On parameterization and debugging of PPU programs

Internship report

Philipp Spilger

August 7, 2018

Contents

1 Introduction

The HICANN-DLS is a hybrid neuromorphic hardware processing unit. It features an analog part consisting of neurons (LIF neurons on DLSv2) with synapses and a digital part, spike routing and especially the Plasticity Processor Unit (PPU), which can alter network parameters at runtime.

1.1 Neuron and synapse layout

Each synapse can receive stimulus by exactly one input, which can be either external (from the FPGA) or internal from one of the neurons on the chip. To control, which synapse receives stimulus from which source, each synapse holds a hash (address) to check the origin of potential stimulus and only accepts stimulus, if the source address matches the address set at the synapse. To control the strength of connections, each synapse exhibits a 6bit weight, which scales incoming stimulus.

1.2 PPU

The PPU is a microprocessor, which incorporates the PowerPC-ISA 2.06 [2, p. 245] and additionally exhibits a vector unit for parallel operations on 128Bit vectors for faster execution, which are sliceable in 16x8bit or 8x16bit. The PPU's main purpose is modifying the synapse weights based on state parameters, e.g. spiking causality of individual synapses or current weights as well as external input, from now on called plasticity. To enhance modification speed, the vector unit can directly process synapse parameters vector wise, as depicted in figure 1.1.

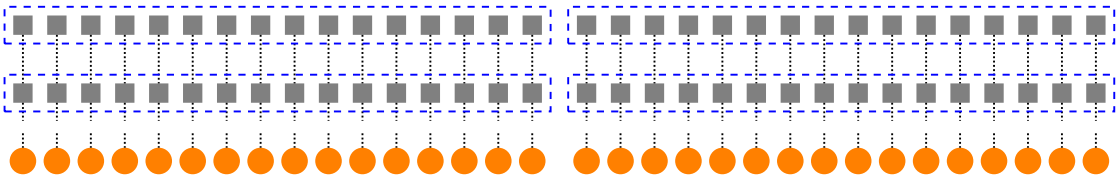


Figure 1.1: Vector access to synapses (gray rectangles) on DLSv2, displayed as blue dashed lines, together with corresponding neurons (orange circles) and synapse-neuron connections (black dotted lines).

1.3 Motivation

The prototype chip features 32 neurons with 32 synapses per neuron. Until now, modification programs were mostly written monolithic, i.e. only incorporating one weight modification rule addressing all synapses. This leads to several upscaling problems. It is expected, that bigger sized chips feature defect synapses and neurons, which exhibit e.g. false weight scaling or time constants, or fire constantly even without synaptic input. These parts should be excluded from modification rules to stabilize their behavior to e.g. a non spiking state for defect neurons or a blocking state for defect synapses. Additionally, larger chips enable multi-rule networks, with multiple (possibly different) modification rules. The monolithic kernels forbid easy restriction of rules to dedicated portions of the chip and to easily configure existing rules to coexist. Besides that, execution of the plasticity rule often was carried out through a simple loop, which makes it difficult to stabilize timing on changes in the plasticity algorithm and to enable different timing constraints of different tasks to be executed side by side. This work starts to address these problems by:

1. introducing a mask commands and implementing storage methods, which mask pasticity rules together with a modified (masking aware) vector write command,
2. deploying a common plasticity rule interface to enable interchangeability and reuseability of plasticity kernels,
3. implementing a realtime scheduler, executing operations based on timing constraints.

In addition, remote debugging using the gnu debugger (gdb) to ease debugging of programs written and libc as well as libstdc++ support for the PPU to enhance usability of the general purpose part of the processor is implemented.

2 Masking and Plasticity Rules

2.1 Masking

2.1.1 Mask command in vector update rule

Update rules are typically written in one monolithic assembler block using the vector extension of the PPU to parallelize the algorithm to increase the execution speed. An update rule somehow generates a vector (out) and afterwards stores it, e.g. to a hardware address, see listing 2.1.

```
fxvoutx %[out], %[out_base], %[out_index]
```

Listing 2.1: Vector unit indexed write instruction. Vector out is written to the hardware address specified by out_base and out_index.

To implement masking in an existiting update rule, only the write commands (fxvoutx and fxvstax) have to be modified, see listing 2.2.

```
fxvinx   %[tmp], %[out_base], %[out_index]
fxvcmpb  %[mask]
fxvsel   %[tmp], %[out], %[tmp], 1
fxvoutx  %[tmp], %[out_base], %[out_index]
```

Listing 2.2: Vector unit mask instructions.

First, the unchanged values are read and stored in the vector tmp. Then there's a mask vector, which is entrywise compared to zero. The instruction fxvsel, with fourth parameter set to one, entrywise selects an out entry, if mask is larger than zero or the unchanged entry in tmp, if mask is zero and stores the result in tmp. Last, the tmp vector is written to the hardware address specified by out_base and out_index. This allows to only update specific vector entries, leave the other entries unchanged and by that executing arbitrarily masked algorithms with the vector unit.

Note on performance

By masking a vector several times, for example if one half of a vector should be updated by another rule than the other half, the number of vector executions rises linearly with the number of rules to be executed on that specific vector. Eventually, the speed advantage through parallelization of the vector unit is lost, when each vector entry is changed individually. Therefore, speed optimization should consider minimizing the number of vector executions by grouping entries with the same update rule together in as few vectors as possible.

2.1.2 Boolean Mask storage

A typical update rule needs several mask vectors, e.g. to mask the whole DLSv2 chip, 64 possibly unique mask vectors are needed. Therefore, an efficient storage method is needed. There are mainly two characteristics to be considered, memory consumption and access speed. Three storage methods for single vector boolean masks are considered in the following, vector intrinsics, i.e. byte arrays aligned in memory to 16B, unaligned byte arrays, and bitsets. They are compared in table 2.1. The speed penalty has to

method	advantage(s)	disadvantage(s)
vector intrinsic	Does not have a read speed penalty compared to loading a vector intrinsic.	16 bytes needed per mask vector, storage structures are 16 bytes padded, thereby there may be unused space.
16B array	Does not imply 16 bytes padded structures.	Compared to vector intrinsic has a read speed penalty of 164(20) ppu cycles (measured with compiler optimization -O2)
bitset	Only needs two bytes per mask vector, which is 8 times less than the other two methods.	Compared to vector intrinsic has a read speed penalty of 166(20) ppu cycles (measured with compiler optimization -O2)

Table 2.1: Comparison of storage methods for boolean masks.

be set in relation to typical update rule execution times. In table 2.3, two rules are measured to take about 31 ppu cycles (simple constant rule) and 933 ppu cycles (stdp rule) to update weights in one masked vector. The speed penalty of accessing a mask vector relative to the rules therefore ranges between about 535% and 18%. The array storage method has equal memory consumption to the vector intrinsic storage method, but with significant speed penalty and was therefore discarded. A full set of mask vectors, stored as intrinsics, i.e. 64 for DLSv2, needs 1kB space in memory. As the ppu has 16kB internal memory available, the space needed in comparison to the bitset (128B for 64 vectors) was weighted less important than the speed penalty of generating a vector from a bitset entry each update run. Additionally to the mask vector, there's information needed as to which hardware vector address space the mask vector corresponds (out_index in 2.2). There are two ways to deliver this information. Either all vector addresses are iterated, and therefore, a mask vector for each hardware vector address is needed, or the address has to be specified explicitly alongside the mask vector. The latter allows to not specify vectors, which are fully disabled, thereby saving memory, but with more memory consumption for every vector, which is (partially) enabled. This is chosen, because the address memory usage is small (only 1B more for each vector) and the linear memory

consumption scaling with number of (partially) enabled vectors treats few and many mask vectors equally. This mask storage method is implemented in the Mask structure. In addition to the above, fully enabled vectors are only stored by address, which saves 16B per fully enabled vector. Thus, the proposed performance optimization in 2.1.1 also optimizes the memory usage of the mask storage, thereby not creating an additional, inflicting goal to be achieved.

2.1.3 Mask structure

The Mask is a size-templated structure holding an arbitrary number of vector addresses and addressed vectors. Vectors to be fully updated are only stored by address, partially masked vectors are stored by address and associated mask vector. The size and division between fully and partially enabled vectors is compile time fixed. The Mask is size optimized and because structures incorporating vectors are 16B aligned, separate address and vector arrays were chosen over a more convenient array of address-vector pairs for the partially masked vectors. One addressed vector therefore needs 17B space (16B vector, 1B address) as opposed to 32B when implemented in mixed pairs (in which the address would be 15B padded).

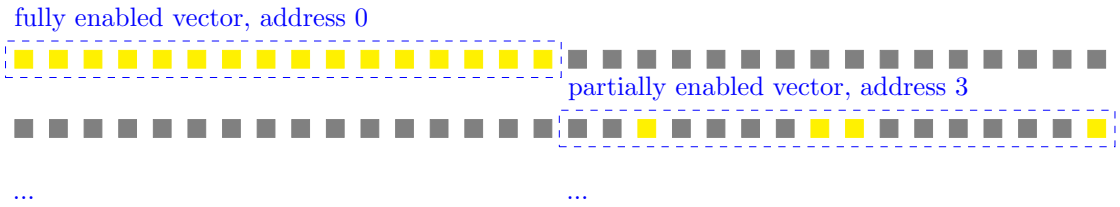
```
typedef uint8_t vec_addr;

template<size_t num_full_vectors, size_t num_partial_vectors>
struct Mask {
    vec_addr full_vec_addr[num_full_vectors];

    vec_addr partial_vec_addr[num_partial_vectors];
    vector uint8_t vectors[num_partial_vectors];
    ...
};
```

To allow arbitrary modification rules to simply update all vectors in a Mask, Mask has a functor, executing a vector update rule on all vectors stored in it, as shown in listing 2.3

```
template<size_t num_full_vectors, size_t num_partial_vectors>
struct Mask {
    ...
    template<class T>
    void apply_vector_rule(
        T& t,
        void (T::*vector_rule)(
            vec_addr,
            vector uint8_t&));
};
```

Listing 2.3: The interface of the vector rule functor of a Mask.

The Mask storage method is depicted in figure 2.1.




```
Mask<1,1> this_mask = {
    {0}, {3},
    {{0,0,1,0,0,0,0,1,1,0,0,0,0,0,0,1}}
};
```

Figure 2.1: Abstract mask with enabled synapses highlighted yellow and a corresponding Mask structure.

Note on optimization

For smallest memory consumption, chip usage should be optimized for masking preferably full vectors. Best case memory consumption of a Mask is $1 \cdot \#$(vectors enabled)B. Worst case memory consumption is established, if there are only partially masked vectors. Then, memory usage is $17 \cdot \#$(vectors partially enabled)B, upward ceiled to full 16B multiples because of struct alignment.

2.1.4 Tagged Mask structure

The previous approach fits static masking without the need to switch plasticity of synapses during runtime. If switching is required, the single synapse state would need to be stored in as many Mask structures as there are plasticity rules to be switched between, which in principle unnecessarily increases memory consumption. To accomodate this usecase, a different storage method is described in the following. By again using $16 \cdot 8$bit vector intrinsics as base storage type, the membership of synapses to a certain plasticity rule is encoded in the value of the vector entries, in the following called tag. This leads to possibly 256 differentiable plasticity rules without space overhead compared to the Mask structure. The structure storing tag vectors is called TaggedMask in the following. In order to still work with the same mask command, described in listing 2.2, and thereby to omit the need to modify a plasticity rule because of a different mask storage method used, a binary mask vector has to be created from the TaggedMask storage at each invocation. This storage method permits to dynamically change the synapse-plasticity rule link at the cost of a significant speed penalty because of the tag to binary vector conversion. Figure 2.2 shows the TaggedMask storage method. Similar to the Mask structure, the TaggedMask provides a functor for executing a plasticity rule operating on vectors, shown in listing 2.1.4.

```
template<size_t num_full_vectors, size_t num_partial_vectors>
struct TaggedMask {
    ...
```

```
    template<class T>
    void apply_vector_rule(
        T& t,
        void (T::*vector_rule)(
            vec_addr,
            vector uint8_t&),
        uint8_t tag);
};
```


```
enum tags : uint8_t {
    blue,
    green,
    yellow
};

TaggedMask<2> this_mask = {
    {0,3},
    {
    {yellow,...,yellow},
    {blue,blue,green,blue,blue,blue,blue,green,
     green,blue,blue,blue,blue,blue,blue,green}
    }
};
```

Figure 2.2: Abstract multiple-plasticity-rule-mask with synapses highlighted according to tags and a corresponding tags enumeration together with a TaggedMask structure.

2.2 Plasticity Rule

To transparently allow using several update rules at once and reusing rules, the update rules are to be encapsulated, and made accessible through a common interface. A plasticity rule has to have a function updating vectors, compare listing 2.4. If the vector update function is masking aware, the described Mask or TaggedMask functor then can update all entries enabled by that mask according to the vector update function of a plasticity rule. This calling sequence is unintuitive though compared to calling a function of a plasticity rule, that updates entries according to an associated mask. The sequence flip is done in the MaskWrapper and Tagged::MaskWrapper wrapper classes.

```
class VectorRule
{
```

```
public:
    VectorRule()
    {
    }

    void vector_rule(vector uint8_t& mask_vector,
        uint8_t vector_address)
    {
    }
};
```

Listing 2.4: An example vector rule implementation.

2.2.1 MaskWrapper class

A plasticity rule is an object promoting an update function (run), that updates e.g. synapse weights according to an associated Mask or TaggedMask mask. Therefore a templated wrapper class was developed, inherited from a vector rule class.

```
template<class VectorRule, typename Mask>
class MaskWrapper : VectorRule
{
    Mask& mask;
public:
    template<class... Args>
    MaskWrapper(Mask& mask, Args... args) : mask(mask), VectorRule(args...)
    {
    }

    void run()
    {
        this->mask.apply_vector_rule(
            static_cast<VectorRule*>(this),
            &VectorRule::vector_rule);
    }
};
```

Listing 2.5: An example implementation of a wrapper class for a vector plasticity rule VectorRule operating on a Mask.

The run function calls the associated mask functor with the MaskWrapper's vector update rule inherited from VectorRule. Similarly, the Tagged::MaskWrapper stores a reference to a TaggedMask and additionally needs a tag, stored during construction.

2.3 Performance measurements

To measure the performance drawback of binary masking, two plasticity rules are examined, a constant rule and a stdp sampling rule [4, p. 58]. In the constant rule, only the write command of weights is to be masked, in the stdp sampling rule, the causal and acausal measurement reset command is to be masked additionally. It's therefore expected, that the performance drawback in the stdp sampling rule is larger (absolutely)

than in the constant rule. Additionally, the time consumption is expected to depend linearly on the number of vectors updated and not to depend on the entry values of the mask vector, e.g. the time consumption shouldn't differ between partially and fully enabled vectors. Tables 2.2, 2.3 and figure 2.3 show the measurement results.

	computation time [ppu cycles]	
rule	masking enabled	masking disabled
constant	3720(100)	2510(50)
stdp sampling	70050(80)	48930(100)

Table 2.2: Comparison between the computation time with and the computation time without masking enabled, with compiler optimization -O2.

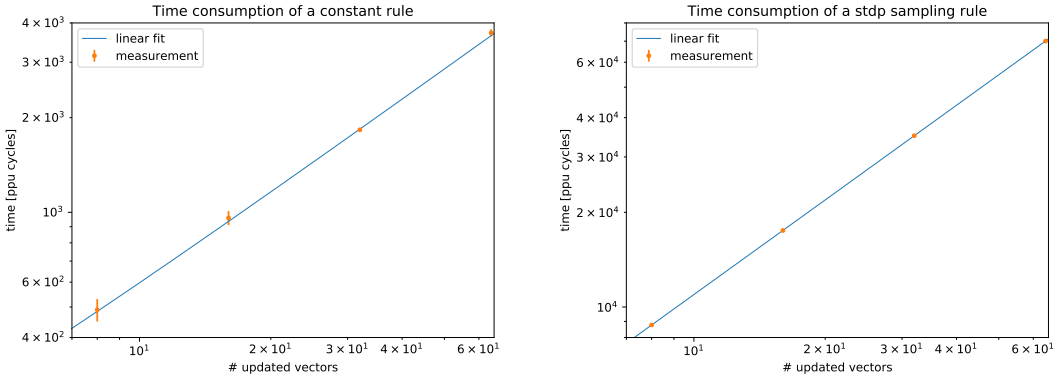


Figure 2.3: The computation time in dependence of the number of vectors updated, with compiler optimization -O2.

	computation time [ppu cycles]	
rule	full mask	partial mask
constant	3621(27)	3816(25)
stdp sampling	69999(80)	70094(51)

Table 2.3: Comparison of the computation time of masking enabled rules between full vector and partial vector masking, with compiler optimization -O2.

For the two rules examined, the binary masking is about 30% of the total computation time. Figure 2.3 shows a linear dependence of the computation time on the number of vectors updated, as expected. Additionally, the computation time doesn't differ significantly between updating whole vectors and updating partial vectors, only the total vector count matters, compare table 2.3.

3 Scheduling

Deploying several plasticity rules at once and allowing defined timing of changes to hardware parameters asks for a task scheduling system, that runs arbitrary commands at defined times and allows defining execution order as well as logging, whether requested timing constraints were fulfilled. Until now, plasticity rules are typically run in a loop on the PPU, as can be seen in listing 3.1.

```
bool condition = true;

while(condition) {
        run_my_rule();
        condition = check_condition();
}
```

Listing 3.1: Plasticity rule execution in a simple loop.

This approach has mainly two drawbacks. First, there's no easy possibility to ensure a certain period time between successive executions of the plasticity rule. This leads to uncontrolleable time period changes, if the underlying plasticity algorithm is changed, e.g. by changing the number of synapse vectors, it operates on. Second, there's no easy possibility to implement multiple tasks with different timing constraints independently from each other. Dependent implementation can be done via only updating a certain task e.g. each 3rd loop count, which nontheless only implements relative and doesn't implement absolute timing constraints. To solve the need for absolute timing constraints to be specified for multiple tasks independently from each other, an earliest deadline first scheduler was implemented. An earliest deadline first scheduler tries to ensure execution start of a task until a certain absolute time (deadline) by prioritizing execution of the task with the earliest (i.e. smallest) deadline. It does not need to know the task's time consumption beforehand, which is the case with a rate monotonic scheduler. Because knowing the time consumption beforehand involves measuring each task's duration in advance, which is not possible for stochastic rules or not feasible and by online change of task priority creates additional computation time needed, the earliest deadline first scheduler is chosen. The scheduler handles an arbitrary number of services and event sources. Services are identified by an ID and execute a function or a class member function via service.exec(). An event is a pair of a service ID and a deadline, until which the event's service should have been started. Event sources are abstract objects, which can be queried for new events via source.next_event(time). A timer is an event source using the supplied absolute time to decide whether to generate a new event or not.

3.1 PPU timer register

The PPU provides a 64bit wide special purpose timer register counting single clock cycles. It's precision is therefore sufficient for arbitrary timing demands. The PPU doesn't have a 64bit integer division unit. Therefore, operating on 64bit times is significantly slower than operating on 32bit times. With a clock frequency of around 500MHz [2, p. 172], 32bit times experience overflow after $t \approx 8.6s$. Because experiment time is not bound to be that short, times can't be always cast down to 32bit integers. To account for that, the time type can be set to either 32bit or 64bit unsigned integers and additionally, the time can be bitshifted for a larger count until overflow occurs with the tradeof of a smaller resolution, see eq. 3.1, where t is the time from the timer register in 64bit representation, t' is the shifted time in 32bit representation and s is the number of bits shifted.

$$t_{32} = \frac{t_{64}}{2^s}, \ s \in \{0, ..., 63\} \tag{3.1}$$

3.2 Services

In order to establish a common interface for execution commands, the Service class is developed, allowing to execute an arbitrary function or class member function of an arbitrary class by calling service.exec() A service has only one exec function and therefore allows only access to one command. In order to allow several e.g. class member functions of the same class to be made available as services, a service for each member function has to be created. A service is identified by ID, an integer, in order to allow mapping (non-templated) events to existing services, as described in 3.4. A service can be created as in the following example described in listing 3.2

```
void func() {}
service_id func_id = 1;

class C
{
public:
        void run() {}
};
...
C c;
service_id class_id = 2;

auto func_service  = Service_Function<func_id>(&func);
auto class_service = Service_Class<class_id>(c, &C::run);
```

Listing 3.2: Exemplary creation of a Service for a function and a class member function.

3.3 Event sources

A scheduler organizes events to be executed. Since the PPU's memory is limited, events have to be created during runtime. To allow different sources of events, e.g. sources for single events or multiple events at once, a common event source interface is needed. An event source can be queried for (a) new event(s) through a function call, which takes the current absolute time as argument and returns an event, if an event is to be scheduled. The declaration can be seen in listing 3.3

```
bool Source::next_event(Event& event, time_t t);
```

Listing 3.3: Event source next_event function declaration.

Two commonly demanded cases of event sources are periodic and oneshot timers, which are implemented and are described in the following.

3.3.1 Periodic timer

To allow for periodically timed events, a timer class was developed. A timer has the three timing parameters start time, period time or time until repeat and number of periods to run. Additionally, a timer has an associated service for which events get generated. The timer class also stores the number of lost periods, i.e. periods without an execution, for timing validation. The timer generates a new event once at entering a new period with the next execution time as deadline. Events for missed periods don't get generated, but their loss is counted for later examination.

3.3.2 Oneshot timer

To allow for single events, a oneshot timer was developed. It has two timing parameters, deadline and earliest time to generate the event. As the periodic timer, a oneshot timer has an associated service for which the event eventually gets generated. Unlike with the periodic timer, missing the earliest to be run until deadline timeframe does not imply, that the event doesn't get generated, since execution at all is weighted more important than exact timing for single events.

3.4 Scheduler implementation

The scheduler provides an event queue and an execution loop. All incoming events are stored in the queue, implemented as fixed-size circular buffer of events. The execution loop repeadetely queries event sources, pushing fetched events onto the queue, sorts the earliest deadline event first, afterwards pops the first event in the queue and executes its service. The execution loop can be controlled externally either by PPU timing or by e.g. signals written from the FPGA to PPU memory. The control flow implemented in the scheduler execution loop is shown in figure 3.1.

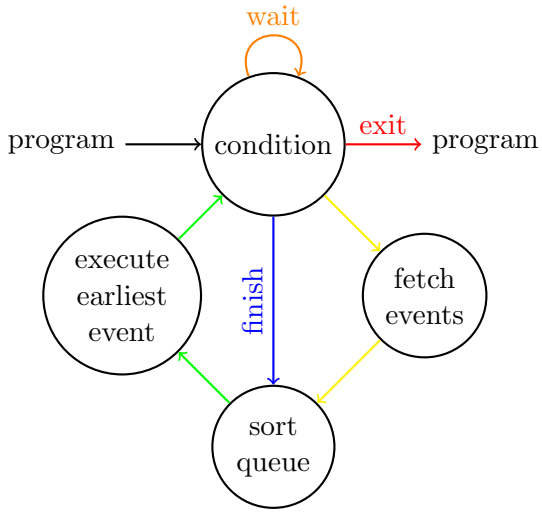


Figure 3.1: The scheduler execution loop and its control flow.

3.4.1 Queue class

Queue is a templated circular buffer of compile-time fixed size. Elements can be pushed to or poped from the Queue. Additionally to this circular buffer capabilities, the elements currently stored in the Queue can also be accessed as if Queue was a dynamically sized array (with upper size limit), indexed from the next to the last to be poped element. This enables easy implementation of ordering algorithms as needed by the scheduler. Queue keeps track of occuring buffer overflow and also saves the maximally reached number of elements stored since creation. This enables users to examine performance issues related to possibly too small queue size.

3.4.2 Scheduler class

The Scheduler class is created with the queue size:

```
auto scheduler = Scheduler<Queue_size>();
```

Scheduling starts by calling

```
scheduler.Execute(SchedulerSignaller,
                  Tuple<Services...>,
                  Tuple<EventSources...>);
```

with event sources and services provided as tuples. The SchedulerSignaller controls the execution loop flow by providing an interface to the control signals wait, finish and exit, as described in figure 3.1. The signaller is implemented in two ways. There's a mailbox signaller listening on a memory address for external control signals and there's a timer based signaller, similar to a timer providing control signals based on start, finish and exit times to be specified.

3.5 Notes on error handling

Providing the scheduler execution loop with event sources generating events for services not included in the provided service tuple is unrecognized but does no harm except unnecessary time consumption. It is assumed by design, that service IDs are unique to an execute loop. Providing an execute loop with several services with the same ID causes the foremost service in the service tuple to be executed, once an according event is scheduled. Scheduling correctness in terms of timing can be verified by checking, that neither queue overflow nor event loss to timers occured.

3.6 Performance measurements

The scheduling performance is tested by measuring the time consumption of (parts of) the execution loop in dependence of the number of services and timers to be scheduled. Measurements are carried out for the 32bit and 64bit time type. Table 3.1 shows the time consumption for queue read and write operations. Table 3.2 shows the time consumption of Timer::next_event (...) in dependence of whether the time supplied is after the timer's stop time, at a period or at a period with a missed period. Table 3.3 shows the

	computation time [ppu cycles]	
time type	pop	push
32bit	85(+7)	90(+51)
64bit	87(+32)	83(+53)

Table 3.1: Comparison of queue read and write operations between 32bit and 64bit time type, with compiler optimization -O2. The uncertainty is due to branch prediction.

	computation time [ppu cycles]		
time type	next_event == true	next_event == true and a missed period	next_event = false
32bit	136(+75)	150(+28)	118(+23)
64bit	211(+131)	264(+20)	166(+20)

Table 3.2: Comparison of Timer::next_event between 32bit and 64bit time type, with compiler optimization -O2. The uncertainty is due to branch prediction.

time consumption of TimerOneshot::next_event in dependence of whether the event has not been fetched and there's an event to be fetched, there's no event to be fetched because time is too early or because an event already has been fetched. Queue access time doesn't depend on the time type used, as shown in table 3.1, indicating, that memory access is fast compared to calculating the correct position to be read or altered. The implemented periodic timer is between about 40% and 70% slower, when used with 64bit

time type	computation time [ppu cycles] next_event == false, too early	next_event == true	next_event = false, already fetched
32bit	105(+36)	119(+20)	97(+20)
64bit	123(+71)	137(+21)	101(+20)

Table 3.3: Comparison of TimerOneshot::next_event between 32bit and 64bit time type, with compiler optimization -O2. The uncertainty is due to branch prediction.

times compared to 32bit times. The oneshot timer also shows this difference, but not as pronounced as the periodic timer. Both timers are fastest, if the submitted time does not issue creation of a new event, i.e. next_event returns false, which is good, since then unused or waiting timers don't slow down as much as working timers. The sorting is implemented as a linear search and a swap. The time consumption is therefore expected to depend linearly on the number of events to be searched, as shown in figure 3.2. Sorting of events with 64bit time type is about 10% slower compared to 32bit times.

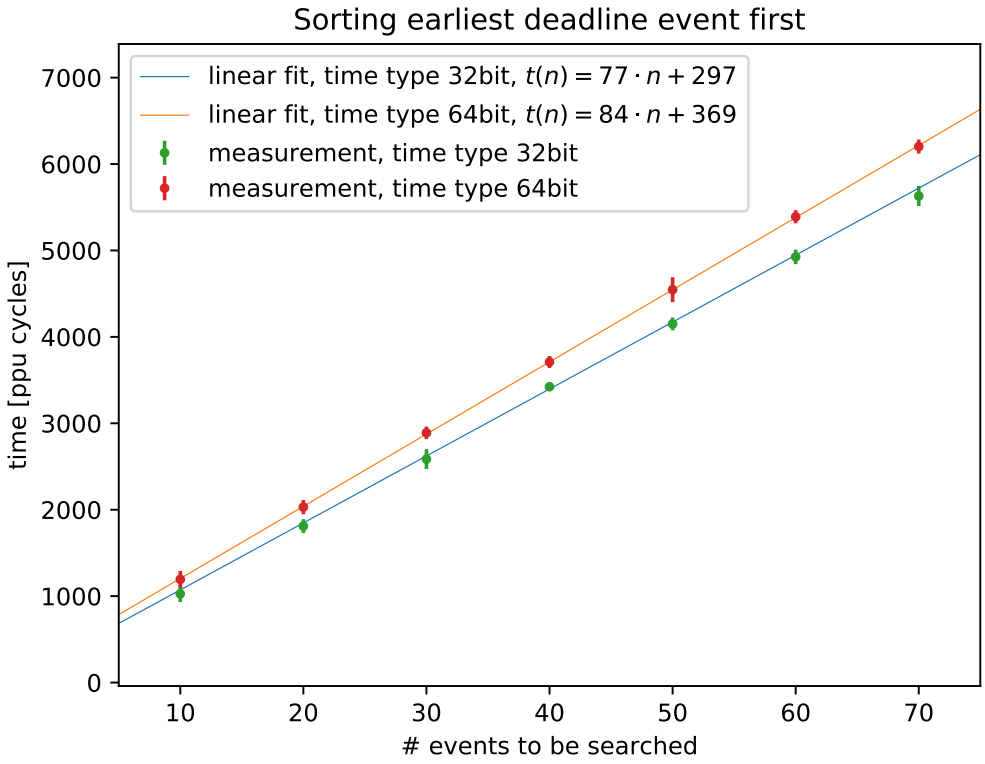


Figure 3.2: The computation time of sorting the earliest deadline event first in the queue in dependence of the number of events to be sorted, with compiler optimization -O2.

4 Remote debugging

Debugging of PPU programs was already determined necessary by other DLS users. During developement of plasticity rule masking and time scheduling, it became apparent also to the author, that an online debugging capability would ease developement. Until now, debugging is limited to saving program states during runtime to the mailbox, a dedicated 4kB fixed size area in the PPU memory, reading out the written variables or results after a program run, evaluating the results and running the program again, until the result matches the goal to be achieved.

This approach is problematic. First, since it's a pure kernel, that's executed, there are e.g. no perfect memory guards in any case preventing the stack from growing into the program memory region [1]. This may lead to faulty behavior of the program execution such that the program starts over and over again, no results get written at all to the mailbox region to be evaluated or the mailbox region gets overwritten by the faulty program. This prevents the developer from gaining any insight into why the program fails in executing correctly. Second, there's no way of gaining insight into the program state or changing the program during runtime. This especially leads to longer debugging time needed, if there are multiple problems with the program, since one has to implement an assumend fix of one error, run the program again from the beginning, check, if the error is gone and continue with the next.

On the contrary, host-software, e.g. software written to be executed under a linux operating system, can be debugged online, which means, the program can be started with a debugger, run to a certain state, stopped and investigated, eventually changed and continued to run from the point it was stopped until the next specified state. This way, the program state, e.g. register values, can be looked at at arbitrary positions in the program execution and presumed fixes for faulty behavior can be implemented without having to restart or even recompile the program under investigation.

Under gnu/linux, the standard debugger is called gdb (gnu debugger). It is widely used for debugging host-software and thus its application to debugging PPU software doesn't involve learning a new tool from a developers point of view. The difficulty with porting gdb to be used with PPU kernel programs is, that gdb itself can't be run on the PPU, since it's the operating system on the PPU, which is to be debugged, there wouldn't be enough memory to run gdb and there's no online communication implemented to the PPU until now, which is needed in order to issue commands to gdb or query state

[1] There is a program memory region protection, which works by checking the stack pointer position against a redzone region, located between stack pointer initial position and the program memory region, during function prologue. It thereby only catches too large stack growth, if the faulty function's stack allocation doesn't exceed the redzone size. Otherwise, the redzone is possibly unrecognized overstepped.

variables. Luckily, gdb is already set up for debugging remote targets not running on the same platform or operating system. To achieve that, gdb has a remote protocol implemented[1], with which the debugger can communicate with the target program to be debugged. With this, the first two problems are solved, leaving the gdb to target communication, translating gdb requests into tasks for the PPU and creating responses back to gdb to be implemented.

4.1 Online PPU to host communication

The hicann-dls is run with host software creating FPGA programs, which are sent to the board's FPGA executing the created programs. After execution, the host can evaluate the last program's results and create new FPGA programs to be executed. With this, it is also possible to set and read out PPU memory. The PPU runs independently from the FPGA program or the host software, until something in its memory is changed by the FPGA. This allows communication during runtime by having a dedicated input and output buffer region in the PPU memory. The PPU program is started and afterwards, sufficiently short FPGA programs get created, which read out the output buffer memory region. Changes in this region get evaluated by host software, processed and sent to gdb. On the other hand, the host can communicate with the PPU by issueing FPGA programs, which change the input buffer memory region, at which the PPU watches for changes. Communication speed is limited by the duration of a single FPGA program and the buffer size, but since debugging only requires communication, if interaction with the developer is necessary, communication speed of PPU and host isn't the limiting factor.

4.2 Remote target communication

The host software uses the haldls hardware abstraction layer to build FPGA programs in order to start up and communicate with the PPU program to be debugged. Communication of the host software and gdb is established via TCP/IP and it's therefore possible to debug programs without the need of a physical dls board standing nearby the computer running gdb.

The combination of PPU program with debugging communication extension and host software communicating with the PPU and gdb is called gdbserver and typically is not split, but is one program, if the machine to be debugged has direct access to e.g. an ethernet or a serial line connection, and the memory overhead of the debugging communication software is not significant compared to the program to be debugged. Since these two requirements don't hold for the PPU, taking away as much complexity of the debug communication as possible from the PPU program is chosen. This way, the PPU debugging software deals with querying and setting register values, flushing the cache, replacing instructions at step requests and exiting the debugging loop, since these are the only things not accessible from the host software. The host software deals with deploying the PPU program, querying run status, reading and writing PPU memory, request categorization from gdb requests and response creation complying with the gdb

remote protocol. Figure 4.1 shows the distribution of handling gdb requests and the communication between gdb and the PPU.

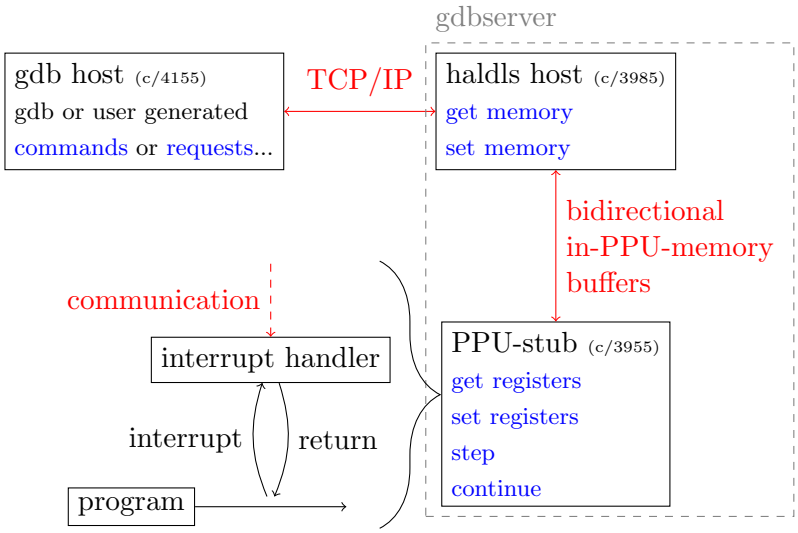


Figure 4.1: Description of debugging communication (red), request distribution of gdb requests in gdbserver (blue) and structure of the PPU-stub.

4.3 PPU-side debugging software

The PPU-side debugging software, developed by Timo Wunderlich, is split in mainly two parts. An exception handling function is implemented, which gets called, whenever the PPU program can't handle an error that occurred. This exception handling function then enables communication to the host software via reading and writing to the in-memory buffers. Eventually, when the exception is resolved, the developer can e.g. exit the exception handler function and continue execution of the program. The second part is a breakpoint function, which executes a trap instruction. A trap is an instruction, which prohibits the program counter to increase, triggering an exception and by that allowing the developer to manually enter the exception handling function, enabling debugging communication at arbitrary program states.

4.4 Debugging workflow

In order to use the debug facility, a developer needs to include the PPU debugging software header and insert the breakpoint function at program positions of interest.

```
#include "libnux/ppu-stub.h"
...
// interesting program state
breakpoint();
```

```
...
```

Listing 4.1: Manual setting of debug breakpoints in a PPU program.

The host debug software is to be run with an allocated dls board, the PPU program to be executed and a specified TCP port number. The PPU program has to be provided both stripped for inserting into the PPU memory and non-stripped in order to let the host software automatically determine the addresses of the communication buffers in the PPU memory.

```
user@host: ppu_gdbserver \
  --port 54321 \
  --board-id 07 \
  --program my_ppu_program.bin \
  --stripped-program my_ppu_program.binary
```

Listing 4.2: Options to startup the PPU gdbserver.

At the same time, gdb has to be started, the architecture has to be set to nux, and the remote target has to be connected.

```
set architecture powerpc:nux
target remote host:54321
```

Listing 4.3: Setting of the target architecture and connecting to a remote target inside a running gdb instance.

This establishes a connection of gdb to the remote program and queries information, once the remote program has reached a breakpoint. The subset of gdb commands implemented so far for a PPU target can be seen in table 4.1.

command	functionality
quit	Exits gdb and the remote host software as well as the PPU program.
info register	Shows the current general and special purpose register values.
info vector	Shows the current vector unit register values.
info all-registers	Shows all register values.
continue	Continues execution by leaving the current interrupt.
backtrace	Shows backtrace of all stack frames.
finish	Executes until the current stack frame returns.
return	Return to the outer stack frame.
set variable name = value	Set the variable name to some value
print name	Print the value of variable name.
break function	Set a breakpoint at the beginning of the function function.
set $kv0.v16_int8 = {...}	Set the value of vector register 0 to some value.

Table 4.1: Currently supported gdb commands for PPU programs.

5 Towards a semi-hosted software environment

The programs written for execution on the PPU are kernel programs. That means, there's no operating system running on the PPU executing written applications. The written programs themselves are the operating system and the only program running at a time on the PPU.

As a consequence thereof, the software environment is missing parts normally found when developing applications to be run by an operating system. Most notably, there's neither a standard c library nor a standard c++ library, since these require an underlying operating system to issue certain requests to, called system calls, e.g. to allocate more memory on the heap, to open a file or to exit an application.

The c++ standard library provides certain features, e.g. std::bind or std::tuple, which are helpful in providing scheduler functionality, compare e.g. 3.4.2, because they allow variables to be bound to a certain task, meaning the task 'knows', which arguments it was created with (std::bind), or handling several possibly different tasks in a uniform manner (std::tuple).

The standard c++ library can only be build including these features, if there's an underlaying standard c library available, because for certain features, e.g. new or file handling, there are again system calls required.

5.1 Standard c library

There exist serveral standard c library implementations. Typically found on gnu/linux systems, there's glibc (gnu c library), which has the largest functional coverage, but therefore creates large executables not suited for embedded systems. Also there exist several c libraries requiring linux header files to be present, e.g. musl libc or uClibc. Because of that, they can't be used for the PPU, since on the PPU a custom kernel, i.e. not linux is to be targeted. Besides those two, there's Newlib. It doesn't require special operating system headers but just implementations for some system calls to be present and because of that was chosen to be suited best for porting to targeting PPU kernel programs. Also, the developers of Newlib help in porting to a new operating system by specifically stating the required systemcalls to be implemented and even provide stub (i.e. non functional) implementations [3] of every system call required.

The required 20 system calls are shown in table 5.1 together with a short description and explanation on whether a functioning implementation can be provided for the PPU or more specifically for kernel programs running on the PPU.

system call	functionality	Can it be implemented targeting kernel programs for the PPU?
_exit	Exit a (userspace) program.	Yes, wrapper to exit of kernel program.
close	Close a file.	No, since there's no file system implemented in the PPU kernel.
environ	A pointer to environment variables.	No, since a kernel has no surrounding operating system providing these variables.
execve	Thransfer control to a new process.	No, since there's no surrounding operating system executing userspace programs and creating new processes.
fork	Create new process.	No, see execve.
fstat	Status of an open file.	No, see close.
getpid	Get process ID.	No, see execve.
isatty	Query, whether an output stream is a terminal.	No, since there's no output stream functionality implemented.
kill	Send a signal to a process.	No, see execve.
link	Change the name of an existing file.	No, see close.
lseek	Set position in a file.	No, see close.
open	Open a file.	No, see close.
read	Read from a file.	No, see close.
sbrk	Increase allocated memory.	Yes.
stat	Query the status of a file.	No, see close.
times	Timing information.	Yes, but the only functionality without gettimeofday is a wrapper to a assembler call already implemented in libnux.
gettimeofday	Get current time.	Yes, in principle, but because of clock resets at kernel startup, it's implementation doesn't make sense and a clock frequency measurement would be necessary to translate the clock counter state to e.g. seconds.
unlink	Remove a file's directory entry.	No, see close.
wait	Wait for a child process.	No, see execve.
write	Write to a file.	No, except for a custom implementation writing e.g. to in-memory buffers.

Table 5.1: List and evaluation of system calls to be provided for newlib.

It can be seen, that several system calls deal with executing other programs or handling processes, which is both neither implementable nor relevant in kernel only programming. The system calls dealing with files may in principle be implementable, but aren't necessary in a monolithic kernel program, since information storage in memory can also be (more efficiently) be handled directly without a file system layer.

The newlib malloc implementation is an implementation meant for userspace programming, since it relies on an underlying operating system to provide sbrk in order to allocate memory pages, larger memory regions. By providing a sbrk implementation, the malloc implementation can also be used in kernel programming, though. The drawback compared to developing a dedicated malloc implementation is an additional function call every time, the allocated memory region grows larger than multiples of the page size. Additionally, in a monolithic kernel, the pages are not necessary, since only one program, the kernel, is allocating memory and there's no need to keep track of several programs each allocating memory by requests to a kernel, as would be in an operating system running several userspace programs. On the other hand, the implementation can be customized in terms of page size, thus minimizing the negative effects of it, and alignment of allocated memory. The former is important because of the restricted memory size of the PPU compared to e.g. personal computers, whereas the latter is important, because S2PP vectors have to be aligned to 16B in memory in order to be accessible from the vector unit and it should in principle be possible to dynamically allocate vectors during runtime. Thus, a sbrk implementation is provided in order to make newlibs malloc functional for the PPU kernel. The implementation is shown in listing 5.1. Since only one program (the kernel) is requesting memory pages, it suffices to store the current uppermost used memory page location and eventually increase or decrease it.

```
caddr_t sbrk(int incr)
{
        extern char heap_base;
        static char* current_end;
        if (current_end == NULL) {
                current_end = &heap_base;
        }
        char* prev_end = current_end;
        current_end += incr;
        return (caddr_t) prev_end;
}
```

Listing 5.1: The sbrk implementation for PPU kernel software.

5.2 Target toolchain

With sbrk implemented and the other system calls provided as stubs, a working newlib can be build to target the PPU. In order for the newlib build to know, which systemcalls are available, a new build target has to be created. Since the PPU's architecture is PowerPC, the target triplet has to be powerpc*-*-OperatingSystemName. As operating system name, oPPUlance, proposed by Dr. Eric Mueller was chosen. A new operat-

ing system name also needs changes in the binutils and gcc sources, since during the newlib build, the target specific compiler, linker, etc., such as powerpc-oppulance-gcc are expected to exist and cannot be provided, if gcc isn't set up to be built targeting the powerpc-oppulance target. Additionally, in contrast to a normal operating system target, the target is to be used for kernel developement and therefore, several automatic linkage objects, e.g. crtend or crtbegin, needed for hosted programs have to be suppressed from getting linked in into kernel executables.

5.3 Libstdc++

Libstdc++ is part of the gcc sources. It is set up to only compile it's freestanding version, which includes only the headers initializer_list, type_traits, new, limits, exception, typeinfo and some wrappers for c headers, if it is compiled to target an unknown operating system. In order to get the missing functionality, the new operating system target has to be added to the libstdc++ sources and libstdc++ has to be told to check it's features against the now to be built newlib standard c library.

The presented changes generate a new toolchain targeting specifically the PPU with a standard c library and a standard c++ library suited for kernel developement.

6 Discussion

In this internship, the execution of plasticity algorithms on the PPU was parameterized. Mask abstraction and plasticity rule encapsulation as well as timed scheduling was developed. Additionally, remote debugging support of PPU programs was established and libstdc++ support enabled.

Masking enables usage of static sized binary masks for the vector unit as well as tag based masks. Measurements show, that masking slows down execution of the two examined plasticity rules (constant and stdp sampling) by about 30% and, that execution time does linearly depend on the number of vectors processed, as expected.

The implemented scheduler enables formulation of timing constraints independent of the task to be executed and timed execution of an arbitrary number of tasks. Computation time measurements show 64bit integer time type to generate about 40% to 70% slower scheduling algorithms than with a 32bit integer time type. The time type is integrated such that the user can transparently choose between fast computation and good time resolution with large maximal time.

Remote debugging via gdb for PPU programs enables examination and alteration of register values during runtime at manually set breakpoints.

Libc and libstdc++ support enables meaningful usage of all their features which don't deal with file system access, especially e.g. `std::tuple` or `std::bind`.

7 Outlook

The implemented masking deals with the explicit mask command and methods to store mask information efficiently. Until now, only the two examined rules, constant and stdp sampling have been implemented using the plasticity rule framework and thus implementing more and more complex rules is necessary to verify broad useability.

The vector operations still are written either in inline assembler or using builtin operations. This restricts portability, since neither the assembler instructions nor the builtin operations or even the vector type exist on other platforms. Portability is necessary to test vectorized plasticity algorithms on a host computer, e.g. to ease inspection. A solution would be to implement or port a cross platform vector instruction library [1] to the PPU, implementing the operations using intrinsics. Thereby, the execution speed is optimized on all platforms, the vector library is developed for. There already exist several libraries especially targeting x86 simd extensions. Adding a PPU target to such a library would align usage in PPU programs and on host computers by providing the fastest computation method on either platform without adjustments to the algorithm's code.

The scheduler is written with a custom tuple implementation and without usage of std::function or std::bind. Since libstdc++ support is enabled now on the PPU, this functionality is to be included, as it further generalizes usage of the service class and thereby of the tasks, the scheduler can handle.

The debugging communication is parallel from the PPU to the host software, i.e. the incoming and outgoing packets are not serialized, but are large enough to hold all register values at once. In order to minimize the additional memory overhead introduced by the debugging software, this communication is to be serialized to omit the large buffers. Additionally, expanding the implemented subset of supported gdb commands, especially implementing a functional step at branch instructions, is recommended, as debugging demands increase. For step and setting breakpoint, it is necessary to flush the instruction cache for these addresses. At the moment this is only workaroundeable by branching to another address mapping to the same instruction cache address and back. Therefore, manual hardware enabled control over cache flushing would ease and robustify debugging programs.

[1] Suitable candidates might be Vc (`https://github.com/VcDevel/Vc`) and rts::vec (`https://github.com/ekmett/rts`), since they both should be zero overhead implementations space as well as speed wise, already provide integration of x86 vector extensions and strictly separate targets or backends from the interface, which enables integration of a new target without changing the interface.

Bibliography

[1] Inc. Free Software Foundation. *Appendix E GDB Remote Serial Protocol*. 2018. URL: `https//sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html` (visited on 07/09/2018).

[2] Simon Friedmann. "A new approach to learning in neuromorphic hardware". PhD thesis. Heidelberg, Univ., Diss., 2013, 2013.

[3] Inc. Red Hat. *The Red Hat newlib C Library*. 2018. URL: `https://sourceware.org/newlib/` (visited on 08/06/2018).

[4] David Stöckel. "Exploring Collective Neural Dynamics under Synaptic Plasticity". Masterarbeit. Universität Heidelberg, Nov. 2017.