POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Fault Injection Framework for AI Hardware Accelerators

Original
A Fault Injection Framework for AI Hardware Accelerators / Pappalardo, S., Ruospo, A., O'Connor, I., Deveautour, B., Sanchez, E., Bosio, A.. - ELETTRONICO. - (2023), pp. 1-6. (2023 IEEE 24th Latin American Test Symposium (LATS) Veracruz, Mexico 21-24 March 2023) [10.1109/LATS58125.2023.10154505].

Availability:
This version is available at: 11583/2981738 since: 2023-09-06T15:38:05Z

Publisher:
IEEE

Published
DOI:10.1109/LATS58125.2023.10154505

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

11 November 2026

A Fault Injection Framework for AI Hardware Accelerators

Salvatore Pappalardo[1], Annachiara Ruospo[2], Ian O'Connor[1], Bastien Deveautour[3], Ernesto Sanchez[2], Alberto Bosio[1]
[1],[3]*Univ Lyon, ECL, INSA Lyon, CNRS, UCBL, CPE Lyon, INL, UMR5270, 69130 Ecully, France*
[2]*Politecnico di Torino, Dip. di Automatica e Informatica, Torino, Italy*
Email: [1]{name.surname}@ec-lyon.fr, [2]{name.surname}@polito.it, [3]{name.surname}@cpe.fr

***Abstract*—Deep Neural Networks (DNNs) have proven to give very good results for many complex tasks and applications, such as object recognition in images/videos and natural language processing. Some relevant applications of DNNs are defined by real-time safety-critical systems, which typically require the adoption of DNN accelerators that are usually implemented as systolic arrays. Assessing their reliability is not trivial and may depend on several factors such as the size of the array and the data precision.**

In this paper, we present a cross-layer framework for systolic array DNN accelerators described at RTL level allowing to inject faults at channel granularity for convolutional layers. The basic idea is to simulate the execution of the Channel Under Test (ChUT) at RTL level. Faulty outputs collected from the RTL simulation are then used at software level to complete the execution of the DNN and thus determine the impact of the injected faults at application level. Interestingly, the software execution is more than 100 times faster than the corresponding hardware simulation.

***Index Terms*—DNN Hardware accelerators, Fault Injection, Reliability**

I. Introduction

Nowadays, DNNs are ubiquitous and used in a lot of different applications spanning from object detection/recognition, image segmentation to natural language processing in the context of real-time safety-critical systems.

Dedicated DNN HW accelerators—usually implemented as systolic arrays—are widely used to satisfy performance and power budget constraints of real-time safety-critical systems [1].

However, DNN hardware accelerators may be subject to hardware faults due to several causes: variations in fabrication process parameters, fabrication process defects, latent defects, i.e., defects undetectable at time-zero post-fabrication testing that manifest themselves later in the field of application, silicon aging, e.g., time-dependent dielectric breakdown, or even environmental stress, such as heat, humidity, vibration, and Single Event Upsets (SEUs) stemming from ionization. All these faults may cause operational errors impacting the system reliability, and potentially leading to very negative consequences, especially for safety-critical systems. It is therefore mandatory to evaluate the reliability of hardware accelerators to design dedicated fault tolerance mechanisms.

979-8-3503-2597-3/23/$31.00 ©2023 IEEE

Unfortunately, assessing the reliability of a DNN deployed on hardware accelerators is not a trivial task, since it depends on several factors, such as the size of the systolic array and the data precision.

In the literature, several works target the reliability assessment of DNNs through fault injection, and they are usually classified depending on their abstraction level.

Some works propose to perform Fault Injections (FIs) at software-level [2]–[4] so far. They mainly differ from the software platform (e.g., PyTorch or TensorFlow), the type of faults (e.g., permanent, transient) and their locations (e.g., weights, biases, and operators). The main drawback of software-level FIs is the lack of information of the underlying hardware platform, and therefore they are relatively less accurate. For example, it is not trivial to take into account the size of the systolic array, or the number of processing elements (PEs) in the hardware GPU during the simulation at software-level. On the other hand, the main advantage is the reduced FI time.

For these reasons, several works propose to perform FI at hardware-level by simulating an RTL model of the DNN accelerator [5]–[7]. The reliability assessment is performed by taking into account both the application-level specifications (the DNN weights, inputs, and intermediate values) and the architectural-level ones (the specific data representation and the amount of computational resources, i.e., the PEs). However, the higher accuracy of hardware-level fault injection, the higher the computational time cost.

This paper proposes a cross-layer fault injection framework capable of significantly reducing the FI execution time without any impact on the DNN accuracy. In other words, it aims at joining together the accuracy of the hardware-level fault injection with the efficiency of the software-level one. More in details, the proposed framework targets systolic array DNN accelerators described at RTL, allowing to inject faults at channel granularity for convolutional layers. The basic idea is to simulate the execution of the Channel Under Test (ChUT) at RT-level. Faulty outputs collected from the RTL simulation are then used at software level to complete the execution of the DNN, and thus determine the impact of the injected faults at application level.

The rest of the paper is organized as follows. Section II overviews briefly the basic information about neural networks and fault injection techniques. Section III presents the frame-

work and section IV shows an implementation example and V concludes the paper.

II. Background

This section provides a background of Fault Injection techniques and the peculiarities of DNNs that have to be considered during fault injections.

A. Fault Models

Faults affecting electronic devices can be classified according to their temporal characteristics as permanent or transient. The former is stable with time and represents irreversible physical damage. The latter, instead, is active only for a short period of time and arises as a result of external disturbance or abnormal conditions or events (e.g., high-energy particle strikes). Starting from this broad fault types classification, the following fault models have been proposed over the years as abstractions of physical defects in electronics devices: stuck-at faults and bit-flips. The stuck-at fault is a very common fault model [8]. Indeed, it has been shown that many transistor and interconnection defects can be modeled with fair accuracy as permanent defects at the logic level. On the other hand, a random bit-flip model can represent the occurrence of transient faults, usually affecting registers or memory regions. Transient faults (i.e., soft errors) may be caused by different sources of interference phenomena such as electrical noise, electromagnetic interference, and impinging ionizing particles.

B. Fault Injection Techniques

To evaluate the system reliability, fault injection is a well-known and powerful technique to observe the impact of generated errors on the system behavior. It is based on the realization of controlled experiments to evaluate the system behavior in the presence of artificial faults. Many research works discuss fault injection in detail [8], [9]. In the following, the main types of FI are summarized.

The **hardware-based** fault injection techniques apply external perturbations to the circuit under test to evaluate the reliability. Particle radiations, laser beams or pin forcing are used to create realistic faults. Despite the accuracy of the obtained results, hardware-based FI are extremely costly in terms of equipments. Moreover, it can only be applied at the end of the design process, when the real device is available it cannot thus be used to perform design space exploration. It is complex, and sometimes not even possible, to fully control the injection process and select the injection point. Similar issues exist regarding the observability of the injection.

The **simulation-based** fault injection techniques do not operate on the physical device, but they employ a model of the device described using a simulation language, such as VHDL. They can inject faults in the VHDL model either at run-time or at compile-time. Compared to the physical fault injection techniques, the simulation-based techniques are cheaper in terms of set-up and can better control where the fault is injected. In addition, they present no risk to damage the hardware system under evaluation. However, they create a computational overhead depending on the complexity of the device under evaluation.

Finally, the **software-based** fault injection technique do not consider the hardware level at all. Faults are injected on program variables or on instructions. The main advantage is low implementation costs, high controllability and observability. On the other hand, it suffers from the lack of knowledge of the hardware layer, leading to non-realistic results.

C. Fault Injection for Deep Neural Networks

Even though DNNs can be seen as a software executed on a given hardware, there are some peculiarities that need to be considered to set up an effective fault injection campaign.

Firstly, the fault impact has to be measured differently compared to classical fault injection. Three main categories are defined as follows:

- **Masked** faults: the output of the faulty DNN is exactly the same as the fault free output.
- **Benign** faults: the faulty output differs from the fault free output, but it is still acceptable by the end user.
- **Malignant** faults: the faulty output is not acceptable by the end user.

Unfortunately, the definition of "acceptable" depends on the DNN, the chosen metric and the error threshold. For example, Figure 1 shows the output of an object detection algorithm used in autonomous driving. Figure 1a) is the fault free output, and Figure 1b) corresponds to the output when the fault leads to a "small" deviation (dark green rectangles) with respect to the expected behavior (light green rectangles). In this case, the fault is considered as benign, since the objects (i.e., the pedestrians) are still properly detected. On the contrary, Figure 1c) depicts the output when the fault significantly affects the application behavior. This fault is classified as malignant since the objects cannot be correctly identified, leading to a critical failure, e.g., the car may not stop itself hitting a pedestrian.

The second aspect is the need to consider the hardware used to run a given DNN. This means that the same network can behave differently depending on the hardware used to deploy it. Figure 2 depicts two abstraction levels, namely the HW-AI level and the DNN model (i.e., the neural network topology). The HW corresponds to a systolic-array-based architecture, where the single neural computation is executed by a single Processing Element (PE). Due to hardware limitations, each PE elaborates more than one neural computation. In this example, the HW fault is a permanent stuck-at logic '0' fault

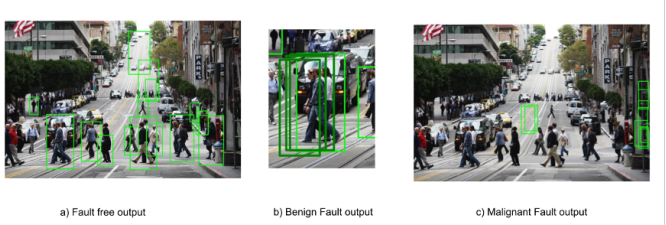


Fig. 1: Different fault Impacts

for the net b, due to an undesired short to ground. This means that faults affecting one PE may correspond to **multiple** faulty neurons. Therefore, a comprehensive resilience assessment of the systems can be obtained by only considering the hardware platform running the NN.

In the literature, many fault models have been proposed so far. For example:

- DNN-level fault models: these faults are hardware-agnostic. They consist in altering the value of a synapse weight, neuron bias, convolutional operations, activation function [3], [4]. They can be permanent or transient;
- Hardware level fault models: these faults are DNN model-agnostic. They can be permanent (i.e., stuck-at-fault) or transient (i.e., bit-flips affecting memory elements) [5], [7], [10];

The last aspect that is relevant to mention is the complexity of the input workload. For DNNs, reliability is usually evaluated when all the validation test set is used as input. For example, let us consider the MNIST dataset [11] composed of 10k images and a relatively small set of faults to be injected (only 1,000 faults). In this case, the proposed experiment needs to perform 10^6 fault injections; assuming an execution time of 1s per fault injection (supposing RTL simulations), the overall time results in about 115 days. To reduce the workload, some works propose to generate a reduced set of functional stimuli [7].

III. Proposed Framework

As described in the introduction, the proposed fault injection framework works at both software and hardware layer. Fig. 3 illustrates the main concept.

The starting point is the DNN under analysis implemented as a software application. The DNN is a pretrained model, and both the trained parameters and the dataset (or any other set of inputs) are assumed to be available. The intent of the work is to perform a fault injection campaign at the hardware level in a specific channel of a given layer of the DNN and execute the rest of the inference process at the software level to speed up the simulation process. Note that the framework can be used with basically any combination of hardware model and DNN implementation. The only requirements for the framework are two: (i) having some API to inject faults in the hardware model and (ii) modifying the DNN implementation to accept data as input for a layer different than the first. If these are satisfied, the framework can be used to simulate specific types of faults, as explained later in this section.

In our example, we target an image classifier based on a convolutional neural network. The user can select the layer and the channel of the DNN that has to be the targeted for the fault injection campaign; this channel is called Channel Under Test (ChUT). In the example, the first channel of the first layer is the ChUT. The ChUT is mapped and executed on the hardware description at RTL of the systolic array. During the simulation, a fault is injected. The outputs of the faulty convolution are then fed back into the software implementation and used in the next layers as input values. The rest of the execution can purely run in software in order to obtain the faulty outputs of the DNN and determine the fault impact. The next sections detail each component of the fault injector.

A. Systolic Array

We developed our own systolic array from the classical architecture as shown in Figure 4a. Each PE performs a Multiply and Accumulate (MAC) operation and forwards the inputs to the neighboring PEs. Figure 4b shows a single PE. The `NORTH` and `WEST` inputs are multiplied together and accumulated in a register that also function as the `RESULT` output.

The proposed systolic array corresponds to the **Output stationary** implementation, since the results are accumulated directly in each PE. This means that at every clock cycle (CC) both values: the weights and the activation values reach the PE for performing the MAC operation [12].

B. Mapping the ChUT

The ChUT mapping depends on the type of layer. In this work, we describe the mapping for convolutional layer only, but fully connected layers can be mapped as well. A DNN convolutional layer can be defined by the following triplet:

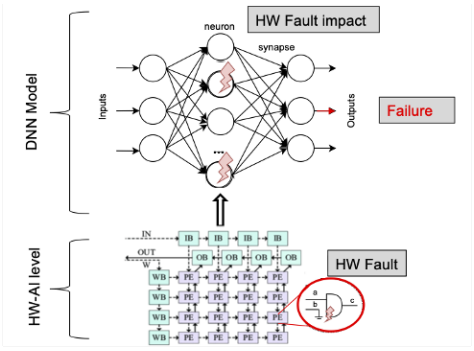


Fig. 2: Hardware fault impact

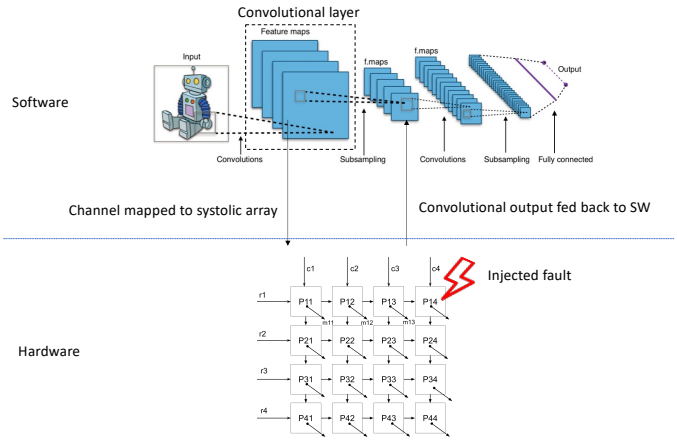
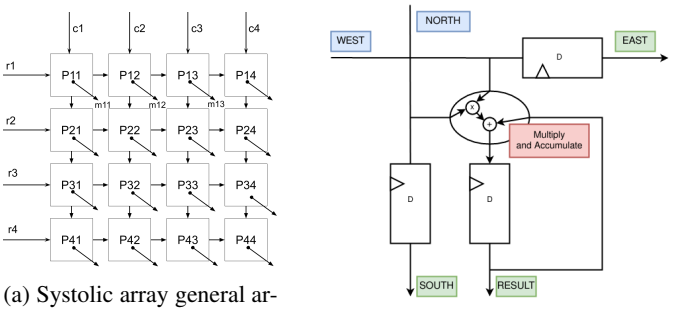


Fig. 3: Fault Injector framework



(a) Systolic array general architecture

(b) Logic representation of the PE

Fig. 4: Systolic array architecture

$$Conv = \#C@N \times N \tag{1}$$

Where #C is the number of channels composing the layer and N is the size of the convolution. In other words, it executes #C times the NxN convolution. The proposed systolic array is able to compute a single convolution at a time with a O(N) complexity (i.e., linear w.r.t. to one dimension of the convolution). Please note that the goal of our systolic array is not to accelerate the execution of the DNN, but to simulate the execution of a single channel (i.e., a single convolution) at the RTL level. In other words, we intend to have the smallest RTL description able to run the ChUT and perform a fault injection.

The targeted faults are permanent stuck-at faults affecting the two inputs of a given PE, i.e., NORTH and WEST. We injected faults only on the first channel. This kind of injection can be interpreted in two different ways:

- either the systolic array is used to process every layer, in which case, the injected fault is in reality a transient fault that only affects the computation of the first layer;
- or, the systolic array is used only for processing the first layer (due to specific requirements), while additional hardware takes care of the rest of the network.

The fault injection process is implemented by leveraging on the application programming interface (API) of the RTL simulator. More formally, a fault is defined by the following tuple:

$$Fault = \{PE, Input, Bit, Polarity\} \tag{2}$$

Where:

- PE: is one of the PE in the systolic array;
- $Input$: is one of the two inputs of the PE, i.e., NORTH or WEST
- Bit: is the bit that is modified by the fault. It depends on the PE data precision (i.e., bit-width),
- $Polarity$: is '0' or '1'.

By fixing the dimension of the array as K and a data precision of M bits, we can compute the fault universe of the systolic array as:

$$FaultUniverse = \underbrace{2}_{\text{Input}} \times \underbrace{2}_{\text{Polarity}} \times K \times M \tag{3}$$

C. Software implementation

The software implementation of the DNN is done by leveraging on the open-source framework N2D2 [13]. Such framework comes with several DNN model descriptions. We first used [13] to perform the training and then, the trained DNN has been exported as C code using two different data representations:

- *Int 16* weights are quantized as 16-bit integers;
- *Int 8* weights are quantized as 8-bit integers;

For the last two data types, the quantization is done after training through the following steps.

1) all weights are rescaled in the range $[-1.0, 1.0]$ and activations at each layer are rescaled in the range $[-1.0, 1.0]$ for signed outputs and $[0.0, 1.0]$ for unsigned outputs;
2) inputs, weights, biases and activations are quantized to the desired n_{bits} by converting $[-1.0, 1.0]$ and $[0.0, 1.0]$ to $[-2^{n_{bits}-1} - 1, 2^{n_{bits}-1} - 1]$ and $[0, 2^{n_{bits}-1} - 1]$

Details are given in the N2D2 documentation [13].

D. Orchestrator

The last component of our fault injector is the Orchestrator. As the name suggests, it has the role to manage the whole fault injection campaign and synchronize the hardware simulation and DNN software execution. The orchestrator was written in C, and it is sequentially responsible for:

1) creating the hardware simulation and DNN software thread;
2) generating the fault list;
3) managing the interaction between the hardware simulation and the DNN software thread.

Due to the system implementation, it is possible to perform the fault injection experiments in parallel, improving the fault injector performances. At the appropriate time, the orchestrator has the important role of synchronizing the different threads for a successful injection. As explained later, parallelization is extremely important for reducing the running time of the FI campaign.

IV. Validation

In this section, we will describe the setup used for our experiments. Firstly, the used DNN corresponds to the LeNet-5 [11] architecture composed of 7 layers: the first four are convolutional, while the last three are fully connected. The convolutional layers' sizes are 6@28x28, 6@14x14, 16@10x10 and the last is 16@5x5. We trained it on the MNIST handwritten digit dataset by using 32 x 32-pixel cropped pictures. The training set contained 48,000 images, with an additional 12,000 for the validation set and 10,000 for the testing set. The learning rate started at 0.05, with the decay of 5×10^{-4} every $375(*128)$ iterations, and momentum was set to 0.9. We define as "accuracy" of the CNN the capability to correctly classify

the input picture. The accuracy is computed by using the top-1 score [11]. The achieved accuracy over the 10,000 testing images is 99%.

We carried out three set of experiments, varying the precision of the network and the injection point. Each set of experiments was performed on a systolic array of variable size. A total of 9 experiments were performed.

For simplicity, we characterize each experiment using three parameters:

1) **Precision** - this value can be either 8 or 16 and describes whether the implemented network uses 8-bit or 16-bit integer values,
2) **Injection point** - can be either 'w' or 'i' and describes whether the injections were made in the **w**eights inputs of the PEs, or in the activation **i**nputs of the PEs.
3) **Array size** - can be either 28, 14 or 7 and describe the length of each side of the systolic array in terms of PEs.

Given these three parameters, for example, we could refer to experiment *8w7* representing to the experiments in an 8-bit data precision network implemented on a 7x7 systolic array, and injecting in the weights.

The complete list of the performed experiments is then *8w28*, *8w14*, *8w7*, *16w28*, *16w14*, *16w7*, *8i28*, *8i14*, and *8i7*.

The dimension of the fault universe changes on the base of the configuration of the array. In details, the experiment with the largest fault universe is 16w28, with a total of 50,176 possible injections, while the experiment with the smallest fault universe are 8w7 and 8i7, with only 1,568 possible injections (both values do not consider the 6 different channels).

As HDL simulator we used QuestaSim [14] which is able to run VHDL and Verilog simulations at RTL level.

For gathering statistically significant results, we injected a different number of random faults depending on the experiment.

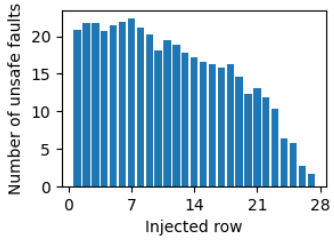
We now briefly illustrate some of the obtained results.

A. Malignant faults with respect to the faulty PE

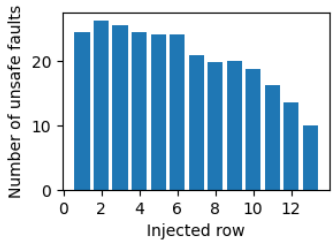
A simple expected result we obtained with this framework is the frequency of malignant faults with respect to the faulty PE. More specifically, we counted the number of malignant faults with respect to the row and column of the PE where the fault was injected. Since the weights are forwarded from NORTH to SOUTH, we expected that injecting a PE in a higher row would result in more "broken" values, thus leading to more severe faults. Indeed, that is what we found out.

Figure 5 shows how the number of malignant faults is correlated with the injected row. In particular, the figure shows two experiments: *8w28* and *8w14*. In the figures, it is possible to notice a descendant trend, showing that the nearer the fault is to the "top" of the array, the more critical it is.

A similar trend was observed when injecting the activation inputs. In that case, though, the same trend was visible in the columns of the array, rather than in the rows. This is because the activation inputs are forwarded from WEST to EAST.



(a) Experiment 8w28



(b) Experiment 8w14

Fig. 5: Number of malignant faults per injected PE

Both of these results show the effectiveness of the framework, especially when considering the amount of time needed for the completion. Indeed, using a traditional workflow and injecting (as in the case of the experiments involving a 28x28 systolic array) 15,000 random faults for 100 stimuli, with an average of 2s per simulations, would take around 35 days. With the help of this framework, also able to run parallel simulations, the actual running time was about 3 days.

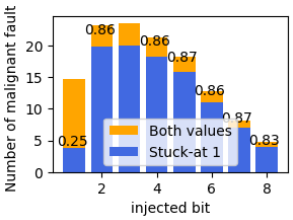
B. Malignant faults with respect to polarity

We counted the total number of malignant faults with respect to the polarity of the injection. We found out that the majority of the malignant injections are derived after injecting a stuck-at-1 injection, rather than stuck-at-0. Figure 6 shows part of the gathered data. It is possible to see that in almost every experiment, stuck-at-1 faults are the most critical. Indeed, consistently, they are more than 50% of the total malignant faults. This trend is visible in all the experiments. However, it is also interesting to highlight that when injecting in the DNN weights, the MSB (i.e., the sign bit) is most critical to stuck-at-0 faults. These results probably depend on the specific training of the network and come from the distribution of the weights values.

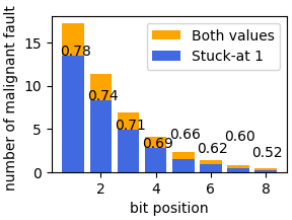
Notably, the reuse of the same hardware paired with higher precision also changed this trend. Indeed, experiments 16w28 and 16w7 show that about 50% of the malignant faults comes from stuck-at-1 injections.

C. Malignant faults per injected channel

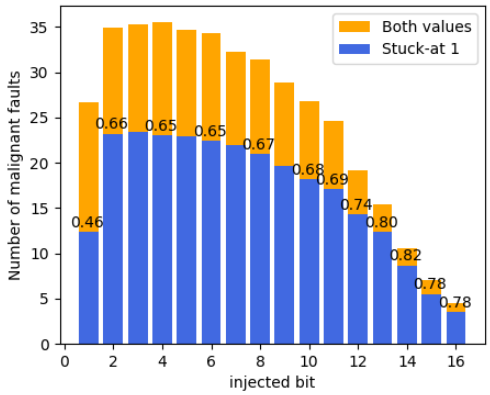
We randomly injected one channel among the six of the first layer. Figure 7 shows the number of malignant faults per each channel injected for experiments 8w28, 16w28, and 16w7. As the reader may notice, channel 4 is consistently the most critical one. The number of malignant faults is the highest in that channel with respect to the others. Furthermore, we have



(a) Experiment 8w28

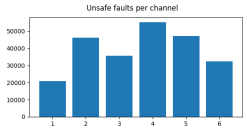


(b) Experiment 8i14

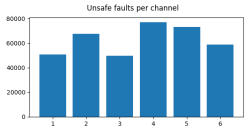


(c) Experiment 16w28

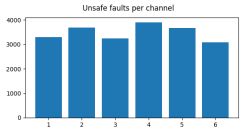
Fig. 6: Number of malignant faults, fault polarity and injected bit. The blue bars indicate the stuck-at-1 faults, while the yellow ones include both faults per bit, the text on the bars show the fraction of stuck-at-1 malignant faults. Bit 1 is the MSB. In "w" experiments, it also corresponds to the sign bit.



(a) Experiment 8w28



(b) Experiment 16w28



(c) Experiment 16w7

Fig. 7: Number of malignant faults per injected channel

seen that this result is consistent in every experiment when injecting faults in the inputs related to the weights. Although interesting, it should be investigated with a different set of weights (i.e., resorting to a new training of the neural network) in order to establish whether this result is purely related to the specific training process or not.

On the other hand, injecting in the activation inputs resulted in a different critical channel. Indeed, in that case, the fifth channel was the one with the highest number of malignant faults. Furthermore, the peak is not as "dramatic" as the previous case, underlying more input-resilience.

V. Conclusions

We showed a cross-layer injection framework with parallel capabilities, able to sensibly reduce the fault injection time. The proposed framework can reduce the total required time for running the experiments by a factor 10. The same framework can also be used at different levels than the ones illustrated in this paper, for example, including different abstraction levels in the simulation. In addition, the framework flexibility due to its parallelization possibilities have experimentally demonstrated to be really effective in reducing the total execution time.

Furthermore, the paper shows some insights on a stationary output systolic array. We noticed that the training of the network affects its reliability, and that might be a direction for future works.

Acknowledgment

This work has been founded by the RE-TRUSTING project, ANR-21-CE24-0015.

References

[1] E. Dupuis, S. Filip, O. Sentieys, D. Novo, I. O'Connor, and A. Bosio, "Approximations in deep learning," in *Approximate Computing Techniques*, pp. 467–512, Springer International Publishing, 2022.

[2] A. Ruospo, E. Sanchez, M. Traiola, I. O'Connor, and A. Bosio, "Investigating data representation for efficient and reliable convolutional neural networks," *Microprocessors and Microsystems*, vol. 86, p. 104318, 2021.

[3] Z. Chen *et al.*, "Tensorfi: A flexible fault injection framework for tensorflow applications," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, (Coimbra, Portugal), pp. 426–435, IEEE, Oct. 2020.

[4] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "Pytorchfi: A runtime perturbation tool for dnns," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 25–31, 2020.

[5] B. Salami, O. S. Unsal, and A. C. Kestelman, "On the resilience of RTL NN accelerators: Fault characterization and mitigation," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, (Lyon, France), pp. 322–329, IEEE, 2018.

[6] A. Ruospo, A. Balaara, A. Bosio, and E. Sanchez, "A pipelined multi-level fault injector for deep neural networks," in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, 2020.

[7] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu, "Toward functional safety of systolic array-based deep learning hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 485–498, 2021.

[8] G. D. Natale, D. Gizopoulos, S. D. Carlo, A. Bosio, and R. Canal, eds., *Cross-Layer Reliability of Computing Systems*. Institution of Engineering and Technology, Oct. 2020.

[9] M. Eslami, B. Ghavami, M. Raji, and A. Mahani, "A survey on fault injection methods of digital integrated circuits," *Integration*, vol. 71, pp. 154–163, 2020.

[10] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," in *Proceedings of the 55th Annual Design Automation Conference*.

[11] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.

[12] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," 2017. Publisher: arXiv Version Number: 2.

[13] CEA-LIST, "N2D2." [Online]. Available: https://github.com/CEA-LIST/N2D2.

[14] Intel, "Questasim." [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/questa-edition.html.