SRAM tests of synapse weight memory on neuromorphic hardware

Nils Bergler

17.10.2024

Spiking neural networks need a synaptic weight memory to function, thus it is crucial that this memory is reliable and fault free. In this work, we investigate the fidelity of the synaptic memory on the neuromorphic BrainScaleS-2 platform. After implementation in software, the performance of different tests at finding faults is evaluated on the synaptic memory of the BrainScaleS-2 platform. Afterwards the impact of SRAM timing configurations on fault occurrence is tested, and finally a calibration created that configures the timings into a fault-free state.

Contents

1 Introduction **3**

2 SRAM Theory **3**
2.1 SRAM Basics and Faults 3
2.2 SRAM Timing 4

3 SRAM test patterns **6**
3.1 Implementation of SRAM tests 7
3.2 Preliminary tests on BSS-2 9
3.3 Comparison of tests on BSS-2 9

4 SRAM Timings **12**
4.1 Impact of SRAM Timing on Faults 12
4.2 Impact of timing configuration on execution time 15

5 Calibration of the SRAM Timings **17**
5.1 Result of the Calibration 18

6 Digital Timing **18**
6.1 Time impact of `wait_ctr_clear` 18
6.2 Impact of `wait_ctr_clear` on read faults 19

7 Hardware Database **20**

8 Discussion **21**
8.1 Summary 21
8.2 Outlook 22

References **22**

9 Appendix **23**

1 Introduction

A BrainScaleS-2 (BSS-2) platform chip is made up of 2×256 neurons that are interconnected by 256×256 synapses each (Pehle et al., 2022). Synapses are defined via a 6-bit weight and address which are used during the execution of Spiking neural networks (SNNs) on the platform. These weights are stored in an onboard memory in the form of a Static Random Access Memory (SRAM). To ensure correct execution of SNNs the weights must be correctly written and read back. The goal of this internship was to create a software library for verification of the synapse memory, investigate the impact of SRAM configuration and fix occurring faults by calibrating the available parameters.

2 SRAM Theory

2.1 SRAM Basics and Faults

A SRAM memory cell is made up of two interlocked CMOS inverters that form a bistable circuit, fit for storing binary data. Each of the inverters is accessed via an NMOS transistor, connecting each to a (respectively inverted) bitline. These access transistors are activated via a wordline, which allows addressing specific cells in a SRAM.

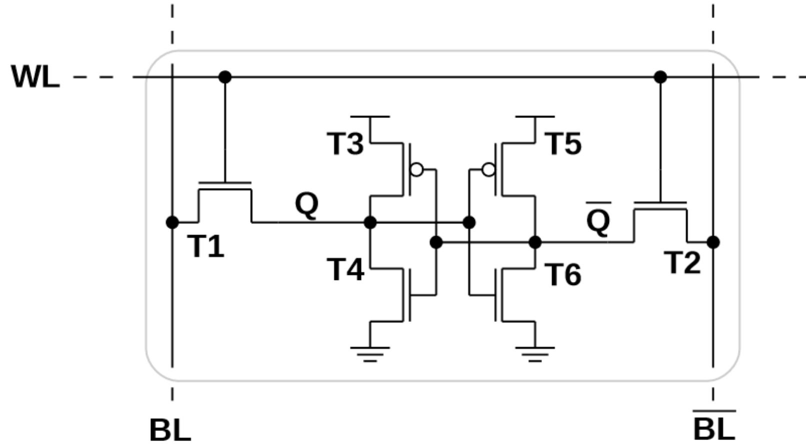


Figure 1: Schematic of the standard SRAM cell. BL is bitline and WL wordline, bars indicate logical inversion. Taken from Hock (2014).

SRAM operation consists of two basic operations:

— Write: To write a value into a cell, the bitlines are driven at the desired logical values, and the wordlines are activated afterwards. This forces the logical values onto the cell and writes the bit. The bitlines are only able to overpower the logical “1” with a “0” if the pull-up transistor of the cell is weaker than the access transistor.

— Read: To read a cell, the bitlines are charged to the supply voltage of the SRAM. After the activation of the wordlines, the logical “0” side of the cell discharges the connected bitline. Then a sense-amplifier detects the difference between the bitlines and outputs the read value. This works only without changing the internal state, if the pull-down of the cell is stronger than the access transistor.

From this we see there are mainly four areas where faults can occur according to Bosio et al. (2012):

1. Cell: If the cell itself is faulty, after a given time, the internal state of the cell could change, leading to a Data Retention Fault (DRF) of the written value.

2. Address Decoder: For the operation, the memory has to select the correct cells via the wordlines, thus the address decoder could lead to faults if either the wrong, multiple, or no cells are selected under a given address leading to faulty read/writes at undesired locations or bad read/write operations if no cell is selected.

3. Write: If the write operations are faulty cells could be stuck at a certain value (Stuck-At Fault) or fail transition form high to low or low to high (Transition Fault). In addition, the write/read of other cells could influence the state of the faulty cell, leading to a coupled fault behavior (Coupled Fault)

4. Read: A faulty read operation could either read incorrect values for a correct cell state (Read Fault) or change the internal state of the cell invalidating subsequent reads of the cell ([deceptive] destructive Read Fault)

Both Write and Read faults can be caused by different components such as the precharge circuits, write drivers and sense amplifiers. This leads to a rich spectrum of fault behavior which often is dynamic in nature and may only produce fails as a consequence of process variations.

2.2 SRAM Timing

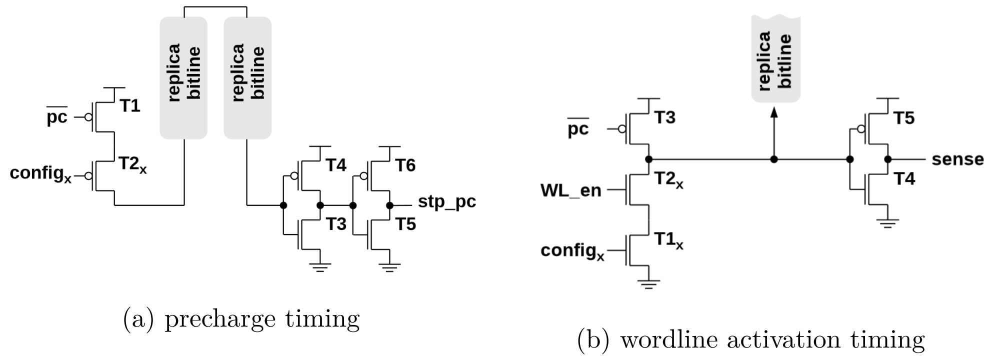
For read/write operations, 3 different timings on the synapse memory can be adjusted (Hock, 2014)

1. Precharge Config (`pconf/pcconf`): The precharge time before a read access ensures fully charged and equalized bitlines before the wordlines are activated to ensure correct sensing of the logical values. This timing is generated by a replica bitline that gets charged by a configurable amount of up to 4 minimum width transistors to simulate the charging dynamics of the bitline. The end of precharge is then determined by an inverter connected to the replica bitline.

2. Wordline Config (`wconf`): The wordline activation timing controls the duration the access transistors are activated during an operation. It must be long enough such that a read operation causes a sufficient voltage difference to be sensed consistently but not change the state of the cell. For a write operation it must be long enough

to switch the state of the cell. This timing is also generated by a replica bitline which in this case is discharged by a configurable amount of up to 8 transistors, to simulate the dynamics of the bitlines from the bitline to the cell. The end of the wordline activation is then sensed by an inverter.

3. Digital Timing Config (`wait_ctr_clear`): The digital timing sets the amount of cycles the digital logic waits after an analog read/write operation has been initiated on the SRAM. This must be long enough such that the read/write operations have completed, before the digital logic proceeds with the next steps. Especially for read operations this ensures that the sense amplifier has reached a well-defined state before the digital logic proceeds to access the value.



(a) precharge timing

(b) wordline activation timing

Figure 2: Schematics of the circuits responsible for generating the timings. The configuration of the timings is applied at $config_x$ to the circuit. Taken from Hock (2014).

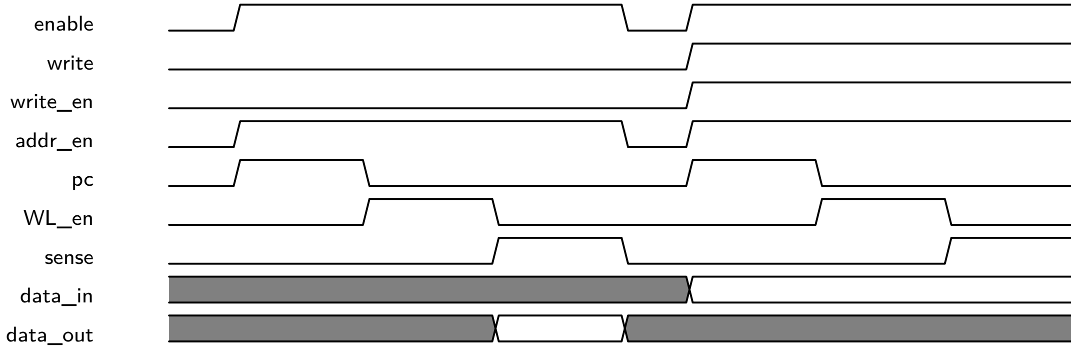


Figure 3: Timing diagram of a read and write operation

The individual transistors can be activated using bitflags contained in 4-bit (`pconf`) or 8–bit (`wconf`) integers, respectively. Therefore, there are multiple configurations that, in theory, produce the same timing, giving the possibility for calibration to account for

variance in the components of the SRAM. The `wait_ctr_clear` setting is set as an 3-bit integer (0-7) indicating the amount of cycles to wait.

3 SRAM test patterns

To detect SRAM faults, tests are used which test for a range of different fault behaviors. The basic concept is almost always to write a certain pattern and/or sequence of values into the memory and then read them back to check if the values read from the memory match with those written. Depending on the pattern and sequence of values different types of faults of varying complexity can be found or identified (some tests can detect a fault but not identify its type). In general, the more complex a fault behavior is, the more complex the test for it has to be (Bosio et al., 2012). The tests I picked for this internship are:

— Zero-One: The memory is written with all “0” and all “1”. This tests the basic functionality of the cells, so it can detect stuck cells, transition faults and some address decoder faults.

— Checkerboard: A alternating pattern of values/bits is written to the Memory. This maximizes any sort of “drainage” between neighboring cells.

— Hammer Method: Instead of writing/reading a value just once it is written/read multiple times in succession. This can uncover dynamic faults which may only appear after multiple operations (e.g., charges building up). This method is applicable to any pattern.

— Value Sweep: This Test considers all possible values for a Synapse written across the memory simultaneously. This test checks the basic functionality of each synapse to store all required values and can detect coupled faults within a synapse, but does not consider crosstalk of cells belonging to different synapses.

— Galpat/Walking-1: For this Test a single high bit is “walked” through the memory while leaving/setting the rest 0. Ideally the write operations only change 2-bits at each step, to detect for faults occurring in different cells as response to the single bit. This test checks for basic crosstalk between cells or in my case synapses. Usually the test is also repeated with an inverted pattern of a “walking 0”.

— Pseudorandom Patterns: Instead of considering specific patterns a sequence of Pseudorandom memory matrices are checked. The Test is Pseudorandom to make the test repeatable. This Method can test for any fault, but is in most cases less efficient at detecting specific faults. Also, this test only can ensure no faults up to a probability dependent on the amount of matrices tested and the complexity of the faults. Nonetheless, is this approach useful as one does not have to take into account/know the type of faults that may occur making it more robust for general validation and is still fit for finding faults given enough time.

Importantly I left out the class of “marching tests”, which consist of sequences of operations applied to the memory cells in specific order (e.g., writing 1 on all cells in rising address order). These are in most cases more efficient in terms of the amount of operations performed but are very inefficient to implement in python as opposed to the strongly optimized array operations available in python. Considering the relatively small size of the SRAM I choose to ignore marching tests.

3.1 Implementation of SRAM tests

The SRAM operations are executed by a FPGA in form of a pre-built Playback-Program composed of basic hardware operations and program control i.e. read/write or barriers. Necessarily “read” operations supply tickets to retrieve the data after execution of the program on the hardware. Specify parts of the hardware are addressed using Coordinate objects and their corresponding data handled as Container objects available as classes in python. For my SRAM tests the synapse memory is addressed using the *SynapseMatrix* containers which stores a matrix of values of the entire synapse memory. Considering these implementations a SRAM test is composed of the following steps:

1. Iterate a series of matrices to be written and add write and read operations to the program according to the used test method
2. Store the expected values and tickets for the read operations
3. Execute the program on the chip and wait until the hardware has finish reading the memory
4. Retrieve the values read by the hardware and compare them with the expected values store previously

My final implementation introduces the following abstract interfaces for testing:

— *RamTest*: This object defines a sequence of memory matrices dynamically dependent on the shape of the memory to be tested. Optional details of the test (e.g., length, values to consider) are supplied at creation.

— *TestMethod*: This object defines the operations to be executed for each memory matrix given as an Iterator.

— *RamTest*: This object represents one test and provides methods for building and evaluating the tests and stores the data for evaluation. The hardware is addressed using general interfaces for writing and reading matrices to/from memory as well as a method to retrieve the read values given the tickets. Furthermore, the shape of the memory is specified as a property which is passed to the used *TestPattern* at build. To build a test the build method takes a *TestPattern* and *TestMethod* and writes the Program into a *PlaybackProgramBuilder*.

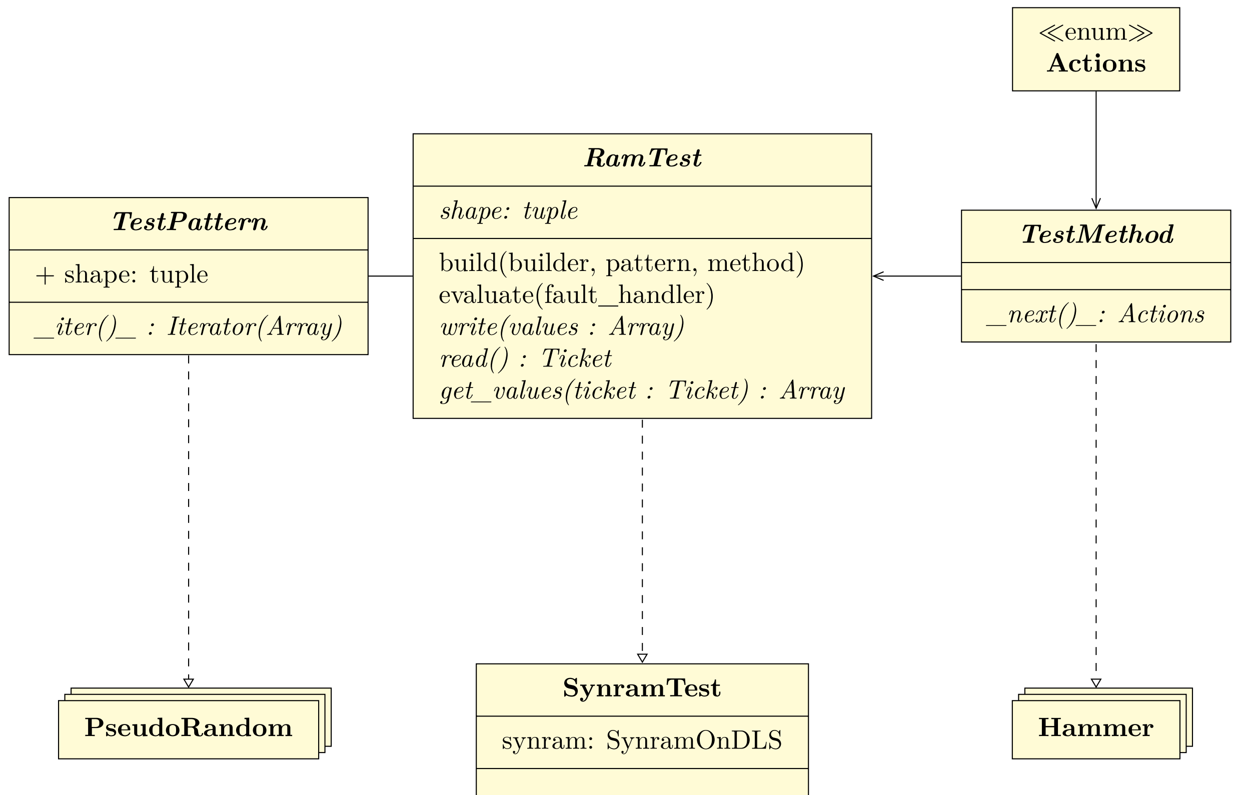


Figure 4: UML diagram of the Implementation. The *Actions* enum categorizes the possible operations *RamTest* supports. Example implementations such as *SynramTest* are shown.

For the synapse memory the *RamTest* interface is implemented using the hardware API described above, and the shape set as a hardware property. The specific memory to be tested (i.e, the hemisphere and weight or label memory) is specified at creation.
The patterns specified in section 3 are implemented using the TestPattern interface. The Hammer method as well as a basic "Write and Read once" method are implemented using the TestMethod interface.

3.2 Preliminary tests on BSS-2

With the first iteration of my code I ran tests on some few setups to test my implementation and make some first observations about the synapse memory. The tests were only run a few times, so these observations don't account for any statistics or dynamic faults. The main takeaways from these preliminary tests are:

- Faults are present but sparse relative to the size of the synapse memory and only appear on few setups. At this point on 3 out of 10 tested hemispheres I found faults on 2 of 5 tested setups and when faults were present they appeared in low numbers (< 10).
- Faults often to occur on only one hemisphere of a chip, so the causes seem to be localized within a hemisphere.
- Faults always occurred in the same column, and within that column most of the time on the same bit for all faulty values. So faults seem to stem from some problem that depends/interact on the bitline.
- Some faults are of dynamic/stochastic nature, as they do not appear in all test runs of the same method and pattern.

This for now comprises no relevant analysis of the faults but will be useful for further investigation of the tests.

3.3 Comparison of tests on BSS-2

For further proceedings I compared the performance of test on a faulty setup to learn about the nature of the faults and the capability of the test to detect those faults. The tests/methods considered are those specified above and run each 10 times. To evaluate the tests I choose the setup on Wafer 70 and FPGA 0 where I had already found faults in Column 110 while running the preliminary tests, including dynamic faults. The total amount of faults occurring in a synapse are added for each test and normalized by the amount of matrices each pattern read.

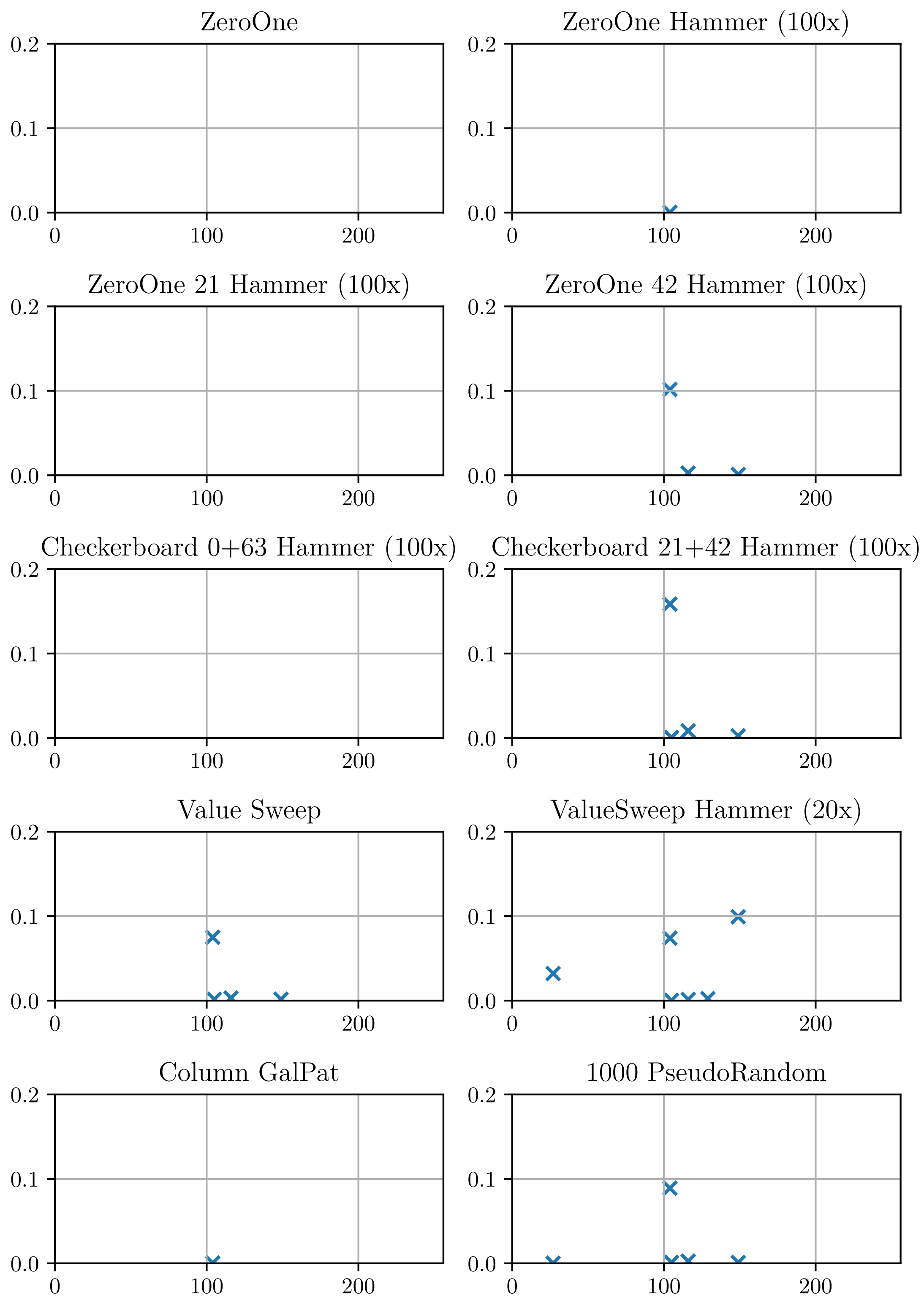


Figure 5: Faults found in Column 110 on one a faulty setup

As shown in fig. 5 the basic functionality of all cells is given by *ZeroOne*, but there are specific values (e.g., 42) which cause faults in synapses, so there seems to be coupled fault behavior for bits in a synapse. The faults found in the *Checkerboard* test are a coincidence for the value 42, because *ValueSweep* as *Hammer* finds more faults which must occur at different values. Furthermore, the *Column GalPat* test shows that there seems to be no significant crosstalk between synapses, as the fault occurred when the synapse that was set high itself. From the *Hammer* Method we can conclude that some faults are dynamic as they don't happen each run of the pattern. The *Column GalPat* pattern shows in this instance also that it seems to be irrelevant if the faulty synapse is accessed during multiple write/read beforehand or only at the faulty read/write, but could also just be a statistical coincidence as the fault was not found every run. Whether the difference between the *Hammer* and the normal execution of *ValueSweep* is due to the hammering or if it's just statistics of testing more matrices for the dynamic faults is not clear so far. Finally, the *PseudoRandom* test performs the best at finding faults with 1000 matrices per run compared to *ValueSweep Hammer* with 1280 matrices per run.
To determine whether the *Hammer* method results in more faults found I ran the *ValueSweep* test with different amounts of hammer 30 times and considered the amount of faults they found on average per run. To compare the different amount of writes/reads. The amount of faults is normalized by the amount of reads per matrix. In total the performed reads differ but if it's a fixed probability the normalization should account for this fact and equalize the average amount

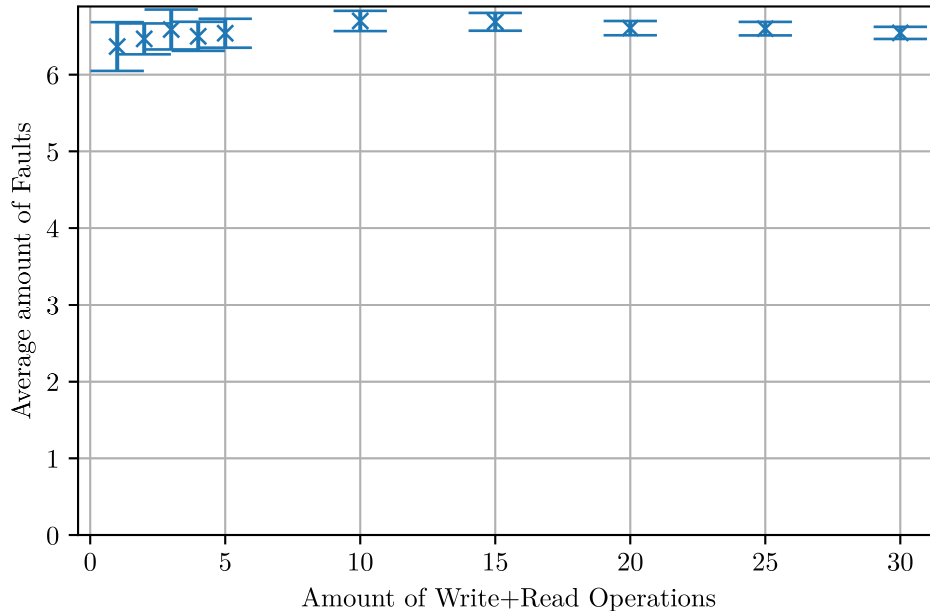


Figure 6: Average amount of faults found by the ValueSweep pattern using different amounts of read/write (hammer) on Column 110 on the same setup as in fig. 5

From fig. 6 it's clear that the amount of hammer makes no significant difference in the amount of faults that are found, in other words it's equivalent to hammer the values or run them multiple times separately. The standard deviation decreases with rising amount of hammer because the total amount of reads increases and the variance between reads of one hammer sequence is not considered. Thus, for further investigations the *Hammer* Method is not considered unless to improve statistics.

4 SRAM Timings

4.1 Impact of SRAM Timing on Faults

In the next step I investigated how the timings `pcconf` and `wconf` influence the amount of faults occurring on the synapse memory. To do this I did a full parameter sweep of pcconf and wconf combined and counted the total amount of faults occurring on the entire array for each setting. As the parameter space is quite large I chose to use the *ValueSweep* test pattern as it is the most effective regrading runtime. As seen in the previous section this pattern does not always find all faults, but for now the presence (or more importantly their total absence) of any fault is more important and so far faults that were not detected by this pattern came along with faults that always were detected. The parameters were both swept from 1 to their max value, excluding 0 because it does not represent a working hardware configuration but would instead

generate infinite long timings. To visualize the results I choose to sort them after the amount of transistors activated by each configuration as this reflects the state of the hardware the best. Furthermore, I set the equal for the west and east hemispheres as the tests always address both of them and for now it doesn't make a difference if they are distinguished or not, because faults appear on only one hemisphere usually:

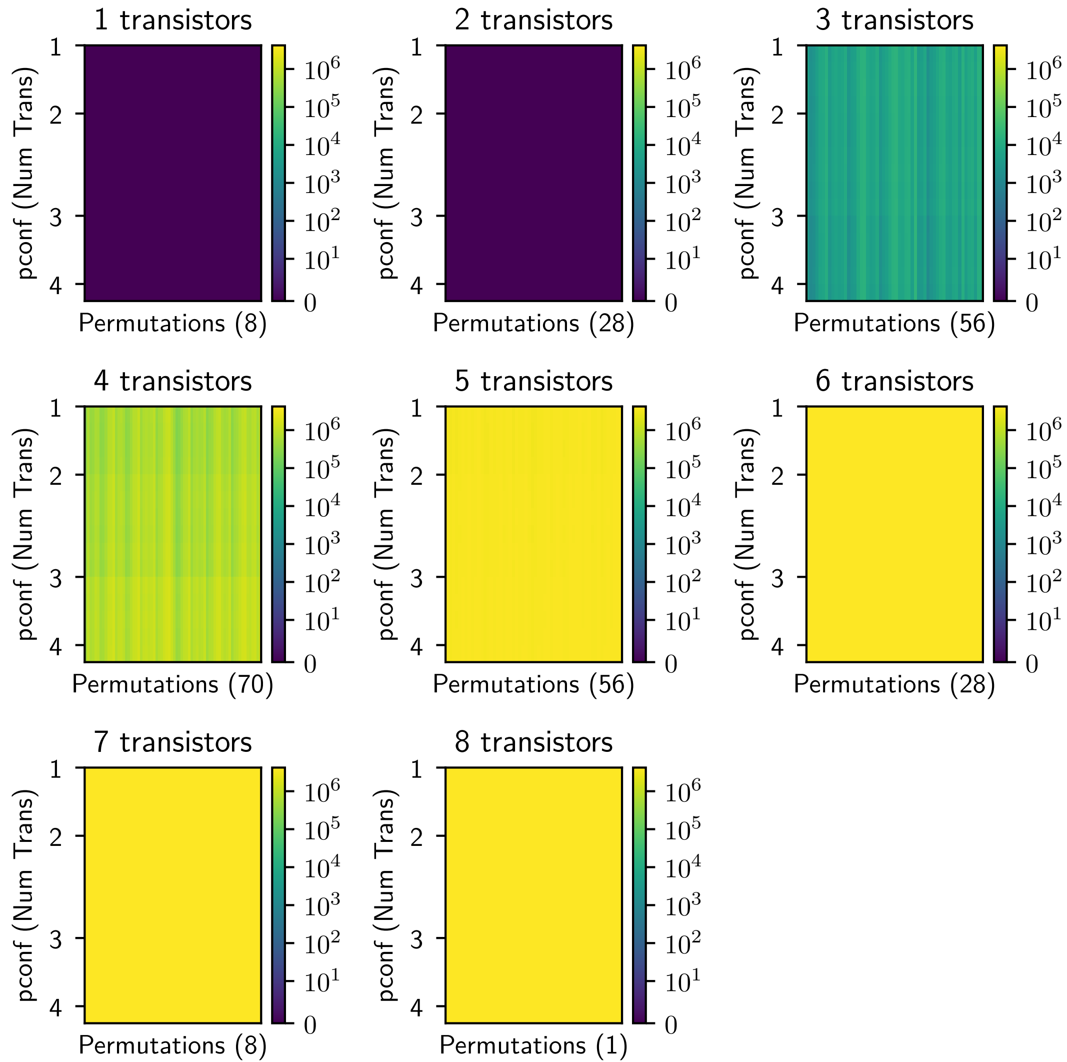


Figure 7: Result of the sweep for W70F3 northern hemispheres of one setup. The Colormap is linear up to 10 faults and logarithmic for the rest of the range and is normalized with respect to the maximal amount of faults $256 \times 256 \times 64$. The red dot indicates the standard configuration at the time of recording the sweeps.

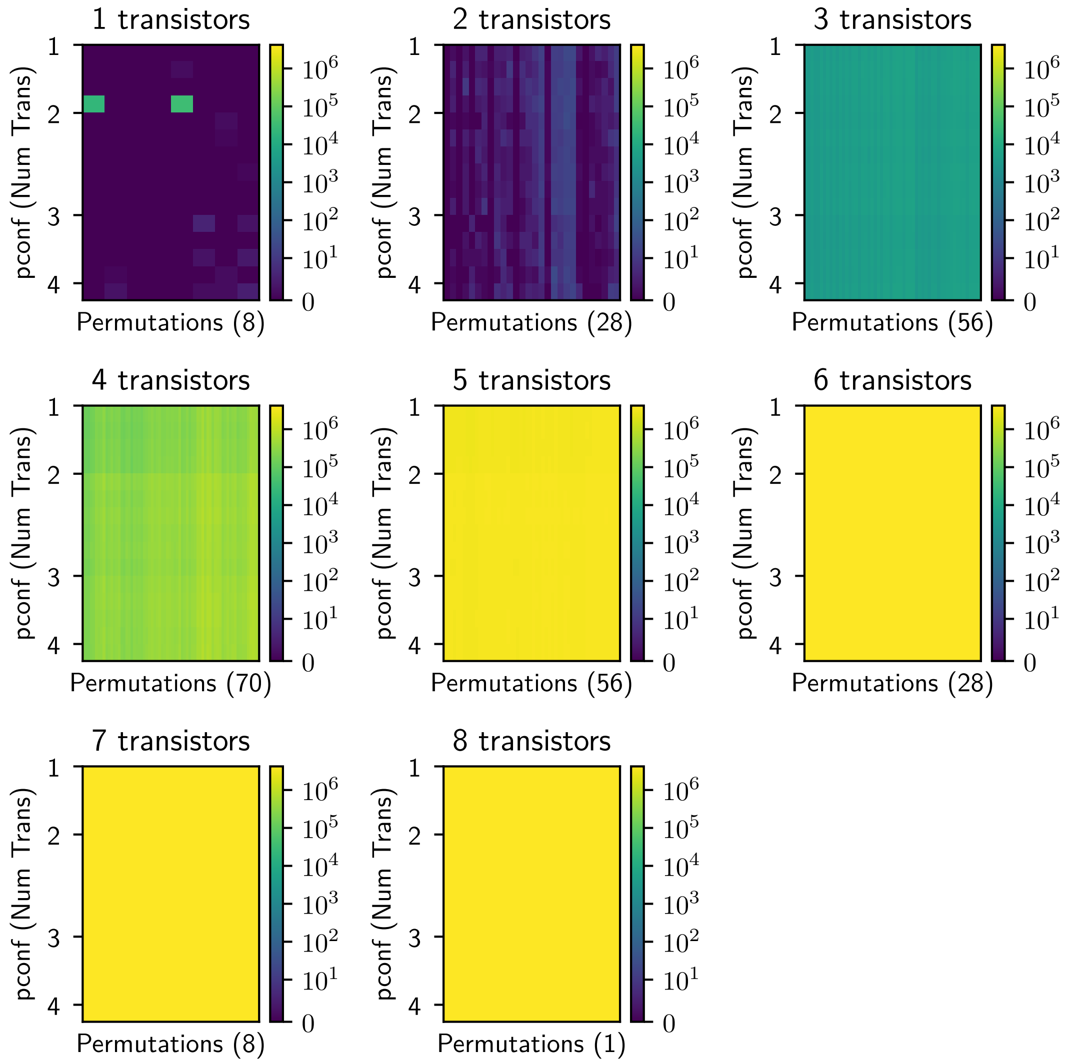


Figure 8: Result of the sweep for W70F3 southern hemisphere. Same settings as in fig. 7

The example in fig. 7 shows the best behavior we can expect from a hemisphere, all configs are good for one and two `wconf` transistor, all other configurations with more than 2 `wconf` transistors are "broken" regardless of the memory. The plot in fig. 8 shows a typical behavior for a hemisphere with faults, some `wconf` transistors alone are faulty, and many configurations are faulty for two `wconf` transistors. The observations made from these sweeps in general are:

— The timing configurations and the transistors used have a very strong impact on the faults occurring For this reason different hemispheres of the same chip can show very different fault behavior, because their timing is generated by different circuits/transistors.

— Using more than 3 `wconf` transistors for timing always produces faults, we can conclude that this setting is too fast in general. This restricts the useful parameter space we have to consider later.

— Increasing the `pcconf` speed most of the time also causes more faults, especially when faults are present for 2 `wconf` transistors.

— In the combined parameter space of 1 and 2 `wconf` transistors there always seem to be a fault free setting so far. For 2 `wconf` transistors this is not clear yet for all setups and must be verified further using other tests.

— There are cases where 1 `wconf` transistor alone causes faults but in combination with other is fault free. Conversely, there are pairs of transistors that alone don't produce faults but do when they are activated together, although this only happened on chips that have many faulty setting.

— In the case of single faulty transistors increasing the amount of `pcconf` transistors sometimes fixes the faults

— `wconf` has a greater impact on faults, as can be seen by the "stripes" in the plots along varying `wconf`

The most important result for the synapse memory is that there seems to be always a fault free configuration of the timings, therefore it's viable to introduce a calibration for the synapse memory. As I did not distinguish east and west in this sweep and still found fault free areas, they can be treated as one configuration regarding faults.

4.2 Impact of timing configuration on execution time

One question is if the timing configuration impacts the execution time of programs on the synapse memory. To test this I ran a set of 100 Pseudorandom matrices write and read on the synapse memory for different settings. I did one sweep for all configurations with 1 wconf transistor and one sweep with one representative for each equivalent configuration (i.e., testing 1 to 7 wconf transistors and 1 to 4 `pcconf` transistors). For the latte

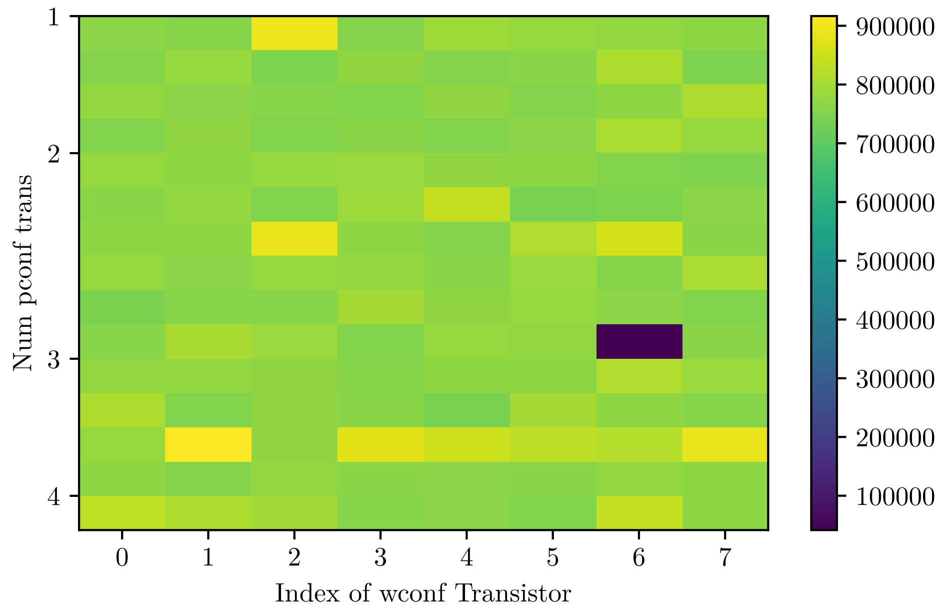


Figure 9: Complete `pcconf` Sweep for 1 wconf transistor

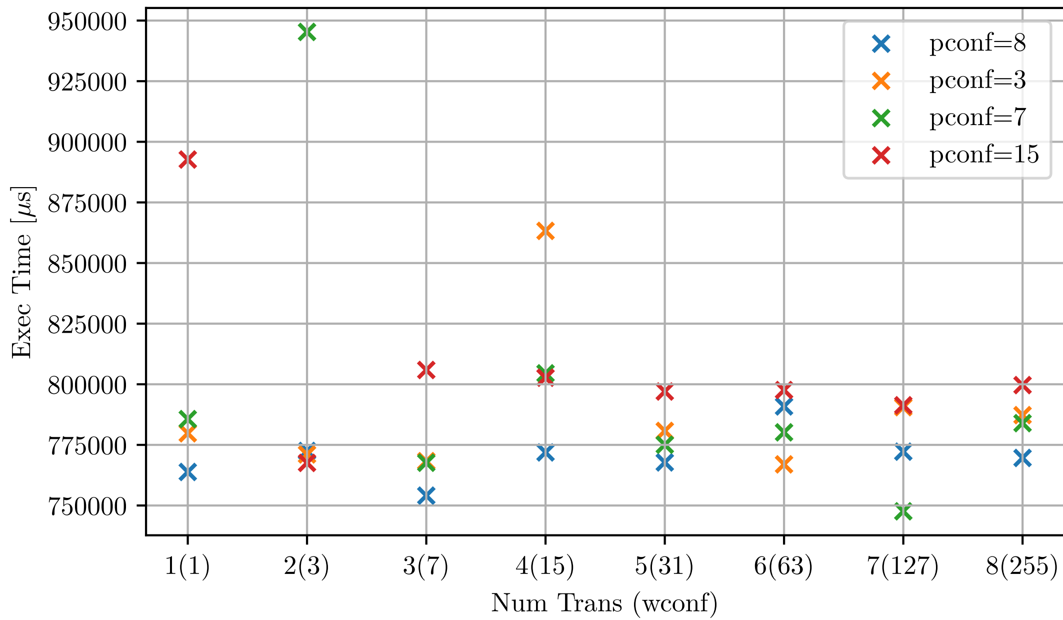


Figure 10: Sweep for representative configurations. The values in the parentheses give the values chosen to represent the amount of `wconf` transistors.

In figs. 9 and 10 it is shown that the analog timing configuration does not have a significant influence on the execution time. There are some unexpected deviations of single measurements, but these are not related to the timing configuration. This behavior is expected as the time of an operation on the digital side is not coupled to the analog operations but is dependent on the amount of cycles the operation takes which is set separately by the `wait_ctr_clear` configuration. With this knowledge, we now can say that the entire parameter space for 1 and 2 wconf transistors and 1 `pcconf` transistor is viable in most cases.

5 Calibration of the SRAM Timings

With all the investigation from before I can now consider the calibration of the analog timing settings to attain a fault free state of the synapse memory. The previous measurements have shown no systematic in the position of the faulty configurations in configuration space, therefore I perform a grid search of the viable configurations (as discussed in section 4.1).
The tests and algorithm used for the grid search have to full-fill two goals:

— Find Faults Efficiently: The Algorithm must exclude faulty configurations as quick as possible, therefore it makes sense to first perform tests with short runtime and quit testing the configuration early if these already find a fault. Furthermore, long tests should be split up if possible such that the algorithm can check regularly if faults occurred instead of running the entire test. Lastly the amount of tests used should be a small as possible, so tests that serve a similar purpose should not be included both.

— Guarantee a Fault Free Configuration: The algorithm should cover all tests needed to confirm the absence of faults with large confidence. This especially has to cover dynamic faults that don't occur every time the same test is run.

Taking into account these two points and the results from section 3.2 I choose the *ValueSweep* and *Pseudorandom* Matrices for this task. The *ValueSweep* pattern is employed to find strong faults quickly such that the algorithm can skip over such "obviously" faulty settings. The *Pseudorandom* Matrices are then used to validate the fault free state of the setting, here I choose to run sets of 10×500 distinct random Matrices to ensure good statistics for dynamic faults and allow for evaluation of the tests in between the sets. Finally, the algorithm reports the first fault-free configuration found. These tests are performed on the entire synapse memory (weights and labels) to ensure function of all memory cells.
As discussed in section 4.1 the configuration for east and west will be set equal for the calibration, because the hardware has fault free configurations without this distinction, thus reducing the parameter space dimension and making the calibration more efficient. For the calibration I set `wait_ctr_clear` to its maximum value to exclude any interference by the digital logic.

5.1 Result of the Calibration

The calibration was run on 10 setups of which 6 setups had faults on 7 different hemispheres. The calibration was able to find a fault free configuration for all of these setups, of which all except the southern hemisphere W70F0 where within the area of 2 `wconf` transistors and 1 `pcconf` transistors. It takes approximately 1.5 minutes to verify a fault free config and depending on the state of the chip on average 3–6 minutes to find a fault free config for a faulty hemisphere. The theoretical worst case for the used parameter space would be 216 minutes if the faults would occur only at the last step of the verification. The algorithm finds faults usually much earlier thanks to the multiple test steps, especially for very faulty setups, so even if it would have to check all configurations it would perform much better than the worst case.

6 Digital Timing

In addition to the analogue timing configurations for the synapse memory there is also the digital `wait_ctr_clear` configuration which sets how many clock cycles the digital logic waits before ending a read/write operation. If the setting is chosen to short it may cause the digital logic to abrupt reads to early before the sense amplifier could establish a well-defined value. On the other hand there is also the question of how this setting impacts memory access time.

6.1 Time impact of `wait_ctr_clear`

To investigate how `wait_ctr_clear` impacts time performance of the synapse memory, I recorded the execution time of writing and reading 100 random matrices for the possible setting of `wait_ctr_clear` (1–7), excluding 0 because it is not a valid configuration. I ran this test on Wafer 63 FPGA 0 for 50 times and took the average:

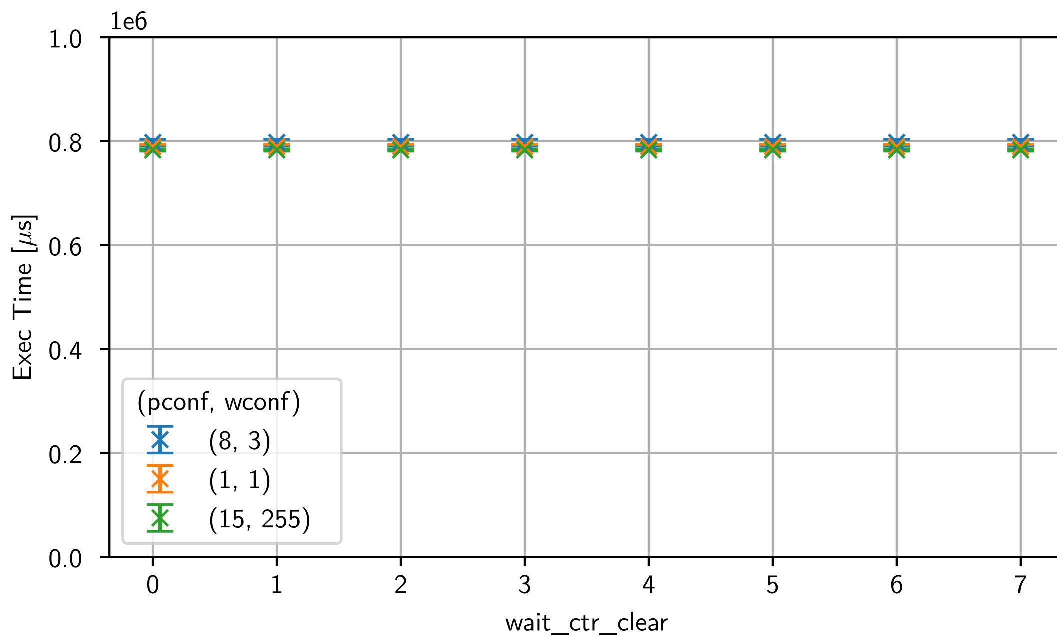


Figure 11: Execution time for different `wait_ctr_clear` on

As one can see in fig. 11 there is at most a weak correlation between `wait_ctr_clear` and execution time, therefore this factor will no longer be considered. On the other setups I ran this test once, where i also only saw deviations from the mean unrelated to the setting.

6.2 Impact of `wait_ctr_clear` on read faults

The more important question is which settings of `wait_ctr_clear` can be chosen without producing faults. To do this `wait_ctr_clear` is swept similarly as before but this time the amount of faults occurring for 1000 *Pseudorandom* Matrices are recorded. As the parameter space is quite small for this sweep I was able to use a more accurate test but which performs worse on runtime. For these tests the results from the calibration were used to only consider faults dependent on `wait_ctr_clear`.

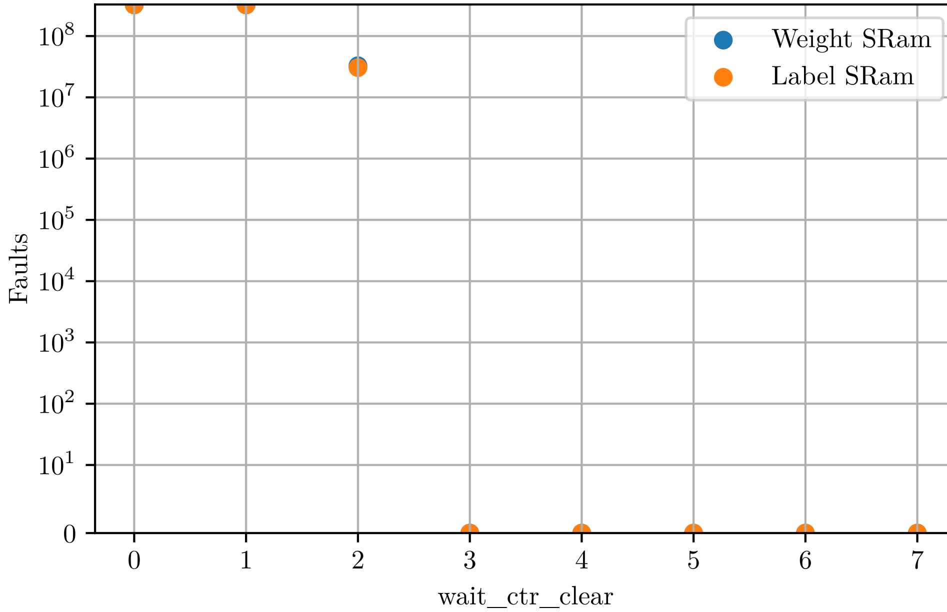


Figure 12: Amount of faults depending on `wait_ctr_clear` setting on W72F2

The sweeps fig. 12 show that `wait_ctr_clear` can be chosen too low resulting in faults starting from 1 or 2 clock cycle or less depending on the chip. It is therefore safe to choose 3 or more for a working hardware, thus I choose 4 clock cycles for just to have a "margin of error" to the lowest of 3.
We know now that we can ignore `wait_ctr_clear` during calibration and leave it at 4 for normal operation.

7 Hardware Database

The hardware configurations of the setups are handled by the Hardware Database, which stores the configurations as a YAML file accessed using the *yaml-cpp* library for C++. The data structures for the configurations are defined as C++ data structures in a header file which get serialized and deserialized by the *yaml-cpp* library as fields the YAML. The behavior for de-/serialization is described in a C++ file. For handling by the SLURM service the C++ objects must be transformed into pure-C objects defined in a separate header file. This conversion is also described in a C++ file.
For the results of my calibration script I added two 2D-arrays for `wconf` and `pcconf` which accounts for the north/south bearing of the configuration on the first axis and the east/west bearing on the second axis. As mentioned in section 4.1 the east/west bearing is not distinguished during Calibration, hence the values are set equal for both bearings.

```yaml
hxcube_id: 13
fpgas:
  - fpga: 0
    ip: 192.168.00.0
    fuse_dna: 0x0123456789ABCDEF
    ...
    synram_timing_pcconf:
      - [1, 1]
      - [1, 1]
    synram_timing_wconf:
      - [3, 3]
      - [144, 144]
  - fpga: 3
    ...
```

Figure 13: Example for the implemented YAML fields

8 Discussion

8.1 Summary

The goal of this internship was to investigate the fidelity of the synapse memory SRAM on the BSS-2 platform. As the first step a test library for the SRAM memory was developed in Python and multiple test methods and patterns implemented. After the implementation of the synapse memory into the library I ran a number of different tests on the Hardware. These tests showed that there are faults present on some setups. Further examination gave insights about the nature of the faults and which tests are fit to find them (see section 3.2). Using the insights from the previous tests I then inspected how the timing configuration `wconf` and `pcconf` impact the faults. To do so I did a sweep over the parameter space and recorded the amount of faults for each setting. This showed that the timing configurations have a very strong impact on the faults and that there is a reduced parameter space which always contained a fault free configuration. Further investigations also showed that these configurations do not impact the executions time of operations on the SRAM. Following these results I developed a calibration algorithm for the timing configuration using the knowledge gained from the previous tests. Before the calibration 4 hemispheres on 3 out of 10 setups showed faults in memory tests using the standard configuration at the time. With the presented calibration, I was able to reduce this number to zero: Thus for every tested setup a working configuration was found. Afterwards I studied the impact of the digital timing configuration `wait_ctr_clear` on execution time and faults. I concluded that the impact on execution time is negligible, and the setting is fault free above 3 and more cycles. As a final step I created the fields necessary for application of the calibration in the hardware configuration database for the setups.

8.2 Outlook

Concluding this internship there are still open questions/problems regarding the setups and configuration as well as possible improvements that go beyond the internship:

- Implementation of March tests: The approach I've taken using Matrices for the entire memory was only possible to the relative small size of the SRAM. For larger memories it would be necessary to implement march tests like mentioned in section 3. This implementation must be done in C++ due to the inefficiency of Python.
- Identification of the fault origin: In this internship I did not identify the fundamental reason for the faults, but just did an analysis of the general behavior and test performance, to calibrate the SRAM. For future hardware iterations it would be useful and interesting to investigate origin of the faults by employing more precise (and time intensive) tests and direct measurements.
- There are other memory arrays on the chips, which have not been tested (e.g., memory for capacitors). Using my library these could be implemented, tested and calibrations employed in a similar fashion if necessary.
- The configurations in the hardware database are currently not being applied automatically. To do so one would have to modify the C++ library responsible for loading the hardware configurations and add the timing configuration.
- On fundamental problem I discovered with the `wconf` setting was, that its usable range is limited below 2 `wconf` transistors. For future chip iterations this could be fixed by e.g., increasing the length of the transistors used.

References

Bosio, A. et al. (2012). "Advanced test methods for SRAMs". In: *2012 IEEE 30th VLSI Test Symposium (VTS)*, pp. 300–301. DOI: `10.1109/VTS.2012.6231070`.

Hock, Matthias (2014). "Modern Semiconductor Technologies for Neuromorphic Hardware". PhD thesis. Ruperto-Carola University of Heidelberg.

Pehle, C. et al. (Feb. 2022). "The BrainScaleS-2 Accelerated Neuromorphic System With Hybrid Plasticity". In: *Front. Neurosci.* 16.795876. DOI: `10.3389/fnins.2022.795876`.

9 Appendix

Software Versions

Name	Version/Git Hash
code-format	09f3a985a6f264359b10a6a129dd6dce7e55c9e8
halco	34cbdc02f6ae82c31309a1116815f6f006ab149a
haldls	5c3cce468fdae10f0849733fe2435afff613ed65
fisch	6120fc0ac0d90b3c66a212b3cc5cc25034bf584e
rant	722edd57c9e42462a660db8a1febb0211ffad07c
ztl	b6745261d8bfdce44516d58d632c3c73834839d2
pywrap	5e2af30e9593882b471d3cd02df00b93f13ff479
lib-boost-patches	136c5b41cb046afe2c726aa4646928bf5190622e
libnux	fc3b137384596ea5adbd5d4ee1ddfc9761a2aabc
hate	0471609e365195012d66aa7178ae393d48999d96
logger	73dadb3ce413c521845ef7d36f818073eee4fefa
hxcomm	95abf25670bd8cb7cc5b499cde56f653130cf20c
sctrltp	1d854f953f7e8c8ead44406a22bb80421ca3857c
visions-slurm	8f41ea4f5bd1573d8f4623e9ed698a29f30036a3
flange	28e729d59df3b4ff380f84351c40d4da3086bed8
lib-rcf	21fbcb0a7c30efed98278ee997754f28092b9736
bss-hw-params	b7be7827b51536804f0bda76f8ba4be693df23a8
hwdb	f7262189b0e55b686896a3dea952065c2f1a3789
bss2-devops	65a9f028b9580a975cb09805b0f2b27b179a1960
calix	a706868c6ba285b1f8fd7cdef1a19d7328e02912
bitfile	157
container	stable/2024–04–17_1

Table 1: Software Versions used during the internship

Glossary

SRAM Static Random Access Memory

SNN Spiking neural network

BSS-2 BrainScaleS-2

Funding Statement

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

Erklärung

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

Heidelberg, den 21. 10. 2024, N. Bergler