Reliability Analysis of a Spiking Neural Network Hardware Accelerator

Theofilos Spyrou, Sarah A El-Sayed, Engin Afacan, Luis A Camuñas-Mesa,
Bernabé Linares-Barranco, Haralampos-G. Stratigopoulos

The prior art on dependability analysis of SNNs is still at a very early stage. In [11], a functional Built-In Self-Test (BIST) is proposed for biologically-inspired spiking neurons. The idea is to test in one-shot with a specially crafted stimulus that the neuron is capable of producing all the expected firing patterns. If one or more firing patterns are missing, then the neuron is declared faulty.

Transistor-level defect and process variation simulations for a single spiking neuron have been performed in [12]. Main observed faulty behaviors include saturated neuron behavior, i.e., nonstop spiking at the absence of excitatory input, dead neuron behavior, i.e., halt in spiking despite the presence of excitatory input, and timing variations, i.e., variations in time-to-first-spike and firing rate.

In [6], such faulty behaviours are mapped in a Python-based SNN model. Fault simulations showed that saturated neuron faults occurring anywhere in the network can be lethal, while all other fault types can impact the classification accuracy if they occur in neurons in the last layers. A two-step neuron fault tolerance approach is proposed. First, training the network with dropout [13] helps to passively tolerate all except the saturation faults and the faults occurring in the last layer. Then, an active fault tolerance scheme is used based on detecting saturation on a per neuron basis and cancelling it out by the "fault hoping" concept where a saturated neuron is transformed to a dead neuron. TMR is used for the last and most critical layer.

The work in [7] presents a fault taxonomy and discusses behavioral-level fault injection experiments for a SNN architecture with memristor-based synapse implementation. The fault model includes worst case faulty behaviours, such as dead neuron and dead synapse faults, where a dead synapse is modeled with a zero weight. Results show that a high fault density is required for noticeable decrease in recognition rate.

The work in [8] studies the resilience of feed-forward SNNs to dead synapse faults when trained with different algorithms. Synapses are selected to be faulty at random with different fault rates. Results show that resilience characteristics depend largely on the training algorithm and in all cases the accuracy drops rapidly with increasing fault rates. It is shown how to modify an evolutionary optimization-based training algorithm so as to improve fault tolerance. In particular, the fitness function is re-designed to become a weighted sum of the baseline accuracy and the average accuracy obtained on versions of the SNN with dead synapse faults. The resilience is improved but the baseline accuracy is not recovered.

In [9], fault injection is performed in a Python-based SNN model. The fault model is bit-flips in the memories storing the weights of the network. A uniform random distribution with different rates is considered. Fault-tolerance schemes to mitigate memory failures are also proposed, namely Fault-Aware Mapping (FAM) and Fault-Aware Training and Mapping (FATM). First, the memory fault map, i.e., the location of the faulty memory cells, is derived using testing. FAM consists in identifying the memory segment with the highest number of subsequent non-faulty cells and prioritize placing the Most Significant Bits (MSBs) of the weight in this segment, which is done using a circular shift. FATM follows FAM and consists in performing re-training while considering bit-flips for different rates during training epochs. In this way, the network adapts its accuracy to different bit-flip probabilities.

In all the aforementioned works, resilience characteristics of SNNs have been studied by performing fault injection at transistor-level for single neurons or in a behavioral-level model for entire networks. Although experimenting on higher abstraction models allows flexibility, the particularities of a hardware implementation are not taken into consideration.

Similarly, in the case of ANN hardware accelerators reliability analysis has been mainly performed by simulation using a higher abstraction model [14], [15]. Few works exist demonstrating fault injection in actual hardware, in particular fault injection in FPGA implementations [16] and radiation experiments on ANNs running in GPUs [17], [18] and FPGAs [19].

In this work, we present results of fault injection on actual neuromorphic hardware. For this purpose, we designed in VHDL a convolutional SNN for the poker card symbol recognition task and implemented it on an FPGA. For our implementation, we use the Zynq UltraScale+ MPSoC ZCU104 board from Xilinx. The network parameters are configured automatically with a MATLAB script to allow easy experimentation. The fault injection framework runs on the processor of the board allowing us to perform an accelerated large-scale reliability analysis. Our fault model includes bit-flips in the memories storing the different network parameters. We analyse the resilience of the entire network and layer-by-layer at different fault rates and bit positions. The experiments pinpoint the critical parts of the design for which a fault-tolerance scheme is demanded, as well as the parts that are inherently fault-tolerant.

The rest of the paper is structured as follows. In Section II, we describe the SNN hardware architecture. In Section III, we introduce the fault model. In Section IV, we provide the findings of the fault injection experiments. Section V concludes the paper.

II. SNN Hardware Architecture

The building block of the SNN is an event-driven configurable convolutional node proposed in [20] as a generic block that can be used to build multi-layer feature maps for convolutional SNNs communicating through the Address Event Representation (AER) protocol. In this section, we present the node along with a brief description of its most important features. Then, we present the SNN built using this node to
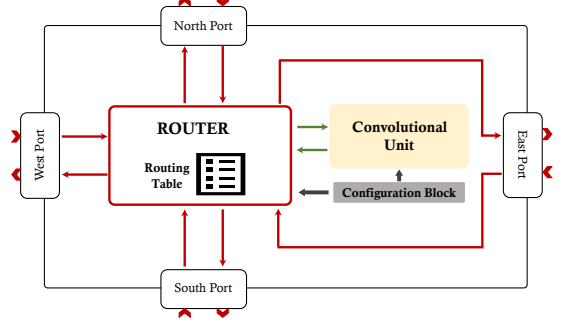
Fig. 1: The convolutional node.
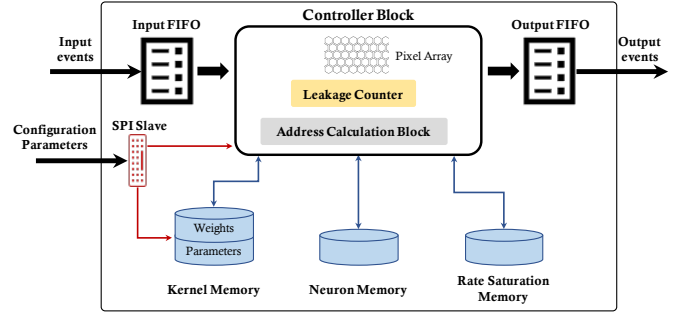
Fig. 2: The convolutional unit.

classify poker card symbols recorded using a Dynamic Vision Sensor (DVS).

A. The Convolutional Node

The node consists of three main blocks, namely a convolutional unit, an internal configuration block, and a router, as shown in Fig. 1. The ports of the node are optimized for a 2-D layout, an efficiently adopted structure in hardware Convolutional Neural Networks (CNNs) since it optimizes the use of on-chip space. Each node has four bidirectional ports connecting it to its immediate neighbors to the north, south, east, and west.

1) The Convolutional Unit: Shown in Fig. 2, it consists of an array of Integrate & Fire (I&F) neurons representing pixels, three main memory blocks (i.e., kernel memory, neuron memory, and rate-saturation memory), First In, First Out (FIFO) input and output registers, a controller block, and a Serial Peripheral Interface (SPI) block.

The convolutional unit is where the convolution of an input flow of events $ev_{in}(t, x, y, p, k)$ and a kernel $w_k(x, y)$ takes place to produce an output flow of events $ev_{out}(t, x, y, p)$, where t is time, x and y are the pixel address coordinates, p is the polarity of the event, and k is the kernel ID in the kernel memory. In addition, the unit has two more important features: *global leakage* and *rate saturation*. Leakage is the decay of the neuron membrane potential in between incoming spikes and it is implemented by forcing the neuron state to converge towards the reset value after a certain time interval, which is determined by a global counter. The rate saturation feature, on the other hand, is the imposition of the minimum refractory period property found in biological neural networks, i.e., the neuron is not allowed to produce an output spike for a

certain period after the last output spike, hence controlling the maximum spiking frequency of a neuron.

With every incoming event read from the input FIFO register, a convolution operation is executed and the values of the corresponding pixels are updated and compared to the positive and negative thresholds. If the value of a threshold is reached by a pixel and the condition imposed by the rate saturation mechanism is fulfilled, the pixel produces an output event with address (x_{out}, y_{out}) and polarity p_{out} and writes it in the output FIFO register.

To control the traffic, a signal is activated when the registers get full, and the incoming events are discarded until there is room for more events in the register. While this implies that the output events would get down-sampled and some information will eventually be lost, the spatio-temporal correlation of the passing events is preserved, keeping the integrity of the carried information.

2) The Router: In hardware implementations of neural networks, the highly dense connectivity required between neurons poses a challenge in terms of on-chip area. Routers handle the transmission of events from their origin to their destination, hence providing a practical solution to this problem. In this design, the destination-driven addressing scheme is adopted, which means that for every event, there is a routing header that carries the x and y coordinates of the destination node in the mesh distribution of the network.

3) The Configuration Block: The convolutional node has a set of adjustable parameters that need to be configured prior to the inference operation of the network. Some parameters belong to the convolutional unit, such as the neuron threshold and the kernel weights. Others belong to the router, such as the local address of the node and the routing table information necessary for redirecting events through the ports of the node. Each parameter value is sent to the node through a SPI, with an index indicating its identity. The configuration block interprets the parameter identities and handles their allocation in their corresponding locations in the different memory blocks.

B. The Poker Card Symbols Dataset

The SNN used in this work is built to classify a dataset representing the 4 poker card symbols [21]. A deck of 40 poker cards was presented in front of a DVS for a period of around $1\ s$. The events were recorded and processed in order to generate 40 samples of 32×32 pixel windows showing only the centered symbols. The resulting stimulus has a total of 174644 events, a duration of $950\ ms$, and an average speed of $184K$ events per second. In our experiments, we use a version of the dataset slowed down to 1% of the original speed in order to ensure a scenario where no input events are discarded.

C. The Convolutional SNN

The convolutional SNN is designed and trained in software in a frame-based format using backpropagation, and then transformed into the equivalent spiking form [21]. Afterwards, the weights and parameters are scaled, rounded, and then tuned to make up for the discrepancies between hardware and software
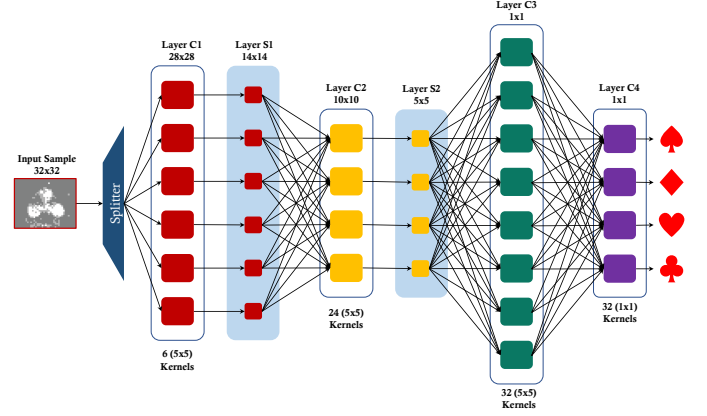
Fig. 3: Convolutional SNN for poker card symbol recognition.
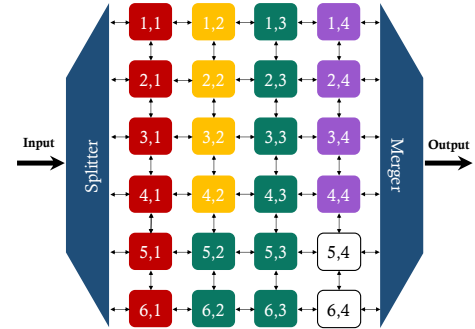
Fig. 4: 2-D mesh SNN implementation on the FPGA.

implementations using simulated annealing as an optimization algorithm [20].

As shown in Fig. 3, the SNN consists of 4 convolutional layers (C1, C2, C3, and C4) made up of 22 convolutional nodes. The first 2 layers are followed by 2 sub-sampling layers (S1 and S2). The network has 94 kernels in total, where layer C1 has 1 kernel per node, layer C2 has 6 kernels per node, layer C3 has 4 kernels per node, and layer C4 has 8 kernels per node.

The 2-D hardware layout of the FPGA implementation is shown in Fig. 4, where the nodes are arranged in a 6×4 mesh with bidirectional connections between the routers of each block and its immediate neighbors. Every node carries an identification corresponding to its x and y address in the mesh, and its color indicates the respective layer in the network. Nodes (5,4) and (6,4) are extra nodes added for routing purposes but do not perform any processing.

Input events do not have any specified destination address and they need to be sent to all nodes of the first layer. Therefore, there is an extra *splitter* block at the input side, which creates 6 copies of every incoming event, adds the address of a node in the input layer to each copy, and delivers them to the corresponding nodes. At the output side, there is a *merger* block which simply forwards events from the 4 nodes of the output layer to the output of the network without altering them.

D. Embedded system design

To automate the reliability analysis, we built a hardware platform and a software framework to support it. More specifically, we designed a standalone embedded system application where

the convolutional SNN is handled by a C program running on the ARM Cortex-A53 APU of the MPSoC. The C program iterates over the fault injection experiments that have been stored to an SD card. For each experiment, the C program coordinates the following actions: (i) configuration of the faulty SNN; (ii) generation of the input spiking events according to the dataset; and (iii) monitoring of the output spiking events and storing to the SD card. The hardware platform communicates with a PC through a serial connection. The software framework consists of three MATLAB scripts: (i) the *configurator* which generates the configuration data for the nominal SNN; (ii) the *injector* which performs the fault injection into the *configurator*'s output to generate configuration data for faulty SNN versions which are then written to the SD card; and (iii) the *classifier* which processes the output spiking events written to the SD card and performs the classification. The winning poker card symbol is the one whose node produces the largest number of spikes. The classifier also calculates the SNN recognition accuracy over the testing set.

III. FAULT MODEL

As discussed in Section II, the generic convolutional node used as a building block in this hardware implementation is configurable through a set of modifiable parameters. These parameters are stored in mutually exclusive memory blocks inside the node, which are the subject of our reliability experiments. The parameters have an 8-bit representation in hardware and can be categorized into:

1) Splitter Parameters: The splitter is parametrized by the number of input event copies to generate and send to each first-layer node, as well as the node addresses. Splitter parameters make up 0.52% of the used memory.

2) Router Parameters: The router needs two important settings, namely the local address of the node and the routing information necessary for redirecting events through its ports, i.e., addresses of next-layer nodes and the direction towards them (down, right). The router also carries the kernel ID information so that the correct kernels corresponding to each event can be retrieved from the memory. Router parameters make up 15.25% of the used memory.

3) Neuron Parameters: Neuron parameters govern the key features of the I&F neurons within the node. They include the neuron threshold, the leakage pulse amplitude and period, and the refractory period. These parameters are set for the whole node, i.e., they are global to the whole array of neurons inside a node. Neuron parameters make up 7.8% of the used memory.

4) Kernel Parameters: Every node of the network has a specific number of kernels and needs two parameters per kernel, namely the kernel size and the center-shift of the kernel which determines whether the kernel is applied to the pixels in the zone around the one given by the event destination address or it is shifted to another pixel. Kernel parameters make up 10.75% of the used memory.

5) Kernel Weights: The number of kernel weights per kernel is determined by the kernel size, i.e., a $n \times k$ kernel has $n * k$ weights. The kernel weights considering all kernels in all nodes of the network make up 68.55% of the used memory.
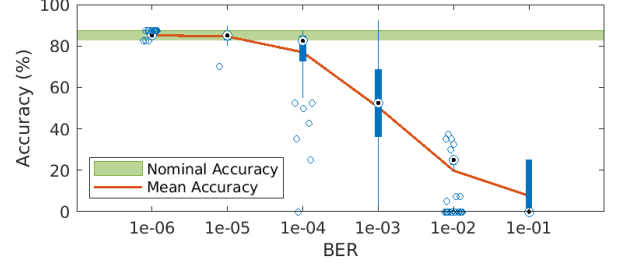

Fig. 5: Network accuracy for different BER values.

The fault model consists of permanent bit-flips in the aforementioned memories. Bit-flips are injected in two different ways, in particular with a Bit Error Rate (BER) probability, leading to a multiple-bit fault scenario with uniform random distribution of bit-flips, or considering a single-bit fault scenario. In the former scenario, assigning different BER probabilities helps assessing the BER that can be tolerated by the SNN. In the latter scenario, we study the effect of single bit-flips parameter-by-parameter, layer-by-layer, and for different bit positions. The goal is to identify critical parts of the design, as well as critical bit positions.

IV. RESULTS

For each fault injection experiment we evaluate the SNN recognition rate and compare it with the baseline value for the nominal design which is $85 \pm 2.5\%$.

Each fault injection experiment takes approximately 2 minutes. In the multiple-fault scenario, for a given BER value, we perform 100 repetitions. As the total memory size is 18464 bits, it is very time-consuming to perform all single bit-flip scenarios even in hardware where run-time is accelerated. Thus, for the kernel weights in the first three layers C1, C2, and C3 that occupy the largest fraction of the memory, we perform fault sampling, randomly selecting 20%, 10%, 10% fault locations, respectively. For the rest of the parameters we perform exhaustive fault injection. In total, we performed 11925 fault injections which took approximately 16.5 days of simulation time.

Fig. 5 shows the accuracy versus BER in the case where bit-flips are injected uniformly at random across the entire network. We visualize summary statistics using box plots. The bottom and top edges of the box indicate the 25th and 75th percentile, respectively, the whiskers extend to the most extreme data points not considered outliers, and the outliers are plotted individually using the 'o' symbol and are not always aligned vertically for illustration purpose. We also report the median shown with a dotted circle and the average accuracy across repetitions of the same experiment. Experiments with 0% accuracy correspond to an application crash as the result of fatal errors which made the system unable to respond to any incoming event activity. As it can be seen from Fig. 5, the accuracy drops with increasing BER and beyond 10^{-4} the drop is below the tolerated zone shown with green color. This shows that the maximum tolerated BER is 10^{-5} or less. We also observe that for moderate BER values the variance of
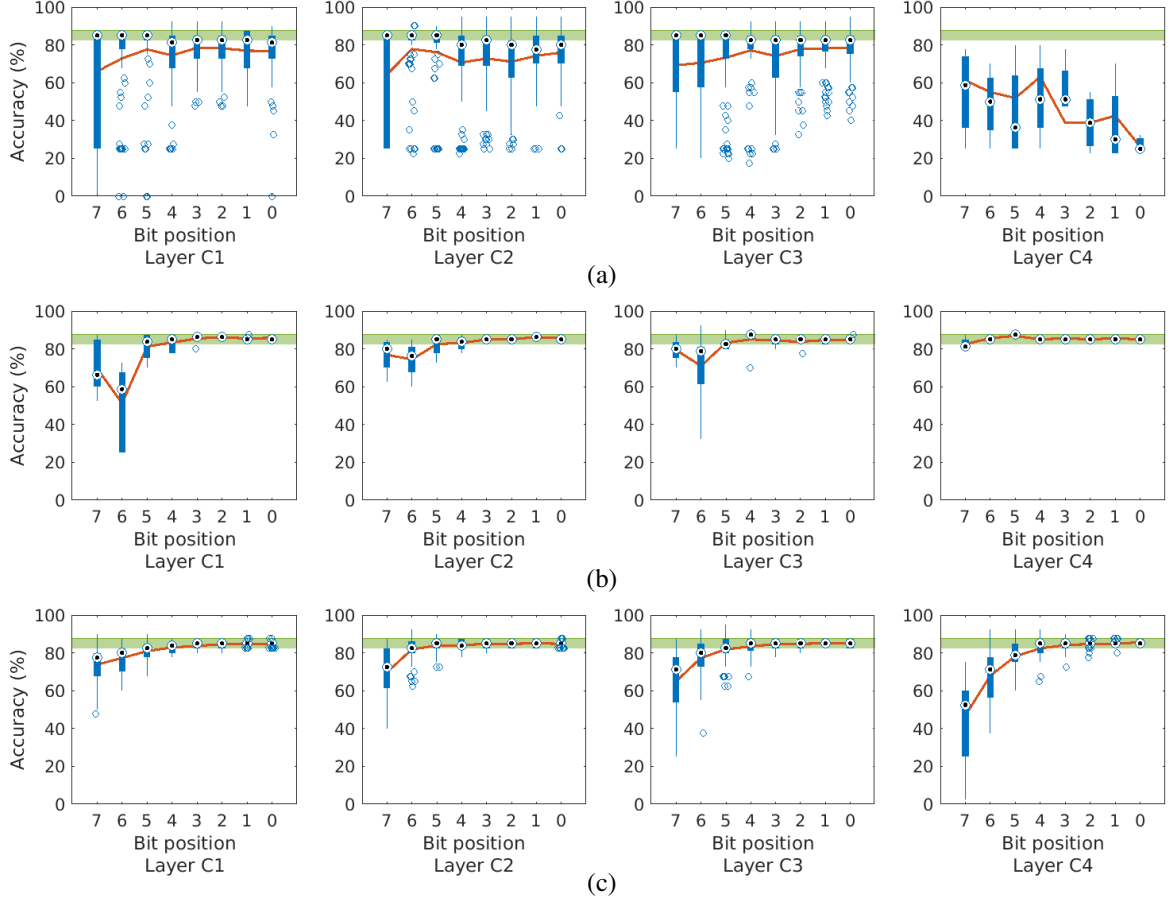
Fig. 6: Single bit-flips layer-by-layer: (a) Router Parameters; (b) Neuron Threshold; (c) Kernel Weights.

the accuracy increases as BER increases, which shows that accuracy drop is largely dependent on the combinations of faulty bits and their locations.

The outliers for moderate BER values in Fig. 5 are due to faults occurring is the smaller yet far more critical memory blocks storing splitter, router, and kernel parameters. In fact, the network is susceptible even to single bit-flips affecting these parameters. Faults in the splitter may lead to generated addresses that do not correspond to a valid node inside the mesh, and this leads to input events not reaching any destination address and, thereby, not being processed. Faults in the kernel parameters change the kernel's size and center-shift and faults in the router parameters change the routing of the spikes. Thus, such faults essentially result in a structurally different network architecture. For example, Fig. 6(a) shows single bit-flips in the router parameters layer-by-layer and across different bit positions. The network is very sensitive and the bit position where the flip occurs is irrelevant. Thus, we conclude that splitter, router, and kernel parameters are critical and protecting them is of utmost importance.

Figs. 6(b)-(c) show results for single bit-flips layer-by-layer for the neuron threshold and kernel weights, respectively. For the single-bit fault scenario, among the different neuron parameters, accuracy drop was observed only for the neuron threshold, thus Fig. 6(b) shows results only for the neuron

threshold. Figs. 7(a)-(b) show results for multiple bit-flips with different BER values layer-by-layer for the neuron parameters and kernel weights, respectively.

From Fig. 6(b) we observe that the network performance is sensitive to single bit-flips in the neuron threshold only if these occur in the 3 Most Significant Bits (MSBs), while the last layer C4 shows no vulnerability. From Fig. 7(a) we observe that layers C1 to C3 start showing vulnerability for BER values larger than 10^{-2}.

Regarding kernel weights, from Fig. 6(c) we observe that the network sensitivity increases with the layer number. Thus, some faults in the beginning of the network tend not to propagate. Another observation is that single faults affecting Least Significant Bits (LSBs) can be tolerated. From Fig. 7(b) we observe that the network can tolerate up to a BER of 10^{-5}. These results show that leaving unprotected the 4 LSBs of the kernel weights, which always occupy the largest fraction of the memory, is feasible, leading to significant cost reduction in fault tolerance schemes.

V. Conclusions

We presented a fault injection experiment and fault resilience analysis for a SNN hardware accelerator. We developed a fault injection framework that creates and maps SNN faulty instances into the hardware, allowing to accelerate fault injection and
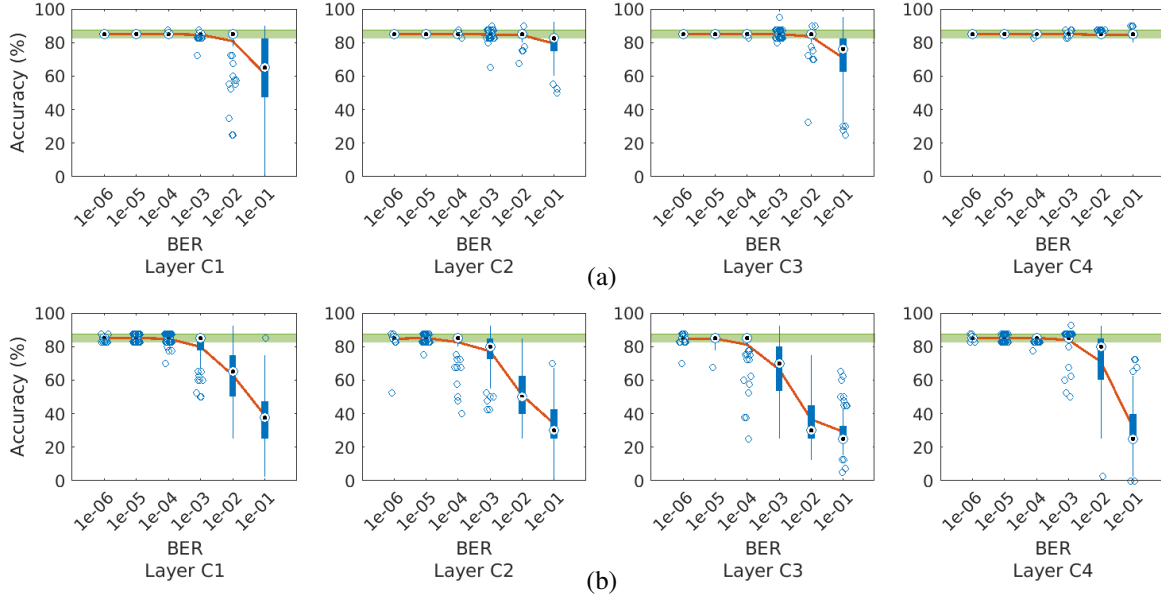
Fig. 7: Multiple bit-flips for different BER values layer-by-layer: (a) Neuron Parameters; (b) Kernel Weights.

assess the fault criticality on actual hardware. Our findings show that certain SNN parameters, i.e., splitter, router, and kernel parameters, are critical and must be protected, while for others, i.e., neuron threshold and kernel weights, the network shows some degree of resilience to faults occurring in LSBs. Therefore, these parameters can be the subject of selective fault tolerance reducing the cost of an all-around fault-tolerance.

REFERENCES

[1] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Netw.*, vol. 10, no. 9, pp. 1659–1671, Dec. 1997.

[2] M. Pfeiffer and T. Pfeil, "Deep learning with spiking neurons: opportunities and challenges," *Front. Neurosci.*, vol. 12, Oct. 2018, Article 774.

[3] F. Akopyan *et al.*, "TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 10, pp. 1537–1557, Oct. 2015.

[4] M. Davies *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan./Feb. 2018.

[5] L. A. Camuñas-Mesa, B. Linares-Barranco, and T. Serrano-Gotarredona, "Spiking neural networks and their memristor-CMOS hardware implementations," *Materials*, vol. 12, no. 17, Aug. 2019, Article 2745.

[6] T. Spyrou, S. A. El-Sayed, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Neuron fault tolerance in spiking neural networks," in *Proc. Design Autom. Test Europe Conf. (DATE)*, Feb. 2021.

[7] E. Vatajelu, G. Di Natale, and L. Anghel, "Special session: Reliability of hardware-implemented spiking neural networks (SNN)," in *Proc. IEEE VLSI Test Symp.*, Apr. 2019.

[8] C. D. Schuman *et al.*, "Resilience and robustness of spiking neural networks for neuromorphic systems," in *Proc. Int. Jt. Conf. Neural Netw. (IJCNN)*, Jul. 2020.

[9] R. V. W. Putra, M. A. Hanif, and M. Shafique, "ReSpawn: Energy-efficient fault-tolerance for spiking neural networks considering unreliable memories," in *Proc. 40th Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2021.

[10] C. Liu, M. Hu, J. P. Strachan, and H. Li, "Rescuing memristor-based neuromorphic design with high defects," in *Proc. 54th ACM/EDAC/IEEE Design Autom. Conf.*, Jun. 2017.

[11] S. A. El-Sayed, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Self-testing analog spiking neuron circuit," in *Proc. Int. Conf. Synth. Model. Anal. Simulat. Methods Appl. Circuit Design (SMACD)*, Jul. 2019.

[12] S. A. El-Sayed, T. Spyrou, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Spiking neuron hardware-level fault modeling," in *Proc. 26th IEEE Int. Symp. On-Line Test. Robust Syst. Des. (IOLTS)*, Jul. 2020.

[13] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jun. 2014.

[14] G. Li *et al.*, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2017.

[15] M. A. Neggaz, I. Alouani, S. Niar, and F. Kurdahi, "Are CNNs reliable enough for critical applications? An exploratory study," *IEEE Des. Test*, vol. 37, no. 2, pp. 76–83, Apr. 2020.

[16] U. Zahid, G. Gambardella, N. J. Fraser, M. Blott, and K. Vissers, "FAT: Training neural networks for reliable inference under hardware faults," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2020.

[17] F. F. dos Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on GPUs," *IEEE Trans. Reliab.*, vol. 68, no. 2, pp. 663–677, Jun. 2019.

[18] A. Lotfi *et al.*, "Resiliency of automotive object detection networks on GPU architectures," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2019.

[19] F. Libano, B. Wilson, J. Anderson, M. J. Wirthlin, C. Cazzaniga, C. Frost, and P. Rech, "Selective hardening for neural networks in FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 66, no. 1, pp. 216–222, Jan. 2019.

[20] L. A. Camuñas-Mesa, Y. L. Domínguez-Cordero, A. Linares-Barranco, T. Serrano-Gotarredona, and B. Linares-Barranco, "A configurable event-driven convolutional node with rate saturation mechanism for modular convnet systems implementation," *Front. Neurosci.*, vol. 12, Feb. 2018, Article 63.

[21] J. A. Pérez-Carrasco *et al.*, "Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward ConvNets," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 11, pp. 2706–2719, Nov. 2013.