapsulated as the projection AB. Modified from Spilger 2021.

3.4.2 Abstract Network Description

The signal-flow graph representation presented in the previous section provides a structured way of describing the hardware configuration for neural network experiments. However, formulating neural network experiments with this description still would require detailed knowledge about the routing capabilities of the neuromorphic hardware. To handle this issue a high level abstract description for neural network topology is defined. This abstract representation is then translated to a valid hardware signal-flow graph by a place and route algorithm.

Abstract Network Graph

Again a graph-based representation of the neural network topology fits naturally, as is realized by many neural network frameworks and simulators [Davison et al. 2009b; Bekolay et al. 2014; Hazan et al. 2018]. Inspired by these the topology of a neural network is described by so-called populations and projections between them. Populations describe collections of neurons whereas projections represent collections of connections between neurons of different or the same populations. Figure 3.14 illustrates such an abstract neural network graph.

There are three different types of populations for a BSS-2 chip. First, there are populations representing the on-chip neuron circuits. The other two types represent spike sources, namely off-chip spike arrays and on-chip spike generators. The former are arbitrarily timed spike events generated by the control host released during real-time execution. The latter produce regular or Poisson distributed events directly on-chip. Connectivity restrictions are explicitly checked by the graph description, i.e., spike source can only have output whereas actual on-chip neurons can have in and output connections. Connections are defined by their source, destination and a synapse type. This generic synapse type abstracts the various possible implementations on hardware. For example a signed synapse can be implemented by combining two hardware synapses where one is configured to be excitatory and the other to be inhibitory.

Place And Route

Translation of an abstract network to a valid hardware configuration depending on network topology size and complexity can be highly involved and computational costly. It is typically separated into two tasks. First is placement of abstract neurons to actual neuron circuits on hardware while considering hardware constraints, e.g., limited routing capabilities or defect chip components. Second is then finding a valid network routing of the specified connections between those neurons. Figure 3.15 illustrates the concept of placing and routing an abstract network graph to a possible hardware configuration. The bidirectional nature of

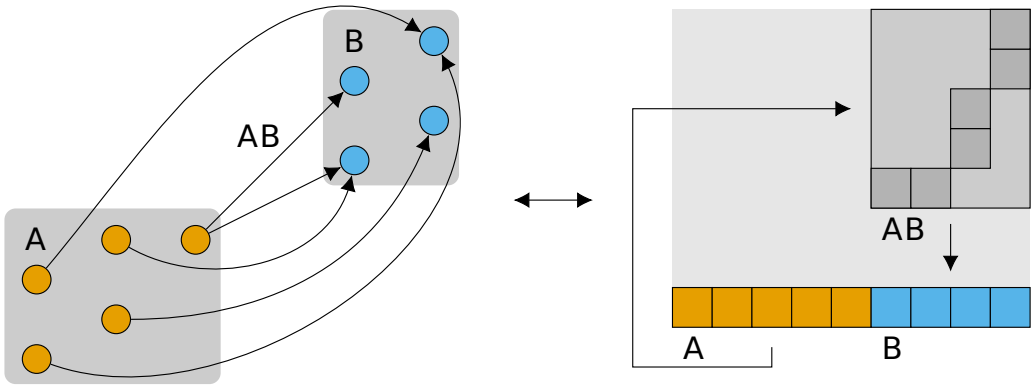


Figure 3.15: Abstract network graph (left) and potential realization on neuromorphic chip (right) Neurons of population A and B correspond to sets of hardware neurons (blue and orange squares). The projection AB is realized by appropriate setting of synapses in the synapse array. Modified from Spilger 2021.

this translation is facilitated to also provide a backwards translation from hardware entities to their abstract representation. This allows to provide experiment results to the respective abstract entities.

Different place and route algorithms are better suited for different network topologies. The actual implementation of a place and route algorithm is therefore separated in a free function which can be provided by the experimenter. The currently utilized place and route algorithm for single chips implements a greedy sequential strategy. First, all neurons are linearly placed on chip. Then all described connections are placed and routed in order of their creation. Consequently, no abstract knowledge about the network topology is exploited. A more detail explanation can be found in Czierlinski 2020.

In sum, the presented experiment description layer provides a population-based interface that allows intuitive and efficient formulation of neural network experiments. It achieves this by automatically handling translation of the high-

level graph representation to the specifics concerning signal-flow and its corresponding hardware configuration. This lays the foundation for large-scale multi-chip experiments through its generalized and scaleable design.

3.5 Modeling Wrapper

The intended goal for the BSS-2 neuromorphic systems, as for most other neuromorphic hardware and simulators, is to be accessible to as many researchers as possible. Accessibility to systems from the software point of view is defined by their supported APIs. Many neural simulators and neuromorphic chips provide similar capabilities and therefore similar interfaces, i.e., description of network topology or parameters of the neuron model. To reduce the burden on researchers to learn multiple similar but different APIs common interfaces for neural network experiments where developed, for example *PyNN* [Davison et al. 2009b], *nengo* [Bekolay et al. 2014] or *BindsNET* [Hazan et al. 2018]. It furthermore improves reproducibility and comparability between different simulators. Supporting such common interfaces also potentially lowers the threshold for researchers to adopt novel devices. Furthermore, it opens access to many tools and frameworks developed by the research community. However not all use cases can be covered by a single interface, leading to multiple different interfaces that excel at different tasks. Therefore, wrappers for the prevalent APIs *PyNN* [Davison et al. 2009b] and *pytorch* [Paszke et al. 2019a] were developed. The former provides a Domain Specific Language (DSL) for describing SNN models, their topology and input with the main focus being computational neuroscience. The latter is a widespread machine learning framework which in recent years grew in popularity also in the neuroscience community [He 2019]. The authors direct contribution to the *PyTorch* wrapper is minimal and is therefore only outlined.

3.5.1 PyNN

PyNN is a simulator-independent DSL for neural network modeling. Its goal is to provide a common interface for various software simulators as well as neuromorphic hardware so that researchers can utilize them while only needing to define their network models once. In the following *PyNN*'s interface is shortly outlined. It is similar to the abstract network graph interface introduced in section 3.4.2 as *PyNN* was its original inspiration. The topology of a neural network is described by populations and projections between them. Populations are collections of neurons of the same model type, e.g., the LIF model (cf. section 2.1.1). The individual neurons in a population can share the same configuration or have individually sets of model parameters, like leak or threshold. Projections are collections of

connections between neurons of different populations. Their properties are defined by synapse and receptor type. Synapses at least define a weight but can also represent types of plasticity. Again parameters can be set for the whole projection or for each individual connection. *PyNN* provides connector classes utilized by projections to conveniently describe the connectivity between populations. For example the `FixedProbabilityConnector` allows to construct a projection where each individual connection is created with a specific probability. Thereby providing an API that allows to describe all possible network topology types like feed-forward networks or recurrent networks.

PyNN defines a standard set of neuron, synapse and plasticity models with the intent that back-ends implement support for these standard sets where applicable. Recording of different neuron observables like emitted spike train or membrane potential can be activated for individual neurons and populations. After the network topology is set up its time evolution is triggered by a `run` instruction for a certain amount of time. After such a run network responses can be read out, analyzed and acted upon. For example topology or parameters can be modified before time evolution is continued by a further `run` call.

Each back-end needs to implement a translation between the *PyNN* interface and its own while hiding simulator specifics as much as possible. Especially for neuromorphic hardware back-ends this is not always possible or even desirable to fully exploit the emulator capabilities. Back-ends can to this end define their own neuron or synapse models as well es extend or modify the API. As this defeats the purpose of *PyNN* as a unified simulator-independent front-end to some degree it should only be considered if absolutely necessary. In the following the developed BSS-2 *PyNN* back-end called `pynn.brainscales` is explained with special focus on parts where chosen design deviates from the standard *PyNN* API or workflow. The primary task of `pynn.brainscales` is to provide a translation of the *PyNN* API to the network description layer described in section 3.4.

BSS-2 PyNN Back-End

A short hypothetical experiment illustrating *PyNN* workflow is given in listing 3.5. In this experiment some input, e.g. spike encoded image, is give to a fully connected feed forward-network. Based on the read back output of the previous run weights are updated and the network is repeatedly emulated on hardware for a number of epochs. First, the external input population and its spike train are created. Then the on-chip population consisting of the custom `HXNeuron` type is created. Subsequently, both populations are connected via an all-to-all projection. The network is then iteratively run and outcome of each run is saved. Finally, the accumulated results are analyzed.

Now aspects relevant for the usage and implementation of `pynn.brainscales`

Listing 3.5: Example pyNN experiment flow

```
pynn.setup()
spiketimes = [0.1, 1, 2, 3.5]
ext_stim = pynn.Population(
        256,
        SpikeSourceArray(spike_times=spiketimes))
hw_pop = pynn.Population(
        256,
        pynn.cells.HXNeuron(**neuronparams))
hw_pop.record(["spikes"])
proj = pynn.Projection(
        ext_stim, hw_pop,
        pynn.AllToAllConnector(),
        synapse_type=StaticSynapse(weight=63),
        receptor_type="excitatory")

result = []
for _ in range(number_of_epochs):
    pynn.run(5)
    spikes = hw_pop.get_data('spikes')
    result.append(analyse_result(spikes))
    proj.setWeights(weight_update(spikes))

process_result(result)
pynn.end()
```

back-end are presented.

Standard *PyNN* neuron and synapse models operate with biological units, e.g., µS for weights or µF for neuron capacitance. At time of writing there does not yet exist a suitable calibration framework that would allow for automatic translation of the full BSS-2 hardware parameters, e.g., unit-less integers, to the biological domain. Therefore, a custom neuron type is developed that directly maps applicable parameters defined by the logical containers (cf. section 3.3.2). Directly reusing lower level containers provides a rudimentary interface as a first step for researchers familiar with the hardware specifics. It also automatically provides the same container features like early range checks of parameters.

One of the key differences between software simulator back-ends and BSS-2 is its continuous time evolution. Where in software simulation the complete state of the simulator can simply be frozen in-between `run` calls this is not possible on hardware. Therefore, in the current implementation a `run`-call always corresponds to an isolated hardware execution. Thus conditions at start of each experiment are undefined, i.e., in whatever state the neuromorphic substrate is at that moment,

typically idle at resting potential.

Another task needed for neuromorphic hardware, as well as multi-node simulations, is place and route of the abstract *PyNN* network to the hardware constraints. How this issue is approached algorithmically is explained in section 3.4.2, with the question now being at which point place and route should be executed in the context of a *PyNN* experiment. One of the requirements is that there should be no "magic knowledge" required, e.g., if certain types of populations would need to be created before others. However, researchers still need to be able to modify the place and route behaviour to better utilize the hardware by providing knowledge about the network structure. Especially for multi-chip experiments this can be valuable [Plank et al. 2021]. This implies in general that map and route are only performed after the full network topology is defined. However, current placement strategy is a simple direct assignment from *PyNN* neurons to hardware neurons on creation of a population. Experimenters can take advantage of the direct placement by providing an arbitrary permutation of the assignment to investigate behaviour of individual neuron circuits even on this high level of abstraction.

The direct placement approach furthermore enables injection of a rudimentary on-demand calibration that provides hardware parameters for a desired operation point. Such a calibration is generated before *PyNN* utilization and provides its result in a set of vectors of coordinate container pairs. The `pynn.brainscales`-API was enhanced to enable injection of such a calibration result to automatically set the parameter values of neuron populations accordingly. This allows researchers to modify parameters relative to the calibration result.

Routing, however, is only possible after the network topology is completely defined and is therefore performed in the `run` call. As routing can be a time-consuming operation (cf. section 3.6.2) measures are taken to reduce overhead for iterative experiments where the network topology is not changed in-between runs. Primarily this is implemented by checks if network topology or parameters changed that would warrant a renewed routing. For example abstract synaptic weights that are larger than individual synaptic circuits weights are supported by combining multiple synapses to larger virtual ones. This weight virtualisation influences routing and needs to be handled. One approach is to statically assign the maximum needed synapses per virtualised weight which potentially wastes a lot of resources but does not require rerouting with weight changes. The other approach is to re-route for each weight change which depending on complexity of network can be time-consuming. These again are trade-offs which the user should be able to specify by choosing an appropriate synapse type abstracting such behaviour.

A performance and scaling analysis of the complete software stack on the basis of *PyNN* is conducted in section 3.6. Section 3.7 presents a Sudoku solver

network utilizing the developed *PyNN* back-end.

Outlook

There are several features and improvements that should to be added to the current `pynn.brainscales` implementation to further enhance its usability.

One key feature is the support for standard cell types defined by *PyNN* which facilitate back-end agnostic experiment description. To this end design and implementation of a framework that joins calibration and place and route strategies is essential. However yet, an intermediate representation working in hardware SI-units is also desirable especially for expert users closely familiar to it.

Another aspect that can be improved is the behaviour of `pynn.run`. As explained in the current implementation each call corresponds to a concrete hardware execution run. However, this is not standard behaviour and also introduces overhead for iterative calls. One possible approach is to compose a larger experiment over multiple `run` calls without execution. Execution could then for example be automatically triggered when trying to access to be read output data.

Currently, the on-chip PPU is only supported as a quasi black-box algorithm that is executed in parallel to the network experiment defined in *PyNN*. A more native support of PPU capabilities, in particular for plasticity rules, is desirable. Its automated incorporation is a challenge as code needs to be generated and compiled that implements a desired plasticity algorithm. A first step could be to provide parameterized pre-compiled plasticity algorithms abstracted as synapse types in *PyNN*. A more advanced approach would be to support arbitrary rules that can be defined by the experimenter as mathematical equations. There the challenge lies in an automated code-generation for the PPU [Blundell et al. 2018].

A relative new pursued concept at least in neuromorphic hardware are multi compartment neurons (cf. section 2.1.3). Therefore, a new version of *PyNN* incorporating description of such multi compartment neurons is still in development stage[6]. However, to utilize the multi compartment capabilities of the BSS-2 chip an early adoption of this newer version seams valuable.

3.5.2 PyTorch

With the advent of ML techniques in computational neuroscience, e.g., adaptations of the back-propagation algorithm, so came the desire to harness the wealth of ML-tools. One such tool is the widespread *PyTorch* framework [Paszke et al. 2019a]. Its key features are the `autograd` functionality which is an automatic differentiation engine used for gradient descent-based network optimization

[6] https://github.com/NeuralEnsemble/PyNN/tree/mc 2021-09-13

strategies as well as the native support of GPUs for efficient computation. One of the main reasons for their success is the plethora of utility features they provide on top of performant compute graph compilation and execution engine, that allow for rapid prototyping. Examples are convenient importing of common training datasets like MNIST [LeCun et al. 1998] or Imagenet [Krizhevsky et al. 2012].

To take advantage of the *PyTorch* framework a thin software layer called *hxtorch* was developed. Its main task is translating between the *PyTorch* world and the BSS-2 software stack, in particular the experiment description layer. Compared to previous ML inspired research performed on BSS-2 (cf. section 3.3.6), *hxtorch* allows to fully utilize the abstract network description capabilities to support arbitrary network topologies within *PyTorch*.

hxtorch was established to facilitate the non-spiking operation (cf. section 2.2.4) in Spilger 2021 and was subsequently utilized for SNN experiments in Arnold 2021.

3.6 Full Stack Analysis

The previous sections described the BSS-2 experiment software stack developed during the course of this thesis. Now its overall performance especially its scalability in the wake of future multi-chip experiments is investigated. Again, to fully exploit the hardware speedup software overhead should be as small as possible.

First, the scaling with hardware execution time, more specifically with spike event readout is investigated. Then overhead of abstract experiment description and transformation to a valid hardware configuration is examined. Finally, their impact on the experiment workflow is discussed.

All measurements are performed with the repository state given in appendix B.1.2 and on *RyzenHost* nodes (cf. appendix B.2) if not explicitly stated otherwise.

3.6.1 Scaling with Run Time

There are various experiment scenarios where large amounts of data are aggregate on hardware and sent to host for analysis. Such data could be spikes, membrane voltage traces or synaptic weights. Increasing the execution time on hardware for such experiments leads to more output data. Therefore, the following section investigates how the software stack scales with amount of read back spike data. Spike data is chosen as it can be the largest source of generated data.

Spike Loss

Recording of experiment data like responses of read instructions and, predominantly, spike data is facilitated by a readout buffer memory on FPGA. It auto-

matically starts sending recorded data on experiment start. The current FPGA implementation provides a buffer size of 32 MB. The spike throughput between chip and FPGA (≈200 MHz) [Karasenko 2020] is an order of magnitude larger than the throughput between FPGA and host (≈13 MHz). This potentially leads to spike loss if event rate is larger than the FPGA-to-host bandwidth for a prolonged time.

To get a better understanding of this dynamic read back spike sequences for increasing on-chip spike rates are analyzed in fig. 3.16. Utilizing the *PyNN* stack a single regular input spike train with rate of 1 MHz is connected one-to-all to an increasing number of neurons which are read out. Neurons are configured in so-called bypass mode meaning an incoming spike directly results in emitting an output spike, i.e., bypassing the analog neuron behaviour. This yields reproducible results independent of fixed pattern noise of different chip setups. With this spikes are basically duplicated for each neuron and looped back to the host. For each neuron count an experiment run of 10 s is conducted, yielding $1 \cdot 10^7$ spikes per neuron. The resulting spike train is read out and segmented into logarithmically increasing time slices for which the spike rate is determined. As expected a decrease in spike count and therefore rate is observed for increasing run time and neuron count.

To provide better context of these results the relevant theoretical limits are calculated. The maximum theoretical run time t_{max} for n_{neuron} neurons where no spike loss occurs is given by:

$$t_{max} = \frac{m_{fpga}}{n_{neuron} \cdot f_{neuron} \cdot m_{spike}} = \frac{32\,\mathrm{MB}}{60 \cdot 1\,\mathrm{MHz} \cdot 4\,\mathrm{B}} = 133\,\mathrm{ms}$$

where m_{fpga} denotes the available FPGA readout memory, f_{neuron} the spike frequency and m_{spike} the memory requirement of a single spike event. This assumes that no other events are stored in the FPGA memory and spikes are optimally packed section 2.2.4. The theoretical maximum number of continuously firing neurons where the FPGA-to-host bandwidth $b_{fpga2host}$ is sufficient indefinitely is given by:

$$n_{neuron} = \frac{b_{fpga2host}}{\cdot f_{neuron} \cdot m_{spike}} = \frac{115\,{}^{\mathrm{MB}}/_{\mathrm{s}}}{1\,\mathrm{MHz} \cdot 4\,\mathrm{B}} = 14$$

Value for $b_{fpga2host}$ is taken from fig. 3.3.

Looking at the measurement no spike loss occurs until at least 100 ms for all neuron counts. Likewise, spike rate stays at maximum for 16 neurons and below over the complete 10 s run time. For neuron counts above 20 spike loss occurs later than the plain theoretical limit given by FPGA buffer size. This makes sense as events are already being sent to host from the start.

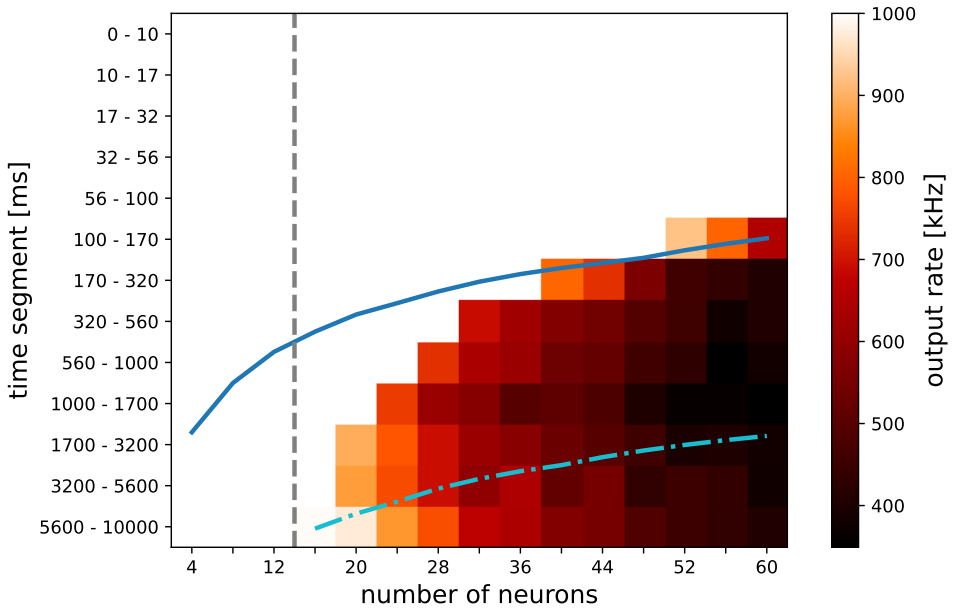


Figure 3.16: Spike loss due to limited FPGA readout buffer memory and bandwidth. Regular external spike source of 1 MHz stimulates an increasing number of on-chip neurons. Each neuron is configured to produce a spike for each input event. Spike output is recorded for 10 s and segmented in logarithmically increasing time intervals. Meaning, each column corresponds to a single execution on hardware. If output spike count exceeds the FPGA readout memory size spike loss occurs if the rate is higher than the FPGA-to-host bandwidth. Solid blue curve shows the theoretical maximum run time until the FPGA memory is full for the currently implemented 32 MB readout memory. Dash-dotted cyan curve represents this limit for 512 MB which is physically available on the system but not yet utilized. Dashed grey line shows minimum neuron count n=14 where bandwidth is sufficient indefinitely.

Large Spike Experiment Run Time Analysis

Based on spike loss results a full stack spike scaling experiments is performed. First, the same minimal *PyNN* network experiment of reading out $n = 12$ continuously firing neurons is profiled, resulting in fig. 3.17.

It covers the execution of the entire python experiment script. Illustrated are memory consumption and run time segments of the respective tasks carried out by the software stack. Table 3.1 provides short descriptions of the individual tasks. Memory consumption is obtained utilizing the python tool *mprof*[7]. A Population size of 12 is chosen as no spike loss occurs leading to better comparability for varying run time in later scaling comparison.

Execution Segment	Description
python_import	Import of all needed python libraries including `pynn.brainscales`
setup	*PyNN* setup call including loading of calibration data
build_network	Creation of *PyNN* network
build_graph	Build and route of grenade experiment description
run_on_hw	Sending, executing and receiving playback program
process_response	Process response data, e.g., decode packed spikes to individual spike events
extract_spikes	Convert spikes into experiment description format type
sort_spikes	Spike train needs to be fully sorted as hardware returns only partial ordered sequence
filter_spikes	Filter out events with time stamps outside of experiment range
grenade_run	Remaining overhead from grenade execution
assign_spikes	Converts spike train from hardware container to a map of abstract neuron IDs and floating point time stamps.
get_spikes	Request spikes from *PyNN* API
end	Destruction of *PyNN* network

Table 3.1: Short descriptions for experiment execution segments of fig. 3.17.

Initially an overhead of importing python libraries and loading up the *PyNN* environment can be seen. This needs to be performed once for an experiment and is independent of the network itself. As expected creation of the network topology, conversion to a signal-flow graph representation and routing are only

[7] https://github.com/pythonprofilers/memory_profiler 2021-09-06

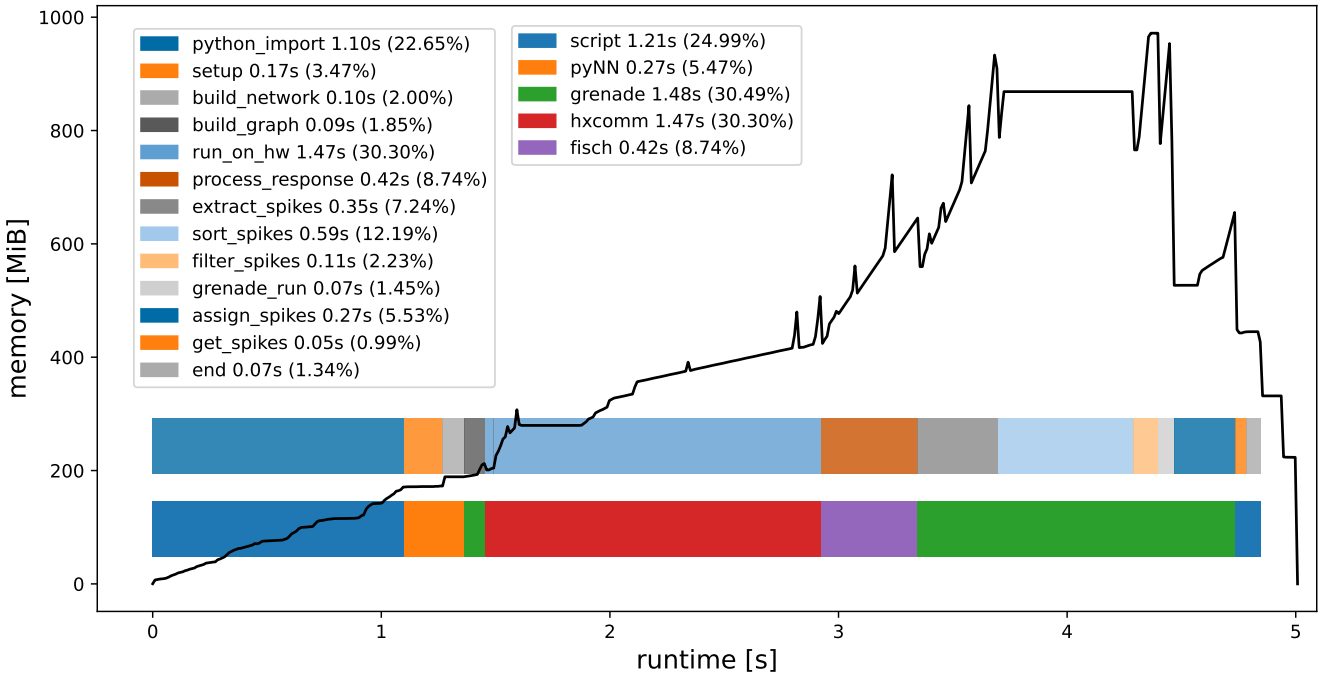


Figure 3.17: Run time analysis of a *PyNN*-based experiment with large spike count. A Population of 12 neurons in bypass mode is excited by a regular spike train with frequency of 1 MHz. Network is emulated for 1 s on hardware resulting in $1.2 \cdot 10^7$ spike events. Black line graph shows memory consumption during execution. Horizontal bars represent various segments performed during experiment execution (top) and the corresponding software layer (bottom). Annotations in legend present individual run time of steps and percentage of overall run time. Individual segments are described in table 3.1.

a few milliseconds due to the simple network structure. Most time is spent in executing the experiment on hardware and then processing the $1.2 \cdot 10^7$ spikes. Run time on hardware is about 1.5 s which includes initial configuration and sending of the input spike train. Where the former takes about 125 ms and the latter roughly $\frac{32\,\mathrm{MB}}{115\,\mathrm{MB/s}} = 278\,\mathrm{ms}$. The former is obtained by averaging execution of 100 empty runs, meaning run time of 0 seconds and only one population without connections. Peak memory consumption is reached during sorting where three different versions of the spike sequence are held. A spike event has different representations and therefore varying storage consumption on the various abstraction layers. For FPGA-to-host communication spikes are packed and can take up 4 B or 8 B. In upper layers spike events store information as separate variables namely source address label, FPGA time stamp and chip time stamp. Then closest fitting types are used resulting in 4 B+8 B+8 B=20 B per event. For $1.2 \cdot 10^7$ events this leads to 96 MB and 240 MB storage consumption respectively. These numbers agree with the measurement results.

Memory profile and run time are the outcome of various optimization efforts. For example care is taken that few temporary copies of spike sequence are created or APIs are aligned to prevent overly costly conversions. Still, there is room for improvement, for example, the spike types could be unified over most layers to reduce for example overhead in the `extract_spikes` step. Another example is sorting of the read back spike sequence. Sorting is necessary due to de-serialization and serialization of parallel communication channels between chip and FPGA. This can lead to flips in ordering, however time stamps are preserved allowing later sorting. The resulting spike sequence is therefore almost sorted, which is taken into account for choosing an appropriate sorting algorithm. For example the default sorting algorithm of the standard library[8] takes 1042 ± 17 ms for $1.2 \cdot 10^7$ spikes averaged over 10 iterations. Alternatively, the *boost spinsort* algorithm[9] which is optimized for almost sorted sequences takes 583 ± 19 ms.

Execution Time Scaling

Next the previous experiment is repeated with increasing hardware execution length to investigate scaling with increasing amount of read back spike events. Figure 3.18 shows overall software run time and peak memory consumption depending on the requested hardware execution time. Constants offset of the `python_import` step is subtracted from memory consumption and run time respectively for each run.

[8] https://en.cppreference.com/w/cpp/algorithm/sort 2021-09-06

[9] `https://www.boost.org/doc/libs/1_68_0/libs/sort/doc/html/sort/single_thread/spinsort.html` 2021-09-06

Overall run time and memory consumption are constant for hardware execution time below 100 ms. There overhead is dominating and mainly caused for example by configuration of all chip components. Especially the configuration of analog parameters which requires a subsequent analog settling time of 100 ms causes a large portion of the overhead. Memory consumption as well as overall run time start to increase after 100 ms hardware execution time. After 1000 ms both memory and overall run time scale linearly. This is expected as spike-event related operations have a runtime complexity of O(n). This includes the sorting algorithm given the spike data is almost sorted.

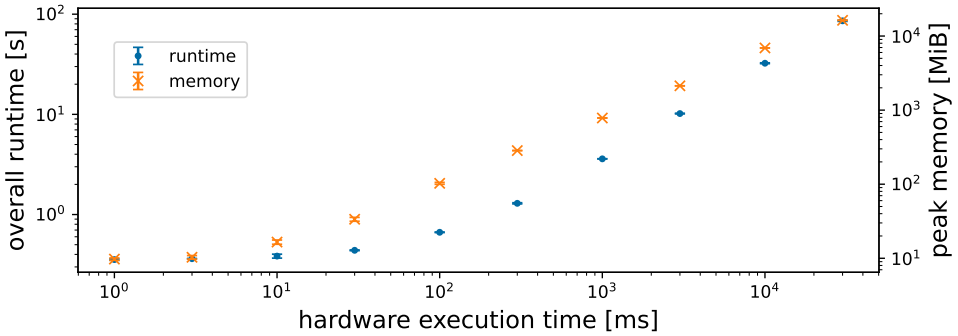


Figure 3.18: Scaling of experiment run time (left axis) and peak memory consumption (right axis) depending on requested hardware execution time. Utilizes same loop back experiment as fig. 3.17 with varying run time averaged over 5 iterations each. Constants offset of `python_import` step is subtracted from memory consumption and run time.

Multi-Chip Readiness

Finally, readiness for execution of experiments on multiple-chip setups is investigated. As there is no true multi-chip support yet available, performance for multi-chip setups is estimated by parallel execution of the same experiment on independent chips. This estimate is relatively accurate for segments of hardware execution and spike handling as these would be independent even for real multi-chip experiments. However, network description as well as routing would scale differently.

Two cases of interest are investigated. First, the scaling of run time and memory consumption for increasing amount of parallel executions is measured. There the goal is to verify that the software can handle concurrent executions without too much overhead. In the second case full utilization of the 10 $^{\text{Gbit}}/_{\text{s}}$ Ethernet connections during runtime is verified.

For the first case the same spike loop back experiment as described in the previous sections (cf. fig. 3.17) is executed in parallel for increasing number of chips. This is facilitated by starting multiple synchronized processes via MPI on the same host. A Synchronisation point is set before the `pynn.setup` step. Figure 3.19 shows the resulting run time and peak memory consumption.

As expected memory consumption shows a linear increase. Regarding run time a steady increase from 4 s to 5 s is observable. One possible cause for this increase are bandwidth limitations between CPU cores and memory during spike processing. Another limiting factor for ideal parallelization is the 10 $^{\text{Gbit}}/_{\text{s}}$ bandwidth between experiment host and the neuromorphic hardware setups which in turn are connected each with 1 $^{\text{Gbit}}/_{\text{s}}$. Therefore an increase in run time would be expected for more than 10 setups, assuming full bandwidth utilization. However, the experiment parameters are explicitly chosen such that the bandwidth for each hardware setup does not exceed 1 $^{\text{Gbit}}/_{\text{s}}$. This is done as exceeding the bandwidth would result in spike loss and therefore influence scaling.

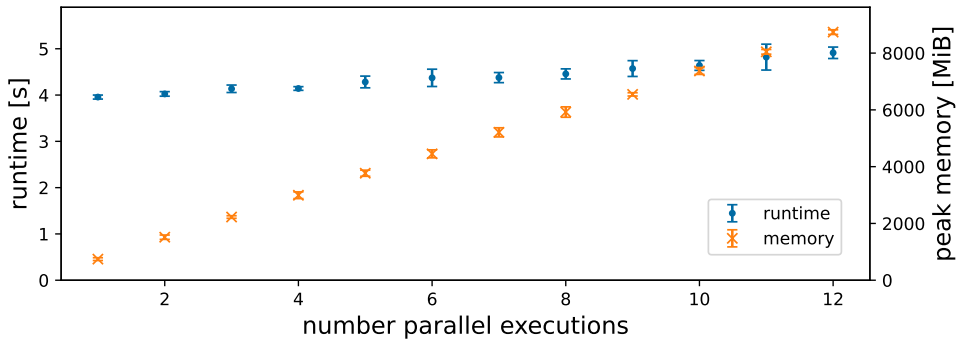


Figure 3.19: Scaling of spike processing time (left axis) and peak memory consumption (right axis) depending on number of parallel executions. Utilizes same loop back experiment as fig. 3.17 with varying run time averaged over 10 iterations each. Processes are synchronized via MPI before `pynn.setup` step, which marks start of run time. Low enough spike activity ensures no spike loss and therefore comparability for increasing parallel jobs. End of run time is end of last finished process. Used setups are randomized for each execution. Measurement performed on *EpycHost*, see appendix B.2.

For the second case, again, the loop back experiment described in fig. 3.17 is utilized. Run time and neuron count are set to 1 s and 24 respectively to be on the edge of no spike loss, see fig. 3.16. The experiment is executed concurrently for increasing number of processes. The read back spike count is averaged over all processes resulting in fig. 3.20.

For the chosen parameters each experiment should return about $2.4 \cdot 10^7$

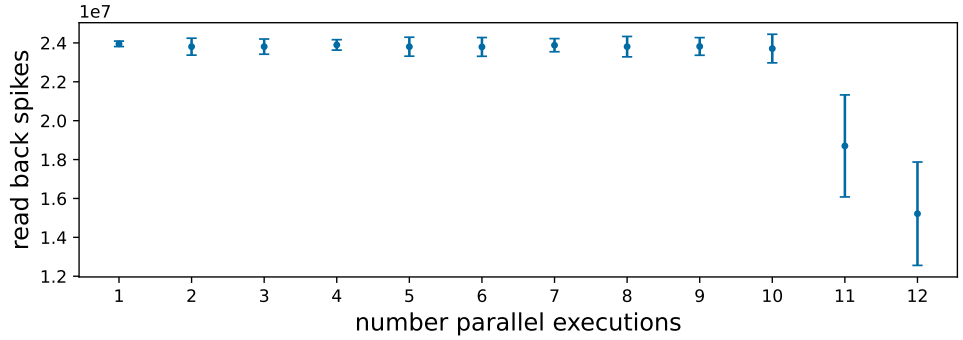


Figure 3.20: Utilization of communication back end for parallel experiments. The same loop back experiment as fig. 3.17 is run concurrently for increasing number of processes. Execution run time is set to 1 s and neuron count to 24 neurons to be at edge of spike loss fig. 3.16. 10 $^{\text{Gbit}}/_{\text{s}}$ Ethernet FPGA to host connection saturates for more than 10 concurrent experiments. Processes are synchronized via MPI. Hardware setups are randomized for each of 10 iterations. Measurement performed on *EpycHost*, see appendix B.2.

spikes with some expected spike loss. This is true as long the individual 1 $^{\text{Gbit}}/_{\text{s}}$ connections due not saturate. As expected only with 11 and 12 concurrently running experiments significant spike loss occurs. This shows that the software stack can handle concurrent hardware executions efficiently.

3.6.2 Scaling with Network Topology

Next performance and scaling dependent on network size and complexity are investigated.

The performance impact of network description and routing naturally depends on the network topology itself. A fully connected feed-forward network topology is chosen for performance evaluation as it provides the highest connection density. However, there are multiple ways an abstract network can be described, e.g., many small populations or few large ones. Figure 3.21 illustrates how the same feed-forward network can be described with varying granularity of populations. This allows to investigate overhead introduced by scaling of network topology graph.

Abstract Network Graph

First, impact of construction of the abstract network graph is examined. Figure 3.22 shows time and memory consumption for construction depending on population granularity.

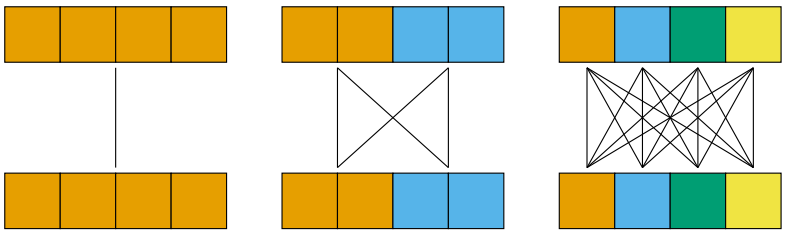


Figure 3.21: Illustration of network topology to benchmark the abstract population-based description. All sketches represent the same fully connected feed-forward network, however with increasing population granularities. Lines represent all-to-all projections between the individual populations. However, the total amount of connections, which are not illustrated, stays the same. Left: one population (with all four neurons) per layer; Middle: two populations (with two neurons each) per layer; Right: four populations (with one neuron each) per layer.

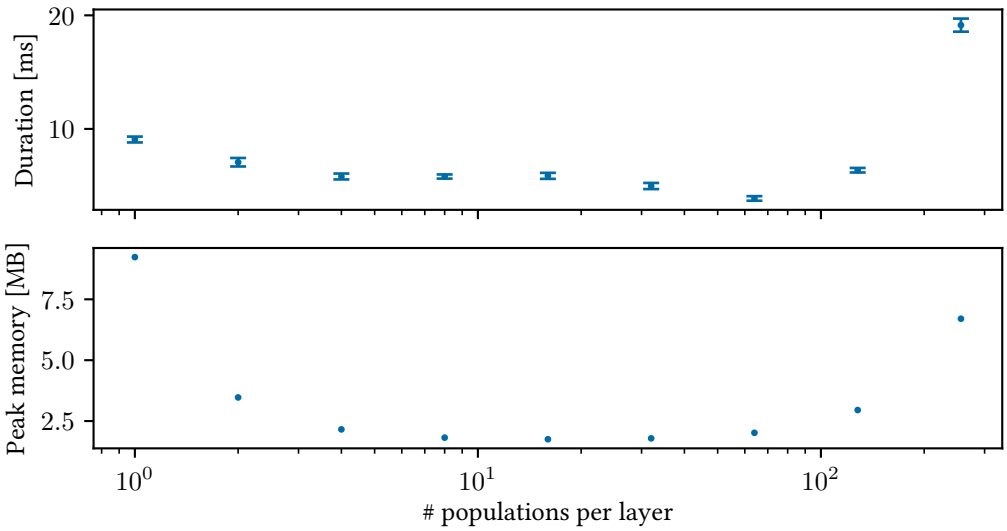


Figure 3.22: Run time (top) and memory consumption (bottom) of an abstract network description. A fully connected feed-forward network with 256 neurons in each layer is constructed with increasing granularity of populations as described in fig. 3.21. Modified from Spilger 2021.

Time consumption initially decreases with more granular description. Then, after 64 populations per layer it increases quadratically. For many populations per layer quadratic time expenditure is expected due to the quadratic growth of projection count. A similar progression can be seen for peak memory consumption, however, there initial peak memory allocation is actually the largest with about 9 MB. This large memory requirement for large populations does not stem from the resulting graph but during construction of it. During construction, connections of each projection are stored in a memory vice wasteful `std::set` object. For one population per layer only one large set is created in contrast to many populations where many sets are created for which memory is successively allocated and freed. Likewise, this leads to longer execution times.

Network Size

Next run time overhead of the network description via `pyNN.brainscales` is investigated. This includes description of networks in *PyNN*, translation to abstract graph representations, subsequent routing and its eventual conversion to a hardware signal-flow graph. First, run time scaling with network size is examined in fig. 3.23. Again a fully connected, i.e. all-to-all, feed-forward network is chosen due to its high connectivity and therefore routing complexity. Population and projection count stay constant (two and one respectively), whereas connection count increases quadratically. Run time consumption stays constant in order of 10 ms below population sizes of 10. It then transitions roughly into quadratic scaling as is expected.

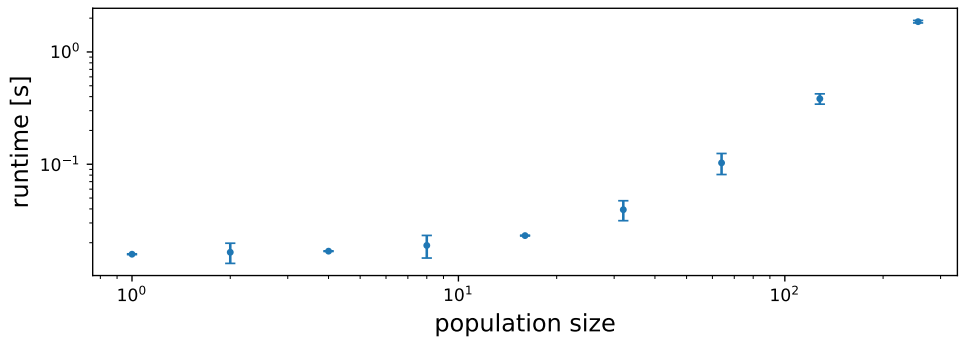


Figure 3.23: Full stack run time scaling with network size. All-to-all connected feed-forward network is described in *PyNN*, routed and translated to hardware signal-flow graph. Quadratic scaling due to all-to-all connectivity. Averaged over 10 iterations.

Network Description Granularity

Next the impact of network description granularity is investigated. To this end the same feed-forward network description with increasing population granularity as described in fig. 3.21 is utilized. Figure 3.24 shows run time and memory consumption of the abstract *PyNN* network graph construction and translation to hardware constraints.

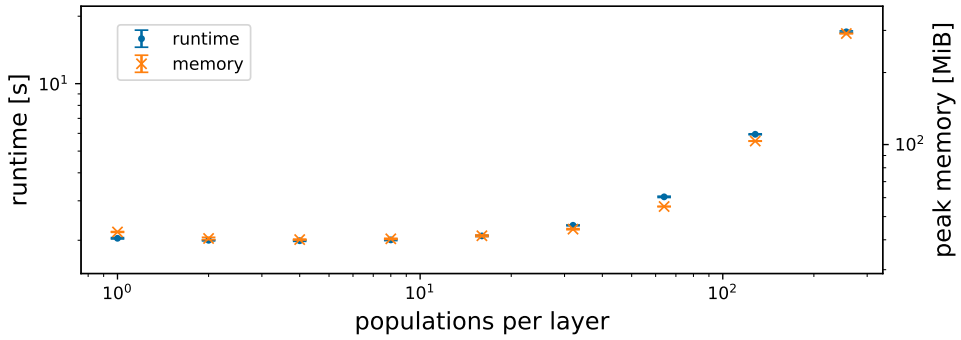


Figure 3.24: Full stack run time (left axis) and memory consumption (right axis) for varying description granularity of an all-to-all connected feed-forward network (cf. fig. 3.21). Averaged over 10 iterations.

Again, quadratic scaling for increasing number of populations per layer is expected. A run time of 1 s for coarser granularity agrees with results from scaling with network size (cf. fig. 3.23). For finer granularities, i.e., many small populations with many projections, run time reaches 20 s.

To investigate impact of the individual steps a detailed run time analysis of a single iteration run is performed in fig. 3.25. First, most notably, overall run time is dominated by construction of the network description in *PyNN*, especially for a high number of projections. Secondly run time of routing stays constant at about 800 ms independent of description granularity. This is expected as routing operates on the realized connections which stay constant for all description granularities. All other steps as well as memory consumption scale with description granularity. Routing dominates run time of the experiment description segment (*grenade*) with the exception of max granularity, i.e., one population for each neuron.

In summary, the central result of network scaling is that network topology should be described as coarse as possible, i.e., least granularity regarding populations and projections. In other words one should describe a network with few large populations compared to many small ones.

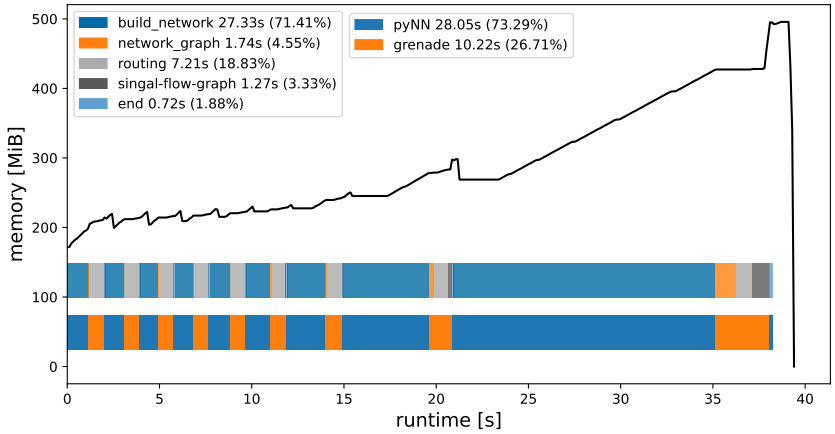


Figure 3.25: Run time analysis of a single measurement repetition of fig. 3.24. The same abstract all-to-all connected feed-forward network topology is constructed with increasing population granularity (cf. fig. 3.21). Overall, the network is constructed 9 times, where each iteration consist of 5 steps. First, the network is built in *PyNN* (`build_network`), then converted to abstract network graph(`network_graph`). This graph is then routed (`routing`) and subsequently translated to a signal-flow graph (`signal-flow-graph`). Finally, the *PyNN* state is reset (`end`). Horizontal bars represent these 5 steps (top) and their corresponding software layer (bottom). Annotations in legend present individual run time of steps and percentage of overall run time. Black line graph shows memory consumption during execution. Memory consumption starts at roughly 200 MiB due to import of necessary python libraries including `pyNN.brainscales`. As the overall topology of the network is constant so is routing time consumption. All other steps however scale with description granularity.

3.6.3 Impact on Experiment Workflow

Concluding full stack analysis, the ramifications of the obtained results on experiment workflow are discussed. Two use cases, long-running hardware executions and short iterative ones, are addressed.

Regarding long-running experiments it was shown that the developed software frameworks can in principle arbitrarily scale with run time under the constraints of available memory on host machine. Larger experiments would then needed to be split either temporally or physically. Nevertheless, run time larger than 10 s hardware wall-clock time at maximum recording rate will not be exceeded in the foreseeable future.

Regarding iterative workflow, however, the obtained results have a more im-

mediate impact. First of all, the constant overhead by always writing the full chip configuration for every hardware execution limits iteration rate to under 10 Hz. Fortunately, for many experiment types the aforementioned analog parameters which take up 100 ms do not change between runs. So, the natural approach to address this shortcoming is to only update parameters that have changed. A straight forward approach is to split fast and slowly configurable parameters and track them with dirty-flags. Another approach to this issue could be batching of experiment segments, i.e., executing multiple independent segments in a single run. The outlook of section 3.5.1 already outlines how such an approach could be automatically facilitated within the *PyNN* framework.

Execution times in order of seconds for the place and route algorithms likewise pose a limitation on experiment throughput. However, as explained in section 3.5.1, there are already mechanism implemented addressing this issue. Routing is omitted if a routing result from a previous run is available and no relevant change to network description was performed. Furthermore, the currently utilized routing algorithm is a straight forward, barely optimized implementation for which a faster and scaleable design is in development.

Regarding scaling to multi-chip setups it was shown that in principle execution can be parallelized without unexpected overhead. Of course providing true multi chip support still requires more than simply parallelizing experiments for individual chips. One of the greatest challenges will be the development of suitable routing algorithms for large-scale systems. Nevertheless, the architectural decisions outlined in this chapter provide a solid base for coming extensions to support multi-chip experiments.

3.7 Sudoku Solver

In the preceding sections the motivation, design, implementation and verification of the developed software architecture are explained. Now the usability of this framework is are demonstrated by a neural network implementing a 4×4 sudoku solver. This section presents the work jointly conducted with Alexander Nock and subsequent further investigation by the author.

As part of the physics advanced lab course taken by Bachelor students at Heidelberg University, the Electronic Vision(s) group offers an introductory course on neuromorphic hardware. Historically experiments for this course were conducted on the rather dated *Spikey* chip, an early neuromorphic chip also developed in this group (cf. section 2.2.2). The content of this course were migrated to the BSS-2 chip architecture as part of Alexander Nock's bachelor thesis [Nock 2021], which was supervised by the author. One of the main achievements of said work was the development of a Sudoku solver network experiment running on BSS-2.

Sudoku tasks are a subset of constraint satisfaction problems for which neural network solvers exist [Maass 2014], which have been deployed on various neuromorphic hardware devices [Fonseca Guerra et al. 2017; Kugele 2018; Ostrau et al. 2019; Yakopcic et al. 2020]. Thus, a Sudoku solver was chosen for the advanced lab course as it provides a real world application while still having an easy to comprehend implementation.

3.7.1 Experiment Setup

We will start with a short introduction on the rules of Sudoku and in that process elaborate on their representation in the solver's network structure Figure 3.26 illustrates the principle of the Sudoku solver. The goal of Sudoku is to fill empty

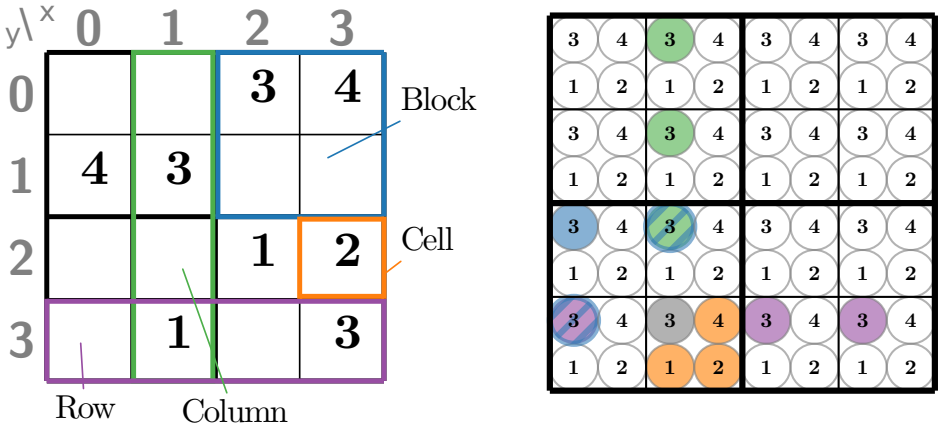


Figure 3.26: Principle of the neural network Sudoku solver. Left: Setup and terminology of a 4×4 Sudoku with 8 clues. Each number between 1-4 must appear exactly once in each row, column and block. Right: Setup of neural network. Each number in each cell is represented by a neuron. General concept is to connect neurons so that they implement the Sudoku rules. This is exemplarily shown for the neuron highlighted in gray. It is inhibitorily connected to each other neuron in the same cell and to each neuron representing the same number in the same row, column and block. Furthermore, it is self connected excitatorily therefore implementing a winner-take-all network. Taken from Kugele 2018.

cells according to given clues. In that process, each numbers is allowed to appear exactly once in each row, column and block. The general concept is to set up a neural network topology such that it implements these rules in a winner-take-all fashion. In this network each number of each cell is represented by a neuron. Each of those neurons is then connected inhibitory to each other neuron in its cell so that only one neuron, i.e. number, is active at a given time. This is repeated for

each other neuron representing the same number in the same row, column and block. Finally, neurons are excitatorily self-connected to stimulate themselves and thus implement winner-take-all behaviour. A 4×4 Sudoku is chosen because an implementation for a typical 9×9 Sudoku would have required $9 \cdot 9 \cdot 9 = 729$ neurons, whereas a BSS-2 chip only has 512 neurons. Experiment conditions are stated in appendix B.1.4.

To solve a Sudoku, the neurons representing clues are pinned by external stimulation with a high frequency input of 1 MHz. Furthermore, all neurons receive Poisson background input with lower frequency of 300 kHz facilitating solution exploration. Network dynamics should then lead to only neurons being active that conform to the Sudoku rules.

3.7.2 Chosen Sudoku Puzzles

It is crucial to carefully evaluate the performance of the network and to this end choose suitable Sudoku puzzles to be presented to the network. For 4×4 Sudoku there exist 13 581 312 valid puzzles [*Shi Doku* n.d.]. However, only 85 632 of those puzzles are minimal, i.e., no further digit can be removed while still having only one unique correct solution. Furthermore, these can be canonicalized by reduction of all grid symmetries and permutations of numbering. This results then in only 36 minimal canonical 4×4 Sudoku puzzles, 13 with 4 clues, 22 with 5 clues and 1 with 6 clues respectively. These 36 Sudokus where chosen as benchmark for the solver network.

3.7.3 Network Analysis

To verify the desired behaviour the network is emulated for 4 ms and recorded spike trains of all neurons are analyzed. For each Sudoku cell its 4 corresponding neuron activities are segmented in 100 µs intervals. The neuron with the highest activity then determines the number of the cell. For each time segment its classification is compared against the known solution. In the case of all cells representing the correct solution, the time segment is marked as correct. Figure 3.27 presents exemplary visualizations of network activity for two observed activity cases. For most presented Sudoku puzzles the network activity quickly finds the correct solution and finds a stable equilibrium. However in few cases the network activity is unstable and fluctuates between correct and wrong solutions. Longer experiment run times revealed no convergence to a stable solution.

The exact dynamics of the network and therefore its capability to correctly solve a puzzle depend on various network parameters. One such parameter are the weights of the connections describing the rules. A parameter sweep is conducted to investigate impact of the weights w_{ex}, w_{P} and w_{inh}. w_{ex} represents the weight

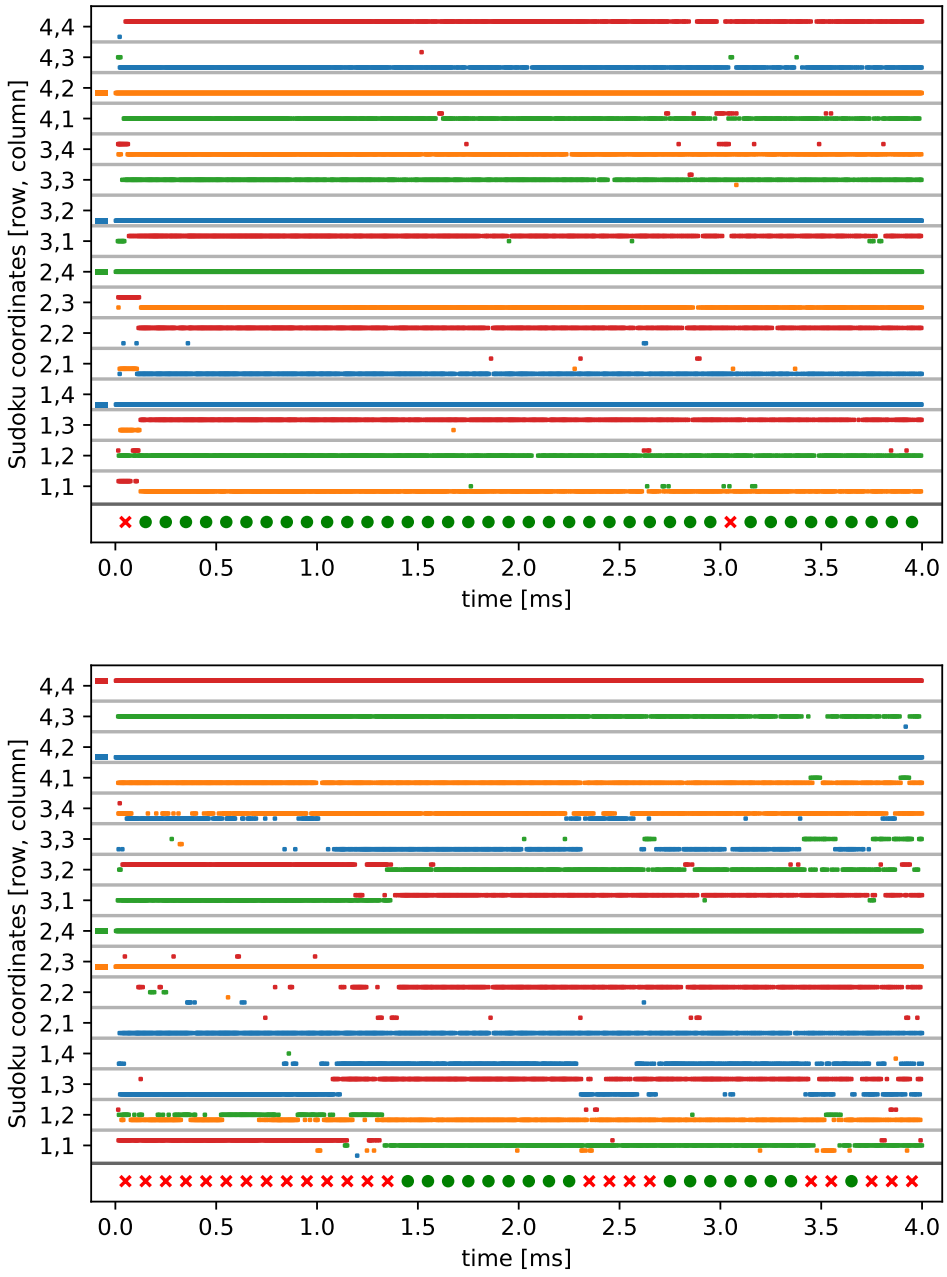


Figure 3.27: Example activity of Sudoku solver network for a stable solution (top) and a fluctuating network (bottom). Y-axis represents individual Sudoku cells with 4 neurons each. Separation of cells is visualized by grey horizontal lines. Presented clues are marked in front of the respective neurons as small lines in same color. Bottom row shows if given time segment correctly solves the Sudoku (green circle) or not (red cross). Top: Typical activity for most presented Sudoku puzzles. Bottom: Uncommon, fluctuating activity with fewer correctly solved segments and no convergence to stable solution.

of self-excitation, w_{P} of the Poisson background sources and w_{inh} the inhibition weights implementing the rules. Figure 3.28 shows the result of a sweep over w_{P} and w_{inh} while keeping w_{ex} constant as well as sweep over w_{ex} and w_{inh} while keeping w_{P} constant. To quantify the success of the solver network the ratio of correctly solved time segments versus wrongly solved ones is chosen.

First, sweep with constant w_{ex} is discussed. For low weights in either case the network is not able to correctly solve any Sudoku. This is reasonable as for low inhibition the Sudoku rules stop applying and for low background noise the network shows no activity. Similarly, in case w_{inh} dominates the network is too strongly inhibited resulting in too low activity or in case of strong w_{P} activity becomes too random.

When considering the sweep with constant w_{P}, a similar behavior can be observed for too low inhibitory weights Increasing self-excitation weights impairs result exploration and thus decreased classification rate. Interestingly small self-excitation weights are already sufficient if not even optimal for network performance. Even for no self excitation, i.e. $w_{\mathrm{ex}} = 0$, most Sudoku puzzles are still solved correctly. This suggests that for the chosen neuron dynamics a high enough inhibition and Poisson background activity is sufficient for quasi winner-take-all behaviour . General classification performance distribution is comparable to results in Ostrau et al. 2019, Figure 3 obtained for BSS-1.

One relevant metric for problem solvers is how fast they can ascertain a correct solution. To this end the time to first solution for all 36 Sudokus with 10 iterations each is plotted in fig. 3.29. The segmentation into 100 µs determines the lower bound to first solution, with 18% of Sudokus being solved in that time. In over 55 % of cases a correct solution is found within 200 µs. All presented Sudokus are correctly solved within the 4 ms run time. Time to first solution results are in the same order of magnitude as for BSS-1 conducted in Ostrau et al. 2019, Table 1.

3.7.4 Run Time Performance

Next, the run time analysis of the Sudoku solver implemented via `pyNN.brainscales` is shortly discussed. Table 3.2 presents the run time consumption of the different execution steps. For each Sudoku puzzle one hardware run is executed with subsequent analysis. *PyNN* setup and network description as well as route and construction of the signal-flow graph are only done once. This is made possible by defining a fully connected projection between clue stimuli and all neurons and initialize all of its weights to 0. To then pin neurons to the respective clues, the corresponding weights are set to 63 and the remaining again back to 0.

Hardware execution dominates as in the current implementation all neuron parameters are updated and not just the weights (cf. section 3.6.3). This settling time for analog parameters alone takes about 100 ms. For 36 iteration this adds

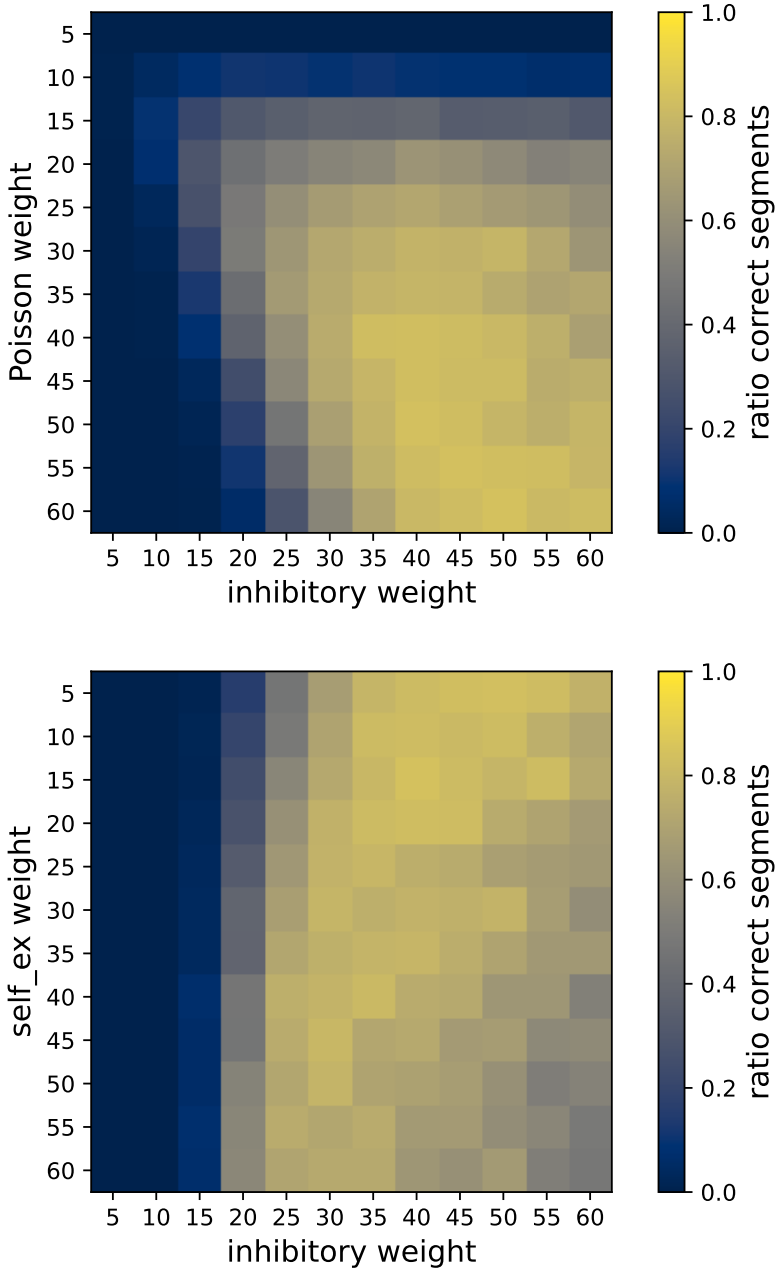


Figure 3.28: Parameter sweep over interconnecting weights of the Sudoku network. Top: Poisson background noise weight w_{P} over inhibitory connection weights w_{inh} with constant self-excitation connection $w_{\mathrm{ex}} = 5$. Bottom: w_{ex} over w_{inh} with constant $w_{\mathrm{P}} = 45$ Weight values are unit-less hardware values in the range of $[0 - 63]$. Element values are ratio of correctly vs wrongly solved time segments averaged over all 36 minimal Sudokus and 5 iterations.

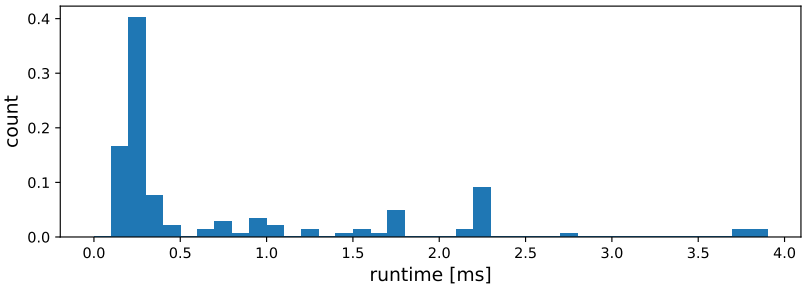


Figure 3.29: Histogram of the time to first correct solution. Time is defined by end of first time segment, i.e., best case 100 µs. In over 55 % of cases the correct solution is found within 200 µs. The bin resolution corresponds to the segment size of 100 µs. Each of the 36 minimal Sudokus was presented 10 times. All runs resulted in successful solution for at least one valid segment. Weight values: $w_{\text{ex}} = 5$, $w_{\text{P}} = 50$, $w_{\text{inh}} = 42$

up to 3.6 s of the 5.75 s spent in hardware execution.

Execution Steps	Run Time [s]
PyNN setup	0.77
Route and abstract graph	0.37
Hardware execution	5.75
Analysis	2.97
Total	9.86

Table 3.2: Run time of experiment steps for solving all 36 minimal Sudokus.

3.7.5 Outlook

The obtained results for the 4×4 Sudoku solver are promising but still several aspects can be improved.

First of all, a 100% classification rate is not always reached with small deviations in weight or background activity. A more thorough exploration regarding impact of different parameters on classification rate is therefore needed. Especially the robustness against deviations of background noise or hardware parameter variations is of interest. For example simply permutation of the number of a minimal sudoku could lead to a different result.

Furthermore, a convergence of the network to a stable solution would be desirable. Possible approaches to this would be a tempering mechanism, e.g., of

the Poisson background stimulus [Korcsak-Gorzo et al. 2021]. Solution space exploration could be increased by a regulation on high neuron activity, for example, Short Term Depression (STD) [Leng et al. 2018].

One major point that could be addressed is the implementation of the solver itself. In its current form it is specifically crafted towards solving Sudoku of size 4×4. A more flexible approach allowing arbitrary Sudokus, or better yet a general constraint satisfaction problem solving framework, would be desirable.

Overall run time could also be significantly improved. For example multiple Sudoku could be solved during one hardware run by batching the presented clues as outlined in outlook of section 3.5.1.

One issue that needs to be addresses before the setup can be transformed into a real solver is the nature of concrete execution times. Ideally one wants to exploit the short time to solutions however that time is not known beforehand. One possible approach would be to utilize the on-chip PPU to check for correct solution during run time. The results can than be stored and new weights for the next problem can be loaded.

Nevertheless, the primary goal of this experiment was reached. Namely, providing a functional and comprehensible application for the advanced lab course and demonstration of the capabilities of the developed software architecture.

Chapter 4

Neuromorphic Platform Operation

The previous chapter demonstrated how utilization of the BSS-2 neuromorphic hardware systems was facilitated by an extensive software framework. Before that, sections 2.2.1 and 2.2.2 gave an introductory overview of HPC platforms in general and neuromorphic computing platforms in particular. This chapter will now present the necessary measures taken to allow both BSS-1 and BSS-2 hardware systems to operated as such platforms. In particular the differences in operation that arise from the characteristics of the custom hardware are addressed.

First, section 4.1 describes how multi-user access to the heterogeneous hardware setups is managed by a customized resource scheduler. This includes for example isolation of hardware resource access to individual jobs. Additional requirements of the hardware like managing interdependencies to neighboring chips are also covered.

Secondly, the essential monitoring infrastructure for stable and reproducible operation is explained in section 4.2. It covers the different types of data generated by the systems and how they are aggregated and visualized.

Most features explained in this chapter where primarily developed in the context of BSS-1 operation. Thus, the main focus in most sections lies on BSS-1 systems (see section 2.2.3).

4.1 Resource Management

The amount of available compute devices, be it conventional or neuromorphic, are typically limited due to cost and other constraints. Thus, if multiple users want to utilize such devices some mechanism needs to be in place that distributes these resources. Providing shared access of limited compute resources is a common

problem in HPC which is solved by resource schedulers. In a typical workflow users prepare there compute tasks, e.g. simulations, on so called front-end nodes. From these they submit long-running experiments to a multitude of compute nodes where they are executed. Most conventional HPC cluster have rather homogeneous hardware, typically there are at most a few dedicated nodes types like GPU nodes. However, this is not the case for analog neuromorphic hardware. Neuromorphic setups vary from each other due to the inherent fixed pattern noise, described in section 2.2.3. The also described calibration efforts try to compensate these variations to a certain degree but cannot obscure them entirely. Consequently, the typical workflow for experimenters is to pick a hardware instance and stick to it. Researchers must therefore be able to specify a particular hardware setup for their experiment jobs.

In early stages of hardware setup development and commissioning, there are typically more setups than developers. Even then, there are several benefits in having resource management in place from the get-go. One advantage, especially true during the recent pandemic, is the constant availability through remote access. Another example are researchers that particularly investigate the aforementioned variations need access to multiple setups. At the latest when providing access to the systems for external researchers, it is inevitable to have robust shared access in place.

A short overview of the section structure is given. First, the state regarding resource management prior to this thesis is described in section 4.1.1. The subsequent sections will then present the efforts carried out to enhance usability, convenience and correctness. Section 4.1.2 gives a quick overview of the configuration of the utilized Slurm instance, in particular focusing on requirements of the research workflow on the neuromorphic systems. Then, in section 4.1.3 a performance baseline is defined which subsequently described features are compared against. The first described new feature is then isolation of hardware resources to individual jobs, thus ensuring no interference between experiments. This is facilitated by the subsequently explained native support for request of custom hardware setups to the scheduler tools via implementation of a Slurm plugin. Employing said plugin, an idiosyncrasy of the BSS-1 wafer systems is managed by an automated initialization feature, presented in section 4.1.6. Furthermore, the utilization of the hardware system is analysed in section 4.1.7. The section is then concluded by description of a micro scheduler that drastically increases hardware utilization.

4.1.1 Prelude

The widespread resource scheduler Slurm was chosen for BSS to manage access to conventional as well as custom-developed hardware. Key features are its fault

tolerance, heterogeneous resource support, high configurability and customizability.

In sections 2.2.3 and 2.2.4 the BSS-1 and BSS-2 hardware setups are explained in detail. However, a short introduction with most relevant information for this chapter is given.

The BSS-1 neuromorphic system facilitates wafer-scale integration to utilize a large interconnected mesh of 384 chips. These chips are in turn grouped in blocks of 8 called reticles. Each reticle is connected to an FPGA which handles off-chip communication via 1 $^{\text{Gbit}}/_{\text{s}}$ Ethernet. Furthermore, 12 ADCs boards are utilized for analog readout. The synchronization of the analog readout with experiment is controlled by 12 trigger groups. In the following the terms reticle and FPGA will be used frequently and somewhat interchangeably as they represent the same shared resource.

The BSS-2 single chip setups provide access for up to two independent neuromorphic chips. Each one is again connected to a communication FPGA.

Figure 4.1 illustrates these custom neuromorphic systems and how they are incorporated into the conventional compute cluster. The following explains how access to the custom systems are represented in Slurm. As seen in fig. 4.1 there are two access paths for an experiment setup. Either it is physically attached to a single node via USB or reachable via Ethernet from multiple nodes. Resources connected via USB can natively be represented in Slurm as *gres*[1] since they are only reachable from one specific node. For resources not exclusively attached to one node there is no immediately corresponding concept in Slurm. Therefore, Slurms software licenses framework[2] is misappropriated (to some degree). Due to the multitude of connotations the term *license* already carries it is hereinafter called GCR. GCRs operate similar to semaphores, a job can require a certain amount of globally defined GCRs and execution will be deferred until all GCRs can be allocated. Each license corresponds to a string-like constant and an amount, which is set to one as they represent unique physical entities. For each Ethernet connection a unique license with count one is defined, e.g., for the 3rd FPGA of BSS-1-Wafer 20 the license `W20F3` is created. An experimenter would than specify which hardware instances to allocated via `--licenses` Slurm argument.

All measurements are performed on *HBPHost* nodes if not explicitly state otherwise, see appendix B.2.

Other Contributions The initial integration of the Slurm infrastructure was established by E. Müller who is also a common contributor to Resource Management. The development of the neighbor initialization feature (cf. section 4.1.6)

[1] https://slurm.schedmd.com/gres.html 2021-07-23
[2] https://slurm.schedmd.com/licenses.html 2021-07-23

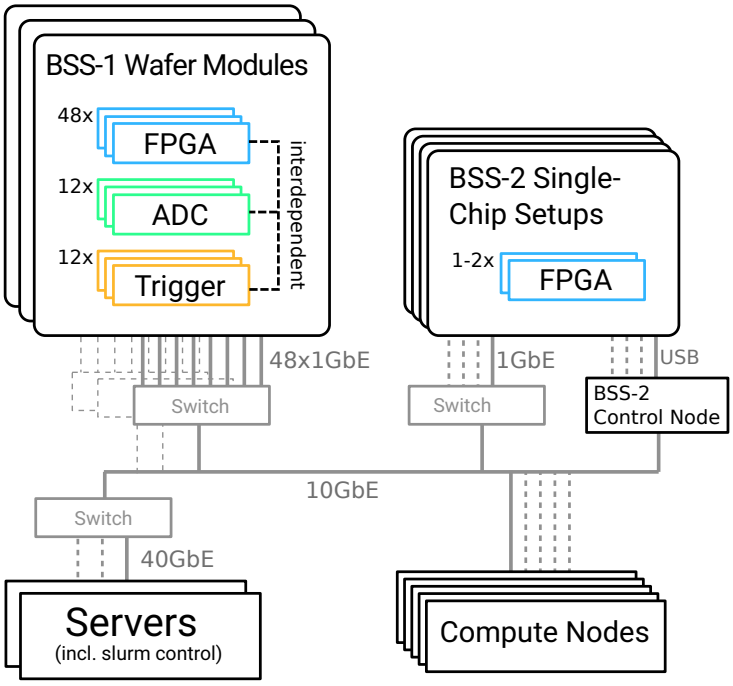


Figure 4.1: Overview schematic of the cluster setup incorporating the BSS-1 and BSS-2 hardware platforms. On the top both systems are shown with their respective managed resources and connections to the conventional compute nodes. Each BSS-1 wafer module is connected through its 48 FPGAs via 1 GbE. Each of those FPGAs as well as ADCs and corresponding triggers to start analog readout are non-shareable resources and are therefore tracked via GCRs. The FPGAs of BSS-2 prototype setups are tracked as GCRs the same way, yet their USB connections to a specific control node are tracked via *gres*. All experiment setups are connected via Ethernet over a mixed $^{10}/_{40}$ GbE backbone to compute nodes and servers, e.g., front-end and Slurm control node.

was performed in close collaboration with P. Häussermann during his bachelor thesis [Häussermann 2018] who also refactored the resource isolation capabilities (cf. section 4.1.4), thereby increasing their performance. Section 4.1.8, the experiment micro scheduler, was predominantly developed by O. Breitwieser in his PhD thesis [Breitwieser 2021]. Furthermore, he contributed to the general cluster operation.

4.1.2 Resource Scheduler Configuration

The Slurm-scheduler provides a plethora of configurations options from cluster composition to scheduling schemes and parameters. With this Slurm can be adapted to specific requirements of various use cases. The following presents an overview of how the scheduler configuration is set up to the needs for development and operation of the neuromorphic hardware platforms. First, general distribution of compute tasks is discussed followed by specific optimization of scheduling latency.

Compute nodes shown in fig. 4.1 are not only utilized explicitly for neuromorphic experiments but also for general compute tasks like compilation and simulations. Therefore, utilization needs to be balanced with higher priority to experiment execution. Slurm partitions are set up for the various tasks and chip generations. With this different features can be utilized to balance access according to task relevance. For example jobs scheduled into a hardware experiment partition are assigned a much higher priority than jobs in the compile partition, resulting in hardware jobs being able to overtake queued compile jobs. Furthermore, some nodes are exclusively assigned to hardware experiment partitions to always ensure minimal availability.

During development or experiment script prototyping researchers require a short response time for experiments as to not break their interactive work flow. Afterwards, experiments are scheduled in a batch fashion where the relevant metric is overall throughput. Therefore, a balance between high throughput and fast scheduling needs to be found. Due to Slurm's prevalence there are several studies investigating its overall performance and the impact of different configurations parameters [Simakov et al. 2018; Zhou et al. 2013]. Regarding high throughput Slurms documentation also provides suggestions for various parameters[3]. Based on this a suitable configuration of scheduling plugins and parameters was determined. The resulting impact is investigated in section 4.1.3. Furthermore, a node sharing feature[4] is used to increase utilization of the conventional compute nodes as well as job response time [Minami et al. 2021]. In default configuration

[3] https://slurm.schedmd.com/high_throughput.html 2021-07-23
[4] https://slurm.schedmd.com/cons_res_share.html 2021-07-23

only one job can be executed on one node. This would be very prohibitive for operation as amount of neuromorphic setups exceeds the amount of compute nodes. Especially in case individual chips on the wafer scales systems are to be utilized by individual jobs.

4.1.3 Baseline Performance

To verify that the developed features covered in the subsequent sections do not introduce too large of an overhead, a baseline to compare against is defined. On the one hand base overhead that resource scheduler itself introduces needs to be determined and on the other hand run time of typical experiments for BSS-1 and BSS-2.

Resource Scheduler Overhead

Regarding resource scheduler overhead there are two relevant metrics, individual jobs run time and overall job throughput. All measurements are conducted with a separate testing scheduler instance using the same configuration as production with disabled custom plugin and pro-/epilog scripts, see section 4.1.4 and section 4.1.5. Measurements are executed under a best case scenario, i.e., no other users where present and therefore no running jobs and minimal load.

Individual job run time overhead is determined by measuring wall-clock time of 10^5 jobs serially executed, using *srun*[5]. Repetitions over 5 iterations and averaging results in 106 ± 1 ms per job. However, this holds only true if scheduling is trivial meaning no pending jobs need to be overtaken. A pending job is waiting for execution as it cannot be scheduled due to some constraint for example blocked GCR. Repeating the same measurement but with pending jobs in queue results in 2723 ± 5 ms. Therefore, scheduling introduces a significant overhead for interactive usage but is imperative for platform operation.

Next overall job throughput is determined in batch operation, i.e., using *sbatch*[6]. To investigate the effect of scheduling overhead the job rate is measured dependent on number of submitted jobs resulting in fig. 4.2. An increasing number of no-op jobs allocating the same GCR is submitted where wall-clock time measurement is started on spawn of the first job and stopped once the last job finishes. The measurement is conducted with and without pending jobs in queue to be overtaken and each data point averaged over 10 repetitions. As expected, scheduling overhead diminishes the job rate for low number of submitted jobs and pending jobs increase the scheduling overhead leading to slower rise. In both cases jobs a plateau is reached at a rate of around 14 Hz meaning individual

[5] https://slurm.schedmd.com/srun.html 2021-07-23
[6] https://slurm.schedmd.com/sbatch.html 2021-07-23

job overhead is about 70 ms. Larger queues however lead to high load on the scheduling controller resulting again in lower rates.

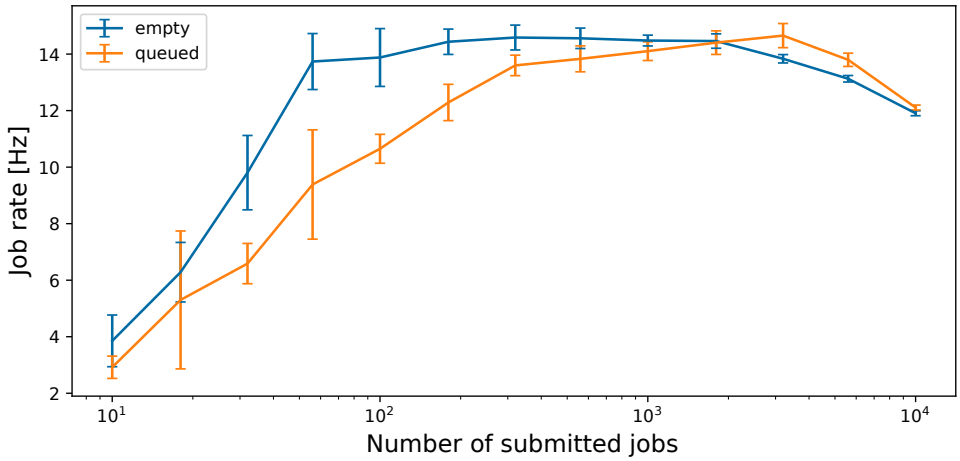


Figure 4.2: Resource scheduler job throughput. Wall-clock time is measured for an increasing number of submitted no-op jobs allocating same resource. *queued* is measured with pending jobs to be overtaken by scheduling, *empty* no pending jobs to be overtaken. Data points show mean and standard deviation over 10 repetitions.

Experiment Run Time Baseline

To provide an estimate of experiment run time baseline software overhead and typical minimal experiments are investigated. Software stacks of the BSS-1 and BSS-2 platforms are examined separately. Due to physical locality to the respective setups all BSS-1 related measurements are performed on *HBPHost* machines and on *RyzenHost* for BSS-2 respectively, see appendix B.2 for specifications.

The intended experiment for the BSS platforms consists of a python script executed by a user, see section 3.1. Therefore, as a bare minimum the overhead of importing the relevant python libraries is benchmarked by minimal python scripts (cf. appendices B.1.1 and B.1.2). An average over 100 iterations yields 1.37 ± 0.04 s and 1.80 ± 0.07 s for BSS-1 and BSS-2 respectively.

Next, run times of minimal experiments are estimated. As example experiment for BSS-1 the `nmpm1_single_neuron.py` demo script is taken from the BSS-2 guidebook web-page [HBP 2021]. For the BSS-2 systems the `single_neuron_demo.py` from the *pynn.brainscales* github repository[Demo 2021] is taken as minimal example. Both scripts are chosen to provide a similar overview of the feature-sets of the respective *PyNN* back-end implementations. Run time

averaged over 100 executions results in 18.4 ± 0.8 s and 5.0 ± 0.3 s for BSS-1 and BSS-2 respectively, including full overhead of python imports.

Impact of Overhead

Regarding interactive workflow resource scheduler overhead and run time of small experiments are in the same order of magnitude, i.e., seconds. The relative large overhead introduced when needing to overtake jobs is therefore acceptable.

Regarding iterative workflow scheduling frequencies of over 10 Hz are promising to take full advantage of the high acceleration factor of the neuromorphic hardware. However, bare minimum overhead due to python importing makes such an approach unviable. Therefore, typically iterative experiments are performed by allocating longer running jobs and iterate within the job body. Section 4.1.8 will present a solution for higher hardware utilization.

4.1.4 Resource Isolation

Solely relying on experimenters specifying a correct set of GCR for their experiments is insufficient as people simply make mistakes or, even worse, may have malicious intent. A common case would be a user changing used hardware in their script but forgetting to adapt the GCR. When they now run a new job it could interfere with already running jobs utilizing the same setup. It is therefore necessary to allow access only to experiment setups for which GCRs are allocated.

This restriction is realized by setting up firewall rules via the *iptables*[7] tool utilizing Slurm's pro- and epilog framework[8]. This framework allows execution of arbitrary programs on a node immediately prior to and after execution of a job. *bash*[9] is used for pro- and epilog scripts as it is readily available on node machines and relatively easy to maintain. All relevant information is extracted via *scrontrol* command for example requested hardware GCRs names.Per default for each neuromorphic device, be it BSS-1 wafer or BSS-2 single-chip setups, access via User Datagram Protocol (UDP), TCP as well as Internet Control Message Protocol (ICMP) are blocked for the relevant subnet, e.g., for wafer 20 this would be `192.168.20.0/24`. Once one or more hardware resources are allocated by the user an exception rule is added in the pro log script providing access to the respective setups. Once the job has finished the created exception rule is deleted in the epilog script. With this access to a neuromorphic resource is securely confined within a job. Additionally, task-prolog and task-epilog are used to add

[7] https://linux.die.net/man/8/iptables 2021-07-23
[8] https://slurm.schedmd.com/prolog_epilog.html 2021-07-23
[9] https://www.gnu.org/software/bash/ 2021-07-23

further information to the user environment as they are, contrary to pro- and epilog, executed in the user environment.

Performance

To determine the overhead introduced by resource isolation run time of prolog, task-prolog, task-epilog and epilog script where measured. Figure 4.3 shows the run time of the individual scripts, their sum and overall run time with increasing number of requested resources. For each script the time at start and end point were logged with the difference determining their run time. The overall run time is given by start of prolog and end of epilog script. For each amount of GCRs 100 jobs executing a no-op where submitted and averaged. The chosen numbers of GCRs correspond to an empty run, increasingly allocating all FPGAs of one wafer module and then two and three whole wafer modules. Linear growth is expected as all operations for isolation are executed serially in loops without parallelization as many tools, e.g., *iptables* are not reentrant. All scripts show the expected linear growth and linear regression for the sum of all scripts yields a slope of 0.0135 s with intercept at 0.882 s.

The baseline overhead of ≈0.882 s for empty run is dominated by acquiring information of the run job via the *scontrol* command, which in current implementation is called in each individual script. This is done as prolog and epolig scripts do not share the same environment. Alternatives for example file based information sharing would conceptually be feasible but very involved and introduce further technical debt.

The prolog script has the largest contribution as the most costly tasks, e.g., setting up the firewall exception rules is performed there. Each additional GCR increases run time by ≈13.5 ms.

Next, acquired results are compared with the baseline defined in section 4.1.3. In general, an overhead of about a second was deemed acceptable as it should not significantly interfere with interactive work flow of researchers. For BSS-1 prototyping experiments with run times ranging from 20 s to several minutes such an overhead is negligible. In case of BSS-2 prototyping experiments ranging in the order few seconds overhead is more significant but still acceptable.

4.1.5 Native Resource Request API

The wafer-scale BSS-1-system has various interdependent resources which may or may not be needed in an experiment, see section 4.1.1. An example of such resources are the 12 external ADCs used for analog readout. Each HICANN-chip on a wafer module can be read out by two of those ADCs but each ADC can only read out one signal from a chip at a time. As not every experiment needs analog

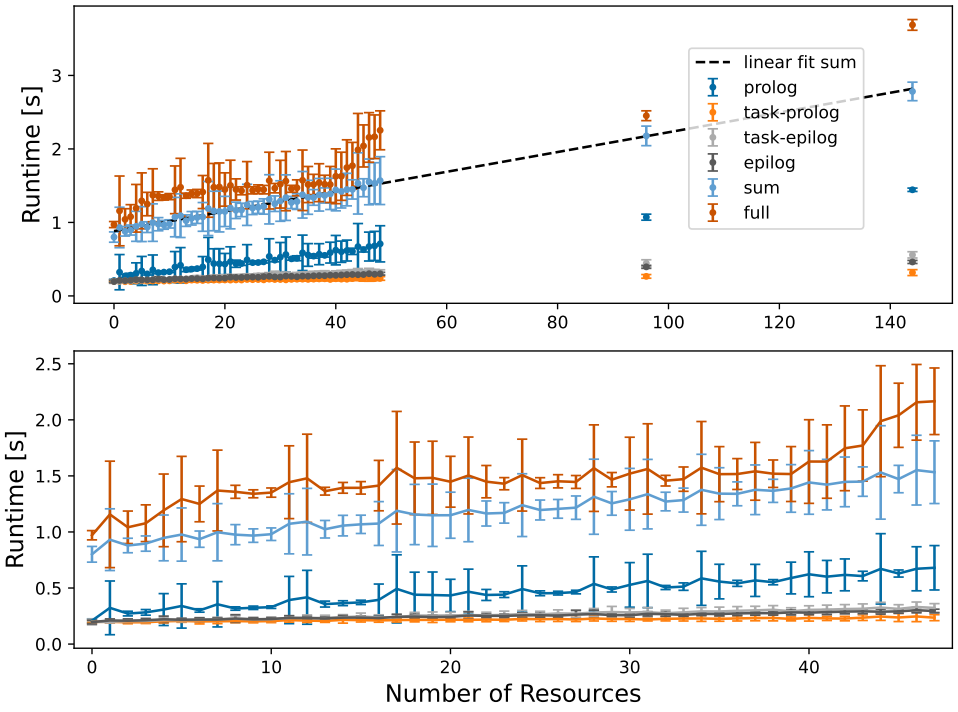


Figure 4.3: Resource Isolation run time overhead. Run time for prolog, task-prolog, task-epilog, epilog, their sum and total run time for an increasing numbers of allocated GCRs. Each point is averaged over 100 runs and standard deviation is taken as error. 1-48 GCRs correspond to individual FPGAs of one wafer module and 96 and 144 GCRs for two and three whole wafer modules respectively. Top shows full range, bottom subset for better resolution. Linear fit result: intercept 0.882 s, slope 0.0135 s, $R^2 = 0.99$.

readout, e.g., only spike trains are relevant, it is desired to explicitly state the need for such an ADC. They are tracked via their unique serial number as GCRs, e.g., *B219566*, similar to the communication-FPGAs. As the affiliation between chip and corresponding ADC is not described by a simple rule, a lookup is necessary, see fig. 4.4.

As such a lookup is tedious and prone to errors an automation is needed. To this end a Slurm plugin was developed. Its goal is to extend the Slurm job submit tools to provide human-writable command line arguments for the various resources. Slurm provides an extensive framework[10] for plugins that can be executed at various steps within a compute job's lifetime. Specifically a job

[10]https://slurm.schedmd.com/plugins.html 2021-07-23

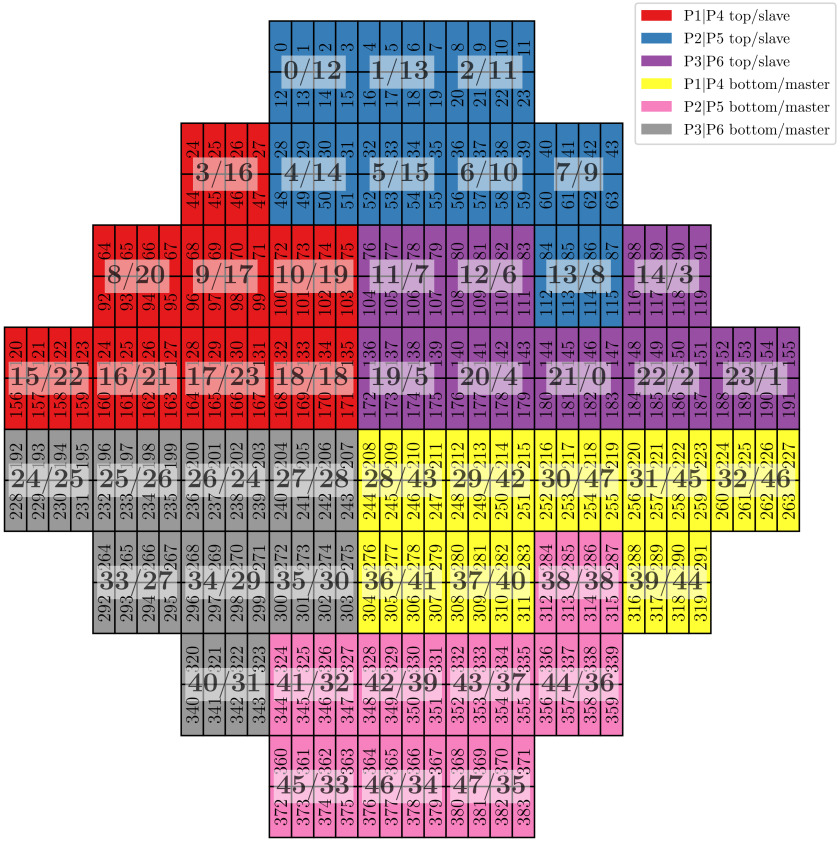


Figure 4.4: Overview of the different components and numbering schemes on a wafer module. A square corresponds to a reticle and its corresponding communication FPGA denoted by the two white highlighted numbers. Each reticle consisting of 8 HICANNs. Analog readout groups are color coded. Not marked are the trigger groups.

submit plugin[11] was added. It is injected after a job was submitted but before it is scheduled, this allows freely modifying all parameters of a job, e.g., requested GCRs. Job submit plugins however do not by themselves support adding new command line arguments for the job-spawning tools srun[12] and sbatch[13]. For this purpose, a plain spank plugin[14] was added in conjunction. It adds additional command line arguments which are then parsed and acted upon in the job submit plugin. Figure 4.5 outlines the control flow of a job including the plugin and is interaction with the prolog and epilog scripts. Most features in subsequent sections built upon these plugins.

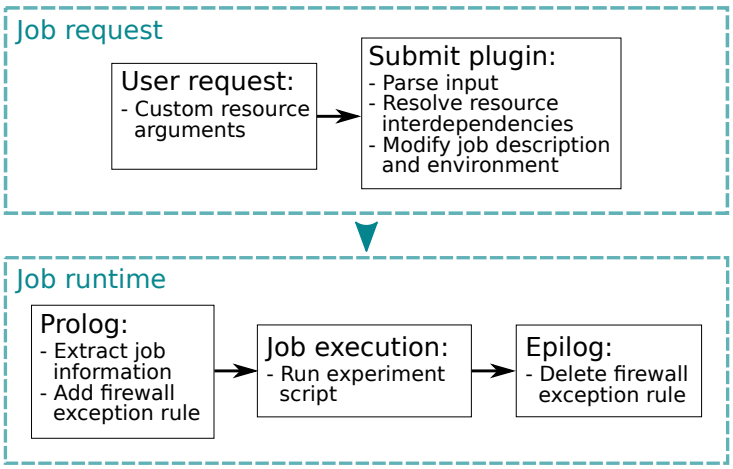


Figure 4.5: Schematic of job control flow including plugin prolog and epilog scripts

Different granularities of resource allocation are supported. Users can allocate single chips, reticles, FPGAs and/or whole wafer modules. In case multiple single chips are requested it is looked up to which FPGAs they correspond and then those FPGA GCRs are requested. For individual components, e.g., chips or reticles, one can specify if and which analog readout should also be automatically allocated. When allocating a whole wafer module all corresponding resources, i.e., GCRs, are requested. Some exemplary parameters can be seen in listing 4.1. It also shows exemplarily how long-winded a call directly requesting hardware GCRs utilizing the native Slurm API would be.

Hardware executions reaching over more than one reticle need to be synchronise. To this end all FPGAs share a signal line which is utilized by one specific FPGA, the master-FPGA, at experiment start. If multiple reticles or FPGAs are

[11] https://slurm.schedmd.com/job_submit_plugins.html 2021-07-23
[12] https://slurm.schedmd.com/srun.html 2021-07-23
[13] https://slurm.schedmd.com/sbatch.html 2021-07-23
[14] https://slurm.schedmd.com/spank.html 2021-07-23

Listing 4.1: Example resource scheduler calls to allocate wafer resources

```
# allocate the full module
srun --wafer 33 experiment_example.py
# allocate reticle 4,5 and 6 (with analog readout and trigger by
↪  default)
srun --wafer 33 --reticle 4,5,6 experiment_example.py
# equivalent call if manual allocation was necessary
srun --licenses
↪  W33F10,W33F12,W33F14,W33F15,B291728,B291712,W33T5,W33T6
↪  experiment_example.py
# allocate only HICANN 0 (without analog readout and trigger)
srun --wafer 33 --hicann-without-aout 0 experiment_example.py
```

requested this master-FPGA is also automatically requested additionally. This automated allocation can be disabled via an optional argument.

The plugin utilizes the coordinate system, see section 3.3.1, for parameter translations and range checks as well as the hardware database, see section 3.3.4. As the plugin is written in *C*, a full *C*-wrapper was written for the hardware database. It also includes wrappings of needed coordinates and is located in the same repository.

To provide various information regarding hardware allocations during a running job the user environment is extended. Some of this information, e.g., allocated GCRs, can be extracted via the Slurm native tool *scontrol*, but for example the information about which individual HICANNs are requested would be lost. Other examples are all allocated FPGA-IP or the database entry in its YAML source format.

One of the limitations of the current implementation is the explicit design of the query API described in listing 4.1. It was designed with a concrete wafer utilization in mind. This could be improved by unifying the job submit plugin API and hardware database APIs. Furthermore, a switch from `C` to `C++` as programming language for the plugin would be desirable. This is possible by compiling the plugin with a from Slurm separated build flow with `C` linkage. An advantage would be easier coupling with the main software stack and therefore reduced technical debt.

Performance

Overhead of the custom Slurm plugin is directly measured in code. Allocating a full wafer results in a run time of 68 ± 5 ms averaged over 100 iterations. Yet, total run time of the submit plugin is dominated by loading the hardware database

which takes 65 ± 5 ms. Hence, plugin run time is nearly independent of number of requested GCRs. If no hardware resources are requested all functionality is skipped resulting in run time of 0.14 ± 0.13 ms

Run time could be reduced by caching the loaded hardware database between calls of the plugin. This could be implemented by storing and comparing modification date of the database file. As overall overhead is dominated by the prolog and epilog scripts, see section 4.1.4 this improvement was not yet implemented.

4.1.6 Automated Neighbor Initialization

A HICANN-chip on the wafer system is not fully independent of its directly neighboring chips. Digital inter-chip communication buses, also called L1 (see section 2.2.3), are directly connected to repeaters of their neighboring chips. If L1 is to be used during an experiment then these repeaters need to be in an initialized state. In general a chip is assumed to be uninitialized as checking if it is in a fully initialized state would take the same time as just initializing it. Hereinafter an initialized chip will be called "clean" and a chip in undefined state "dirty". This interdependency to neighboring chips was discovered late in commissioning (roughly mid of 2019) and could not be remedied in hardware. Again taking off the burden from experimenters to keep track of these dependencies an automated solution is needed. The naive approach is to additionally allocate all neighboring HICANN and initialize them, which has some downsides. As fig. 4.6 shows, every HICANN has at least one and at most two neighboring reticles. Those neighboring reticles would be blocked during experiment run time for potential other jobs even though they are only needed to be initialized in the beginning. However, a user cannot release resources during job run time. Furthermore, chip initialization takes about 18 s (fig. 4.8). Thus, it would be ideal to skip initialization if a prior run already performed it.

The resource scheduler is prime candidate to handle both these issues. From within scheduler control resources can be modified during run time. Furthermore, as the scheduler handles all hardware access it can initialize the hardware and track its state.

To this end the previously introduced Slurm plugin, see. section 4.1.5, as well as prolog script were enhanced. The goal is to automatically allocate and initialize neighboring reticles if not already clean from a previous run. Furthermore, the GCRs of neighboring chips should only be held as long as necessary. An flow chart of the implementation is shown in fig. 4.7.

First, the plugin determines all reticles with neighboring HICANN chips of the requested resources. Lists of all user-requested GCRs and neighboring GCRs is stored. The collection of all GCRs is then used for scheduling as all resources need to be available in the beginning of the job. After the job is allocated to a

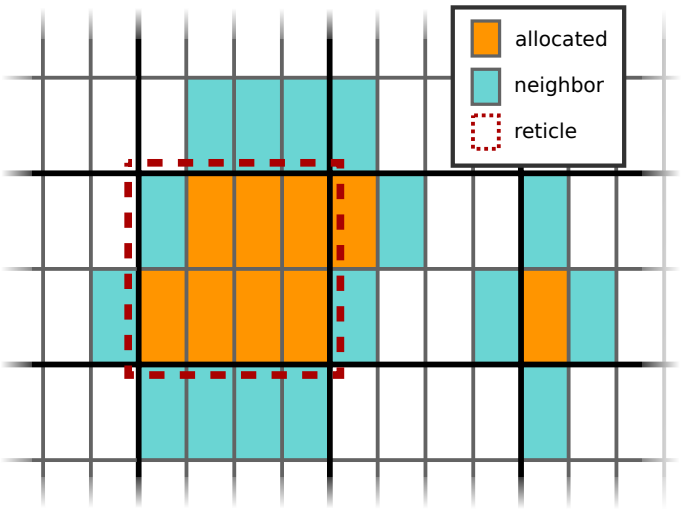


Figure 4.6: Illustration of exemplary HICANN chip resource allocations and corresponding neighbors. Shown are two exemplary resource request. Right: a single chip. Left: an arbitrary allocation larger than a reticle. For the single allocation two neighboring reticles would need to be requested and initialized while for the multi-chip four reticles are required.

node all further steps are performed via the scheduler prolog script. All relevant information is extracted from the job description. The state including a time stamp for all GCRs is tracked in the same SQL database already utilized by Slurm. A GCR is considered dirty either if its state is dirty or the time stamp is older than 24h hours. This timeout of 24 hours was empirically chosen to mitigate possible variations over time, e.g., voltage drifts. First, all user-requested resources are marked as dirty since, after user access, the state of hardware must be considered undefined. Now all neighboring GCRs are queried and tagged for initialization if they are dirty. Resource isolation firewall exceptions, see section 4.1.4, are temporally set via an additional firewall rule for neighboring resources during prolog execution. A pre-deployed version of the BSS-1-software-stack is then used to initialize the neighboring reticles. SQL entries of the freshly initialized reticles are updated to a clean state with the current date and time. Finally, the neighbor GCRs are release from the job. Additionally, there are two optional command line arguments to disable or force initialization regardless of database state execute.

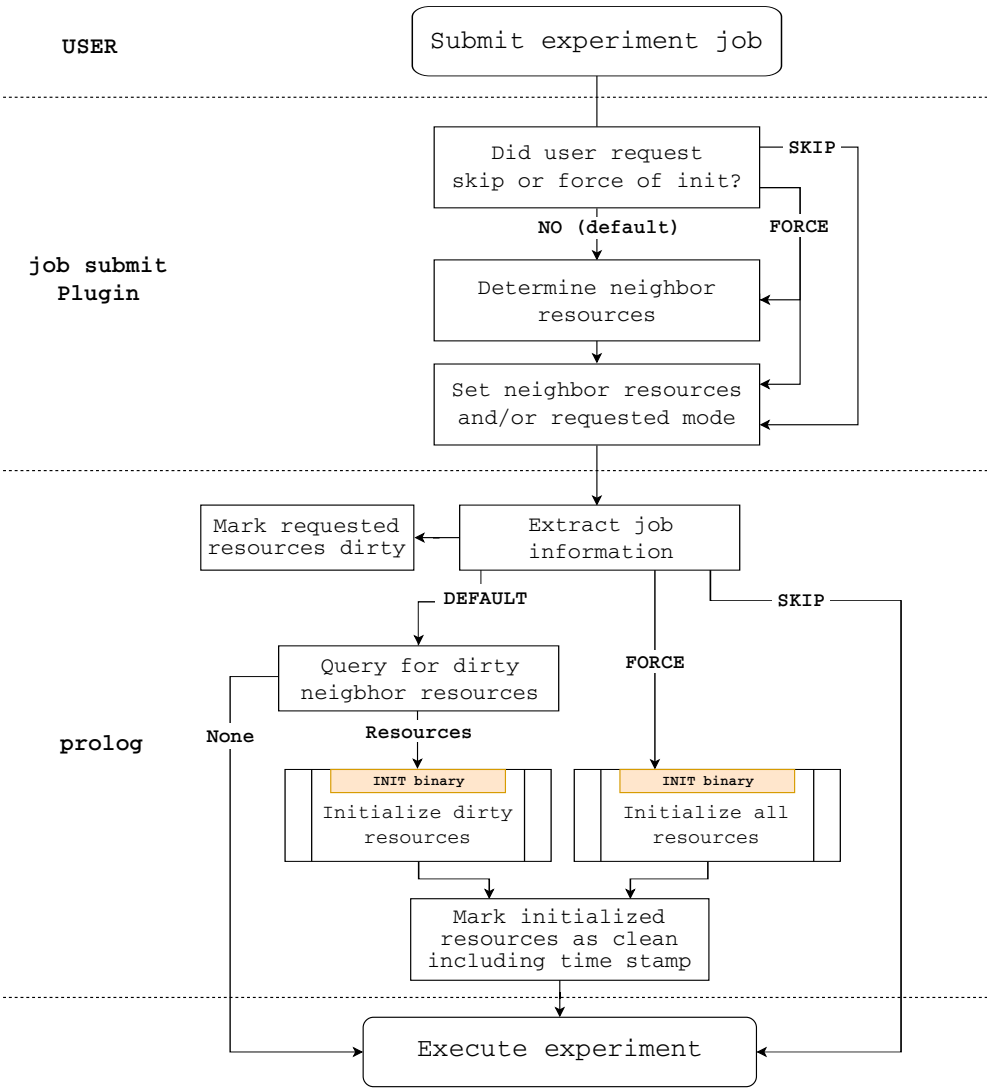


Figure 4.7: Flow chart of automated neighbor initialization feature for BSS-1 wafer modules. User-requests resources for experiment run and optionally if neighbor initialization should be forced or skipped. Job submit plugin determines neighbors accordingly an modifies job information to provide information to prolog script. Prolog script then executes neighbor initialization if necessary followed by the experiment execution.

Performance

Now the impact of the introduced automated neighbor initialization feature is investigated. First of all overhead of the reticle initialization itself is determined. Figure 4.8 shows run time of the reticle initialization script for increasing number of reticles in parallel. The small increase in run time is expected as the initialization of each reticles is independent of the others it can be easily parallelized.

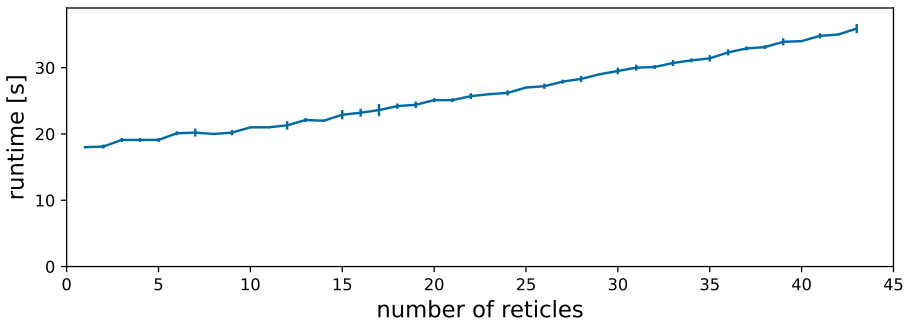


Figure 4.8: Parallel reticle initialization script run time analysis for full wafer module. Script is called for increasing number of reticles averaged over 10 iterations each. Experiment conducted on wafer module 30 (D9NKP16G4) which has 4 reticles with non-functioning high-speed links which are excluded (see section 2.2.3).

Next overhead introduced through the neighbor initialization feature is analyzed where three cases are distinguished:

1. Neighbor initialization feature is skipped, but allocated resources still need to be marked as dirty. Defines base overhead even when feature is not utilized for an individual job.

2. Feature is active and no neighbors need to be initialized. Defines lower bound.

3. Feature is active and all neighbors need to be initialized. Defines upper bound.

For the first case the same measurement protocol as for resource isolation overhead (section 4.1.4, fig. 4.3) was used now with neighbor initialization features active, resulting in fig. 4.9. Again, a linear increase with additional requested GCRs is expected. Linear fit yields an overhead for an empty run of 0.850 s. This agrees with results from resource isolation measurement, see fig. 4.3. An increase of 44.7 ms per GCR resulting in an overhead of over 2 seconds for 48

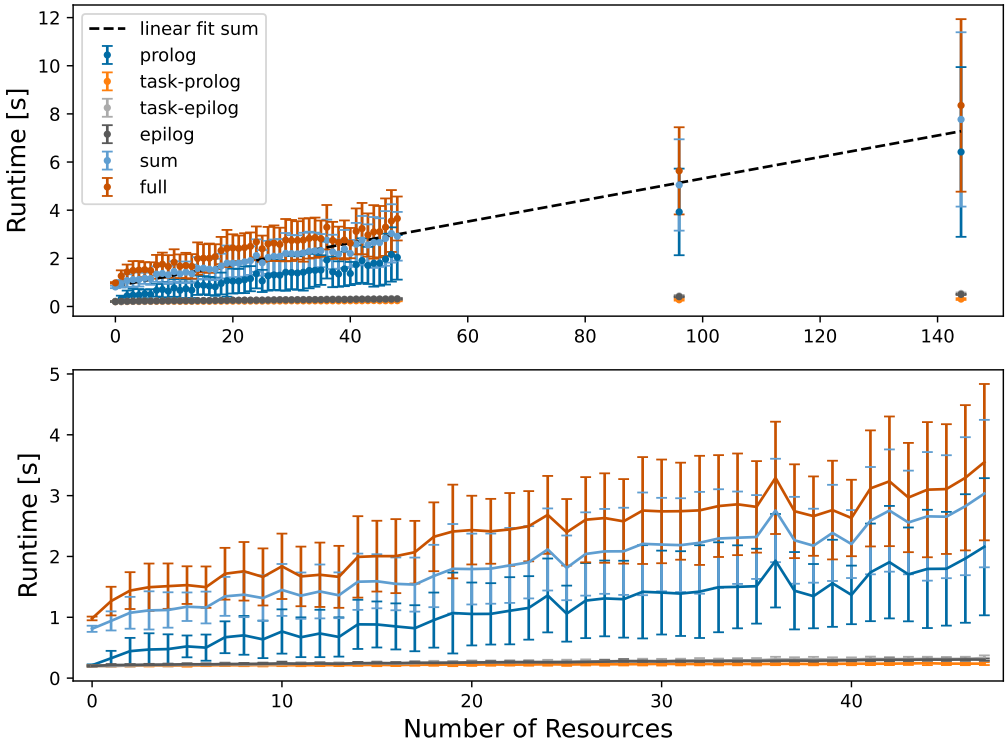


Figure 4.9: Neighbor initialization feature base run time overhead, i.e., case 1. Same measurement protocol as fig. 4.3 including neighbor initialization feature but initialization is skipped, i.e., only additional overhead is marking resources as dirty in SQL database. Linear fit result: intercept 0.850 s, slope 0.0447 s, $R^2 = 0.98$.

GCRs is unexpectedly high. From multiple requested GCR only one command is constructed and sent to the SQL database. Therefore only a small linear increase would be expected.

The SQL calls are measured explicitly to further investigate both long run time and its high variances specifically, resulting in fig. 4.10. Again, run time shows linear growth. However, in most cases the median is close to the minimal run time. Thus, strong outliers lead to a widened standard deviation and skewed mean run time. Potential reasons for strong outliers in SQL command execution could be that the Slurm control node is running in a Virtual Machine (VM). VMs can introduce additional jitter during process scheduling and I/O [Abeni et al. 2020]. Potential solutions could be switching to a real-time kernel or even a bare metal installation. As most run times do not introduce a strong overhead, i.e., only a few milliseconds for a single GCR, these solutions were not yet investigated.

Now case 2 and 3 of neighbor initialization are investigated. Figure 4.11 shows

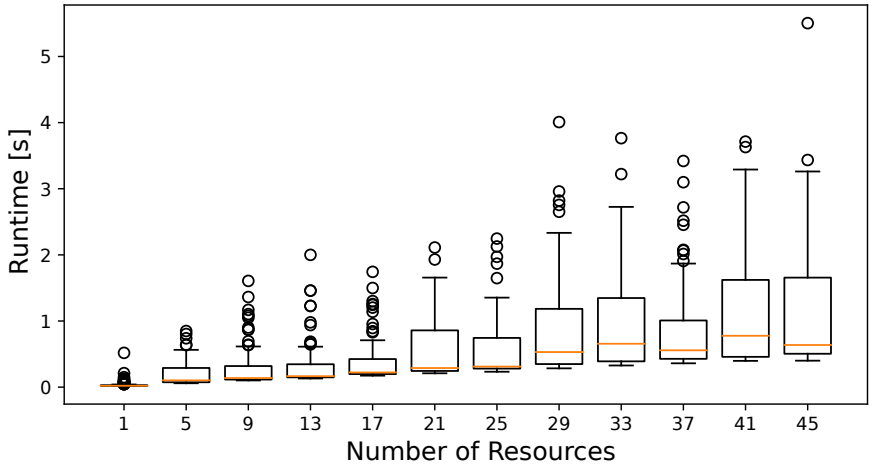


Figure 4.10: SQL query run time overhead analysis for same experiment setup as fig. 4.9. Box plot of run time distribution of SQL calls for an increasing number of dirty resources. Medians close to shortest run time and many outliers show that mean and standard deviation are skewed.

run time of both cases, again measured via prolog and epilog scripts analog to fig. 4.3. Only sum and total run time are shown as other scripts do not vary from previous measurements. Additionally, the run time of the raw reticle initialization script is plotted for comparison (data taken from fig. 4.8). As expected, for case 2 run time is in the same range as for case 1 (fig. 4.9) as there is marginal overhead trough checking if neighbors are clean. For the 3rd case run time is dominated by initialization of neighboring reticles, as can be seen by the raw reticle initialization run time. In typical experiment use cases the actual requested hardware instances do not change for an experimenter between most runs. Thus, full advantages of case 2 can be taken.

Use Case: Calibration Routine

One of the main use cases where automated neighbor initialization is required is the wafer calibration routine, developed in Kleider 2017. Historically, it was designed for single-chip granularity. Calibration requires timed analog signal readout which leads to resource interdependencies, i.e., the limited number of 12 ADC, on a wafer module. For each HICANN, a calibration job is submitted where the resource scheduler handles these interdependencies. Run time of the developed automated neighbor initialization is compared with a naive implementation where all neighboring resources are held for each calibration step. As shown in

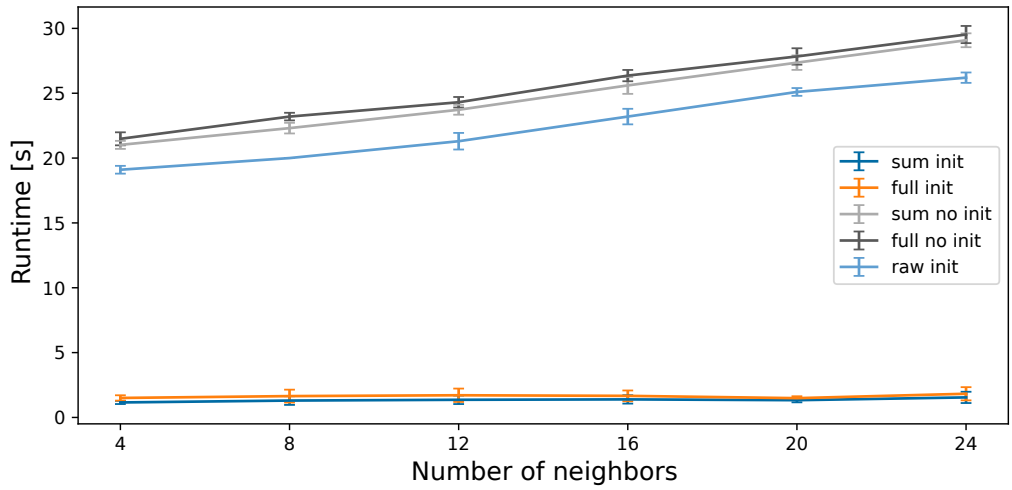


Figure 4.11: Neighbor initialization run time overhead for case 2 and 3. Analogous to fig. 4.3 sum and full run time of prolog/epilog scripts is plotted against increasing number of neighboring reticles. Additionally, run time of raw reticle initialization script is plotted.

fig. 4.6, individual HICANNs have one or two neighbors on different reticles and in case they are at the edge of a wafer can also have zero neighboring reticles. As calibration for a single HICANN takes nearly 2 hours and the actually executed job is not relevant to measure overhead, dummy jobs are submitted with same resource constraints as for a real calibration. In case of automated initialization the job body is a sleep of 300 s whereas for the naive approach a manual initialization precedes this sleep. 300 s are chosen so that the initialization itself is not dominant and the overall run time is not unnecessarily long. The additional manual initialization in case of naive dummy runs serves for better comparability between both implementations, as the initialization routine has varying run time depending on the connection quality of the specific chip instance.

Figure 4.12 shows the run time distribution of all individual jobs for 5 dummy calibrations. As expected in case of automated initialization of a significant amount of jobs only last for the sleep duration plus few seconds scheduler overhead as neighbors are already initialized. Jobs for naive implementation that also only have 300 s run time correspond to HICANNs at the edge of a wafer which have no neighboring chips and therefore need no initialization. Peaks at roughly 340 s and 360 s correspond to HICANNs with non-functioning high-speed links, where communication-fallback is performed via the much slower JTAG protocol. A wider spread and outliers of run times is possible as steps in the initialization routine are retried if not successful immediately.

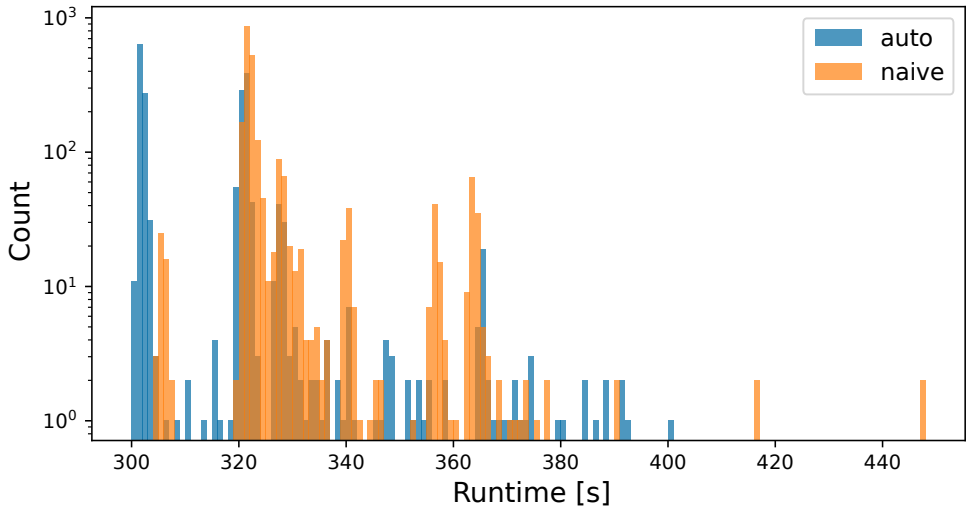


Figure 4.12: Histogram of individual job run time for dummy calibrations with naive and automated neighbor initialization. 384 jobs, one for each HICANN chip, are spawned with same neighbor constraints as real calibration. Execution body is a sleep of 300 s in both cases including a preceding initialization in naive case. Preceding initialization severs better comparability so both cases execute the initialization which has varying run time dependent on chip connection quality. These variations lead to wider spread and outliers in run time.

To put these numbers into perspective, the theoretical minimum run time of a full dummy calibration run is estimated. With ideal parallelism, no scheduling overhead and no resource allocation overlap the minimally possible overall run time t_{min} would be:

$$t_{min} = \frac{n_{FPGA} \cdot n_{chip} \cdot t_{sleep}}{n_{ADC}} + t_{init} = \frac{48 \cdot 8 \cdot 300\,\mathrm{s}}{12} + 36\,\mathrm{s} = 9636\,\mathrm{s}$$

where n_{FPGA} represents number of FPGAs, n_{chip} number of chips, t_{sleep} execution of dummy body, n_{ADC} number of parallel analog readouts and t_{init} one full wafer initialization in the beginning. One parallelized wafer initialization takes 36.1 ± 0.3 s, see fig. 4.8.

Table 4.1 shows the comparison of theoretical, automated and naive case run times averaged over five executions. Comparing full wall-clock run time, i.e., from start of first job to end of last job, and mean of single job run time show a difference in relative overhead to theoretical minimal time. Assuming full parallelism of single jobs an estimated wall-clock run time of $10\,048 \pm 480$ s for automated and $10\,432 \pm 416$ s for naive implementation could be reached. This results in an discrepancy of 551 ± 483 s and 1558 ± 439 s compared to theoretical

	full wall-clock time [s]	single run [s]
theoretical	9636	301
automated	10 599 ± 53 (11%)	314 ± 15 (5%)
naive	11 990 ± 140 (24%)	326 ± 13 (9%)

Table 4.1: Dummy calibration run time comparison between automated and naive implementation, see fig. 4.12 for experiment setup. Full wall-clock time is defined by start of first executed job and end of last finished job. A single run corresponds to mean run time of all individual jobs. Number in brackets are relative run time overhead compared to theoretical run time. Results are averaged over five runs.

minimal wall-clock run time.

To investigate the discrepancy, fig. 4.13 shows pipelined execution of individual dummy jobs for one arbitrarily chosen run. As resource sharing of the analog readout is the main bottleneck jobs are grouped for one of the limiting resources, the 12 trigger groups. One can see a higher utilization for automated compared to naive implementation in the later half of runs. This congestion for the naive implementation can be explained by the higher resource constraints as the neighboring reticles are not relinquished after initialization. Assuming wall-clock run time discrepancy for naive and automated case are caused by congestion alone it should be linearly proportional to job run time. Therefore, an overhead factor of roughly 1.7 and 4.8 calibration execution times is expected for automated and naive case. Extrapolation to full calibration run time of 2 h this would result in an overhead of 3.4 h and 9.6 h respectively compared to theoretical run time of 64 h. Therefore, automated initialization saves about 6 h or 10% of overall calibration run time for one wafer module.

In summary, automated neighbor initialization provides a convenient and robust solution to ensure proper experiment behaviours. In cases where the feature is not needed it only introduces a small overhead. Compared to a naive implementation, which would execute neighbor initialization always, it saves significant time if the same resources are requested consecutively. For the frequent use case of calibration it reduced run time overhead by roughly 6 h or 10% compared to a naive implementation. Ultimately, the goal for future hardware designs needs to be to avoid such interdependencies.

4.1.7 Scheduler Utilization Analysis

Slurm's job accounting feature *sacct*[15] provides a wealth of usage statistics. In the following this feature is harnessed to estimate hardware utilization by analyzing

[15]https://slurm.schedmd.com/sacct.html 2021-07-23

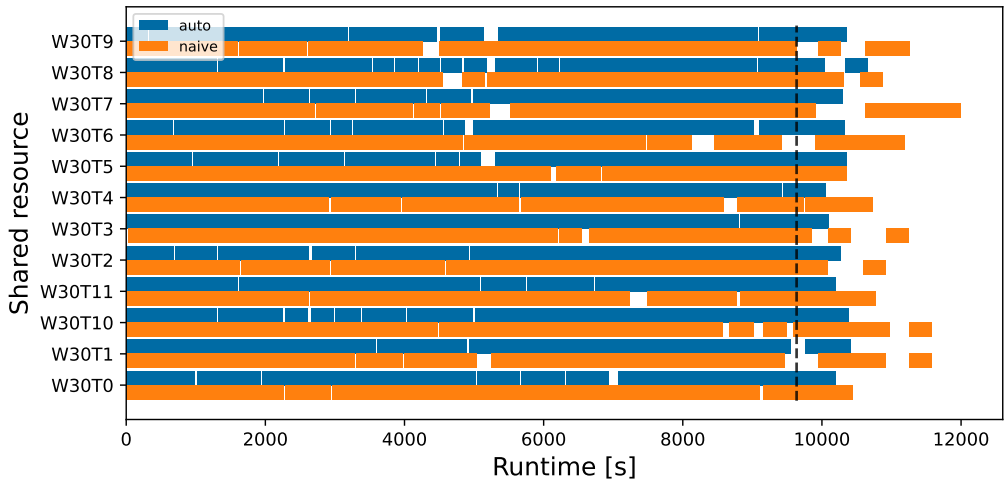


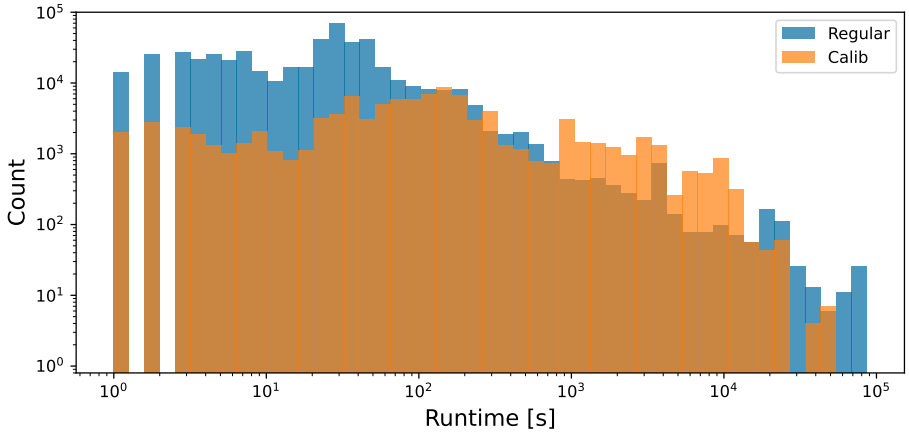
Figure 4.13: Run time of dummy calibration jobs, see fig. 4.12 for experiment setup. Each rectangle corresponds to one of 384 jobs for each HICANN calibration. Jobs are grouped by one of the constraining shared resources, the trigger for analog readout. Dashed black line denotes minimal theoretical run time. The wall-clock run time, i.e. start of first job to end of last job, in case of automated neighbor initialization feature is $10\,599 \pm 53$ s. In case of naive implementation wall-clock run time is $11\,990 \pm 140$ s. Congestion in later stages of naive implementation due to more resource allocation of individual runs, i.e., neighboring reticles are allocated for the whole run.

run time of jobs with allocated hardware resources. This also allows to further investigate the possible run time impact of the aforementioned features as well. As a reference, between 2016 and mid 2021 a total of 228 user accounts were registered for neuromorphic cluster access. Please note that, for technical reasons, all experiments conducted via the Human Brain Project (HBP) collaborative web interface[16] are performed via one account.

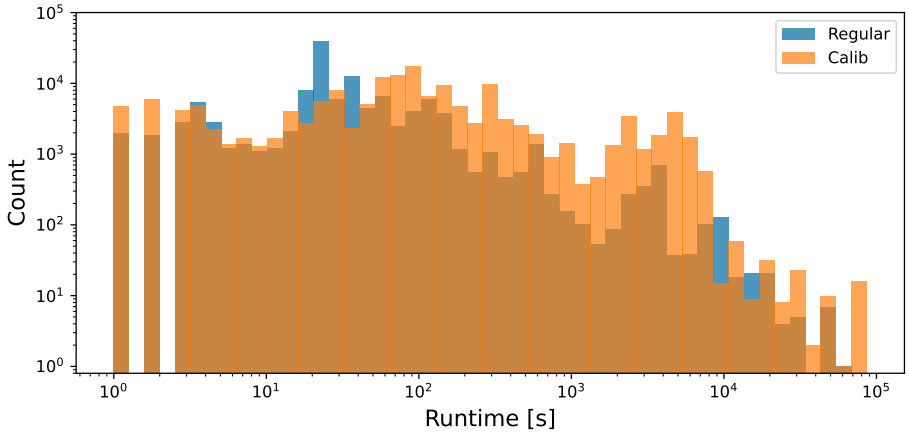
BrainScaleS-1

Figure 4.14 shows run time histograms of BSS-1 related jobs before and after introduction of the automated neighbor initialization feature. This separation was done as the additional overhead of initialization counts towards the run time statistics of a job. Overhead caused by resource isolation is present in both datasets. Calibration-related jobs are separated to provide a better understanding

[16] https://www.humanbrainproject.eu/en/silicon-brains/neuromorphic-computing-platform/ 2021-08-14



(a) Before neighbor initialization feature (2017-01-17 until 2019-07-09). Total run time: regular 222 151 h_{fpga}; calibration 15 910 h_{fpga}



(b) After neighbor initialization feature (2019-07-09 until 2021-07-09) Total run time: regular 72 487 h_{fpga}; calibration 66 478 h_{fpga}

Figure 4.14: Run time distribution of BSS-1 jobs before and after automated neighbor initialization feature. Separation is done as overhead of the feature is included in job run time. Calibration runs are separated to provide better insight for regular hardware usage. Run time in for histogram is not converted to h_{fgpa} as the actual job wall-clock time is the relevant metric.

of jobs executed for regular experiments. Additionally, the total accumulated hardware allocation time is shown. Summing up all job run times is not a sufficient metric to describe hardware allocation as jobs have a varying amount of hardware resources allocated. Analog to core hour used in HPC environments the unit FPGA hour, short h_{fpga}, is defined and used hereinafter. The total number of FPGA hours for a given job is calculated by multiplying the run time with the number of FPGA resources allocated to that job.

First, run time of regular jobs is discussed. The general distribution is influenced by the typical work flow described in section 4.1.3, i.e., short jobs dominate due to testing and prototyping of experiments. Prior to the neighbor initialization feature, most runs are in the 20-40 s range which is expected as chip configuration takes about 20 seconds. This shows that for most runs the resource isolation overhead of O(1 s) is acceptable.

Comparing distributions of calibration jobs and regular jobs, a shift of distribution towards longer run times is visible. This can be explained as jobs submitted in the calibration partition are less of a prototyping nature and more systematical, e.g., parameter sweeps. Especially after introduction of the neighbor initialization feature there are two distinct peaks visible at 2400 s (40 min) and 5400 s (90 min) which correspond to the two separate calibration steps.

Comparing total run time of regular jobs between fig. 4.14a and fig. 4.14b yields a decline of roughly 58% in utilization. Reason for this decline in regular experiments is the transition to the newer BSS-2 systems. However, utilization for calibration runs nearly doubled, which stems from two causes. First, resources temporally allocated for automated neighbor initialization are tracked by job accounting for the whole run time. On average, a HICANN has 1.23 neighboring reticles, see fig. 4.4 and fig. 4.6. Assuming most calibration jobs after neighbor feature where single-chip allocations yields an over utilization by a factor of 2.23. Second reason for higher utilization is the extended development of the calibration framework conducted by Jose Montes and Hartmut Schmidt.

BrainScaleS-2

Figure 4.15 shows job run times for the BSS-2 HICANN-X single-chip systems which are active since May 2019. Again, run time also includes resource isolation overhead of roughly 900 ms(section 4.1.4) as nearly all jobs only allocate one chip. General distribution with small run times dominating follows the same work flow principal as described for BSS-1. Most run times between 5-7 s expected from baseline estimates described in section 4.1.3. Peaks at 3600 s (60 min) and $1.7 \cdot 10^5$ s (48 h) correspond to default and maximum allowed run time of jobs.

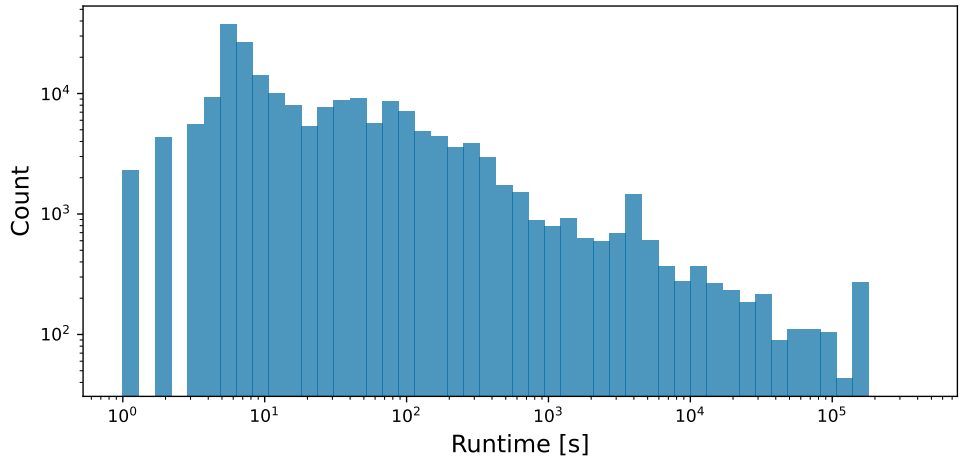


Figure 4.15: Run time distribution of BSS-2 related jobs. Total run time 44 525 h_{fpga} from 2020-02-27 until 2021-07-09.

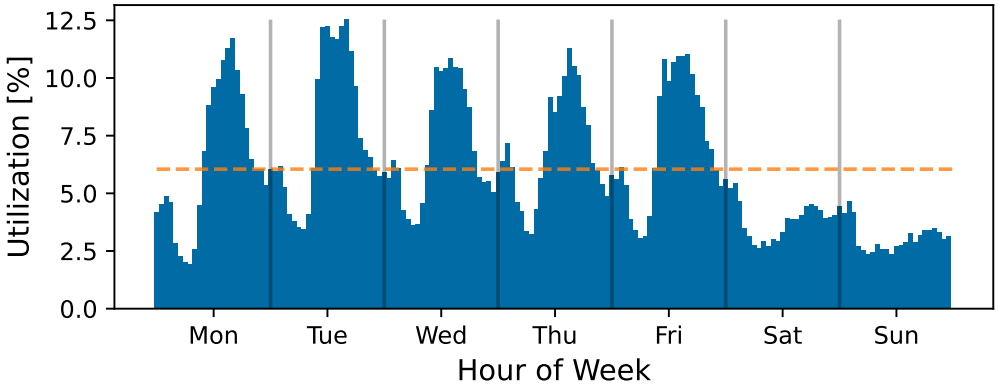
Hardware Utilization Estimate

Next the utilization of the BSS hardware setups is investigated. Of particularly interest is the distribution over time of day as it gives insight on the workflow of researchers.

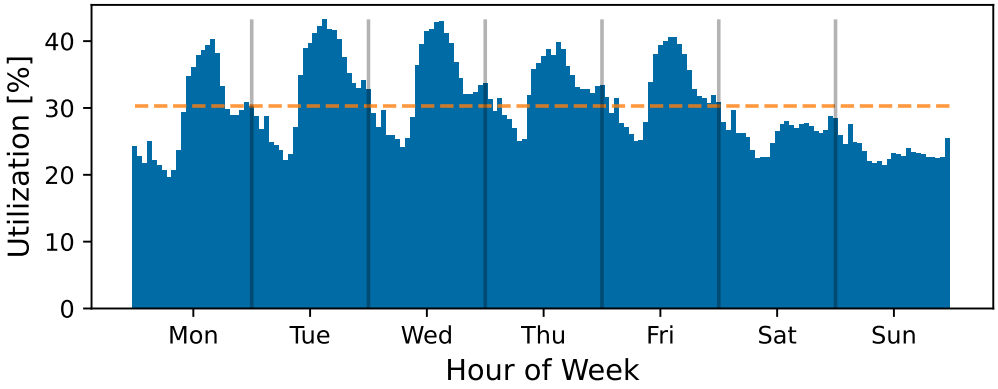
To this end, The corresponding run time portion of all hardware execution jobs is matched to the hours over a week resulting in fig. 4.16. To put the resulting aggregated run times into perspective, full theoretical utilization is estimated. The number of available setups changes over time for example because new systems are commissioned or due to maintenance. An analysis of hand written logs or change history of the hardware database would be very involved. Therefore a constant number of available hardware setups is assumed according to available setups for July 2021. Overall available time is determined by date of first and last executed job.

Several observations can be made that apply to both systems. The general distribution agrees to typical working hours of researchers, i.e., peak activity in the afternoon Monday till Friday with some activity on Saturday. Around midnight smaller peaks in utilization can be observed which are caused by automated tests running on hardware (see section 2.3).

One difference in utilization of the systems is the baseline utilization. BSS-1 wafer systems are sparsely used during the night wheres BSS-2 systems have a high base utilization also through the night. This is most probably caused by long running iterative experiments. Furthermore, overall utilization is higher for single chip systems with rough average of 30% compared to the 6% for wafer



(a) BSS-1 wafer. Full utilization estimated with 3 modules with 48 reticles each. Usage statistics from 2017-01-17 until 2021-07-09



(b) BSS-2 single chip. Full utilization estimated with 10 setups. Usage statistics from 2020-02-27 until 2021-07-09

Figure 4.16: Utilization of BSS hardware systems over the week. Gray vertical lines correspond to midnight. Orange horizontal line represents average utilization. Utilization percentage is estimated by assuming full availability of certain amount of BSS-1 and BSS-2 setups. Peak around midnight correspond to jobs for automated testing run on hardware.

modules. These numbers are rather low but not entirely unexpected as average user count is still in the same order of magnitude as hardware setups.

The conducted estimates are only an upper bound for hardware utilization as they neglect the software overhead during the hardware resource allocations. Especially in iterative experiments that perform expensive analysis computations, software overhead can be significant. Section 4.1.8 presents an approach that allows to drastically increases hardware utilization.

4.1.8 Micro Scheduler

In sections 4.1 and 4.1.3 it was argued why fixed pattern noise leads to researchers working on specific hardware setups and why iterative experiments are executed within single long-running Slurm scheduler jobs to mitigate overheads. Of course such a workflow comes with downsides. Most notably, hardware utilization is significantly diminished as experiment pre-/post-processing and analysis are executed in the same job. Section 4.1.7 showed that current overall utilization is still not at capacity, nevertheless this changes if more researchers request access to the systems. Particularly if two researcher want to work on the same setup friction is inevitable. To remedy this a micro scheduler framework, called *quiggeldy*, was developed predominantly by Oliver Breitwieser in his PhD thesis [Breitwieser 2021]. Contribution of the author was during conceptualization and integration into the *PyNN* back end (see section 3.5.1). Principle of this micro scheduler is shortly presented for sake of completeness, for a detailed explanation refer to Breitwieser 2021.

The issue of blockage by prolonged hardware allocation is tackled in two steps. First, the actual hardware execution is automatically separated from the remaining experiment. Secondly, these hardware execution segments are then sent to and managed by a micro scheduler instance for each hardware setup. The principle is explained by fig. 4.17 in more detail.

The main advantage of higher hardware setup utilization is evident, however this approach brings also other upsides. Long running sweep experiments now do not completely block a setup allowing other experiments to be interleaved. This is especially helpful during the prototyping phase of experiment script development. To this end, the experiment state can be tracked, to a certain extend, and reapplied in case another experiment was interleaved. Furthermore, the serialization between experiment script and scheduler instance facilitates cross platform support. The scheduler instances only depend on the low-level communication layer of the software stack (see section 3.2). This significantly reduces implementation effort for other computer architectures, for example , ARM. As the scheduler instances themselves are running in Slurm jobs benefits from all features explained in previous sections are exploited.

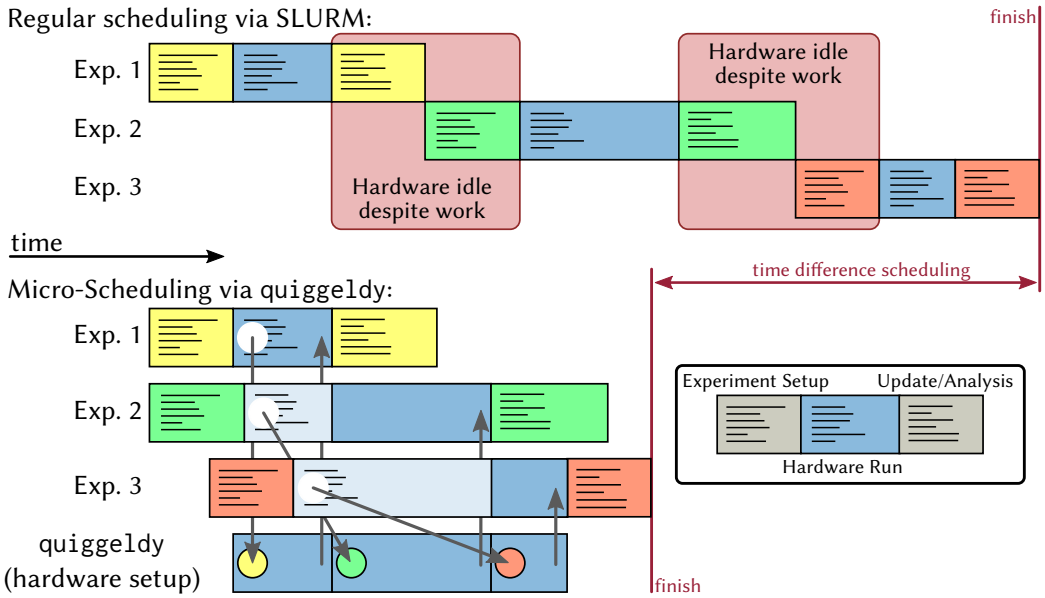


Figure 4.17: Principle of the *quiggeldy* experiment micro scheduler. The bottom right shows the 3 principle steps each experiment has, where experiment setup and analysis are not running on the neuromorphic hardware. The top illustrates the downsides of whole experiment execution in one Slurm job. Three separate experiments utilizing the same hardware are executed sequentially, thereby increasing overall run time and wasting hardware resources. The bottom left demonstrates the advantages of separated hardware execution. The same three experiments as before are executed however with the micro scheduler in place. For each experiment the hardware execution is not performed by the experiment script itself but serialized and sent to a scheduler instance corresponding to the requested hardware. Once finished, the instances then send the resulting data back to the experiment scripts. The scheduler instances themselves are running in a Slurm job and therefore benefit from all features explained in previous sections. The automated separation of hardware execution is facilitated by an additional opaque communication back-end (see section 3.2). It serializes the hardware instruction sequences generated by the upper layers and transports it to the scheduler instances via standard TCP. Taken from Breitwieser 2021.

This micro scheduler however is not yet fully integrated into the current software stack. Nevertheless its advantages and capabilities could be demonstrated on a workshop during the NICE 2021 conference[17]. There, over 80 000 individual experiments were executed by 66 participants utilizing shared access to 8 hardware setups in two time slots of 3 hours each.

4.2 Monitoring and Alerting

Measuring the conditions under which an experiment was performed is crucial for all scientific work. Knowing as many system parameters as possible is also vital for development and debugging, especially for custom analog hardware systems. Starting with small single-chip prototype setups it is still manageable to handle this on demand, i.e., manually perform data aggregation and analysis during an experiment. This becomes infeasible when migrating to larger systems due to the ever increasing number of components as well as the need for stability. In addition continuous data aggregation allows investigation of potential issues and unanticipated measurement parameters after the fact. It furthermore facilitates providing access to system state to external researchers.

Monitoring a multitude of various devices and components is a common issue in Information Technology (IT). There already exist several approaches and tools that tackle this problem. This section describes the monitoring and alerting infrastructure for the neuromorphic systems and the conventional computing cluster, which was mainly established during this thesis.

Figure 4.18 gives an overview of the monitoring infrastructure, which is described in the following sections in more detail. First, data aggregation and storage for the myriad of components is explained. Then structured visualization of this data is presented which is needed to facilitate quick analysis. Automated alerting of critical parameters and components will be described afterwards. Finally, the impact of this infrastructure on platform operation will be discussed.

This section presents the substantial restructuring and enhancement of the rudimentary monitoring framework employed prior to this thesis.

Other Contributions Development was done in close collaboration of Daniel Kutny during his bachelor thesis [Kutny 2018]. In particular, he established the event data aggregation and designed several dashboards. Furthermore, Patrick Häussermann integrated aggregation and visualization of Slurm statistics during his bachelor thesis [Häussermann 2018] supervised by the author. Maurice Güttler primarily developed the aggregation capabilities of the Raspberry Pi-based

[17] https://niceworkshop.org/nice-2021/ 2021-09-07

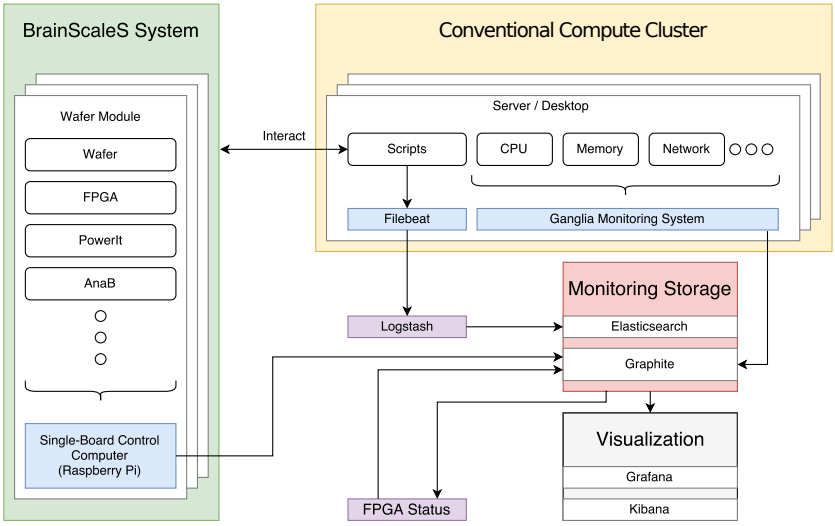


Figure 4.18: Overview of monitoring infrastructure. Various metrics of the BSS-1 system and conventional compute nodes are aggregated and stored to Graphite and *Elasticsearch*. Stored data is then visualized by *Grafana* and *Kibana*. Taken from Kutny 2018

hardware control hosts, and contributed to overall monitoring operation.

4.2.1 Aggregation and Storage

Data aggregated from neuromorphic systems as well as conventional compute nodes can be separated in two qualitatively different categories, time-series data and event data. Time series data describes data, typically numerical values, that are measured continuously with a regular time interval. Example metrics are voltage, temperature or CPU load. Event data are as the name implies events at certain points in time, powering on and off hardware entities for example. Generation, storage as well as visualization of these types of data differ greatly. Therefore different aggregation and storage tools are utilized. The different tools and strategies to handle those types of data are explained in the following.

Time series

There are two relevant sources of data generation, custom neuromorphic hardware and conventional compute nodes. The vast majority of data are time-series metrics of BSS-1 wafer modules. Custom software running on the control Raspberry Pi has the main task of managing access and operation of wafer components but also carries out data aggregation. Conventional compute nodes utilize the widely

used *ganglia* monitoring system[18] for data aggregation. *Ganglia* daemons running on nodes gather metrics like CPU-utilization, memory allocation and network throughput.

Graphite was chosen as data storage for time series data. It provides an easy to use *netcat*-based[19] workflow on the data aggregation side. From the data query point of view it provides an HTTP based URL-API[20], which provides additional functionality like sort, filter, and arithmetics. One of the important decisions for aggregation and storage is to arrange the data in an intuitive hierarchical structure to ease later querying. Figure 4.19 exemplarily shows an abridged version of the chosen metrics tree structure and some exemplary metric curves generated with *graphite* (section 4.2.2).

To efficiently store the data Graphite uses *RRDtool*[21] which implements a Round Robin Database (RRD). An RRD requires to define a sensible data retention scheme, i.e., which metrics should be stored for which time period with what resolution. Naturally one would want to store all data with the highest possible time-resolution that data aggregation allows. But one needs to compromise resolution with storage space.

The upper bound of time-resolution is defined by the aggregation tools themselves. Most components on a wafer module could be read out with sub second resolution. However, they are connected via the same Inter-Integrated Circuit Link (I2C) communication bus behind a tree of multiplexer, thus can only be read out serially. The sheer number of components therefore leads to effective resolutions in the order of 10 s. Furthermore, a maximum aggregation resolution leads to high load on the control Raspberry Pis. All these aspects lead of a maximum aggregation resolution of 60 s for on wafer components.

Yet other metrics can effectively be aggregated with higher time-resolution. Graphite even supports the definition of retention schemes for an arbitrary number of metrics. However, it was decided that more granular resolutions are not worth the overhead of individually defining retention schemes for the individual metrics. All metrics are therefore stored with a maximum resolution time of 60 s.

The nature of an RRD now allows to calculate the required storage space for given resolution. A storage space estimate is only calculated for BSS-1 wafer modules as they are the source for the vast majority of data. For each data point the time stamp and value are saved as a `float`(4 B) and `double`(8 B) respectively. Additional metadata overhead of a few bytes is neglected. Each wafer module has over 1800 metrics resulting in a storage consumption of 10.73 GB for one year with full 60 s resolution. This would already require 214.6 GB for the originally planned

[18] http://ganglia.sourceforge.net/ 2021-07-23
[19] https://nc110.sourceforge.io/ 2021-07-23
[20] https://graphite.readthedocs.io/en/latest/render_api.html 2021-07-23
[21] https://oss.oetiker.ch/rrdtool/ 2021-08-07

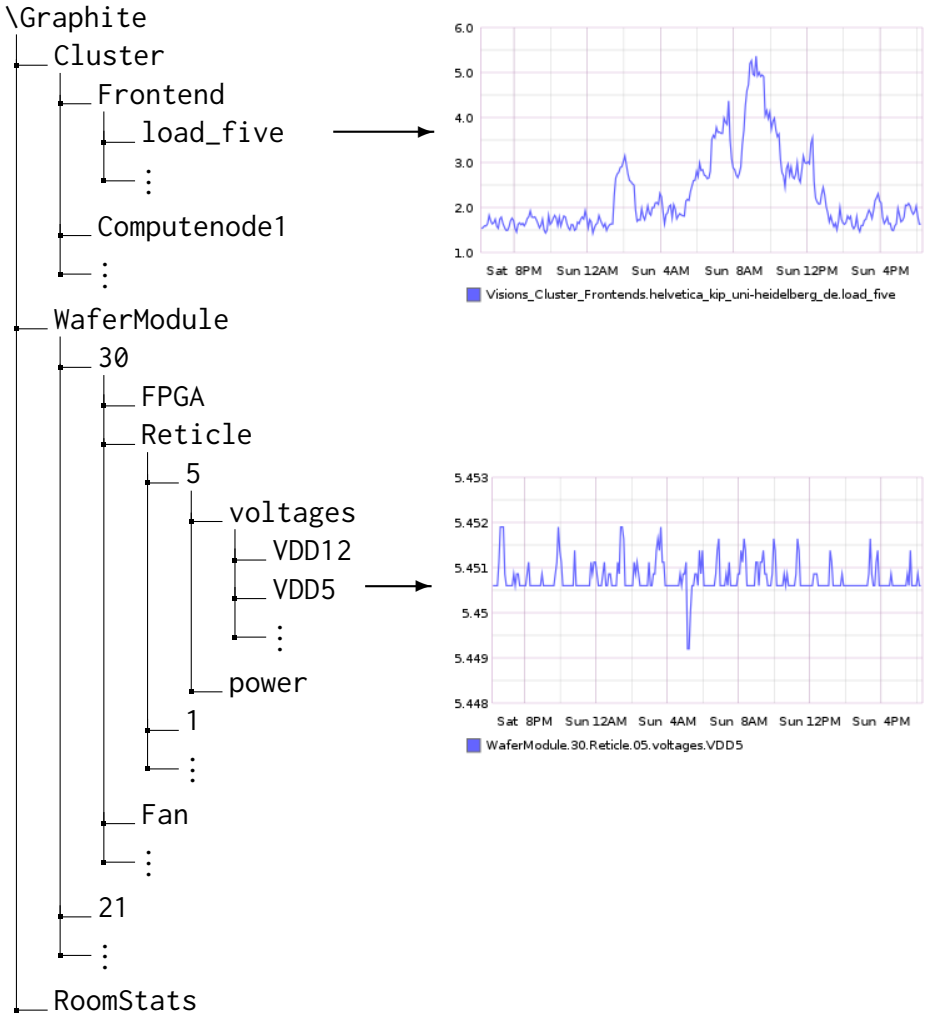


Figure 4.19: Tree hierarchy of time series metrics. Exemplary drill-down of conventional compute node statistic and chip supply voltage. Voltage is expected to be constant whereas load on node is dynamic.

operation of 20 wafer modules. A resolution of one hour was therefore chosen for older data, i.e., 10 years, resulting in an additional 1.79 GB per wafer module. This resolution still allows to observe changes within a day which typically are larger then average changes between days.

To simplify data extraction a helper script is provided allowing dumping of all metrics relevant of a wafer module in a given time range to a JavaScript Object Notation (JSON) file.

Event Data

Event data compared to time series data is not as structured but more free form. Its only defining characteristic is that *something* happened at a certain point in time. That *something* can be for example alerts, error messages or control events like powering some hardware component on or off. To manage this free form data the widely spread open source ELK-stack (*Elasticsearch*, Logstash and *Kibana*)[22] was chosen. Where Logstash handles data aggregation from log files, *Elasticsearch* manages data storage and querying and *Kibana* visualization which is covered later on, see section 4.2.2. In principle *Elasticsearch* could also be used to store time series data albeit significantly less efficient as graphite storage and latency wise.

One of the main use cases of the ELK stack is the aggregation and analysis of log files. This workflow is utilized to implement data aggregation of events without much overhead to overlaying software, as writing syslog events is supported by many languages. Relevant tools, for example FPGA control scripts, write events to specific system log files which are further processed by Logstash. From changes to those log files Logstash generates JSON formatted queries to *Elasticsearch* via Hypertext Transfer Protocol (HTTP).

As the ELK-stack went proprietary in early 2021 a change to the open source for *OpenSearch*[23] would be desirable in the future.

4.2.2 Visualization

Aggregated data is only of value if it is analyzed. One of the easiest and quickest ways for humans to extract relevant information from data is visualization. The challenge is now to visualize the myriad of data produced by the neuromorphic systems in an clear, intuitive and structured fashion.

Graphite itself provides a visualization web Graphic User Interface (GUI) which is helpful for quick exploration of data. This feature is used especially in early commissioning of the monitoring and when adding new data types.

[22] https://www.elastic.co/elastic-stack/ 2021-08-10
[23] https://www.opensearch.org/ 2021-08-10

However, persistent dashboards are needed to fully exploit the aggregated data. *Kibana* the analytics and visualization tool of the ELK stack used for event data provides such dashboards but does not support graphite as a storage back-end. It is therefore mainly used to quickly explore event data. *Grafana*[24] was chosen as central visualization front end as it provides highly customizable dashboards, fully supports time-series data and also event data albeit to a limited extend. Access to *Grafana* is managed via Lightweight Directory Access Protocol (LDAP) authorization allowing users to simply use the same account as for platform access.

Monitoring Dashboards

Monitoring dashboards provide an highly customizable, intuitive and structured way to visualize the plethora of data produced by the neuromorphic systems as well as conventional infrastructure. They supply various visualizations tools like line graphs, bar plots, pie charts or tables and allow easy selection of time ranges. In the following exemplary dashboards are presented that provide researchers better insight into the systems.

The central dashboard for BSS-1 wafer systems is shown in fig. 4.20. It aims to present most relevant information of a single wafer module on one page allowing to analyze system health on a short glance. Links to more detailed dashboards and vice versa are given allowing quick switching to more relevant information in a drill-down fashion. Augmenting time series data with event data helps identifying correlations, as seen with power on and off events from FPGAs.

Another feature that is used to great effect are templates[25]. They allow to define parametrized dashboards, where parameters are for example wafer and/or reticle IDs. This provides a quick and convenient way to switch between systems or compare metrics of different components and systems. For example a researcher can easily compare supply voltages of different chips to identify potential outliers as shown in fig. 4.21.

Monitoring is also vital for maintenance of the computing platform. Figure 4.22 shows an overview dashboard of cluster health. It provides information of relevant metrics like machine load, resource scheduler utilization and network traffic.

There are many additional dashboards providing more detail to different system aspects like FPGAs, individual cluster nodes or overall scheduler stats which are not shown for abbreviation.

Monitoring is always changing as the systems and the needs of researchers are ever evolving. However, yet another advantage of *Grafana* is its relative low learning curve allowing researcher to contribute and create their own dashboards.

[24] https://grafana.com/ 2021-08-12

[25] https://grafana.com/docs/grafana/latest/variables/ 2021-08-12

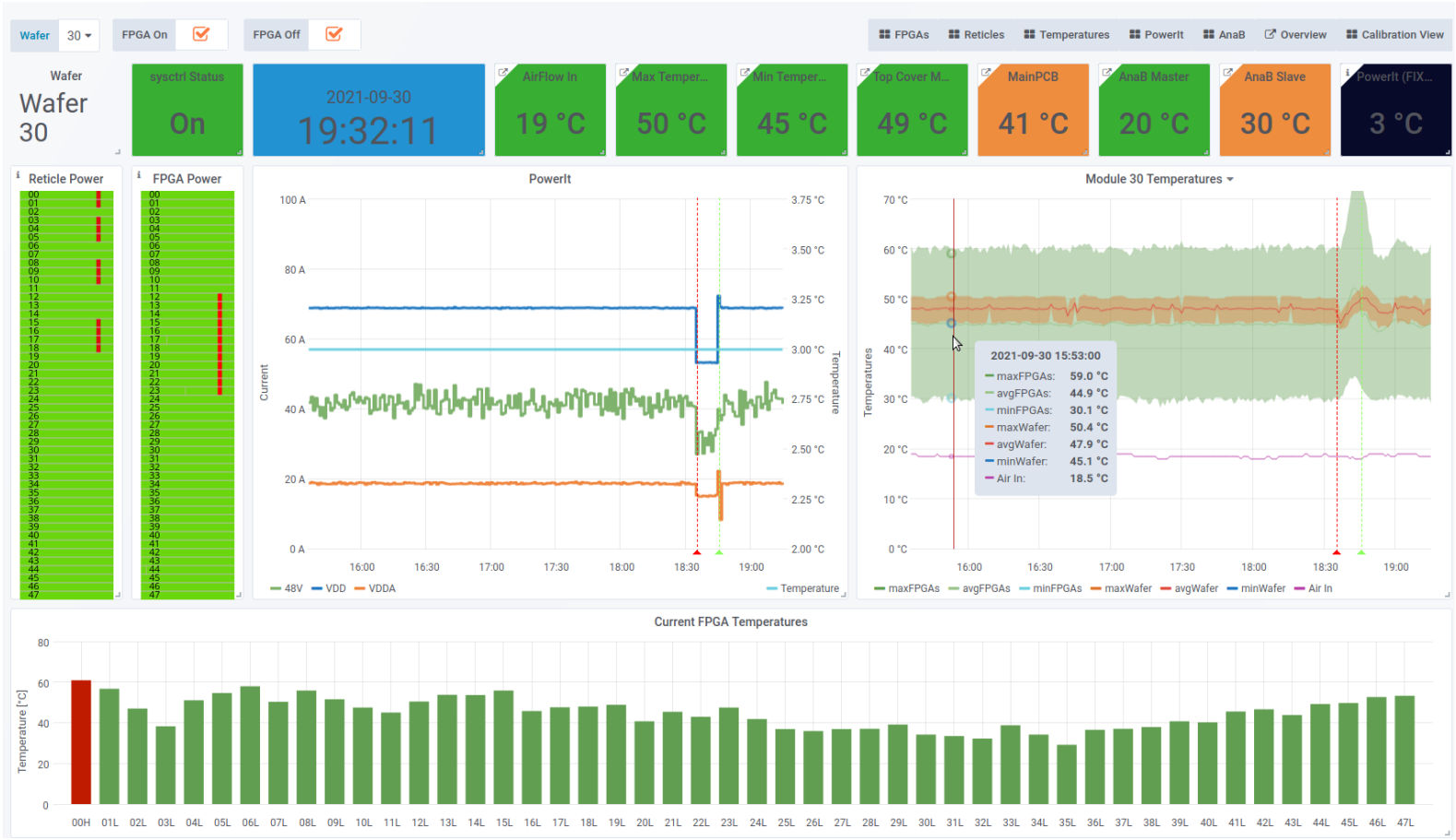


Figure 4.20: BSS-1 wafer module overview dashboard. Central landing page for wafer module monitoring. Header: Wafer module ID drop-down list. Links to other relevant detailed dashboards providing drill-down navigation. Top row: Temperatures of various components with links to detailed dashboards. Middle row: Reticle and FPGA power state history. Line graphs of power supply board metrics. Wafer, FPGA and cooling fan intake temperature profile (average, min, max). Vertical lines correspond to FPGA power up/down events. Bottom row: Current temperature of all individual FPGAs, color coded with critical temperature.

Figure 4.21: Dashboard providing overview of BSS-1 reticle related voltages. Header: Wafer module ID and reticle number drop-down list. Allows to compare arbitrary sets of reticles on a wafer module. Top row: History of on-off state of reticles Right middle: Voltages relevant for FPGA-to-chip high speed connection. Right bottom: Voltages relevant for on wafer communication. Left bottom: Remaining voltages related to reticles.

Figure 4.22: Cluster utilization overview dashboard. Top Left: Cluster server and node load. Top Right: Scheduler usage stats and overall activity. Bottom: Various network statistics of cluster nodes and switches.

4.2.3 Alerting

No software nor hardware is free of bugs, especially not if custom built. With increasing number of components the probability of things going awry increases. Simultaneously overseeing monitoring data manually becomes impossible. Automated checking of parameters and subsequent alerting, i.e., notifying platform operators and users, is essential. Yet another reason that *Grafana* was chosen is its built in alerting framework. It allows to define thresholds for arbitrary metrics and sends notifications either via email or to the group-managed chat. This quick and direct feedback channel to researchers is valuable as it reduces down-time of experiments and saves people from prolonged bug hunts.

One example for metrics that have alerting in place are different temperatures of wafer modules. Another one are supply voltages that are checked for their absolute value as well as their time-derivate. Small shifts in voltage would not necessarily reach an undesired absolute value but are nonetheless not expected. Furthermore, alerts are set up for conventional compute nodes to ensure proper platform operation. Examples are load of compute nodes and severs or checking for free disk space.

It is to note that altering is not solely relied upon concerning damage to hardware but is regarded as a first line of defense. All crucial components have built in automatic shutdown in case dangerous values are reached.

4.2.4 Findings

The following section presents examples where monitoring was essential in identifying issues with the systems.

Wafer Temperature Gradients

Behaviour of the analog neuron circuits, especially the exponential term, is temperature dependent. Therefore, care is taken to keep wafer modules at a constant temperature. To this end a PID-controller of the cooling fans was implemented with a target temperature of 50 °C. There are several factors leading to varying temperature gradients on individual BSS-1-wafer setups. For example AC-units and heat sources are not spread evenly over the machine room. Based on monitoring data a strategy for a more constant cooling was devised. Increasing ambient temperature of the machine room as well as reducing cooling fan speed resulted in a smaller more stable gradient over wafer modules.

Reticle Voltage Drifts

Initial symptoms of stochastically failing high speed links to reticles where discovered through the regular creation of calibration and blacklisting data. Investigation of the monitoring data reveled irregular fluctuations in various supply voltages for the reticles. Figure 4.23 shows an exemplary monitoring excerpt of such a drift.

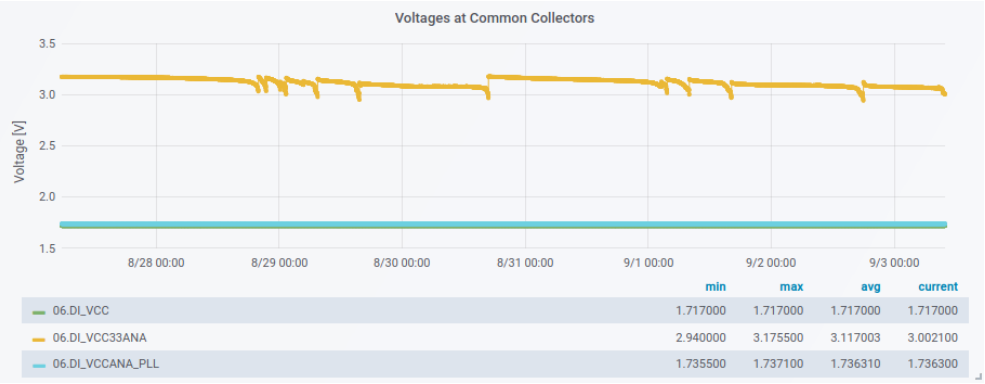


Figure 4.23: Exemplary monitoring data showing irregular drops in supply voltage of reticle 6 on wafer module 33. Year of monitoring data is 2020.

Suspected cause of these fluctuations are failing plug contact in turn caused by vibrations induced by the high rotation frequency cooling-fans custom mounted to wafer modules. Improving contact of the plugs by re-pressing them onto the wafer modules temporarily reverted behaviour back to normal. Directly bridging the plug contacts through soldered wires permanently stabilized the voltages further indicating bad plug contact as the cause. A more permanent solution of improving damming between cooling fans and wafer modules is under investigation. Discovery of these voltage drops greatly benefited from the continuous monitoring of all voltages and the insight from dashboards.

Chapter 5

Conclusion and Outlook

Neuromorphic hardware promises energy-efficient and fast simulation of neural networks compared to classical von-Neumann architectures. However, these advantages come with trade-offs like neuron model and topology constraints or device variability. Fully exploiting these novel devices therefore requires sophisticated software managing their operation. The ultimate goal is that researchers can efficiently describe experiments and the software automatically handles hardware specifics.

To this end, one of the main achievements of this thesis is the facilitation of the second-generation BrainScaleS accelerated neuromorphic system through a performant, scalable and sustainable software architecture covered in chapter 3. First, the ramifications of neuromorphic hardware and typical experiment workflow, defining the design constraints, are introduced in 3.1.1. Especially the unique acceleration factor of 10^3 results in high throughput and low latency demands. Furthermore, the multi-year multi-person development effort of hard- and software requires high emphasis on sustainability. This is especially true in light of multi-chip operation and wafer-scale integration. Therefore, much deliberation is put into the design of a software architecture with well-defined layers, outlined in section 3.1.2. These well-defined layers facilitate modular design and provide suitable access points to tackle various issues at different stages of development. At the lowest level, a communication interface facilitates data exchange for multiple back-ends like neuromorphic hardware or simulators for hardware/software co-design. Section 3.3 presents the abstraction of configuration space and runtime control of the hardware. It provides an interface encapsulating the hardware operation as a timed sequence of read and write instructions. This interface is facilitated by coordinate-container pairs which uniquely describe the configuration of individual hardware components. Furthermore, a hardware database is utilized as a single source of truth. Building on top of the hardware abstraction an experiment description layer is presented in section 3.4. It provides means

for high-level network description and therefore allows constructing neural networks intuitively in a graph-based manner as populations and projections. The translation from this high-level description to a valid hardware configuration is managed by place and route algorithms. To ease usage for non-hardware-experts the back-end agnostic modelling frameworks *PyNN* and *PyTorch* are supported. These generalized interfaces and the accompanying trade-offs are discussed in section 3.5. Full stack measurements conducted in section 3.6 verify suitable performance as well a scalability. Section 3.7 finally demonstrates soft- and hardware capabilities via a winner-take-all network implementing a Sudoku solver. All 36 minimal canonicalized 4×4 Sudoku puzzles were solved, where for 55% of puzzles the solution was found within 200 µs.

However, there are still performance and usability shortcomings which should be addressed, especially in the *PyNN* back-end discussed insection 3.6.3. One limitation is low experiment repetition rate of under 10 Hz due to unnecessary repeated chip configuration on successive experiment runs. This however can be addressed by omitting redundant configurations for example via utilization of a dirty-flag pattern. Experiment rate could be improved further by batching multiple iteration steps into a single hardware execution, elaborated in section 3.5.1. Regarding usability, convenient native support for the on-chip PPU would be desirable. In particular, auto generation of PPU programs that implement customizable plasticity rules would facilitate a variety of learning experiments.

The developed software stack further opens the possibility for external researchers, and not only for experts, to exploit the advantages of BSS neuromorphic hardware. However, operation of the BSS systems still requires system-expert intervention. Consequently, the custom hardware cannot simply be shipped to and easily utilized by other research groups. Therefore, the Electronic Visions(s) group, as part of the Human Brain Project, strives to provide access to these systems as computing platforms for the scientific community. The aspiration on usability of the platforms is that they behave much like any conventional HPC cluster. Users should be able to conveniently schedule experiments on the desired neuromorphic hardware and can trust in correct execution.

Chapter 4 describes measures taken for providing such a platform while incorporating the characteristics of analog hardware. The high customizability of the widespread resource scheduler Slurm was exploited to handle robust multi-user access for BSS-1 wafer systems as well as BSS-2 single-chip systems. Its setup and configuration was tuned to fit the workflow on the neuromorphic hardware, i.e., low iterations cycles during prototyping and high throughput for batch operation. Slurm's plugins as well as prolog and epilog frameworks are extensively utilized to manage the hardware specifics. Additional command line options are added to the scheduler submit tools so that researchers can natively request particular hardware instances. To ensure no interference between

experiments a resource isolation mechanism was added. Access to the hardware is only granted within a job if the corresponding request was made via the scheduler. A particular need of the BSS-1 wafer modules, namely the necessity for neighboring chips to be initialized, is also automatically handled. All features are thoroughly investigated to verify no disruptive overhead to job run time is introduced. Scheduler usage statistics are analyzed estimating an upper bound to hardware utilization of 7% and 30% for BSS-1 and BSS-2 respectively assuming round-the-clock availability. To increase utilization and support preemption of iterative experiments a micro-scheduler for BSS-2 systems was developed on top of Slurm. In sum, these features assure experimenters a well-defined environment to work in.

Limitations of the current implementation are its hard-coded interface for hardware requests. It was originally only designed with BSS-1 wafer scale usage in mind. This could be remedied by migration to the hardware database interface which incorporates both systems.

Another crucial requirement to a scientific computing platform is reproducibility. The high complexity of the custom hardware systems results in a myriad of components which need to be monitored. For each BSS-1 wafer module alone over 1800 metrics can be observed. The multitude of components furthermore increase the probability for faults. An extensive monitoring framework is therefore also vital for robust operation of the platform. Another achievement of this thesis was the development of such a monitoring infrastructure. Readily available open-source monitoring services were adapted to the specific requirements of the experiment platform. It manages aggregation of time-series and event data of the neuromorphic devices as well as conventional compute infrastructure. Time-series data comprises voltages, temperatures or control host load and event data includes for example powering on/off certain devices. The plethora of time-series data are efficiently stored via a Round Robin Database culminating in about 13 GB storage requirement per wafer module over 10 years. Key feature of good monitoring is an intuitive and structured way of presenting the data. To this end, visualization dashboards are designed allowing experimenters to quickly navigate and analyze the system state. Nevertheless, manual oversight of such a large amount of data becomes unfeasible. Therefore, alerting is employed to increase system health and stability. Example cases demonstrated the value of the developed monitoring infrastructure.

One shortcoming of the current monitoring is limited aggregation and exploitation of event data. For example analysis of software and system stability could greatly profit if error cases would be automatically logged to the event database.

The sophisticated software architecture combined with robust platform operation, established in the course of this thesis, enabled, at least in part, several

studies not only by group members but also external researchers [Bohnstingl et al. 2019; Czischek et al. 2021; Klassert et al. 2021].

Outlook

The next large step for the BSS-2 systems will be the facilitation of multi-chip operation to enable experiments beyond 512 neurons. In a first intermediate approach multiple single-chip setups are interconnected via FPGA-to-FPGA. The current Gigabit-Ethernet-based host-FPGA communication method is not suitable for this application as it is not designed for low-latency real-time applications. To remedy this, an alternative interconnection via the *EXTOLL* [Neuwirth et al. 2015] technology is under development.

This has varying implications on the different software-layers. On the one hand the communication layer simply has to utilize a different back-end. Likewise, only minimal change is necessary to the hardware abstraction layer as it encapsulates a single chip instance. From the modelling wrapper perspective no change to the topology description is needed. Ideally, researchers can simply request more neurons in their experiments. Most changes are required in the experiment description layers, especially place and route algorithms need to be enhanced significantly. An important aspect of multi-chip operation is the additional delay of spike transfer between the chip instances during the continuous network emulation. Synaptic and axonal delay has an influence on SNNs functionality [Xu et al. 2013] and is thus shortly estimated. As a base reference, the on-chip spike communication takes in the order of 50 ns[1]. Event transfer between chip and FPGA introduces a delay of roughly 300 ns [Karasenko 2020, p110]. *EXTOLL* based FPGA-to-FPGA connection adds about 60 ns per hop [Resch et al. 2014, p133]. This results in a delay between two setups of roughly 1 µs, an order of magnitude higher than on-chip delay. More notably it approaches typical membrane and synaptic time constants which are in the order of 10 µs. Consequently, the additional delay might affect network dynamics depending on distribution of neurons over single chip instances. Therefore, place and route strategies need to factor in these different delays. A first naive approach could be placement of the neurons of individual populations only on the same chip instances or with same hop distance. The relatively contained extent of required changes further demonstrates the advantages of the developed multi-layer software architecture.

Multi Compartment (MC) models provide an interesting prospect to neuroscience research [London et al. 2005]. Yet, software simulations of MC neuron models are significantly more involved than for single point neurons. The BSS-2 chips are equipped with inter-compartment capabilities to tackles these issues.

[1] Personal correspondence with chip designer Sebastian Billaudelle

However, the software implementation does not yet support these capabilities on the experiment description layer. Especially the parameter translation from abstract model to hardware units will pose a challenge. As such a translation depends on calibration which in turn depends on the specific circuit instances, coupling of calibration and place and route strategies should therefore be a high priority for further software development. Furthermore, depending on the complexity of the models, the number of compartments vary by 3 orders of magnitude, up to 1000 [Branco et al. 2010; Miyasho et al. 2001], individual chip resources can quickly be exhausted or be insufficient. A significant scale up of the systems is therefore desirable in particular to better explore multi-compartment capabilities, albeit this does not cover cases requiring more compartments than on chip neuron circuits.

A more scalable approach than inter-connected single chips is the direct interconnection of dozens or hundreds of chips on the same PCB or even same silicon substrate. The former is used by several large-scale neuromorphic systems [Furber et al. 2012; Intel 2020], whereas the latter could be realized through wafer-scale integration as it was achieved for the BSS-1 systems. In either case, the established platform infrastructure and software stack is well-equipped to support the upscaling of the BSS-2 neuromorphic systems.

One key result of the Human Brain Project is the introduction of the EBRAINS collaboratory, which will provide a common interactive workflow to various HPC sites and neuromorphic platforms. The results of the presented work facilitate the integration of the BSS-2 hardware into this infrastructure, thereby creating a low threshold entry point for the neuroscience community.

The emphasis on sustainable software development practices employed throughout this thesis will ensure long-term usability of the BSS-2 accelerated neuromorphic research platform.

Appendix A

Contributions

A.1 Publications

The author contributed to the following publications (* marks equal contributions):

Peer-reviewed

Timo Wunderlich*, Akos F. Kungl*, Eric Müller, Andreas Hartel, Yannik Stradmann, Syed Ahmed Aamir, Andreas Grübl, Arthur Heimbrecht, Korbinian Schreiber, David Stöckel, Christian Pehle, Sebastian Billaudelle, Gerd Kiene, **Christian Mauch**, Johannes Schemmel, Karlheinz Meier, Mihai A. Petrovici **Demonstrating Advantages of Neuromorphic Computation: A Pilot Study**, *Frontiers in Neuroscience — Neuromorphic Engineering, 36 March 2019 Volume 13 pages 260*, 2019, https://doi.org/10.3389/fnins.2019.00260

Contribution: The Author contributed to the development of the BSS-2 software architecture for HICANN-DLS covered in section 3.3.

Akos F. Kungl, Sebastian Schmitt, Johann Klähn, Paul Müller, Andreas Baumbach, Dominik Dold, Alexander Kugele, Eric Müller, Christoph Koke, Mitja Kleider, **Christian Mauch**, Oliver Breitwieser, Luziwei Leng, Nico Gürtler, Maurice Güttler, Dan Husmann, Kai Husmann, Andreas Hartel, Vitali Karasenko, Andreas Grübl, Johannes Schemmel, Karlheinz Meier and Mihai A. Petrovici, **Accelerated Physical Emulation of Bayesian Inference in Spiking Neural Networks**, *Frontiers in Neuroscience — Neuromorphic Engineering, 14 November 2019 Volume 13 pages 1201*, 2019, https://doi.org/10.3389/fnins.2019.01201

Contribution: The Author contributed to the development of the BSS-1 software architecture. Its contents are not discussed in this thesis.

Sebastian Billaudelle*, Yannik Stradmann*, Korbinian Schreiber*, Benjamin Cramer*, Andreas Baumbach*, Dominik Dold*, Julian Göltz*, Akos F. Kungl*, Timo C. Wunderlich*, Andreas Hartel, Eric Müller, Oliver Breitwieser, **Christian Mauch**, Mitja Kleider, Andreas Grübl, David Stöckel, Christian Pehle, Arthur Heimbrecht, Philipp Spilger, Gerd Kiene, Vitali Karasenko, Walter Senn, Mihai A. Petrovici, Johannes Schemmel, Karlheinz Meier, **Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate**, *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, https://doi.org/10.1109/ISCAS45731.2020.9180741

Contribution: The Author contributed to the development of the BSS-2 software architecture for HICANN-DLS covered in section 3.3.

Philipp Spilger, Eric Müller, Arne Emmel, Aron Leibfried, **Christian Mauch**, Christian Pehle, Johannes Weis, Oliver Breitwieser, Sebastian Billaudelle, Sebastian Schmitt, Timo C. Wunderlich, Yannik Stradmann, Johannes Schemmel **hxtorch: PyTorch for BrainScaleS-2**. *Gama J. et al. (eds) IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning. ITEM 2020, IoT Streams 2020. Communications in Computer and Information Science, vol 1325. Springer, Cham.*, 2020, https://doi.org/10.1007/978-3-030-66770-2_14

Contribution: Development of BSS-2 software architecture for HICANN-X covered in chapter 3. **Contribution:** The Author contributed to the development of the BSS-2 software architecture for HICANN-X covered in section 3.3.

Johannes Weis, Philipp Spilger, Sebastian Billaudelle, Yannik Stradmann, Arne Emmel, Eric Müller, **Oliver Breitwieser**, Andreas Grübl, Joscha Ilmberger, Vitali Karasenko, Mitja Kleider, Christian Mauch, Korbinian Schreiber, Johannes Schemmel, **Inference with Artificial Neural Networks on Analog Neuromorphic Hardware**. *Gama J. et al. (eds) IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning. ITEM 2020, IoT Streams 2020. Communications in Computer and Information Science, vol 1325. Springer, Cham.*, https://doi.org/10.1007/978-3-030-66770-2_15

Contribution: Development of BSS-2 software architecture for HICANN-X covered in chapter 3.

Preprint/Submitted

Eric Müller, **Christian Mauch***, Philipp Spilger*, Oliver Julien Breitwieser*, Johann Klähn*, David Stöckel, Timo Wunderlich, Johannes Schemmel **Extending BrainScaleS OS for BrainScaleS-2**. *arXiv preprint arXiv:2003.13750*, 2020,

https://arxiv.org/abs/2003.13750

Contribution: This study discusses in detail the concepts described in chapter 3, in particular section 3.3. Submitted to the ACM Journal on Emerging Technologies in Computing Systems.

Eric Müller*, Sebastian Schmitt*, **Christian Mauch***, Sebastian Billaudelle, Andreas Grübl, Maurice Güttler, Dan Husmann, Joscha Ilmberger, Sebastian Jeltsch, Jakob Kaiser, Johann Klähn, Mitja Kleider, Christoph Koke, José Montes, Paul Müller, Johannes Partzsch, Felix Passenberg, Hartmut Schmidt, Bernhard Vogginger, Jonas Weidner, Christian Mayr, Johannes Schemmel **The Operating System of the Neuromorphic BrainScaleS-1 System**, *arXiv preprint arXiv:2003.13749*, 2020, https://arxiv.org/pdf/2003.13749

Contribution: Software Development for BSS-1 that is not discussed in this thesis and platform operation as well as monitoring infrastructure covered in chapter 4. Submitted to the Elsevier Journal Neurocomputing.

A.2 Supervision

The following Students where supervised, however, not all work could be included in this thesis. Nevertheless, the author wishes to thank their valuable contribution to this thesis and the BSS computing platform overall.

Towards Fast Iterative Learning On The BrainScaleS Neuromorphic Hardware System
Lukas Pilz was co-supervised by the author. Work not is contained in this thesis [Pilz 2016].

Development of a Modern Monitoring Platform for the BrainScaleS System
Daniel Kutny was co-supervised by the author. The work is covered in section 4.2 [Kutny 2018].

Integration of the Slurm workload manager into the BrainScaleS monitoring platform
Patrick Häussermann was supervised by the author. The work is covered in sections 4.1.6 and 4.2 [Häussermann 2018].

Spike-based Expectation Maximization on the HICANN-DLSv2 Neuromorphic Chip
Philipp Spilger was co-supervised by the author. The work is covered in sec-

tion 3.3 [Spilger 2018].

Analyzing and optimizing the configuration time of the BrainScaleS-1 system by implementing differential configuration
Paul Meehan was co-supervised by the author. The Work is not contained in this thesis [Meehan 2019].

PyNN for BrainScaleS-2
Milena Czierlinski was co-supervised by the author. The work is covered in section 3.5 [Czierlinski 2020].

From Neural Network Descriptions to Neuromorphic Hardware — A Signal-Flow Graph Compiler Approach
Philipp Spilger was co-supervised by the author. The work is covered in sections 3.3 to 3.5 [Spilger 2021].

Migration and Enhancement of the Advanced Lab Course on Neuromorphic Computing
Alexander Nock was supervised by the author. The work is covered in section 3.7 [Nock 2021].

Appendix B

Measurement Conditions

B.1 Software State for Performed Measurements

B.1.1 BrainScaleS-1

Measurements for BSS-1 baseline estimates in section 4.1.3 were conducted with module `nmpm-software/2021-09-08-1`. Minimal *PyNN* overhead script is given in listing B.1.

Listing B.1: Minimal BSS-1 PyNN python import

```
import pysthal
import pyhmf as pynn
from pymarocco import PyMarocco, Defects
from pymarocco.runtime import Runtime
from pymarocco.coordinates import LogicalNeuron
from pymarocco.results import Marocco
```

B.1.2 BrainScaleS-2

All measurement and experiments related to BSS-2 were performed with the git repository state given in table B.1 and the following *gerrit* changesets applied bottom to top:

- `https://gerrit.bioai.eu/c/grenade/+/15517`
- `https://gerrit.bioai.eu/c/grenade/+/15515`
- `https://gerrit.bioai.eu/c/grenade/+/15542`

- `https://gerrit.bioai.eu/c/grenade/+/15520`
- `https://gerrit.bioai.eu/c/grenade/+/15446`
- `https://gerrit.bioai.eu/c/grenade/+/15468`

Minimal *PyNN* overhead script is given in listing B.2.

Listing B.2: Minimal BSS-2 PyNN python import

```
import pynn_brainscales.brainscales2 as pynn
```

B.1.3 Slurm

Measurements for Slurm overhead in chapter 4 where conducted with the Slurm testing environment in the state given in table B.2 and corresponding added logging messages in prolog/epilog scripts.

B.1.4 Sudoku Solver

State of Sudoku solver can be found in *gerrit* changeset `https://gerrit.bioai.eu/c/fp-neurocomp/+/13216/23`. Experiment for the Sudoku solver were conducted on setup `hxcube8fpga3chip28`. Model parameters are given by the default nightly calibration presented in listing B.3. The date of used calibration file is `2021-09-13`.

Listing B.3: Settings of nightly calibration

```
def calibrate(
        connection: hxcomm.ConnectionHandle, *,
        leak: Union[int, np.ndarray] = 80,
        reset: Union[int, np.ndarray] = 70,
        threshold: Union[int, np.ndarray] = 125,
        tau_mem: pq.quantity.Quantity = 10. * pq.us,
        tau_syn: pq.quantity.Quantity = 10. * pq.us,
        i_synin_gm: Union[int, np.ndarray] = 500,
        membrane_capacitance: Union[int, np.ndarray] = 63,
        refractory_time: pq.quantity.Quantity = 2. * pq.us,
        synapse_dac_bias: int = 400,
        readout_neuron: Optional[halco.AtomicNeuronOnDLS] = None
) -> NeuronCalibResult:
```

Repository	Commit-Hash	Commit Message
pynn-brainscales	`34cf34d38ccf2d501a8d`	Adapt to extract_neuron_spikes to return a dict
haldls	`e5d62eff021b956a6ed4`	Update CommonNeuronBackendConfig default values
grenade	`d93f9fbfa3f219c196bb`	Improve time checking unique connections …
code-format	`ce6a510471f26d78a2f9`	pylint: Add hxtorch.constants.* to generated members
logger	`bc006238ecfdc483d5b9`	Add log4cxx_level_v2
halco	`77afb3bba4288a9fb9ce`	Introduce vx::v3 namespaces
hate	`b9120c53ffcda9159623`	Add indent function for multi-line string
fisch	`33460b26570fc5589b49`	Revert "Migrate hardware verification to HXv2"
ztl	`d900ab073f6aa8df4bf7`	Add .gitreview
hxcomm	`acc953574b094d90e960`	Move decoder log-messages from debug to trace
pyublas	`fb538e8c313a3f04d1a5`	Add .gitreview
pywrap	`83ddbad8a114b4730b82`	Support builds w/o generating Python bindings
rant	`7d992b2efb2e49897300`	Remove unused includes
lib-boost-patches	`2d7e07d4e74827c42d9e`	Update content description in Readme
libnux	`191f1357090071c8130a`	Add gtest-compatible testing framework
sctrltp	`03ff11557f934d8440eb`	Avoid zombie hostarq daemon processes
hwdb	`1718735ddaaf20ad7154`	Update w23 entry, including now ananas and new Pi
visions-slurm	`279c839287ca2278eb68`	Use slurmviz view in configure step directly
flange	`fcde2aafe69805487789`	Support builds w/o generating Python bindings
bss-hw-params	`cccc90d3b56e2facbc1b`	Execute jenkins jobs in -nodev app
lib-rcf	`5b16326ae30ee08a322a`	Fixes for log4cxx@0.11.0

Table B.1: State of BSS-2 software stack for all measurements. All repositories in head state except for *grenade* which has *gerrit* changes given in text.

Repository	Commit-Hash	Commit Message
config-slurm	`f84601ebc2a47901aef6`	Update licenses file
	`92e86e38a1dddd67078c`	with new creation tool
vision-slurm	`54e85324edeb071e0022`	Increase allowed
	`bdfd134c29e500edd18a`	license length

Table B.2: State of testing Slurm environment.

B.2 Compute Node Specifications

HBPHost

- Intel(R) Core(TM) i7-4771
  - 4C/8T
  - 3.5GHz base/3.9GHz boost
  - 25.6 GB/s memory bandwidth
- 32G DDR3 RAM

RyzenHost

- AMD Ryzen 7 3800X
  - 8C/16T
  - 3.9GHz base/4.5GHz boost
  - 47.68 GiB/s memory bandwidth (dual channel)
- 64G DDR4 RAM

EpycHost

- AMD EPYC 7402P
  - 24C/48T
  - 2.8GHz base/3.35GHz boost
  - 190.7 GiB/s memory bandwidth (octa channel)
- 256G DDR4 RAM

Appendix C

Acronyms

ADC	Analog-to-Digital Converter	18
AdEx	Adaptive Exponential Integrate-and-Fire	7
ANN	Artificial Neuronal Network	2
API	Application Programming Interface	12
ARM	Advanced RISC Machines	11
ASIC	Application Specific Integrated Circuit	11
BSS	BrainScaleS	2
BSS-1	BrainScaleS Generation 1	4
BSS-2	BrainScaleS Generation 2	3
CMOS	Complementary Metal-Oxide-Semiconductor	13
COBA	Conductance Based	8
CPU	Central Processing Unit	9
CUBA	Current Based	8
DSL	Domain Specific Language	53
fMRI	functional Magnetic Resonance Imaging	19
FPGA	Field-Programmable Gate Array	2
GPU	Graphics Processing Unit	2
GUI	Graphic User Interface	112
HBP	Human Brain Project	101
HICANN	High-Input Count Analog Neuronal Network	12
HPC	High Performance Computing	3
HTTP	Hypertext Transfer Protocol	112
I2C	Inter-Integrated Circuit Link	110
ICMP	Internet Control Message Protocol	86
IT	Information Technology	108
JSON	JavaScript Object Notation	112
JTAG	Joint Test Action Group	14
LDAP	Lightweight Directory Access Protocol	113

LVDS	Low-Voltage Differential Signaling	14
MC	Multi Compartment	122
ML	Machine Learning	2
MPI	Message Passing Interface	9
PPU	Plasticity Processing Unit	18
PyNN	PyNN neural network modeling language	10
RCF	Remote Call Framework	32
RRD	Round Robin Database	110
SIMD	Single Instruction Multiple Data	18
Slurm	Simple Linux Utility for Resource Management	4
SNN	Spiking Neural Network	1
SPI	Serial Peripheral Interface	31
SQL	SQL database language	93
STD	Short Term Depression	78
STDP	Spike Timing Dependent Plasticity	8
STP	Short-term Plasticity	8
TCP	Transmission Control Protocol	32
TPU	Tensor Processing Unit	2
UDP	User Datagram Protocol	86
VM	Virtual Machine	96
YAML	YAML human-readable data-serialization language	91

Appendix D

Bibliography

Abadi, Martín et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. URL: http://download.tensorflow.org/paper/whitepaper2015.pdf.

Abeni, Luca and Dario Faggioli (2020). "Using Xen and KVM as real-time hypervisors". In: *Journal of Systems Architecture* 106, p. 101709.

Akar, Nora Abi et al. (Feb. 2019). "Arbor — A Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance Computing Architectures". In: *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 274–282. DOI: 10.1109/EMPDP.2019.8671560.

Arnold, Elias (2021). "Biologically Inspired Learning in Recurrent Spiking Neural Networks on Neuromorphic Hardware". Master's thesis. Universität Heidelberg.

Bekolay, Trevor et al. (2014). "Nengo: a Python tool for building large-scale functional brain models". In: *Frontiers in Neuroinformatics* 7, p. 48. ISSN: 1662-5196. DOI: 10.3389/fninf.2013.00048. URL: https://www.frontiersin.org/article/10.3389/fninf.2013.00048.

Bellec, Guillaume et al. (2018). "Long short-term memory and learning-to-learn in networks of spiking neurons". In: *arXiv preprint arXiv:1803.09574*.

Bellec, Guillaume et al. (2019). "Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets". In: *arXiv preprint arXiv:1901.09049*.

Benjamin, Ben Varkey et al. (2014). "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations". In: *Proceedings of the IEEE* 102.5, pp. 699–716.

Billaudelle, S. et al. (Oct. 2020). "Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate". In: *2020 IEEE International Symposium*

on Circuits and Systems (ISCAS). IEEE. DOI: 10.1109/iscas45731.2020.9180741.

Bird, Christian and Alberto Bacchelli (May 2013). "Expectations, Outcomes, and Challenges of Modern Code Review". In: *Proceedings of the International Conference on Software Engineering*. IEEE. URL: https://www.microsoft.com/en-us/research/publication/expectations-outcomes-and-challenges-of-modern-code-review/.

Blundell, Inga et al. (2018). "Code Generation in Computational Neuroscience: A Review of Tools and Techniques". In: *Frontiers in Neuroinformatics* 12, p. 68. ISSN: 1662-5196. DOI: 10.3389/fninf.2018.00068. URL: https://www.frontiersin.org/article/10.3389/fninf.2018.00068.

Bohnstingl, Thomas et al. (May 2019). "Neuromorphic Hardware Learns to Learn". English. In: *Frontiers in neuroscience* 2019.13, pp. 1–14. ISSN: 1662-4548.

Branco, Tiago, Beverley A. Clark, and Michael Häusser (2010). "Dendritic Discrimination of Temporal Input Sequences in Cortical Neurons". In: *Science* 329.5999, pp. 1671–1675. DOI: 10.1126/science.1189664. URL: https://www.science.org/doi/abs/10.1126/science.1189664.

Breitwieser, Oliver (2021). "Learning by Tooling: Novel Neuromorphic Learning Strategies in Reproducible Software Environments". PHD thesis. Ruprecht-Karls-Universität Heidelberg.

Brette, R. and W. Gerstner (2005). "Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity". In: *J. Neurophysiol.* 94, pp. 3637–3642. DOI: 10.1152/jn.00686.2005.

Brüderle, Daniel (2009). "Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System". PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Chou, Ting-Shuo et al. (2018). "CARLsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters". In: *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, pp. 1–8.

Cramer, Benjamin et al. (2021). *Surrogate gradients for analog neuromorphic computing*. arXiv: 2006.07239 [cs.NE].

Czierlinski, Milena (2020). "PyNN for BrainScaleS-2". Bachelor thesis. Universität Heidelberg.

Czischek, Stefanie et al. (2021). *Spiking neuromorphic chip learns entangled quantum states*. arXiv: 2008.01039 [cs.ET].

Davies, Mike et al. (2018). "Loihi: A neuromorphic manycore processor with on-chip learning". In: *IEEE Micro* 38.1, pp. 82–99.

Davison, A. P. et al. (2009a). "PyNN: a common interface for neuronal network simulators". In: *Front. Neuroinform.* 2.11. DOI: 3389/neuro.11.011.2008.

— (2009b). "PyNN: a common interface for neuronal network simulators". In: *Front. Neuroinform.* 2.11. DOI: 3389/neuro.11.011.2008.

Demo (2021). *BSS2 PyNN single neuron demo script*. https://github.com/electronicvisions/pynn-brainscales/blob/master/brainscales2/pynn_brainscales/brainscales2/examples/single_neuron_demo.py. accessed August 25, 2021.

Diesmann, Markus and Marc-Oliver Gewaltig (2002). "NEST: An Environment for Neural Systems Simulations". In: *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*. Ed. by Theo Plesser and Volker Macho. Vol. 58. GWDG-Bericht. Göttingen: Ges. für Wiss. Datenverarbeitung, pp. 43–70.

Eklund, Anders, Thomas E Nichols, and Hans Knutsson (2016). "Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates". In: *Proceedings of the national academy of sciences* 113.28, pp. 7900–7905.

EPFL and IBM (2008). *Blue Brain Project*. Lausanne. URL: http://bluebrain.epfl.ch/.

Feldmann, Johannes et al. (2019). "All-optical spiking neurosynaptic networks with self-learning capabilities". In: *Nature* 569.7755, pp. 208–214.

Fonseca Guerra, Gabriel A. and Steve B. Furber (2017). "Using Stochastic Spiking Neural Networks on SpiNNaker to Solve Constraint Satisfaction Problems". In: *Frontiers in Neuroscience* 11, p. 714. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00714. URL: https://www.frontiersin.org/article/10.3389/fnins.2017.00714.

Friedmann, Simon (2013). "A New Approach to Learning in Neuromorphic Hardware". PhD thesis. Ruprecht-Karls-Universität Heidelberg.

— (2015). *Omnibus On-Chip Bus*. forked from https://github.com/five-elephants/omnibus. URL: https://github.com/electronicvisions/omnibus.

Furber, Steve B. et al. (2012). "Overview of the SpiNNaker System Architecture". In: *IEEE Transactions on Computers* 99.PrePrints. ISSN: 0018-9340. DOI: http://doi.ieeecomputersociety.org/10.1109/TC.2012.142.

Gerstner, Wulfram and Werner Kistler (2002). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press.

Göltz, Julian et al. (2021). *Fast and energy-efficient neuromorphic deep learning with first-spike times*. arXiv: 1912.11443 [cs.NE].

Grüning, André and Sander M Bohte (2014). "Spiking neural networks: Principles and challenges." In: *ESANN*. Citeseer.

Hatton, Les (2007). "The chimera of software quality". In: *Computer* 40.8, pp. 104–103.

Häussermann, Patrick (2018). "Integration of the Slurm workload manager into the BrainScaleS monitoring platform". Bachelorarbeit. Universität Heidelberg.

Hazan, Hananel et al. (2018). "BindsNET: A Machine Learning-Oriented Spiking Neural Networks Library in Python". In: *Frontiers in Neuroinformatics* 12,

p. 89. ISSN: 1662-5196. DOI: 10.3389/fninf.2018.00089. URL: https://www.frontiersin.org/article/10.3389/fninf.2018.00089.

HBP (2021). *HBP Neuromorphic Guidebook single demo example*. https://electronicvisions.github.io/hbp-sp9-guidebook/pm/using_pm_newflow.html#running-tenpynn-scripts. accessed August 11, 2021.

He, Horace (2019). "The State of Machine Learning Frameworks in 2019". In: *The Gradient*.

Hebb, Donald O. (1949). *The Organization of Behaviour*. New York: Wiley.

Heimbrecht, Arthur (Mar. 2017). "Compiler Support for the BrainScaleS Plasticity Processor". Bachelorarbeit. Universität Heidelberg.

Hines, M.L. and N.T. Carnevale (2003). "The NEURON simulation environment." In: *The Handbook of Brain Theory and Neural Networks*. M.A. Arbib, pp. 769–773.

Hock, Matthias (2014). "Modern Semiconductor Technologies for Neuromorphic Hardware". PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Hodgkin, Alan Lloyd and Andrew F. Huxley (Aug. 1952). "A quantitative description of membrane current and its application to conduction and excitation in nerve." In: *J Physiol* 117.4, pp. 500–544. ISSN: 0022-3751. URL: http://view.ncbi.nlm.nih.gov/pubmed/12991237.

Hu, Miao et al. (2014). "Memristor crossbar-based neuromorphic computing system: A case study". In: *IEEE transactions on neural networks and learning systems* 25.10, pp. 1864–1878.

IEEE (2001). "IEEE Standard Test Access Port and Boundary-Scan Architecture". In: *IEEE Std 1149.1-2001*, pp. i–200. DOI: 10.1109/IEEESTD.2001.92950.

Indiveri, Giacomo et al. (2011). "Neuromorphic silicon neuron circuits". In: *Frontiers in Neuroscience* 5.0. ISSN: 1662-453X. DOI: 10.3389/fnins.2011.00073. URL: http://www.frontiersin.org/Journal/Abstract.aspx?s=755&name=neuromorphic%20engineering&ART_DOI=10.3389/fnins.2011.00073.

Insel, Thomas R, Story C Landis, and Francis S Collins (2013). "The NIH brain initiative". In: *Science* 340.6133, pp. 687–688.

Intel (Mar. 18, 2020). "Intel Scales Neuromorphic Research System to 100 Million Neurons". In: URL: https://newsroom.intel.com/news/intel-scales-neuromorphic-research-system-100-million-neurons/.

Jeltsch, Sebastian (2014). "A Scalable Workflow for a Configurable Neuromorphic Platform". PhD thesis. Universität Heidelberg.

Jouppi, Norman P et al. (2017). "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12.

Kaiser, Jakob et al. (2021). "Emulating dendritic computing paradigms on analog neuromorphic hardware". Publication in Neuroscience pending.

Karasenko, Vitali (2020). "Von Neumann bottlenecks in non-von Neumann computing architectures". PhD thesis. Universität Heidelberg.

Klähn, Johann (2020). *genpybind software v0.2.0*. DOI: 10.5281/zenodo.372674. URL: https://github.com/kljohann/genpybind.

Klassert, Robert et al. (2021). "Variational learning of quantum ground states on spiking neuromorphic hardware". In: *in prep.*

Kleider, Mitja (2017). "Neuron Circuit Characterization in a Neuromorphic System". HD-KIP 17-135. PhD thesis. Universität Heidelberg. URL: http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=3657.

Koke, Christoph (2017). "Device Variability in Synapses of Neuromorphic Circuits". PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Korcsak-Gorzo, Agnes et al. (2021). *Cortical oscillations implement a backbone for sampling-based computation in spiking neural networks*. arXiv: 2006.11099 [q-bio.NC].

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

Kugele, Alexander (2018). "Solving the Constraint Satisfaction Problem Sudoku on Neuromorphic Hardware". Master's thesis. Universität Heidelberg.

Kulkarni, Shruti R et al. (2021). "Benchmarking the performance of neuromorphic and spiking neural network simulators". In: *Neurocomputing* 447, pp. 145–160.

Kutny, Daniel (2018). "Development of a Modern Monitoring Platform for the BrainScaleS System". Bachelor thesis. Ruprecht-Karls-Universität Heidelberg.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (May 2015). "Deep learning". In: *Nature* 521.7553, pp. 436–444. ISSN: 0028-0836. DOI: http://dx.doi.org/10.1038/nature1453910.1038/nature14539.

LeCun, Yann and Corinna Cortes (1998). *The MNIST database of handwritten digits*.

Leng, Luziwei et al. (2018). "Spiking neurons with short-term synaptic plasticity form superior generative networks". In: *Scientific reports* 8.1, pp. 1–11.

Leveson, Nancy G and Clark S Turner (1993). "An investigation of the Therac-25 accidents". In: *Computer* 26.7, pp. 18–41.

Li, Yibo et al. (2018). "Review of memristor devices in neuromorphic computing: materials sciences and device challenges". In: *Journal of Physics D: Applied Physics* 51.50, p. 503002.

Lin, Chit-Kwan et al. (2018). "Programming Spiking Neural Networks on Intel's Loihi". In: *Computer* 51.3, pp. 52–61.

London, M. and M. Häusser (2005). "Dendritic computation". In: *Annu. Rev. Neurosci.* 28, pp. 503–532.

Maass, W. (2014). “Noise as a Resource for Computation and Learning in Networks of Spiking Neurons”. In: *Proceedings of the IEEE* 102, pp. 860–880.

Markram, H. (2012). “The Human Brain Project”. In: *Scientific American* 306.6, pp. 50–55.

Markram, H. et al. (1997). “Regulation of Synaptic Efficacy By Coincidence of Postsynaptic Aps.” In: *Science* 275, pp. 213–215.

Markram, Henry et al. (2004). “Interneurons of the neocortical inhibitory system”. In: *Nature Reviews Neuroscience* 5.10, pp. 793–807.

Mauch, Christian (2016). “Commissioning of a Neuromorphic Computing Platform”. Masterthesis. Universität Heidelberg.

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

— (1990). “Neuromorphic Electronic Systems”. In: *Proceedings of the IEEE* 78, pp. 1629–1636.

Meehan, Paul (2019). “Analyzing and optimizing the configuration time of the BrainScaleS-1 system by implementing differential configuration”. Bachelor thesis. Universität Heidelberg.

Merali, Zeeya (Oct. 2010). “Computational science: ...Error”. In: *Nature* 467, pp. 775–7. DOI: 10.1038/467775a.

Merolla, Paul A et al. (2014). “A million spiking-neuron integrated circuit with a scalable communication network and interface”. In: *Science* 345.6197, pp. 668–673.

Minami, Shohei, Toshino Endo, and Akihiro Nomura (2021). “Measurement and Modeling of Performance of HPC Applications towards Overcommitting Scheduling Systems”. In: JSSPP. URL: https://jsspp.org/papers21/nomura.pdf.

Miyasho, Tsugumichi et al. (2001). “Low-threshold potassium channels and a low-threshold calcium channel regulate Ca2+ spike firing in the dendrites of cerebellar Purkinje neurons: a modeling study”. In: *Brain research* 891.1-2, pp. 106–115.

Moore, G. E. (Apr. 1965). “Cramming more components onto integrated circuits”. In: *Electronics* 38.8.

Müller, Eric Christian (2014). “Novel Operation Modes of Accelerated Neuromorphic Hardware”. HD-KIP 14-98. PhD thesis. Ruprecht-Karls-Universität Heidelberg. URL: http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=3112.

Müller, Eric et al. (Mar. 2020a). “Extending BrainScaleS OS for BrainScaleS-2”. In: *arXiv preprint*. arXiv: 2003.13750 [cs.NE]. URL: http://arxiv.org/abs/2003.13750.

Müller, Eric et al. (Mar. 2020b). “The Operating System of the Neuromorphic BrainScaleS-1 System”. In: *arXiv preprint*. arXiv: 2003.13749 [cs.NE]. URL: http://arxiv.org/abs/2003.13749.

— (Mar. 2020c). “The Operating System of the Neuromorphic BrainScaleS-1 System”. In: *arXiv preprint*. arXiv: 2003.13749 [cs.NE].

Neftci, Emre O., Hesham Mostafa, and Friedemann Zenke (2019). “Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks”. In: *IEEE Signal Processing Magazine* 36.6, pp. 51–63. DOI: 10.1109/MSP.2019.2931595.

Neumann, J. von (1945). *First draft of a report on the EDVAC*. Tech. rep. Transscript in: M. D. Godfrey: Introduction to “The first draft report on the EDVAC” by John von Neumann. IEEE Annals of the History of Computing 15(4), 27–75 (1993). Moore School of Electrical Engeneering Library, University of Pennsylvania.

Neuwirth, Sarah et al. (2015). “Scalable communication architecture for network-attached accelerators”. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, pp. 627–638.

Nock, Alexander (2021). “Migration and Enhancement of the Advanced Lab Course on Neuromorphic Computing”. Bachelor thesis. Ruprecht-Karls-Universität Heidelberg.

OpenAI et al. (2019). *Dota 2 with Large Scale Deep Reinforcement Learning*. arXiv: 1912.06680 [cs.LG].

Ostrau, Christoph et al. (July 2019). “Comparing Neuromorphic Systems by Solving Sudoku Problems”. In: pp. 521–527. DOI: 10.1109/HPCS48598.2019.9188207.

Paszke, Adam et al. (2019a). “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035.

— (2019b). “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Philipp, S. et al. (Sept. 2007). “Interconnecting VLSI Spiking Neural Networks Using Isochronous Connections”. In: *Proceedings of the 9th International Work-Conference on Artificial Neural Networks (IWANN’2007)*. Vol. LNCS 4507. Springer Verlag, pp. 471–478.

Pilz, Lukas (2016). “Towards Fast Iterative Learning On The BrainScaleS Neuromorphic Hardware System”. Bachelor thesis. Universität Heidelberg.

Plank, Philipp et al. (2021). *A Long Short-Term Memory for AI Applications in Spike-based Neuromorphic Hardware*. arXiv: 2107.03992 [cs.NE].

Pronold, Jari et al. (2021a). *Routing brain traffic through the von Neumann bottleneck: Efficient cache usage in spiking neural network simulation code on general purpose computers*. arXiv: 2109.12855 [cs.DC].

Pronold, Jari et al. (2021b). *Routing brain traffic through the von Neumann bottleneck: Parallel sorting and refactoring*. arXiv: 2109.11358 [q-bio.NC].

Resch, Michael M et al. (2014). *Sustained Simulation Performance 2014: Proceedings of the Joint Workshop on Sustained Simulation Performance, University of Stuttgart (HLRS) and Tohoku University, 2014*. Springer.

Rhodes, Oliver et al. (2018). "sPyNNaker: A Software Package for Running PyNN Simulations on SpiNNaker". In: *Frontiers in Neuroscience* 12, p. 816. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00816. URL: https://www.frontiersin.org/article/10.3389/fnins.2018.00816.

Rueckauer, Bodo et al. (Jan. 2021). "NxTF: An API and Compiler for Deep Spiking Neural Networks on Intel Loihi". In: *arXiv preprint*.

Rumelhart, D. E., G. E. Hinton, and Williams R.J. (1986). "Learning internal representations by error propagation". In: *Parallel Distributed Processing: Explorations in the Microstructures of Cognition* I. Ed. by D. E. Rumelhart and J. L. McClelland, pp. 318–362.

Schemmel, J. et al. (2006). "Implementing Synaptic Plasticity in a VLSI Spiking Neural Network Model". In: *Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN)*. IEEE Press.

Schemmel, Johannes et al. (2010). "A Wafer-Scale Neuromorphic Hardware System for Large-Scale Neural Modeling". In: *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1947–1950.

Schemmel, Johannes et al. (2020). "Accelerated Analog Neuromorphic Computing". In: *arXiv preprint*. arXiv: 2003.11996 [cs.NE]. URL: https://arxiv.org/abs/2003.11996.

Schreiber, Korbinian (Jan. 2021). "Accelerated neuromorphic cybernetics". PhD thesis. Universität Heidelberg.

Schuman, Catherine D. et al. (2017). *A Survey of Neuromorphic Computing and Neural Networks in Hardware*. eprint: arXiv:1705.06963.

Shi Doku (n.d.). http://sudopedia.enjoysudoku.com/Shi_Doku.html. accessed Aug 27, 2021.

Silver, David et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587, pp. 484–489.

Simakov, Nikolay A. et al. (2018). "A Slurm Simulator: Implementation and Parametric Analysis". In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Ed. by Stephen Jarvis, Steven Wright, and Simon Hammond. Springer International Publishing, pp. 197–217.

Spilger, Philipp (2018). "Spike-based Expectation Maximization on the HICANN-DLSv2 Neuromorphic Chip". Bachelorarbeit. Universität Heidelberg.

— (Feb. 2021). "From Neural Network Descriptions to Neuromorphic Hardware — A Signal-Flow Graph Compiler Approach". Master's thesis. Universität Heidelberg.

Stein, R. (Jan. 1967). "Some Models of Neuronal Variability". In: *Biophysical Journal* 7.1, pp. 37–68. ISSN: 00063495. DOI: 10.1016/S0006-3495(67)86574-3. URL: http://dx.doi.org/10.1016/S0006-3495(67)86574-3.

Stimberg, Marcel, Romain Brette, and Dan Fm Goodman (Aug. 2019). "Brian 2, an intuitive and efficient neural simulator". In: *eLife* 8. DOI: 10.7554/eLife.47314.

Stradmann, Yannik et al. (Mar. 2021). "Demonstrating Analog Inference on the BrainScaleS-2 Mobile System". In: *arXiv preprint*. arXiv: 2103.15960 [cs.AR].

Tan, Kar-Han and Boon Pang Lim (2018). "The artificial intelligence renaissance: deep learning and the road to human-Level machine intelligence". In: *APSIPA Transactions on Signal and Information Processing* 7, e6. DOI: 10.1017/ATSIP.2018.6.

TBB (2021). *TBB flow-graph documentation*. https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation/top/intel-threading-building-blocks-developer-reference/flow-graph/overview.html. accessed September 18, 2021.

Thakur, Chetan Singh et al. (2018). "Large-Scale Neuromorphic Spiking Array Processors: A Quest to Mimic the Brain". In: *Frontiers in Neuroscience* 12, p. 891. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00891. URL: https://www.frontiersin.org/article/10.3389/fnins.2018.00891.

Theis, Thomas N. and H.-S. Philip Wong (2017). "The End of Moore's Law: A New Beginning for Information Technology". In: *Computing in Science Engineering* 19.2, pp. 41–50. DOI: 10.1109/MCSE.2017.29.

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

Venters, Colin C et al. (2014). "Software sustainability: The modern tower of babel". In: *CEUR Workshop Proceedings*. Vol. 1216. CEUR, pp. 7–12.

Wang, Runchun M, Chetan S Thakur, and André van Schaik (2018). "An FPGA-based massively parallel neuromorphic cortex simulator". In: *Frontiers in neuroscience* 12, p. 213.

Werbos, Paul J (1990). "Backpropagation through time: what it does and how to do it". In: *Proceedings of the IEEE* 78.10, pp. 1550–1560.

Wunderlich, Timo et al. (2019). "Demonstrating Advantages of Neuromorphic Computation: A Pilot Study". In: *Frontiers in Neuroscience* 13, p. 260. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00260. URL: https://www.frontiersin.org/article/10.3389/fnins.2019.00260.

Xu, Bo, YuBing Gong, and BaoYing Wang (2013). "Delay-induced firing behavior and transitions in adaptive neuronal networks with two types of synapses". In: *Science China Chemistry* 56.2, pp. 222–229.

Yakopcic, Chris et al. (2020). “Solving Constraint Satisfaction Problems Using the Loihi Spiking Neuromorphic Processor”. In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*.

Yamamoto, Keiji et al. (2014). “The K computer Operations: Experiences and Statistics”. In: *Procedia Computer Science* 29. 2014 International Conference on Computational Science, pp. 576–585. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2014.05.052. URL: https://www.sciencedirect.com/science/article/pii/S1877050914002294.

Yamazaki, Tadashi, Jun Igarashi, and Hiroshi Yamaura (2021). “Human-scale Brain Simulation via Supercomputer: A Case Study on the Cerebellum”. In: *Neuroscience* 462. In Memoriam: Masao Ito—A Visionary Neuroscientist with a Passion for the Cerebellum, pp. 235–246. ISSN: 0306-4522. DOI: https://doi.org/10.1016/j.neuroscience.2021.01.014. URL: https://www.sciencedirect.com/science/article/pii/S030645222100021X.

Yavuz, Esin, James Turner, and Thomas Nowotny (2016). “GeNN: a code generation framework for accelerated brain simulations”. In: *Scientific reports* 6.1, pp. 1–14.

Young, Aaron R et al. (2019). “A review of spiking neuromorphic hardware communication systems”. In: *IEEE Access* 7, pp. 135606–135620.

Zenke, Friedemann and Wulfram Gerstner (2014). “Limits to high-speed simulations of spiking neural networks using general-purpose computers”. In: *Frontiers in Neuroinformatics* 8.76. ISSN: 1662-5196. DOI: 10.3389/fninf.2014.00076. URL: http://www.frontiersin.org/neuroinformatics/10.3389/fninf.2014.00076/abstract.

Zhou, Xiaobing et al. (2013). “Exploring Distributed Resource Allocation Techniques in the SLURM Job Management System”. In:

Acknowledgments

The author would like to extend his gratitude to the following people:

The late Prof. Dr. Karlheinz Meier whose clear guidance made all the outstanding work conducted by the Electronic Vision(s) group possible.
Dr. habil. Johannes Schemmel for continuing the vision of Prof. Meier and for taking over the supervision of my thesis.
Prof. Dr. Hans-Christian Schultz-Coulon for agreeing to review this work as well as Prof. Dr. Tilman Plehn and Prof. Dr. Jürgen Hesser for participating in my oral examination.
Dr. Eric Müller for his invaluable contributions to the group as well as teaching me the ways of sustainable research software development.
All my proofreaders for helping me, namely Eric, Yannik, Andi, Philipp, Billi, Oli and Sebastian.
All students I had the pleasure to supervise for their work throughout this thesis.

The container crew (including the Breitwieser enclave) for cultivating a very joyful work environment.
My Slurm brother Oli for tooling our way to greatness.
My desk buddy Philipp for providing a very short feedback loop.
Joscha for starting the visionary brewing culture.

All the wonderful people who took part in visionary recreational activities that kept me sane through the pandemic.
All contributors to the visionary infrastructure for sustaining a productive workflow.
All the Visionaries for making, facilitating and putting the systems to good use.

My parents for supporting me in all my life decisions that lead me here.

GG EZ

Funding Statement

This research has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Nos. 720270, 785907 and 945539 (Human Brain Project, HBP) and from the European Union Seventh Framework Programme (FP7) under grant agreement no 269921 (BrainScaleS).

Statement of Originality (Erklärung)

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

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, October 12, 2021

Signature (Unterschrift). . . . . .