Department of Physics and Astronomy

Heidelberg University

Bachelor thesis

in Physics

submitted by

Tim Auberer

born in Ravensburg

2024

Runtime-dynamic reconfiguration

for a time-continuous neuromorphic system

This Bachelor thesis has been carried out by

Tim Auberer

at the

Kirchhoff-Institute for Physics

under the supervision of

Dr. Johannes Schemmel

Runtime-dynamic reconfiguration for a time-continuous neuromorphic system:

Public interest and demand for neural networks is growing worldwide. To meet the need for greater computational resources to train larger and more precise machine learning models, different kinds of specialized hardware are being developed. One approach that takes its cue from nature's proven success is neuromorphic computing. The BrainScaleS-2 neuromorphic platform is designed to provide a flexible and accelerated substrate for emulating spiking neural networks: from neuroscientific experiments, exploring local plasticity and dynamics in networks with structured neurons, to machine learning applications focusing on event-driven computation. To describe these experiments, an intuitive way of thinking about a continuous experiment flow with discrete time points of changes in the configuration is desired. In this work, the necessary software changes are implemented, to enable this way of describing an experiment for users of the BrainScaleS-2 platform. To realize this feature of dynamically reconfiguring an experiment during runtime, conceptual changes were required on several layers of the BrainScaleS-2 software stack. This includes a new kind of program builder and a new way of assembling the experiment from its different individual parts. Finally, the changes made in this work, are evaluated with a performance test and proven useful at use case examples.

Laufzeitdynamische Rekonfiguration für ein zeitkontinuierliches neuromorphes System:

Das öffentliche Interesse und die Nachfrage nach neuronalen Netzen wächst weltweit. Um den Bedarf an größeren Rechenressourcen zum Trainieren größerer und präziserer maschineller Lernmodelle zu decken, werden verschiedene Arten spezialisierter Hardware entwickelt. Ein Ansatz, der sich an der erfolggekrönten Natur orientiert, ist das neuromorphe Rechnen. Die neuromorphe BrainScaleS-2-Plattform wurde entwickelt, um ein flexibles und beschleunigtes Substrat für die Emulation von spikenden neuronalen Netzen zu bieten: Von neurowissenschaftlichen Experimenten zur Erforschung der lokalen Plastizität und Dynamik in Netzwerken mit strukturierten Neuronen bis hin zu Anwendungen des maschinellen Lernens, die sich auf ereignisgesteuerte Berechnungen konzentrieren. Um diese Experimente zu beschreiben, ist eine intuitive Art gewünscht, die mit der Vorstellung von einem kontinuierlichen Experimentfluss mit Konfigurationsänderungen zu diskreten Zeitpunkten einhergeht. In dieser Arbeit werden die notwendigen Software-Änderungen implementiert, um diese Art der Beschreibung eines Experiments für Benutzer der BrainScaleS-2 Plattform zu ermöglichen. Um diese Funktion der dynamischen Rekonfiguration eines Experiments während der Laufzeit zu implementieren, werden konzeptionelle Änderungen auf mehreren Ebenen des BrainScaleS-2 Software-Stacks getätigt. Diese beinhalten einen neuen Typ von Program Builder und eine neue Art, das Experiment aus seinen verschiedenen Einzelteilen zusammenzusetzen. Abschließend wurden die in dieser Arbeit vorgenommenen Änderungen mit einem Performancetest evaluiert und in Anwendungsbeispielen als nützlich erwiesen.

Contents

1 Introduction

Year by year, the demand for neural networks and therefore the amount of necessary computational resources increases [16]. In comparison to artificial neural networks (ANNs), spiking neural networks (SNNs) promise to be more energy efficient [4] and are closer related to their biological equivalent and thus become more and more important [1]. Neuromorphic Systems try to replicate the behaviour of biological neural networks by emulating certain neuroscientific models.
The analog neuromorphic hardware platform BrainScaleS-2 (BSS-2) [14, 10], developed by the Electronic Vision(s) Group in Heidelberg, can emulate spiking and non-spiking neural networks. The application programming interface (API) of the platform [10] makes it possible, to control the system with user-frontends, that are commonly used for working with neural networks, like e.g. PyNN [5] or PyTorch [13]. PyNN is a framework for building neural networks and is used for several different simulator backends as a frontend, to describe the experiments [5]. PyTorch is a tensor library for deep learning, whereas hxtorch [18, 19], the BSS-2-adapted PyTorch extension, already distinguishes itself a little bit from the original PyTorch library. In contrast to ANNs, SNNs have a time axis, as they evolve continuously in time and the dynamics in the system can hardly be paused; in case of the BSS-2 system not at all [10], as this isn't part of the design goal. Especially for biologically inspired experiments, arbitrary dynamics of the system shall be recorded. The main challenge is to give the user an image of a perfectly emulated neural network as it could exist in nature, where among other things, it is expected to be possible to change the experiment configuration at discrete points in time during the experiment. Other simulators like NEST [7] and Brian [20] also allow parameter changes during the runtime of an experiment, which aims to make the description of neuromorphic experiments more simple and intuitive.

In the current operating mode, the experiment starts with an initial configuration of the system, where all parameters are set according to the definition of the experiment, followed by a time evolution of the emulated network with spike stimulus only and the readout of measured data in the end.
The goal of this bachelor thesis is to enable dynamic reconfiguration of the system in the realtime section of an experiment, where the external inputs are fed to the system

and its reaction on it is recorded. Targeting the frontend PyNN, it shall be possible to define different experiment configurations and run them one right after another for individual time durations. The user shall get the feeling of a time-continuous experiment execution, divided into different parts by instant reconfiguration of the simulated system. These reconfigurations can include parameter changes, changes in the network topology or changes, on which values are recorded. To put this feature into effect, new concepts of experiment construction on multiple layers of the software stack are developed. Some of the opportunities this feature opens up are demonstrated in sample experiments.

Following an introduction to the BSS-2 platform, these concepts are elaborated in the methods chapter (chapter 2). There, conceptual solutions tailored for the BSS-2 system are established and justified, regarding the possibilities and restrictions of the hardware.
In the implementation chapter (chapter 3), the changes to the software of the BSS-2 platform are presented, which are necessary to implement the concepts presented in chapter 2. There, the newly introduced data structures and procedures are explained, beginning from the hardware abstraction layer of the software stack up to the modeling wrapper, where all user-visible changes are explained with generic examples.
In chapter 4, a tutorial demonstrating the new feature is presented, as well as an application of it in an already existing experiment about the dynamics of the neurons in the BSS-2 chip. At last, the performance of the new state of the software stack is tested in terms of the compile time for an experiment.
In the end, the results of the implementation are concluded, followed by an outlook on future improvements of this feature and possibilities for other projects to build on this work.

2 Methods

This chapter gives an introduction to the BrainScaleS-2 (BSS-2) platform and how it operates in terms of hardware and software. Only the parts relevant to this thesis are referred to. For a complete presentation of the BSS-2 platform, the papers about the hardware system [14] and the software [10] can be looked up. Then the necessary concepts to reach the goal of a reconfigurable experiment will be elaborated and how the application programming interface (API) can be adapted to these concepts.

In this chapter, some terms are defined, which enable the use of a simple, yet accurate language throughout this thesis. However, these terms may be associated with other meanings in other publications.

2.1 The BrainScaleS-2 platform

BSS-2 is a mixed-signal neuromorphic platform, which emulates spiking and non-spiking neural networks in realtime using analog circuits. The dynamics of the synapse and neuron circuits develop 1000 faster than their biological equivalent. Users can operate it with frameworks, that are commonly used for neuroscientific, resp. machine learning applications, like PyNN [5] and PyTorch [12]. All BSS-2 hardware setups are located at the European Institute for Neuromorphic Computing (EINC), but users from all over the world can remotely execute their experiments on these machines[1].
At the moment, a BSS-2 hardware device consists of a field-programmable gate array (FPGA) coupled with a custom-made ASIC, which is commonly referred to as the BSS-2 chip. The FPGA of each of these hardware setups can be linked to an experiment host which is also stationed at the EINC.

The common workflow of building and executing an experiment is:

1. Description of an experiment on a personal computer with one of the provided frameworks or directly on a lower level of the BSS-2 API.

[1] https://wiki.ebrains.eu/bin/view/Collabs/neuromorphic/BrainScaleS/

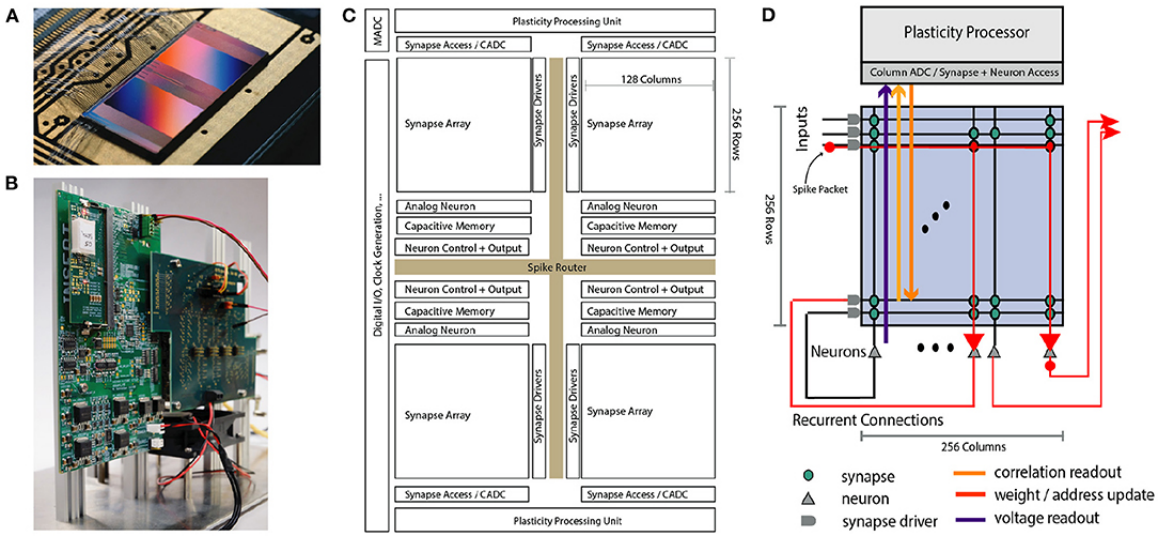


Figure 2.1: A: Picture of the application-specific integrated circuit (ASIC), i.e. the BSS-2 chip. B: Picture of a hardware setup: The chip is located under the white plastic cap on the top left. C: Schematic of the chip with its different components. The components, which are important for this thesis are explained briefly in sec. 2.1.1. D: A closeup on the chip components shown in the top right quarter of C. The red arrows illustrate a possible signal flow. The orange, yellow and blue arrow illustrate the possible read- and write operations of the plasticity processing unit (PPU).
This figure was taken from [14].

2. Interpretation of the code or alternatively build of an executable and a run on one of the experiment hosts provided by the EINC.

3. The program(s) are loaded onto the FPGA by the experiment host.

4. The FPGA then interprets the commands of the program and initiates according operations on the chip or on itself.

5. The specified input signals are generated and processed on the chip to generate the wanted data. All read out data, except for analog recordings, is buffered on the FPGA and sent back to the experiment host after the execution, where it is further processed. Spike recordings or recordings of other parameters are streamed from the chip directly to the host during an experiment.

6. The high-level data structures used in the user program, to describe the experiment, are re-populated with the read out data and are ready for evaluation.

2.1.1 Hardware

At the moment, a BSS-2 hardware device consists of a FPGA coupled with a custom-made ASIC, which is referred to as the BSS-2 chip and several custom PCB, that aren't further relevant for this work. Both the chip and the FPGA will be introduced in this section in order to get an understanding, of how the system works.

The BSS-2 chip

The BSS-2 chip can be divided into different components, which are displayed in fig. 2.1 C.
The upper half and the lower half are roughly symmetrically identical. These are the so-called top and bottom hemispheres. Each of those contains two synapse arrays and the according synapse drivers and neuron circuits with their analog storage, that contains the parameters, which describe the neuron behaviour. The input to each row of synapses is controlled by an individual synapse driver, whereas all synapses in each column of an array are linked to a neuron. In total, there are 256 neurons and 256 synapse drivers on each hemisphere. Each neuron is capable of emulating the adaptive exponential integrate-and-fire (AdEx) [3] equations(2.1)(2.2) and any other model that can be described with parts of the AdEx model.

$$C\frac{\mathrm{d}\,V}{\mathrm{d}\,t} = -g_l\,(V - E_l) \;+\; g_\mathrm{l}\Delta_\mathrm{T}\exp\left(\frac{V - V_\mathrm{T}}{\Delta_\mathrm{T}}\right) \;-\; w \;+\; I_\mathrm{stim} \tag{2.1}$$

$$\tau_w\frac{\mathrm{d}\,w}{\mathrm{d}\,t} = a\,(V - E_\mathrm{l}) - w \tag{2.2}$$

In these equations, C is the membrane capacitance, V the membrane potential, g_l the leak conductance, E_l is the resting potential, Δ_T is the exponential slope, V_T the threshold potential, w is the adaptation current, I_stim the current stimulus, a the subthreshold adaptation strength and τ_w the time constant of the adaptation current. By default, the neurons emulate the leaky integrate-and-fire (LIF) model, which is fully described by the terms in eq. (2.1), that are marked by a blue font. Depending on the signals a neuron receives, it might trigger spikes at certain times, when the membrane potential crosses a certain threshold. These spikes can optionally be fed back into the synapse array, by routing the spikes to a specific synapse driver. In the digital parts of the chip, e.g. the spike router, a spike is nothing else than a time information and a source label, which is used for filtering during event propagation to targets, like e.g. the synapse drivers. The spikes of each neuron can also be recorded by routing the

events off the chip and stream them back to the host during the experiment.
The PPU, which is located at the opposite side of each synapse array (cf. fig. 2.1 D), than the neurons, is responsible for adaptations of the synapses. It has read and write access to all synapses and also can read out different analog values from the neurons. Via the columnar analog-to-digital converter (CADC), the PPU can read out correlations between pre- and post-synaptic spikes or the membrane potential. The PPU is also capable of changing synaptic weights and neuron parameters according to a user-defined plasticity-rule.
The membrane analog-to-digital converter (MADC), located in the to left corner in fig. 2.1 C, can record different analog signals, which are probed at the neurons, e.g. the membrane potential.

The FPGA

The FPGA is mainly responsible for the control flow in experiments. It is connected to the chip via a digital link. It also is connected to the experiment host via the Host-ARQ [10] protocol using Ethernet. The main component of the FPGA is the so-called executor, which can process different kinds of commands, e.g. using a timer to wait until a certain point in time. The FPGA has a memory buffer, off of which it will process these commands. There is only one timer on the FPGA and it has only a single computing unit, so it can only process commands serially.
In the beginning of an experiment, the host generates a set of commands and loads them into this instruction buffer of the FPGA. If this command memory of the FPGA is not big enough to contain a whole experiment, the command batch is split into several parts, that fit into the memory. The FPGA then proceeds to interpret these commands and write data to or read data from the chip. Each command will be interpreted at a certain point in time, either right after the last command, or at a specific time. This can be done by a preceding so-called block-until command, that blocks the execution, until a certain clock cycle count on the timer of the FPGA is reached. If the current timer value is higher than the requested value to be blocked until, the command is ignored. Alternatively, the execution can be blocked to wait for specific events, e.g. until certain access control has finished. When the FPGA encounters a write-command, it changes properties of the system or supplies input data, by writing certain digital values at specified addresses on the chip.

2.1.2 Software

To make the usage of neuromorphic hardware like the BSS-2 system as easy as possible, an abstract way of describing experiments with the same language and structures used in scientific models is wanted. Users prefer to work with abstract objects like neuron populations or projections, which link different populations to each other. To make the operation of the BSS-2 system as easy as coding the experiments in Python-based frameworks, there is need for an API, the implementation of which can translate from such an abstract experiment description to plain serial hardware instructions. In our case, the task is to translate from our two supported user frontends PyNN and PyTorch to hardware instructions for the FPGA. The software stack can be divided into different layers as shown in fig. 2.2, where the top corresponds to the frontend and the bottom to the backend. In this context, "high layer" and "low layer" need to be understood as "rather front- or backend" throughout the thesis. The PyNN-derivate pynn.brainscales [10] and the PyTorch-extension hxtorch [18, 19] appear to a user pretty similar to the original frameworks, but give the possibility to adjust these frontends to the BSS-2 system. They handle abstract objects, like for example populations or projections in PyNN, which are used to describe groups of neurons, resp. the linkage between these groups.
The translation of the abstract experiment description in form of network graphs to signal-flow graphs [17] (cf. fig. 2.3) happens in the experiment description layer. These signal-flow graphs specify the sequence of signal processing steps on the chip, that correspond to a hardware emulation of the abstractly defined neural network.

With the signal-flow graph describing what should happen where and when on the chip, the only thing left to do is to construct sequences of FPGA instructions, that lead to the execution of the signal processing steps specified in the signal-flow-graph (e.g. write specific words to the spike router, that lead to a spike being routed to a certain synapse driver). This is done in the hardware abstraction layer, where the information on changes of all hardware parameters is described with a pair of a coordinate and a container [11]. A coordinate is a specification on the system component, the parameters of which are going to be changed. The container holds the values, these parameters are changed to.

Definition 2.1 *A command corresponds to a set of hardware instructions that induce a certain action in specific components of the BSS-2 system at a certain time. Each command has a (possibly unknown) time stamp, at which it is released to be processed on the FPGA. This time is referred to as a property of a command throughout this thesis.*

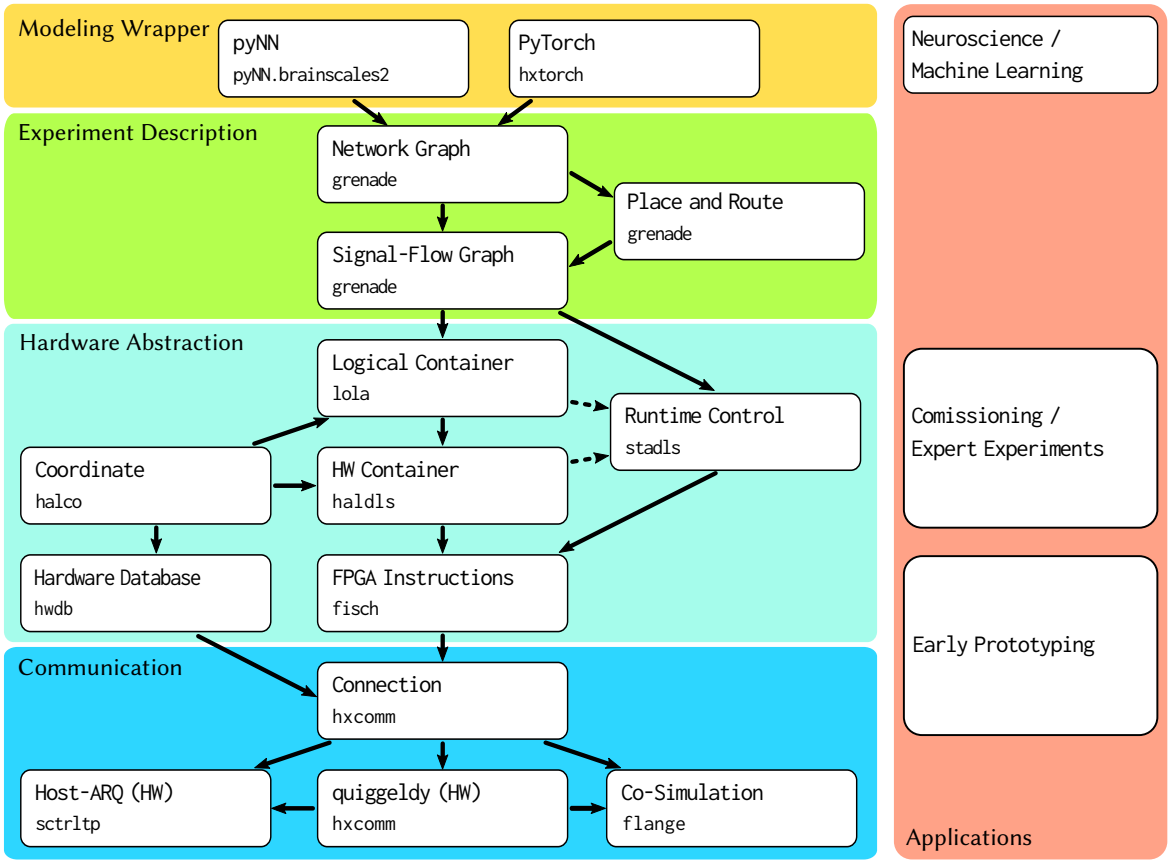
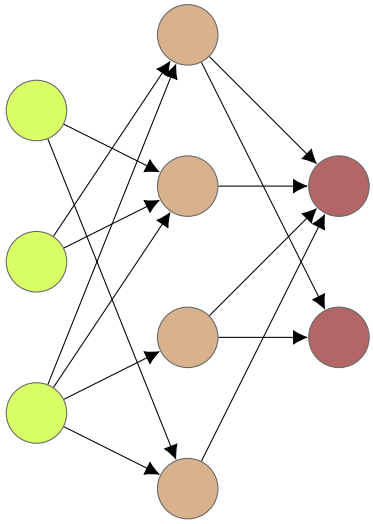
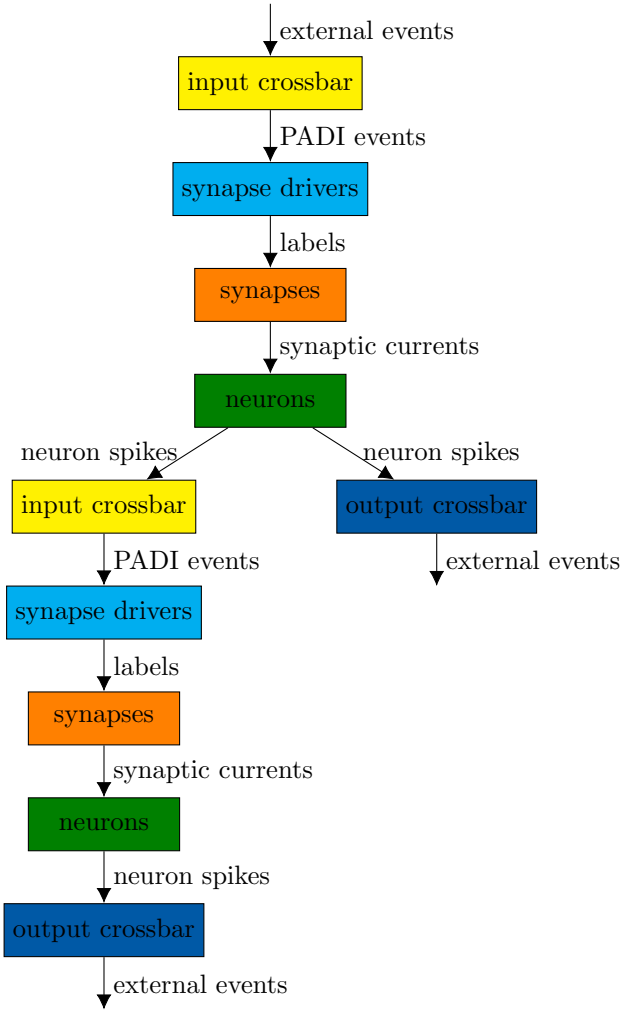


Figure 2.2: The software stack [10] of the BSS-2 platform: The different conceptual layers are marked with different background colors. The white squares resemble different modules of the software stack and show in their upper line with what kind of objects the experiment is described with on the according level of the software stack. On the right side, one can see on which layer the programs want to be coded, depending on the applications. Figure taken from [9].



(a) Neural network representation. The circles resemble neurons, the arrows resemble synapses. The different colors resemble the different populations.



(b) Possible signal-flow graph representation. The squares resemble components of the chip and the labeled arrows resemble the signal type of the communication between these components.

Figure 2.3: Example for source- (left) and end representation of the experimental description layer, adapted from [17].

In the hardware abstraction layer, these commands can be queued with an instance of the PlaybackProgramBuilder (PPB) class, which is used for generating an executable playback program. Such a PPB holds a list of commands and provides different functions to add the three types of commands to this list:

- `write(coordinate, container)`:
  Modifies certain hardware parameters on the system.
- `read(coordinate)`:
  Reads out certain hardware parameters of the system.
- `block_until(coordinate, condition)`:
  Pauses the execution of the playback program on the FPGA until the system component(s) specified by "coordinate" match a certain condition.

The command sequence can be transformed into a playback program, that contains a sequence of FPGA instructions, with a call of the member function `done()`. This playback program now has to be transferred from the experiment host, where it has been built, to the FPGA of a certain setup in order to execute it. In the communication layer of the software stack, the required network headers are added to transfer the program to the specified FPGA and possibly schedule the individual programs of an experiment, alternately with other programs, that are being executed on this setup.

2.2 Experiment procedure

Before this bachelor thesis, the typical procedure of an experiment coded in a high-level interface consisted of:

1. The initialization phase, where all needed chip components are powered up and the initial configuration, which is persistent over the entire duration of the experiment, is written onto the chip.
2. The realtime phase, in which all necessary commands to perform the experiment are executed at their according times and all wanted data is recorded. This phase could optionally consist of multiple batch entries, which can be regarded as repetitions of the experiment, where each of which would only distinguish itself from one another by varying external input data. Using multiple batch entries would be useful for example to present different pictures of the same dataset to the emulated neural network.

3. The final phase, where some of the used chip components are powered down and the system state is read out.

There would also optionally be so-called hooks available, where a more experienced user could inject their own hardware abstraction layer commands at the according places in the experiment, which would be:

1. The pre-static-config-hook: At the very beginning, even before the initial configuration.

2. The pre-realtime-hook: Between the initialization phase and the realtime phase of the experiment.

3. The inside-realtime-begin-hook: At the beginning of each batch entry.

4. The inside-realtime-end-hook: At the end of each batch entry.

5. The post-realtime-hook: Between the realtime phase and the final phase of the experiment.

For a graphic visualization of the former experiment procedure, see fig. 2.4.
So an experiment in the former experiment procedure could only be configured one single time in the beginning, followed by a realtime section where the actual experiment happens. For problems, that cannot be solved with the functionalities of the frontends alone, there also could be hooks injected at multiple different places in the experiment to get the desired results in a more or less "hacky" way.

In a future of functionally complete user frontends, there would be no use for these hooks anymore. This bachelor thesis is a big step in this direction, as it makes it possible to realize the imagination of time-continuously evolving experiments with reconfigurations at given points in the experiment flow. Other kinds of neural network simulators allow for stepping bit by bit through time, possibly with breaks in between these steps, where new configurations can be made to the system. The feature of multiple configurations inside an experiment is getting close to this stepping through time, with the only difference, that all configurations have to be known in the beginning and the experiment is executed continuously afterwards.

A model experiment, where reconfigurations happen at certain times, can be split up into individual snippets for each configuration. One can look at such a snippet as a (re)configuration in the beginning, followed by a time evolution phase, that reaches

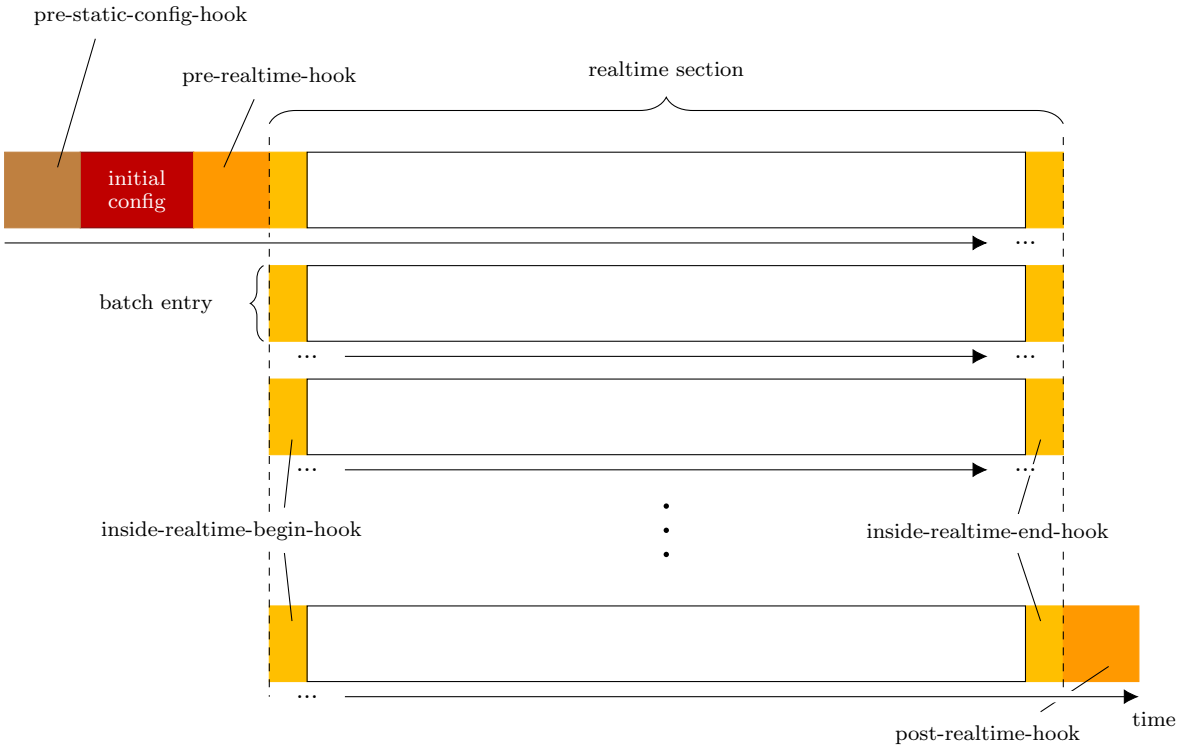


Figure 2.4: Generic schematic of the former experiment procedure.
The temporal order of the different bits of the experiment is from left to right inside each line, whereas the batch entries are ordered from top to bottom. The playback hooks are PPBs, which can be used by advanced users to inject their own commands at the according places in the experiment. The realtime section includes the part of the experiments which is defined in one of the frontends. A realtime section can optionally consist of multiple batch entries, which are repetitions of the experiment with other input data.

to the next reconfiguration, i.e. the next snippet. Thus, a batch entry of the emulation must be further split down into several realtime snippets, that contain a small reconfiguration step at the beginning followed by a portion of the experiment. This adds a new dimension to our experiment procedure schematic, as can be seen in fig. 2.5. This schematic is called the realtime matrix.

There are no hooks depicted in fig. 2.5, because they are not supported anymore for experiments with reconfigurations. Reasons for that are stated in 3.2.

One might notice in fig. 2.5, that the realtime columns overlap a bit with their neighboring columns. This has to do with the desire, to create the appearance of an instant reconfiguration, that takes no time to apply onto the chip. Of course, this

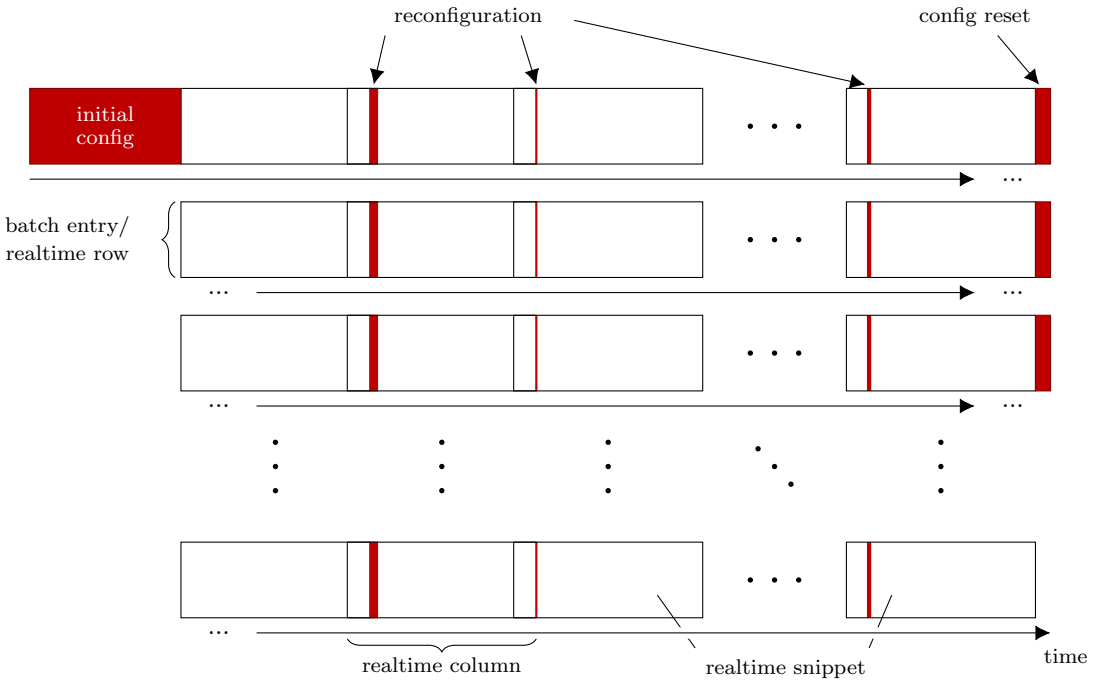


Figure 2.5: The generic realtime matrix, a schematic to describe the new experiment procedure. Each row of the matrix corresponds to a batch entry and each column to a different chip configuration. The "elements" of the realtime matrix are the realtime snippets. Another change to the old experiment configuration is, that at the end of each batch entry, the configuration is reset to the state of the initial config, i.e. all changes of the reconfigurations in the according realtime row are reverted. That has to be done, because the first realtime snippet of the next realtime row demands the chip to be in the state of the initial configuration again.
The time in this schematic evolves, similarly as in the schematic in fig. 2.4 from left to right in each batch entry and the batch entries are ordered from top to bottom in time.
Although not depicted explicitly, the duration of each individual realtime snippet can vary along both axes, i.e. between different realtime columns as well as different realtime rows.

is not possible in reality. But to make the most out of what actually can be done, it is important that the reconfiguration of the chip does not pause the execution of the program, but that these individual snippets are executed one right after another and that the absolute time base of the experiment isn't lost during reconfiguration. Taking a closer look at the exact operations inside a realtime snippet, these can be divided into separate parts. Besides the process of feeding input data to the simulated network and record its reaction to it, some other procedures are required, e.g. starting and stopping individual recordings of analog parameters, cf. fig. 2.6.

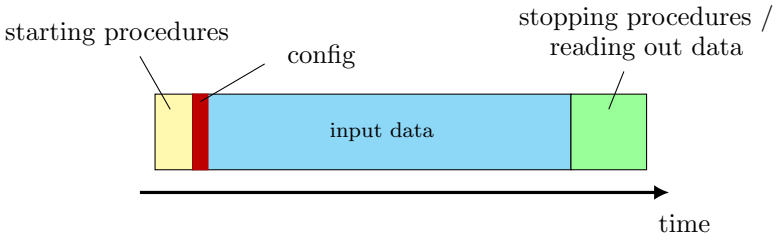


Figure 2.6: Different sections of a realtime snippet:
The yellow section includes commands, like e.g. starting the event recording, if needed. These commands don't require precise timing as they are not directly part of the experiment, but need to be done until the beginning of the input section (blue), so that no data is missed, that is to be recorded. The same goes for the green section, which contains stopping procedures, which mustn't start until the end of the input section. Both these starting- and stopping sections include only commands, that can be triggered quickly on the FPGA, but then need some time on the chip to finalize, so the FPGA is used very sparsely in these sections. The reconfiguration (red) is performed at the beginning and inside of an input section. The reason for that is, that we want to append each input section of a snippet right after the one of its predecessor, but do not want to reconfigure the chip before the previous input section is finished.

The alignment of two realtime snippets, that results from this targeted constellation, is depicted in fig. 2.7.

As the operations in these two parts of a realtime snippet require only a few sparsely distributed trigger commands from the FPGA, the input section would hardly be affected at all by such a merge, as this will only cause few and short delays of a few FPGA clock cycles in the worst case. But as we know from section 2.1.1, playback programs can be executed only serially. Thus, we need to merge the individual realtime snippets together to one singular playback program, which the PPB cannot do - there's need for a mergeable program builder.

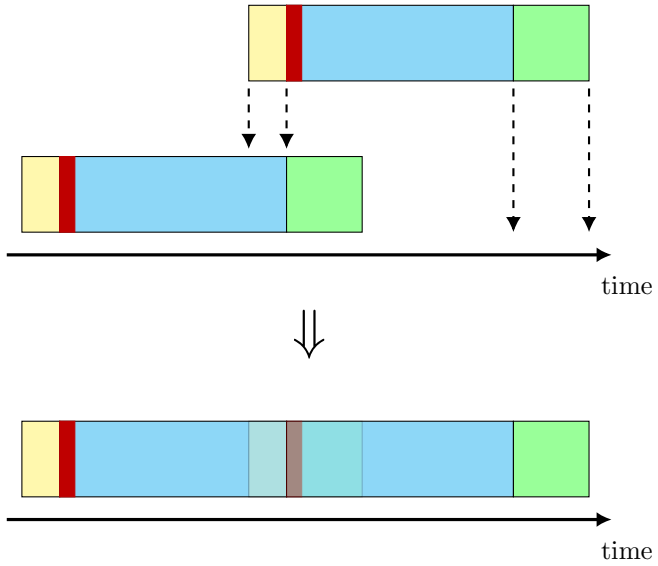


Figure 2.7: Conceptual merge of two realtime snippets:
The yellow block of starting procedures is merged into the previous input section and the green block of stopping procedures with the subsequent input section.

2.3 Concept of a mergeable program builder

To fulfill a merge like shown in fig. 2.7, we need to create a command sequence, that unites the contents of both original command sequences. There can be defined certain rules for this process, which are elaborated in the following paragraphs.

Definition 2.2 *The beginning of a command sequence is the time of the first command. This point in time is called t_0 throughout this thesis.*

t_0 is a fundamental property of a command sequence, as the execution of a command sequence is going to start at some point in time.
In the context of combining realtime snippets together, merging means to create one command sequence out of two and bypass a parallel execution by "shuffling" the commands in a such way together, that all events of both command sequences are still executed at the originally intended times (cf. def. 2.1). A more clear way of expressing our demands for merging two sequences of commands would be the following:

Definition 2.3 *A proper pairwise merge leaves the time interval between each command of a command sequence and its respective t_0 unchanged.*

In the realtime section of an experiment, it is wanted to execute commands at specific times. This involves usually waiting between two commands, which is done with

the block-until command. But apart from waiting on certain events on the chip, e.g. the completion of some operation, it is also possible to block the execution until the timer on the FPGA has reached a certain value, cf. sec 2.1.1. This can be used to wait a fixed time duration between successive commands, e.g. to write input spikes at a certain rate. There are also commands to modify the value of the FPGA timer. By resetting it, one can also wait a relative time duration between two commands. A command sequence, that is mergeable according to def. 2.3 must only contain block-until commands, that block the execution until a fixed point in time, that is already known at construction time of the experiment. Blocking commands, that pause the execution for an unknown time duration might lead to unknown delays of commands, that are scheduled right after these blocking commands. These commands are still guaranteed to be executed, but not at the intended times, which can possibly ruin an experiment. Also, such occurrences of time delays can be hardly detected by a user, as delays are only measurable by reading out the current time before execution of every instruction and comparing against the expected execution times. This is possible, as there is another non-editable timer on the FPGA, as well as on the chip, that is used for time-annotating measurements. It can be read out indirectly by annotating simply nothing with the current time value of this timer. With this, the user can search the whole experiment for the delayed commands by trial and error and fix the problem manually by adjusting the timings. In principle, this check could be implemented automatically for every command by injecting the readout of this timer in front of each command, so any delays would be tracked, but this wouldn't be feasible in terms of runtime and evaluation time costs.
Using the PPB, this problem, is prevented by resetting the timer of the FPGA right after the blocking command. This which makes it possible to schedule commands intentionally after it. But the timer reset leads to a reference problem of the timer values, when merging such command sequences, as an absolute time reference cannot be found. A general example for an erroneous merge is illustrated in fig. 2.8. For this example, a merge according to def. 2.3 is not possible. And most important: An arbitrary command sequence, that can include a block-until command, the blocking duration of which is unknown at build time of the experiment, cannot be merged according to def. 2.3.

In case, that a command sequence is planned to not be merged with any other command sequence, blocking commands with unknown blocking durations are no problem even without using a timer reset, as their occurrences are known inside the command sequence. Therefore, one is aware that absolutely timed commands scheduled right

after these blocking commands are likely to be significantly delayed, if put too close after the time, where the blocking command is initiated, i.e. scheduled for. But for command sequences that are being merged together with other command sequences, that might have scheduled a command right after the time, the blocking command is scheduled in the one sequence, this would lead to an unforeseeable delay of this command and thus is a problem for modular experiment construction.

For merging command sequences into each other, it is necessary, that both of them contain only commands, of each of which the scheduled time (cf. def. 2.1) is known. This would principally also include commands that are timed relatively to another command of which the time is known, if the relative time difference is also known at compile time. In practice, this would be possible by making a timer reset after a command and block until a certain timer value, or schedule several commands, the execution duration of which is known, right after another. But this is unnecessary, as the same command sequence could be scheduled without any relative timing in any such case. Looking again at the example in fig. 2.9, scheduling the timer reset relatively to the blocking command with the unknown execution duration led to the problem of losing the time reference to the part before the blocking command, i.e. to the beginning of the command sequence. Thus, to guarantee, that a command sequence can be merged with another arbitrary command sequence, it is sensible to ban any relative timing inside command sequences. Without allowing relative timing, modifications of the FPGA timer would also be unnecessary, as they simply create a known offset to all scheduled times, which can also be added by simple calculations in the software. But having it as part of a command sequence would still take additional computing effort when merging. The merging process would namely be much more complicated, if it would have to account for timer modifications with according offsets for the times of all commands of the other command sequence coming after the timer reset.

But functionalities like blocking commands and relative timing can't just be thrown overboard, as they are required for operating some components of the chip, e.g. the PPU. The duration of the startup of the PPU for example varies depending on the initialized global data structures, as it has to make several external memory accesses, if necessary. The PPB still has its right to exist. Thus, we need a new builder pattern that excludes all forms of relative timing, including blocking commands with unknown blocking duration and timer modifications: The AbsoluteTimePlaybackProgramBuilder (ATPPB).

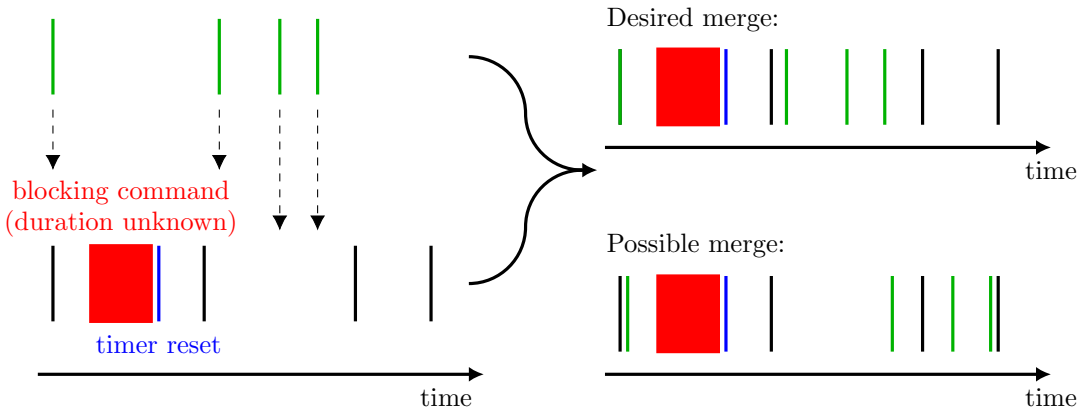
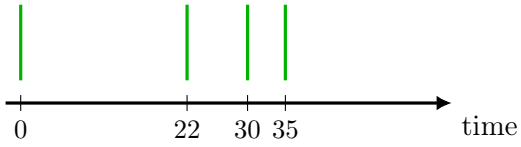


Figure 2.8: Example for two command sequences, that cannot be merged according to Def. 2.3. The vertical lines on the time scale resemble commands executed at the according times. Line thickness resembles the time it takes to process the command. According to def. 2.3, an execution like shown in the upper right command sequence would want to be seen, where each command of both sequences would still be executed at its scheduled time, where it initially was meant to be executed. But that is not possible for two reasons: The FPGA has only one single execution unit, that can process the commands one at a time. Thus, the commands can only be processed sequentially and not at the same time, as wanted for the first two commands right at the beginning. Secondly, to account for the timer reset, all numbers of FPGA clock cycles in the block-until statements coming before the `write` commands according to the green lines in fig. 2.8 have to be adjusted by a certain offset.
To keep the lines at the same distance to the beginning of the command sequence, as before the merge, the duration between the FPGA timer reset and the beginning of the sequence has to be subtracted from the original values in the block-until commands. But this duration depends on the duration of the preceding blocking command, which is unknown before the experiment execution. However, these merges of command sequences happen while building the experiment, i.e. before the duration of the blocking command can be known.
In reality, a merged command sequence could possibly look like the one on the bottom right. One could call this the naive merge, as the blocking commands that regard the FPGA timer are just ordered by the magnitude of their argument, i.e. the FPGA timer value, until they block the execution. All other commands are scheduled after the same block-until command as before (cf. fig. 2.9). But this solution violates def. 2.3 and isn't practicable, as the delays of all green commands after the timer reset can be too large that the constructed experiment still can work.

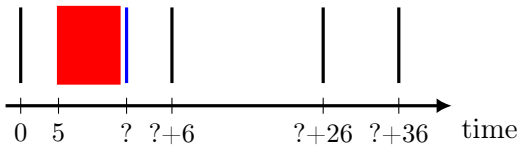
Base sequence 1:

```
write(spike)
block_until(21)
write(spike)
block_until(29)
write(spike)
block_until(34)
write(spike)
```



Base sequence 2:

```
write(spike)
block_until(4)
block_until(omnibus_is_ready)
write(timer_reset)
block_until(5)
write(spike)
block_until(25)
write(spike)
block_until(35)
write(spike)
```



Merged sequence:

```
write(spike)
write(spike)
block_until(4)
block_until(omnibus_is_ready)
write(timer_reset)
block_until(5)
write(spike)
block_until(21)
write(spike)
block_until(25)
write(spike)
block_until(29)
write(spike)
block_until(34)
write(spike)
block_until(35)
write(spike)
```

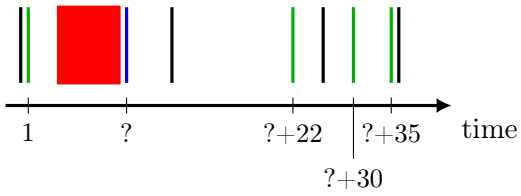


Figure 2.9: Naive merge of two command sequences. Base sequence 1 is merged into base sequence 2. The naive result is shown at the bottom.
On the left, the sequences are shown in form of a list of pseudocode on the hardware abstraction layer, as they might be scheduled by a user. To get this result, each `block_until` command together with the next subsequent command of base sequence 1 is inserted in the according place in base sequence 2, so that in the resulting command sequence all occurring `block_until` statements are sorted by the value of their argument.
On the right, you can see the according timeline of the actual execution.

The conceptual differences of the ATPPB compared to the PPB allow for some changes in the handling of the command timings. As now the times of all commands (cf. def. 2.1) are known due to their absolute timing, this information can be strictly coupled to each command as a mandatory property. This makes the block-until commands that block until a certain FPGA timer value unnecessary. Thus, the ATPPB can be limited to just the other two remaining kinds of commands:

- `write(time, coordinate, container)`:
  Modifies certain hardware parameters on the system at a certain time.
- `read(time, coordinate)`:
  Reads out certain hardware parameters of the system at a certain time.

A short comparison of the two different ways of scheduling the commands is shown in fig. 2.10 by a merge of two command sequences, that contain only absolutely timed commands.

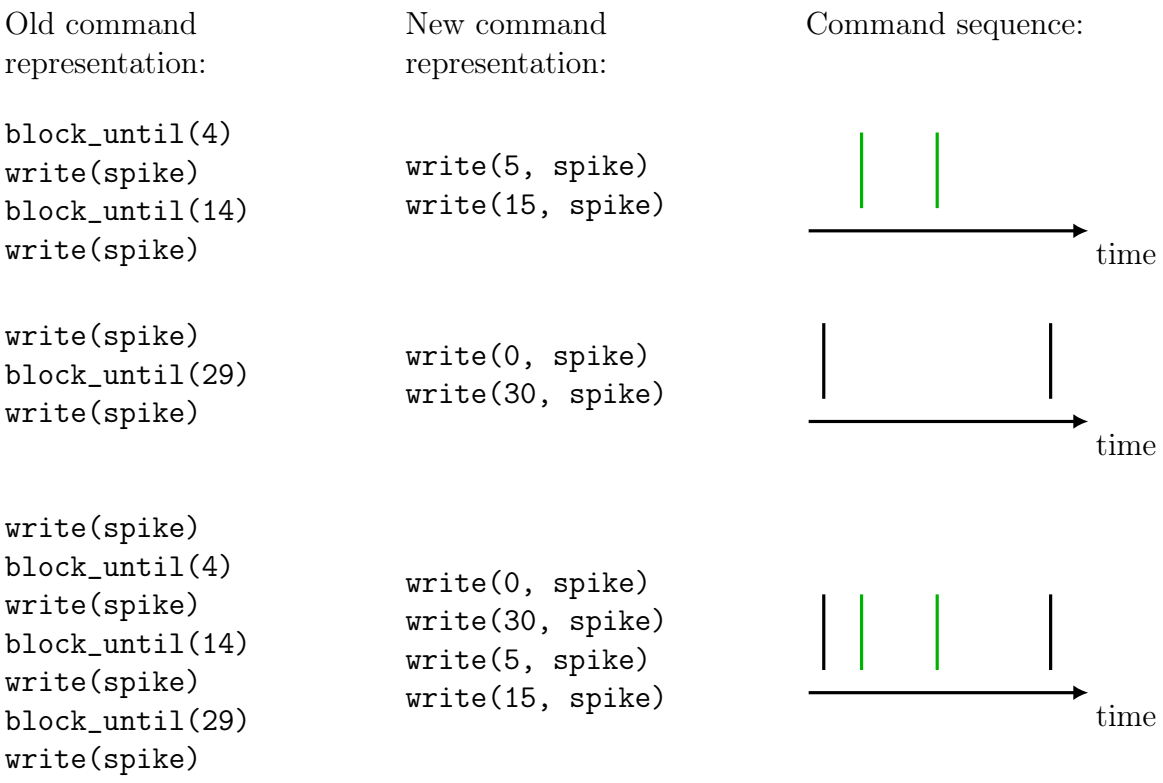


Figure 2.10: Comparison of old and new semantic for scheduling commands on an example of a successful merge.

3 Implementation

This chapter will focus on the concrete changes to the implementation of the BSS-2 API, that are made in order to implement the concepts elaborated in the previous chapter. All created and used data structures, as well as important implementation details are presented, beginning with the ATPPB in the hardware abstraction, over the GRaph-based Experiment Notation And Data-flow Execution (grenade) layer to the user-frontend pynn.brainscales.

3.1 The AbsoluteTimePlaybackProgramBuilder

For scheduling a command at a certain point in time, the PPB already offers the necessary tools: The `block_until()` command, that blocks the execution of the playback program on the FPGA until the count of clock cycles that passed since the last timer reset reaches the given argument.

> **Example 1:**
> To write a spike to be released at the 1000th FPGA clock cycle after the latest timer reset, one can code:
>
> ```
> ppb.block_until(999);
> ppb.write(SpikePack1ToChipOnDLS(),
>           SpikePack1ToChip({SpikeLabel(0)}));
> ```
>
> where `ppb` is the name of the PPB object, to the command list of which these two commands are added to.

So by scheduling all commands in that manner and not modifying the value of the FPGA timer after its initial reset in the beginning, all commands in a command list are - at least in the context of an individual experiment - absolutely timed. So our ATPPB-commands, that have their additional time argument (cf. sec. 2.3), could be translated to just such a pair of commands, as shown in example 1. Thus, the ATPPB isn't an entirely new builder class standing on the same layer of software as the PPB, but rather can be seen as an interface of the PPB, that accepts and holds data in a

different structure and passes its commands restructured to a PPB instance to create an executable playback program. In the following paragraphs, this new structure is firstly presented and then the process of restructuring the data and feeding it to an instance of the PPB is looked at.

3.1.1 Data structure

The most important piece of data, each ATPPB object holds, is the set of commands, that are fed into the builder. This is implemented as a vector named `m_commands` containing elements of the type `CommandData`, which by itself is a struct that holds all data according to one call of a `write` or `read` function (see fig. 3.1).

```
std::vector<CommandData> m_commands;
struct CommandData
{
    Timer::Value time;
    std::unique_ptr<Container::Coordinate> coord;
    std::unique_ptr<Container> write_config = nullptr;
    std::unique_ptr<Container> write_config_reference = nullptr;
    std::shared_ptr<AbsoluteTimePlaybackProgramContainerTicketStorage>
        read_ticket_storage = nullptr;
                                        ⋮
};
bool m_is_write_only = true;
```

Figure 3.1: Extract from the header file of the ATPPB showing the most important member variable `m_commands` containing all information on the scheduled commands. Each instance of `CommandData` holds the according data for one command, which includes the time it is scheduled for, a coordinate and optionally a container, a reference container or an AbsoluteTimePlaybackProgramContainerTicketStorage (ATPPCTS). The coordinate specifies the addresses on the system, to which the data contained in the container is written to, or read from, depending on the command (see below).

The ATPPB supports three different types of commands, that each correspond to a slightly different set of data, which has to be stored in their according `CommandData` instance:

1. (normal) write:
   `void write(Timer::Value time, Container::Coordinate coord, Container config)`

is executed on the FPGA at time `time` and overrides words at the addresses specified by `coord` with the according values specified in `config`.
Thus, a corresponding `CommandData` Object holds `time`, `coord` and `config`.

2. differential write:
`void write(Timer::Value time, Container::Coordinate coord, Container config, Container config_reference)`
is executed on the FPGA at time `time` and overrides only certain words at the addresses specified by `coord`, where the values of `config` and `config_reference` differ (cf. fig. 3.2). Thus, a corresponding `CommandData` Object holds `time`, `coord`, `config` and `config_reference`.

3. read:
`AbsoluteTimePlaybackProgramContainerTicket read(Timer::Value time, Container::Coordinate coord)`
is executed on the FPGA at time `time` and reads out the words at the addresses specified by `coord`. At the end of the execution, where all expected data is collected and has arrived at the host, the read out data can be identified with the according `AbsoluteTimePlaybackProgramContainerTicket (ATPPCT)`, which then can be used to retrieve the data (cf. sec. 3.1.2). Thus, a corresponding `CommandData` Object holds `time`, `coord` and `config`.

From that, it can be seen, that not every command needs the members `write_config`, `write_config_reference` or `read_ticket_storage` to store all its information in a `CommandData` instance. Thus, these members are `nullptr` by default, which can be used to identify their type later on when building the program. There's also a boolean `m_is_write_only` for an ATPPB, that holds the information on whether `m_commands` contains any elements corresponding to a read command. This decides, whether an ATPPB instance can be copied or not, cf. sec. 3.1.4.

3.1.2 Retrieval of experiment data

Obviously, parameters can only be measured as soon as the experiment is running on hardware and thus, they aren't available to be returned directly by the `read()` function, when called. But there has to be some immediate return value, because otherwise there is no way to identify this `read()` statement or the API user, that induced this read with the returning data after the experiment.
The solution for that is a so-called container ticket, that can access the data, when it is retrieved from the experiment. This container ticket is returned when calling the

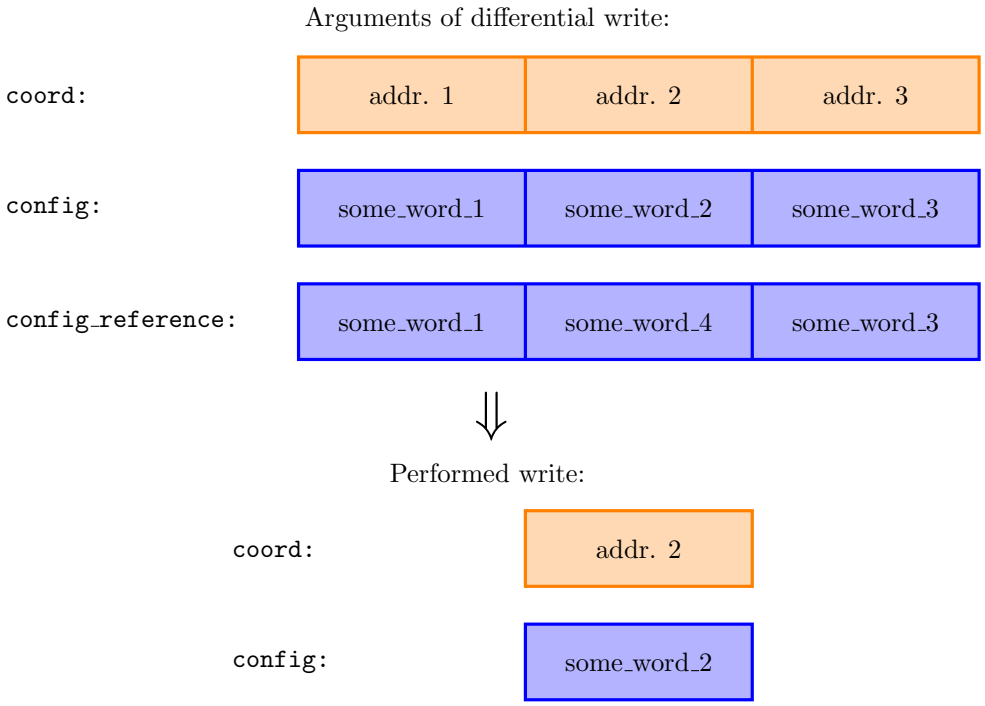


Figure 3.2: Example for a differential write: The container `config` consists of 3 words, so the coordinate `coord` holds the three according addresses where each of these words shall be written to. The only word differing between `config` and `config_reference` is the word according to addr. 2. Thus, only some_word_2 needs to be written to addr. 2, to get the container `config` written at the coordinate `coord`.

`read()` function of the PPB.
But when calling `read()` on the ATPPB, the according container ticket is not existent yet. It will be created in `done()`, cf. sec. 3.1.3, so the same problem as with the real container data, that is to be measured arises now for the container ticket.
This is where the AbsoluteTimePlaybackProgramContainerTicket (ATPPCT) comes into play. When calling `read()` on an ATPPB an instance of the ATPPCTS class is constructed. The ATPPCTS can optionally hold a container ticket, which it initially doesn't, because it can't out of the reasons described above. But when `done()` is called on the ATPPB, a container ticket is generated from the emerging PPB (cf. sec. 3.1.3 and filled into the ATPPCTS. Both the ATPPB and the freshly generated ATPPCT, that is going to be returned by the `read()` function, hold a shared pointer to this newly constructed ATPPCTS instance. The ATPPB holds the shared pointer on the ATPPCTS inside the `CommandData` object belonging to the `read()` statement and the ATPPCT holds it as a private member.
When the ATPPCTS is finally filled with a container ticket, the ATPPCT can access the measured data via the container ticket, by calling the `get()` function on the ATPPCT. However, this process will fail, before the underlying container ticket isn't valid, i.e. doesn't hold yet the according data, because the experiment hasn't already terminated. If the data is already ready for retrieval can be checked by using the `valid()` function of the ATPPCT. Besides these two functions, the entire interface of the ATPPCT is identical to the interface of the underlying container ticket, which is used when handling read commands of a PPB.

3.1.3 Building process of the PlaybackProgramBuilder

To generate a PPB out of the vector `m_commands` that contains all command data, the function `done()` is called.
The first thing that is done after the call of the function is to sort all elements of `m_commands` by the magnitude of their `time` attribute. The approach to sort the `m_commands` vector one single time in the end was made over the approach of inserting each newly added command at the right place into `m_commands`, as it is overall more performant. Then, a PPB is created and one initial timer reset is written, yielding a FPGA timer value of 0 as our time reference for the beginning of the command queue.
Then, iterating over each `CommandData` instance in `m_commands`, blocking commands and write or read commands take turns to be scheduled on the PPB in a similar fashion to ex. 1. There is a `Timer::Value`, that keeps track of the currently expected value

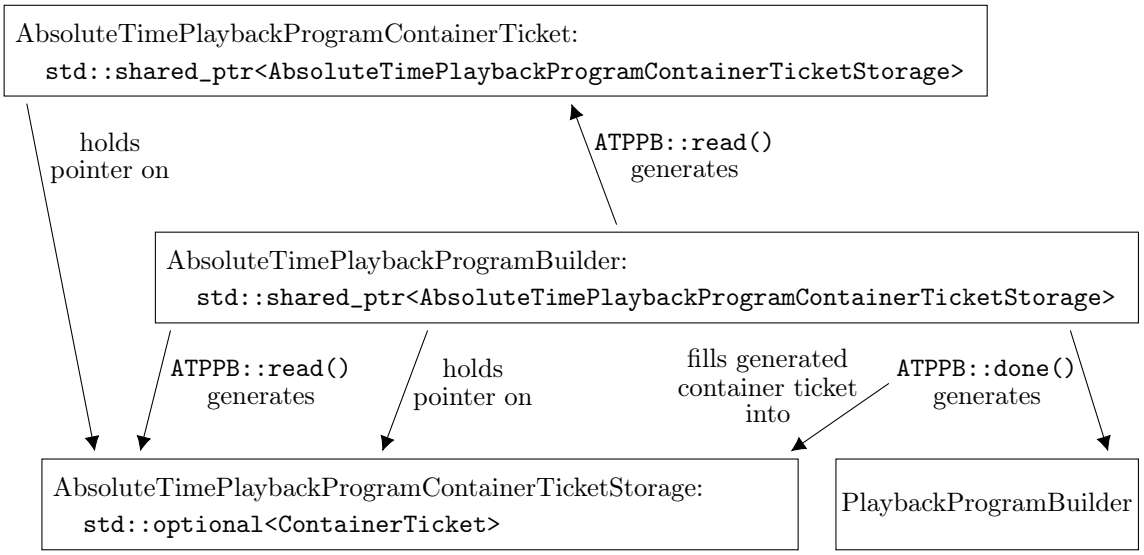


Figure 3.3: Necessary data structures and processes to make data, measured due to a certain `read()` statement, accessible via an ATPPCT. The `read()` function of the ATPPB generates an ATPPCTS and an ATPPCT pointing on it, which is returned for the retrieval of the measured data after the experiment is run. The retrieval works internally via retrieving the data from a container ticket, which is generated and put into the ATPPCTS during the process of generating a PPB out of the ATPPB in `done()`, see sec. 3.1.3.

of the FPGA timer, whereby it accounts one FPGA clock cycle for each write or read command. If the value of this timer isn't exceeded by the time of the following command, the `block_until` command is omitted, as it is unnecessary. This provides the benefit, that the execution on the FPGA happens quicker at passages of the sequence, where the commands are packed densely, as the FPGA doesn't need to interpret additional `block_until` statements, that wouldn't block the execution anyway. In the other case, that the currently expected value of the FPGA timer is lower, than the time the current command is scheduled for, the according blocking statement is made and the `Timer::Value`, that keeps track of the currently expected value of the FPGA timer is updated.

As explained in sec. 3.1.1, the type of command according to a `CommandData` object can be identified by looking at which of the member variables are set. Depending on this, a write, a differential write or a read command is called on the PPB. If the latter is the case, the new container ticket, which is returned by the read function of the PPB, is filled into the ATPPCTS, which validates the issued ATPPCT.

For the purpose of sorting the vector `m_commands` in the beginning, `std::stable_sort`[1]

[1] https://en.cppreference.com/w/cpp/algorithm/stable_sort

is used. As the name of this function intends, it uses a stable sorting algorithm, which keeps commands that are scheduled for the same time in their original order, regarding each other. This is a useful feature, as it provides the possibility to execute multiple commands right after each other by passing the same time argument for every command, while the predefined order is retained.
After the PPB is created and filled up with all the command data, the `m_commands` vector is emptied to release the acquired resources as quickly as possible. Any ATPPB, that contains a read command cannot be used twice either way, due to reasons stated in sec. 3.1.4. In other words, the call of `done()` "consumes" the ATPPB to create the according PPB and all that remains is just an empty instance of the ATPPB. Thus, it is usually the last step in the lifetime of an ATPPB.

3.1.4 Additional features

Until now all data structures and the basic working principle of the ATPPB got to be known, but the main feature which the ATPPB was developed for is yet to be presented:
The functionality of merging the contents of two ATPPBs. This can be done with the `merge()` function, which combines the `m_commands` vectors from both ATPPBs. To be more precise, it moves all elements from the command list of the ATPPB instance, which was passed as a function argument into the `m_commands` vector of the ATPPB, on which `merge()` was called on. The movement of the commands leaves the other ATPPB empty, which can be avoided by using the `copy()` function. This function copies all entries from the other ATPPB's command list and appends them to the `m_commands` member of the own ATPPB. So, the only difference between `merge()` and `copy()` is, that contents of the merged ATPPB are copied or emptied.
Thanks to the data structure of the ATPPB and the way, `done()` works, merging is as simple as throwing both sets of command data together into one, without considering the content of any `CommandData` instance. The only thing, that has to be recalculated is the `m_is_write_only` member of the ATPPB into which is merged, which is a pretty quick and simple task.
Another difference between `merge()` and `copy()` is, that the `copy()` function restricts the other ATPPB to be write-only, as a duplication of an ATPPB, that includes read statements is problematic. Because if both these ATPPBs will be converted into a PPB, two container tickets are created and tried to be stored in the ATPPCTS of the according ATPPCT, that was created in the initial read statement, which happened only once. But each ATPPCTS stores only one container ticket, so providing two, that

possibly contain different data is ambiguous. Therefore, it is prohibited by banning the usage of the `copy()` function on ATPPBs, that aren't write only.

Having the time information of a command stored in `CommandData` allows for operations on the time attribute of all `CommandData` instances stored inside of an ATPPB. The ATPPB offers four operators in this matter:

1. `+=` operator:
   adds a certain `Timer::Value` onto the time attribute of all `CommandData` instances inside the ATPPB

2. `+` operator:
   makes a copy of the ATPPB and then applies the `+=` operator on the copy

3. `*=` operator:
   multiplies the time attributes of all `CommandData` instances inside the ATPPB by a floating point number.

4. `*` operator: makes a copy of the ATPPB and then applies the `*=` operator on the copy

Especially the `+=` and `+` operators are handy tools for assembling an experiment out of several ATPPBs, as they are a shift of t_0 of the command sequence of the according ATPPB. This can be used, to move the individual ATPPBs to their correct place in an experiment, while constructing it out of its individual parts, see sec. 3.2.

3.2 Experiment construction

As described in sec 2.2, the key to a reconfigurable experiment is a new modular way of putting these experiments together. For reconfiguring an experiment, the user just needs to provide a list of different experiment configurations and an information, on how long they want to run the experiment in this configuration. That corresponds to a list of realtime columns, as described in sec. 2.2. As a short repetition: A realtime column is the entirety of all realtime snippets of the realtime matrix, that belong to the same system configuration. A realtime column contains the realtime snippets of all batch entries, that share the same temporal index inside each batch entry.
In the grenade layer, such an experiment configuration is described by an `IODataMap` object containing the input data of this realtime column, a signal flow graph describing the realization of the network topology on hardware (cf. sec. 2.1.2) and a `Chip` object

that holds the information on the required settings of all chip components. The output data, that was returned from the system as a result of this realtime column is also contained in an `IODataMap`.
In the old experiment procedure, the entire realtime section of the experiment consisted of a singular realtime column, so for the description of the entire experiment, only a singular of each of these objects was required as an argument, resp. returned. In order to get the additional dimension of the consecutive realtime columns, these four data structures have to be vectorized. In other words, instead of single objects, there will be vectors containing possibly multiple of these objects as a replacement in all parts of the experiment description layer of the software stack, that handle these objects in the process of experiment construction.
To initiate an experiment run, the data necessary for the experiment construction is passed through the function `grenade::network::run()`, which among others, takes vectors of IO-data-maps, network graphs, from which the signal-flow graphs can be extracted and chip objects as arguments and returns the generated output data also in form of a vector of IO-data-maps, which matches the vector size of the other three vectors, i.e. one per realtime column.

Ideally, one would expect, that an experiment with multiple realtime columns still would support playback hooks and plasticity, i.e. the usage of the PPU. Especially the usage of the PPU in combination with the multi-configuration feature is highly desirable. However, when having an experiment with multiple realtime columns, playback hooks which are nothing else than PPBs, cannot be implemented into the experiment procedure at their intended places, as they cannot be merged. In general, they also could break the absolute time reference, as they can contain modifications of the FPGA timer. Also, everything PPU related is forbidden in experiments with reconfigurations, because it is mostly based on relative timing of commands, like executing most commands right after another without knowing the execution durations of any of them. As the old experiment procedure still has applications, playback hooks and the usage of the PPU must still be supported. Thus, these the two scenarios of having only a singular realtime column, where the old experiment procedure can be applied, or the new experiment procedure with multiple realtime columns have to be distinguished with checks, that assure that neither the PPU nor any playback hooks are used, when having multiple realtime columns.
However, this doesn't mean that it is impossible to support the use of the PPU in experiments with multiple realtime columns. This only requires some additional features of the software and the FPGA, that enable the PPU to stream the measured

results to the FPGA, where for each realtime snippet, a dedicated area in the memory is reserved. Same goes for the management of the different plasticity rules as input for the PPU, according to the different realtime columns in the FPGA memory, that would all together have to be pre-loaded before the realtime section of the experiment. In terms of software, a memory management system would be needed for that to work out and in terms of hardware, an effort to make the memory used for the PPU data quicker, so that it is no bottle neck to the process of streaming the measured data from the PPU to the FPGA. This prevents long durations for writing data to the host between different realtime snippets. The data transfer to the host could then happen between different batch entries, where timing is no crucial factor.

After the `grenade::network::run()` procedure is called, the process of constructing the experiment is triggered. There, one realtime column at a time, the ATPPBs of the individual realtime snippets are generated out of the input data, the signal-flow graph and elements of the configuration. In the process of scheduling the commands of each realtime snippet into an ATPPB, a `Timer::Value` counts the current time by accounting for most commands the number of FPGA clock cycles needed for the FPGA to execute the commands. In some occasions, this duration is not known and a fix time interval in the order of a few μs is added to the current time value. These basically replace the former blocking commands, but were manually tuned to achieve the best possible performance without blocking commands, while still waiting long enough for the previously scheduled commands to finish.
This `Timer::Value` is passed as the time argument of each command, that is added to the ATPPB, excluding commands of the input section of a realtime snippet (cf. sec. 2.2), as they are generated separately. This ensures, that all commands are packed as tightly together as possible. Also, the value of this timer right before the injection of the input data into the command queue is stored separately, as it is used later in the assembly of the different realtime snippets to shift them accordingly in time, cf. ex. 2.

Then, all parts of the experiment can be combined. The first step for that is to assemble all individual batch entries, i.e. realtime rows, which is very simple for the case of a singular realtime column, as the inside realtime begin and inside realtime end hooks only have to be prepended, resp. appended to the delivered ATPPB, after converting it to a PPB. In the case of a multi-configuration experiment however, the individual snippets have to be merged according to the concepts developed in sec. 2.2, which is done snippet by snippet as shown in ex. 2.

Example 2:

The following code paragraph shows how a realtime snippet `snippet` is merged into the already assembled part of a batch entry, i.e. an ATPPB, which is called `program_builder` in this example.

Let `j` be the index of this snippet inside the batch entry, i.e. the index of the according configuration. Each `RealtimeSnippet` object that is provided for assembly, holds an ATPPB named `builder` where all commands are scheduled and the previously described `Timer::Value` named `pre_realtime_duration`, cf. fig 3.4, as well as a `Timer::Value` named `realtime_duration`, which is the duration from the beginning of the realtime snippet until the end of the input section, depicted blue in fig. 3.4. Let `config_time` be the time of the end of the yet assembled part of the batch entry. Then this process looks like this:

```
program_builder.write(config_time, ChipOnDLS(),
                      configs[j], configs[j-1]);
snippet.builder += (config_time - snippet.pre_realtime_duration);
program_builder.merge(snippet.builder);
config_time += snippet.realtime_duration;
```

In reality, this of course is done by iterating over all the snippets from the different batch entries and realtime columns, but the process is the same. One can see, that the differential write function of the ATPPB is used for writing the new configuration of each realtime snippet, as it saves a lot of time to only write the usually small changes in the configuration, compared to the overall size of a configuration.

After combining all realtime snippets into a batch entry, the configuration has to be reset to the state of the initial configuration, as the first realtime snippet of the next batch entry needs to run on the initial configuration. This is simply done by using the differential write again in a similar fashion to how it is used in ex. 2.

The resulting ATPPB, that contains all commands of the whole batch entry, can then be converted to a PPB. The only pieces of the experiment, that still have to be mounted to the already constructed part of the experiment, are PPBs, that need to be concatenated according to fig. 2.4 resp. 2.5, depending on whether there are multiple realtime columns given from the higher layers.

Because the FPGA has a limited buffer for the playback program, i.e. the instructions it has to process, there is a maximal size for each playback program. Therefore, all the available PPBs cannot be put together at once, but have to appended one by one to

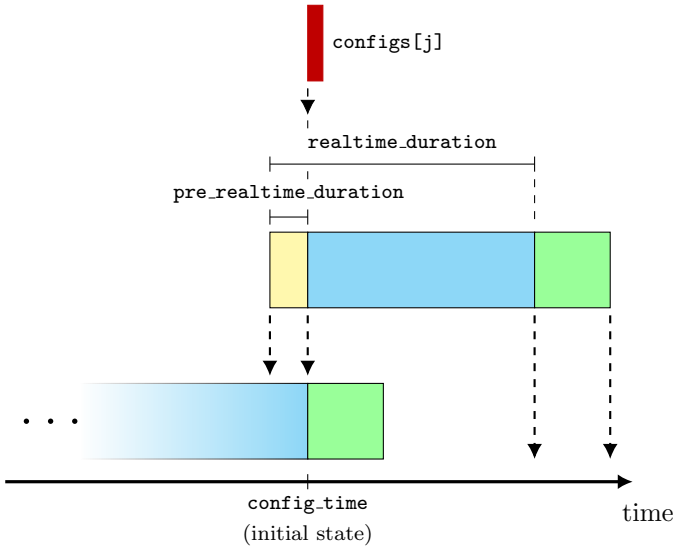


Figure 3.4: Graphic explanatory support for ex. 2. The realtime snippet `snippet`, which is shown in the middle, is to be merged into the already constructed part of the batch entry, depicted at the bottom of this image. For this purpose, the time reference t_0 (cf. def. 2.2) of `snippet` is shifted by a `Timer::Value` of `config_time - snippet.pre_realtime_duration`, cf. ex. 2, by using the `+=` operator of the ATPPB (cf. sec. 3.1.4). In addition to that, the configuration `configs[j]`, that belongs to this snippet, is injected at the beginning of its input section (blue), as intended in sec. 2.2. A schematic of the result can be seen in fig. 2.7.

a PPB, that is filled up until it cannot take another pre-built PPB without exceeding the size limit for the resulting playback program. Sometimes it might be, that some bigger PPBs, e.g. the ones containing a whole batch entry, exceed the size limit, even if they aren't combined with any other PPBs. In that case, the part of the playback program, that doesn't fit into the instruction memory of the FPGA must be streamed from the host to the FPGA while the buffer is already emptied during execution, in hope of managing to never let the FPGA run out of commands until the end of the playback program, as this would cause delays in the experiment, which would violate correct experiment execution requirements.

After the playback programs are generated, the hardware run is triggered and the playback programs are executed, returning all measured data of the experiment. This data is then postprocessed for each individual realtime column. There, the data is brought into the shape, which it is expected to be of higher level procedures.

The last step in the experiment description layer is to return the bundled output data to the frontends via detours as the return value of the `grenade::network::run()` function.

3.3 Changes to pynn.brainscales

As pynn.brainscales is coded in Python, we deal with lists instead of vectors on this layer of the software stack, which are converted to `std::vector` by the Python wrapping, when calling `grenade::network::run()`.
To supply the experiment description layer with lists of input data, signal-flow graphs and chip objects, we need to acquire the different configurations and temporarily store them, until the hardware run is performed. The aim is to change the operation of pynn.brainscales as little as possible by implementing this new feature.
Until now, a typical experiment in pynn.brainscales consists of these three parts:

1. In the beginning after the initialization of a simulator object, which is done by calling `pynn.setup()`, the experiment is configured. This includes the definition of so-called populations, which each is a set of neurons with customizable properties and projections, that hold the information about which populations are connected to each other in which way. External input events for example can be encoded in the population, by using a neuron type, which fires spikes at the given times.

2. When the configuration is finished, the experiment can be run on the BSS-2 hardware for a certain time duration, which can be passed as an argument to the `pynn.run()` function.

3. After the run, all measured neuron data is stored in the population objects and labeled with information on the index of the neuron, to which the measured data belongs.

3.3.1 Changes in experiment description

To gather a list of experiment configurations, the first of these three steps, i.e. the network configuration, has to be repeated in some way, followed each time by some function call, which takes care of caching the currently defined configuration into lists of network graphs, lists of IO-data-maps for the input data and lists of chip objects. Every defined configuration state corresponds to a triplet of these objects, so the number of different configurations corresponds to the length of these lists, i.e. the number of realtime columns.
For the purpose of filling these lists, the new function `pynn.add(runtime)` is introduced, which appends the experiment by a phase with a duration of `runtime`, that runs in the currently defined configuration. How this works is, that the function

`preprocess()` is called, which generates the according network graph out of the currently defined configuration. Out of the network graph, the input data can be extracted in form of a `IODataMap`. The chip object is created when initializing or changing model parameters of a calibrated `HXNeuron`, or injected manually by the user once in the beginning and updated in each call of `preprocess()`. The current states of all three objects are appended to their according lists, whereas the chip object and the network graph have to be copied explicitly, as they might be modified in the upcoming sections of the experiment. The runtime argument of the `add()` function is passed as a floating point number of milliseconds, adhering to the PyNN upstream time unitization. Then, it is converted to the according duration in FPGA clock cycles and injected as runtime information into the IO-data-map of the inputs. It also is cached in a list, as it is later after the execution needed for the preparation of the measurement data.

This updating and making a snapshot of the current configuration state inside the `add()` function, ensures that the user can define the network topology and the initial configuration once in the beginning, but then only has to modify the configuration between the `add()` calls. Instead of writing the complete definition of the populations and projections again, only according parameters need to be adjusted, e.g. which parameters are to be recorded in this phase of the experiment.
After the last reconfiguration, instead of calling `add()` for a last time, `run()` can be called directly, as the `add()` function is called inside of the `run()` function, if a non-zero value is passed as a time argument. On first thought, this might seem a little confusing, as it would be more logical to have pairs of configurations and `add()` calls until the end and call `run()` afterwards to only trigger the hardware run. This is done for reasons of backward compatibility, as also here in pynn.brainscales, the support for working the same as in the past must not be lost to prevent all formerly created experiments from failing. Also, the API conventions of upstream PyNN must not be broken, which they would with the necessity to call `add()`, in order to execute an experiment. In this PyNN-conform case, writing a program like shown in the enumerated list above is still possible.

The code for scheduling multiple configurations for an experiment in pynn.brainscales can for example look like this:

Example 3:
In this example, a population `input_pop` of one neuron sends spikes at certain rates to another population `pop` of one single neuron, that switches the recording

of its spikes on and off during the experiment. In this instance, the experiment is split into realtime columns with an equally long time duration for simplicity, where the number of spikes sent out from `input_pop` is increased between the different realtime columns.

```
pynn.setup(enable_neuron_bypass=True)

runtime = 10  # ms, runtime of each realtime column / config
pop = pynn.Population(1, pynn.cells.HXNeuron())

# initial config (spike recording on, 1000 spikes):
n_spikes = 1000
pop.record('spikes')
# Inject spikes
spikes_1 = np.linspace(0, runtime, n_spikes)
input_pop = pynn.Population(1, pynn.cells.SpikeSourceArray(
    spike_times=spikes_1))
pynn.Projection(input_pop, pop, pynn.AllToAllConnector(),
                synapse_type=StaticSynapse(weight=63))

pynn.add(runtime)

# second config (spike recording off, 2000 spikes):
n_spikes = 2000
pop.record(None)
# Inject spikes
spikes_2 = np.linspace(0, runtime, n_spikes)
input_pop.set(spike_times=spikes_2)

pynn.add(runtime)

# third config (spike recording on, 3000 spikes)
n_spikes = 3000
pop.record('spikes')
# Inject spikes
spikes_3 = np.linspace(0, runtime, n_spikes)
input_pop.set(spike_times=spikes_3)

pynn.run(runtime)
```

One can see in the code, that these minor changes in the experiment configuration can be done using little code and stating only the concrete parameter changes, without having to initialize `input_pop` and especially `pop` and the projection between those two over and over again after each call of `add()`.

3.3.2 Changes in experiment evaluation

Since the measured data of each realtime column is stored in an own IO-data-map, the data of each realtime column can be managed separately. The output data can be divided into different parts, that need to be handled differently:

- spike recordings
- MADC recordings
- PPU observables

The spike recordings and the MADC recordings are both stored inside a `recordings` object. These objects are already created in the process of constructing the experiment, when calling `preprocess()` and are filled with all information about the experiment configuration. Thus, all measured data can be identified with the according hardware instance, where it was measured at, e.g. which spikes correspond to which neuron. The `recordings` objects are updated in the `preprocess()` function and appended to a list in the `add()` function, so there is already a list of `recordings` objects with the right length, to inject the spike recordings and the MADC recordings of each realtime column into the according object.
Regarding the PPU observables, no changes were made to the API, as they can only contain data when using the PPU, or to be more precise, a plasticity rule. In other words, they only can contain data if there is an experiment with only one realtime column, and then the data can be handled in the same way as in the past.
In PyNN, all measurement data that was recorded at hardware components, that are associated with a certain population, can be retrieved by calling the function `get_data()` on this according population, as shown in ex. 4. This returns all recorded data of this population, or only specific observables, if specified in the function argument. The entirety of the data of a population can be divided into segments, each of which contains all data of one hardware run. These segments are filled with the according experiment data in the function `get_current_segment`, which is called for each segment inside of `get_data()`. There, all data is either associated with the category “spiketrains” [5], which only includes recorded spikes, or the category “irregularly

sampled signals" [5], which includes all other measured parameters.
Iterating over all `recordings` objects, i.e. all realtime columns, the recorded spikes, which are nothing else than times, where a spike was recorded at a specific neuron, are filled for each neuron inside the population into different `neo.SpikeTrain` [6] objects, which contain not only the times of the individual spikes, but also the beginning and end times of the spiketrain and other information like e.g. the neuron id. In this way, a `neo.SpikeTrain` object is created for each neuron and each realtime snippet and collected in a list of spiketrains. How this list of spiketrains can be retrieved is also shown in ex. 4.
For the irregularly sampled signals, it works almost the same. There is also a list of `neo.IrregularlySampledSignal` [6] objects created, but the list of irregularly sampled signals is only appended by an object, if there was data measured in the according snippets. The list of spiketrains however can also contain empty spiketrains for realtime columns, where the spike recording was disabled intermittently.

Example 4:
Let `pop` be the population from ex. 3, which contains one single neuron, for which spikes were recorded in an experiment with three different realtime columns. However, the spikes were only recorded for this neuron in the first and third realtime column, but not in the second one. Thus, the second spiketrain, which belongs to the second realtime snippet, is empty, i.e. doesn't contain any spikes. The following code snippet shows the data retrieval after the already happened experiment execution:

```
spiketrains = pop.get_data().segments[0].spiketrains
assert(len(spiketrains) = 3) # one spiketrain per realtime column
assert(len(spiketrains[0] > 0)) # spikes were recorded
assert(len(spiketrains[1] = 0)) # empty spiketrain
assert(len(spiketrains[2] > 0)) # spikes were recorded
```

For many applications however, it is more practical to receive one singular spiketrain per neuron, instead of one per realtime column per neuron. But this can be easily achieved by concatenating all spiketrains, which contain the same neuron id as an identifier. merging these different data snippets together is a lot more convenient and quicker, than tearing apart a singular large spiketrain, which contains all spikes of one neuron, in case one wants to examine these parts separately. Thus, it was decided

for the more modular and transparent way of returning the data by gaining access to the data of the individual realtime columns separately. An example for merging the different spiketrains together into one, can be seen in the demo experiment presented in sec. 4.1.

4 Evaluation

In this chapter, two sample applications are presented. The first one is a simple demonstration of the changes in the API, which doesn't focus on gaining new scientific knowledge modelling-wise, but rather on the new possibilities in the operation of pynn.brainscales. The second experiment is an adaptation of an already existing experiment, exploring the neuron dynamics of the neuron model implemented on the BSS-2 chip. The experiment is remade by using the newly implemented features in such a way, that the usage of complex low-level routines can be renounced on.
At last, a performance test of the compiling process of an experiment is examined, which experienced fundamental changes in the course of this thesis, cf. sec. 3.2. The goal of this is to get a feeling for the wall clock time spent on compiling in relation to the actual experiment runtime.

4.1 Demo experiment

The goal of the sample experiment, which is presented in the following, is to demonstrate a use case for the multi-configuration feature. It also can be used as a tutorial on how to operate the newly integrated `add()` function. This tutorial is in a way designed, that it can be used as an ipython notebook [15], so users can try out the new feature by themself. The idea of the experiment is quite simple and not of any scientific use, but showcases both the new way of describing the experiment and processing the experiment data.
A black and white image of 64 by 64 pixels, which is shown in fig. 4.1 is taken as input.

The goal is, to feed signals with the pixel values of this image through the synapse array and record the according values with the neurons to recreate the image in the end. This is done, by sending spikes at a certain rate to 64 different synapses, which in turn have set their synaptic weights according to the pixel values of a certain column of the pixel array of the picture. Then, 64 different neurons each recording spikes coming from a different synapse will have their spikes recorded, which of course only will occur, if their according synapse has set the synaptic weight to a high value, corresponding to a white pixel in the original picture.

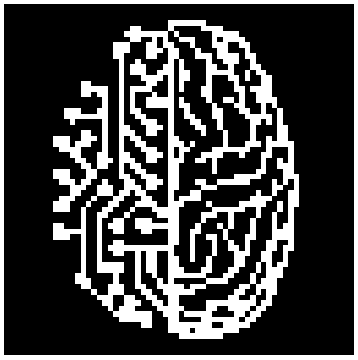


Figure 4.1: A black and white image of the logo of the ElectronicVision(s) Group, which is used as input for the experiment.

The following function is used to get the according values for the synaptic weights out of the pixel array:

```
from os.path import join
    import numpy as np
    import matplotlib.pyplot as plt
    import pynn_brainscales.brainscales2 as pynn

def read_image(path: str) -> np.array:
    image = np.asarray(plt.imread(path))
    # Scale to weight range [0, 63]
    image = image / image.max() * hal.SynapseWeightQuad.Value.max
    return np.flipud(image).T # has to be adjusted to account for weird
                              # choice of origin in np.asarray() and to
                              # allow for taking the value row-wise
                              # instead of column wise
```

After that, we apply this function on our picture to get a numpy array with the according synaptic weights. The only two values `image` contains, are the lowest and highest possible values 0 resp. 63, as we used a black and white picture as an input.

```
image = read_image(join("_static", "tutorial", "visions.png"))
```

Then, we define our experiment with a population `input_population` of one neuron, that constantly outputs spikes at a rate of 10kHz, which are transferred to a population `recording_population` of 64 neurons, that have spike recording enabled.
In the construction of the `pynn.simulator` object with `pynn.setup()`, we pass as an

argument, that neuron bypass shall be enabled, which means in other words, that a neuron will automatically spike on each pre-synaptic spike, if the synaptic weight is greater than zero:

```
pynn.setup(enable_neuron_bypass=True)

runtime = 10  # runtime per configuration in ms
n_spikes = 100
spikes = np.linspace(0, runtime, n_spikes)

input_population = pynn.Population(1, pynn.cells.\
                                   SpikeSourceArray(spike_times = spikes))
recording_population = pynn.Population(64, pynn.cells.HXNeuron())
recording_population.record('spikes')

synapse = pynn.standardmodels.synapses.StaticSynapse(weight=32)
projection = pynn.Projection(input_population,
                recording_population,
                pynn.AllToAllConnector(),
                receptor_type="excitatory",
                synapse_type=synapse)
```

What has to be done now, is to add 64 short sequences to the experiment, in each of which, the synaptic weights of our projection are configured differently, according to the current row `image[i]` of our `image` array. To trigger the hardware execution, `run()` has to be called instead of `add()` in the last iteration.

```
for i in range(64):
    projection.set(weight=image[i])
    if i < 63:
        # add() will let this configuration run for a duration of
        "runtime" when executing
        pynn.add(runtime)
    else:
        # run() calls add() a last time and performs hardware run
        pynn.run(runtime)
```

The only thing now left to do is to get the returned data into according shape. In this case, we want to have one singular spiketrain per neuron, that contains all spike

times out of the overall experiment. Thus, the individual spiketrains of the different realtime columns have to be concatenated in time for each neuron, as already described in sec. 3.3.2. This is because our `spiketrains` list contains a spiketrain per realtime column per neuron. The concatenation is done by initializing a list of 64 empty lists, into each of which all spiketrains of the neuron with the according index are appended to.
Because all the spiketrains already come correctly ordered by the temporal order of the realtime columns, the spiketrains don't need to be reordered.

```
#read out results
spiketrains =
    recording_population.get_data('spikes').segments[0].spiketrains
spiketrains_concatenated = [ [] for _ in range(64) ]
for spiketrain in spiketrains:
    spiketrains_concatenated[spiketrain.annotations["source_id"]-1].\
        extend(spiketrain.times)
```

With `plt.eventplot` the concatenated spiketrain of each neuron can be plotted in a way, that a red line is inserted at every spike time. The result is, that the plot is colored in places, where the neuron received spikes. The spiketrains of the different neurons are plotted above each other, so that each neuron represents a line in the plot:

```
#plot results
fig = plt.gcf()
fig.set_size_inches(4, 4)

plt.eventplot(spiketrains_concatenated, color='#990000')
plt.xlim(0,640)
plt.ylim(0,63)
plt.xlabel("time [ms]")
plt.ylabel("neuron index")
fig.show()
```

This gives us the desired result, as shown in fig. 4.2.

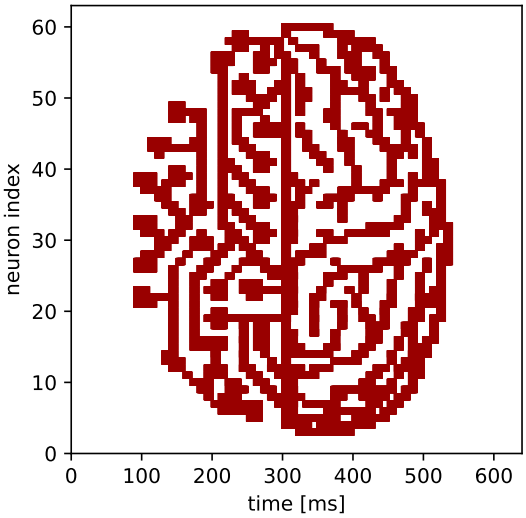


Figure 4.2: Plot of the spike times of the different neurons. For each time, a certain neuron triggered a spike, the plot shows a thin red line at the according spike time and the according neuron index. The experiment time, corresponding to the different realtime columns, is plotted along the horizontal axis and the neuron indices along the vertical axis. This results in the recreation of the logo of the ElectronicVision(s) Group.

4.2 Adaptation of AdEx-neuron experiment

As one of the last things done in the course of this bachelor thesis, the code of the experiment in the already existent tutorial on the AdEx dynamics of the neurons on the chip[1] [2] is simplified by using the reconfiguration feature.
In the tutorial, the AdEx model, which is also briefly explained in sec. 2.1.1 is presented in form of the model equations, a schematic of the circuit of an AdEx neuron and an interactive experiment. In the tutorial, the membrane trace and the adaption state are shown in dependency of several parameters, that can manually be adjusted with sliding bars. For each change in the parameter configuration, the experiment is executed on hardware and returns the newly measured data.

A fundamental part of this experiment is the so-called current stimulus, which is described as I_{stim} in eq. (2.1), which has to be applied in a certain interval of the experiment. This is done, by activating an on-chip current source 100 µs after the

[1] https://github.com/electronicvisions/brainscales2-demos/blob/master/ts_08-adex_complex_dynamics.rst

beginning of the experiment and deactivating it 600 µs later again.

Without the multi-configuration feature, this is no easy task, as it isn't possible to

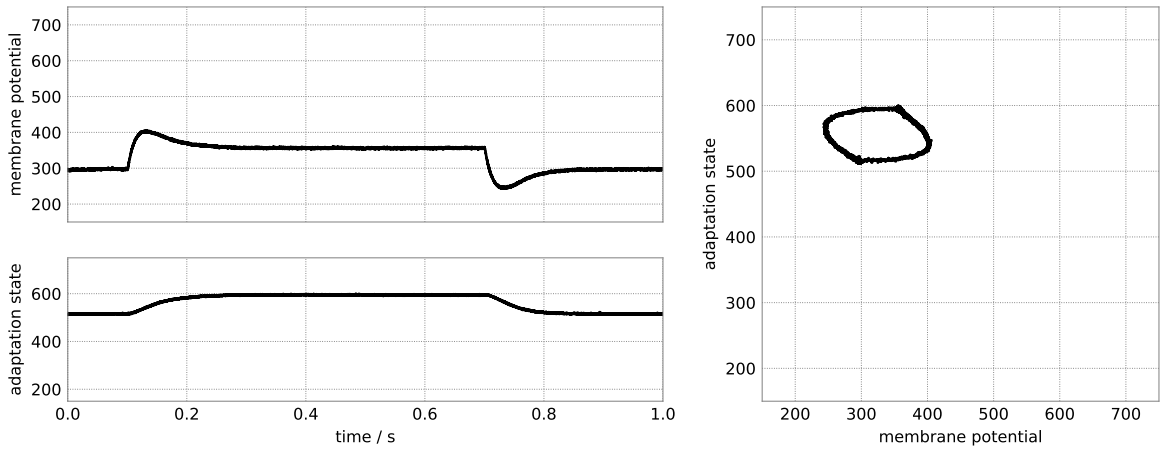


Figure 4.3: Plots taken from the AdExTurorial [2], which show the membrane potential and the adaption state in dependency of the experiment time for a certain set of parameters. One can clearly see the influence of the current stimulus, as the membrane potential quickly increases at 0.1 s in biological time, which corresponds to 100 µs in emulation time, where the current is enabled. At 0.7 s biological time, i.e. 600 µs later, the potential drops again and relaxes to the resting potential. A similar influence of the current stimulus can be recognized in the plot of the adaption state.

change this on-chip current with the pynn.brainscales API without it. So the only way to still make it work was to manually construct a PPB that switches this current on and off, which can be injected as a playback hook into the experiment. However, this is fairly complicated to pull off for a standard PyNN user, as this requires expert knowledge in many different areas of the hardware abstraction layer of the API. With the new reconfiguration feature, this can be done easily in a few lines of code, by just switching this on-chip current by setting the parameter `constant_current_enable` of the according population to `true` or `false` in between some `add()` calls with the according runtime, like so:

```
pop[0:1].set(constant_current_enable=False)
pynn.add(0.1) # deactivated in the first 0.1ms
pop[0:1].set(constant_current_enable=True)
pynn.add(0.6) # activated for 0.6ms
pop[0:1].set(constant_current_enable=False)
pynn.run(0.3) # deactivated in the last 0.3ms
```

This method only uses elements of PyNN and is also a bit shorter in the implementation.

4.3 Build performance

At last, the performance of the experiment compilation is to be examined in dependency of the size of the experiment to get a measure of how much wall-clock time it takes to build the experiments with the new way of generating the playback programs. The performance test, which triggers the execution and thus the compilation of the experiment, is written with pynn.brainscales. It features a sweep over different numbers of realtime columns, as well as different amounts of commands inside each realtime snippet, in this case different numbers of spikes. The code for the experiment is explained in fig. 4.4.

Each experiment consists of only one single batch entry, as PyNN doesn't support the use of multiple batch entries in one experiment. For simplicity reasons, no changes were made to the experiment configuration between the different realtime columns. Thus, each realtime column has the same configuration, the same input data and the same network graph as base products. This is sufficient to test the performance of the API, as the individual portions of the experiment corresponding to different realtime columns, are still generated and assembled individually. The only difference to a case with varying configurations for each realtime column is, that no differential write has to be applied for each reconfiguration. For each of the scheduled experiments, the times for different parts of the compilation process are measured:

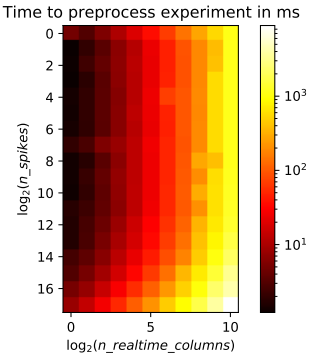
- The time required for preprocessing the provided objects. This includes the finalization of the configuration and checking, which hardware components responsible for recordings and readouts have to be enabled for which realtime columns in the experiment, depending on the signal-flow graphs.
- The time required for generating the individual realtime snippets in dependency of the according signal-flow graphs and input data.
- The time required for assembling the different pieces of the program together to one large executable on the hardware.
- The total time needed for the entire compile process of the experiment

The timer used for measuring the time durations is the `hate::Timer`[2], an API-internal tool for measuring times, which in turn uses `gettimeofday` of the Linux operating system[3] as a timer.

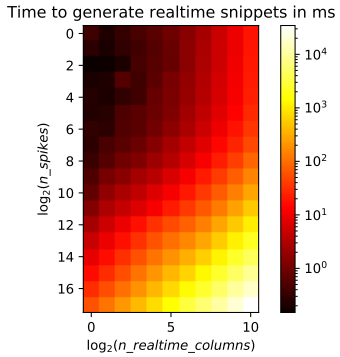
[2] https://github.com/electronicvisions/hate/blob/master/include/hate/timer.h
[3] https://linux.die.net/man/2/gettimeofday

```
import numpy as np
import pynn_brainscales.brainscales2 as pynn

max_power_add = 10  # maximal 1024 realtime columns
max_power_spikes = 17  # maximal 131072 spikes pre realtime column

pynn.setup(enable_neuron_bypass=True)

input_population = pynn.Population(1, pynn.cells.SpikeSourceArray())
recording_population = pynn.Population(1, pynn.cells.HXNeuron())
synapse = pynn.standardmodels.synapses.StaticSynapse(weight=32)
projection = pynn.Projection(input_population,
                             recording_population,
                             pynn.OneToOneConnector(),
                             synapse_type=synapse)

for power_spikes in range(max_power_spikes + 1):
    number_spikes = pow(2, power_spikes)
    runtime = number_spikes*0.01 # Choose a spike rate of 100kHz
    spikes = np.linspace(0, runtime, number_spikes)
    input_population.set(spike_times=spikes)
    for power_add in range(max_power_add + 1):
        number_add = pow(2, power_add)
        for i in range(number_add-1):
            pynn.add(runtime)
        pynn.run(runtime)
        pynn.reset()
```

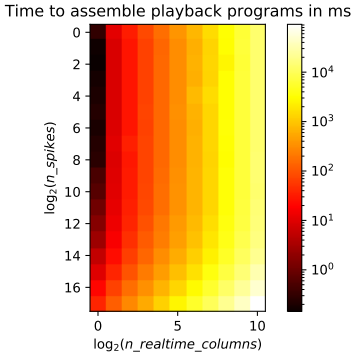
Figure 4.4: Performance test, written in PyNN:
The network consists of a population of one single neuron sending spikes at a rate of 100 kHz, which corresponds to a spike rate of 100 Hz in biological terms, to another population containing only a single neuron, which has spike recording enabled. Then, the number of realtime columns and spikes per realtime column is iteratively increased by factors of two, starting from one. The limit for the number of realtime columns or `add()` calls in this case is set to 2^{10} and the maximal number of spikes per realtime column is set to 2^{17}. As a fix spike rate of 100 kHz is chosen, the runtime is proportionally adapted to the number of spikes in each iteration. To call the `add()` function as many times, as actually intended by `number_add`, the one final call of `add()` inside the `run()` function has to be included. Thus, there's one explicit `add()` call less than `number_add`.



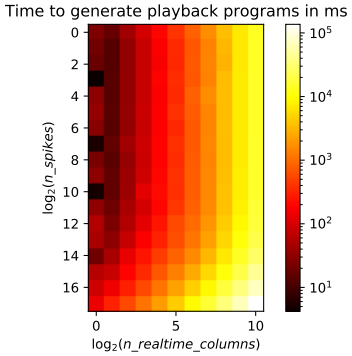
(a) Time for preprocessing the experiment



(b) Time for generating all realtime snippets



(c) Time for assembling the playback programs



(d) Time needed for the entire compilation

Figure 4.5: Wall-clock time required for the different compiling steps of the experiment in ms using an AMD EPYC 7543 as a processor. n_spikes is the number of spikes per realtime column and n_realtime_columns the number of the realtime columns in the experiment.

The results are shown in fig. 4.5. What can be seen in any of these four figures, is that all compiling processes scale uniformly for an increasing number of realtime columns. One can see from the concrete values (see app. A), that the duration approximately doubles, for each doubling step of the number of realtime columns for any of the measurements, which can be seen especially well for higher numbers of realtime columns. This behaviour is also independent of the number of spikes per realtime column.
This seems pretty logical, as all realtime columns are exactly the same, so scheduling more of them just multiplies the compiling effort, as the preprocessing, the generation of the realtime columns, as well as the assembly are all done realtime column by realtime column. Due to this, a double in the number of `add()` calls just doubles the number of which the code of the compiling process is run and thus the total time duration for any of the subparts of the compiling process.

The number of spikes per realtime column doesn't seem to have a great effect for small numbers, but the time it takes to generate the realtime columns seems to start scaling at roughly 2^6 spikes in some dependency of the number of spikes per realtime column. Looking at the numerical data, the measured durations start to approximately double beginning roughly at 2^{10} spikes per realtime snippet, for each doubling step of the number of spikes. However, in the first five doubling steps of the spike number, the duration seems to stay roughly constant. Between these areas of constant and linear scaling, there seems to be some continuous transition.
The linear scaling in the number of spike times arises from the rising number of `write()` statements made on each ATPPB, when generating the realtime snippets. However, the process involves also other commands like enabling the event recording for the spikes and checks for almost every chip component, whether it is used or not, which also takes time. This is dominant over the time to add the spikes to the ATPPB command queue for low amounts of spikes, but for higher amounts of spikes, the effort for adding the spikes is dominating, which is reflected in the behaviour, which can be observed above in fig. 4.5 (b).
The plot in fig. 4.5 (a) of the times needed for preprocessing shows a larger area of a constant time duration in dependency of the number of spikes, but also has a linear component for large numbers. In principle, the preprocessing of the experiment is independent of the input data and thus should be independent of the number of spikes. But at one place in this procedure, a copy of the input data is made, which explains the linear scaling for very large numbers of spikes, as this copy operation gets increasingly more expensive.

A similar behaviour can be recognized in fig. 4.5 (c) for the time needed for the assembly of the experiment. It also scales linearly for a great number of spikes, which is due to the effort of shifting the many commands of the individual ATPPBs in time and having to sort larger vectors of commands in the end, when the final ATPPB is converted into a PPB. This effort is only dominant for larger number of spikes and is surpassed by the only effort of this process, that is not dependent from the number of spikes, which is the differential write of the configuration. The default configuration is in fact a quite large container translating to over one hundred thousand FPGA instructions, half of which have to be compared in the differential write against the according instructions of the other configuration.

One can also see from the plots, that the assembly of the playback programs is the most time-consuming procedure. This is due to the fact, that so many ATPPBs have to be shifted in time, which happens command by command, as explained in sec. 3.1.4. The most time-consuming process is the sorting of the commands in the finally assembled, large ATPPB, when converting it to an PPB.
Because the total time duration of the compilation is just the sum of the other three sub-processes and the computing effort of the sub-process of assembling the experiment is dominant, the total compile duration shows a similar behaviour as the duration for assembling the playback programs.

As we don't have multiple batch entries, the experiment consists, apart from the initial configuration, only of one singular playback program. The limit, to which size a playback program fits entirely into the command buffer of the FPGA is 2^{22} instructions, which corresponds to a total number of 2^{21} spikes per realtime row and in this case also of the total experiment due to the fact, that everything is put into one single playback program. Even an overrun by a factor of two of the size of the instruction memory FPGA would for most experiments already result in failure, as the rest of the playback program most of the time cannot be streamed quickly enough to the FPGA and many commands would miss their timing[4]. Thus, the time durations of the compilation measured for a total amount of spikes greater than 2^{21} are likely to never be encountered, when using pynn.brainscales in a serious way - at least for the current state of the hardware.
The longest total compile times measured for a total number of spikes of 2^{21} or lower is roughly 16.6 seconds, which is the case for 2^{10} realtime columns, which was the maximum in our test. For our chosen spike rate of 100 kHz, this corresponds to an

[4]The data link between the host and the FPGA can transfer data at a rate of 1 Gbit/s, but the FPGA works off its instructions with a corresponding rate of up to 8 Gbit/s.

experiment duration of roughly 21 seconds. So in case of many reconfigurations per batch entry, a user has to account for some additional time to the actual experiment duration, for compiling the experiment. In case of actually reconfiguring the experiment in between the different `add()` calls, the compiling process would take even a little bit longer. But the only thing, that would take longer is the calculation of the differential config, the computing effort of which is negligible against most other operations inside the compiling process.
For the compilation of experiments with fewer reconfigurations, which are more common, the duration to compile the experiment takes up less and less wall-clock time compared to the total experiment runtime.
Based on the above observed linear behaviour for a large number of total spikes and realtime columns, an estimate about the expected compile time per spike and per realtime column can be drawn from the measurement data (cf. fig. A.4: The time taken to compile the experiment, i.e. to produce the playback programs takes 0.90 µs per spike in the whole experiment and 10.7 ms per realtime column. For the estimate of the time per spike, only the time differences according to a differing spike count per realtime snippet were regarded. For the estimate of the time per realtime column, the time differences between measurements with the same total number of spikes, but a different amount of realtime columns, were taken. For sufficient precision, the measurements featuring the six greatest differences in the total number of spikes were looked at.

5 Conclusion and outlook

In this thesis, the application programming interface (API) of the BrainScaleS-2 (BSS-2) platform [14, 10] was extended by the possibility to redefine an experiment's configuration at any point in time. Now, any experiment can abstractly be described as a system evolving continuously in time, where at discrete points in time certain changes are applied to the configuration.
To make this feature available in the user frontend pynn.brainscales [10], new concepts in command timing and experiment assembly were developed, cf. sec. 2.2, resp. sec. 2.3. To implement these concepts into the software stack [10], changes and additions were made on several different abstraction layers:
At first, a new kind of playback program builder, which is used to schedule commands for the system at the according times, was developed. In comparison to the old PlaybackProgramBuilder (PPB), this new builder handles only absolute times, i.e. a certain time after the start of an experiment, at which a command is to be executed. Thus, the name AbsoluteTimePlaybackProgramBuilder (ATPPB). Another difference between the PPB and the ATPPB is, that the latter one links each command directly to the time at which it shall be scheduled (cf. sec. 3.1.1), which makes it easy, to merge command queues of different ATPPBs into each other (cf. sec. 3.1.4), instead of only being able to concatenate them which is the case for the PPB.
This feature is the base for the implementation of the changes in the experiment procedure, which require different pieces of the experiment, i.e. different ATPPB objects that partly have to be executed in the same time intervals, being merged together into one single ATPPB, cf. sec. 3.2. This has to be done to retain an absolute time base over the entire course of a batch entry, which ensures an immediate transition between sections of the experiment with different configurations, the so-called realtime snippets, cf. sec. 2.2. Handling an experiment as different pieces, that include a reconfiguration as a first step and then a evolution of the system for a certain time duration, opens up a new dimension in the experiment procedure, cf. fig. 2.5. To account for this additional dimension, many objects in the experiment description layer of the software stack and upwards were sequentialized.
This includes some objects in the user frontend pynn.brainscales, in which this multi-configuration feature is available now. To integrate this feature well into the PyNN [5]

environment, the new function `pynn.add()` was introduced, with which one can extend the currently defined experiment by a section of a given runtime with the current configuration. This additional dimension was also made visible in the readout of observable data (cf. sec. 3.3.2) to grant the user maximal modularity and transparency.

The new multi-configuration feature makes the description of experiments with a dynamically changing configuration a lot easier. This is shown by the example of the experiment in the interactive adaptive exponential integrate-and-fire (AdEx) demo [2], presented in sec. 4.2, which is one of many experiment profiting from this new feature. At the same time, all formerly valid experiment descriptions are still supported and function the same as before. This challenge of finding the right balance between innovation and the support of former concepts was also present in the implementation of the new experiment procedure, which now distinguishes between the case of a singular initial configuration and an experiment with multiple configurations. In the former case, playback hooks and the usage of the plasticity processing unit (PPU) are supported, to let the API lose no other features in the process of implementing a new one.
Finally, the performance of the compile procedure was examined, which experienced changes due to the new way of constructing an experiment. The observed qualitative behaviour of the time it took for different sub-processes of the experiment compilation met the expectations, cf. sec. 4.3. Looking at the actual numbers, total time for the compilation never exceeds 80% of the actual runtime of an experiment, which is still bearable, as these processes still happen on a scale of seconds. However, for experiments, that aren't reconfigured hundreds and thousands of times, the compile duration in relation to the experiment runtime is fairly lower.

Looking ahead, one of the greatest improvements, that can be made on the multi-configuration feature is the support of the PPU in experiments with multiple different configuration states, which is currently not possible, as explained in sec. 3.2. This however is no problem that can be solved entirely in software, but also requires the hardware to support real time streaming of data from the PPU to the field-programmable gate array (FPGA), so that no longer breaks with unknown duration would occur during two different realtime snippets, which currently would ruin the absolute time base and delay successive realtime snippets too much. This however requires memory management on the FPGA for the data received from the PPU.

Another thing, which builds on the multi-configuration feature, would be an event-

driven machine learning API that uses this feature for batch support in large hyperparameter sweeps, as an instant reconfiguration would save lots of time by just writing the differential configuration, compared to executing individual experiments with different configurations after another, which was the only possibility for a parameter sweep in the past.
Similar to that, analog parameter sweeps for the calibration algorithms, which could be used to get the chip calibrated according to the wishes of a user in a single hardware run, can be implemented on a high level with the multi-configuration feature. Some other appliances, that are helpful for experiment description, are the reconfiguration of background sources, e.g. rate shaping or dynamic changes in the amplitude of the external current to a neuron.

6 References

[1] Arindam Basu, Lei Deng, Charlotte Frenkel, and Xueyong Zhang. "Spiking Neural Network Integrated Circuits: A Review of Trends and Future Directions". In: *2022 IEEE Custom Integrated Circuits Conference (CICC)*. 2022, pp. 1–8. DOI: 10.1109/CICC53496.2022.9772783.

[2] Sebastian Billaudelle. *Complex neuron dynamics with a silicon adaptive exponential integrate-and-fire neuron*. https://electronicvisions.github.io/documentation-brainscales2/latest/brainscales2-demos/ts_08-adex_complex_dynamics.html. 2023.

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

[4] Manon Dampfhoffer, Thomas Mesquida, Alexandre Valentian, and Lorena Anghel. "Are SNNs Really More Energy-Efficient Than ANNs? an In-Depth Hardware-Aware Study". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 7.3 (2023), pp. 731–741. DOI: 10.1109/TETCI.2022.3214509.

[5] Andrew Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. "PyNN: a common interface for neuronal network simulators". In: *Frontiers in Neuroinformatics* 2 (2009). ISSN: 1662-5196. DOI: 10.3389/neuro.11.011.2008.

[6] Samuel Garcia, Domenico Guarino, Florent Jaillet, Todd Jennings, Robert Pröpper, Philipp Rautenberg, Chris Rodgers, Andrey Sobolev, Thomas Wachtler, Pierre Yger, and Andrew Davison. "Neo: an object model for handling electrophysiology data in multiple formats". In: *Frontiers in Neuroinformatics* 8 (2014). ISSN: 1662-5196. DOI: 10.3389/fninf.2014.00010.

[7] Marc-Oliver Gewaltig and Markus Diesmann. "NEST (NEural Simulation Tool)". In: *Scholarpedia* 2.4 (2007), p. 1430.

[8] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific containers for mobility of compute". In: *PLOS ONE* 12.5 (2017), pp. 1–20. DOI: 10.1371/journal.pone.0177459.

[9] Christian Paul Mauch. “Operating Accelerated Neuromorphic Hardware - A Scalable and Sustainable Approach”. PhD thesis. Heidelberg University, 2021.

[10] Eric Müller, Elias Arnold, Oliver Breitwieser, Milena Czierlinski, Arne Emmel, Jakob Kaiser, Christian Mauch, Sebastian Schmitt, Philipp Spilger, Raphael Stock, Yannik Stradmann, Johannes Weis, Andreas Baumbach, Sebastian Billaudelle, Benjamin Cramer, Falk Ebert, Julian Göltz, Joscha Ilmberger, Vitali Karasenko, et al. “A Scalable Approach to Modeling on Accelerated Neuromorphic Hardware”. In: *Frontiers in Neuroscience* 16 (2022). ISSN: 1662-453X. DOI: 10.3389/fnins.2022.884128.

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

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, 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, F. d’Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019.

[14] Christian Pehle, Sebastian Billaudelle, Benjamin Cramer, Jakob Kaiser, Korbinian Schreiber, Yannik Stradmann, Johannes Weis, Aron Leibfried, Eric Müller, and Johannes Schemmel. “The BrainScaleS-2 Accelerated Neuromorphic System With Hybrid Plasticity”. In: *Frontiers in Neuroscience* 16 (2022). ISSN: 1662-453X. DOI: 10.3389/fnins.2022.795876.

[15] A. B. Raju. “IPython Notebook for Teaching and Learning”. In: *Proceedings of the International Conference on Transformations in Engineering Education*. Ed.

by R. Natarajan. New Delhi: Springer India, 2015, pp. 611–611. ISBN: 978-81-322-1931-6.

[16] Precision Reports. *Artificial Neural Networks Market Size, Growth & Statistics to 2031*. https://www.linkedin.com/pulse/artificial-neural-networks-market-size-growth-ogeef. 2023.

[17] Philipp Spilger. *From Neural Network Descriptions to Neuromorphic Hardware — A Signal-Flow Graph Compiler Approach*. 2021.

[18] Philipp Spilger, Elias Arnold, Luca Blessing, Christian Mauch, Christian Pehle, Eric Müller, and Johannes Schemmel. "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, 2023, pp. 57–62. ISBN: 9781450399470. DOI: 10.1145/3584954.3584993.

[19] Philipp Spilger, Eric Müller, Arne Emmel, Aron Leibfried, Christian Mauch, Christian Pehle, Johannes Weis, Oliver Breitwieser, Sebastian Billaudelle, Sebastian Schmitt, Timo C. Wunderlich, Yannik Stradmann, and Johannes Schemmel. "hxtorch: PyTorch for BrainScaleS-2". In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Ed. by Joao Gama, Sepideh Pashami, Albert Bifet, Moamar Sayed-Mouchawe, Holger Fröning, Franz Pernkopf, Gregor Schiele, and Michaela Blott. Cham: Springer International Publishing, 2020, pp. 189–200. ISBN: 978-3-030-66770-2.

[20] Marcel Stimberg, Romain Brette, and Dan FM Goodman. "Brian 2, an intuitive and efficient neural simulator". In: *eLife* 8 (Aug. 2019). Ed. by Frances K Skinner, e47314. ISSN: 2050-084X. DOI: 10.7554/eLife.47314.

Acknowledgements (Danksagung)

Mein größter Dank geht an Philipp für seine dauerhafte und reaktionsschnelle Hilfsbereitschaft in sämtlichen Belangen. Ob remotely per Mattermost oder persönlich neben mir an meinem Arbeitsplatz, ob zu normalen Arbeitszeiten, oder am Wochenende: Mit seiner unfassbar weitreichenden Kompetenz, die im Bereich meiner Arbeit auch bis in extremste Tiefen reichte, hat Philipp sämtliche meiner Fragen beantwortet. Nahezu alles, was ich über die BrainScaleS-2 Plattform und die Bedienung von Git, Gerrit, Jenkins und den für diese Arbeit benötigten shell-tools wie z.B. Slurm weiß, hat Philipp mir beigebracht. Er war immer meine Rettung in der Not, wenn ich kein Land mehr gesehen habe vor lauter Kompilierfehlern, oder in Phasen, in denen ich aufgrund simpler Denkfehler den Sinn meiner kompletten Arbeit angezweifelt habe. Auf Gerrit hat Philipp meine Changesets fast in Echtzeit reviewt und stand mir immer mit Rat und Tat zur Seite, wenn ich mit Jenkins erbittert um die "+1"-Verifikation gerungen habe.

Weiterhin möchte ich Philipp, Jakob, Yannik und Eric für das Korrekturlesen meiner Arbeit danken. Durch die Verbesserungsvorschläge habe ich noch einige Dinge über das Schreiben einer wissenschaftlichen Arbeit gelernt.

Ein ganz besonderer Dank geht auch an Joscha, der den Tischkicker und gemeinsam mit Dan auch die Dart-Ecke in unserem "Showroom" hegt und pflegt, damit dem mittlerweile zur Tradition gewordenen täglichen Tischkickerspiel nach dem Mittagessen und den Dart-Sessions nichts im Wege steht.
An dieser Stelle möchte ich natürlich auch noch all meinen Teamkameraden und Gegnern danken, mit denen, bzw. gegen die ich im Tischkicker und bei Darts spielen durfte; insbesondere Jakob, Tobi, Michael, Joscha, Philipp, Dan, Eric, Liam, Yannik, Sebastian, Kaspar und Ilya.

Zudem möchte ich Amani und Simon danken, die mir in meinem Büro Gesellschaft leisteten und somit das Büro zum besten Büro der Welt transformiert haben.

Ich möchte außerdem Philipp und Dan danken, dass sie in gemeinsamem Ein-

satz das EINC um Philipps alte Tischtennisplatte bereichert haben. Die regelmäßigen Tischtennis-Matches am Donnerstagnachmittag mit Philipp waren immer eine willkommene Abwechslung.

Die Zeit hier in der Electronic Vision(s) Gruppe war sehr schön, ich werde euch nie vergessen!

The work carried out in this bachelor thesis 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)

Acronyms

AdEx adaptive exponential integrate-and-fire. 12, 50, 59, 72

ANN artificial neural network. 8

API application programming interface. 8, 10, 14, 28, 30, 41, 43, 46, 51, 52, 58–60, 69

ASIC application-specific integrated circuit. 10, 11

ATPPB AbsoluteTimePlaybackProgramBuilder. 24, 27–30, 32–35, 37–39, 55, 56, 58

ATPPCT AbsoluteTimePlaybackProgramContainerTicket. 30, 32–34

ATPPCTS AbsoluteTimePlaybackProgramContainerTicketStorage. 29, 32–34

BSS-2 BrainScaleS-2. 8–12, 14, 15, 28, 40, 46, 58

CADC columnar analog-to-digital converter. 13

EINC European Institute for Neuromorphic Computing. 10, 11

FPGA field-programmable gate array. 10–14, 17, 21, 23–25, 27, 28, 30, 32, 33, 36–39, 41, 56, 59

grenade GRaph-based Experiment Notation And Data-flow Execution. 28, 35

LIF leaky integrate-and-fire. 12

MADC membrane analog-to-digital converter. 13, 43

PPB PlaybackProgramBuilder. 17, 19, 21, 23, 24, 27–29, 32–34, 36–39, 51, 56, 58

PPU plasticity processing unit. 11, 13, 24, 36, 37, 43, 59

SNN spiking neural network. 8

A Performance measurement data

These are the files with the measured performance data. The different columns resemble the different amounts of realtime columns increasing from left to right in doubling steps and the different lines correspond to different number of spikes per realtime column, increasing from top to bottom in doubling steps.
The data was recorded with the application programming interface (API) intern `hate::Timer`, which by itself uses `gettimeofday` of the Linux operating system[1] for measuring the time.

4.9140	3.1520	5.7270	10.626	19.959	37.140	71.080	144.10	285.35	564.33	1135.3
1.2150	2.1140	3.5100	6.7720	11.873	24.569	47.000	96.698	192.62	389.37	1165.9
1.2830	2.4860	3.9770	7.1360	12.723	24.100	48.259	93.545	191.77	413.12	1158.6
1.2360	1.8800	3.5410	6.6840	12.335	23.590	48.703	93.174	194.97	584.51	1160.2
1.3990	2.0610	3.6240	6.4520	12.200	25.285	72.444	100.39	195.82	584.75	1155.6
1.3800	2.1120	3.6330	6.6570	13.358	25.051	49.773	99.178	192.92	579.88	1164.0
1.4550	2.0350	3.5710	6.5870	12.163	23.820	47.424	99.316	195.37	586.55	1170.0
2.0760	2.8200	5.6790	8.0600	13.388	25.993	51.166	97.196	196.59	586.08	1168.7
1.3730	1.8390	3.5440	6.7300	13.098	25.172	51.231	99.599	295.69	394.45	1186.0
1.3900	2.2430	4.0480	7.5690	13.083	25.893	50.835	101.45	205.72	603.65	1201.7
1.3380	2.0950	3.7780	7.2400	13.581	25.580	53.884	107.39	310.76	637.02	1245.9
1.8190	2.3690	3.9490	6.8990	13.316	26.409	52.497	104.15	209.38	634.89	1281.5
1.7150	2.7120	4.4940	8.5170	16.407	31.917	61.081	116.77	351.73	699.39	1394.4
1.6710	2.8390	5.2470	10.055	18.080	35.026	68.934	138.65	409.74	836.10	1695.4
2.6720	3.7530	6.5980	12.597	23.917	45.098	89.390	180.54	469.48	1059.8	2200.5
3.1800	5.1900	9.4330	17.631	34.250	66.632	138.20	362.86	728.77	1494.0	3366.6
5.0770	7.6920	15.016	28.931	57.521	113.61	236.28	504.60	1021.0	2309.3	4293.2
8.1690	14.106	27.598	53.585	109.16	221.00	476.11	932.06	1777.5	3874.1	8965.1

Figure A.1: Time taken to preprocess the experiment in ms.

[1] https://linux.die.net/man/2/gettimeofday

0.4080	0.2270	0.3400	0.4840	0.6450	0.9580	1.5230	2.5880	5.1580	9.6730	19.547
0.2820	0.2100	0.3190	0.4170	0.6040	0.8680	1.6520	2.9370	5.4610	9.9510	19.795
0.1500	0.1700	0.2200	0.4650	0.6340	0.8510	1.6570	3.1310	5.5830	10.651	20.955
0.2040	0.2250	0.6060	0.4110	0.6980	1.1660	1.9210	3.2940	5.8800	11.225	22.434
0.2480	0.2190	0.3150	0.4030	0.8190	1.2500	2.0950	3.6030	6.5960	12.068	24.684
0.2360	0.2660	0.5210	0.6770	0.9310	1.3050	2.5260	4.3060	7.9260	14.299	28.882
0.3180	0.2930	0.5630	0.6420	1.0470	1.7300	2.8890	5.4570	10.112	18.898	37.339
0.2860	0.4820	0.6550	0.9020	1.5020	2.6230	4.4930	8.0020	14.923	26.285	53.778
0.3590	0.5970	0.9870	1.3430	2.2720	3.5840	6.6950	12.249	23.079	43.823	88.314
0.5270	1.0840	1.5690	2.6070	3.7980	6.7910	11.803	21.877	40.798	78.825	160.46
0.7500	1.8070	2.9520	4.2910	7.2500	11.953	21.337	38.661	74.736	147.00	287.15
1.4180	3.0720	4.6720	7.3520	12.530	21.813	39.081	70.182	130.97	272.76	552.95
2.6770	5.4770	9.7800	17.959	33.333	54.861	80.274	145.88	279.06	571.57	1121.2
4.9530	9.8540	17.945	34.465	62.583	107.33	169.99	302.00	577.07	1165.8	2278.0
9.0210	17.549	33.141	61.926	115.49	203.08	344.29	630.54	1222.7	2452.0	4496.1
16.514	30.663	59.327	111.60	209.95	390.85	687.75	1324.8	2560.7	4991.7	9175.5
30.792	58.463	110.86	206.56	396.26	746.09	1390.6	2672.6	5106.4	9909.5	17241.
61.817	115.30	222.73	412.15	799.61	1532.1	3235.1	5886.9	10726.	19659.	34579.

Figure A.2: Time taken to generate the realtime snippets in ms.

0.3320	11.390	34.500	82.115	175.87	371.97	758.12	1541.7	3219.3	6406.6	12807.
0.1650	9.0280	28.593	68.879	149.63	315.03	639.64	1307.3	3013.5	6550.2	13176.
0.1450	9.3750	29.757	70.435	156.48	321.12	681.78	1361.4	2657.5	6345.1	13273.
0.1830	9.4530	28.481	69.175	152.99	313.18	646.17	1295.5	3226.3	6651.8	13417.
0.2130	9.3070	28.820	69.208	150.66	313.24	776.78	1546.7	3209.3	6703.3	13225.
0.2510	9.3650	29.240	72.762	155.54	334.54	666.16	1361.3	3261.8	6607.1	13351.
0.2020	9.5230	29.101	70.401	152.08	315.36	646.07	1368.1	3323.4	6633.0	13265.
0.2500	10.834	35.078	72.903	158.62	328.10	674.57	1375.9	3244.5	6595.6	13441.
0.2810	9.7860	30.334	72.624	156.49	323.21	657.65	1337.6	3085.4	6785.7	13479.
0.5320	10.347	31.993	75.926	165.32	340.09	691.16	1409.5	3325.1	6726.2	13475.
0.6740	10.740	32.834	77.598	169.21	350.80	694.13	1382.4	3374.6	6740.5	13303.
1.1290	11.872	34.944	81.084	175.12	361.77	714.12	1415.1	3402.2	6866.3	14786.
1.6920	13.765	40.723	97.100	209.23	415.79	800.55	1583.8	3712.8	7364.9	14973.
2.8030	17.577	48.863	117.42	231.08	471.66	923.27	1868.3	4188.0	8551.2	16809.
5.7180	25.035	67.483	148.76	316.47	591.80	1176.3	2355.2	5150.6	10512.	21169.
10.451	40.738	102.61	219.44	433.11	845.50	1726.6	3650.2	7354.3	14444.	31179.
21.773	68.989	169.24	344.14	692.82	1369.9	2894.0	5683.0	11879.	22785.	48884.
40.778	125.17	292.05	583.42	1217.9	2544.1	5282.7	10213.	21117.	42113.	92856.

Figure A.3: Time taken to assemble the playback programs out of the different pieces in ms.

25.952	17.889	44.206	96.299	199.58	423.83	850.96	1691.3	3549.4	6984.0	13992.
22.077	13.622	35.866	78.646	180.60	361.84	721.25	1434.9	3232.6	7010.3	14408.
18.261	13.446	35.737	80.514	172.05	348.18	734.29	1479.9	2894.8	6797.4	14481.
4.2160	14.001	35.293	79.039	193.13	366.21	699.89	1414.5	3480.9	7314.9	14629.
33.125	13.903	35.348	78.712	182.38	361.03	854.53	1653.7	3440.3	7348.0	14452.
37.583	14.525	36.356	82.613	188.66	405.76	759.25	1486.7	3506.1	7205.2	14563.
30.003	14.188	35.348	80.387	185.12	344.21	731.91	1498.0	3576.4	7260.3	14494.
5.5060	16.966	44.431	85.044	194.40	382.52	733.82	1526.6	3478.3	7211.7	14685.
23.358	14.328	37.285	82.856	191.86	373.51	759.96	1453.0	3460.6	7253.8	14783.
32.181	16.214	39.751	89.355	203.28	408.40	800.94	1547.1	3609.8	7442.9	14842.
5.0790	16.672	41.944	126.48	220.85	391.89	796.04	1570.4	3780.8	7560.7	14885.
34.311	20.388	48.695	97.503	204.23	431.66	850.64	1633.4	3820.8	7822.2	16646.
57.844	25.004	57.914	126.66	262.01	574.39	970.35	1865.0	4373.5	8664.4	17534.
48.875	32.924	75.303	182.16	314.84	625.17	1182.2	2313.1	5205.0	10621.	20868.
20.576	49.654	136.84	255.11	485.26	885.46	1638.9	3196.1	6942.3	14079.	27896.
61.825	80.341	192.86	353.03	711.58	1358.8	2612.4	5424.4	10685.	20961.	43812.
88.766	153.28	299.19	584.00	1182.9	2286.4	4551.6	8943.5	18091.	35061.	70475.
143.66	316.27	583.60	1113.1	2173.1	4354.8	9010.7	17093.	33635.	65761.	136406

Figure A.4: Time taken to for the entire experiment compilation in ms

B Experiment environment

The demo experiment (cf. sec. 4.1), the experiment of the adaptive exponential integrate-and-fire (AdEx) tutorial (cf. sec. 4.2) and the performance test (cf. sec. 4.3) were run on the same software state, which is documented in the following. To be able to identify the exact state, the commit hashes of each repository are listed in table B.1 and additionally the numbers of the changesets on Gerrit, if the software state was ahead of the remote master branch at the time.

To track the used third-party software, the necessary data to identify the used singularity container [8] is shown in tab. B.2. This singularity container contains all shell tools, libraries and other software that is not from the Electronic Vision(s) Group itself, but was available to use for all conducted experiments in chapter 4.

repository	git hash	changeset
haldls	a887e8a678556a3dcf11a29570ab6d3c5858748d	
code-format	e718d56928d006669376fde991bb9b4605a6818a	
logger	9457eb8031dd902780f1aa695438ac2d18ed2f3f	
halco	8129637b2612b1559eb0ab441c7cb55057b0172b	
hate	e8b4bf7a7be4d99911f66b9c1681b20d28866e3c	
fisch	0d877547b5021da78b82753b1b3428c0639e1ad7	
libnux	f98f962d848c0338f1bc5a9f184c05b9f3ee31a9	
hxcomm	5f8e49f89a954c551d51ff22f8381d8b83cbc766	
rant	0d494ce6eedfb74889cf7cee09105258819acb35	
ztl	773660f435e56b1ee7b962e8babfe004ff487cdd	
pywrap	536c5fc5e102c5f9c3a8eb404ffa55ec40b3d75e	
lib-boost-patches	ed89665b4c066629b69617ede2e8b1fbe65822d9	
sctrltp	1d854f953f7e8c8ead44406a22bb80421ca3857c	
nhtl-extoll	a140dc1ae3114bda0ae08f1cd67833887fc0bd17	
hwdb	40e8627e64219a5c5e7ade49b5d738c07965aa94	
visions-slurm	8f41ea4f5bd1573d8f4623e9ed698a29f30036a3	
flange	28e729d59df3b4ff380f84351c40d4da3086bed8	
lib-rcf	af6fc768c1735b67c3bf189de7c1fca58e67c0e3	
bss-hw-params	be3c6c33fc3a85bf296be779c9b4cd5b96f4ae29	
librma	3fae4e359b8cbb2760b32046aff93a3da24a3943	
extoll-driver	fcfad13745f2434ee63834478d67b1141b62da2f	
pynn-brainscales	9faef5492844ae04b87be36722feef845642416e	22131
grenade	08c27c2ddc22bafef2babdd464bbb744df2b3812	22133
calix	9bfe424a107494cb9e2ab1a585bd4995c93241f0	
much-demos-such-wow	8e906b1008d56b4f4274f2b3128cc97aa7a43044	22115
hxtorch	071c183bda155e1b0c7bc7ea6ff526f5519692dd	

Table B.1: Software state used for the execution of both sample experiments and the performance test in chapter 4.

key	value
path	/containers/stable/2024-01-03_1.img
fingerprint	07b48f8c-ee00-4560-bb20-a6fa21895084
app	dls

Table B.2: Data of the singularity container used for all experiments, that are presented in this thesis.

Erklärung:

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

Heidelberg, den 27.02.2024