Ruprecht-Karls-Universität Heidelberg

Department of Physics and Astronomy

Electronic Vision(s) group

Internship Report

Implementation of a prefetching-module

Author:
Niels Fiedler

Supervisors:
Johannes Schemmel
Vitali Karasenko

July 2021

Contents

1 Introduction

Reducing latency and thereby increasing the bandwidth is an indisputable goal in any kind of communication network. This is true for the internet, a global connection around the world, as well as in the case of a microprocessor accessing a memory unit (e.g. a static random-access memory (SRAM)), at the scale of centimeters.
One possibility to lower or rather hide latency is called prefetching.

This report is about the results of a 3 months long internship at the *Electronic Vision(s) group* in which after learning the necessary skills like basic tooling and the hardware describing language *SystemVerilog*, a module, hereinafter called "prefetcher", was designed, allowing a microprocessor of the *BrainScaleS-2* setup to prefetch reads from an SRAM, located on another printed circuit board (PCB).

1.1 Prefetching

Since the term "prefetching" comes up repeatedly in this report, its meaning and background shall be briefly discussed, to later understand the difference between common prefetching and the work presented here.

In general prefetching describes the process of heuristically loading data in advance, expecting it to be needed in the near future. For this there usually exists a cache, being either a piece of hardware or software, which can store small amounts of data and be accessed very fast (e.g. by a central processing unit (CPU)), compared to the main memory component of the system (e.g. some kind of RAM).
In case of a CPU cache, the data in the cache memory is stored with a tag, containing the address of the same data in main memory. When the CPU wants to load data from main memory, it first searches for it inside the cache. If a cache hit happens, meaning the data is found in the cache, the latency between the CPU and main memory was bypassed. Given a cache miss, the data must be loaded from main memory, resulting in the latency becoming visible. CPU caches are used for over three decades, nowadays being split into several levels.
Web browser programs similarly use caches to prefetch the contents of web addresses.
To maximize the hit rate of the cache, deciding which data to store in the cache is essential and can be handled in different ways, which are not further discussed. [4]

As the prefetching mechanism implemented here does not include a cache, it fundamentally differs from what is explained above. Yet one should notice, that the goal, as well as the effects, are to some degree the same, leading to no misuse of the term "prefetching".

2 The BrainScaleS-2 setup

2.1 Overview

The *BrainScaleS-2* (*BSS-2*) setup, in terms of hardware, consists out of one *HICANN-X* (high input count analog neural network) chip, which is the latest neuromorphic chip developed by the *Electronic Vision(s) group* at the University of Heidelberg, and a field-programmable gate array (FPGA), which provides the I/O interface between a host and the *HICANN-X*.

The gem of the whole setup is the analogue hardware on the *HICANN-X* which tries to emulate the functioning of 512 neurons and $512 \times 256 = 131072$ synapses. The neurons can be connected via the synapses, forming complex neuromorphic networks into which initial spikes can be injected and from which spikes can be read. A synapse is furthermore adjustable by setting its weight, determining how strongly the post-synapitc neuron is affected. The change of the synapse-weights, resulting in a better functioning neuromorphic network can be compared with learning.

The neuromorphic network as well as the synapse-weights need to be configured before an experiment. The configuration is done via the FPGA, on which a playback program is loaded by the host. The playback program contains all the information needed to configure the *HICANN-X* plus that of the initial spikes. Measured spikes and other data travels back from the *HICANN-X* to the FPGA and will be sent to the host as soon as the experiment has finished. [1]

2.2 A closer look

In this subsection an abstract insight into the FPGA and the *HICANN-X* is given, to understand the intention of the work presented in this report.
Figure 1 aims to display the mentioned abstraction. The goal is not to explain the the whole *BSS-2* setup, but rather to understand some of the data streams.

When a playback program is uploaded to the FPGA by the host, it passes through several modules which allow host to FPGA as well as FPGA to host communication and are represented here by the NET-module. The up to multiple megabytes large playback program is then buffered by the AXI (advanced extensible interface) bus fabric making use of the DRAM (dynamic random-access memory). The executor grabs the playback program out of the DRAM in packages and executes them by decoding the data, maybe modifying it and sending it on, into the right direction.
The L2-module instantiated on the FPGA and the *HICANN-X* enables back and forth communication between the two latter.
Configuration data is not sent directly from the executor via the L2 to the *HICANN-X* but rather to an Omnibus fabric, allowing the data to be stored in an SRAM block or sent to other slaves of the bus, including the L2. When arriving on the *HICANN-X*, the configuration data is again fed into an Omnibus fabric and sent to slaves of the bus, one of them being the PPU (Plasticity Processor unit), another one the SynRAM (synapse RAM).
The PPU is a microprocessor, allowing plasticity, the change of synapse-weights during an experiment. Yet one needs to be careful, because the in figure 1 shown PPU block actually includes not only the microprocessor itself, but e.g. also a 16 kiB SRAM block, connected with the latter via another Omnibus. While configuring the *HICANN-X*, instructions and data are written to this SRAM, which the PPU will process while the experiment is running.
The synapse-weights are saved in the SynRAM, which is configured by the FPGA in the beginning, but can be accessed by the PPU as well as the FPGA during an experiment.
Event data, meaning the data of initial and recorded spikes, will be sent and received by the executor to and from the *HICANN-X* directly via the L2. [3]

The for the work in this report crucial aspect is the ability of the PPU to access the SRAM on the FPGA. This means the PPU features 16 kiB of local memory and an external memory, by the size of the on-FPGA-SRAM (besides e.g. SynRAM). In a future FPGA configuration it is the goal

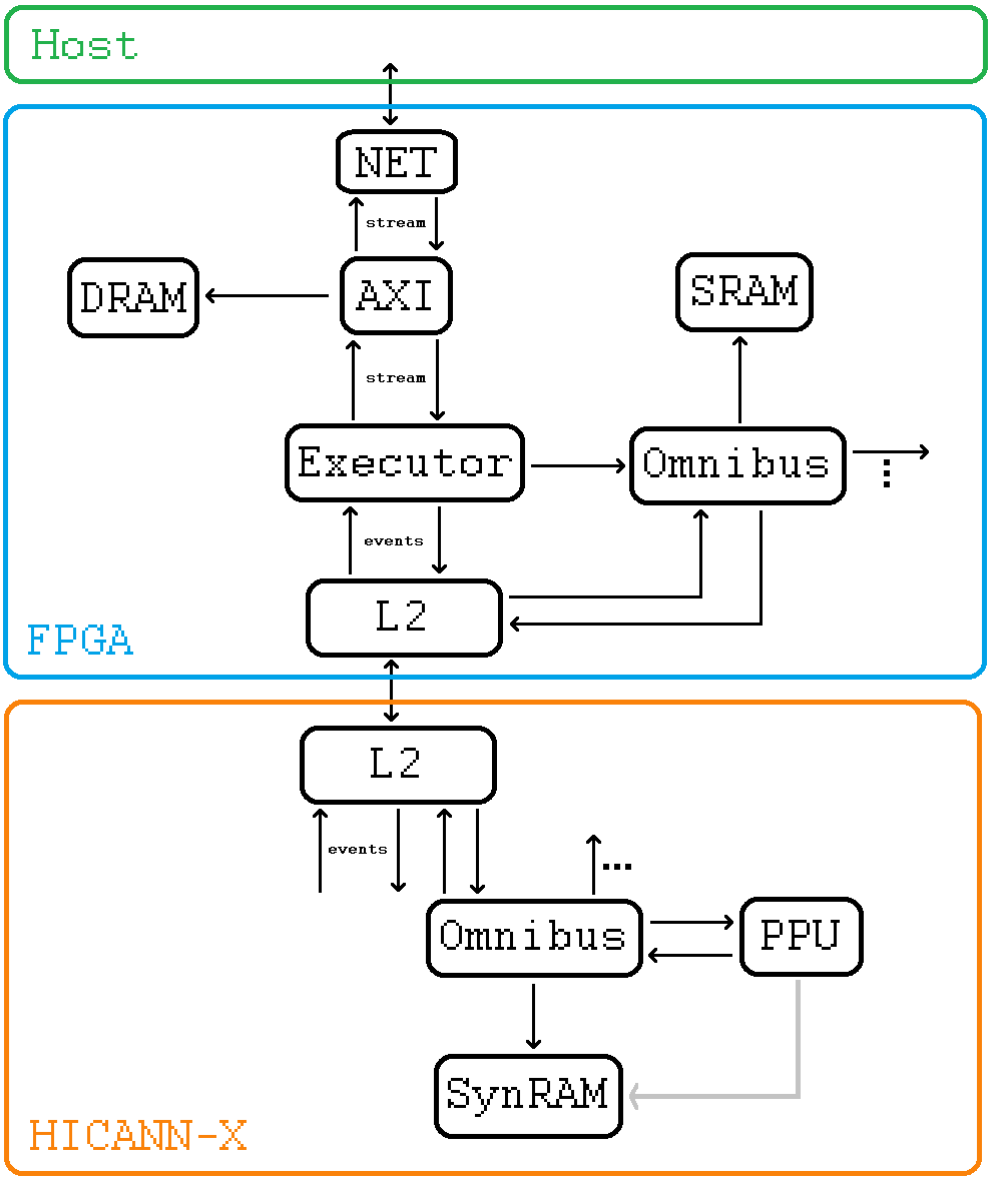

Figure 1: Abstraction of the components inside the FPGA and *HICANN-X* important for this report

to replace the role of the on-FPGA-SRAM with a part of the DRAM, making even more memory accessible for the PPU.

In reality, two PPUs exist, resulting in everything discussed in section 3 to be present twice. Yet this is not really important and will not be mentioned again.

2.3 The inconvenience

A disadvantage of the external memory is the high latency caused by the long pathway through several modules.

The PPU "is a 32 bit scalar, in-order issue, and out-of-order retire RISC [(reduced instruction set computer)] processor"[2]. What this means is that it will issue instructions in the exact order they were given, but can retire, another word for the receiving of the response (e.g. the read data of a READ), instructions in a different order. As a result, the PPU generally does not need to wait for the response of an instruction. But of course there is a limit of instruction-responses, the PPU can wait

[1]The term "word" refers to 32 bit, since the PPU is a 32 bit scalar processor.

for, while issuing new ones.
When writing e.g. a loop in C++ to copy words[1] from the on-FPGA-SRAM into some on-*HICANN-X*-memory, this code will be compiled into instructions. The compiler decides, whether to accomplish the task by issuing all READs first and then all WRITEs, or by issuing a READ and a WRITE followed by a JUMP-instruction, resulting in an actual loop. In the first case (which will occur up until 7 word READs), the PPU can issue all READs without waiting for a response. Only when trying to issue the first WRITE, it needs to wait for the response of the first READ, containing the data to be written. In the second case however, the PPU needs to wait for the response-data of a READ before issuing the next READ, as a WRITE follows its related READ directly. This is the reason for the high latency becoming visible while copying more than a hand full of words from the external memory, being an inconvenience.

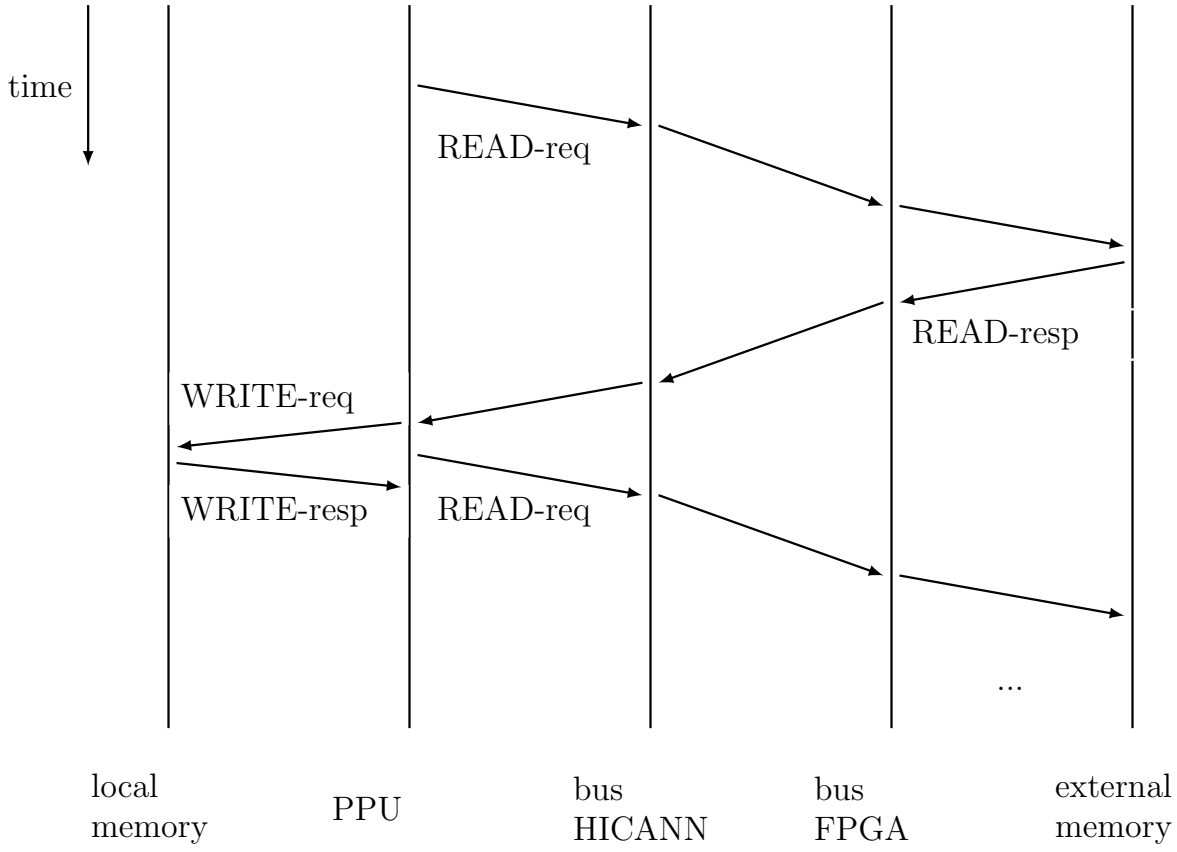

Figure 2: Data flow of copies from external into local memory

During this report the second case, depicted in figure 2, will always be assumed to happen, as the prefetcher is designed for copying up to several kilobytes from the external memory into any memory on the *HICANN-X*.

To show the time scale of the inconvenience, the request and response data of a bus interface (`dmem_bus_if`), close to the PPU was looked upon.

	request					response		
MClk	MCmd	MAddr	MData	MByteEn	SCmdAccept	MRespAccept	SResp	SData

Table 1: The implemented bus interface, sorted into request and response

Table 1 shows the components of the implemented bus interface. The M stands for master, who commands over the slave, represented by the S. The master requests a READ onto the slave by `MCmd`

depicting `RD` and a WRITE by `MCmd` depicting `WR`. With `SCmdAccept` the slave accepts the request. The other way around, a response from the slave, during which `SResp` displays `DVA`, is accepted by the master via `MRespAccept`[2]. [2]
Now it is important to keep in mind, that the bus interface which is observed, does not directly connect the PPU and the external SRAM, as several other modules, connected with other interfaces lie between them.
In the case of an external WRITE, the response of the actual SRAM is not really necessary, so a "fake" response is issued in the data-pathway close to the PPU, which can be seen in figure 3.
This can not be done, when requesting a READ, as `SData` in the response of the SRAM needs to reach the PPU. If followed by a correlated WRITE, the latency between the PPU and the SRAM becomes visible, as shown in figure 4.

Table 2 lists some addresses of the on *HICANN-X* Omnibus for better understanding of the figures. (In reality there are several buses on the *HICANN-X*. The one referred to here is not shown in figure 1, as it is contained in the PPU block)

	byte aligned address	word aligned address
SRAM on *HICANN-X*	0 - 16383 ≙ 0x0 - 0x3FFF	0 - 4095 ≙ 0x0 - 0xFFF
Synapse-RAM	5242880 et seq. ≙ 0x500000 et seq.	1310720 et seq. ≙ 0x140000 et seq.
SRAM on FPGA	1073741824 et seq. ≙ 0x40000000 et seq.	268435456 et seq. ≙ 0x10000000 et seq.

Table 2: On *HICANN-X* Omnibus addresses

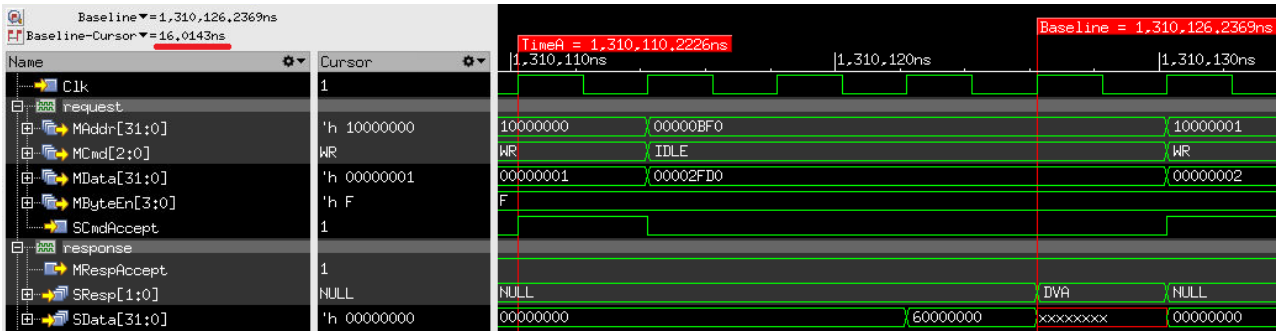

Figure 3: Request and response of an external memory WRITE of the PPU, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

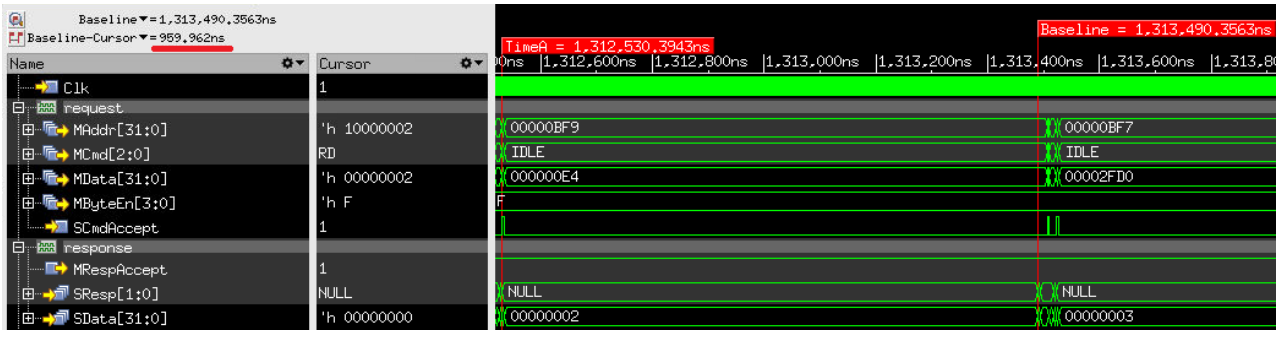

Figure 4: Request and response of an external memory READ of the PPU, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

During a simulation of the *BSS-2* setup, at the bus interface observed, the time it takes for a response to follow a request in case of an external memory WRITE was measured to be about 16 ns

[2]When looking closely at the figures 3 to 6 , one notices that `MRespAccept` is always high, which is due to a slightly different usage of the latter here.

("Cursor-Baseline" top left in figure 3) whereas regarding an external READ, it took about 0.96 µs. The requests and responses are marked with the red cursors.
For comparison, the same measurement was done on the local memory, leading to an about 12 ns interval for a local memory WRITE as well as a local memory READ.

This leaves us with the conclusion, that when having to wait for responses, an external memory READ is approximately 80 times slower than a local one.

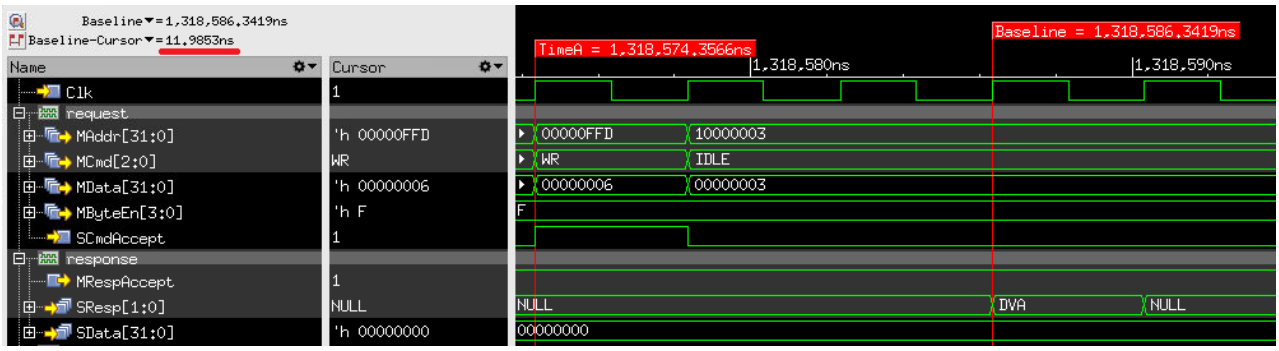

Figure 5: Request and response of a local memory WRITE of the PPU, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

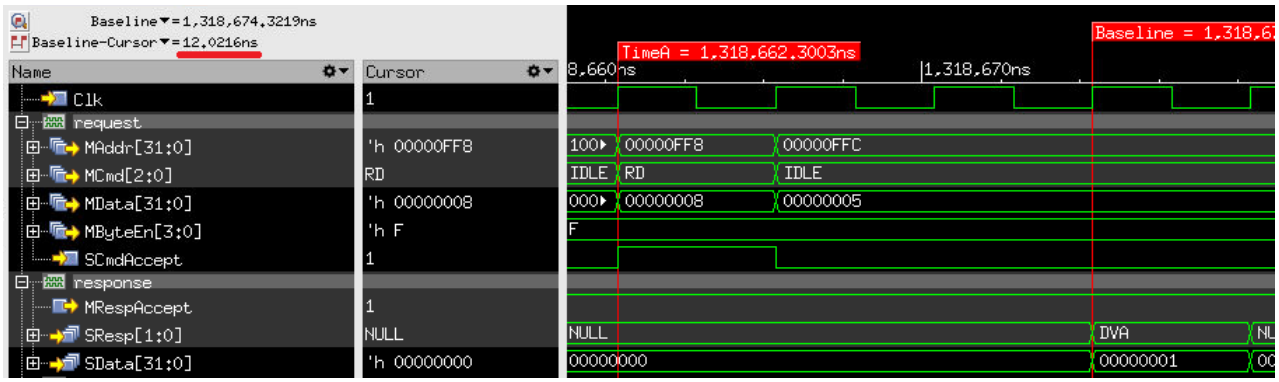

Figure 6: Request and response of a local memory READ of the PPU, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

It should be mentioned, that the measurements presented here were the best ones observed. Due to e.g. bus traffic, the actual times between requests and responses differ.

3 The prefetcher

To overcome the inconvenience of high latency in the event of several external memory READs, the task was to implement a prefetcher somewhere in the PPU to on-FPGA-SRAM data-pathway.
The prefetcher was written in the hardware describing language *SystemVerilog*.

3.1 Implementation location

Since the FPGA is a reprogrammable device, while the *HICANN-X* being an application-specific integrated circuit (ASIC), specifically synthesised by order of the *Electronic Vision(s) group*, the prefetcher was implemented on the the FPGA.

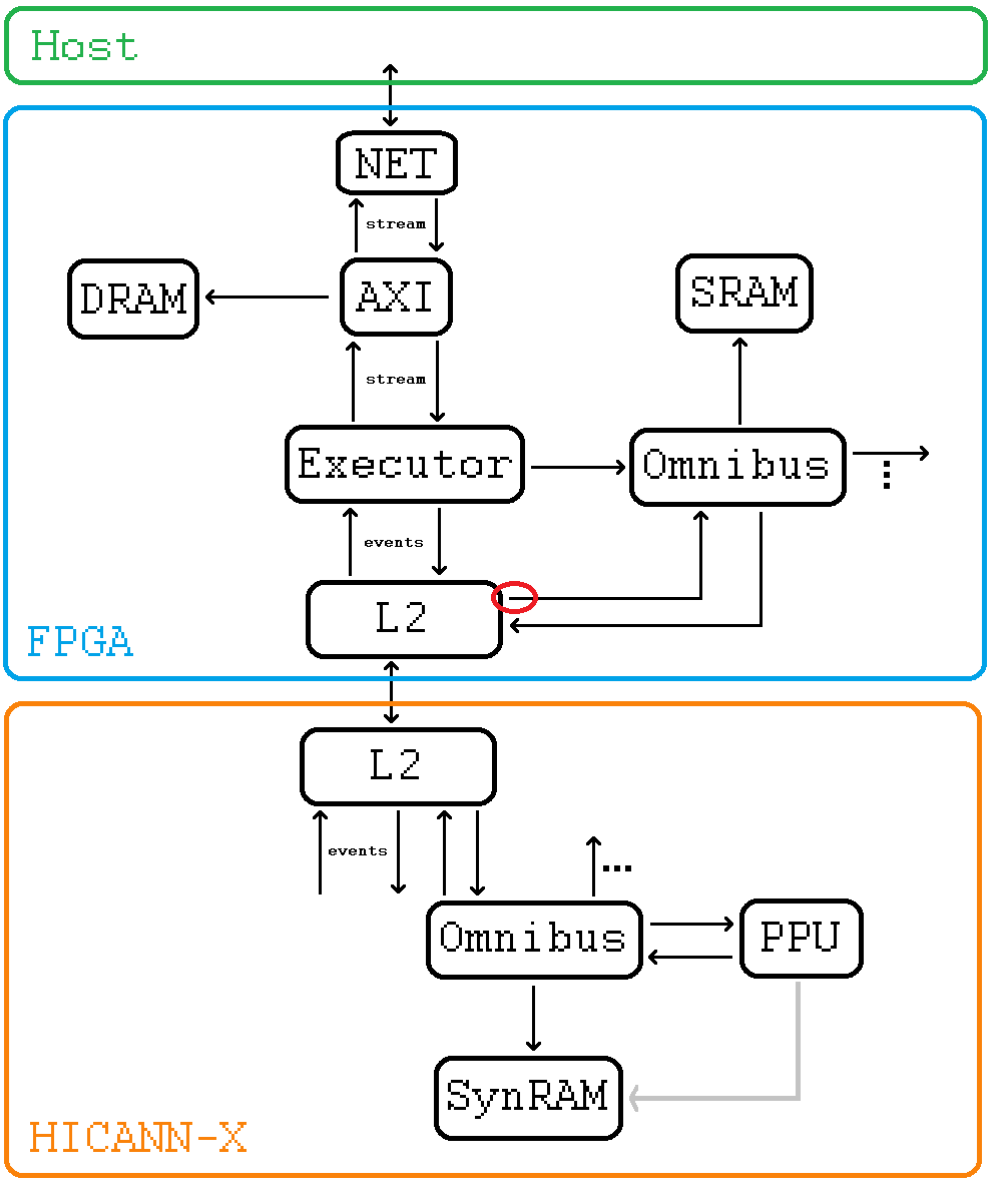

Figure 7: Prefetcher location

The red ellipse in figure 8 indicates, the rough location of the prefetcher.
There, three FIFO interfaces from the L2 are converted into an Omnibus interface.
A FIFO-structure (first in, first out) or just FIFO is a usually small data structure, from which one can only read data in the same order, it was written to the FIFO.
Table 3 shows the interfaces of a FIFO. When wanting to push something into a FIFO, `push` needs to

Write interface			Read interface	
clock_in	→		←	clock_out
push	→		←	pop
data_in	→	**FIFO**	→	data_out
half_full	←		→	almost_empty
almost_full	←		→	empty
full	←			

Table 3: Write and read interface of a FIFO

be high and the data present at `data_in` during the same clock circle. The same applies for poping data out of a FIFO. The `full` and `empty` signals are self explanatory. It is very important, never to push into a full or pop an empty FIFO, as errors will occur.
In case of an asymmetric FIFO, the widths of `data_in` and `data_out` may differ (e.g. 4 bit to 8 bit). FIFOs are used for data buffering, data flow control, but also the crossing of clock domains.

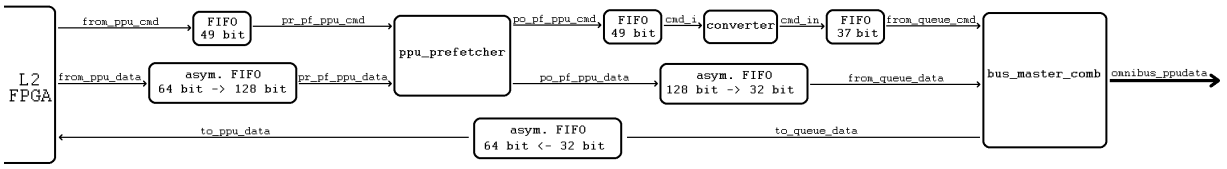

Figure 8: Surrounding modules of the prefetcher

Figure **??** shows the modules involved in converting the three FIFO interfaces `from_ppu_cmd`, `from_ppu_data` and `to_ppu_data` into the Omnibus interface `omnibus_ppudata`, as well as the prefetcher, which was implemented before the conversion. The arrows of the FIFO interface are drawn into data flow direction, while for the Omnibus interface (thicker arrow), it points in master direction.
It should be clear, that the bus interface discussed in subsection 2.3, which is actually the same as `omnibus_ppudata`, was somehow converted into the three FIFO interfaces along the data-pathway.
The numbers of bits inside the FIFOs describe the width of `data_in` and/or `data_out`.
This means the data of `from_ppu_cmd` is 49 bit wide, the most significant bit being a write-not-read-bit (high for WRITEs, low for READs), followed by 16 byte-enable-bits and a 32 bit long, 128 bit aligned address. The crucial step of the conversion in figure 8 is the transformation of this address into a 32 bit long word aligned address, happening in the converter-module (which does not really exist).
The alignment of the address in `from_ppu_cmd` would indicate, that the data of `from_ppu_data` contains 128 bits or four words. However this is not the case, as the 128 bits related to a `from_ppu_cmd` are sent in two 64 bit slices. This also applies to `to_ppu_data`, through which the response data of a READ is transported back. To simplify the design of the prefetcher, the asymmetric FIFO from 64 to 128 bit was included.
The reason for the data of e.g. `pr_pf_ppu_data` (pre-prefetch ppu data) being 128 bit wide is not directly related to the work of this report and will not be discussed, but one should remember from the bus interface, that only up to 4 bytes are written to or read from memory[3]. This just means that the 32 bits of `MData`(`SData`) are located in one of the four 32 bit blocks of `pr_pf_ppu_data`(`to_ppu_data`, transported in two 64 bit slices).

Important for the design of the prefetcher is the fact that it is implemented between two pairs of FIFOs, one of them transporting a command with a 128 bit aligned address and the other one the corresponding 128 bits of data, given the command is a WRITE.

[3]Generally the PPU also features a vector unit, which allows instructions to operate on several data words, but was not used (see section 6).

3.2 Functionality of the Prefetcher

The central idea of the prefetcher-module is that it will issue a number of READs, based on information in WRITE-data of two WRITEs to specific addresses.
The WRITE-commands containing these addresses are hereinafter called "magic commands" and the WRITEs (meaning WRITE-command plus WRITE-data) "magic WRITEs".
After these two magic WRITEs, the already issued READs arrive at the prefetcher and are swallowed.
The issuing in advance and swallowing of a READ can be summarized here by prefetching a READ.
Now the difference to a prefetching method including a cache becomes clearer. In the latter data is automatically prefetched into the cache, while here one always has to actively prefetch data by issuing magic WRITEs. The prefetched data is then not stored in a cache, but rather filled into buffers (like FIFOs) on the data-pathway to the PPU.

WRITE-data of the the first magic WRITE contains a byte aligned address, which defines the address of the first READ to be prefetched. In the current version of the prefetcher this address needs to be dividable by four without leaving a remainder.
How many READs to prefetch is defined by the content of WRITE-data of the second magic WRITE, containing the number of bytes to be read. Since the current version of the prefetcher only prefetches 32 bit READs, this number is divided by four, rounding up, to determine the actual number of 32 bit READs to issue and to swallow.
The prefetcher itself has two counters, holding track of the number of 32 bit READs to issue (issue-counter, which actually counts in bytes) and the number of 32 bit READs to swallow (dump-counter), as well as a register for the address of the READ to be issued next (address-register).

The last paragraph of the previous subsection leads to the conclusion that the prefetcher is able to pop and push into the FIFOs attached to it, thereby controlling the data flow. Of course it never does this if one of the FIFOs is empty or full, respectively, which has to be kept in mind while reading on.

The functioning of the prefetcher is best described by listing the different scenarios that can occur:

- Detection of a normal WRITE:
  If a WRITE-command, not being a magic command is detected, both the WRITE-command and WRITE-data are poped out of their FIFOs and pushed into the subsequent ones. The WRITE passes the prefetcher without any changes.

- Detection of a normal READ:
  If a READ-command is detected, while no already issued READs are to be swallowed, so the value of the dump-counter equals zero, it is poped and pushed.

- Detection of the first magic WRITE:
  When detecting the first magic command, WRITE-data is written to the address-register of the prefetcher, if the value of the dump-counter equals zero. Either way, both WRITE-command and WRITE-data are poped but not pushed, thus swallowed.

- Detection of the second magic WRITE:
  When detecting the second magic command while the value of the dump-counter equals zero, WRITE-data is directly written to the issue-counter and also divided by four, rounding up, and written to the dump-counter. Again both WRITE-command and WRITE-data are poped but not pushed, thus swallowed.

- Issuing READs:
  When the value of the issue-counter is larger than zero, a READ is issued. The artificially created READ-command contains the byte aligned address in the address-register divided by 16, turning it into a 128 bit aligned address. This is fine because all byte-enable-bits are set high since the PPU later on knows, which of the four 32 bit blocks to look at.

The READ-command is pushed into the subsequent FIFO and the value of the issue-counter is reduced by four or set to zero, if it was already smaller than four, as the issue-counter counts in bytes. The address-register, being a byte aligned address is raised by four.

- Swallowing already issued READs:
  If a READ-command is detected, while the value of the dump-counter is larger than zero, it is swallowed, meaning poped but not pushed.
  The dump-counter is reduced by one.
  This will happen in parallel to the issuing of READs.

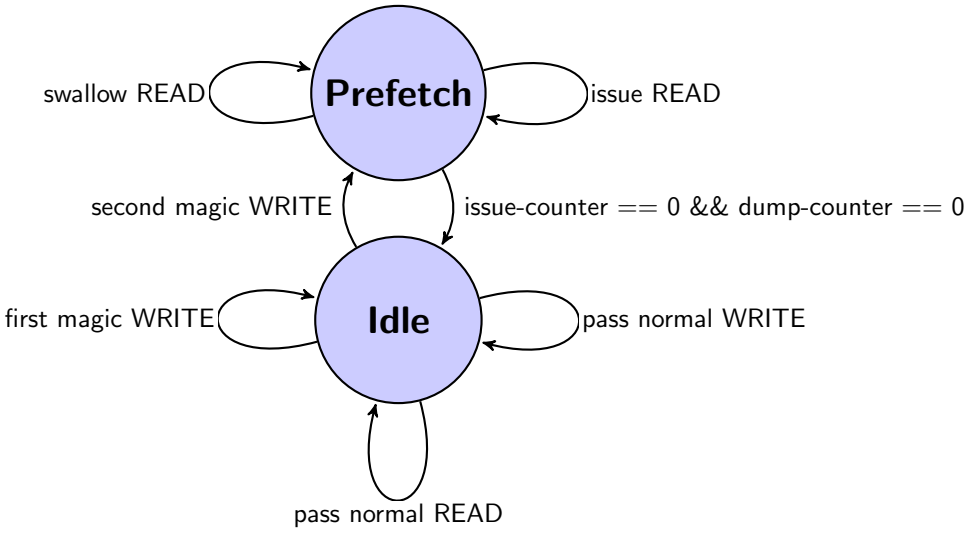

Figure 9: The prefetcher as a finite state machine (FSM)

In figure 9 the operation of the prefetcher as a finite state machine (FSM) is show. It can basically be in two states, the Idle state, in which normal READs and WRITEs are passed through, and the Prefetch state in which READs are issued and swallowed.
Using multiplexers, figure 10 depicts the command and data flow control by the prefetcher. Based on the incoming command and data, the multiplexers are switched and artificial commands may be created.

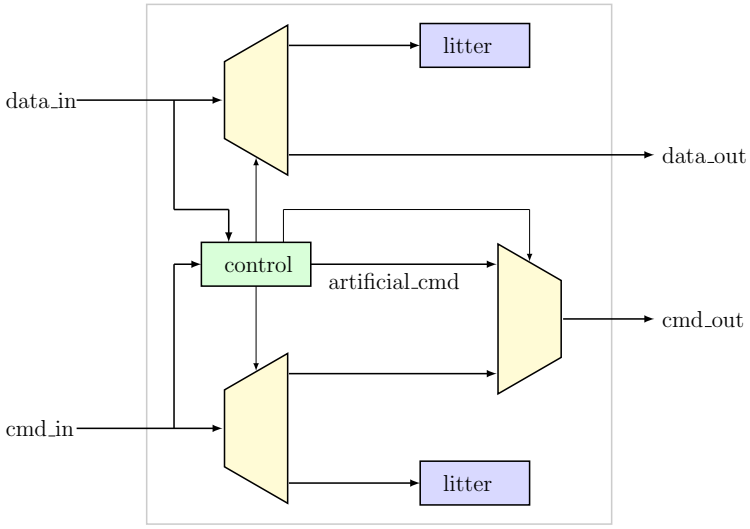

Figure 10: Command and data control of the prefetcher

Remembering figure 2, the data flow while using the prefetcher is shown in figure 11.
The magic WRITE requests reach the prefetcher, are swallowed and trigger the latter to issue READ

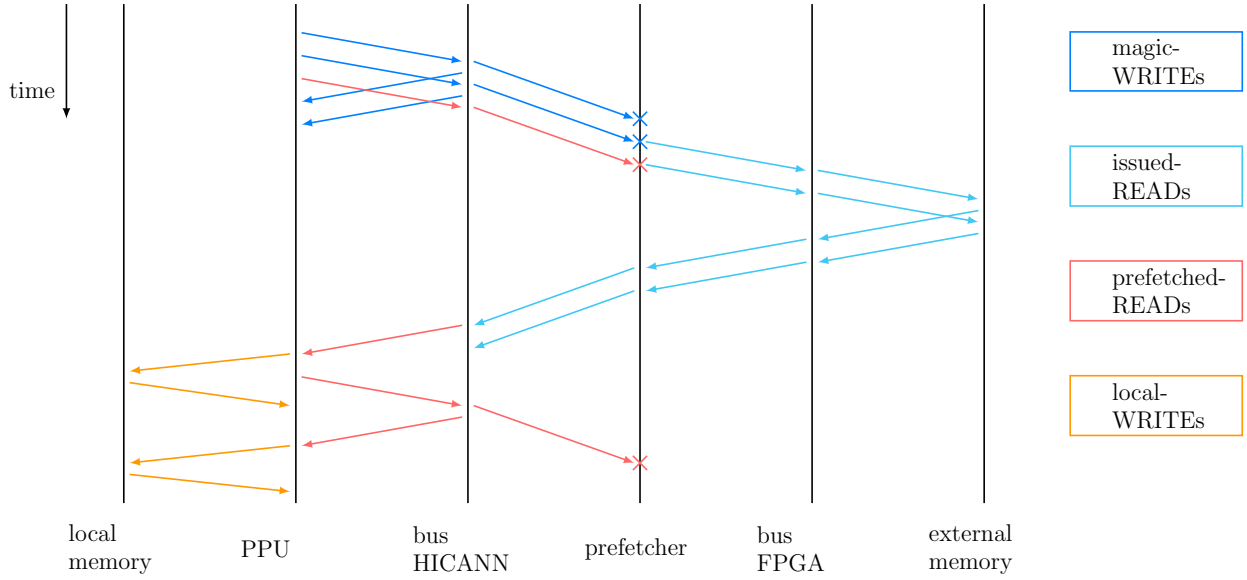

Figure 11: Data flow of prefetched copies from external into local memory

requests. These issued READ requests travel on resulting in READ responses which travel back towards the PPU. The actual READ requests, issued by the PPU, are swallowed by the prefetcher, as it has already issued them itself.
It can be seen that the latency is still visible in case of the first READ. But when looking at the second READ, the READ request, issued by the PPU, is followed by an immediate READ response, as the latter was stored in a buffer close to the PPU. This shows the fundamental principle of the prefetcher.

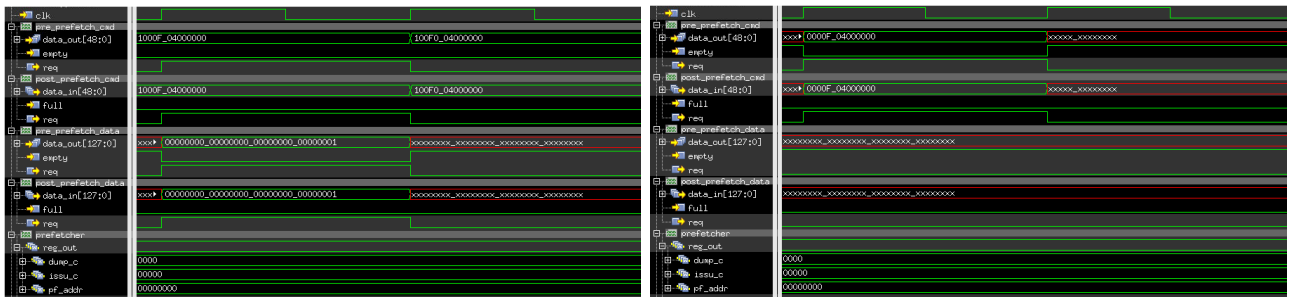

Figure 12: Reaction of prefetcher to a normal WRITE (left) and normal READ (right)

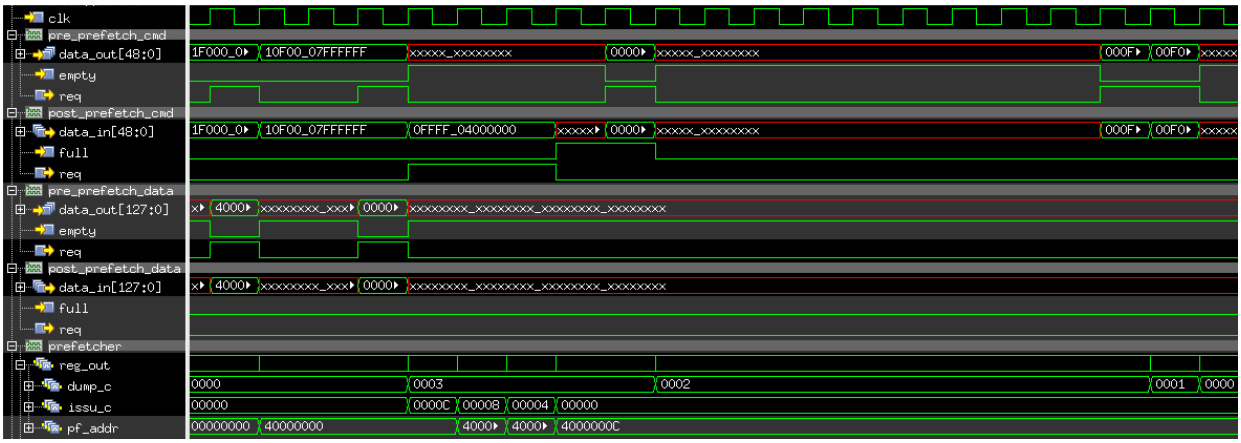

Figure 13: Prefetcher prefetching

In figure 12, one can see the prefetcher passing a WRITE and a READ not related to a prefetching

process. The latter can be witnessed in figure 13. The first magic WRITE fills the address-register and the second one the counters. The reason for the three prefetched READs, which are swallowed, to be this close is discussed in subsection 2.3.

4 Usage of the prefetcher

To use the prefetcher one needs to know the two magic addresses to write to. The first byte aligned address to prefetch from, contained in WRITE-data of the first magic WRITE, of course needs to be at least the lowest byte aligned address of the SRAM on the FPGA. Further more the number of bytes to prefetch, contained in WRITE-data of the second magic WRITE, is limited. Table 4 lists the current magic addresses for prefetching and the constrains of their WRITE-data contents.

	magic address (byte aligned, word aligned)		WRITE-data
1	2147483644	536870911 $\widehat{=}$ 0x1FFFFFFF	$\geq$ 1073741824 (byte aligned address)
2	2147483640	536870910 $\widehat{=}$ 0x1FFFFFFE	$\leq$ 131072 (bytes to be prefetched)

Table 4: Current magic addresses for prefetching

The following shows an example C++ function, resulting in the correct usage of the prefetcher. The name `memcpy_p` underlines the similarity to the `std::memcpy` function.

```
1  /**
2   * Memory will be copied from src to dst.
3   * ------------------------------------
4   * @param src: ---- word aligned addresses
5   * @param dst: ---- word aligned addresses
6   * @param num: ---- defines number of bytes to copy, will be divided by 4, rounding up,
7   *                  as just words (32 bit) are copied; needs to be <= 131072 (hardware)
8   * @param prefetch: enables prefetching
9   * @return dst: --- is returned
10  */
11 void* memcpy_p(void* dst, void* src, size_t const num, bool const prefetch,
12                uint32_t const extmem_data_base = 1073741824,
13                uint32_t const extmem_size = 131072,
14                uint32_t const magic_addr_1 = 2147483644,
15                uint32_t const magic_addr_2 = 2147483640)
16 {
17     static_assert((sizeof dst == 4) && (sizeof src == 4),
18                   "Only 32-bit words are supported!");
19
20     if (prefetch && (src >= reinterpret_cast<uint32_t*>(extmem_data_base))
21         && src <= reinterpret_cast<uint32_t*>(extmem_data_base + extmem_size - 4))
22     {
23         *(reinterpret_cast<uint32_t*>(magic_addr_1)) = reinterpret_cast<uint32_t>(src);
24         *(reinterpret_cast<uint32_t*>(magic_addr_2)) = num;
25         asm volatile("nop");
26     }
27
28     for (size_t i = 0; i < (num+sizeof(uint32_t)-1)/sizeof(uint32_t); i++)
29     {
30         *(static_cast<uint32_t*>(dst)+i) = *(static_cast<uint32_t*>(src)+i);
31         asm volatile("nop");
32     }
33
34     return dst;
35 }
```

The NOP (no operation) in line 22 after the magic WRITEs is essential to ensure that the compiler does not the READs before the magic WRITEs.
Further more it is crucial, that the number of READs which are issued by the PPU matches the one issued by the prefetcher, based on the information of the second magic WRITE. The for-loop achieves this. Here again the NOP is necessary, because without it, the compiler sometimes changes the order of the READs issued by the PPU, resulting in words being written to wrong addresses, without noticing.

5 Performance of the prefetcher

To test the prefetcher, some values were written into the external memory, then copied onto the SRAM on the *HICANN-X* using `memcpy_p` and afterwards compared to the original values. This was done for up to 2 kiB without any errors.

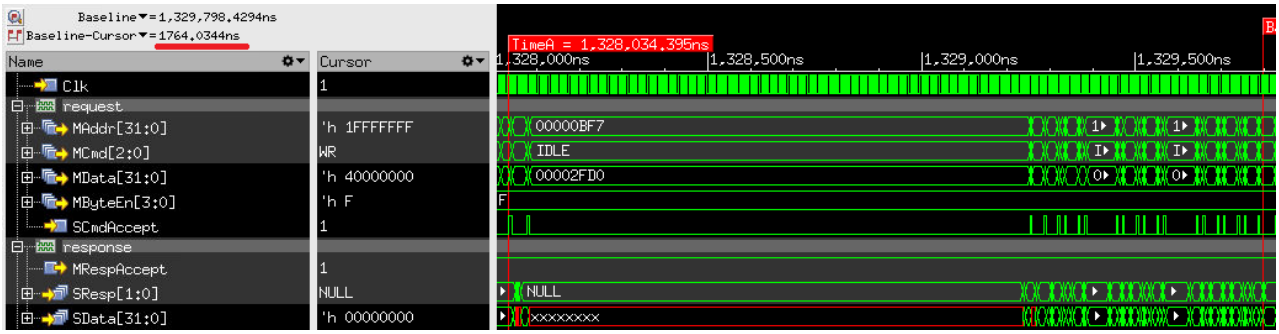

Figure 14: 10 words copied from the on-FPGA-SRAM into the on-*HICANN-X*-SRAM using the prefetcher, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

In figure 14 READs from the external memory and WRITEs into the SRAM on the *HICANN-X*, issued by the PPU because of `memcpy_p` can be seen. Again the Omnibus interface close to the PPU (`dmem_bus_if`) was looked upon.
The time between the request and response of the first READ is measured to be approximately 1.16 µs, which can be seen in figure 15. But this is to be expected, as the READ is issued by the prefetcher just before it swallows the real READ. Because of that it can be compared somewhat to the 0.96 µs measured in the non-prefetching case (figure 4).

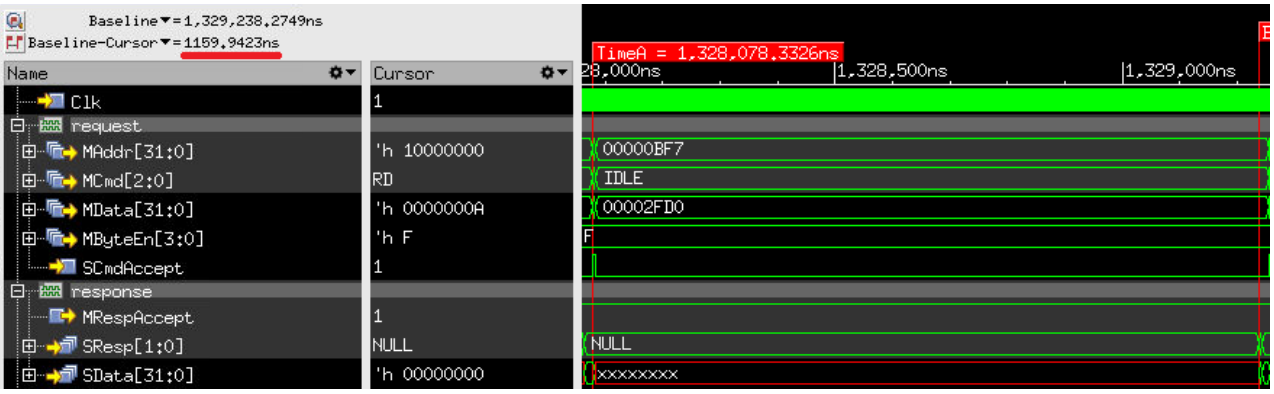

Figure 15: First prefetched external memory word READ, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

Figure 16 depicts the request and response of a prefetched external memory word READ with the shortest time between the two latter. Here the response took only about 16 ns to reach the Omnibus interface, after the request passed through. Here the effect of prefetching becomes visible, as the high latency is immensely reduced because of the response of an external READ being already inside a buffer close to the PPU. A prefetched READ from external memory (not being the first prefetched READ) can actually be as fast an external memory WRITE (figure 3). Yet due to e.g. bus traffic, not all prefetched READs achieve this low latency as can be seen in figure 14.

Just for comparison, figure 17 shows the operating of `memcpy_p` without prefetching (`prefetch = false`). The approximately 10.1 µs this process takes are to some degree expected,

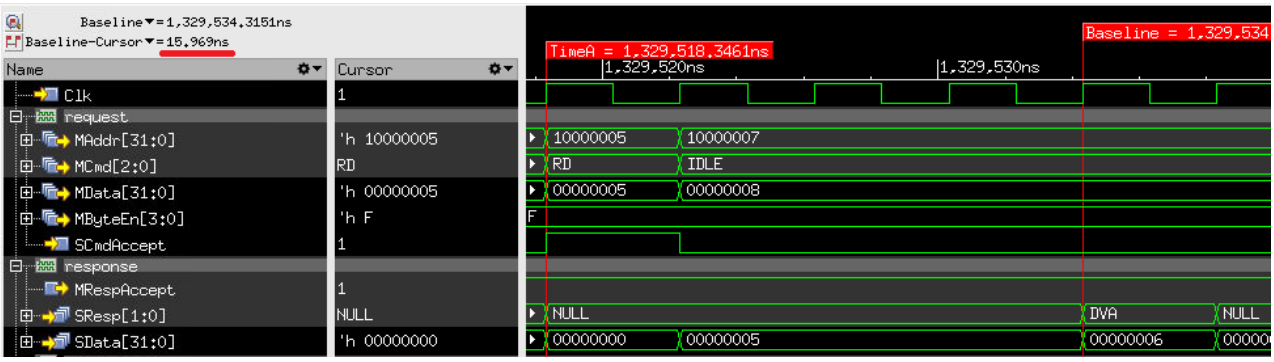

Figure 16: Fastest prefetched external memory word READ, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

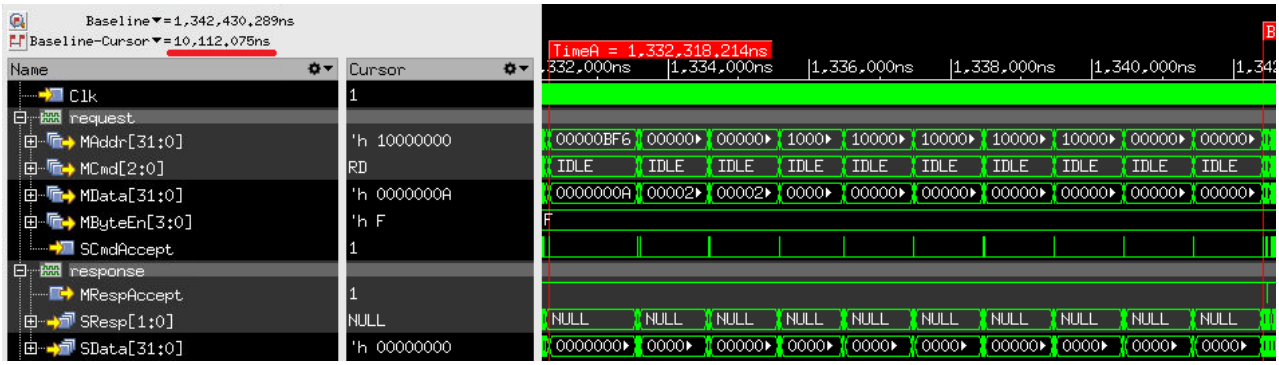

Figure 17: 10 words copied from the on-FPGA-SRAM into the on-*HICANN-X*-SRAM not using the prefetcher, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

remembering the 0.96 µs an non-prefetched external memory READ and the 12 ns a local memory WRITE takes.
For clarity, the 10.1 µs is the time interval between the first READ-request and the last WRITE-response on the level of the observed Omnibus interface. This measured time can be used to determine the bandwidth (bcopy convention [5]), while neglecting the few clocks it takes the first READ-request to travel from the PPU to the Omnibus interface and the last WRITE-response to travel the the other way around.
Doing that also for the measurement using prefetching (figure 14), which took about 1.8 µs (including the magic WRITEs), one obtains a bandwidth of 4.0 MB s^{-1} without and 22.2 MB s^{-1} with the usage of prefetching.

Since the visibility of the latency in case of the first prefetched READ is a constant problem, the efficiency of the prefetching process rises with the number of READs being prefetched.

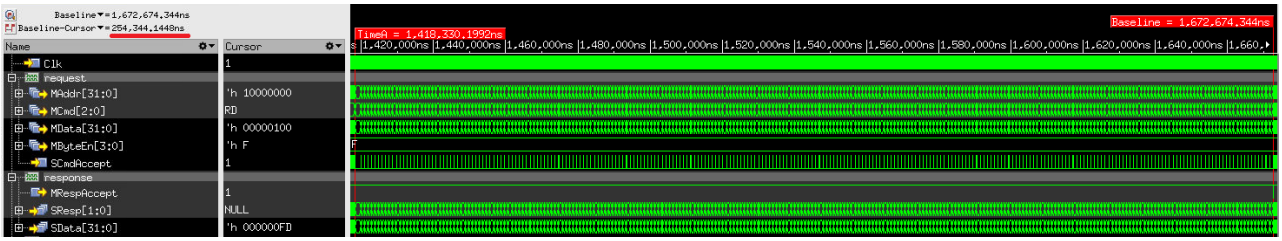

Figure 18: 256 words copied from the on-FPGA-SRAM into the on-*HICANN-X*-SRAM not using the prefetcher, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

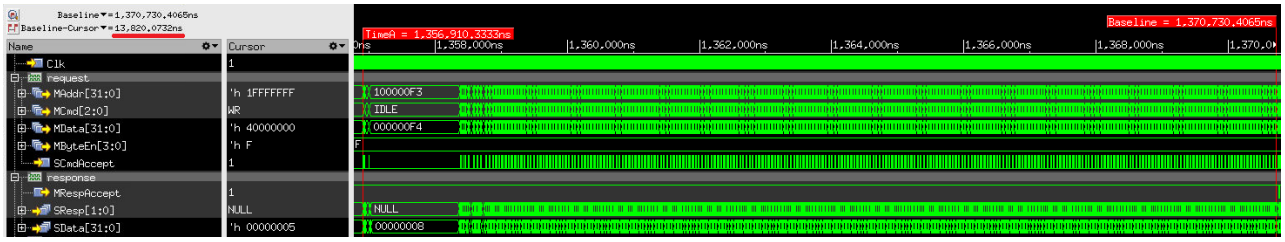

Figure 19: 256 words copied from the on-FPGA-SRAM into the on-*HICANN-X*-SRAM using the prefetcher, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

Figures 18 and 19 display the same scenarios as the figures 17 and 14 with the only difference being the amount of data copied (in words).
The measured time intervals of approximately 254.3 µs and 13.8 µs lead to a bandwidth of 4.0 MB s^{-1} without and 74.2 MB s^{-1} with the usage of prefetching in the case of a 1 kiB data transfer.

non-prefetched copies from external into local memory (at dmem_bus_if)			
data copied	time interval	bandwidth	times faster than 4.0 MB s^{-1}
40 B	10.1 µs	4.0 MB s^{-1}	1.0
1024 B	254.3 µs	4.0 MB s^{-1}	1.0
prefetched copies from external into local memory (at dmem_bus_if)			
data copied	time interval	bandwidth	times faster than 4.0 MB s^{-1}
40 B	1.8 µs	22.2 MB s^{-1}	5.6
400 B	6.5 µs	61.5 MB s^{-1}	15.4
1024 B	13.8 µs	74.2 MB s^{-1}	18.6
2048 B	26.2 µs	78.2 MB s^{-1}	19.6
(4096 B	51.0 µs	80.3 MB s^{-1}	20.1)
copies from local into local memory (at dmem_bus_if)			
data copied	time interval	bandwidth	times faster than 4.0 MB s^{-1}
40 B	0.45 µs	88.9 MB s^{-1}	22.2
400 B	4.4 µs	90.9 MB s^{-1}	22.7
1024 B	11.3 µs	90.6 MB s^{-1}	22.7
WRITEs into the local memory of the PPU by the executor (at omnibus_from_l2)			
data copied	time interval	bandwidth	times faster than 4.0 MB s^{-1}
3252 B	23.6 µs	137.8 MB s^{-1}	34.4

Table 5: Comparison of bandwidths measured during simulations

Table 5 lists some measured bandwidths in case of different amounts of data copied from external memory into local memory without and with using the prefetcher, as well as copying data from local into local memory. Furthermore the last block shows a measurement of the bandwidth reached by the executor writing into the local memory of the PPU.
The last line of the second block is bracketed, as the simulation of this measurement was only confirmed by eye in the simulator. None of the values have an error, since it is not possible to measure the statistical fluctuation of the bandwidth in simulation. Because of that all values presented are to be interpreted as optima.
The differences between the bandwidths measured during the local memory copies is caused by bus traffic.
Conclusively it can be said, that for more than 2 kiB the prefetcher allows copies from external into local memory to be nearly as fast as local memory copies.
However the bandwidth witnessed during WRITEs by the executor into the local memory of the PPU can not be reached, as it also surpasses the one in case of local memory copies.

To underline the fact, that prefetching is possible into any on-*HICANN-X* memory, figure 20 shows 256 word copies from external memory into SynRAM. Here the copy process was with 13.9 µs actually slightly faster as in figure 19. However reason for that is once again bus traffic.

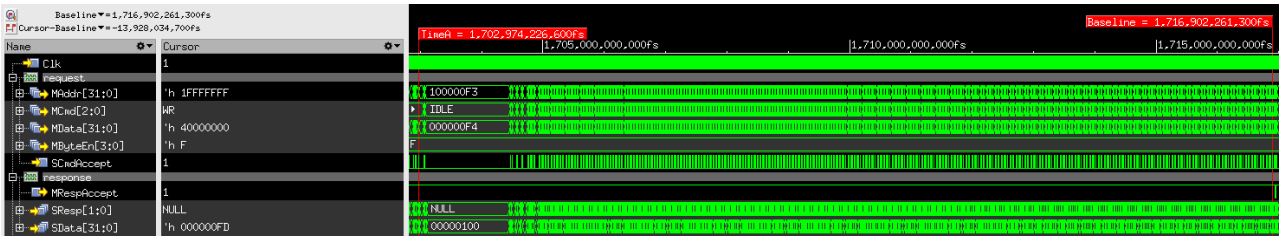

Figure 20: 256 words copied from the on-FPGA-SRAM into SynRAM using the prefetcher, measured at a bus interface on the *HICANN-X* close to the PPU during a simulation

6 Discussion and Outlook

The *BSS-2* setup, consisting of the *HICANN-X* and an FPGA, is constantly evolving towards a faster, more efficient system.
With the goal to slightly assist this process, the prefetcher-module presented in this report was designed. It allows the PPU on the *HICANN-X* to prefetch data from an external memory located on the FPGA.
Prefetching is initialised by two magic WRITEs to the external memory, which can be hidden in software by programming e.g. a C++ function as shown in section 4.
The bandwidth when copying data from external to local memory (on-*HICANN-X*-SRAM) increases from about $4.0\,\mathrm{MB\,s^{-1}}$, by five times for 40 B copies to $22.2\,\mathrm{MB\,s^{-1}}$ and up to twenty times for 2048 B copies to $78.2\,\mathrm{MB\,s^{-1}}$, nearly reaching the bandwidth measured during local memory copies of about $90.6\,\mathrm{MB\,s^{-1}}$.

Even though no data cache is used, the prefetching here can somewhat be compared to cache-prefetching, with a miss occurring each time searching in the cache for data. The data would then need to be read from the actual external memory, similar to the first READ prefetched with the prefetcher. Depending on the strategy the cache follows, it would read and store the data of addresses subsequently to the one at which the miss occurred, resulting in cache hits after the one miss in the beginning and leading to a similar data flow as observed while the prefetcher is prefetching. Thinking about it, in context of several READs from external memory, a cache would actually need to contain at least the data of the first couple of addresses, to be more efficient than the prefetcher. Furthermore it would have to start prefetching at least the next 40 words in advance, to prevent further misses, assuming that reading from the cache takes 12 ns (local READ), writing the data somewhere another 12 ns (local WRITE) and knowing a non-prefetched external READ to take about 960 ns.
This makes one argue, that the prefetching method used here might not be such a bad alternative to a standard data cache, the use case being the copying of at least several hundred bytes from external memory.

In subsection 2.3 it was mentioned that the FIFO interfaces coming out of the L2-module are actually made for the transport of four words at a time. Only one of these four word spaces ever being occupied is due to the fact that only the scalar unit of the PPU is used throughout the work presented here. However the PPU indeed has a vector unit, which was designed to accelerate the speed at which weight-data is written into the SynRAM[4] by SIMD (single instruction multiple data) processing. This vector unit can also read from and write to external memory, using all four word spaces.
Knowing this, the obvious next step is to update the prefetcher, allowing not only the scalar, but also the vector unit of the PPU to prefetch READs from external memory.

[4] See broad gray arrow in figure 1.

References

[1] David Stöckel; Timo Wunderlich Eric Müller; Christian Mauch; Philipp Spilger; Oliver Julien Breitwieser; Johann Klähn and Johannes Schemmel. *Extending BrainScaleS OS for BrainScaleS-2*. URL: http://www.kip.uni-heidelberg.de/Veroeffentlichungen/download.php/6482/temp/4050.pdf. (accessed: 16.07.2021).

[2] Simon Friedmann. *A New Approach to Learning in Neuromorphic Hardware*. URL: http://archiv.ub.uni-heidelberg.de/volltextserver/15359/1/Diss_Simon_Friedmann_2013_screen.pdf. (accessed: 19.07.2021).

[3] Marco Rettig. *Characterizing the Event Interface of the HICANN-X*. URL: http://www.kip.uni-heidelberg.de/Veroeffentlichungen/download.php/6336/temp/3903.pdf. (accessed: 18.07.2021).

[4] *WIKIPEDIA: Cache (computing)*. URL: https://en.wikipedia.org/wiki/Cache_(computing). (accessed: 21.07.2021).

[5] *WIKIPEDIA: Memory bandwidth*. URL: https://en.wikipedia.org/wiki/Memory_bandwidth. (accessed: 27.07.2021).