Implementing Closed-Loop Interaction with the Real World for the BrainScaleS-2 Software Stack

Lukas Jahnel

Studienarbeit
Ruprecht-Karls-Universität Heidelberg
Fakultät für Ingenieurwissenschaften
ZITI
Electronic Vision(s) - Dr. J. Schemmel

February 6, 2026

Contents

1 Introduction

The pursuit of embodied intelligence, where cognitive systems are grounded in the sensorimotor loops of a physical body, stands as a central pillar of neuromorphic research (Bartolozzi et al. 2022; Sandamirskaya et al. 2022). To investigate systems that not only process information but actively perceive and act within the physical world, computational substrates must transcend the isolation of offline simulation / emulation (Bartolozzi et al. 2022; Stewart et al. 2015). Real-time interaction with external environments is therefore a prerequisite, particularly for experiments involving stimulus-driven processing, dynamic perturbations, or closed-loop robotic control (Milde et al. 2022; Indiveri and Liu 2015; Stewart et al. 2015; Sandamirskaya et al. 2022). Unlike conventional simulation / emulation , which are decoupled from physical time, accelerated neuromorphic systems like BrainScaleS-2 (BSS-2) can process sensory streams with microsecond latency (Pehle et al. 2022; Schemmel et al. 2022). The BSS-2 platform supports this via SpikeIO, a dedicated FPGA-based interface designed to connect external setups with the accelerated analog neuron array, providing the physical connectivity necessary for stable, low-latency control loops (Stradmann and Schemmel 2024). However, leveraging this hardware capability requires seamless integration into the system's software stack. Despite the hardware's readiness, unified support for real-time event ingress has been absent from the *grenade* (Graph-based Experiment Notation And Data-flow Execution) execution layer, the framework responsible for mapping network descriptions to hardware-compatible graph representations (Müller, Arnold, et al. 2022) as well as all user-facing frontends, such as *pynn* (Davison et al. 2009), *hxtorch* or *jaxsnn* (Spilger et al. 2023; Müller, Althaus, et al. 2024). Prior to this work, the software stack fully supported event handling, allowing on-chip neurons to communicate via the routing network. However, the software representation of external stimuli was limited to pre-calculated playback sequences or on-chip background generators. Real-time streams via SpikeIO were not represented in the software abstraction, meaning they could not be targeted by standard routing algorithms, spike label generation, or synchronized with the execution flow. Consequently, researchers requiring real-time input were forced to bypass the higher level software stack entirely, manually configuring the FPGA and routing tables. While functional, this reliance on low-level hardware access effectively restricted real-time capabilities to expert users. It prevented standard users from exploiting the usability of the high-level software stack and bypassed critical compiler safety checks, increasing the risk of inconsistent hardware states. Furthermore, it broke compatibility with the standard modeling API, preventing the portable description of closed-loop networks. This work addresses this limitation by introducing a comprehensive integration of SpikeIO ingress into the *grenade* network, signal-flow building, and execution layers. We establish real-time external injection as a first-class mechanism within the graph-based execution model, spanning from the network definition to the low-level bitstream generation. The integration introduces dedicated `SpikeIOSourcePopulation` types, incorporates external neurons into the automated routing and labeling infrastructure, and augments the signal-flow graph with explicit ingress vertices. Additionally, the real-time executor has been extended to handle non-deterministic real-time events alongside deterministic playback, allowing arbitrary combinations of recorded and live stimuli. This extension restores the safety guarantees of the software stack while preserving the low-latency characteristics of the hardware. Note that this work focuses exclusively on the ingress (input) pathway; real-time event extraction (egress) remains a subject for future architectural extensions.

2 System Representation

2.1 BrainScaleS-2 Platform Overview

BrainScaleS-2 (BSS-2) is a mixed-signal neuromorphic computing platform designed for the emulation of spiking neural networks using continuous-time analog circuits (Pehle et al. 2022). Neuronal and synaptic dynamics are implemented as physical processes, such that membrane integration, synaptic interaction, and spike generation evolve continuously in time rather than being discretized by a numerical simulation scheme. The system operates at an accelerated time scale, emulating neural dynamics approximately three orders of magnitude faster than biological real time (Pehle et al. 2022; Müller, Arnold, et al. 2022). This acceleration enables efficient exploration of long biological timescales while preserving temporal precision, as spike timing is determined by the underlying circuit dynamics and not by a global simulation clock.

At the hardware level, the platform is centered around a custom neuromorphic application-specific integrated circuit (ASIC) that combines analog neuron and synapse circuits with a digital infrastructure for configuration and event-based communication. Synaptic weights and neuron parameters are stored locally on the chip, while spikes are represented digitally and propagated through an on-chip routing fabric using spike labels to address synaptic targets (Müller, Arnold, et al. 2022). This architecture supports flexible network topologies within the constraints imposed by the physical layout of the ASIC. The neuromorphic ASIC is embedded into a larger system architecture that includes an FPGA-based digital periphery and a host-controlled software stack (Schemmel et al. 2022). The FPGA is responsible for low-level chip configuration, experiment control, and coordination of data transfer between the accelerated neuromorphic core and the host system. As a result, time-critical neural dynamics and spike routing are handled entirely on-chip, while experiment orchestration and configuration synthesis are performed off-chip. This separation between analog computation and digital control is a defining design principle of the BrainScaleS-2 platform. It allows the system to combine the efficiency and parallelism of analog computation with the flexibility of programmable digital control, while maintaining a clear boundary between real-time neural execution and higher-level experiment management.

Several interfaces exist to connect the neuromorphic core to external data sources and sinks through the FPGA layer. Among these is SpikeIO, an event-based interface that enables the injection of externally generated spike events into the on-chip routing fabric and the extraction of spike events from the system (Stradmann and Schemmel 2024). The role of SpikeIO within the overall software stack is discussed in detail in later sections. The hardware and system architecture described above form the basis for the software abstractions used in this work. In particular, the structure of the routing fabric, the hierarchical control flow, and the separation of execution and configuration directly constrain how experiments must be represented and compiled in the software stack.

2.1.1 Problem Statement: SpikeIO Configuration Space

The BrainScaleS-2 software stack represents experiments using a graph-based notation that is compiled into a concrete hardware execution state. Within this model, the *grenade* layer is responsible for expressing all experiment components in a unified representation and for ensuring that the resulting graph encodes a complete and executable hardware configuration. From this perspective, integrating SpikeIO into the standard compilation and execution flow requires that all hardware-relevant degrees of freedom associated with SpikeIO are represented explicitly within the experiment graph and validated prior to execution. This is non-trivial for two reasons. First, SpikeIO is a shared hardware resource whose operating mode must be configured uniformly per `ExecutionInstance`. Second, SpikeIO provides a real-time event interface, such that injected spike events do not exist at graph-construction time but must nevertheless be integrated into the static routing and labeling infrastructure.

The scope of this work is not the implementation of the SpikeIO hardware or its low-level control logic. Instead, the objective is to make existing SpikeIO functionality accessible through the user-facing experiment description APIs. This requires that SpikeIO-related configuration is captured within the same graph-based experiment representation that is used for all other components and synthesized as part of the standard compilation process. Before describing the corresponding software representation, we therefore explicitly define the SpikeIO configuration space that must be captured by the experiment graph.

Global SpikeIO operating mode The SpikeIO FPGA block must be configured via the setting of bits of the 32-bit `SpikeIOConfig` into one consistent operating mode for the entire `ExecutionInstance` (Stradmann and Schemmel 2024). In this work, the parameters are:

- `data_rate_scaler`: hardware timing parameter governing the SpikeIO link behavior (bits 00-15).
- `enable_tx`: enable egress of ASIC internal spike events off the FPGA (bit 16).
- `enable_rx`: enable ingress of externally generated spike events over FPGA to ASIC (bit 17).
- `enable_internal_loopback`: activate on-FPGA loopback of Spike routing for end-to-end verification (bit 18).

Ingress routing and spike-label binding (per logical channel) To inject events that are semantically equivalent to on-chip spikes, each external SpikeIO channel must be bound to the same spike-label address space used by the on-chip routing fabric and synapse configuration (Pehle et al. 2022; Müller, Arnold, et al. 2022). The required routing state is:

- **Input routing**: Externally injected events must be integrated into the on-chip event routing fabric using the same spike-label space as on-chip spikes. At the FPGA level, this is realized by an input routing table with one entry per SpikeIO channel, where each entry specifies the target spike label. The compilation flow must therefore synthesize a complete input routing table for all active channels and reject configurations with missing or ambiguous label bindings.
- **Output routing**: For validation and loopback experiments, selected internal events may be transmitted via the FPGA. This is realized by a complementary output routing table with one entry per channel, mapping spike labels to serial output addresses. While this functionality is supported at the hardware level, it is not yet exposed through the

grenade experiment representation and is therefore not used as part of the compilation flow in this work.

2.2 User Interface (*pynn*)

While the SpikeIO functionality is represented internally within the *grenade* framework, it must also be exposed at the level of the user-facing experiment description. In the BrainScaleS-2 software stack, this role is fulfilled by *pynn*, which provides a simulator-independent network description language and serves as the primary interface for configuring experiments. To integrate SpikeIO into this frontend while adhering to *pynn*'s cell-type taxonomy, the `OffChipSource` celltype is introduced. This abstraction represents populations of neurons whose spike activity originates off-chip and is injected into the experiment in real time, without exposing hardware-specific configuration details to the user.

To inject external stimuli, users instantiate the source similarly to standard neuron populations. The following listing demonstrates the minimal configuration required to instantiate an external source and connect it to an on-chip population:

Listing 2.1: Minimal *pynn* configuration for external event injection via spikeio.

```
import pynn.brainscales2 as pynn

# Instantiate external source population
ext_input = pynn.Population(1,
    pynn.cells.OffChipSource(...),
)

# Connect to on-chip neurons
on_chip_pop = pynn.Population(10, pynn.cells.HXNeuron())
pynn.Projection(ext_input, on_chip_pop, pynn.AllToAllConnector())
```

From the perspective of a *pynn* script, an `OffChipSource` behaves like a conventional population and can be instantiated, connected, and recorded using standard *pynn* constructs. During graph construction, an `OffChipSource` population is translated into a corresponding `SpikeIOSourcePopulation` within the *grenade* network graph. This translation preserves the structural semantics defined at the *pynn* level, including population size, per-channel spike-label assignments, and the recording capabilities, while deferring hardware-specific routing and configuration to the backend representation. A corresponding sink abstraction for real-time spike egress is not part of the *pynn* frontend in this work.

2.3 Graph Representation (*grenade*)

To represent the FPGA-based external event interface within the *grenade* framework, SpikeIO is exposed at the network level through the `SpikeIOSourcePopulation` abstraction. This allows the SpikeIO interface to be instantiated and referenced within the user-defined experiment graph using the same population-based structure as other sources, while deferring hardware-specific configuration details to later compilation stages. Structurally, the `SpikeIOSourcePopulation` follows the conventions used throughout *grenade*. It is defined by a configuration object and a set of logical neurons, and can be connected to other populations via standard projection mechanisms. Unlike other virtual sources whose activity is

either model-driven or precomputed, the `SpikeIOSourcePopulation` has no intrinsic temporal definition. Each logical neuron instead serves as a placeholder for externally generated spike events that occur in real time and are injected via the FPGA interface.

Although the semantics of a `SpikeIOSourcePopulation` differ from those of neural populations mapped onto the ASIC, it represents a population of neurons whose activity originates off-chip. Conceptually, such populations correspond to external neuronal sources (e.g., sensory neurons in a retina or other peripheral processing stages) that project onto on-chip neurons via spike-based communication. It is defined by a population-global `SpikeIOConfig` structure and a set of logical neurons. The configuration maps directly to the hardware-level `SpikeIOConfig` and exposes the previously defined control flags: `enable_rx`, `enable_tx`, `enable_internal_loopback`, and `data_rate_scaler`. Together, these parameters define the operating mode of the uniformly shared SpikeIO FPGA block. The population size is determined by a vector of logical Neuron objects. While these neurons do not model analog membrane or synaptic dynamics on the ASIC, they serve as logical endpoints for spike-based connections and can be targeted by projections in the same manner as on-chip populations. In addition, they expose standard attributes such as `enablerecordspikes`, allowing external spike sources to be integrated into the existing recording infrastructure, effectively treating off-chip input channels as recordable spike sources within the experiment graph.

At the signal-flow level, which represents the execution state used for hardware synthesis, the population is mapped to a dedicated vertex of type `SpikeIOSourcePopulation`. This vertex extends the network-level description with the routing information required for SpikeIO configuration. In particular, it stores the input and output routing tables that define the mapping between FPGA serial channels and the on-chip spike-label address space. A second vertex, `SpikeIOIngress`, represents the logical ingress interface between the FPGA SpikeIO block and the on-chip event fabric. In contrast to `SpikeIOSourcePopulation`, which acts as a configuration and routing container, `SpikeIOIngress` provides the signal-flow–level abstraction of external spike injection by exposing a timed spike output compatible with the chip-level event stream.

2.4 Routing and Hardware Synthesis

The core complexity of integrating SpikeIO lies in mapping the logical populations defined in the graph to the physical hardware constraints of the BSS-2 routing fabric. This process is handled by the compiler backend in a multi-stage pipeline, spanning from configuration verification to the final generation of hardware registers. The *pynn* and *grenade* part of this execution pipeline, illustrating the transformation from high-level *pynn* definitions to low-level hardware execution, is shown in Figure 2.1.

2.4.1 SpikeIO Configuration Synthesis

The integration of SpikeIO ingress into the pipeline first requires that the hardware-level operating mode of the FPGA SpikeIO block be derived unambiguously from the populations defined at the network level. This synthesis process is executed during graph construction and constitutes the primary step in transforming a high-level specification involving `SpikeIOSourcePopulation` objects into hardware-executable configuration data.

The process initiates by scanning the populations assigned to a given `ExecutionInstance`. If no `SpikeIOSourcePopulation` is present, the builder concludes that the instance does not rely on real-time external input, and no SpikeIO configuration is generated. Conversely, if

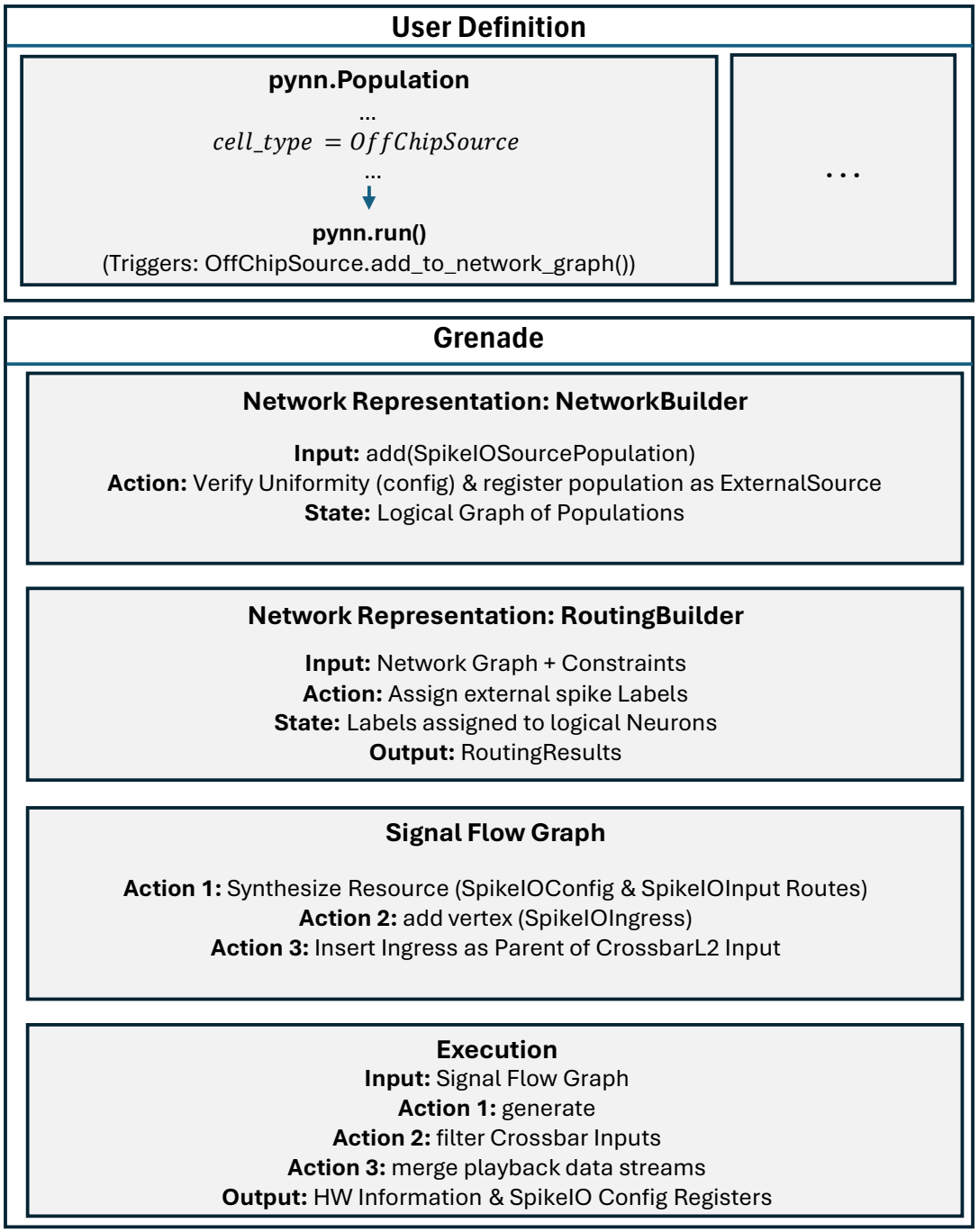


Figure 2.1: The multi-stage process of transforming high-level user definitions into hardware-executable configurations. The pipeline begins with the Network Representation within the *grenade* middleware, where logical graphs are verified and hardware labels are assigned. Subsequently, a Signal Flow Graph is formed to integrate the `SpikeIOIngress` vertex. The final process of the middleware involves the Graph Building and Data Flow synthesis, which prepares the experiment's structure before passing the data to lower-level software packages for the final generation of hardware registers and playback streams.

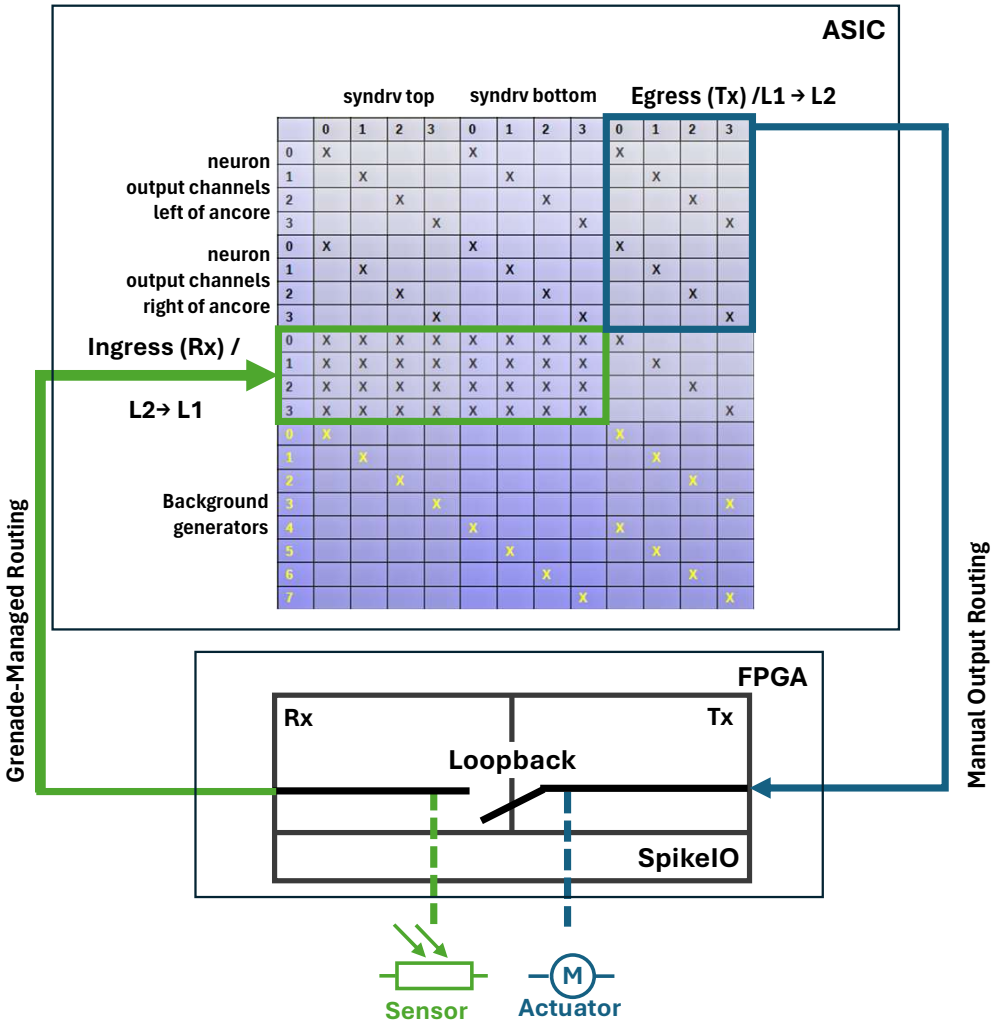


Figure 2.2: Schematic of the ASIC-FPGA system highlighting the bidirectional event transport. The egress path (blue) depicts the routing of neuron output channels through the internal L1 crossbar system to the external L2 link, directed towards the FPGA SpikeIO block for interaction with real-world actuators. The loopback functionality is displayed as a switch, illustrating the internal loopback capabilities of the FPGA SpikeIO block. The ingress path (green) shows the transmission of external stimuli through the Rx-unit and onto the ASIC, enabling real-time closed-loop coupling between the neuromorphic core and the environment.

one or more such populations are detected, the builder extracts the population-level configuration parameters, transmit enable, receive enable, and internal loopback and data rate scaler from the first `SpikeIOSourcePopulation` encountered. These values establish the reference configuration for the entire `ExecutionInstance`.

Since the SpikeIO interface is implemented as a single, shared hardware block, it must operate in a globally consistent mode. To enforce this, the builder verifies that all subsequent `SpikeIOSourcePopulation` objects associated with the same instance specify identical configuration parameters. Any deviation triggers a validation error during graph construction, pre-

venting conflicting operating modes from propagating into the hardware-level configuration. Once uniformity is confirmed, the builder synthesizes the final `SpikeIOConfig`. This object represents the exact binary configuration that will be programmed into the FPGA SpikeIO block. It encodes the three operational flags alongside the user-defined `data_rate_scaler`, which governs the low-level timing and serialization speed of the ASIC–FPGA link. By serializing this configuration into the `ExecutionInstance` resources, the system guarantees that downstream phases, including hardware initialization and real-time event handling, operate on a deterministic specification.

2.4.2 Label Assignment and Input Route Construction

After synthesizing the global configuration, the system must establish the connection between the external SpikeIO interface and the on-chip event fabric. This involves two steps: allocating hardware-compatible spike labels and generating the routing tables that map FPGA serial channels to these labels.

grenade manages external spikes using the same labeling infrastructure that governs on-chip communication. Each logical neuron in a `SpikeIOSourcePopulation` is treated as an external source and assigned a unique 16-bit hardware spike label via the standard labeling pipeline. This 16-bit identifier serves a dual purpose: its lower six bits correspond to the synaptic label stored in the synapse array of postsynaptic neurons, while the full label acts as the address on the routing bus.

Once labels are assigned, the builder translates them into concrete hardware routes. The builder iterates over the 255 supported serial channels (`SpikeIOInputRouteOnFPGA`). For each channel, it inspects the routing results to identify any associated spike labels. If one or more labels are assigned to a channel, the builder constructs a corresponding `SpikeIOInputRoute` object, encapsulating the primary label to be injected, and binds it to the appropriate FPGA lane identifier. If a lane has no assigned labels, a default route is created, ensuring the lane remains silent. This procedure yields a complete mapping for all channels, regardless of their active status. The logic is summarized in Listing 2.2.

The final set of routes is stored within the execution-instance resources. By deriving these routes entirely from the preexisting labeling infrastructure, the integration maintains architectural coherence: SpikeIO ingress is treated conceptually as another external spike source, governed by the same routing semantics as conventional off-chip inputs.

2.4.3 Signal-Flow Graph Integration

Following configuration and route synthesis, the system proceeds to the graph generation phase. The objective of this method is to algorithmically insert the real-time ingress point into the signal-flow graph without breaking the existing connectivity of playback and on-chip sources.

The integration logic begins by querying the `ExecutionInstance` for the presence of valid `SpikeIOSourcePopulation` descriptors. Upon detection, the builder instantiates a `SpikeIOIngress` vertex. Unlike standard population translation, which iterates over neurons to generate compartment-specific vertices, this step instantiates a single EntityOnChip at the chip-level coordinate. This vertex is constructed to act as a pure event generator, devoid of internal state or synaptic memory, and serves as the graph-level proxy for the FPGA interface.

Listing 2.2: Algorithmic derivation of SpikeIO routing tables and label-channel mapping.

```
# N_channels: Total SpikeIO serial channels (indexed 0..255)
# Map: channel_id -> spike_label (Ingress)
# Map: spike_label -> channel_id (Egress)

# 1. Initialize all channels as SILENT
input_routes = {id: SILENT for id in range(N_channels)}
output_routes = {id: SILENT for id in range(N_channels)}

# 2. Ingress (FPGA -> ASIC)
# Bind logical label to physical input channel
for label in routing_results.assigned_labels:
    chan_id = get_channel_id(label)
    input_routes[chan_id] = label

# 3. Egress (ASIC -> FPGA, optional)
# Bind physical label to output channel
for label in routing_results.selected_outputs:
    chan_id = get_channel_id(label)
    output_routes[label] = chan_id

# 4. Serialize to hardware resource
store_resource(input_routes, output_routes)
```

The core procedural step involves the modification of the graph topology to route these external events to the neuromorphic core. The builder locates the `CrossbarL2Input` vertex, which acts as the convergence point for all incoming spike traffic. The integration logic then executes a topology update: it inserts the newly created `SpikeIOIngress` vertex into the adjacency list of the `CrossbarL2Input` as an upstream parent. This operation is conditional: if the crossbar vertex does not yet exist (e.g., in an experiment with no playback), the builder explicitly creates it before establishing the link. If the crossbar is already populated with playback sources, the `SpikeIOIngress` is simply appended to the existing set of parents. This graph manipulation ensures that the `SpikeIOIngress` is topologically indistinguishable from other spike sources. By injecting the vertex directly into the input hierarchy of the crossbar, the builder forces the downstream graph traversals, used by the hardware configuration generator and the executor, to automatically include real-time events in their resource allocation and event-flow calculations.

2.5 Real-Time Execution Semantics

The integration of SpikeIO ingress necessitates extensions to the execution-stage semantics, as real-time external events differ fundamentally from precomputed spike sequences in terms of temporal availability and delivery mode. The real-time executor must distinguish between sources that provide timestamped spike data at build time and those, such as `SpikeIOIngress`, that inject events only during live execution. This distinction is essential for maintaining determinism in playback streams while enabling dynamic, externally driven activity.

In the standard execution model, all spike activity is funneled through the `CrossbarL2Input` vertex. For buffered sources providing precomputed sequences, such as playback programs or background generators, the executor retrieves timestamped data from internal buffers and

merges them into a single, temporally ordered sequence. However, with the introduction of SpikeIO, the `CrossbarL2Input` may also receive inputs from `SpikeIOIngress` vertices, which act as live sources with no build-time data.

During execution preparation, the executor iterates over the parents of the `CrossbarL2Input` to populate the hardware playback buffers. Crucially, if the builder encounters a `SpikeIOIngress` vertex, it explicitly excludes it from the buffer population step, as these events will be injected via the FPGA interface rather than the playback memory. If the sole upstream source is a `SpikeIOIngress`, the executor determines that no build-time spike sequences exist for the instance. Consequently, it refrains from populating its event buffers, delegating spike injection entirely to the real-time SpikeIO interface. If the crossbar has multiple sources, the executor selectively collects sequences only from those providing build-time data. Inputs originating from `SpikeIOIngress` are explicitly ignored during this merge step. After collecting valid sequences, the executor concatenates and sorts them, producing a deterministic spike stream that is delivered alongside, but independently of, the real-time events supplied by SpikeIO. This separation of responsibilities ensures that deterministic inputs retain their reproducible temporal structure, while real-time events are injected without perturbing the semantics of the build-time data.

3 Methods

3.1 Experimental Methodology

To evaluate the functional correctness and timing fidelity of the SpikeIO ingress integration, we conducted a controlled loopback experiment. In this setup, externally injected spike events traversed the complete software and hardware pathway of the BrainScaleS-2 system, allowing for the isolation of interface characteristics independent of neural dynamics.

3.1.1 Loopback Configuration

The functional validation employed a dual-source loopback configuration to decouple the measurement of latency from system load. As shown in the experimental setup in Listing 3.1, the stimulus is split into two distinct populations: a load population and a probe population. The load population is configured to fire at varying frequencies, providing a controlled background event rate to stress the SpikeIO link. Simultaneously, the probe population fires at a constant, lower frequency on a separate channel.

By isolating the probe events on a dedicated SpikeIO channel, we can perform precise timestamping and pairing of events (t_{gen} and t_{rx}) without the interference or potential collision artifacts associated with high-bandwidth streams. This differential measurement ensures that the calculated latencies (Δt) represent the hardware processing time of the ingress pathway under specific load conditions rather than artifacts of high-frequency event matching.

The logical and physical routing configuration for this verification experiment is depicted in Figure 2.2. Spikes emitted by the on-chip relay neurons were routed off-chip to the FPGA and delivered back into the ASIC via the SpikeIO ingress interface. Within the ASIC, these events entered the standard event-routing fabric, where they would normally be forwarded into the synapse array. However, for this experiment, the events were intercepted within the ASIC and looped back directly to the FPGA using the dedicated in-ASIC event loopback path. This configuration enabled the precise timestamping of spike events after their delivery into the ASIC and subsequent re-export to the FPGA, allowing a direct comparison between the originally generated on-chip events (t_{gen}) and their looped-back counterparts (t_{rx}).

The corresponding *pynn* experiment definition is shown in Listing 3.1 and 3.2. Note that while the ingress path is handled automatically by the *grenade* stack, the egress path for this specific validation was configured via a helper function to bridge the current lack of automated egress routing.

Listing 3.1: Minimal PyNN configuration for the end-to-end loopback verification experiment.

```python
def main():
    # Initialize the software stack
    pynn.setup()

    # 1. Instantiate stochastic on-chip spike source
    # This generates the stimulus that drives the loopback
    on_chip_spike_source = pynn.Population(1,
        pynn.cells.SpikeSourcePoissonOnChip(rate=1e3, seed=53))
    on_chip_spike_source.record(["spikes"])

    # 2. Instantiate the external interface via OffChipSource
    # Configured in internal loopback mode for timing validation
    spikeio_ingress = pynn.Population(1,
        pynn.cells.OffChipSource(
            enable_internal_loopback=True,
            data_rate_scaler=1,
            label =[0]
        )
    )
    spikeio_ingress.record(["spikes"])

    # 3. Configure egress routing (Manual helper for validation)
    # This routes the on-chip source to the FPGA for loopback
    setup_spikeio_output(
        output_populations=[on_chip_spike_source],
    )

    # Execute experiment for 100ms
    pynn.run(100)
    pynn.end()

if __name__ == "__main__":
    main()
```

Listing 3.2: Minimal PyNN configuration for the end-to-end loopback including latency measurement adaptation.

```
def main():
    # Initialize the software stack
    pynn.setup()

    # 1. Instantiate stochastic on-chip spike sources for load and probe
    # This generates the stimulus that drives the loopback
    load_src = pynn.Population(
        1, pynn.cells.SpikeSourcePoissonOnChip(rate=load_rate_hz, seed=53)
    )

    probe_src = pynn.Population(
        1, pynn.cells.SpikeSourcePoissonOnChip(rate=probe_rate_hz, seed=54)
    )
    probe_src.record(["spikes"])

    # 2. Instantiate the external interface via OffChipSource
    # Configured in internal loopback mode for timing validation
    off_chip_spikeio_population = pynn.Population(
        2, OffChipSource(enable_internal_loopback=True, data_rate_scaler=1,
                         label=[0,2])
    )
    off_chip_spikeio_population.record("spikes")


    # 3. Configure egress routing (Manual helper for validation)
    # This routes the on-chip source to the FPGA for loopback
    setup_spikeio_output(
        output_populations=[load_src, probe_src],
    )

    # Execute experiment for 100ms
    pynn.run(100)
    pynn.end()

if __name__ == "__main__":
    main()
```

4 Results

4.1 Latency and Timing Analysis

The timing fidelity of the ingress pathway was characterized by analyzing the temporal difference $\Delta t = t_{\text{rx}} - t_{\text{gen}}$ for matched spike pairs in the controlled loopback configuration described in Section 3. As shown in Figure 4.1a, the distribution of latencies exhibits a sharp onset at a minimum of approximately 1.2 µs. The profile is left-skewed, with the vast majority of events accumulating below the 2.0 µs mark.

To characterize the temporal behavior over time, we analyzed the latency drift over the experiment duration of 1000 ms (Figure 4.1b). A linear regression of the data yields a slope of $0.04 \pm 0.09\,\text{ns}\,\text{ms}^{-1}$.

Figure 4.1c summarizes the latency distributions across a sweep of input event rates. As the load increases, the width of the distribution expands. Finally, the throughput saturation was characterized by comparing the measured output rate against the ideal linear relationship (Figure 4.1d). The measurement follows the linear prediction up to an input rate of approximately 1.5×10^3 kHz, after which the output rate plateaus.

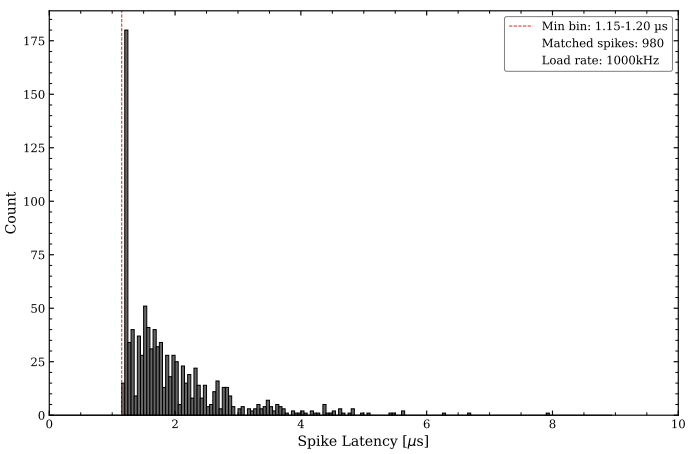
Figure 4.1c summarizes the latency distributions across a sweep of total input event rates (load plus probe). As the load increases, the width of the distribution expands. Finally, the throughput saturation was characterized by comparing the measured output rate against the measured input rate (Figure 4.1d). The measurement follows the linear prediction up to an input rate of approximately 1.5 MHz, after which it plateaus.

4.2 Routing and Configuration Validation

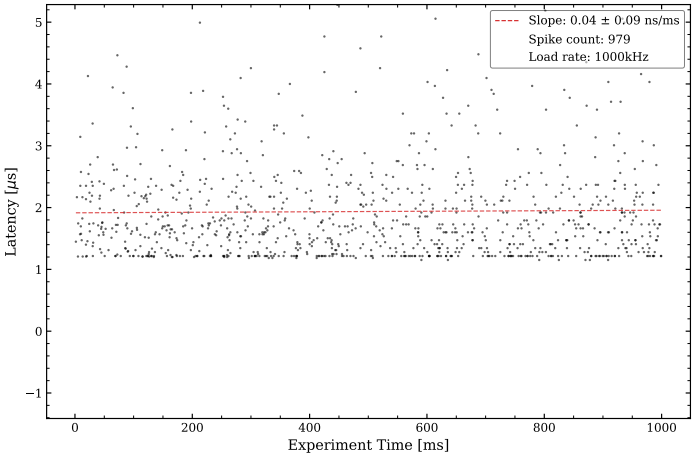
Beyond timing, the experiment validated the functional correctness of the graph synthesis pipeline described in Chapter 2.

During graph construction, *grenade* successfully assigned unique spike labels to the `OffChipSource` population, integrating it into the global `externalspikelabels` map. The subsequent hardware configuration synthesis correctly translated the `enableinternalloopback` parameter into the corresponding bitfields of the `SpikeIOConfig` register. Furthermore, the input route construction generated a deterministic mapping for the active loopback lanes, corresponding to the ingress pathway depicted in Figure 2.2. The specific serial addresses corresponding to the `OffChipSource` were bound to the assigned spike labels, while unused lanes were correctly initialized with silent routes.

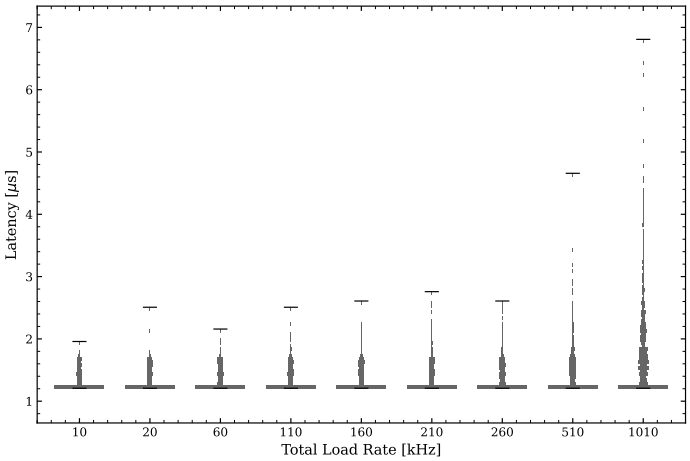
These results confirm that the software stack correctly translates high-level *pynn* definitions into the low-level hardware routing tables required for precise event injection. It should be noted that while the transmission of on-chip events back to the FPGA was utilized here for stimulus generation, the ingress logic validated by this experiment is agnostic to the signal source. The validated reception pipeline operates identically whether stimuli are generated by the loopback mechanism, an external microcontroller, or a physical sensory device.



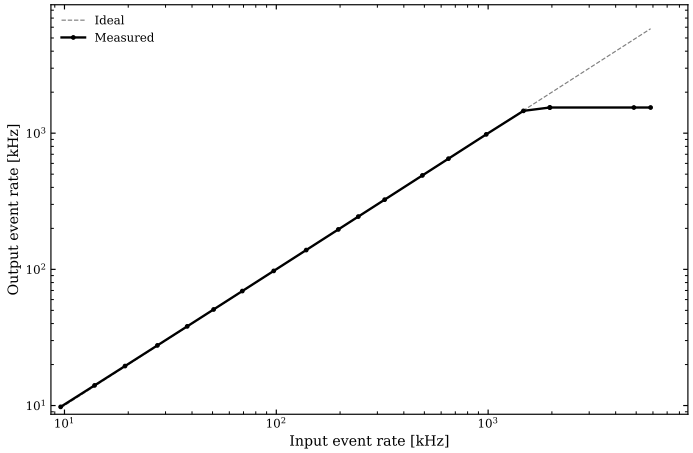
(a) Latency histogram



(b) Latency drift over time



(c) Latency distribution



(d) Throughput saturation

Figure 4.1: Characterization of the SpikeIO ingress pathway regarding latency and throughput. (a) **Latency distribution under load.** Histogram of end-to-end spike latencies measured with a load of 1000 kHz and a probe rate of 1 kHz. The distribution represents matched spike pairs over a 1000 ms runtime, with a bin width of 0.05 µs. (b) **Temporal stability.** Individual spike latencies plotted against experiment time of 1000 ms. A linear regression (red dashed line) indicates a negligible drift of 0.04 ns ms^{-1} and error of ±0.09 ns ms^{-1} for a 1000 kHz load rate and 1 kHz probe rate. (c) **Load sweep.** Violin plots illustrating the latency distributions across total load rates (load rate & probe rate) ranging from 10 kHz to 1010 kHz. (d) **Throughput bandwidth.** Log-log plot of the measured output event rate versus the input event rate. The system maintains an ideal linear response (dashed line) until bandwidth saturation occurs at approximately 1.5 MHz.

5 Discussion

5.1 Temporal Fidelity and Closed-Loop Suitability

The experimental results demonstrate that the integrated SpikeIO ingress pathway achieves minimum round-trip latencies in the 1.15 µs to 1.20 µs bin, with the vast majority of spike pair latencies clustered below the 2 µs mark. In the context of neuromorphic control systems, where sensory-motor loops often operate on millisecond timescales (Stradmann and Schemmel 2024), this sub-2 µs performance effectively classifies the interface as "hard real-time". The distribution profile (Figure 4.1a) is tightly clustered and left-skewed, which is characteristic of a system dominated by fixed processing overheads rather than stochastic jitter. Furthermore, the temporal drift analysis yields a negligible slope of $0.04\,\mathrm{ns\,ms^{-1}}$ ($< 1\,\mathrm{\mu s\,s^{-1}}$) with a error of $\pm 0.09\,\mathrm{ns\,ms^{-1}}$. This stability indicates that the ingress pathway is free from state accumulation or buffer congestion; in systems with throughput mismatches, latency typically rises significantly over time as input queues fill. It should be noted that the reported drift of $0.04\,\mathrm{ns\,ms^{-1}}$ was obtained under the same experimental conditions as the latency characterization, which was optimized short experiment runtimes rather than for isolating long-term timing stability. Under these conditions, residual trends can arise from correlated load-induced effects and finite sampling statistics, despite the absence of any accumulating internal state. To assess whether this observed slope reflects a systematic timing drift of the ingress pathway, we conducted additional measurements tailored specifically to drift characterization. In this configuration, the external load was removed while maintaining the same probe rate, and the experiment duration was extended. Under these conditions, the measured drift was $0.00 \pm 0.03\,\mathrm{ns\,ms^{-1}}$, indicating that the previously observed slope lies within the statistical resolution of the latency measurement and does not represent a systematic temporal instability of the SpikeIO ingress. This confirms that the interface maintains stable timing behavior over extended durations and is suitable for long-running closed-loop experiments.

Regarding bandwidth, the saturation point observed at 1.5 MHz (Figure 4.1d) defines the maximum sustainable event rate for this specific configuration. This limit aligns with the serialization constraints of the FPGA-ASIC link shown by (Stradmann and Schemmel 2024), confirming that the software integration utilizes the full available hardware bandwidth without introducing artificial bottlenecks. Consequently, the integration allows the BrainScaleS-2 system to interact with external physical setups, such as robotic actuators or sensory streams, without introducing variable processing delays that would destabilize fast control loops.

5.2 Validation correctness

Beyond timing, the results validate the functional correctness of the graph synthesis pipeline. The deterministic mapping observed in the loopback experiment confirms that the software stack correctly translates high-level *pynn* definitions into low-level hardware routing tables. Specifically, the generated input routes bound the logical `OffChipSource` addresses to the correct physical serial lanes, while unused lanes were correctly initialized as silent. This

proves that the *grenade* compilation flow successfully abstracts the complexity of the hardware routing fabric while preserving the strict connectivity required for precise event injection.

5.3 System Limitations and Scope

While this work establishes a robust, fully integrated ingress pathway, support for real-time egress (event extraction) remains experimental. As noted in the experimental setup, the egress pathway used for loopback verification relied on manual route configuration, bypassing the *grenade* graph compiler. This limitation does not arise from a hardware restriction: the BrainScaleS-2 platform is fully capable of streaming spike events off-chip in real time. Instead, the current limitation is rooted in the graph abstraction and routing semantics of the software stack. The *grenade* network model exclusively represents connectivity in terms of synaptic projections, whereas real-time egress would require a population-level event export that is not mediated by synapses. Such population-to-population connections are presently not expressible within the implicit signal-flow graph. In addition, no dedicated population-level spike recording primitive currently exists that could serve as a drop-in abstraction for streaming output. Consequently, the present work deliberately focuses on solving the ingress challenge, enabling dynamic, real-time control, while deferring the integration of non-synaptic, streaming egress mechanisms to future development.

A further constraint identified during integration concerns the singular nature of the SpikeIO hardware block. As a shared FPGA resource, it cannot support conflicting configurations, such as simultaneous internal loopback and external data injection, within a single `ExecutionInstance`. Although the software abstraction enforces this limitation to prevent invalid hardware states, it inherently limits the system to one specific interaction mode per experiment run. This design choice prioritizes operational safety and reproducibility over configurational flexibility, embedding hardware constraints directly into the abstraction layer.

6 Conclusion

This work presents the first complete integration of real-time external event ingress into the *grenade* software framework for the BrainScaleS-2 neuromorphic system. By introducing the `SpikeIOSourcePopulation` and its corresponding *pynn* frontend `OffChipSource`, we have closed the gap between the platform's hardware capabilities and its user-facing software stack. The integration fundamentally transforms how the system handles external inputs: rather than relying on ad-hoc, low-level hardware access, real-time events are now first-class citizens in the signal-flow graph, participating in the standard routing, labeling, and configuration pipelines.

Functional validation confirmed that the automated compilation pipeline correctly synthesizes hardware routes from high-level network definitions, ensuring operational safety by preventing invalid hardware states. The timing analysis revealed a deterministic low-latency pathway with minimum response times around 1.2 µs and the majority of events serviced within 2 µs, classifying the interface as suitable for real-time control loops. By abstracting the complexity of FPGA configuration and graph manipulation, this integration empowers researchers to define complex, interactive experiments using standard neural network descriptions, significantly lowering the barrier to entry for closed-loop neuromorphic computing.

7 Outlook

While this work establishes a fully integrated and functionally validated SpikeIO ingress pathway, several extensions are planned to complete the real-time interaction loop and further broaden the applicability of the software stack.

A primary next step is the integration of a real-time egress path to enable the extraction of spike events during execution. Closing the loop for fully bidirectional interaction is a fundamental requirement for neuromorphic benchmarking and robotics interfacing (Milde et al. 2022; Stewart et al. 2015; Stradmann and Schemmel 2024). Consequently, this functionality will be incorporated as soon as the underlying *grenade* infrastructure provides stable support for streaming population data.

Beyond interface completion, future work will focus on demonstrating the software in closed-loop experiments involving physical hardware. Moving beyond functional loopback validation toward application-driven evaluation in robotics and control scenarios will validate the real-world usability of the SpikeIO interface. Such experiments will involve coupling external sensors and actuators to the neuromorphic system, directly addressing the growing demand for hard real-time interaction in neuromorphic control (Stradmann and Schemmel 2024).

Finally, the software will be extended to support additional front-end frameworks, specifically *jaxsnn* and *hxtorch*. These frameworks leverage modern automatic differentiation to define and train spiking neural networks. Integrating real-time ingress into these high-level APIs will open the door to online learning workflows and hardware-in-the-loop training, enabling a wider range of users to leverage the BrainScaleS-2 platform for contemporary machine learning research (Spilger et al. 2023).

Glossary

`Config` A nested configuration structure within a population that holds hardware-specific parameters. 22

`CrossbarL2Input` The convergence point in the signal-flow graph where spike events from playback buffers and real-time sources are merged before entering the synapse array. 11, 12, 22

`ExecutionInstance` The container representing a single hardware run, holding the synthesized resources (such as the SpikeIO configuration and routing tables) for that specific execution. 5, 7, 9, 10, 19, 22

`OffChipSource` The high-level PyNN frontend class introduced in this work that allows users to define external spike sources within a network experiment. 6, 16, 18, 20, 22

`RealTimeExecutor` The runtime engine responsible for loading buffers and managing the experiment execution flow on the host system. 22

`SpikeIOConfig` The hardware configuration structure (register map) that defines the operating mode (Rx/Tx enable, data rate, loopback) of the FPGA block. 5, 7, 10, 16, 22

`SpikeIOIngress` A specific signal-flow graph vertex introduced to represent the real-time event injection point in the hardware graph topology. 7, 8, 10–12, 22

`SpikeIOSourcePopulation` The backend population representation in *grenade* that encapsulates the configuration and routing data for external event sources. 3, 6, 7, 9, 10, 20, 22

`SpikeSourcePoissonOnChip` An on-chip stochastic spike generator used for stimulus generation and system validation. 22

`TimedSpikeSource` A graph vertex representing pre-calculated (playback) spike sequences, utilized for deterministic stimulus generation. 22

`enable_record_spikes` Attribute allowing external spike sources to be integrated into the recording infrastructure. 7

`external_spike_labels` A mapping structure within the routing result that assigns unique 14-bit hardware identifiers to logical external neurons. 16

PyNN A Python package for simulator-independent specification of neuronal network models, used as the primary user frontend for BrainScaleS-2. 2, 3, 6, 7, 13, 16, 18, 20

grenade Graph-based Experiment Notation And Data-flow Execution. The middleware library responsible for mapping logical networks to hardware graph representations and execution states. 2, 3, 5–8, 10, 13, 16, 19–22

hxtorch A PyTorch-based frontend for the BrainScaleS-2 system, enabling hardware-in-the-loop gradient-based training. 3, 21, 22

jaxsnn A JAX-based framework for training spiking neural networks, targeted for future integration with the real-time ingress. 3, 21, 22

`data_rate_scaler` Hardware timing parameter governing the SpikeIO link behavior. 5, 7

`enable_internal_loopback` Activate on-FPGA loopback for end-to-end verification. 5, 7, 16

`enable_rx` Enable ingress of externally generated spike events. 5, 7

`enable_tx` Enable egress of spike events from the FPGA. 5, 7

Bibliography

Bartolozzi, Chiara, Giacomo Indiveri, and Elisa Donati (2022). "Embodied neuromorphic intelligence". In: *Nature Communications* 13.1, p. 1024. DOI: 10.1038/s41467-022-28487-2.

Davison, Andrew P. et al. (2009). "PyNN: a common interface for neuronal network simulators". In: *Frontiers in Neuroinformatics* 2.11. DOI: 10.3389/neuro.11.011.2008. URL: https://doi.org/10.3389/neuro.11.011.2008.

Indiveri, Giacomo and Shih-Chii Liu (2015). "Memory and Information Processing in Neuromorphic Systems". In: *Proceedings of the IEEE* 103.8, pp. 1379–1397. DOI: 10.1109/JPROC.2015.2444094.

Milde, Moritz B. et al. (2022). "Neuromorphic Engineering Needs Closed-Loop Benchmarks". In: *Frontiers in Neuroscience* Volume 16 - 2022. ISSN: 1662-453X. DOI: 10.3389/fnins.2022.813555. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2022.813555.

Müller, Eric, Moritz Althaus, et al. (2024). "jaxsnn: Event-driven Gradient Estimation for Analog Neuromorphic Hardware". In: *2024 Neuro Inspired Computational Elements Conference (NICE)*, pp. 1–6. DOI: 10.1109/NICE61972.2024.10548709.

Müller, Eric, Elias Arnold, et al. (2022). "A Scalable Approach to Modeling on Accelerated Neuromorphic Hardware". In: *Frontiers in Neuroscience* Volume 16 - 2022. ISSN: 1662-453X. DOI: 10.3389/fnins.2022.884128. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2022.884128.

Pehle, Christian et al. (2022). "The BrainScaleS-2 Accelerated Neuromorphic System with Hybrid Plasticity". In: *Front. Neurosci.* 16. ISSN: 1662-453X. DOI: 10.3389/fnins.2022.795876. arXiv: 2201.11063 [cs.NE]. URL: https://www.frontiersin.org/articles/10.3389/fnins.2022.795876.

Sandamirskaya, Yulia et al. (2022). "Neuromorphic computing hardware and neural architectures for robotics". In: *Science Robotics* 7.67, eabl8419. DOI: 10.1126/scirobotics.abl8419. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.abl8419. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abl8419.

Schemmel, Johannes et al. (2022). "Accelerated Analog Neuromorphic Computing". In: *Analog Circuits for Machine Learning, Current/Voltage/Temperature Sensors, and High-speed Communication: Advances in Analog Circuit Design 2021*. Ed. by Pieter Harpe, Kofi A.A. Makinwa, and Andrea Baschirotto. Cham: Springer International Publishing, pp. 83–102. ISBN: 978-3-030-91741-8. DOI: 10.1007/978-3-030-91741-8_6. URL: https://doi.org/10.1007/978-3-030-91741-8_6.

Spilger, Philipp et al. (2023). "hxtorch.snn: Machine-learning-inspired Spiking Neural Network Modeling on BrainScaleS-2". In: *Proceedings of the 2023 Annual Neuro-Inspired Computational Elements Conference*. NICE '23. San Antonio, TX, USA: Association for Computing Machinery, pp. 57–62. ISBN: 9781450399470. DOI: 10.1145/3584954.3584993. URL: https://doi.org/10.1145/3584954.3584993.

Stewart, Terrence C. et al. (2015). "Closed-Loop Neuromorphic Benchmarks". In: *Frontiers in Neuroscience* Volume 9 - 2015. ISSN: 1662-453X. DOI: 10.3389/fnins.2015.00464. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2015.00464.

Stradmann, Yannik and Johannes Schemmel (2024). “Closing the loop: High-speed robotics with accelerated neuromorphic hardware”. In: *Front. Neurosci.* 18. ISSN: 1662-453X. DOI: 10.3389/fnins.2024.1360122.

Funding acknowledgement

The work carried out in this Studienarbeit used systems, which received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements Nos. 720270, 785907 and 945539 (Human Brain Project, HBP) and Horizon Europe grant agreement No. 101147319 (EBRAINS 2.0)

Reproducibility statement

The results of this work can be reproduced using the repositories in the following table.

Table 7.1: Repository versions used for the SpikeIO integration

Repository	Git hash	Change set
grenade	6af5ef866a928c60749b1efdcc04944537a52c63	25407
pynn-brainscales	3c3c29a524dc3f1d42b4f4bf6423cb10c263e7c6	25406
code-format	512f1fafb884a7da30a1943793a3cae6441462be	
logger	73dadb3ce413c521845ef7d36f818073eee4fefa	
halco	e6d3ca2e6648f6041e5f11cf37a7fe66f5629261	
haldls	e89d5cb139f29d73a7a48c7ad8cd404914bea0d5	
hate	b8c8fd03b6fdc103022848a47bc6838d46b3d3ae	
libnux	a69d086ba506ac72b60ed10f210be8e613dc2dd4	
fisch	ca07fa1ce8ff37e16bce5b52c845225fd94c960c	
hxcomm	76e15d0a7d957e750238c85016109ec3a760cdb4	
calix	89fb43c85ffd08029bb69dec4ac985f7f6ddd30e	
rant	53199ee94cae1e1c2e4db10e88d570a761b14e0f	
ztl	c2d4faee05f497010ee55e35bf9c9607eecbf884	
pywrap	5e2af30e9593882b471d3cd02df00b93f13ff479	
lib-boost-patches	136c5b41cb046afe2c726aa4646928bf5190622e	
sctrltp	7a94faf9af3d38b202276d11de44182e3602b8cd	
hwdb	8877d48a6276f1eade6e8bcabe6c8df2fbce031a	
visions-slurm	b32eaa77766fded68524a51980a119da975a9ac1	
flange	522fdd23830d76a072691946e656e6843c43d00d	
lib-rcf	469f4b592207902064c1523b3f2df4b501552eba	
bss-hw-params	74c6403c4de9a0f85e2f6b3380daf49f553f4e7a	
dapr	b14f8cdde6800d4a10fa2e613a5c7f3e47d328b3	

Table 7.2: Apptainer container used for all experiments.

Key	Value
path	/einc/containers/stable/2026-01-15_1.img
fingerprint	36c4b259-c0ca-4a34-9da0-758ee4ae8a62
app	dls

Declaration

I hereby declare that I have written this Studienarbeit independently and that I have not used any sources or aids other than those indicated.

Statement on the Use of AI/GenAI

During the preparation of this Studienarbeit, generative / artificial intelligence–based tools were used in a supportive manner, as listed below.

- Generative AI was used for proofreading, spellchecking, and improving the readability of the written report.
- Generative AI was used for initial code scaffolding for the plots presented in this report.

Heidelberg, February 6, 2026