A Pipelined Multi-Level Fault Injector for Deep Neural Networks

Annachiara Ruospo*, Angelo Balaara*, Alberto Bosio†, Ernesto Sanchez*
*Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy
†INL, Ecole Centrale de Lyon, Lyon, France
{annachiara.ruospo@, ernesto.sanchez@, angelo.balaara@studenti.}polito.it
alberto.bosio@ec-lyon.fr

***Abstract*—Applications leveraging on new computing paradigms, such as brain-inspired computing, are currently being exploited in many fields thanks to their outstanding performance in solving complex tasks. Among them, Deep Neural Networks (DNNs) are gaining growing interest in different research areas spanning from playing complex games to safety-critical applications such as automotive. In the latter case, reliability assumes a dominant role and efficient reliability assessment approaches are thus required. Several works evaluate the DNN reliability by running fault injection campaigns. However, due to the excessive time required to run a single DNN execution (i.e., inference) at Hardware Description Level (HDL), the injections are typically performed at software level. This is clearly important to provide an overall estimation of the DNN behavior in faulty scenarios, however, it might be not accurate enough if the reliability of the *target* HW architecture must be determined. In that case, you need to run the fault injections directly at a hardware description level. The intent of the paper is to present a pipelined multi-layer fault injector for Deep Neural Networks that is able to drastically reduce the fault simulation time at HDL. Mimicking the behavior of the pipeline of a processor core, it allows to drastically reduce the complete fault injection time to be run at hardware level, thereby reducing the required time by about 60%.**

I. Introduction

Deep Neural Networks (DNNs) are implemented resorting to different approaches, an emerging trend is to exploit Application Specific Integrated Circuits (ASICs) for deploying applications based on DNNs [1], [2]. From the hardware perspective, they might be sufficiently equipped to run complete and real-time inference cycles. They typically include more than one core or a set of processing elements as well as hardware accelerators engines. Hence, since from their conceiving, these platforms have been optimized and oriented to run DNN-based applications. However, the ”perfect” micro-architecture is still a big debate in the research community.
In addition, during the last years, the research community has shown growing interest in understanding DNNs reliability. As the shrinking of semiconductor technologies continues, devices are getting more prone to physical errors: the probability that some computing elements fail increases as well. Therefore, even thought researchers claim that neural networks are potentially able to absorb some degrees of vulnerability due to their natural resilience properties [3], there is a pressing need for evaluating their robustness, especially if deployed on safety-critical systems. To this end, a very well known solution consists on performing fault injection campaigns to assess the network resilience to faults. These injection campaigns can be performed at very different abstraction levels, from the application level down to the injection of faults at gate level. However, the lower the abstraction level the higher the required fault injection time. As an example, a small DNN for image classification may count with about 2k neurons, and 1.5M weights to multiply, sum and compare at every inference; then, performing a fault injection during an inference at a very low level will require a high amount of time and computational resources. On the other side, the lower the abstraction level the higher the degree of freedom: while at the application level faults can only be placed on the network weights, activators or input images, at a lower level, i.e., hardware, injection of faults can be done anywhere in the system.

No matter which target platform is adopted, it might be needful to evaluate its robustness during the design phase for a particular DNN application *before* the fabrication process. In this way, *the designer could be able to co-shape the hardware platform with the DNN application, i.e., to make changes on the hardware or software depending on the wished reliability level*. To this end:

- The fault injector must take into account the actual *target* hardware platform. It is necessary to move from the pure software approach to a lower-level one.
- A simulation environment for running fault injection campaigns is required at Hardware Description Language (HDL), such as Register Transfer Level (RTL), to perform these computations and lay out conclusions.

It is well-known that RTL simulations of complete inference cycles take a significant amount of time; clearly, it depends on the hardware equipment, the deep neural network architecture, the simulation tool, and so on. Figures on this topic are provided in Section IV. Thus, there is a need for a faster and more efficient simulation environment that is able to host a considerable amount of fault injections in a timely fashion.
The intent of the paper is to propose a fault injector framework able to cut down the fault simulation time when executing DNN inferences at RTL. It makes use of a multi-level structure that mimics the flow of a pipeline. In particular,

978-1-7281-9457-8/20/$31.00 ©2020 IEEE

the neural network layers are viewed as pipeline stages, as in the case of a normal pipelined process. It is managed by a process that is in charge of synchronizing the single inference computation. The proposed framework supports the designer to evaluate the hardware suitability and robustness when running a specific DNN-based application.

The rest of the paper is organized as follows. The state-of-the-art research work is highlighted in Section II. Section III presents the proposed framework and the Section IV describes the case study and the experimental set up. Section V provides the experimental results. Finally, Section VI concludes the article by outlining some of the possible future research directions.

II. Related Work

As mentioned before, the robustness evaluation of DNNs can be pursued at different levels: from a software abstraction layer to a hardware one, up to silicon measurements on ASICs, GPUs and FPGAs resorting to radiation campaigns. In the literature, a great effort has been spent to propose methodologies to facilitate the network reliability assessment. Many of these rely on specific frameworks allowing fault injection campaigns. The state-of-the-art faults injection methodologies can be classified as follow:

1) **Simulation-based:** The injection process is carried out without relying on the physical device finally running the DNN. Moreover, depending on the abstraction level they can be further ranked.
   a) **Software Level:** The injections are performed on a high level model of the DNN, not considering any detail of the actual hardware architecture.
   b) **Hardware Level:** The injections are performed on a more accurate model of the DNN that simulates the target hardware architecture. This can be described at RT-Level or Gate level, for example.
2) **Emulation-based:** The measurements and the analyses are performed directly on a physical device that emulates the final implementation resorting to hardware accelerators based on for example FPGAs, and GPUs.
3) **Platform-based:** The reliability assessment is performed in the actual platform during a radiation campaign.

A large part of the state-of-the-art approaches comes from Class 2, where the reliability assessment is measured directly on a physical platform that emulates the final implementation. In [4], fault injections are performed on a Xilinx SRAM-based FPGA, emulating a Convolutional Neural Network (CNN), a class of DNN. In [5], the authors present a framework for quantifying the resilience of Deep Neural Networks: Ares. It enables the DNNs execution directly on the GPUs, by injecting static and transient faults in memory at the application level. A similar methodology is described in [6], where the authors assess the reliability of a 54-layers DNN (NVIDIA DriveWorks) on a Volta GPU. The framework is TensorRT and has been developed by NVIDIA [7]. It consists of an API and a runtime system that optimizes the inferences on GPU architectures. More in details, the authors adopt an emulation-based approach for injecting permanent faults on weights and input images and a platform-based (Class 3) for transient errors resorting to radiation campaigns. A second example of platform-based method is provided in [8], where the reliability of a CNN is evaluated on three different GPU architectures (Kepler, Maxwell, and Pascal). The soft errors injection has been done by exposing the GPUs running the CNN under controlled neutron beams. In this work, this kind of fault injectors are not considered.

Regarding the simulation-based techniques (Class 1), software approaches are the most frequently used (Class 1a). In [9], the fault injector is implemented in Python with NumPy, and the DNN model is built with PyTorch. They studied the reliability of LeNet5, a well-known DNN for handwritten digits classification. They consider single and multiple event transients on combinational circuits as well as single and multiple event upsets in memories. The robustness of the same network, LeNet5, is investigated in a second work, where the fault injection is build on darknet, an open-source DNN framework [10]. Implemented in C and CUDA language, it is suited to perform end-to-end deployment of neural network architectures in a very simple way. The authors evaluate the network reliability of LeNet5 and YOLO (a CNN capable of detecting objects in real time) by injecting permanent faults on the network weights. All those software methodologies are relevant for assessing the network resilience to errors and for characterizing the criticality of the network layers, but, they might be not complete to represent the real DNN behavior when deployed on a physical device.

Finally, an example of simulation-based hardware approach (Class 1b) is provided in [11], where the reliability of an hardware accelerator has been investigated by exploiting a framework embedded into the RTL design. They follow a High Level Synthesis (HLS) approach to characterize the effects of both permanent and transient faults. Faults are injected during the inference cycles only on a sub-set of registers: those that are in charge to store weights, input values and intermediate ones. Notwithstanding the benefits coming from all these different methodologies, this work focuses on Class 1b and proposes a simulation-based fault injection methodology at a low hardware level of abstraction. Contrary to [12], the proposed methodology does not apply a statistical fault injection and does not focus on multiple bit upsets but on permanent faults.

III. Proposed Framework

Deep Neural Networks are made of many ”deep” and hidden layers, each of these accomplishing special tasks. For instance, the convolutional layers are able to extract features from input images representing gradually more complex information; fully-connected layers, on the other side, are in charge of classifying the information coming from the previous layers, by awarding a confidence score representing the answer of the network. Regardless the actual hardware performing the inference, it is possible to execute the layers processes in a

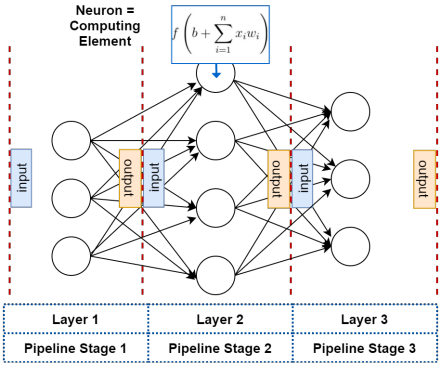

Fig. 1. The Pipeline Concept applied on a 3-layers Neural Network

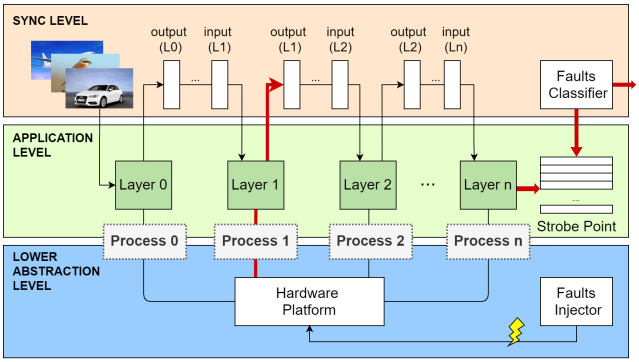

Fig. 2. Proposed Pipelined Multi-Level Fault Injector Framework

sequence of steps. Indeed, in an image recognition process, the input of the first layer is an image, then, when a layer has completed its elaboration, it produces an output that becomes the input for the subsequent layer, and so on. In an inference cycle, trying to understand the network behavior in presence of a fault, requires to inject the fault at low level, for example RTL, and then performing the full inference execution and collecting the DNN results. This process is very time consuming even though using commercial fault simulation tools [13]. Thus, the structure of the proposed framework mimics the working pipelined in a RISC processor, where the incoming instructions are elaborated by a set of stages working in parallel and synchronized by the clock. In this way, a number of instructions equal to the number of stages are processed in parallel, reducing the overall execution time even though the single instruction execution time is actually increased. In a similar way, we adapt the DNN implementation into a pipelined fashion in order to improve the execution time when injecting faults. The main idea is highlighted in Figure 1, where each layer of a DNN turns into one stage of the pipeline. In the case of a DNN, it is important to analyze the behavior of the network in presence of a fault given a set of images, so recalling the connection with the CPU pipeline, it is worth mentioning that the role of the instructions is thus transferred to the input images. The Instruction-level parallelism of a CPU pipeline scenario, effectively becomes an Image-level parallelism.

The architecture of the proposed multi-level fault injector framework is detailed in Figure 2. As evidenced, it is composed of three vertically stacked levels, starting from the bottom: the Lower abstraction level, the Application Level, and the Synchronization Level. They are described in the following.

*1) **Lower abstraction level**:* It is the bottom layer and holds the most accurate hardware model of the platform running the DNN under assessment, as well as the Fault Injector Unit (FIU). The target hardware for the actual implementation of the DNN, as stated in Section I, may be an embedded device, a GPU or a different system running the neural network, for which the reliability analysis is required. The actual device is not physically available but the design is provided as HDL. In this paper, the RTL is the one adopted but nothing prevents the gate level from being exploited in future works. It is necessary to stress on the point that the pipelined fault injector does not require any changes to the target hardware model. On the other hand, the FIU is the module in charge of injecting faults in the RTL design, in this paper, we target permanent faults (s-a-0 or s-a-1), but it is also possible to extend the proposed approach to other fault models. The FIU unit holds the list of faults to be injected. For every fault in the fault list, the inferences of all the images belonging to the test set are executed. Thus, the fault injection unit must be aligned with the synchronization level, receiving an acknowledge when all the inferences for a given fault have been completed. In this way, it may proceed and inject the next fault belonging to the fault list.

*2) **Application Level**:* The neural network application runs on the top of the lower level and may be implemented in a high-level language such as C or Python. This level only requires to be able to perform the inference process in a pipelined way by identifying the network layers separately. Indeed, to communicate with the upper layer (synchronization layer) and lower layer, and creating the pipeline process, for each network layer it is necessary to extract the following features: the input and the output neurons. Additionally, depending on the layer topology (convolutional, fully connected max pooling), other features might be necessary. The main idea is then to split the inference in several stand-alone blocks corresponding to the layers of the network. It means that each layer must be simulated on the lower level as a separate application process. This configuration gives the upper level the possibility to gain control over the inference, by synchronizing the running of the separate stand-alone layers simulations. It is important to highlight that, as the last layer of the neural network returns the network prediction, i.e, the results of the inference, we set these output points as observation points of the fault injector, also known as strobe points. Indeed, to evaluate the impact of the fault on the network, the framework must gather for each inference the faulty results in order to perform statistics. This task is assigned to the Fault Classifier Unit (FCU), lying in the sync level.

*3) **Synchronization Level**:* This level is in charge of coordinating the activity of the neural network layers and, so, the advance of the complete inference cycle. The level

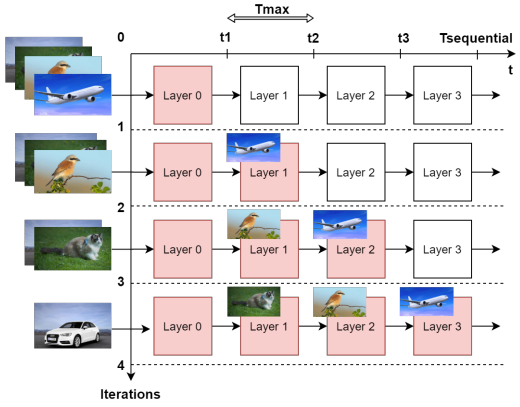

Fig. 3. Framework Set-Up Time for a 4-Layers Neural Network

includes: the database, i.e., the test set of images used for the inferences, and the Fault Classifier Unit (FCU), whose intent is to gather the results of the predictions. Furthermore, it internally computes statistics aiming at classifying the injected faults. However, the hardest task of the level is to synchronize the layers simulations. It means that the neurons input and outputs for any layer should be synchronized in this level. This mechanism is underlined by a red arrow linking the hardware low level with the Output of the Layer 1, in Figure 2.

In a similar way than a processor pipeline, the proposed fault injector requires an initial Set-Up Time (T_{setup}) to work at full capacity, i.e., having all the stages computing a part of a different inference. The T_{setup} time is computed by multiplying the amount of layers or stages (x) by the simulation time of the "slowest" one (T_{max}), i.e., the most computational-intensive (Equation 1). For instance, considering Figure 3, x equals 4 and T_{max} corresponds to the HDL simulation time of Layer1. Once the T_{setup} is done, the pipeline works at full capacity and the framework is able to provide an inference result every T_{max}. This time is much slower than the sequential time $T_{sequential}$. Therefore, as described in Equation 2, the time required to run a pipelined fault injection (T_{pipe}) with y input images is computed by adding the initial T_{setup} with the inferences of the (y−1) images. Without resorting to the proposed pipelined framework, the simulation time (T_{no_pipe}) is computed as described in Equation 3.

$$T_{setup} = x * T_{max} \tag{1}$$

$$T_{pipe} = T_{setup} + (y-1) * T_{max} \tag{2}$$

$$T_{no\_pipe} = y * T_{sequential} \tag{3}$$

IV. Case Study

The proposed fault injector has been evaluated on an open-source Parallel Ultra-Low-Power (PULP) hardware platform [14]. It is a complete system based on multi-core IoT Processors, including a cluster of 8 RISCV cores and a classic micro-controller for communications and security purposes. Moreover, it comprises a Hardware Convolution Engine (HWCE) for running DNN-based applications. The RISCV cores implement a 4 stage in-order single-issue pipeline and support the RV32IMC instruction set plus extensions for machine learning as well as Digital Signal Processing (DSP). The Deep Neural Network running on [14] is a CNN used for image classification. The CNN application code is written in C and exploits the PULP-NN open-source library [15], a multicore computing library allowing inferences on RISC-V Based PULP Cluster. PULP-NN includes a full set of kernels and utilities to support the inference of quantized neural networks (8,4,2 and 1-bit) on a RISC-V based processor. The CNN has been trained on CIFAR-10 database, a dataset consisting of 60,000 32x32 colour images grouped in 10 classes [16]. All the weights and activations are quantized to 8-bit signed integer and are available on the GitHub network repository[1]. Before enabling the CNN to be run and simulated on the target HDL design, we developed a Python model of the network by adopting the Pytorch library [17]. This step is important to create the basic building block of the framework synchronization level. The Python model uses the same pre-trained weights and biases. On the whole, the Quantized CNN model, as claimed in [18], achieves a 70% of accuracy. For the sake of completeness, the final Quantized CNN adopts the following topology. It consists of 7 stacked layers, varying from Convolutional (CV), Max Pooling (MP) and Fully-Connected (FC). Table I reports the main properties for every layer. Particularly, Column 2 determines the layer Type. Column 3 denotes the category, if any, of the Activation Function. For instance, only the three CV layers adopt the Rectified Linear Unit (ReLU)In the 4th and 5th Column, the number of, respectively, Input and Output neurons are given. It is worth noting that for CV layers, these correspond to the Input Feature Maps (ifmap) and the Output Feature Maps (ofmap) values. Finally, from Column 6 to 9, details about the kernels (filters) are provided, i.e., the sets of weights. In order, the Stride specifies the amount by which the kernel shifts over the input. The Size determines the dimension of the filters: it is 3D for the CV layers and 2D for the MP and FC. Next, Column 8 contains the Padding information, i.e., a technique used to maintain the same dimensionality with the input, by surrounding the input image with zeros. The last Column specifies how many kernels are used for the layer: the Quantity. Concerning the last FC layer, it is not quite proper to refer to kernels. Indeed, while the output of both CV and MP layers are 3D volumes, a FC layer expects a 1D vector. Hence, the output of the final MP layer is typically flattened to a 1D vector to fit the FC layer.

V. Experimental Evaluation

To validate the methodology, experiments have been carried out on a server equipped with a dual Intel Xeon CPU E5-2680 v3 and 256 GB of RAM. Given the network topology running on PULP, the duration of a single inference simulation at RT-Level has been computed by resorting to a commercial HDL simulator tool from Mentor Graphics. It turns out to be around

[1]https://github.com/pulp-platform/pulp-nn

TABLE I
QUANTIZED CONVOLUTIONAL NEURAL NETWORK ARCHITECTURAL DETAILS

	Type	Activation Function	Neurons		Kernel			
			Input	Output	Stride	Size	Padding	Quantity
Layer 0	Convolutional	ReLU	32x32x4	32x32x32	1	5x5x4	2	32
Layer 1	Max Pooling	None	32x32x32	16x16x32	2	3x3	0	1
Layer 2	Convolutional	ReLU	16x16x32	16x16x16	1	5x5x4	2	16
Layer 3	Max Pooling	None	16x16x16	8x8x16	2	3x3	0	1
Layer 4	Convolutional	ReLU	8x8x16	8x8x32	1	5x5x4	2	32
Layer 5	Max Pooling	None	8x8x32	4x4x32	2	3x3	0	1
Layer 6	Fully Connected	None	1x512	10	0	1x512	0	10

TABLE II
PIPELINED FAULT INJECTOR TIMING DETAILS

Neural Network			Fault Injector Framework		
Layer	Name	Sim Time (min:sec)	Layer	Name	Sim Time (min:sec)
Layer 0	CV1	10:11	**Layer 0**	CV1	10:11
Layer 1	MP1	4:00	**Layer 1**	MP1	4:00
Layer 2	CV2	9:39	**Layer 2**	CV2	9:39
Layer 3	MP2	1:19	**Layer 3**	MIX	8:04
Layer 4	CV3	4:23			
Layer 5	MP3	1:07			
Layer 6	FC	1:38			

25 minutes ($T_{sequential}$). Then, the simulation times for each single layer have been extracted; they are reported in the 3rd column of Table II. Clearly, these figures are proportional to the amount of computations that they have to perform (look at the amount of neurons from Table I). Thus, starting from these analyses, the pipelined multi-level fault injector has been created. As stated in Section III, the most computational-intensive layer determines the speed of the framework. Concerning our case study, it turns out to be Layer0 (CV1) and T_{max}=$\sim$10 minutes. Furthermore, the amount of stages of the pipeline corresponds to the number of layers of the neural network (referred to as x). However, to optimize the throughput of the pipeline and to reduce the T_{setup}, the last four layers (MP2, CV3, MP3, FC) have been joined in a single pipeline stage, named MIX and T_{MIX}=$\{T_{MP2}$+T_{CV3}+T_{MP3}+$T_{FC}\}$ (Column 5th of Table II). It means that these never compute in parallel fashion, but only sequentially. In short, the pipelined fault injector framework is made up of 4 layers and its T_{setup}=$\sim$40 min. It is important to remark the point about the framework throughput: after an initial T_{setup}=$\sim$40 min, the pipeline works at full capacity, so, every $\sim$10 minutes the framework provides the results of an inference. If the other layers complete their computation before T_{max}, their results are buffered and synchronized by the Synchronization Level. Concerning the timing performance, every time a new set of inferences are going to be executed, a T_{setup} must be considered. Given the topology of the framework, T_{setup}, T_{max}, $T_{sequential}$, and x, Table III provides an example of a single fault injection campaign, with the aim of demonstrating the framework performances as well as the gain in terms of simulation time. Column 1st denotes the amount of layers of the pipelined fault injector (x); while the second column, the number of input images (y) that are simulated for the purpose. T_{pipe} and T_{no_pipe} have been computed by applying, respectively, Equation 2 and Equation 3. As evidenced, a non-negligible time reduction is obtained; the proposed framework improves the sequential fault injection times by 60%. Clearly, such gain implies a cost in terms of memory occupation. It has been computed that the sequential fault injector employs on average from 2.1 up to 4.7 GB of RAM memory for running y inferences, whilst the pipelined fault injector uses from 8.4 up to 18 GB for the same set of y images, although for a shorter time.

TABLE III
A COMPARISON BETWEEN THE PERFORMANCES OF THE SEQUENTIAL FAULT INJECTOR WRT THE PIPELINED

Layers (x)	Images (y)	T_{no_pipe}	T_{pipe}	Time Reduction
4	100	$\sim$ 42 h	$\sim$ 17 h	60%

Furthermore, to prove the effectiveness of the pipelined multi-level fault injector, a set of 10,000 injections campaigns have been performed. The reader should note that: first, a massive injection experiment is out of the scope of the paper, which main idea is to present the framework structure. Second, the framework should serve as an instrument to conduct precise injections on definite locations. Anyway, to attest its feasibility, a random set of 100 permanent faults (s-a-0 or s-a-1) have been injected on the PULP platform running the Quantized CNN: half of these on a CPU, the second half are randomly distributed over the multi-core system (SYS). For instance, on the Cluster-Memory Interconnection, on the DMA, on the Peripheral Interconnection, Bridges and so on. For each fault, the inference of 100 input images from CIFAR-10 dataset have been performed. Then, faults have been classified depending on their effect and according to the ranking proposed in [19]. A fault is *Detected* whether one of the following situations occur:

- **SDC-1**: A Silent Data Corruption (SDC) failure is a deviation of the network output from the golden network result, leading to a misprediction. Hence, the fault causes the image to be wrongly classified.
- **SDC-10%**: The faulty network correctly predicts the result but assigns a confidence score which varies by more than +/-10% of its fault-free execution.
- **SDC-20%**: The faulty network correctly predicts the result but assigns a confidence score which varies by more than +/-20% of its fault-free execution.
- **Hang**: The fault causes the system to hang and the HDL simulation never finishes.
- **Crash**: It is the opposite situation of the previous one. The HDL simulation immediately stops as a consequence

TABLE IV
FAULT INJECTION RESULTS

	Detected [%]					Masked [%]
	SDC-1	SDC-10	SDC-20	Hang	Crash	
CPU	2.06	8.12	9.84	21.82	38.14	20.02
SYS	3.96	2.02	5.96	14.02	26.16	47.88

TABLE V
SEQUENTIAL FRAMEWORK VS PIPELINED MULTI-LEVEL FRAMEWORK

Sequential Framework	Total Injections	Tsequential [min]			y	Duration [h]
	10,000	25			100	$\sim$ 4,166
Pipelined Multi-Level Framework	**Total Injections**	**Tmax [min]**	**Tsetup [min]**	**x**	**y**	**Duration [h]**
	10,000	10	40	4	100	$\sim$ 881

of the fault.

To deal with the *Hang* category, i.e., faults producing an infinite loop, the framework has been equipped with a Timeout. If the framework, after T_{setup}, does not yield any inference results, the fault is classified as detected by system hanging. On the other hand, a fault is *Undetected* and classified as **Masked** whether the network prediction does not belong to any of the previous described classification and the output results are equal to the fault-free execution ones. Table IV reports the fault injection results. As expected, a small percentage of permanent faults leads to a network misprediction (SDC-1), no matter the fault is placed in the system or in the CPU. The majority of faults are masked during the inference or detected by system hanging or crash. Anyway, the reliability analysis of the network is not the key point of the paper. It results more significant to focus on the time needed to run the above-mentioned 10,000 fault injections. As shown in Table V, the pipelined multi-level fault injector allows to drastically reduce the fault simulation time by 78%. This gain exceeds the one you would obtain if applying Equation 2 (it has been computed and it corresponds to $\sim$1,716 h). This improvement is due to all the faults belonging to the Hang and Crash categories, that, once discovered at the beginning of the pipeline process, avoid the fault simulation of the entire image set for that fault. It is important to underline that this mechanism has not been implemented in the sequential framework.

VI. CONCLUSION

The paper presents a pipelined multi-level framework for assessing the robustness of DNNs running on a target device. It arose from the necessity of complementing pure software approaches with lower-level ones, those considering the hardware description of the target platform as well. Indeed, injection methodologies at software level can only target a bounded set of network parameters such as weights, biases and input images. A hardware level approach enlarges the set of possible choices for assessing the DNN-based system reliability. Hence, the paper first provides a detailed description of the fault injection framework. Then, it demonstrate that, by resorting to a pipelined mechanism, the inference time reduces by about 78%. That number clearly depends on several aspects such as the DNN structure, the amount of layers and parameters, the hardware platform. As shown, starting from these factors it is possible to optimize the framework structure, for instance, by joining some layers with the aim of cutting the T_{setup}. Future work aims first at extending the pipelined multi-level framework to cope with transient faults as well as Multiple Bit Upsets (MBUs). Second, reducing even more the HDL simulation time, by for example, exploiting a multi-process environment.

ACKNOWLEDGMENTS

This work has been partially founded by the projects IDEX Lyon OdeLe and ReACT.

REFERENCES

[1] D. Palossi *et al.*, "A 64-mw dnn-based visual navigation engine for autonomous nano-drones," *IEEE Internet of Things Journal*, vol. 6, no. 5, p. 8357–8371, Oct 2019. [Online]. Available: http://dx.doi.org/10.1109/JIOT.2019.2917066

[2] N. Smolyanskiy, A. Kamenev, J. Smith, and S. Birchfield, "Toward low-flying autonomous mav trail navigation using deep neural networks for environmental awareness," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 4241–4247.

[3] C. Torres-Huitzil and B. Girau, "Fault and error tolerance in neural networks: A review," *IEEE Access*, vol. 5, pp. 17 322–17 341, 2017.

[4] B. Du, S. Azimi, C. de Sio, L. Bozzoli, and L. Sterpone, "On the reliability of convolutional neural network implementation on sram-based fpga," in *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2019, pp. 1–6.

[5] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[6] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang, "Resiliency of automotive object detection networks on gpu architectures," in *2019 IEEE International Test Conference (ITC)*, 2019, pp. 1–9.

[7] NVIDIA, "Tensorrt," https://developer.nvidia.com/tensorrt, 2020.

[8] F. dos Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, and P. Rech, "Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures," pp. 169–176, 06 2017.

[9] M. A. Neggaz, I. Alouani, P. R. Lorenzo, and S. Niar, "A reliability study on cnns for critical embedded systems," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 476–479.

[10] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A reliability analysis of a deep neural network," in *2019 IEEE Latin American Test Symposium (LATS)*, 2019, pp. 1–6.

[11] B. Salami, O. Unsal, and A. Cristal, "On the resilience of rtl nn accelerators: Fault characterization and mitigation," 2018.

[12] M. Maniatakos, M. Michael, C. Tirumurti, and Y. Makris, "Revisiting vulnerability analysis in modern microprocessors," *Computers, IEEE Transactions on*, vol. 64, pp. 2664–2674, 09 2015.

[13] A. Floridia, E. Sanchez, and M. Sonza Reorda, "Fault grading techniques of software test libraries for safety-critical applications," *IEEE Access*, vol. 7, pp. 63 578–63 587, 2019.

[14] E. Flamand *et al.*, "Gap-8: A risc-v soc for ai at the edge of the iot," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–4.

[15] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors," 2019.

[16] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 05 2012.

[17] Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[18] M. Rusci, A. Capotondi, and L. Benini, "Memory-driven mixed low precision quantization for enabling deep network inference on micro-controllers," 2019.

[19] G. Li *et al.*, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," ser. SC '17. ACM, 2017, pp. 8:1–8:12.