On-Line Testing of Neuromorphic Hardware

Theofilos Spyrou, Haralampos-G. Stratigopoulos

On-Line Testing of Neuromorphic Hardware

Theofilos Spyrou and Haralampos-G. Stratigopoulos
Sorbonne Université, CNRS, LIP6, Paris, France

***Abstract*—We propose an on-line testing methodology for neuromorphic hardware supporting spiking neural networks. Testing aims at detecting in real-time abnormal operation due to hardware-level faults, as well as screening of outlier or corner inputs that are prone to misprediction. Testing is enabled by two on-chip classifiers that prognosticate, based on a low-dimensional set of features extracted with spike counting, whether the network will make a correct prediction. The system of classifiers is capable of evaluating the confidence of the decision, and when the confidence is judged low a replay operation helps to resolve the ambiguity. The testing methodology is demonstrated by fully embedding it in a custom FPGA-based neuromorphic hardware platform. It operates in the background being totally non-intrusive to the network operation, while offering a zero-latency test decision for the vast majority of inferences.**

***Index Terms*—Neuromorphic computing, spiking neural networks, testing, reliability.**

I. Introduction

Neuromorphic computing having as basis a Spiking Neural Network (SNN) emulates brain-like functionality aiming at solving various problems, such as visual sensing and perception, control-loops for robotics, brain-computer interfacing, audio processing, etc., with comparative advantage to the contemporary von Neumann computing architecture as well as the classical artificial neural networks. Neuromorphic computing is a rapidly evolving field and intense efforts are ongoing on hardware architecture and software design as well as on demonstrating real-world applications [1]. Nowadays there exist large-scale research hardware platforms [2]–[5] and several chips have been demonstrated for various benchmarks [6], [7].

This work addresses the on-line testing of SNNs aiming at detection of abnormal behavior due to hardware-level faults, as well as detection of outlier and corner inputs that are prone to misprediction. Therefore, beyond fault detection, the proposed methodology can be used to assess the confidence in the SNN prediction, contributing to the trustworthiness of the application.

The proposed methodology is based on a lightweight and non-intrusive on-die symptom detector that operates in the background in parallel with the SNN and outputs a decision whether the running input will be correctly predicted by the SNN. The detector is based on a committee of two classifiers trained to make the decision based on a low-dimensional set of spike-based features extracted on-die.

The detector is capable of evaluating the confidence of its decision, thus achieving an optimal zero false negatives/zero false positives trade-off. For the vast majority of SNN inferences, a correct decision is made deterministically in one-shot with zero latency and concurrently with the SNN operation without interfering with it. Whenever the decision has low-confidence, the ambiguity is resolved by performing a single replay operation. The detector is also capable of detecting a percentage of faults that do not affect the SNN prediction for the running input, but are likely to affect the prediction of a future input, thus it serves also as a failure prognosis mechanism that allows taking action to avoid such failures.

The method is demonstrated on hardware using an SNN benchmark for poker card symbol recognition. The SNN is designed in VHDL and is implemented on the FPGA of the Zynq®UltraScale+™ MPSoC ZCU104 FPGA board by Xilinx, Inc. The classifiers run on the ARM quad-core ARM®Cortex™-A53 processor of the board.

The rest of the article is structured as follows. In Section II, we provide a concise overview of SNNs. In Section III, we briefly present the state-of-the-art on SNN testing. In Section IV, we discuss the proposed on-line testing approach. In Section V, we present the case study and hardware platform. In Section VI, we present the experimental results. Section VII concludes this article.

II. SNN background information

Inspired by biology, SNNs are an effort to leverage the advantages of the human brain in Artificial Intelligence (AI). The neurons comprising a SNN use unit instantaneous signals, called spikes, in order to encode and process the information. Sequences of spikes or spike trains are propagated to the network through layers of neurons linked via weighted connections, called synapses.

One of the least computationally complex, and hence hardware-friendly, neuron model is the Integrate & Fire (I&F) model [8]. Initially, the neuron is at a resting state, where the membrane potential is set to a low value. The neuron integrates the incoming spikes from the synapses at its input and the membrane potential is increased correspondingly. Once the potential exceeds a specified threshold, the neuron fires a spike, which is propagated to the next layer of neurons via the synapses connected to its output, and the neuron is reset to its resting state again. The refractory period is the time in-between successive spikes regulating the maximum possible spiking frequency of the neuron. An extension to the aforementioned model is the leaky I&F neuron, where the membrane potential is periodically brought closer to the resting state during the idle time of the neuron, so if there are no incoming spikes, then the neuron is gradually reset.

This work was supported by the ANR RE-TRUSTING project under Grant N^{o} ANR-21-CE24-0015-03. The work of T. Spyrou was supported by the Sorbonne Center for Artificial Intelligence (SCAI) through Fellowship.

The input to a SNN needs to be in a neuromorphic form too, i.e., in a spike train representation. To achieve this, a neuromorphic camera or a Dynamic Vision Sensor (DVS) is usually employed. A DVS resembles the retina of the human eye and is composed by pixels that behave similarly to a neuron responding to changes in the brightness. If the brightness of a pixel has changed sufficiently, a spike is generated. Each pixel operates independently and reports the brightness changes as they occur.

Neurons operate and fire spikes asynchronously. To handle the inter-neuron communication in a realistic neuromorphic architecture, a protocol like the Address Event Representation (AER) is essential. According to AER, a spike is represented in the form of an event containing the address of either the neuron that generated it, or the neurons that the spike is destined to. This allows the neurons to be triggered only when there is an event associated with them.

For a multi-class recognition task, SNNs typically have one neuron per class at their output layer. The winning class is, for example, the one corresponding to the neuron that produced the largest number of spikes within a given time interval or the neuron that fired first a spike.

III. State-of-the-art on SNN testing

SNN hardware implementations inherit to a large degree the remarkable fault tolerance capability of the biological brain. Most faults are benign, that is, they affect a component that does not take part in the computation, they are completely masked thanks to the information propagation through the network, they change the order of the top predicted classes but not the top-1 class, or they lead to inaccurate predictions for only a tiny fraction of the inputs and in this sense they can be tolerated. However, some faults remain critical and can lead to a large drop of correct classification percentage. This fault behavior has been demonstrated in several recent fault injection experiments at software level [9]–[16] and in actual neuromorphic hardware [17].

At software level, faults are injected in synapses and neurons assuming that each component can fail independently. In [18], neuron fault simulation is performed at transistor-level considering defects and process variations, in order to extract failure modes. Main failure modes include a dead neuron, i.e., its output is stuck-at a value independently of the input activity, a saturated neuron, i.e., it produces a non-stop output spike train even in the absence of input activity, and timing variations in the output spike train, such as variations in the time-to-first spike and spiking frequency. Dead and saturated responses can be modelled at software by modifying directly the output spike train, while timing variations can be induced in various ways, for example by modifying the neuron's membrane threshold. Common synapse faults include disabled, stuck-weight or saturated-weight synapses. Given that synapse weights are stored as digital words in a memory while at software are handled as real values, a hardware-aware synapse fault model in software is to digitize the real value, perform bit flips, and then reconvert to a real faulty value.

At hardware-level, traditional fault models such as gate-level stuck-at faults, delay faults, or cell-aware defects can be used. However, studies so far have considered single and multiple bit-flips in the memories storing the various parameters of the network, such as synapse weights, neuron threshold, leakage, and refractory period, routing of events, etc. [17].

Post-manufacturing functional testing methods are proposed in [13], [16], [19]. The idea in [13], [16] is to identify available input samples or craft new input samples, for example adversarial samples [13], that can sensitise faults and output a prediction that differs from this of the nominal fault-free network. A compact set of input samples can be generated achieving this objective. In [19], a built-in self-test for biological spiking neurons is proposed where the neuron is exercised from its bias voltages to span the entire range of operation and output all possible firing patterns. If a pattern is missing, then the neuron is labelled faulty.

Fault tolerance techniques can be proactive or reactive, and typically each can address a subset of fault types and locations. Proactive techniques aim at making the SNN tolerate by design a number or certain types of faults. One approach is to perform fault-aware training where faults are injected during training epochs, for example as synapse weight perturbations, aiming at maximizing simultaneously accuracy and fault tolerance [11], [12], [20]. A second approach is to derive the memory fault map via testing, then prioritize placing of Most Significant Bits (MSBs) of network parameters on non-faulty memory cells [12]. A third approach is to adopt training algorithms that naturally offer fault tolerance [11], [14]. For example, in [14], it is shown that training with dropout can nullify the effect of dead neuron faults and timing variations in all layers except the very last ones. At hardware-level, a proactive technique is to perform Triple Modular Redundancy (TMR) for the most critical layers, with priority given on the last layer [14].

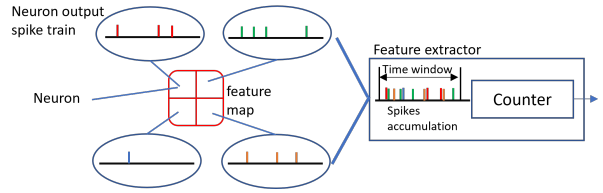
Reactive fault tolerance techniques are implemented at hardware-level and, in general, are composed of two mechanisms, namely a self-test mechanism for fault detection and a fault-mitigation strategy. One approach is to focus on the most lethal faults, namely neuron saturation and large synapse weight increases. For neuron saturation, it is proposed to implement a symptom detector attached to each neuron and silence the neuron if it exhibits saturation behavior [14], [15]. For large synapse weight increases, it is proposed to replace the weight with a pre-defined value, i.e., a zero value [15]. These fault-mitigation approaches essentially translate a critical fault into a benign fault. Finally, another approach is to perform on-line re-learning by disabling components for which re-learning cannot make up for the damage [9].

The reader is referred to [21] for a survey on testability and dependability approaches for AI hardware including SNNs.

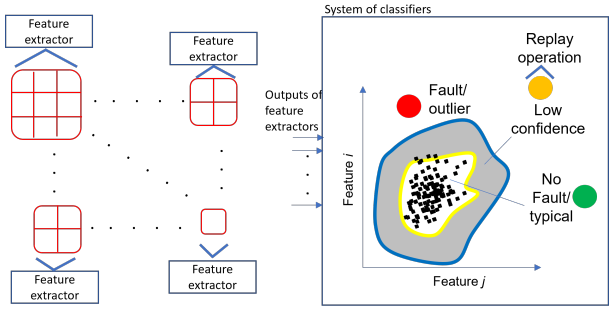
IV. Proposed testing approach

Summarizing the state-of-the-art, we make the following observations:

1) Proactive fault tolerance methods [11], [12], [14], [20] cannot compensate the effect of all faults, thus a number



(a) Feature extractor at feature map-level.



(b) Network-level test.

Fig. 1: Principle of operation.

of faults remain critical and a dedicated test procedure is desired.

2) On-line concurrent error detection is implemented at the component-level, i.e., checking the status of synapses and neurons individually [14], [15], [19]. This approach may result in large overhead.
3) Functional testing based on selected fault-sensitizing inputs [13], [16] is primarily a post-manufacturing testing approach but can also be executed on-line periodically in idle times by storing the functional tests in an on chip memory. Besides the memory overhead, the assumption is that the memory remains fault-free. Moreover, this approach does not guarantee high safety standards as it misses transient errors and detects permanent errors with latency.
4) Testing approaches are demonstrated for hardware-level fault detection only, while their utilisation for outlier and corner input detection is not studied so far.

The primary objective of the proposed testing approach is to detect in real-time any abnormality in the SNN operation, either it is due to a fault occurring or due to an outlier or corner input.

Testing, in general, can be viewed as checking a set of symptoms that point to abnormal operation. We postulate that defining symptoms at the output of neurons is a good strategy since information flows in the form of spike trains and for the SNN prediction to be affected the output spike train of at least one neuron in the network should be appreciably altered.

However, checking the output of every single neuron results in test resources with large overhead. To this end, we propose defining symptoms at a higher-level of hierarchy in the network, specifically at the output of each feature map. By construction, according to the AER protocol, a feature map outputs a flow of spiking events $e(t, d)$, where t is the time of the event and d is the sender or the recipient neuron coordinates. Illustrated in Fig. 1a, we propose to project the spike events of neurons of the feature map in time, consider a pre-defined time window that is dependent on the duration of the input, and define a *test parameter* at the feature map-level equal to the count of accumulated spike events during the time window. The premise is that an abnormal operation will be manifested in the cumulative spiking activity at the output of at least one feature map both in terms of the number of produced spikes and the spike frequency, causing some test parameters to drift away from their expected values. This drift is a *symptom* of abnormal operation. In this way, we drastically reduce the test parameter dimensionality from the neuron size to the feature map size. In fact, it may not be necessary to consider all feature maps since an abnormal spiking activity at the output of a feature map will propagate and spread through the network, thus it may be detectable at the output of feature maps in the next network hierarchy levels.

Next step is making a test decision based on the test parameters. We postulate that checking their combination across the network can be a better test criterion as opposed to checking them individually. This can be done by training a single one-class classifier to map the test parameters to an one-bit test decision addressing the complete network. In machine learning terminology, the test parameters serve as the input features of the classifier, not to be confused with the feature extraction performed by each feature map in the SNN. The classifier is trained on the fault-free network using the available training input samples. Each input is presented to the network and test parameters are collected at the outputs of the feature maps. This is an one-off effort that is already spent during training. The classifier will learn the area in the test parameter space that corresponds to normal operation, enclosing it with a classification boundary, as shown for example with the yellow classifier in Fig. 1b. In abnormal operation, the combination of test parameters will drift outside the classification boundary and the classifier will flag an error detection.

The performance of a classifier is assessed based on two metrics, namely false negatives or *test escapes*, i.e., abnormal operation goes undetected, and false positives or *overkill*, i.e., flagging an error when there is actually none. However, a single classifier is likely incapable of perfectly distinguishing normal from abnormal operation and is bounded to making errors. Using a single classifier, the classifier establishing the optimal trade-off between test escapes and overkill would have been decided based on test economics.

To this end, we adopt a two-tier test approach originally proposed for analog circuits [22]. We propose to avoid using a single classifier making a deterministic decision and instead use a system of two classifiers, as shown in Fig. 1b. The yellow classifier is designed to be *strict*, i.e., in its inner area it encloses only feature patterns corresponding to normal operation. The blue classifier is designed to be *lenient*, i.e., feature patterns falling in its outer area for sure correspond to abnormal operation. If the decision of the two classifiers agrees, i.e., the footprint of the feature pattern lies into the inner area of the yellow classifier or into the outer area of the blue classifier, in other words it lies outside the grey zone

between the two boundaries, then this decision is deterministic and can be trusted. In contrast, if the footprint lies into the grey zone, then the test decision has low confidence. Essentially, the grey zone serves as a guard-band.

To deal with low confidence decisions, we need an extra fast test to make a final decision with incontestable accuracy. For this purpose, we investigate the different scenarios to understand how the system of the two classifiers responds in each case. The input of the SNN can be typical or can be an outlier or corner input that is foreign to the bulk of the training set and, thereby, is prone misprediction. On the other hand, the SNN can be fault-free or contain a fault. The case of fault occurrence or an outlier or corner input will be flagged by the yellow classifier by construction. The problem lies in the fact that the yellow classifier can also inadvertently flag an error for a typical input and a fault-free SNN.

We observed experimentally that this latter scenario happens due to the stochasticity of the SNN. More specifically, a neuromorphic design is by nature asynchronous, meaning that a neuron might receive a spike event at its input or fire one at its output anytime. On the other hand, the controlling processor operates synchronously based on a clock. The synchronization between the two could potentially create some micro-delays in their communication, which propagate during the inference of the SNN. Given that the decision-making in a SNN is based on the temporal characteristics of the spike trains, these delays result in a variance of the triggering time of neurons and, thereby, in a stochasticity at their output spike trains.

To understand the effect of stochasticity, using our case study described in Section V, we repeated the SNN inference multiple times for each sample of the training set. We observed that the scenario where the yellow classifier flags an error for a typical input and fault-free SNN occurs for a handful of repetitions for a few samples. The footprint of the feature pattern of these samples lies closer to the boundary of the yellow classifier, thus they can be marginally misclassified.

Based on this observation, to make a final decision when the system has low confidence, the strategy that we propose is to perform one replay operation presenting the same input sample to the SNN a second time. If now the yellow classifier predicts normal operation, then the SNN prediction can be trusted to be correct. Otherwise, the system flags an error.

V. Case Study

A. Convolutional SNN

As case study we use a convolutional SNN designed for recognizing the symbol on poker cards [23]. The dataset is generated by presenting successively a deck of 40 cards in front of a DVS during 0.96 seconds, and considering their symbol centered to a 32×32 pixel window. The SNN architecture shown in Fig. 2 consists of 4 convolutional layers SC1, SC2, SC3, and SC4 and 2 pooling layers SP1 and SP2 used to downsample the feature maps of layers SC1 and SC2, respectively. Fig. 2 also indicates the number of feature maps per layer, the feature map sizes, and the receptive fields on each feature map.

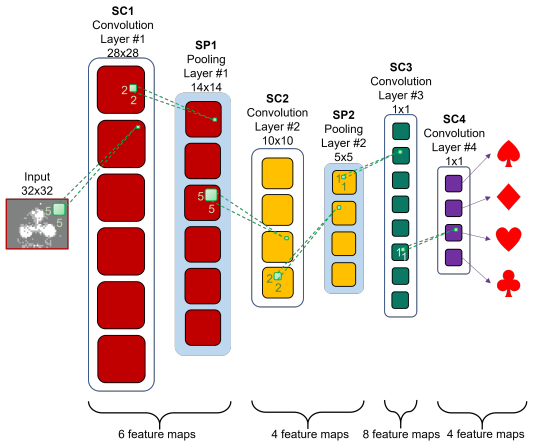


Fig. 2: Convolutional SNN for poker card symbol recognition.

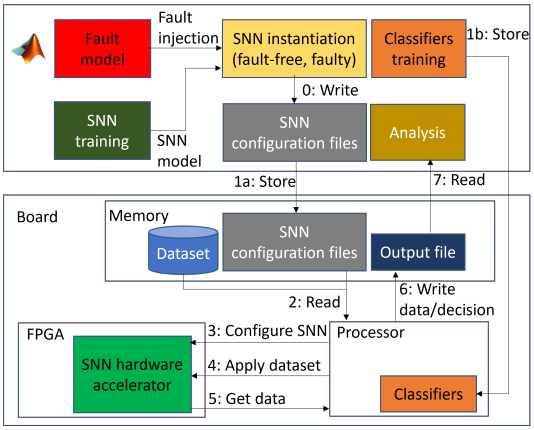


Fig. 3: Embedded neuromorphic hardware accelerator platform.

B. Hardware platform

Fig. 3 presents the architecture of the hardware platform that embeds the testable SNN. The platform is implemented on the Zynq®UltraScale+™ MPSoC ZCU104 FPGA board by Xilinx, Inc., and consists of four main parts:

- *SNN hardware accelerator*: The SNN is designed in VHDL using as building block to implement a feature map a generic event-driven configurable convolutional node [24]. The main parts of the node are the convolutional unit, the router to handle the transmission of the spike events from their origin to their destination, a configuration block for programming the synapse weights, neuron parameters (i.e., threshold, leakage, and refractory period), and feature map parameters (i.e., size and center), and memory to store all these parameters. The SNN is configured in a 2-D mesh of nodes and is implemented on the FPGA of the board.
- *Software*: A MATLAB framework is designed with routines to generate the SNN configuration file, perform the fault injection to generate the configuration files of faulty SNN instances, train the classifiers, and analyse the results to compute the confusion matrix of the classification.
- *Memory*: The memory stores the input samples dataset, the SNN configuration files, and the output file to be processed off-line.

- *Controlling processor*: The processor of the board uses one core for handling the configuration of the SNN instances in batch mode, feeding the input samples to the SNN for each experiment, and storing the SNN output (i.e., spike trains produced by all feature maps). An extra core is used to process the spike trains to extract the features used in the classification, and to host and run the two classifiers.

C. Classifiers

Each classifier is implemented with a Support Vector Machine (SVM) using a Radial Basis Function (RBF) kernel. The two hyper-parameters are ν, which controls the trade-off between overfitting and generalization of the SVM in one-class learning, and γ, which is the coefficient of the RBF kernel [25]. We use the cross-language LIBSVM library [26]. The two SVM models are trained in MATLAB and then they are loaded by the C application running on the processor.

D. Dataset categorization

To account for the SNN stochasticity, for a given input sample, the SNN inference is repeated multiple times and the samples are categorized as follows:

- *Group 1*: Samples whose class is consistently correctly predicted.
- *Group 2*: Samples whose class is consistently erroneously predicted.
- *Group 3*: Samples that are ambiguously predicted during the multiple repetitions due to the SNN stochasticity.

The two SVMs are trained using samples in Group 1. As we care about the impact of faults when the SNN correctly predicts an input sample, the fault detection capability of the SVMs is assessed on Group 1 only. Group 2 comprises the outlier input samples that the SNN did not learn to predict correctly after training. Group 3 comprises corner input samples for which the SNN prediction has high variance and can end up being incorrect.

E. Fault model

The fault model consists of permanent bit-flips in the memories that store the various SNN parameters (i.e., synapse weights, neuron parameters, feature map parameters) and its routing configuration. We consider two scenarios, namely single bit-flips across different bit positions and multiple bit-flips uniformly distributed with a Bit Error Rate (BER) probability from 10^{-6} to 10^{-1}. In total, we consider 7402 SNN instances with single faults and 6278 SNN instances with multiple faults.

To assess the criticality of a fault, we consider its impact on the SNN classification result. Faults are categorized into:

- *Critical faults*: For a given input sample, a fault is critical if the predicted class of the faulty SNN is different than this of the fault-free SNN.
- *Benign faults*: For a given input sample, a fault is benign if the predicted class of the faulty SNN matches this of the fault-free SNN.

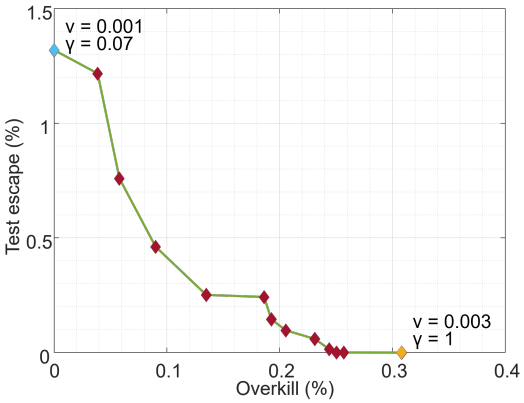


Fig. 4: Pareto front curve test escape vs. overkill for SVMs trained with different pairs of hyper-parameter values ν and γ.

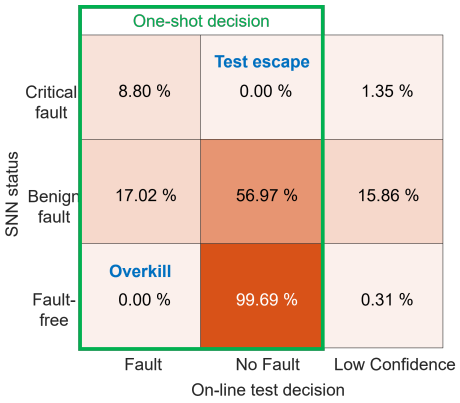


Fig. 5: Performance of system with two SVMs.

VI. Experimental Results

First we consider a single SVM and we assess the performance based on the resultant test escape and overkill. The trade-off is explored in Fig. 4 showing the Pareto front by training different SVMs while varying the combination of ν and γ values. The two end points marked in the Pareto front correspond to test escape for zero overkill and overkill for zero test escape, respectively. These two SVMs are the selected blue and yellow classifiers, respectively, shown in Fig. 1.

Fig. 5 shows the performance of the system of two SVMs for samples in Group 1. Rows correspond to the SNN status (i.e., critical fault, benign fault, fault-free) and the columns correspond to the on-line test decision. Out of all faults, $(8.8 + 0 + 1.35) = 10.15\%$ are critical and $(17.02 + 56.97 + 15.86) = 89.85\%$ are benign. As it can be seen, $(8.8/10.15) * 100 = 86.7\%$ of critical faults are detected in real-time, while test escape and overkill are both zero for the one-shot decisions. However, the system has low confidence for $(1.35/10.15) * 100 = 13.3\%$ of critical faults and 0.31% of fault-free inferences. Regarding benign faults, $(17.02/89.85)*100 = 18.94\%$ are proactively detected, $(56.97/89.85) * 100 = 63.41\%$ are classified as no fault, and for $(15.86/89.85) * 100 = 17.65\%$ the system has low confidence. With a replay operation all uncertainties are lifted. The 0.31% of fault-free inferences that were previously flagged as low-confidence are now flagged as "no-fault" since

their footprint jumps inside the yellow boundary. Whereas, the inferences with the 13.3% of critical faults and 17.65% of benign faults that were previously flagged as low-confidence, are now flagged as "fault" since their footprint remains inside the grey zone.

Regarding the outlier and corner samples from Groups 2 and 3, the yellow SVM flags an error in all cases, thus the system successfully warns when SNN outputs incorrect predictions. Considering the system of two SVMs, all samples in Group 3 and 59.98% of samples in Group 2 lie in the low-confidence grey zone, while for the rest 40.02% of samples in Group 2 an error is flagged directly. The uncertainty is lifted with the replay operation as all these samples remain in the grey zone.

As a final note, deterministic one-shot decisions are completed without delaying the next SNN inference, while whenever there is a low-confidence decision there is a delay equal to the time of one inference due to the replay operation. The SVMs run in software on a second processor core, in parallel with the SNN operation. Thus, they are totally transparent to the SNN without interrupting or interfering with it. The only overhead in our hardware implementation is the utilization of the extra processor core dedicated to the SVMs that increases power consumption.

VII. Conclusions

We presented a generic on-line testing methodology virtually applicable to any SNN hardware accelerator design and cognitive task. Two classifiers monitor the cumulative spike count at the output of feature maps of the SNN and are trained to detect in real-time outlier or corner inputs that are prone to misprediction, as well as hardware-level faults. This is achieved without generating any overkill thanks to the simultaneous assessment of the confidence in the decision. Whenever the confidence is low, a single replay operation suffices to resolve the ambiguity and make an accurate final decision. The methodology is fully demonstrated on a custom FPGA-based neuromorphic hardware platform. It is shown that it enables trustworthy operation with zero-latency transparent decisions for over 99.6% of the SNN inferences, while for the rest the decision is made with a delay of one inference. The only overhead is the power consumption from the utilization of a second processor core dedicated to the integration of the two classifiers.

References

[1] D. V. Christensen *et al.*, "2022 roadmap on neuromorphic computing and engineering," *Neuromorph. Comput. Eng.*, vol. 2, no. 2, May 2022.

[2] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner, "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May/Jun. 2010, pp. 1947–1950.

[3] E. Painkras *et al.*, "SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation," *IEEE J. Solid-State Circuits*, vol. 48, no. 8, pp. 1943–1953, May 2013.

[4] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014.

[5] 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.

[6] G. K. Chen, R. Kumar, H. E. Sumbul, P. C. Knag, and R. K. Krishnamurthy, "A 4096-neuron 1M-synapse 3.8-pJ/SOP spiking neural network with on-chip STDP learning and sparse weights in 10-nm FinFET CMOS," *IEEE J. Solid-State Circuits*, vol. 54, no. 4, pp. 992–1002, Apr. 2019.

[7] C. Frenkel, M. Lefebvre, J.-D. Legat, and D. Bol, "A 0.086-mm^2 12.7-pJ/SOP 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm CMOS," *IEEE Trans. Biomed. Circuits Syst.*, vol. 13, no. 1, pp. 145–158, Feb. 2019.

[8] G. Indiveri *et al.*, "Neuromorphic silicon neuron circuits," *Front. Neurosci.*, vol. 5, May 2011, Article 73.

[9] A. Hashmi, H. Berry, O. Temam, and M. Lipasti, "Automatic abstraction and fault tolerance in cortical microachitectures," in *Proc. ACM/IEEE Annual Int. Symp. Comput. Archit. (ISCA)*, Jun. 2011, pp. 1–10.

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

[11] 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.

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

[13] H.-Y. Tseng, I-W. Chiu, M.-T. Wu, and J. C.-M. Li, "Machine learning-based test pattern generation for neuromorphic chips," in *IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2021.

[14] 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, pp. 743–748.

[15] R. V. W. Putra, M. A. Hanif, and M. Shafique, "SoftSNN: Low-cost fault tolerance for spiking neural network accelerators under soft errors," in *Proc. 59th Design Autom. Conf. (DAC)*, Jul. 2022, p. 151–156.

[16] S. A. El-Sayed, T. Spyrou, L. A. Camuñas-Mesa, and H.-G. Stratigopoulos, "Compact functional testing for neuromorphic computing circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 2022.

[17] T. Spyrou, S. A. El-Sayed, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Reliability analysis of a spiking neural network hardware accelerator," in *Proc. Design Autom. Test Europe Conf. (DATE)*, Mar. 2022, pp. 370–375.

[18] 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.

[19] 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.

[20] R. V. W. Putra, M. A. Hanif, and M. Shafique, "SparkXD: A framework for resilient and energy-efficient spiking neural network inference using approximate DRAM," in *Proc. 58th Design Autom. Conf. (DAC)*, Dec. 2021, p. 379–384.

[21] F. Su, C. Liu, and H.-G. Stratigopoulos, "Testability and dependability of AI hardware: Survey, trends, challenges, and perspectives," *IEEE Des. Test*, vol. 40, no. 2, pp. 8–58, Apr. 2023.

[22] H.-G. Stratigopoulos and Y. Makris, "Error moderation in low-cost machine-learning-based analog/RF testing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 2, pp. 339–351, Feb. 2008.

[23] 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.

[24] 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.

[25] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Comput.*, vol. 13, no. 7, pp. 1443–1471, Jul. 2001.

[26] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, pp. 27:1–27:27, Apr. 2011, Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.