Extending BrainScaleS OS for BrainScaleS-2

Eric Müller*, Christian Mauch*, Philipp Spilger*, Oliver Julien Breitwieser*,
Johann Klähn, David Stöckel, Timo Wunderlich and Johannes Schemmel

Kirchhoff-Institute for Physics
Ruprecht-Karls-Universität Heidelberg, Germany
*contributed equally
Email: {mueller,cmauch,pspilger,obreitwi}@kip.uni-heidelberg.de

***Abstract*—BrainScaleS-2 is a mixed-signal accelerated neuromorphic system targeted for research in the fields of computational neuroscience and beyond-von-Neumann computing. To augment its flexibility, the analog neural network core is accompanied by an embedded SIMD microprocessor. The BrainScaleS Operating System (BrainScaleS OS) is a software stack designed for the user-friendly operation of the BrainScaleS architectures. We present and walk through the software-architectural enhancements that were introduced for the BrainScaleS-2 architecture. Finally, using a second-version BrainScaleS-2 prototype we demonstrate its application in an example experiment based on spike-based expectation maximization.**

I. Introduction

State-of-the-art neuromorphic architectures pose many requirements in terms of system control, data preprocessing, data exchange and data analysis. In all these areas, software is involved in satisfying these requirements. Several neuromorphic systems are directly used by individual researchers in collaborations, e.g., [1–4]. In addition, some systems are operated as experiment platforms providing access for external users [2–5].

Especially the latter calls for additional measures, such as clear and concise interfaces, resource management, runtime control and —depending on data volumes— "grid-computing"-like data processing capabilities. At the same time, usability and experiment reproducibility are crucial properties of all experiment platforms, including neuromorphic systems.

Modern software engineering techniques such as code review, continuous integration as well as continuous deployment can help to increase platform robustness and ensure experiment reproducibility. Long-term hardware development roadmaps and experiment collaborations draw attention to platform sustainability. Technical decisions need to be evaluated for potential future impact; containing and reducing technical debt is a key objective during planning as well as development. Regardless of being software-driven simulations/emulations, or being physical experiments, modern experiment setups more and more depend on these additional tools and skills in order to enable reproducible, correct and successful scientific research.

In Müller et al. [6], the authors already introduced *BrainScaleS OS*, the Operating System for BrainScaleS-1. This article describes modifications and enhancements of the *BrainScaleS OS* architecture in the light of the second-generation BSS-2 hardware generation. The following sections introduce the hardware substrate and its envisioned exploitation model in neuroscientific modeling, machine learning and data processing in general. Section II introduces the methods and tools we employ. In section III, we discuss aspects of hybrid —cf. section III-A— operation, hardware component identification and configuration as well as runtime control. Section IV exemplifies the usage of the BrainScaleS Operation System on a simple experiment and describes larger experiments carried out in the past. We close in section V with a discussion of our work and give an overview over future developments.

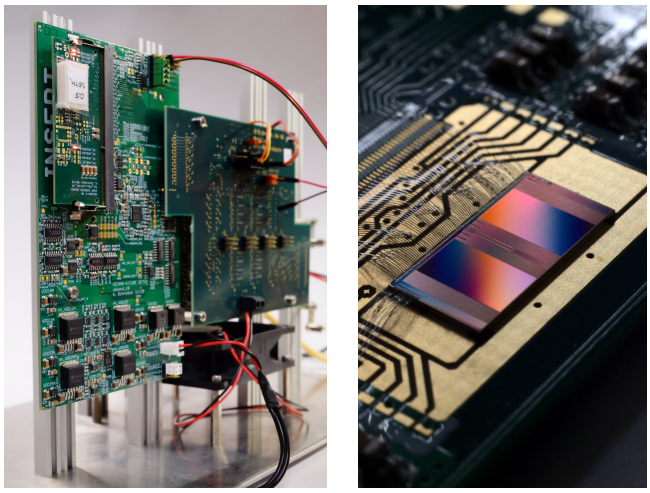

Fig. 1: BrainScaleS-2 single-chip setup. The white plastic cap (top left) covers one accelerated neuromorphic chip (right) which is bonded onto the underlying chip-carrier PCB; other PCBs connect each chip to one FPGA (invisible on the back). The host computer and FPGAs are linked via 1-Gigabit Ethernet. Each BSS-2 chip comprises 512 AdEx neurons and $512 \times 256 = 131\,072$ synapses.

A. BrainScaleS-2 — an Accelerated Mixed-Signal Neuromorphic Substrate

Figure 1 depicts a BSS-2 single-chip lab setup. The main constituent is the neuromorphic mixed-signal chip, manufactured in 65 nm CMOS, carrying 512 AdEx neurons,

$512 \times 256 = 131\,072$ plastic synapses and two embedded SIMD processors capable of fast access to the synapse matrix. A Xilinx Kintex-7 FPGA provides the I/O interface for configuration, stimulus and recorded data. The connection between BSS-2 single-chip setups and a control cluster network is established via 1-Gigabit Ethernet.

The embedded SIMD microprocessor, the Plasticity Processing Unit (PPU), is a Power [7] architecture-based single-core microprocessor with 16 KiB SRAM. It is equipped with a custom vector unit extension, developed in this group and designed to provide digital integer and fixed-point arithmetics which hold up with the parallelism in the analog core and access especially the synapse array in a parallel fashion [8]. In the full-sized chip each PPU features a vector unit with 128 byte vector width, which can operate on 1 or 2 byte entries. Programs are loaded to a PPU via memory writes to the on-chip SRAM and execution is gated via a reset pin. The PPU supports access to off-chip memory regions, e.g., the FPGA's DRAM, for instructions and scalar as well as vector data.

B. Performing Experiments on Neuromorphic Systems

In mueller2020bss1 we introduced the BrainScaleS Operating System for BrainScaleS-1 (BSS-1). It covers aspects of large-scale neuromorphic hardware configuration, experiment runtime control and platform operation. BSS-1 is a wafer-scale neuromorphic system that is available as an experiment platform for external researchers. Compared to single-chip lab systems the system configuration space is large, and aspects of platform operation result in additional requirements for the software stack. In addition, non-expert usability, operational robustness and experiment reproducibility are even more important when offering systems to external users. Previous efforts [9] focused mainly on the neuroscientific community and its view on describing spiking neural networks [10, 11]. However, *BrainScaleS OS* has been providing access to lower-level aspects of the system to expert users [6].

We are still in the early phases of the hardware development roadmap for BSS-2: the first full-sized chip arrived in the labs in 2019 after three small-chip prototypes had been produced and evaluated since 2016. Early experiments were already implemented on the small prototypes [12–15]. We make use of the same prototype version for our example experiment in section IV. However, commissioning of the first full-sized BSS-2 chip is progressing and multi-chip systems are to be expected soon. Therefore, software requirements start to extend into regions already covered by *BrainScaleS OS*. Especially additional features in terms of structured neurons, plateau potentials and the embedded SIMD processors have to be handled by the software stack. Another use case of BSS-2 is its non-spiking mode resembling an analog vector-matrix multiplier that can be used in classical deep neural network experiments.

II. Methods and Tools

We already introduced functional requirements for *BrainScaleS OS* enabling users to perform experiments on the BrainScaleS-1 (BSS-1) neuromorphic hardware platform and additional tasks related to platform operation, cf. Müller et al. [6]. In this work at hand, we focus on software aspects of configuration and control for expert usage. We especially aim to facilitate the process of chip commissioning. In this problem setting users need interfaces to the hardware allowing a transparent and explicit view on configuration as well as runtime control. The implementation of the system configuration layer for BSS-1 already provides some ideas for a structured encapsulation of configuration and runtime control. However, while some aspects of interface are sufficient in terms of software architecture, e.g. the coordinate system and the strongly-typed configuration space, others needed polishing. In particular, the description of experiment control flow was too implicit, relied on many conventions and was hard to extend or modify. Now in BSS-2, the API tracks the experiment control flow explicitly, see sections III-D and III-E.

A. Methodology and Foundations

In Müller et al. [6], we explained the design and implementation methodology: open-sourcing, code review, continuous integration, continuous deployment, explicit tracking of external software dependencies and containerized software development as well as user environments.

Operating custom-built experiment hardware setups poses multiple tasks: secure and fast data exchange, encoding/decoding of hardware configuration as well as result data, and the definition of experiment protocols, i.e. a series of timed events. Taken together, performance and correctness requirements favor the usage of a compiled language. On the other hand, experiment description, input data preprocessing and result data analysis take advantage by interactivity and quicker turn-around cycles.

For the core software stack, we chose C++, a high-performance programming language with strong support for compile-time correctness that evolved in the last years into a multi-paradigm language. One particular popular language for interactive usage and scripted programming is Python. Its use in data science, computational neuroscience as well as machine learning communities enlarges the potential user base.

B. Python APIs

Exposing C++-based programming interfaces to Python can be accomplished in multiple ways. There are at least two libraries providing support for deep integration of Python and C++, *boost::python* and *pybind11* [16]. Both libraries use advanced metaprogramming techniques to simplify the syntax and to reduce the required amount of additional code. Among other things, aspects of type conversion, object lifetime and polymorphism are handled. However, both libraries still need some repetitive coding that could, in principle, be reduced by using a code generator operating on the C++ API's abstract syntax tree. For the BSS-2 software stack, we make extensive use of Clang's C++ AST accessor library, *libclang*, and generate *pybind11*-based wrapper code directly

from C++ header files. This functionality is encapsulated in the tool *genpybind* [17].

III. Implementation

BrainScaleS-2 is a novel compute system: on the one hand, a large fraction of the chip is dedicated to neurons, synapses and model parameter storage; on the other hand, embedded SIMD processors enable conventional computing. In typical neuroscientific experiments, these two parts run closely coupled.

A. Hybrid Operation

In this hybrid operation of the on-chip spiking neural network and the embedded SIMD processors, the latter access observables, modify parameters, perform calculations and change the input of the former to affect its dynamics. In particular, support for flexible learning rules has been one of the main design goals for the system.

The scalar unit of the embedded SIMD processors is based on the Power instruction set architecture [7] allowing to reuse existing open-source software infrastructure such as the C++ language infrastructure by the GNU project. The custom vector extensions have been designed specifically for fast synaptic access. Hence, the authors implemented support for the custom extension to facilitate its use in experiments.

B. Support for the Embedded Processor

Creating executable programs for the embedded processor —the plasticity processor, or PPU— is the main objective of this toolchain. In this case, it comprises a C/C++ compiler, a linker and an assembler for our specific embedded processor. These tools work as a cross-compilation toolchain running on a host computer and generating executables for the PPU. The scalar part of the PPU is already supported by upstream gcc. Extensions for the PPU's custom vector unit are described in section III-B1. In addition to the core C and C++ languages, the respective standard libraries define an extensive set of additional functionality. A subset of this functionality is appropriate for embedded programming [18]. The limited support for the C and C++ standard libraries is presented in section III-B2. Device-specific runtime and hardware-access abstraction, beyond the general-purpose support provided via *libc* and *libstdc++*, is implemented in a dedicated library presented in section III-B3. Post-compilation and runtime supplication of parameters to PPU programs is provided via symbolic access to sections of the program using ELF (*Executable and Linking Format*) information. The supported API is presented in section III-B4. The development of complex programs often necessitates non-trivial debugging techniques. However, embedded systems often lack support for directly interacting with the system. One technique addressing this problem are remote debuggers which allow debugging of problems on a different machine than on which the debugged program is running. In section III-B5 we explain the custom remote debugger implementation for the PPU.

1) Compiler Toolchain: We use the *GNU compiler collection* (gcc) together with the binary utilities package *binutils* to provide this toolchain targeting *C++* as programming language [19]. Since the scalar part of the processor complies with a subset of the embedded Power instruction set architecture 2.06, we can take advantage of the existing gcc backend implementation. We support the custom vector unit by providing the operation code set extension to the assembler. Support for the PPU vector extensions was implemented similar to Power's AltiVec™ [20]. Vector-unit data entities thereby become primary types on the same level as int with synchronization handled by the compiler transparently to the language user. This greatly benefits the conception of plasticity algorithms as it allows, e.g., for functional and object-oriented algorithm design.

2) C/C++ Standard Library Support: The programs written for the PPU are freestanding programs. As a consequence thereof, no system calls are available which otherwise would be provided by an operating system. They are required for the *C* and *C++17* [21] system libraries *libc* and *libstdc++* to work which are typically available in an hosted environment. By supporting a minimal set of required system calls —most notably page acquisition on the heap— a slim *C* library, *newlib* [22], has been integrated. The *libc* then provides the basis for *libstdc++* [23] support. Thereby full standard library support (except file system handling) is available to ease general purpose computation. This library support can be used as a basic set of tools for implementation of re-occuring tasks in abstraction of more complex plasticity problems. For example usage of the STL removes additional development effort of providing custom equivalent implementations.

3) Device-specific support library: In order to facilitate using special features of the processor as well as the hardware, a support library has been implemented. For instance, it provides abstracted access to a wallclock-timer, vector unit access to synapse array, a *C* and *C++* runtime as well as debugging functionality for stack protection facilities. This enables reuse of frequent, at experiment runtime, or typically (e.g., synapse access) needed functionality in programs implementing plasticity rules.

4) ELF-symbol lookup functionality: Complex plasticity kernels typically consume parameters for the initial configuration of the algorithms. These may either be supplied at compile-time, introducing the need to recompile on parameter change, or after compilation via memory access to predefined regions. The latter is supported by providing means to extract ELF symbol positions after compilation to the host software. ELF (*Executable and Linking Format*) [24] is a file format to store binary program data alongside with additional information, e.g., debug symbols or program section information. By extracting section symbol name and location information, sections of the program memory layout can be annotated with symbolic names. The user-facing API allows for *map*-like access, e.g., *program["my_param"] = 24*. This thereby allows simple symbolic access, e.g., to algorithmic parameters or to code sections, after compilation and at runtime.

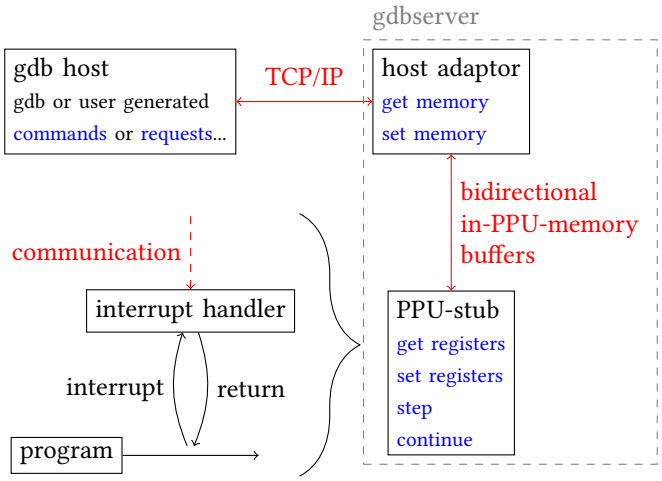

Fig. 2: Schematic showing remote GDB debugger control flow on the embedded processor. A gdb instance running on the host computer communicates with remote debugging protocol via TCP/IP to the gdbserver. Communication from the gdbserver host adaptor to the PPU is established via in memory writes and reads. This allows for transporting register data and control flow information to and from the PPU program under inspection each time the interrupt handler in the PPU program is reached.

5) Debugging Support: An increasing level of complexity in PPU programs in conjunction with the resources limited by programming for a microprocessor with small memory constraints demands for runtime debugging capabilities. Since the toolchain is cross-platform, i.e. development typically happens on a x86-based host computer while the target platform is the embedded Power-based processor, the execution in a debugger on the development platform is not possible. However the *GNU debugger* (gdb) [25], aside from normal debugging on the same machine, also offers support for remote debugging via a TCP connection to a target platform, which also works in a cross-platform setting. The PPU being an embedded processor does neither support TCP natively nor is it feasible to implement a direct client due to memory restrictions.

Figure 2 depicts the implementation of the *gdbserver* targeting the PPU. It is split into a minimal stub in the PPU program which understands base commands such as dumping register content to memory, replacing an instruction through a trap or stepping one instruction and a synchronous program on the host computer which communicates with the PPU through in-memory read and write operations. This adaptor program converts requests from or produces responses to gdb via TCP. This keeps the memory footprint in a PPU program to a minimum while allowing real-time flow control and state inspection.

C. Coordinate System

The multitude of components on a chip leads to a large configuration space of ≈ 350KiB. There are over 100 distinct

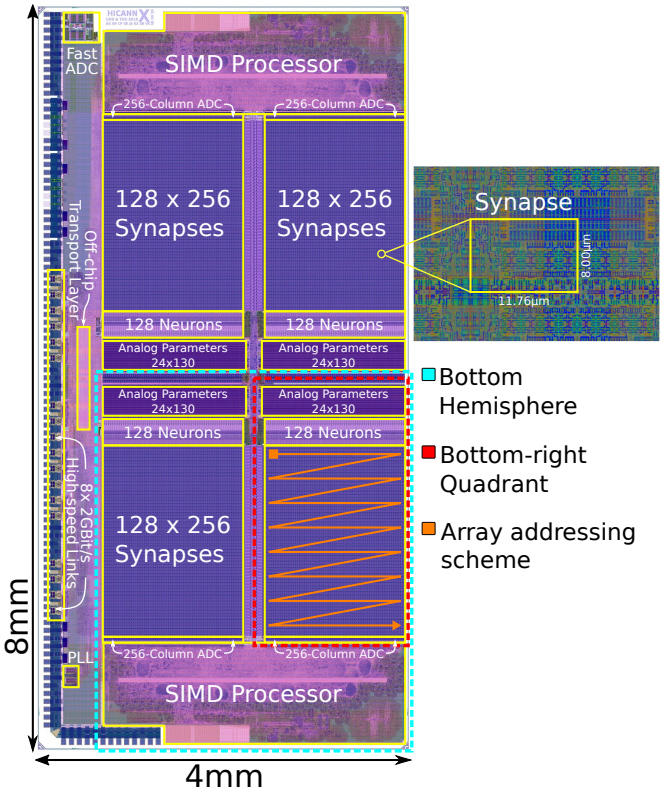

Fig. 3: Layout schematic of the latest BrainScaleS-2 chip. Various component regions are framed in yellow. Framed with dashed lines are logically separable regions of the chip. Synapses and neurons are partitioned into four quadrants, two embedded SIMD processors as well as columnar ADCs are located in the upper and lower chip hemisphere. The ordering scheme of two dimensional coordinates is shown in orange, rows then columns.

ranged integer and 150 boolean registers on hardware which need to be represented in software. To provide type safety as well as other features, e.g., range checking, we do not use the builtin numeric types of *C++* but custom ranged types[26]. High symmetry in chip layout naturally leads to abstraction on different scales.

Figure 3 shows the layout schematic of one chip with annotations for the different component regions. The high symmetry of component distribution is evident, e.g., both chip hemispheres are identically and simply mirrored along the x-axis. Some parts on the hemispheres themselves are again mirrored halves and are therefore called quadrants. This symmetry is reflected in a hierarchical structure of the chip's coordinate system. For BSS-2, the coordinate framework —previously developed for the *BrainScaleS OS* [6]— was extended. To illustrate the idea we use the coordinate defining a Synapse circuit. Depending on the hierarchical level, a synapse can be addressed via *SynapseOnQuadrant*, *SynapseOnHemisphere* or *SynapseOnChip*. This is helpful when implementing functionality which, for example, does not depend on which quadrant it is applied to. Coordinates of a higher level can be cast down to a lower level, e.g.

Listing 1: Example usage of custom coordinate type

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

SynapseOnChip.toSynapseOnQuadrant(). Vice versa, lower views can be combined to create higher-level coordinates, e.g., *SynapseOnChip(SynapseOnHemisphere, HemisphereOnChip)*. It is also possible to convert to different components corresponding to each other. For example, one can convert from a synapse coordinate to a neuron coordinate with *SynapseOnChip.toNeuronOnChip()*. As components are structured differently there is support for linear as well as two dimensional, grid-like, coordinates. Again the synapse is an example for a grid coordinate: it is composed of *SynapseRowOnQuadrant* and *SynapseColumnOnQuadrant*.

Figure 3 also shows the addressing scheme (orange) which adheres to row-major order. Furthermore, the developed ranged types enable coordinates to be used like iterators. This facilitates, for instance, the creation of arrays with typed indexes.

Listing 1 shows an example of how this is used in *C++*. Implementation of this coordinate system can be found at [27].

D. Structuring Data for Configuration

To provide type-safe secure configurability of hardware entities we encapsulate the configuration in so-called containers. A container is an object storing a representation of a possible state of a specific hardware entity or entity group. Application of a represented state to the hardware or retrieval from the hardware is provided in a register-like fashion, the allowed operations are *write* and *read*. Depending on the layer of abstraction, the granularity of access differs. Figure 4 shows the encapsulation at different granularities. The implemented concepts are described in the following.

On the lowest level, access to a register-like memory location on the hardware is abstracted with the user-facing API being a configuration of a variable-length —depending on the coordinate— register word. This encapsulates the state-machine behavior of a heterogeneous set of clients, e.g., SPI [28] or Omnibus [29], the latter being the on-chip bus protocol featuring multi-master operation with guaranteed master-to-client ordering. Thereby, correct usage of the underlying protocol is guaranteed, while the transported word is only restricted to the supported value range.

Building on these register-access containers, smallest-accessible continuous entities are encapsulated in containers of the so-called hardware abstraction layer (*hal*)[30]. For the API user, the composition of sub-word configuration, corresponding to physical entities on a circuit level, are accessible as flat or hierarchical properties. Depending on the property type, type-safe enumerations, ranged integer or boolean value types are used. Representation of sub-word values with a one-to-one correspondence to hardware entities allow for in-code self-documenting parameter names, for

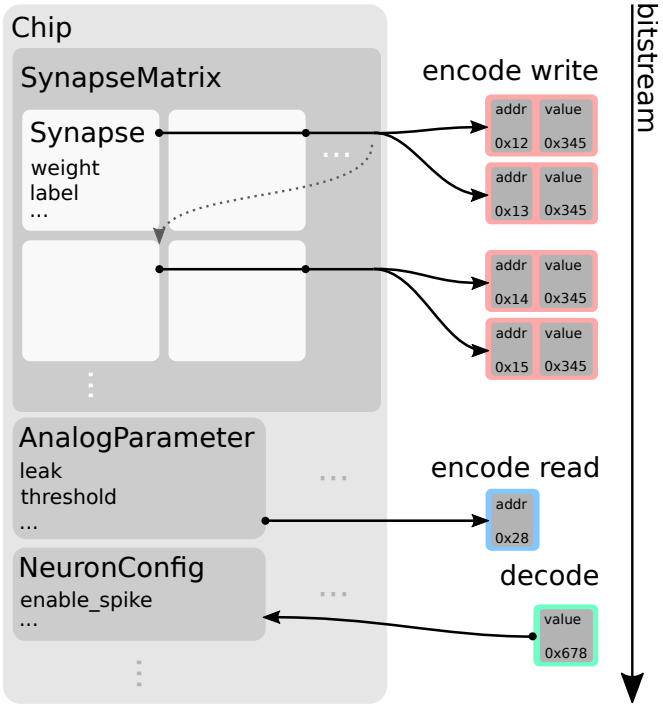

Fig. 4: Configurable hardware entities are modeled by nested data structures encapsulating named data elements. An algorithm visits the nested data structures and generates a hardware configuration bitstream.

example a *NeuronConfig* might have a *enable_leak* boolean property or a *refractory_time* ranged integer value. Named properties inherently state intent opposed to configuring raw unnamed bits of a register word manually. A *hal* container can encapsulate a state spanning over multiple words if the corresponding circuit or configurable entity makes use of distributed configuration bits. On the other side, small containers may be combined, e.g., to form larger heterogeneous or repetitive containers. In addition to type-safe access to sub-word properties, a container implements conversion to pairs of register coordinate and payload types for write operation, called *encoding*, and register coordinates for issuing a read operation together with extraction of container state from read answer register-word payload, called *decoding*. To allow arbitrary grouping, the encoding and decoding for *read* and *write* operation is implemented using a visitor pattern to build a linear sequence of register accesses by recursively visiting sub-containers. There may exist a $1 \rightarrow N$ relation between a *hal* container and multiple register container types, since a specific entity might be accessible via different communication protocols, e.g., JTAG[31] and Omnibus. The protocol is selected upon invocation of a visitor. This allows for a unified interface for the user-facing container API and the conversion to and from register values.

E. Runtime Control

When performing experiments on a neuromorphic system, the timing of input stimulus, output data, access to observ-

ables as well as controllables is essential. Runtime control encapsulates the time-annotated flow of how to actually use the chip. This includes, among other things, bringing the chip into a working state, controlling of the actual spiking neuron network experiment and data transfer in general.

Multiple operational modes are supported by the hardware. First, batch mode is suited for independent experiments. It is characterized by not featuring read-modify-write operation via the host computer. Conversely, the in-the-loop mode makes use of an iterative usage pattern featuring read-modify-write operations from and to an experiment controller. It requires in-experiment synchronization. Finally, a spiking neural network experiment that runs concurrently, time-coupled with an experiment controller is the third operation mode, the real-time closed-loop operation mode, cf. section III-A. The controller might be located on the embedded processors or on another device. It performs read-modify-write operations, e.g., in the form of plasticity updates or environment state variable updates within a sensor-motor loop.

To perform experiments on the neuromorphic chip, experiment descriptions need to include the sequence of timed spike events, the stimulus data. However, the configuration of the chip might also require time-controlled execution for technical or experiment control reasons. For instance, experiments might involve externally-triggered changes to network parameters during runtime. In BSS-2, access to on-chip parameters is possible from multiple locations —or bus masters— the PPUs as well as the FPGA. Hence, experiment control is distributed and the timing between these bus masters needs to be synchronized.

First, we describe the software framework for timed execution that has been developed. Then, we illustrate the control flow of a typical experiment running on BSS-2. The general concept is to construct a temporally ordered stream of commands that is sent to the communication FPGA which than handles timed release of these commands to the chip as well as time-stamped recording of responses from chip. Constructing such a command stream, hereafter called playback program, is facilitated by three functions: *write*, *read*, *wait*. The first two functions allow, as their names suggest, to issue write and read commands of containers, see section III-D, at their respective coordinate locations. Calling the *read* function returns an object which provides access to the read-back data only after the experiment run, inspired by the *std::future* class. Write commands are likewise used to issue spike events. Timed release of commands is facilitated by the *wait* function allowing to delay commands relative to a timer that itself can be modified via write commands. Listing 2 provides a basic usage example.

Figure 5 illustrates the flow of a typical experiment that involves external spike stimulus and concurrent PPU interaction. First, communication with the FPGA is set up and subsequently used for transfer and starting of the compiled playback program. The first stage of each program is the initial configuration of all chips components. This stage has to be timed as analog parameters may require time to settle

Listing 2: Example usage of playback builder pattern

```
PlaybackProgramBuilder builder;

builder.write(NeuronConfigOnDLS(42),
    my_neuron_config);
builder.wait_until(Timer::Value(1000));
auto ticket = builder.read(SpikeCounterOnDLS(3));

auto program = builder.done();
my_executor.run(program);

auto const read_count = ticket.get();
```

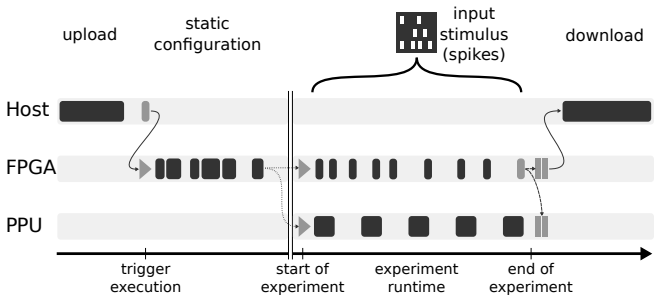

Fig. 5: Schematic showing control flow of a playback program running concurrently with code on the embedded processor.

which makes it necessary for following commands to be delayed. With the chip now being in a working state the experiment is started. For this normally the timer is reset to have a zero-referenced time clock. Likewise, execution of the PPU program is initiated. All read responses as well as spike and ADC data are recorded with time annotation. A special instruction defines the end of the experiment and signals the FPGA to send the recorded data to the host.

F. Preemptive Experiment Scheduling

Going one step further, this framework also allows for a separation of experiment setup, execution and analysis of hardware experiments: instead of executing experiments on locally-attached hardware, the FPGA program is constructed on the client side, transferred to and executed at a shared remote site and its results sent back to the client. Due to the high speed-up factor of the hardware, single experiment runtime typically ranges in the order of milliseconds real time. Experiment assembly and result evaluation — requiring the same order of magnitude in terms of execution time — ordinarily happen in sequence with experiment execution. Relegating both tasks to client side, we eliminate hardware down time. On the remote side both experiment reception and result delivery happen in parallel to experiment execution and hence do not cause down time as well. This allows for the hardware to be shared among several experimenters executing experiments seemingly in parallel on the same chip but more densely packed parameter sweeps for a single experimenter. Plus, the chip remains interactively accessible as one experimenter is able to inject small experiments while a long parameter sweep is underway that would normally block anyone else from using the chip. Overall the measures

increase experiment throughput, thereby effectively speeding-up the hardware even more.

IV. Results

Among the first experiments implemented via the hardware abstraction software framework presented in this work is the Neuromorphic Spike-based Expectation Maximization (NSEM) model. As platform for the experiment the second BrainScaleS-2 prototype [32, 33] is chosen, for which the hardware abstraction software framework presented in this work is fully implemented.

Network Architecture: As seen in fig. 6 (top), the cause layer —comprised of LIF-neurons brought into the stochastic regime by excitatory and inhibitory Poisson input— receives input from an input layer that is modeled via Poisson spike trains. Its aim is to distinguish hidden causes in the presented input stimuli. The cause layer neurons are connected via an inhibitory population with parrot-like behavior: each spike from a cause layer neuron elicits a spike from the inhibitory population, preventing all other cause layer neurons from firing. The cause layer therefore forms a WTA-like structure representing a Boltzmann-machine with very strong inhibitory weights. Therefore, it follows that only one cause layer neuron can ideally respond to each presented input pattern. The weights V_{ik} between input and cause layer evolve according to update rules [34, 35] adapted to the restrictions of the computing substrate. The activity of each cause neuron is kept at a predetermined value via dynamic synapses, implementing a form of spike-based homeostasis heavily inspired by Habenschuss, Puhr, and Maass [36].

Implementation: NSEM employs two plasticity rules (homeostasis as well as learning), acting on different synapses at different time scales. Both are executed on the single PPU of the prototype system at the same time. They are implemented separately and combined using a simple deadline scheduler. In order to facilitate plasticity occurring on different timescales, each plasticity rule has a configurable deadline after which it is applied again.

Results: The center sections in fig. 6 depict the different plasticity rules in action: (center left) while a single neuron receives excitatory input from a background source with varying rate, homeostasis is able to maintain a constant firing rate after (top) sudden shifts in the background rate (middle); the weight evolution of both homeostatic synapses (bottom) reflect the adaptive process. (center right) Several synapses employing NSEM-rule are able to correctly infer the rate of their pre-synaptic neuron despite limited weight resolution. The dashed lines represent the expected target rate while the dotted lines represent the mean of the (colored) infered rates. (bottom) After learning, a network of three neurons is able to differentiate between three input patterns: for every presented pattern (bottom), a different neuron is clearly most active (top). The network is hence able to infer hidden causes of its input in an unsupervised manner, all the while maintaining an activity equilibrium via homeostasis. The successful implementation and execution of the experiment

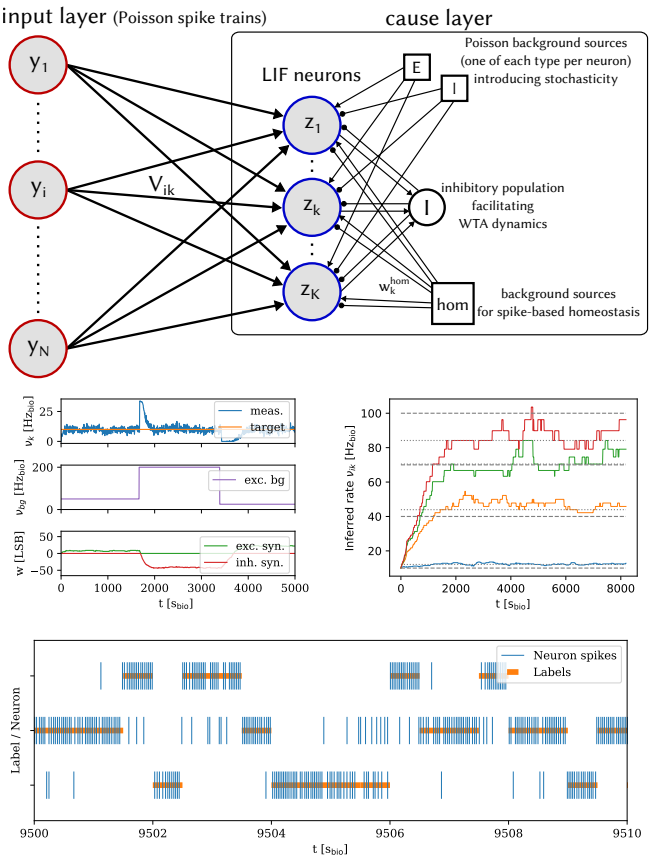

Fig. 6: Example experiment network architecture (top, taken from Breitwieser [37]), homeostasis mechanism (center left), learning of target rates (center right), spike data as well as classification output (bottom). See section IV for more details.

demonstrates the suitability for complex low-level single-chip experiments concurrently employing several distinct plasticity rules running at different time scales on the PPU.

V. Discussion

This work describes the latest developments for *BrainScaleS OS* in the light of BSS-2. In particular, we focus on the expert-level user interfaces for system configuration as well as experiment control. Fundamental ideas of the software design were already devised by the authors in Brüderle et al. [9], the software stack for the precursor of the BrainScaleS-1 architecture [38]. Following that and due to being a large-scale neuromorphic platform [5, 39], the software stack was rewritten for BrainScaleS-1 [6]. Eventually, the second-generation BrainScaleS architecture demands for increased flexibility due to its improved configurability and programmability [8, 40]. In particular, the embedded SIMD processors require additional software during both, experiment design and experiment runtime.

BrainScaleS OS for BSS-2 builds upon libraries as well as development methodologies that have been developed for BSS-1. In the current version, single-chip BSS-2 systems can be robustly configured and controlled during runtime.

The software layers presented in this work focus on expert usage. Higher-level experiment description languages such as PyNN [10] are not yet supported.

We demonstrate the configuration and experiment control in a hybrid experiment setting also involving two plasticity rules, homeostasis and learning. Multiple other experiments, e.g. [12–15], demonstrate the system's applicability in machine learning and computational neuroscience.

The group also focuses on increasing reliability and quality assurance in mixed-signal hardware development [41]. In particular, co-simulating software and hardware enables a continuous-integration-based workflow during hardware development.

VI. Future Developments

BrainScaleS OS for BSS-2 is still under development. The presented software layers have been sufficient for expert experimenters. However, we identified several aspects that need attention in the future.

Structured Data Exchange in Distributed Systems

Communication is a key element of distributed systems. Inter-operation of host computers, FPGAs and PPUs typically requires communication to exchange state as well as to provide synchronization. In particular, data is exchanged between different architectures with varying endianness and alignment constraints. For example, exchanging plasticity parameters between host and PPU already benefits from cross-platform structured data exchange. To solve these tasks, we aim for a thin message passing library that is integrated with a platform-agnostic serialization library.

Full Stack Hardware Design Validation

Verification of hardware design prior to manufacturing is vital as chip production is expensive. Past experiences have shown that unit testing of individual chip components alone is often insufficient. By providing a simulator backend in a lower-level communication layer, the full software stack can be used to run tests/experiments on a simulated hardware device. This will facilitate the validation of new FPGA features and —most importantly— chip designs by utilizing the complete test suite of the software stack prior to fabrication.

Algorithmic Task Offloading

Increasing complexity in plasticity and first steps towards standalone on-chip calibration algorithms demand for access of arbitrary on-chip facilities. In section III-D and section III-C, we introduced an API providing this kind of access. Therefore, the lower-level parts of *BrainScaleS OS* —in particular the hardware abstraction layer— are to be ported to the PPU architecture. The availability of the vector unit for on-chip code enables optimized access to synapses and similar facilities which are unavailable for host or, more precisely, FPGA-based access. At the same time, size and runtime performance overhead needs to be minimized.

Logical Experiment Description

BrainScaleS architectures already support a variable number —scalable to cortical connection densities— of presynaptic connections per "logical" neuron representing multiple linked neuron circuits. In addition, the current version of the BSS-2 architecture makes use of a similar mechanism to build structured neurons consisting of multiple compartments; see Aamir et al. [42] for a detailed description of the hardware implementation. However, while the hardware abstraction layer provides support for type-safe and correct configuration of the system, it does not abstract the constitution of user-friendly encapsulation of configuration entities itself. The set of parameters bound to such a "logical" configuration entity could be included in a collection of hardware abstraction layer containers possibly with constraints on their placement via coordinates. For the encoding and decoding of these larger, logical entities, composition of hardware abstraction layer containers is to be used. The rules under which to group parameters to "logical" configuration entities are currently under development.

Higher-level User Interfaces

For higher-level usage, e.g., as accelerator for spiking neural network models, a topology-centric graph-based configuration API on top of the hardware abstraction established in this document is planned; Similarly, multi-chip systems increase the need for automation of tasks related to transforming a user-defined experiment to a valid hardware configuration, hardware calibration and distributed experiment control. Inspiration is taken from the existing higher-level software infrastructure for the BSS-1 system Müller et al. [6].

Recent developments in the machine learning community affect the way people think about data flow as well as how to programmatically describe learning algorithms [43, 44]; on the other hand, the neuromorphic community starts building a bridge between deep neural networks and spiking neuromorphic substrates [45–47]. As a first step, the BSS-2 non-spiking operation mode allows for a transparent integration into typical libraries for classical neural networks libraries. Furthermore, the exploitation of the same neural network libraries allows for the specification of, e.g., plasticity rules in a computational graph; full integration of BSS-2 into a neural network library would be a large step towards a high-level specification of, e.g., plasticity rules.

VII. Contributions

E. Müller is the lead developer and architect of the BrainScaleS-2 software stack. C. Mauch contributed to the software architecture. P. Spilger contributed to the final software architecture and is the main contributor of the described experiment. O. Breitwieser contributed the preemptive experiment scheduling capabilities, designed the initial experiment and contributed to the implementation on hardware. J. Klähn and D. Stöckel contributed to the initial software architecture. T. Wunderlich contributed to

the remote debugger implementation. J. Schemmel is the lead designer and architect of the BrainScaleS-2 neuromorphic system. All authors discussed and contributed to the manuscript.

Acknowledgments

The authors wish to thank all present and former members of the Electronic Vision(s) research group contributing to the BrainScaleS-2 hardware platform, software development and operation methodologies, as well as software development. The authors express their special gratitude towards: • Arthur Heimbrecht for his initial work on adding vector-unit support to the compiler; • Simon Friedmann for the implementation of the initial commissioning software. We especially express our gratefulness to the late Karlheinz Meier who initiated and led the project for most if its time.

This work has received funding from the EU ([H2020/2014-2020]) under grant agreements 720270 (HBP) and 785907 (HBP).

References

[1] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, et al. "A million spiking-neuron integrated circuit with a scalable communication network and interface". In: *Science* 345.6197 (2014), pp. 668–673.

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

[3] Thomas Pfeil, Andreas Grübl, Sebastian Jeltsch, et al. "Six networks on a universal neuromorphic computing substrate". In: *Frontiers in Neuroscience* 7 (2013), p. 11. ISSN: 1662-453X. DOI: 10.3389/fnins.2013.00011. URL: http://www.frontiersin.org/neuromorphic%5C_engineering/10.3389/fnins.2013.00011/abstract.

[4] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, et al. "Loihi: A neuromorphic manycore processor with on-chip learning". In: *IEEE Micro* 38.1 (2018), pp. 82–99.

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

[6] Eric Müller, Sebastian Schmitt, Christian Mauch, et al. "The Operating System of the Neuromorphic BrainScaleS-1 System". In: *arXiv preprint* (Mar. 2020). URL: TODO.

[7] PowerISA. *PowerISA Version 2.06 Revision B*. Tech. rep. Available at http://www.power.org/resources/reading/. power.org, July 2010.

[8] S. Friedmann, J. Schemmel, A. Grübl, et al. "Demonstrating Hybrid Learning in a Flexible Neuromorphic Hardware System". In: *IEEE Transactions on Biomedical Circuits and Systems* 11.1 (2017), pp. 128–142. ISSN: 1932-4545. DOI: 10.1109/TBCAS.2016.2579164.

[9] Daniel Brüderle, Eric Müller, Andrew Davison, et al. "Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system". In: *Frontiers in Neuroinformatics* 3 (2009), p. 17. ISSN: 1662-5196. DOI: 10.3389/neuro.11.017.2009. URL: https://www.frontiersin.org/article/10.3389/neuro.11.017.2009.

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

[11] Jochen M. Eppler, Moritz Helias, Eilif Muller, et al. "PyNEST: a convenient interface to the NEST simulator". In: *Front. Neuroinform.* 2.12 (2008).

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

[13] Sebastian Billaudelle, Yannik Stradmann, Korbinian Schreiber, et al. "Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate". In: *arXiv preprint arXiv:1912.12980* (2019).

[14] Benjamin Cramer, David Stöckel, Markus Kreft, et al. "Control of criticality and computation in spiking neuromorphic networks with plasticity". In: (2019). arXiv: 1909.08418 [cs.ET].

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

[16] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2019.

[17] Johann Klähn. *genpybind software v0.2.0*. 2020. DOI: 10.5281/zenodo.372674. URL: https://github.com/kljohann/genpybind.

[18] Michael Barr. *Programming embedded systems in C and C++*. eng. 1st ed. Sebastopol, Calif.: O'Reilly, 1999. ISBN: 978-1-56592-354-6.

[19] B. Gough and R.M. Stallman. *An Introduction to GCC: For the GNU Compilers Gcc and G++*. Network theory manual. Network Theory, 2005. ISBN: 9780954161798. URL: https://books.google.de/books?id=yIGKQAAACAAJ.

[20] J. Tyler, J. Lent, A. Mather, et al. "AltiVec/sup TM/: bringing vector technology to the PowerPC/sup TM/ processor family". In: *1999 IEEE International Performance, Computing and Communications Conference (Cat. No.99CH36305)*. Feb. 1999, pp. 437–444. DOI: 10.1109/PCCC.1999.749469.

[21] *Programming languages — C++*. Standard. Geneva, Swiss: International Organization for Standardization, Dec. 2017.

[22] Bill Gatliff. "Embedding with gnu: Newlib". In: *Embedded Systems Programming* 15.1 (2002), pp. 12–17.

[23] Free Software Foundation. *The GNU C++ Library*. URL: https://gcc.gnu.org/onlinedocs/libstdc++.

[24] Tool Interface Standard Commitee. *Executable and Linking Format (ELF) Specification*. 1995. URL: http://refspecs.linuxbase.org/elf/elf.pdf.

[25] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB*. Free Software Foundation, 2011. ISBN: 978-0-9831592-3-0.

[28] Susan C. Hill, Joseph Jelemensky, Mark R. Heene, et al. "Queued serial peripheral interface for use in a data processing system". US4958277A. 1987.

[29] Simon Friedmann. *Omnibus On-Chip Bus*. forked from https://github.com/five-elephants/omnibus. 2015. URL: https://github.com/electronicvisions/omnibus.

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

[32] S. Friedmann, J. Schemmel, A. Grübl, et al. "Demonstrating Hybrid Learning in a Flexible Neuromorphic Hardware System". In: *IEEE Transactions on Biomedical Circuits and Systems* 11.1 (2017), pp. 128–142. ISSN: 1932-4545. DOI: 10.1109/TBCAS.2016.2579164.

[33] S. A. Aamir, Y. Stradmann, P. Müller, et al. "An Accelerated LIF Neuronal Network Array for a Large-Scale Mixed-Signal Neuromorphic Architecture". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.12 (Dec. 2018), pp. 4299–4312. ISSN: 1549-8328. DOI: 10.1109/TCSI.2018.2840718.

[34] Bernhard Nessler, Michael Pfeiffer, Lars Buesing, et al. "Bayesian Computation Emerges in Generic Cortical Microcircuits through Spike-Timing-Dependent Plasticity". In: *PLoS Computational Biology* 9.4 (2013). Ed. by Olaf Sporns, e1003037.

[35] Johannes Bill, Lars Buesing, Stefan Habenschuss, et al. "Distributed Bayesian Computation and Self-Organized Learning in Sheets of Spiking Neurons with Local Lateral Inhibition". In: *PLOS ONE* 10.8 (Aug. 2015), pp. 1–51. DOI: 10.1371/journal.pone.0134356. URL: https://doi.org/10.1371/journal.pone.0134356.

[36] Stefan Habenschuss, Helmut Puhr, and Wolfgang Maass. "Emergence of Optimal Decoding of Population Codes Through STDP". In: *Neural Computation* 25.6 (2013), pp. 1371–1407.

[37] Oliver Breitwieser. "Towards a Neuromorphic Implementation of Spike-Based Expectation Maximization". Master thesis. Ruprecht-Karls-Universität Heidelberg, 2015.

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

[39] Sebastian Millner, Andreas Grübl, Karlheinz Meier, et al. "A VLSI Implementation of the Adaptive Exponential Integrate-and-Fire Neuron Model". In: *Advances in Neural Information Processing Systems 23*. Ed. by J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, et al. 2010, pp. 1642–1650.

[40] Syed Ahmed Aamir, Paul Müller, Andreas Hartel, et al. "A highly tunable 65-nm CMOS LIF neuron for a large-scale neuromorphic system". In: *Proceedings of IEEE European Solid-State Circuits Conference (ESSCIRC)*. 2016.

[41] Andreas Grübl, Sebastian Billaudelle, Benjamin Cramer, et al. "Verification and Design Methods for the BrainScaleS Neuromorphic Hardware System". In: *arXiv preprint* (2020). URL: http://arxiv.org/abs/2003.11455.

[42] S. A. Aamir, P. Müller, G. Kiene, et al. "A Mixed-Signal Structured AdEx Neuron for Accelerated Neuromorphic Cores". In: *IEEE Transactions on Biomedical Circuits and Systems* 12.5 (Oct. 2018), pp. 1027–1037. ISSN: 1932-4545. DOI: 10.1109/TBCAS.2018.2848203.

[43] Adam Paszke, Sam Gross, Francisco Massa, et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[44] Martín Abadi, Ashish Agarwal, Paul Barham, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2015. URL: http://download.tensorflow.org/paper/whitepaper2015.pdf.

[45] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, et al. "Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification". In: *Frontiers in Neuroscience* 11 (2017), p. 682. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00682. URL: https://www.frontiersin.org/article/10.3389/fnins.2017.00682.

[46] B. Rueckauer and S. Liu. "Conversion of analog to spiking neural networks using sparse temporal coding". In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. May 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351295.

[47] Julian Göltz, Andreas Baumbach, Sebastian Billaudelle, et al. *Fast and deep neuromorphic learning with time-to-first-spike coding*. 2019. eprint: arXiv:1912.11443.

Own Software

[26] Sebastian Jeltsch. *rant*. URL: https://github.com/ignatz/rant.

[27] Electronic Visions(s), Heidelberg University. *halco*. URL: https://github.com/electronicvisions/halco.

[30] Electronic Visions(s), Heidelberg University. *haldls*. URL: https://github.com/electronicvisions/haldls.