Internship report

Creating a mergeable program builder for BrainScaleS-2

Tim Auberer

October 2023

supervised by

Philipp Spilger, Eric Müller

Abstract

This internship report is about the development of the AbsoluteTimePlaybackProgram-Builder and its uses for the queuing of commands for the BrainScales-2 System. The Absolute-TimePlaybackProgramBuilder introduces a convenient way to merge command queues, which shall be run in the same section of a program. Besides the development and the use of the AbsoluteTimePlaybackProgramBuilder, also its performance is examined and evaluated in comparison to its lower level analogue, the PlaybackProgramBuilder. Also the use of this new builder is showcased on an example.

Contents

1 Introduction **1**

2 Implementation of the ATPPB **2**
2.1 Base feature 2
2.2 Additional features 2
2.3 Performance 3

3 Integration into the API **5**

4 Example of a use case **6**

5 Discussion **7**

6 Outlook **7**

1 Introduction

The command chain on the BrainScaleS-2[**4**][**5**] System consists of 3 stages: The user front end, where the user can implement their program, the actual chip on which the experiment is performed and a Field Programmable Gate Array (FPGA) which acts as an interface between the front end and the chip. The latter buffers the commands passed by the front end and is responsible for feeding the right signals at the right time into the chip and also buffers the output from the chip. So the real time communication happens only between the FPGA and the chip. The FPGA gets the whole command stack in advance from the user front end, as far as the command stack can fit entirely into the memory of the FPGA.
When pipelining the commands as a user, one needs to pass a time stamp with the actual wanted action to perform, which can be specified by a coordinate (a information on where on the chip your signal is going to be rooted to) and a container (a type of signal).
On a low level, this is possible by using the PlaybackProgramBuilder[**6**] (PPB) and alternating functional commands with `block_until` instructions, which block the execution of the next command, until the FPGA timer reaches the given value. So one can set the time of execution of a command relatively to the time, where the timer is initialized, i.e. the timer on the FPGA starts to count the clock cycles. But managing these relative times as a user can be quite complex, because oftentimes there are heterogenous operations wanted, which are distributed randomly in time.
There were already tools for auto-generating such random spike trains, but if one used these e.g. in the so called inside real time section of their program, one could only perform other actions, like reading measurement values or configuring chip properties in the pre- or post real time sections, but not during the experiment. Because if one has two command pipelines, where each command only has a relative time stamp to the first command of the pipeline, one doesn't have to possibility to merge them, because there is no absolute time order.
To solve primarily this issue, the AbsoluteTimePlaybackProgramBuilder (ATPPB) was developed and integrated into the existing API in the course of my internship. Now one can merge independent command queues in form of ATPPBs into a single ATPPB and also inject their own ATPPB into a inside real time section even while auto-generating spike trains with grenade[**7**], that will be executed in the same time section of the experiment.

2 Implementation of the ATPPB

2.1 Base feature

The AbsoluteTimePlaybackProgramBuilder is based on the PlaybackProgramBuilder, i.e. its output is a PPB. In short terms, an ATPPB is an object that holds a sequence of commands. In this sense a command is a structure, which holds three parameters: absolute execution time in units of a FPGA clock cycle, the coordinate and the container. The latter two of these specify which action the system needs to perform. One can insert a command into the command vector using the `write()` function. As the name of this function might suggest, it only supports "write-type" -commands, which are commands that influence the action on the chip, e.g. setting configurations or generating spikes. "Read-type" -commands must be handled differently, because they return a value, which has to be passed to the original source, which initiated the command being in this final job. This is no easy task, as the wanted return values are only determined after the completion of the measurements on the chip. But when commanding a measurement, some kind of immediate return value is required, as at some point, you need to identify the returning measurements from the chip with their according variables from the program code, that they're assigned to. Thus, the possibility to pass "read-type" -commands to the ATPPB is not yet implemented.
The commands in the vector are stored in the same order, as they were written to it, so at some point they need to be ordered in time, in order to finally get a PPB. Obviously, the best time to sort the commands is in the end, when the PlaybackProgramBuilder is required and no more commands are to be added to the list, because else, when storing a sorted list of commands, the list always needs to be sorted again, when adding another command or merging with another command list. Thus, the sorting happens in the `done()` function, which orders the command vector first and then assembles a PlaybackProgramBuilder. It does that by looping over the command vector and adding a command with the same coordinate and container to the PPB for each element in the command vector, after instructing the FPGA timer to wait until it has reached a value of one clock cycle before the wanted time of execution. If multiple commands in the command vector have got the same time stamp, they are scheduled in the same order, as they were added to the command vector. However each of their execution is delayed by the time, that the previously executed commands with the same time stamp needed for their execution.
The outgoing PlaybackProgramBuilder can be further processed in the same way as usual, to get the job running.

2.2 Additional features

The AbsoluteTimePlaybackProgramBuilder can't only produce a PlaybackProgramBuilder out of its list of commands, but it has to offer some more features, if the user wants to set up his jobs in a modular way. The `merge()` function can merge another ATPPB into the calling ATPPB, i.e. appends the command vector of the other ATPPB to the command vector of the calling ATPPB and deletes the command vector of the other ATPPB. If one wants to perform this action, but don't lose the other builder, they can call the `copy()` function, which appends the command vector of the calling ATPPB with a copy of the other builder's command vector.
These two features are the main advantage of the ATPPB over the PlaybackProgramBuilder. Because all commands of the command vector of any AbsoluteTimePlaybackProgramBuilder reference the same point in time as 0, there is a meaningful order of commands between different Builders. This gives the user not only the advantage to enter the commands in an arbitrary order, but also lets them assemble components of their job with other automatic tools like for example the spike-generator of grenade.

Aside from that, there are some minor convenience features, like an insertion operator for printing the contents of the command vector, so that one can quickly see which commands are already scheduled in a Builder. In contrast to the insertion operator of its lower layer analogue, the PlaybackProgramBuilder, it prints out the containers and the coordinates, not the FPGA hardware commands, which makes reading the output easier for the user. Another small feature is, that one can check with the `empty()` function, if there's anything scheduled yet at all.

2.3 Performance

Because the AbsoluteTimePlaybackProgramBuilder essentially doesn't introduce new possibilities or functionalities into the API (one could technically still build anything with a regular PPB), it mustn't have a too large overhead to call this exchange of performance against comfort feasible. Because the ATPPB practically uses the PPB, it is of course slower to construct a job with the ATPPB, than with the PPB. In order to minimize these performance drops, they were analyzed as shown in the following paragraphs.
The greatest performance issue is of course the sorting of the command vector elements by time. In general, these command vectors can become big. For example if one wants to schedule spike-trains, the command vector will often have in the order of 10^6 entries. Thus we need a sorting algorithm, the effort of which grows slowly for large numbers of elements to be sorted.
As mentioned above, it can be, that the user tags multiple commands with the same point in time. This can be useful to guarantee the quickest possible execution of these commands. But still, the user might want to keep track of the order, in which the actions are going to be performed on the chip, so it makes sense to choose a sorting algorithm, which is stable, i.e. keeps the order of all commands, which are scheduled for the same time.
The standard library offers a sorting algorithm `std::stable_sort` which uses Merge Sort and thus has a time complexity of $n \log n$, if enough memory space is available[**1**].
To get a good performance comparison between building a job with the AbsoluteTimePlaybackProgramBuilder and the PlaybackProgramBuilder, I measured the time duration for constructing a PPB, which can produce following program: Write N spikes, enable event recording, write N spikes, disable event recording, write N spikes again. I made multiple measurements, shifting N from 1 to 10^6 in steps of an order of magnitude and for each value of N I made 10 measurements for statistical stability.

For the construction of the PlaybackProgramBuilder, one expects the computing effort to be linear in the amount of written Spikes with an offset for switching the event recording on and off. When working with the AbsoluteTimePlaybackProgramBuilder, one would expect roughly the same, but with an additional effort for sorting the command vector ($\sim N \log N$) and decoding the ATPPB commands into PPB commands ($\sim N$).
According to that, the fit-function in the plot of the ATPPB is:

$$f_{ATPPB} = a \cdot N \log N + b \cdot N + c$$

and the fit-function for the PlaybackProgramBuilder is of the shape

$$f_{PPB} = b' \cdot N + c'.$$

For the comparison in figure 1 I intendedly chose to compare the build time for an ATPPB, where the user enters the commands already in the correct temporal order, because that's the same thing

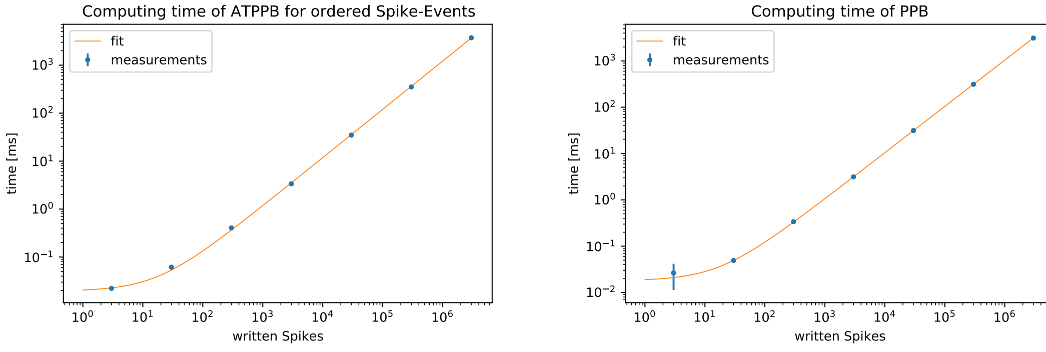

Figure 1: Side-by-side comparison of the computing times for our sample task.

they would do, when building their job with an PPB, so this is the most reasonable comparison. Also, all outputs from automatic program generators will add their commands in the correct order to the builder queue, so this is the most common occurance of an ATPPB.
In order to specifically test the bottle-neck, that emerges through sorting the command vector, I did two other tests, where I entered the commands once in random order, which is close to worst case for Merge Sort and once in reverse order, which is an average case speaking effort-wise.

Fit-parameter	PPB	ATPPB (ordered)	ATPPB (random)	ATPPB (reverse)
a		$8.0 \cdot 10^{-9}$s	$3.4 \cdot 10^{-8}$s	$1.9 \cdot 10^{-8}$s
b	$1.04 \cdot 10^{-6}$s	$1.11 \cdot 10^{-6}$s	$1.14 \cdot 10^{-6}$s	$9.7 \cdot 10^{-7}$s
c	$1.79 \cdot 10^{-5}$s	$1.94 \cdot 10^{-5}$s	$2.94 \cdot 10^{-5}$s	$2.97 \cdot 10^{-5}$s

Figure 2: Fit parameters of the measurements for the PPB and the differently ordered spikes in the ATPPB command queue

When comparing the fit-parameters in figure 2, we see, that the linear coefficient b is roughly the same for all variants. The parameter c shows a high variance throughout the different measurements, but that can be blamed on statistical fluctuations and is negligible anyways (the time effort is in the order of 10^{-5} seconds). To examine the significance of the time loss through using the ATPPB instead of the PPB, I plotted the relative difference of computing effort between using the PlaybackProgramBuilder and the AbsoluteTimePlaybackProgramBuilder when passing the spike commands in correct order (see figure 3).

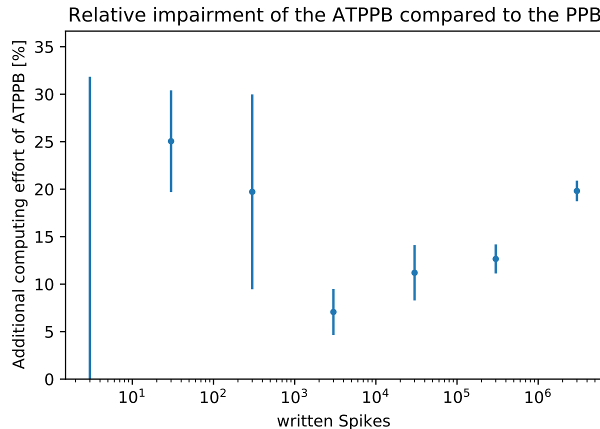

Figure 3: Additional time needed for building our sample job with the ATPPB instead of the PPB

One can see, that the measurements for the four greatest number of spikes are all below 20% and grow slowly, i.e. approximately logarithmically. The three values for the lower number of spikes don't quite match the pattern of the other measurement values and neither the expectation, because one would expect a steadily rising relative impairment and not one that's high to begin with and then drops for a certain number of spikes. I blame this on the missing accuracy of my time measurements, but the behaviour of the impairment is more interesting for high numbers of spikes anyways. To conclude the chapter of performance analysis , one can say that the AboluteTimePlaybackProgramBuilder needs a significant, i.e. noticeable longer time to build, but it doesn't grow quick. In fact, it doesn't even need double the build time for any reasonable amount of spikes, which is feasible for our system anyway.
In my opinion this is a good price for the advantages of the ATPPB.

3 Integration into the API

The API of the BrainScales-2 System can be divided into different layers [**4**]. The ones, that are important for the integration and the use of the ATPPB are the hardware abstraction layer, grenade and PyNN[**8**]. The hardware abstraction layer is a lower level layer, where the PPB and the ATPPB are implemented in. It manages the translation of abstract configurations, e.g. the value of a synapse, into hardware instructions. Grenade is a hardware abstraction layer, which translates from a perspective of a neural network to concrete commands in sense of a coordinate and a container. A layer of neurons can be described as a population of neurons and their synapses as a projection. It thus enables an easy and illustrative use. In the uppermost layer, the whole functionalities are just wrapped, so that they can be used in PyNN, which is a simulator-independent Python dialect for constructing neural networks. To make the AbsoluteTimePlaybackProgramBuilder compatible with grenade and PyNN, some adjustments had to be made in the API: The AbsoluteTimePlaybackProgramBuilder replaced the PlaybackProgramBuilder in the automatic spike generator of grenade. This was necessary, to make the fusion between user-generated ATPPB command-queues in the inside real time section and the automatically generated spike trains from possible. This merge takes place in the execution instance builder, which had to be extended by this functionality. It then took only a few initializations to forward these new functions to the PyNN-frontend, so that they can actually be used.

4 Example of a use case

To demonstrate a use case, where the AbsoluteTimePlaybackProgramBuilder comes in handy, I made a revision of an existing plasticity experiment[**3**]. The experiment is about the dynamic change of synapse weights on the chip with the so called plasticity processing units. The goal of this demonstration is to take a 64 by 64 pixel image as an input (in this case the visions-logo[**2**]) and recreate it by feeding it through the synapse array and logging the output of the neurons. This is done by feeding in a constant spike train and switching the according synapses for each pixel column to the according values, so we can detect these pixel values at the neuron outputs. With the measured data, one can recreate the image and compare it to the original one.
The above mentioned advantage of the ATPPB, that I can use the automatic spike-generator for producing this continuous spike train as input and change the weights at the same time, comes also in this experiment into play. This constant spike train is necessary, so that the weights of the synapses translate to the amount of spikes, which the output neurons can detect. In the original version of the program, this is done by using a plasticity processing unit (PPU), which runs parallel to the rest of the system. This is also a way to bypass the necessity of building the whole job manually with the PlaybackProgramBuilder, but it bring some disadvantages with itself: The user has to know how to operate the PPU, and the PPU doesn't have as much memory as the FPGA to store the input data for example. When using the PPU, one also must embed a C++ section into the Python code, when formulating the plasticity rule, whereas one only needs to know Python and PyNN for solving this problem with the AbsoluteTimePlaybackProgramBuilder. Also, the ATPPB is relatively easy and intuitive to use:
One can initialize it and inject it into the inside real time section of the program like so:

```
builder = AbsoluteTimePlaybackProgramBuilder()
configuration = pynn.InjectedConfiguration()
configuration.inside_realtime = builder
pynn.setup(enable_neuron_bypass=True, injected_config = configuration)
```

Then, one can add commands to this ATPPB later in the program, when one knows the coordinates of the automatically chosen synapses by reading out the placed connections of the projection between the abstract layer of 64 external input neurons and the abstract layer of 64 measuring neurons:

```
synapse_coordinates = projection.placed_connections
```

When knowing these coordinates, one can just loop over the pixel values and command the change of the weights of the according synapses:

```
for j in range(64):
    row_values = lola.SynapseWeightRow()
    for i,coordinate in enumerate(synapse_coordinates):
        row_values.values[coordinate[0].synapse_on_row] = int(image[-i,j])
    builder.write(hal.Timer.Value(j*int(hal.Timer.Value.fpga_clock_cycles_per_us)
        *10000), coordinate[0].synapse_row.toSynapseWeightRowOnDLS(), row_values)
```

In this code section, we take in each outer loop (for each j) an empty SynapseWeightRow container and fill this weight row in the inner loop with the 64 pixel values of the current column of the picture, we want to feed through. After that, we write a command, that we want to change at time j*10ms our synapse weights to the values stored in `row_values` to the command queue of the AbsoluteTimePlaybackProgramBuilder that we injected into the inside real time section.
Now the program is good to go.

5 Discussion

The AbsoluteTimePlaybackProgramBuilder gives up on relative timing and therefore misses some functionalities the old PlaybackProgamBuilder provides. But as this is necessary to create a mergable builder, the ATPPB can rather be seen as an addition, not as a replacement for the PPB. Future experiments might not consist from one final builder queue, but from several different smaller programs, originating from different kinds of builders in different sections of the total experiment. The ATPPB is an important base for improvements of the user experience by providing possibilities for many future projects, which require a mergable builder.
Against this wide range of possibilities and advantages, the increase of computation effort in the order of 20% seems worth it. The approach of implementing the ATPPB on top of the PPB makes it easily extendable and adjustable, as all its member functions only deal with the abstract `(time, coordinate, command)` structures.

6 Outlook

To really use the ATPPB for all manual command queueing, it is inevitable to extend the AbsoluteTimePlaybackProgramBuilder by a `read` function, so that any kinds of operations on the chip can be commanded with the ATPPB. From a user perspective, this would make the switch from PPB to ATPPB really worth it.
The ATPPB also makes another future project possible: Changing the configuration dynamically during a longer experiment in a user-friendly way. The user might want to build an experiment and run it for certain time intervals, between which they'll change the configuration of some chip components. On a lower level, this of course must be constructed out of one single job, which has to manage the absolute times of the run-intervals and the configurations. The AbsoluteTimePlaybackProgramBuilder would be the right tool for that.

References

[1] Luciano Almeida (13.09.2018): Exploring some Standard Libraries sorting functions
https://medium.com/@lucianoalmeida1/exploring-some-standard-libraries-sorting-functions-dd633f838182

[2] Electronic Visions Group (09.05.2022): visions.png
https://GitHub.com/electronicvisions/brainscales2-demos/blob/master/{_}static/tutorial/visions.png

[3] Electronic Visions Group (05.04.2023): BrainScaleS-2 on-chip plasticity experiment
https://github.com/electronicvisions/brainscales2-demos/blob/master/ts_02-plasticity_rate_coding.rst

[4] Müller E, Arnold E, Breitwieser O et al. (2022): A Scalable Approach to Modeling on Accelerated Neuromorphic Hardware
https://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=4461

[5] Pehle C, Billaudelle S, Cramer B et al. (2022): The BrainScaleS-2 Accelerated Neuromorphic System With Hybrid Plasticity
https://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=4462

[6] Eric Müller (30.03.2020): Extending BrainScaleS OS for BrainScaleS-2
https://arxiv.org/abs/2003.13750

[7] Philipp Spilger (2021): From Neural Network Descriptions to Neuromorphic Hardware — A Signal-Flow Graph Compiler Approach
https://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=4172

[8] Andrew P. Davison, Daniel Brüderle, Jochen Eppler et al. (2009): PyNN: a common interface for neuronal network simulators
https://www.frontiersin.org/articles/10.3389/neuro.11.011.2008/full