Image Test Libraries for the in-field test of ultra-low-power devices

Image Test Libraries for the in-field test of ultra-low-power devices

Antonio Porsia
Politecnico di Torino
DAUIN
Turin, Italy

Giacomo Perlo
Politecnico di Torino
DAUIN
Turin, Italy

Annachiara Ruospo
Politecnico di Torino
DAUIN
Turin, Italy

Ernesto Sanchez
Politecnico di Torino
DAUIN
Turin, Italy

***Abstract*—In recent years, research and technology advancements have driven exponential growth in the adoption of Artificial Intelligence (AI)-based systems, even in safety-critical contexts such as autonomous driving and healthcare applications. The joint effort of academia and industry has yielded techniques and standards with the objective of ensuring the safe operation of AI-based technology. In the specific context of Convolutional Neural Networks (CNNs) running on GPUs, Image Test Libraries (ITLs) have been proposed as an effective method for performing on-line functional testing of GPU multipliers. This is achieved by launching the inference of a set of test images containing a set of ATPG-generated functional test patterns. However, while the demand for computational power for DNN models is constantly increasing, another branch of Machine Learning (ML) research, namely TinyML, focuses on minimizing the computational requirements of DNN models in order to bring AI capabilities to edge devices, whose constraints on power usage, memory space and processing power do not allow for the deployment of conventional DNN models. This research work aims to adapt the ITL technique to CNNs running on ultra-low-power edge hardware, while also overcoming some limitations of GPU ITLs. Experimental results demonstrate that a single test image generated using the proposed method is capable of detecting 96.01% of stuck-at faults occurring in the 32-bit integer multiplier of a RISC-V-based ultra-low-power System-on-Chip executing a quantized CNN.**

I. Introduction

Advances in TinyML (Tiny Machine Learning) have greatly reduced the minimum computational requirements needed to execute a Machine Learning (ML) model, allowing to enhance edge devices with AI capabilities. This interest towards efficient AI computation at the edge goes hand in hand with the wide interest towards open architectures such as RISC-V. The adoption of RISC-V for ML workloads spawned a plethora of devices with different configurations and characteristics, from small ultra-low-power platforms [1] to SoCs with over 1,000 cores designed to compete with well-established high-performance solutions such as GPUs [2]. Since ML technologies are increasingly used in safety-critical environments, such as self-driving cars [3] and healthcare applications [4], the academic and industrial research communities are striving to ensure the reliability and safety of these technologies, as testified by the ISO 26262 standard commonly followed in the automotive field, or the ISO/IEC CD TR 5469 standard concerning the functional safety of AI systems. On-line functional testing strategies have become a common solution in the automotive sector [5], for instance with the adoption of Software Test Libraries (STLs), a set of assembly instructions that are able to excite and detect permanent faults on the targeted hardware. While there is a clear advantage in having no hardware overhead, alternating the execution of STL routines and ML models may disrupt the overall performance of the system [6]. To overcome these problems, Image Test Libraries (ITLs) have been proposed as a viable solution to test systems executing neural networks designed for computer vision tasks [7] [8] [9]. In particular, ATPG-based ITLs [7] [8] have proven to be able to excite permanent hardware faults injected in the gate-level representation of the multipliers of a GPU executing a Convolutional Neural Network (CNN), although faults are observed on an intermediate result and not on the network output. This work aims to adapt the ATPG-based ITL technique to edge devices executing TinyML models, as well as improve the technique to be able to observe faults directly on the network's prediction. Experimental results show that the adapted technique is able to detect 96.01% of permanent stuck-at faults occurring in the integer multiplier of a RISC-V-based System-on-Chip executing a Lenet5 CNN quantized to 8-bit integers. The paper is organized as follows: section II gives the reader an introduction on concepts used in this work, such as ITLs and quantized neural networks; section III describes the adapted and improved ATPG-based ITL technique; section IV describes the experimental setup and presents experimental results; section V draws conclusions and future directions.

This publication is part of the project PNRR-NGEU which has received funding from the MUR – DM 118/2023

II. Background

A. Image Test Libraries (ITLs)

In a context where the right blend of performance and reliability is a key requirement, a set of test images, i.e., an ITL, may constitute a viable alternative to conventional on-line self-testing techniques. While methods such as STLs require stopping the application to run test routines, hence causing a performance loss, ITLs leverage the inference process, leaving the model untouched and not incurring in performance penalties. This paragraph will offer an overview of two types of test images that are particularly relevant for this work.

1) ATPG-based Image Test Library: ATPG-based test images were proposed for the on-line detection of stuck-at faults occurring in the multipliers of a GPU running a CNN [7]. The ATPG-based ITL construction process consists of three steps:

1) **Dataflow algorithm extraction** – GPU scheduling policy and convolution algorithm are analyzed to derive a mapping that associates each GPU multiplier with a list of multiplications, i.e., the dataflow algorithm. Each element of the output feature map of the first convolutional layer is associated to the multiplier that computes it and to the list of multiplications that are needed to compute it.
2) **ATPG-based pattern generation** – for each multiplier, weights are extracted from the multiplications list and used as constraints for an ATPG process. The result is a list of input patterns, each associated to a specific weight.
3) **Self-test images generation** – leveraging the dataflow algorithm obtained in step 1, the input patterns generated in the previous step are arranged into a sufficient number of images to contain them all. Each input pattern must be placed in a position where it is guaranteed to be multiplied by the targeted multiplier with the corresponding constraint weight at least once.

Faults are observed at the output of the first layer, with the obvious downsides of requiring an elementwise comparison between large tensors and requiring white-box access to the model. Nonetheless, ATPG-based ITLs allow accurate detection of hardware faults with a low number of test images. Experimental results demonstrate that ATPG-based ITLs are capable of achieving an average test coverage (TC) of 94.74% and 95.46% across all GPU multipliers for ResNet-20 and DenseNet-121 models, respectively, using 6- and 8-image ITLs.

2) Borderline test images: El-Sayed et al. [9] proposed a method to select test images from the training dataset targeting neuromorphic circuits for Spiking Neural Networks. Their technique consists in ranking images according to a quality metric q_i that measures the fault detection capability of the image. q_i is defined as $q_i = \frac{1}{n_i^{(1)} - n_i^{(2)}}$, where $n_i^{(1)}$ and $n_i^{(2)}$ are the spike counts associated to the top-1 and top-2 classes, respectively. In other words, the technique consists in selecting *borderline* images that lie on the boundary between two classes, with the underlying hypothesis that they are more sensible to hardware faults. After q_i has been computed for all images in the training and testing sets, the test library is constructed by adding the first n top-ranked images to the test library, where n can be adjusted as needed. The authors used this technique to detect behavioral-level and hardware-level faults, achieving detection of 100% of *critical* faults, i.e., faults that alter the network's response above a certain tolerance margin, using image test sets of various sizes in the order of tens of images.

B. TinyML

TinyML is a branch of Machine Learning research whose main objective involves enhancing resource-constrained devices, usually battery-powered Microcontroller Units (MCUs), with ML capabilities. This approach allows to analyze sensor data on the spot instead of offloading work to remote compute servers and eliminates various security, privacy and connectivity issues [10]. Use cases include safety-critical systems, such as healthcare devices [4] and self-driving vehicles [3].

1) Quantization: Model compression techniques aimed at reducing model size and computational requirements are among the main techniques that allow ML inference at the edge. The technique on which this work focuses is quantization, in which model weights and activations are stored in a low precision format, such as 16- or even 8-bit integers [11], with the objective of reducing the network size while minimizing accuracy loss. Model compression achieved with quantization allows for the deployment of models on edge accelerators [12].

Among the various quantization schemes available, the most common is asymmetric affine quantization [11]. In this scheme, given an input scale *S* and a zero point *Z*, a 32-bit floating-point number *r* can be quantized to its *n*-bit integer representation *q*:

$$q = \left\lfloor \frac{r}{S} \right\rfloor + Z \tag{1}$$

2) Reliability of quantized models: The effects of quantization on the reliability of Deep Neural Networks (DNNs) have been analyzed in recent studies, showing that quantized neural networks often outperform their 32-bit floating-point counterparts in terms of robustness. In particular, quantized neural networks have been shown to be more resistant to transient faults in the network's weights, modeled as random bit-flips [13]. Another study on binary neural networks, an extreme case of quantization where weights can only assume the values 1 or -1, shows that aggressive quantization reduces the sensitivity of the network to soft errors but increases the rate of misclassifications [14].

III. Proposed Method

The proposed method builds upon the ATPG-based ITL, integrating ideas from the other two methods described in Section II-A. In particular, ATPG-generated patterns are overlaid on top of a randomly-initialized image optimized using gradient descent to produce confidence scores all extremely close to each other, i.e., a *borderline image*. Experiments performed on borderline images [9] strongly suggest that such images are particularly sensible to perturbations. Hence, taking such an image and enhancing it with ATPG-generated test patterns should produce an image with high fault detection capabilities. However, one of the limitations of ATPG-based test images is that they require to observe faults on an intermediate result, namely the output feature map of the first convolutional layer. Nonetheless, borderline images have the additional benefit of *propagating the fault's effect up to the predicted label*. By leveraging this characteristic, faults stimulated by ATPG test patterns should strongly affect the prediction. However, overlaying test patterns on top of an existing borderline image

will affect the predicted class scores, potentially affecting its "borderline-ness". To solve this problem, the proposed solution is to produce a borderline image *from scratch*, by (i) generating a random image, (ii) overlaying the test patterns on top of it and then (iii) optimizing the rest of the image, excluding pixels containing test patterns, to produce a borderline prediction.

A. ITL construction

Unlike ATPG-based ITLs for GPU multipliers, the proposed ITL targets the integer multiplier of a single-core MCU, hence greatly simplifying the process. For the sake of clarity, the same steps described in [7] will be used. However, there are some major differences. Since the target is a single-core single-thread architecture, all multiplications are computed by the same multiplier. For this reason, the first two parts of the dataflow algorithm described for GPUs, namely thread-core mapping and workload-thread mapping, can be skipped, leaving only the convolution algorithm.

The ATPG-based pattern generation phase remains largely unchanged, with some small precautions that will be described in Section IV. The output of this phase is a set of input patterns in the integer representation used by the quantized target network.

The only phase that presents major differences is the self-test image generation phase. The first objective of this phase consists producing an image such that the similarity between the confidence scores produced by the inference and an array of probabilities equally distributed across the available classes is maximized. That is, if the number of classes is N_c, the target output $\mathbf{z}$ of the model should be $\mathbf{z} = \left(\frac{1}{N_c}, \ldots, \frac{1}{N_c}\right)$

To achieve this task, a random image is generated and gradient descent is used to optimize the image. Since the training objective is to produce an output vector that is as similar as possible to the target vector, cosine similarity is used as a loss function, since it is particularly suited for tasks that involve minimizing the distance between two vectors. The cosine similarity is computed on the output of the Softmax activation function, i.e., on the output confidence scores $\hat{\mathbf{z}}$, and is defined as it is used in the Tensorflow framework [15]:

$$\text{CosineSimilarity}(\mathbf{z}, \hat{\mathbf{z}}) = -\frac{\mathbf{z} \cdot \hat{\mathbf{z}}}{\|\mathbf{z}\| \|\hat{\mathbf{z}}\|}$$

Since this function outputs values closer to -1 as $\mathbf{z}$ and $\hat{\mathbf{z}}$ get closer, the learning objective becomes finding $\hat{\mathbf{z}}$ such that the cosine similarity is minimum. Note that since in this phase operations are performed on floating point numbers, the original pre-quantization model is needed. Furthermore, since optimization is performed on an image in floating point format, the ATPG patterns, which are in integer format, must be transformed accordingly using input quantization parameters. Given input scale S and zero point Z, the patterns must be converted in floating point format to ensure that when the resulting test image is quantized, they are converted back to their original value. In other words, given an integer test pattern p, a floating-point value r has to be computed such that quantizing it produces p. This can be achieved by computing $r = S \cdot (p - Z)$. It is easy to observe that substituting r in equation 1 yields exactly p.

Algorithm 1 ATPG-enhanced Borderline Image Generation

Inputs: P - ATPG-generated input patterns in quantization format; S, Z - Input quantization parameters; N_c - N. of classes; N_t - Training steps; *Model* - Pretrained CNN model before quantization
Outputs: T - Test image

```
1: T ← GenRandomImage
2: P_float ← Dequantize(P, S, Z)
3: Target ← (1/N_c, ..., 1/N_c)
4: for step ← 0 to N_t do
5:     OverlayPatterns(T, P_float)
6:     Prediction ← Model(T)
7:     Loss ← CosineSimilarity(Target, Prediction)
8:     Optimize(T, Loss)
9: end for
10: OverlayPatterns(T, P_float)
11: T ← Quantize(T, S, Z)
12: return T
```

Algorithm 1 describes the steps needed to generate the image. As preliminary steps, a random image is generated (line 1), the floating-point representation of ATPG-generated test patterns is computed (line 2) and the target output is computed (line 3). At each step of the optimization loop (lines 4-9) the patterns are overlaid on top of the image (line 5). Then the image is passed to the original pre-quantization model, the prediction is used to compute the cosine similarity loss and the image is optimized (lines 6-8).

After the loop, the floating-point patterns are overlaid on top of the image (line 10) and the image is quantized (line 11). At the end of the procedure, T is an ATPG-enhanced borderline test image.

B. ITL Validation

To validate test images generated with the method described above, the hybrid fault simulation method described in [7] and [8] has been adapted to quantized models. The steps needed to simulate a faulty convolutional layer are: (i) enumerating the multiplications performed in order to compute each output element; (ii) computing faulty multiplications by fault simulation; (iii) computing the faulty input corresponding to each faulty multiplication by performing a division; (iv) overlaying faulty inputs on top of the input feature map; (v) passing the faulty input feature map through the target layer to obtain the faulty output feature map.

This method builds upon the idea that a multiplication performed with a faulty multiplier, therefore having a faulty output $\hat{o}$, is equivalent to a multiplication computed with a clean multiplier with a faulty input $\hat{i}$. Unfortunately, the idea of performing divisions to build a faulty input feature map from faulty multiplication outputs cannot be used in this case. In floating point arithmetic, given a faulty multiplication $I \cdot W =_f \widehat{O}$ that produces a faulty output, it is usually possible to compute the corresponding faulty input $\widehat{I}$ by

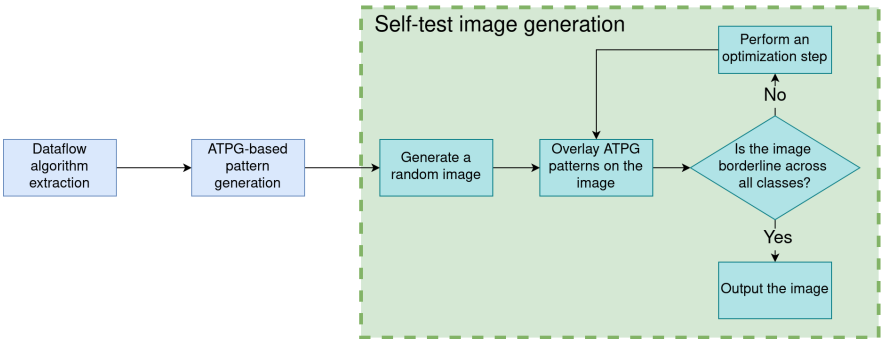

Fig. 1: ATPG-enhanced borderline image construction process

Algorithm 2 Faulty Convolution

Inputs: MUL - Target multiplier; f - Target stuck-at fault of MUL; I - Input feature map; M - Input-weight pairs representing multiplications performed by MUL during the targeted convolution

Outputs: O_f - Faulty output feature map corresponding to fault f

1: $O_f \leftarrow []$
2: INJECT(MUL, f)
3: **for** $i \leftarrow 0$ **to** SIZE(O_f) **do**
4: $\quad m \leftarrow$ GETMULTIPLICATIONS(M, i)
5: $\quad \hat{o} \leftarrow$ MULTIPLY(MUL, m)
6: $\quad O_f[i] \leftarrow$ SUM($\hat{o}$)
7: **end for**
8: CLEAN(MUL, f)
9: **return** O_f

performing a simple division. This does not hold for integer arithmetic. Given a faulty integer multiplication $i \cdot w =_f \hat{o}$, the corresponding faulty input $\hat{i}$ cannot be computed with a division, since the result would be rounded down to the nearest integer and multiplying it with w would not produce $\hat{o}$.

For this reason, instead of manually building a faulty input feature map and passing it through the convolutional layer of interest, convolutions are manually computed by collecting the results of faulty multiplications collected for each fault and computing output elements by summing the right faulty multiplications

Algorithm 2 shows the procedure. First, a fault f is injected into the multiplier (line 2). Then, for each element of the output, the multiplications needed to compute it are collected and performed by the faulty multiplier (lines 4-5). The results are then summed and written into the output feature map (line 6).

IV. Experimental Setup and Results

A. Setup

To validate this approach, experiments have been performed by deploying X-HEEP, a RISC-V-based System-on-Chip, on a TUL PYNQ-Z2 FPGA. X-HEEP (eXtendable Heterogeneous Energy-Efficient Platform) is an open-source RISC-V MCU aimed at edge computing that focuses on configurability, extendability and ease of use. One of its main features consists in the possibility of synthesizing it with various RAM sizes,

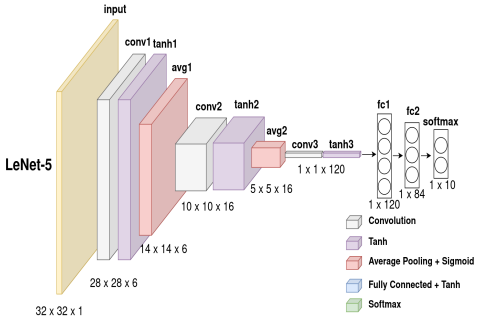

Fig. 2: Lenet5 CNN

bus types and CPUs. Among the available CPUs, cv32e20 (formerly known as Ibex) has been selected. The cv32e20 core allows to choose between three different 32-bit integer multipliers: (i) a single-cycle multiplier and a (ii) fast or (iii) slow multi-cycle multiplier. In this case, the single-cycle multiplier has been used. The proposed test image targets single stuck-at faults in the gate-level representation of the multiplier. ModelSim and Synopsys TestMax have been used to perform fault injections and ATPG pattern generation, respectively.

The Tensorflow Lite for Microcontrollers (TFLM) library has been used to quantize and deploy a Lenet5 CNN (figure 2) on X-HEEP. The model was trained on the MNIST dataset for handwritten digit recognition using Tensorflow, reaching an accuracy of 95.80% on the validation set and a model size of 771 KB. Tensorflow Lite was then used to quantize the model to int8 format, with an accuracy of 95.75% on the validation set and a model size of 69.8 KB.

B. ITL generation

a) Dataflow algorithm extraction: The ITL generation process starts with the dataflow algorithm extraction phase, with the objective of mapping multiplications to the multiplier. As anticipated in section III, the main difference with ATPG-based ITLs for GPU multipliers is that the thread-core mapping and workload-thread mapping parts are not needed. Since TFLM by default computes convolutions using the mathematical definition of convolution, this step consists in the trivial problem of ”unrolling” the convolution into multiplications and additions.

b) ATPG-based pattern generation: The quantized int8 weights of the first convolutional layer are extracted from the Tensorflow Lite model and used as constraints for the

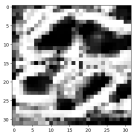

Fig. 3: ATPG-enhanced borderline image for an int8-quantized Lenet5 CNN.

ATPG process. Since the weights are 8-bit integers, but the multiplier works on 32-bit integers, values are sign-extended before using them as constraints. For what concerns the input, the most significant 25 bits are also constrained to be all equal to force the ATPG process to generate a valid sign-extended 8-bit integer. Results of the ATPG process are reported in table II. The test patterns reach a relatively low test coverage of 86.16%, caused by the constraints on input values which do not allow to excite certain paths of the circuit.

c) Self-test images generation: Algorithm 1 has been used to generate a test image with the following characteristics: (i) the output confidence scores produced by both the original and quantized models are equal for all classes, or at least extremely close to each other; (ii) the image contains the test patterns generated in the ATPG phase. A randomly-initialized image has been optimized for 8000 steps using the Adam optimizer with an initial learning rate of 0.1, lowered to 0.01 after 4000 steps. The confidence scores are reported in table I and the generated image is shown in Figure 3. The image needs only 1 KiB of storage, plus 10 additional bytes for the fault-free prediction.

C. ITL validation

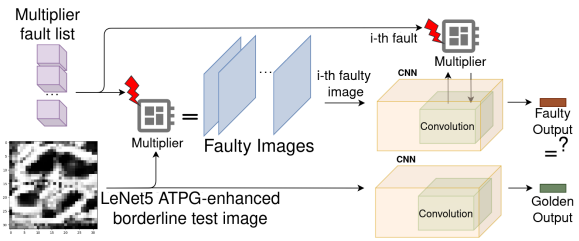

Fig. 4: Faulty inference modeling

Since TFLM does not allow to easily manipulate the internal structure of the model, the whole inference process must be modeled in software by replicating as accurately as possible the internal operations performed by the framework. The approach consists in building a software model of a faulty inference process by substituting each convolutional layer with the faulty convolution described in section III. However, as anticipated in the previous section, it is not possible to replicate exactly the process described in [8]. In particular, the most pressing difficulties come from (i) having to deal with complex activation functions, such as the hyperbolic tangent, computed on quantized values; (ii) the impossibility of passing custom feature maps to hidden layers of the model. For these reasons, some computations have been offloaded to auxiliary Tensorflow Lite models where possible, for example for activation functions and fully-connected layers. The activation auxiliary models contain only the hyperbolic tangent and average pool layers with input and output shapes corresponding to the output shape of each convolutional layer. The fully connected auxiliary model contains the last two Fully Connected layers with the same parameters as the original model, and takes as input the output of the last activation auxiliary model.

Figure 4 shows a graphical representation of the faulty convolution computation.

D. Results

Using the validation procedure just described, fault injections have been performed for different types of images: (i) the borderline image generated using the method described in this paper; (ii) an image filled by repeating the same ATPG-generated test patterns used to build image 1; (iii) a random image from the MNIST dataset; (iv) a randomly generated image; (v) a borderline image selected from the dataset according to the method described in [9]. For each image, the validation procedure has been executed to collect the faulty output confidence scores for each targeted stuck-at fault. To speed up the validation process, a statistically significant number of targeted faults has been selected instead of the full fault list. The minimum number of targeted faults n has been computed using the statistical fault injection process described in [16]. Faults are injected in the gate-level representation of the multiplier used by Modelsim. The total number of possible fault sites where a stuck-at fault can be injected using Modelsim amounts to 1608, hence the population size is equal to $N = 1,608 \cdot 2 = 3,216$. The error margin has been set to 3.8% and the confidence interval to 95%. Therefore, the minimum sample size is equal to $n = 551$ after rounding. After the faulty confidence scores have been produced for each test image and each targeted fault, the images have been compared based on their ability to excite faults and propagate them to the prediction. To this end, for each test image three metrics have been computed, namely:

- **SDC-1** – percentage of outputs with a different prediction from the prediction of a fault-free model
- **SDC-3** – percentage of outputs where the fault-free predicted lbel is not included in the top-3 classes predicted by the faulty model
- **SDC-10%** – percentage of outputs where the confidence score of the top prediction differs by at least 10% with respect to the fault-free top prediction

Results are shown in Table III.

E. Discussion

It can be observed that the SDC-1 percentages are above 90% for all images. This is mainly due to the fact that the space of possible input-weight multiplications in int8 format is quite small: only $2^8 \cdot 2^8 = 65,536$ possible multiplications. For this reason, with a sufficiently high number of multiplications it is possible to test nearly all possible input-weight pairs, including the test patterns found via ATPG. Considering that just the first Lenet5 convolutional layer features $117,600$ multiplications,

TABLE I: ATPG-enhanced borderline image confidence scores

Class	0	1	2	3	4	5	6	7	8	9
Score (original) [%]	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00
Score (quantized) [%]	10.16	10.16	10.16	10.16	10.16	9.385	10.16	10.16	9.385	10.16

TABLE II: Summary of ATPG results for LeNet-5

CNN	**N. of Weights**	**N. of ATPG Patterns**	**TC (%)**
LeNet-5	150	21	86.16

TABLE III: Metrics for each test image

Image	**SDC-1 [%]**	**SDC-3 [%]**	**SDC-10% [%]**
ATPG-enhanced	96.01	98.19	96.92
ATPG-filled	90.40	96.74	89.13
Dataset	94.93	97.64	91.48
Random	93.84	98.37	91.84
Borderline	94.74	99.64	91.67

these numbers are to be expected. However, this does not directly translate to the capability of a test image to propagate faults up to the prediction. In particular, the test image filled with ATPG patterns results to be the worst performing with SDC-1 reaching 90.40%, even though the ATPG patterns are bound to excite each possible stuck-at fault that can affect int8 multiplications. This result is explained by the fact that ATPG patterns may excite faults in the first convolutional layer and affect its output, but if the magnitude of the changes is small, the faulty confidence scores are unlikely to be much different with respect to the fault-free ones, leaving the final prediction unchanged. Furthermore, an image constructed by repeating the same few patterns over and over results in the network repeating the same operations for different parts of the image. Dataset, borderline and random images are able to perform better thanks to their higher variability, which translates to a higher number of multiplications that could be affected by a fault. As for the borderline image, results are in line with the not-borderline dataset and random images and do not show particular improvements except for the SDC-3 case. However, it should be noted that the method described in [9] involves the selection of multiple images, not a single one like in this case. While the metrics may benefit from multiple test images, consideration should be given to the fact that low memory occupation is a fundamental requirement for ultra-low-power embedded systems. Considering the non-borderline dataset image as a baseline for faults that are easy to detect, the proposed ATPG-enhanced borderline test image exhibits a slight improvement, being able to excite and propagate more hard-to-detect faults up to the prediction. If confidence scores are also taken into consideration, the proposed test image shows major improvements over other images in the SDC-10% category, meaning that it is able to cause major differences in the Softmax output in presence of permanent faults.

V. Conclusions

This work described a method to adapt ATPG-based Image Test Libraries to the integer multiplier of an ultra-low-power device running a quantized CNN. It was shown how a single image generated using this method is able to excite up to 96.01% of single stuck-at faults and propagate them up to the model's prediction.

Future work includes a study on the reliability of quantized neural networks in presence of permanent faults using the hybrid software/hardware fault simulation methodology described in [8], as well as an extension of this work to other, more complex CNN architectures, other types of neural networks and different number formats, such as 16-bit integers.

References

[1] S. Machetti, P. D. Schiavone, T. C. Müller, M. Peón-Quirós, and D. Atienza, "X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators," 2024.

[2] D. Ditzel *et al.*, "Accelerating ML Recommendation with over a Thousand RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–23.

[3] M. Bechtel, Q. Weng, and H. Yun, "Deeppicarmicro: Applying tinyml to autonomous cyber physical systems," 2022.

[4] V. Tsoukas, E. Boumpa, G. Giannakas, and A. Kakarountas, "A Review of Machine Learning and TinyML in Healthcare," in *Proceedings of the 25th Pan-Hellenic Conference on Informatics*, ser. PCI '21. New York, NY, USA: Association for Computing Machinery, 2022, p. 69–73. [Online]. Available: https://doi.org/10.1145/3503823.3503836

[5] P. Bernardi, R. Cantoro, S. D. Luca, E. Sánchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Transactions on Computers 65.3*, p. 744–754, 2016.

[6] A. Ruospo, D. Piumatti, A. Floridia, and E. Sanchez, "A suitability analysis of software based testing strategies for the on-line testing of artificial neural networks applications in embedded devices," *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 1–6, 2021.

[7] A. Ruospo *et al.*, "Image Test Libraries for the on-line self-test of functional units in GPUs running CNNs," in *2023 IEEE European Test Symposium (ETS)*, May 2023, pp. 1–6.

[8] A. Bosio *et al.*, "Resiliency Approaches in Convolutional, Photonic, and Spiking Neural Networks," 2024, pp. 1–10.

[9] S. A. El-Sayed, T. Spyrou, L. A. Camuñas-Mesa, and H.-G. Stratigopoulos, "Compact Functional Testing for Neuromorphic Computing Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 7, pp. 2391–2403, Jul. 2023.

[10] D. L. Dutta and S. Bharali, "TinyML Meets IoT: A Comprehensive Survey," *Internet of Things*, vol. 16, p. 100461, Dec. 2021.

[11] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A White Paper on Neural Network Quantization," Jun. 2021.

[12] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018. [Online]. Available: https://arxiv.org/abs/1806.08342

[13] B. F. Goldstein *et al.*, "Reliability Evaluation of Compressed Deep Learning Models," in *2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS)*, 2020, pp. 1–5.

[14] F. Libano, B. Wilson, M. Wirthlin, P. Rech, and J. Brunhaver, "Understanding the Impact of Quantization, Accuracy, and Radiation on the Reliability of Convolutional Neural Networks on FPGAs," *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1478–1484, 2020.

[15] "Tensorflow - Cosine Similarity," https://www.tensorflow.org/api_docs/python/tf/keras/losses/CosineSimilarity, accessed: 2024-11-26.

[16] A. Ruospo *et al.*, "Assessing Convolutional Neural Networks Reliability through Statistical Fault Injections," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.